Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Reduzindo a Duplicação de Código em Aplicações Corporativas: um Arcabouço baseado em Padrões de Renderização

Delano Hélio Oliveira

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Hyggo Oliveira de Almeida (Orientador)
Angelo Perkusich (Orientador)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

O48r Oliveira, Delano Hélio.

Reduzindo a duplicação de código em aplicações corporativas: um arcabouço baseado em padrões de renderização / Delano Hélio Oliveira. – Campina Grande, 2015.

59 f. : color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2015.

"Orientação: Prof. Hyggo Oliveira de Almeida, Angelo Perkusich".

Referências.

1. Aplicações Corporativas. 2. Sistemas Adaptáveis. 3. Arcabouços Scaffold. I. Almeida, Hyggo Oliveira de. II. Perkusich, Angelo. III. Título.

CDU 004.41(043)

REDUZINDO A DUPLICAÇÃO DE CÓDIGO EM APLICAÇÕES CORPORATIVAS: UM ARCABOUÇO BASEADO EM PADRÕES DE RENDERIZAÇÃO"

DELANO HÉLIO OLIVEIRA

DISSERTAÇÃO APROVADA EM 19/06/2015

HYGGO OLIVEIRA DE ALMEIDA, D.Sc, UFCG
Orientador(a)

ANGELO PERKUSICH, D.Sc, UFCG
Orientador(a)

AYLA DÉBORA DANTAS DE SOUZA REBOUÇAS, D.Sc, UFPB
Examinador(a)

KYLLER COSTA GORGÔNIO, Dr., UFCG
Examinador(a)

CAMPINA GRANDE - PB

Resumo

O desenvolvimento de aplicações corporativas modernas para web baseia-se na utilização de padrões e ferramentas para viabilizar a rápida prototipagem, garantindo a separação entre modelo de negócio e interface gráfica de usuário (GUI, do inglês *Graphical User Interface*). As plataformas de *Scaffold*, por exemplo, permitem um aumento da produtividade dos desenvolvedores ao gerarem código a partir dos elementos do modelo conceitual. Porém, o código fonte de GUI gerado apresenta muita replicação, devido ao acoplamento ainda existente entre os componentes das telas de interface gráfica e as propriedades inerentes ao modelo conceitual da aplicação, dificultando a manutenção do software. Os padrões de renderização propostos por Welick et al. se apresentam como uma solução conceitual para este problema, através do mapeamento de metadados do modelo conceitual em componentes gráficos, organizando o código de GUI e reduzindo a replicação de código. Neste trabalho, tem-se como objetivo a criação de um arcabouço para o desenvolvimento de aplicações corporativas com arquitetura web moderna, com foco em GUI, baseado em padrões de renderização. O arcabouço permite que o desenvolvedor construa componentes de GUI sem acoplá-los aos elementos do modelo conceitual. A associação da GUI com o modelo conceitual é feita através de regras de renderização, que podem ser alteradas facilmente. O arcabouço proposto foi validado através de um estudo de caso, no qual foi demonstrada uma redução significativa na duplicação do código quando comparada às plataformas de *Scaffold*.

Abstract

The modern enterprise web application development is based on the use of patterns and tools to enable rapid prototyping, ensuring separation between the business model and graphical user interface (GUI). The Scaffold frameworks, for example, allows an increase in productivity of developers to generate code from the elements of domain model. However, the GUI source code generated presents a lot of replications due to coupling still extant between GUI components and properties inherent to the domain model of the application, making it difficult to maintain the software. The rendering patterns proposed by Welick et al. are presented as a conceptual solution to this problem by mapping domain model metadata to graphical components, organizing the GUI code and reducing code duplication. In this work, we have aimed to create a framework for enterprise applications development with web modern architecture, focusing on GUI, based on rendering patterns. This framework allows the developer to build GUI components without engage to elements of domain model. The GUI link with domain model is made by rendering rules, which can be changed easily. The proposed framework was validated by a case study in which it was demonstrated a significant reduction in code duplication when compared to Scaffold frameworks.

Agradecimentos

Em primeiro lugar, quero agradecer a Deus não apenas por esta grande conquista, mas principalmente pelo Seu favor imerecido derramado sobre mim para me dar vida em abudância. Sem essa graça nada disso faria sentido.

À minha amada esposa, Carlúcia Oliveira, pela grande paciência e renúncia durante todo esse período em prol desse objetivo em comum. Sei que você teve que suportar muito estresse e ausência de minha parte, mas finalmente conseguimos.

Aos meus pais e meus irmãos por investirem e acreditarem no meu esforço e dedicação.

À toda minha família pelo carinho que tiveram comigo durante esse período, principalmente nos momentos de dificuldades.

Aos meus orientadores, Hyggo Almeida e Angelo Perkusich, pelo tempo e dedicação que tiveram em me orientar.

Ao doutorando Rodrigo Vilar, por ter enfrentado junto comigo os desafios deste trabalho e me incentivado bastante durante esse mestrado.

Aos meus amigos de infância, Danilo (que é meu irmão), Thales e Júnior por me mostrarem que a vida é muito mais que letras e números. Ao grupo de amigos DINS++ (Izabela, Natã, Savyo e Demontiê) por terem me influenciado a fazer esse mestrado e por compartilhar das mesmas dores e alegrias de um mestrado apesar da distância.

À minha igreja pelas orações, carinho e compreensão em minhas ausências.

À CAPES, pelo incentivo financeiro.

Conteúdo

Lista de Símbolos

AC - *Aplicações Corporativas*

AOM - *Adaptive Object Model*

API - *Application Programming Interface*

CoC - *Convention over Configuration*

DOM - *Document Object Model*

GUI - *Graphical User Interface*

HTML - *HyperText Markup Language*

HTTP - *Hypertext Transfer Protocol*

JMS - *Java Message Service*

JPA - *Java Persistence API*

JSON - *JavaScript Object Notation*

JSP - *JavaServer Pages*

MVC - *Model-View-Controller*

REST - *Representational State Transfer*

XML - *Extensible Markup Language*

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

1.1 Motivação

As Aplicações Corporativas (AC) são um tipo específico de software que, em geral, manipulam bases de dados complexas [8]. A fim de disponibilizar dados para os usuários finais, as AC disponibilizam interfaces gráficas ou simplesmente GUI, do inglês *Graphical User Interface*. Nestas aplicações, há acoplamento entre as telas da GUI e as entidades do modelo conceitual, pois os componentes gráficos normalmente representam as entidades, as propriedades e os relacionamentos do modelo conceitual do domínio. Esse acoplamento dificulta a reutilização de código [11].

Por exemplo, em uma AC fictícia há dois formulários HTML, o primeiro para editar os dados dos Clientes e o segundo para editar os Departamentos. O formulário de Cliente possuiria os campos "nome", "data de nascimento" e "CPF", enquanto Departamento teria "nome" e "endereço". Nesse caso, o código para exibição do campo "nome", que é idêntico para as duas entidades, não pode ser fatorado, pois os campos fazem parte de formulários que dependem diretamente da entidade em questão, Cliente ou Departamento. Dessa forma, os campos "nome" devem ser codificados duas vezes com estruturas idênticas, violando o princípio DRY (*Don't Repeat Yourself*): "cada pedaço de conhecimento deve possuir uma representação única, inequívoca e oficial dentro de um sistema" [13]. A Listagem 1.1 demonstra o código desse exemplo de duplicação nas linhas 2 e 9.

Além disso, existem outros tipos de duplicação de GUI, que também podem ser vistos na Listagem 1.1. As linhas 9 e 10 são semelhantes, pois os campos "nome" e "endereço"

de Departamento possuem o mesmo tipo, *String*. De fato, os campos com mesmo tipo geralmente possuem o mesmo código de GUI. Se houvesse outro campo data nesse exemplo, ele possuiria código semelhante ao das linhas 3 e 4. As duplicações não ocorrem apenas no nível dos campos. Nota-se que, para cada entidade do exemplo, existe uma tag *form* cuja estrutura se repete na GUI.

Código Fonte 1.1: Scripts HTML para formulários de Cliente e Departamento

```
1  <form action="/clientes">
2    <input type="text" name="nome" value="${cliente.nome}" />
3    <input type="date" name="nasc" value='<fmt:formatDate
4        value="${cliente.nasc}" pattern="dd/MM/yyyy" />' />
5    <input type="text" name="cpf" value='<fmt:formatMask
6        value="${cliente.cpf}" pattern="000.000.000-00" />' />
7  </form>
8  <form action="/departamentos">
9    <input type="text" name="nome" value="${departamento.nome}" />
10   <input type="text" name="end" value="${departamento.end}" />
11 </form>
```

Uma vez que as AC contemporâneas usam arquiteturas web com o padrão MVC (*Model-View-Controller*) [8], a duplicação de GUI não ocorre apenas nas telas, que geralmente são compostas por scripts HTML (*View*). Também há duplicação no *Controller*, que é responsável por trafegar dados entre a *View* e o *Model* (modelo do domínio), além de coordenar a transição de telas. Nos *controllers*, as duplicações ocorrem em vários pontos: na conversão dos campos de formulário em atributos de objetos do *Model*, tais como os campos do tipo *Date*, que sempre são convertidos e formatados da mesma forma em uma AC; na transição de telas, por exemplo, em módulos CRUD (criação, leitura, edição e remoção) semelhantes para todas as entidades, nos quais, após salvar um item com sucesso, deve-se retornar do formulário para a tela de listagem; e nos métodos HTTP que precisam ser implementados, no caso de *RESTful web services* [6].

A Listagem 1.2 apresenta o código de dois *controllers*, onde pode-se ver as duplicações de código. Primeiramente, a estrutura das classes se repete (declaração da classe e assinatura dos métodos), variando apenas a parte do nome da classe que é referente a sua respectiva entidade no modelo de domínio. Também há repetição de código na conversão dos campos

de formulários em atributos, de forma bem semelhante ao que foi detectado na Listagem 1.1: linhas 4 e 20, para o campo "nome"; linhas 20 e 21 para campos do tipo *String*; e todos os campos *Date* passariam pelo processo das linhas 5 e 6. As linhas 9-14 e 22-27, referentes à transição de telas, estão bem similares para as duas entidades. Por fim, esse código apresenta mais uma duplicação no nível da entidade, pois para salvar toda entidade é preciso: criar um objeto, linhas 3 e 19; converter e povoar seus atributos, linhas 4-8 e 20-21; salvar o objeto numa fachada (que representa a lógica do domínio), linhas 9 e 22; e decidir qual a próxima página que será exibida, linhas 9-14 e 22-27.

Código Fonte 1.2: HTTP Controllers para Cliente e Departamento

```
1  class ClienteController extends Controller {
2    public void doPost(HttpRequest request, HttpResponse response) {
3      Cliente cliente = new Cliente();
4      cliente.setNome(request.getParameter("nome"));
5      cliente.setNasc(toDate(
6          request.getParameter("nasc"), "dd/MM/yyyy"));
7      cliente.setCpf(removeMaskInt(
8          request.getParameter("cpf"), "000.000.000-00"));
9      Result result = facade.save(customer);
10     if(result.isOk()) {
11       response.redirectTo(LISTING, facade.findAllClientes());
12     } else {
13       response.redirectTo(FORM, result.getErrors());
14     }
15   }
16 }
17 class DepartamentoController extends Controller {
18   public void doPost(HttpRequest request, HttpResponse response) {
19     Departamento departamento = new Departamento();
20     departamento.setNome(request.getParameter("nome"));
21     departamento.setEnd(request.getParameter("end"));
22     Result result = facade.save(departamento);
23     if(result.isOk()) {
24       response.redirectTo(LISTING, facade.findAllDepartamentos());
25     } else {
26       response.redirectTo(FORM, result.getErrors());
```

```
27          }
28      }
29  }
```

Uma conclusão desses dois exemplos de código é que, devido ao acoplamento entre a GUI e as entidades de domínio, existem muitas ocorrências de duplicação na GUI. Os *scripts* e *controllers* têm muitos códigos similares que não podem ser fatorados em elementos abstratos de software, o que dificulta a manutenção dos sistemas. Por exemplo, o cliente decide alterar a lógica de transição de telas em um CRUD. Após um item ser salvo com sucesso, a AC deve abrir uma tela para visualizar este item, em vez de voltar diretamente para a listagem. Nesse caso, todos os *controllers* existentes precisarão ser modificados, a fim de implementar apenas uma mudança conceitual.

De fato, Guerra et al. concordam que reusar código de GUI que interage com entidades de um domínio específico é custoso. Assim, criar componentes de GUI que podem ser usados em mais de uma tela é difícil [11].

O surgimento da tecnologia *Ajax* [10] permitiu às páginas HTML se atualizarem assincronamente, sem precisar recarregar todo o *DOM* (*Document Object Model*), porém potencializou a duplicação de código de GUI. Com a finalidade de reduzir a carga de processamento no servidor e aumentar a velocidade de resposta das páginas web, a arquitetura das aplicações web modernas usa *Ajax* para remover o processamento da GUI do servidor web e realizá-lo no cliente (navegador). Consequentemente, o código GUI é implementado (e duplicado) usando a linguagem *JavaScript* e seu modelo complicado de orientação a objetos, que dificulta a organização e fatoração do código [2].

Dessa forma, as camadas de GUI e de domínio foram definitivamente separadas em dois processos diferentes e sua comunicação se tornou remota. É comum que, em aplicações web modernas, a comunicação GUI - servidor siga a abordagem REST [6]. Os *controllers* devem implementar verbos e códigos de retorno do protocolo HTTP, fornecendo dados (em JSON ou XML) em vez de páginas HTML. No cliente, também há *controllers* que implementam a conversão de dados e a transição de telas.

Em resumo, após Ajax, ainda há duplicação de código nos *controllers* do servidor e nas páginas HTML do cliente. Porém, a duplicação no cliente também ocorre em *controllers* que precisam ser implementados em *JavaScript* no navegador web. Portanto, acreditamos que a

dificuldade de manutenção de GUI deve ser ainda maior nas aplicações web modernas.

Diante desse contexto, algumas abordagens foram desenvolvidas para reduzir os efeitos da duplicação de GUI e, consequentemente, aumentar a produtividade dos programadores de GUI. Dentre essas abordagens, identificamos os arcabouços de *Scaffold*, os Componentes Gráficos baseados em Metadados e os Padrões de Renderização.

Arcabouços de Scaffold. Em inglês, o termo *Scaffold* representa uma estrutura simples para sustentar temporariamente uma construção definitiva. Em software, as plataformas de *Scaffold* geram automaticamente código funcional a partir das entidades de domínio. O programador precisa codificar apenas as entidades do modelo conceitual, com seus atributos, relacionamentos e alguns metadados, e as plataformas – tais como Ruby on Rails [18], Grails [16], Django [7] e Spring Roo [17] – geram código para *view*, *controllers*, classes de persistência, validação e até o banco de dados. Como resultado, sistemas podem estar funcionais em poucas horas ou dias, ao invés das semanas ou meses que são necessárias quando desenvolvedores usam tecnologias tradicionais.

Entretanto, há cenários onde *Scaffold* não consegue gerar todas as especificidades dos requisitos do usuário, obrigando o desenvolvedor a ler, testar e adaptar o código já gerado. Essa situação revela dois níveis de **Interface de Programação de Aplicações** ou API, do inglês *Application Programming Interface*, nas plataformas *Scaffold*: o **alto nível** das entidades de modelo conceitual, que são fáceis para codificar e entender; e o **baixo nível** do código gerado que é difícil de manter manualmente e possui os mesmos problemas de duplicação identificados nas listagens acima. Consequentemente, as aplicações que utilizam *Scaffold* podem apresentar dificuldades para manutenção.

Componentes Gráficos baseados em MetaDados. Guerra et al. propuseram o uso de metadados para gerar componentes gráficos baseados na estrutura das entidades de domínio [11]. Os componentes gráficos são definidos a partir dos metadados do modelo conceitual e outros metadados adicionais que podem ser usados para personalizar a visualização e o comportamento desses componentes.

O SwingBean[1], proposto por Guerra, é um arcabouço para construção de GUI baseado em metadados para aplicações Java. Esse arcabouço provê componentes de GUI para re-

[1] http://swingbean.sourceforge.net/

presentar formulários e tabelas a partir de *Java Beans* [2]. Para isso, o desenvolvedor cria um arquivo XML, descrevendo informações como o posicionamento dos campos, conteúdos de caixas seletoras, configuração de componentes e validação. Dessa forma, a aplicação Java provê o arquivo XML e uma classe Bean para o arcabouço, que por sua vez constrói um componente gráfico baseado nas descrições do XML e dos metadados do Bean, conectando os campos do componente às propriedades do Bean. A Figura 1.1 ilustra esse processo.

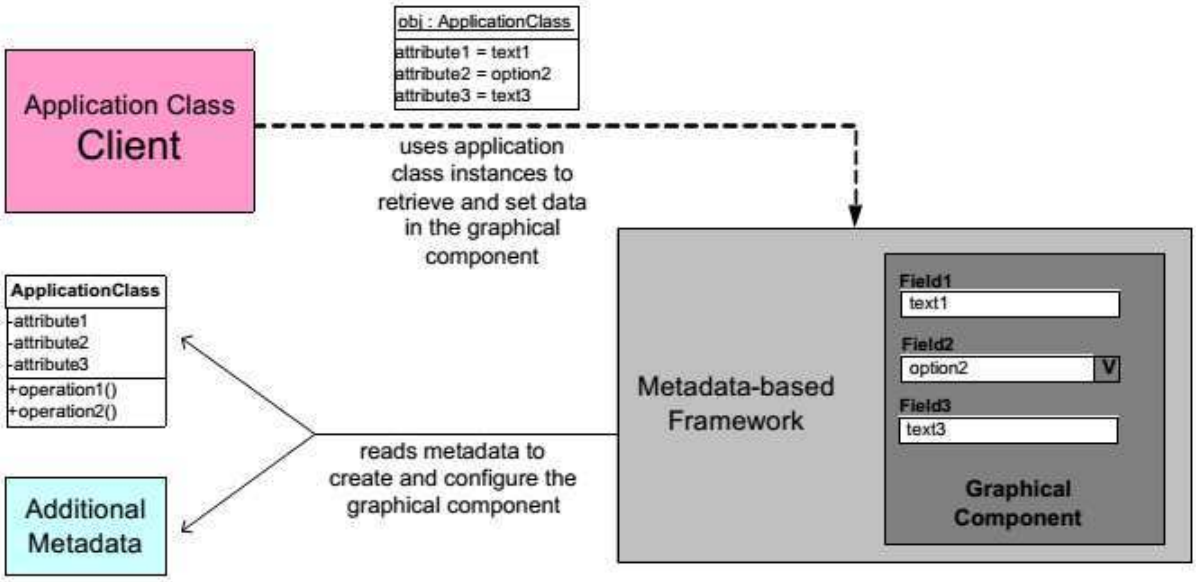

Figura 1.1: Representação de Componentes Gráficos baseado em Metadados

Quando uma aplicação contem duas entidades que possuem atributos com o mesmo nome, é possível reusar o mesmo descritor XML para construir os componente gráficos. Por exemplo, quando as entidades Cliente e Funcionário possuem nome e data de nascimento como propriedades. Todavia, essa abordagem não permite a criação de componentes genéricos que se adaptem a qualquer entidade do sistema. De fato, com essa abordagem não é possível fatorar duplicação de código no nível de entidade, o que implica nos mesmos problemas apresentados nas Listagem 1.1 e Listagem 1.2.

Padrões de Renderização. Welick et al. definiram uma outra abordagem para organizar código de GUI em aplicações adaptativas que seguiam a abordagem AOM (*Adaptive Object Model* [21]). Eles propuseram três padrões de projeto para encapsular responsabilidades de

[2]Java Bean é uma classe Java que expõe propriedades, seguindo uma convenção de nomeclatura simples para os métodos getter e setter

GUI, baseado nos elementos do modelo conceitual [20]. O padrão com granularidade mais fina é o *Property Renderer*, que define classes para renderizar GUI para tipos específicos de propriedades em um certo contexto. Por exemplo, poderia haver uma classe para renderizar todos os atributos *String* no contexto de edição e outra classe para renderizar todos os atributos *Date* no contexto de visualização. Os padrões *Entity View* e *Dynamic View* coordenam um conjunto de *Property Renderers* para renderizar telas para suas entidades. *Entity View* renderiza GUI para uma instância de entidade, tal como formulários de criação, e *Dynamic View* renderiza GUI para várias instâncias, como uma Listagem em tabela.

Uma das vantagens dessa abordagem é a redução na replicação de código de GUI [20]. Pode-se ter, por exemplo, um *Entity View* que, após o processamento dos dados submetidos por um formulário web, exibe a tela de listagem, assim como é feito na Listagem 1.2. Esse mesmo renderizador pode ser reusado para qualquer entidade, eliminando a duplicação de código no *controller*. Outra vantagem é que os renderizadores podem ser evoluídos independentemente do modelo conceitual. Isso permite que uma mudança conceitual de GUI, por exemplo, redirecionar para a tela de detalhes ao invés da tela de listagem, seja feita em um único lugar e refletida em todas as telas das entidades já existentes.

Os Padrões de Renderização direcionam o programador a criar elementos de GUI com responsabilidades bem definidas [15], em relação às funcionalidades de GUI relativas a entidades e propriedades do modelo conceitual. Desse modo, as duplicações de código que foram mostradas nas Listagem 1.1 e Listagem 1.2 podem ser eliminadas.

No entanto, não foi encontrado nenhum trabalho que detalhe como implementar os Padrões de Renderização e executá-los a fim de gerar interface gráfica a partir do modelo conceitual. Welick et al. [20] apresentam vários diagramas de classe e algumas telas de GUI mas não abordaram os detalhes de implementação dos Padrões de Renderização. Dois arcabouços para construção de aplicações AOM, como Ink [1] e o *Oghma* [5], foram propostos e implementados[3], entretanto não há nenhuma evidência de implementação de GUI, portanto não implementaram os Padrões de Renderização.

Em resumo, não foram identificadas tecnologias disponíveis na indústria para implementar GUI com os Padrões de Renderização, o que contrasta com a abundância de arcabouços existentes para a técnica de *Scaffold*. Além disso, ainda não foi possível verificar de fato a

[3]https://code.google.com/a/eclipselabs.org/p/ink/

eficácia dos Padrões de Renderização para reduzir a duplicação de código de GUI em aplicações corporativas.

1.2 Problema

No desenvolvimento de GUI para Aplicações Corporativas, a replicação de código é comumente identificada. Isso resulta em um aumento do tamanho e da complexidade do sistema e torna a manutenção custosa.

- **Problema de Negócio**: Manter GUI para Aplicações Corporativas é custoso, pois muitos elementos estão repetidos nas telas.

Os arcabouços de *Scaffold* conseguem gerar GUI a partir dos elementos do modelo conceitual. Entretanto, quando é necessário alterar o código gerado, o desenvolvedor irá enfrentar os mesmos problemas citados anteriormente, visto que o código gerado também apresenta duplicações.

Os Padrões de Renderização permitem fatorar o código de GUI duplicado, a partir dos elementos do modelo conceitual. Todavia, não foram encontrados detalhes de como implementar esses padrões, na literatura ou na indústria.

- **Problema Técnico**: Não se conhece a viabilidade e a eficácia dos Padrões de Renderização, em relação à eliminação de duplicação de código de GUI em Aplicações Corporativas.

1.3 Objetivos

O propósito deste trabalho é construir e avaliar um arcabouço para geração de GUI em Aplicações Corporativas baseado em elementos do modelo conceitual, com a finalidade de reduzir a duplicação de código, no contexto de aplicações web modernas.

Os objetivos específicos são:

1. Analisar os cenários em que os arcabouços de *Scaffold* geram duplicação de código.
2. Definir um modelo arquitetural de um arcabouço, baseado nos Padrões de Renderização, para GUI de aplicações corporativas.

3. Validar o arcabouço proposto com um estudo de caso, utilizando os cenários nos quais a geração de GUI com arcabouços de *Scaffold* produz duplicação de código.

1.4 Metodologia

Para o desenvolvimento deste trabalho, foram realizadas as seguintes atividades:

1. Revisão Bibliográfica sobre as abordagens *Scaffold* e Padrões de Renderização para desenvolvimento de GUI;

2. Elaboração e análise de cenários que revelam a duplicação de código de GUI no código gerado por arcabouços de *Scaffold*;

3. Especificação de uma arquitetura de um arcabouço para construção de GUI baseada nos Padrões de Renderização;

4. Desenvolvimento de um arcabouço para geração de interface gráfica conforme modelo arquitetural elaborado;

5. Realização de um estudo de caso para verificar a eficácia da solução proposta em relação à eliminação de duplicação de código de GUI em Aplicações Corporativas.

1.5 Contribuições

Este trabalho apresenta as seguintes contribuições:

1. Uma avaliação das limitações do *scaffold*. Essa avaliação apresenta cenários que evidenciam as limitações do *scaffold* e suas consequências. Ela poderá ser utilizada como referência para trabalhos relacionados com o *scaffold*.

2. Um arcabouço *open-source* baseado em metadados para reduzir a duplicação de código em interfaces gráficas de aplicações corporativas. Esse arcabouço está disponível gratuitamente na internet e pode ser usado livremente. Ele foi dividido em dois projetos: o *meta-gui-server*[4] e o *meta-gui-web*[5].

[4]Disponível em https://github.com/rodrigovilar/meta-gui-server

[5]Disponível em https://github.com/delanohelio/meta-gui-web

3. Um artigo aceito no EuroPlop 2015. O artigo de título *Rendering patterns for modern Web Enterprise Applications* foi aceito na *20th European Conference on Pattern Languages of Programs*.

1.6 Estrutura do trabalho

No Capítulo 2 são apresentados os conceitos básicos sobre *Scaffold* e Padrões de Renderização, as suas mecânicas e seus componentes. Também é detalhada a estrutura de cada *Rendering Pattern* e apresentam-se exemplos de uso.

No Capítulo 3 discutem-se cenários onde *Scaffold* apresentam dificuldades para fatorar código de GUI, apresentando as consequências e suas limitações.

No Capítulo 4 é apresentado em detalhes um arcabouço baseado nos Padrões de Renderização para reduzir o problema de duplicação de código de GUI em aplicações corporativas.

No Capítulo 5 é apresentado um estudo de caso onde o arcabouço do Capítulo 4 é comparado com uma plataforma *Scaffold*.

Por fim, no Capítulo 6 são feitas as conclusões deste trabalho e propostos trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

Neste capítulo são apresentados os conceitos básicos de GUI, como telas e *widgets*. Além disso, duas tecnologias para prototipagem rápida de GUI serão fundamentadas: *Scaffold* e *Rendering Patterns*.

2.1 Conceitos Básicos de GUI

Em sistemas com GUI, a interação entre usuário e o software é feita através de telas, que apresentam dados em um plano cartesiano. Embora o usuário final não perceba completamente, cada tela é formada por diversos componentes, que são denominados de *widgets* [19].

De forma análoga às responsabilidades dos objetos de uma linguagem orientada a objetos [15], os *widgets* possuem responsabilidades com diferentes granularidades. Alguns *widgets* são responsáveis por desenhar um módulo completo do sistema em muitas telas, enquanto que outros são responsáveis por desenhar pedaços pequenos da tela, por exemplo: caixas de texto, para entrada de dados do usuário; ou botões que representam ações na aplicação. Comumente, os *widgets* com muitas responsabilidades delegam partes da tela para serem renderizadas por *widgets filhos*. Dessa forma, pode haver telas que são formadas por uma composição de dezenas ou centenas de *widgets*.

2.2 Scaffold

Alguns arcabouços para desenvolvimento ágil, tais como Ruby on Rails [18], Grails [16], Spring Roo [17] e Django [7], possuem ampla adoção no mercado de desenvolvimento de software. Isso possivelmente se dá pela simplicidade que tais arcabouços oferecem para construir software funcional de forma rápida, fácil e interativa.

Esses arcabouços fazem uso de uma técnica chamada **Scaffold** que gera código funcional automaticamente. Para isso o *Scaffold* processa as entidades do modelo de domínio, junto com *templates* de geração de código pré-definidos, resultando em artefatos para *view*, *controller*, classes de persistências, validação e banco de dados. Assim sendo, o usuário final pode executar operações CRUD poucos minutos após requisitar uma funcionalidade. Esse *feedback* rápido ajuda a corrigir eventuais desvios nos requisitos do software, reduzindo o retrabalho.

A geração de código é uma solução para evitar os problemas causados com a duplicação de código [13]. Isso é possível pois o código redundante é abstraído em um código modelo. Assim, com *Scaffold*, o desenvolvedor deveria se preocupar apenas em modelar as entidades de domínio, adicionando, se necessário, metadados referentes a configurações específicas para a geração do código.

Ruby on Rails, por exemplo, foi modelado baseado em dois conceitos chave: DRY e **Convenção sobre Configuração**, do inglês *Convention over Configuration* (CoC). O conceito DRY combate a duplicação de código, guiando o desenvolvedor a especificar o que é necessário em seu devido lugar. CoC permite que o desenvolvedor especifique o que é indispensável, ou seja, aquilo que não é convencional.

A Listagem 2.1 refere-se a duas classes de entidades que um programador poderia escrever e submeter para um arcabouço de *Scaffold*, para que assim seja gerado o código da Listagem 1.1 e Listagem 1.2 automaticamente. A propósito, o código da Listagem 2.1 tem um alto nível de abstração e é mais fácil de ler que as listagens anteriores.

Código Fonte 2.1: Entidades de domínio usadas para gerar código por plataformas *Scaffold*

```
1 class Cliente {
2   String nome
3   @Format("dd/mm/yyyy") Date dataNasc
4   @Mask("000.000.000-00") String cpf
```

```
5 }
6 class Departamento {
7   String nome
8   String endereco
9 }
```

A Listagem 2.2 é um trecho de código de GUI gerado por uma plataforma *Scaffold* a partir da Listagem 2.1 que apresenta todas as instâncias da entidade Cliente em uma tabela. Pode-se perceber que nas linhas 4 e 5 existe uma referência direta às propriedades de Cliente. A princípio isso não é uma violação do DRY, pois esse código foi gerado automaticamente e ele não faz parte do espaço de trabalho do desenvolvedor.

Código Fonte 2.2: Trecho de código GUI gerado por *Scaffold*

```
1 <tbody>
2   <g:each in="${clienteList}" status="i" var="cliente">
3     <tr class="${(i % 2) == 0 ? 'even' : 'odd'}">
4       <td><g:link action="show" id="${cliente.id}">${fieldValue(bean:
            cliente, field: "nome")}</g:link></td>
5       <td><g:formatDate format="dd/mm/yyyy" date="${cliente.dataNasc}"
            /></td>
6         ...
7     </tr>
8   </g:each>
9 </tbody>
```

2.3 Padrões de Renderização

Os Padrões de Renderização são parte do estilo arquitetural *Adaptive Object Model* (AOM), que é uma solução para sistemas com intensa adaptação de requisitos, mesmo quando é necessário mudar funcionalidades do sistema em tempo de execução. Para prover essas funcionalidades, AOM interpreta os metamodelos em tempo de execução [22]. Assim, para possibilitar mudanças, sistemas AOM podem prover uma API para editar seus metamodelos.

Type Square é o principal padrão de AOM e define um modelo conceitual com entidades e suas propriedades em objetos do tipo *EntityType* e *PropertyType*. Ele também representa

instâncias de entidades e propriedades como objetos de *Entity* e *Property*. Os Padrões de Renderização dependem do *Type Square* para organizar a renderização de código de GUI.

Em AOM, quando o usuário executa o módulo de um *EntityType*, tal como Cliente ou Produto, a aplicação procura por uma classe *Dynamic View*, que vai desenhar a estrutura do módulo na GUI. Por exemplo, uma classe *ListingTable* renderiza uma tela principal, com uma lista de todos os clientes e links para criar, editar e removê-los. Uma alternativa poderia ser usar um *IconView* para mostrar as fotos dos clientes como o *Windows Explorer* faz com arquivos de imagens.

Além disso, a classe *Dynamic View* delega a renderização de alguns fragmentos da GUI, tais como páginas de formulários e relatórios, para *Entity Views* e *Property Renderers* [20]. Os Padrões de Renderização serão detalhados a seguir.

Property Renderer. Esse padrão é responsável por gerar a interface gráfica de um tipo de propriedade dependendo de um dado contexto, como nos exemplos ilustrados na Figura 2.1. Esse padrão precisa conhecer o tipo da propriedade (String, Inteiro, Data, Booleano, Binário, etc.) e o contexto (visualização, edição, listagem, etc.) que estão sendo utilizados para que possa gerar o *widget* adequado. Inicialmente, o padrão provê uma implementação *default* capaz de gerar uma mínima interface gráfica para todos os tipos e contextos. Essa funcionalidade é útil nas primeiras fases de prototipagem.

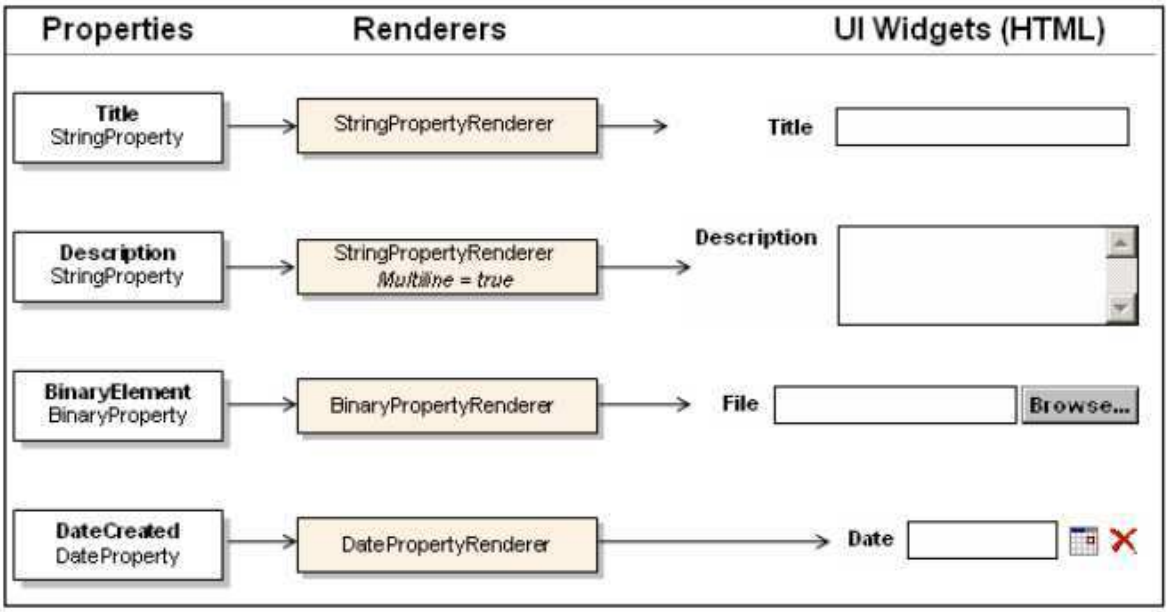

Figura 2.1: Exemplos de Property Renderer

Cada *PropertyType* demanda GUI para vários contextos. Pode-se ter, por exemplo, um *DateProperty* sendo usado em modo edição ou de leitura, onde cada um desses contextos exige o uso de um *widget* específico. Assim é preciso ter um *Property Renderer* especializado para cada tipo e cada contexto a fim de produzir os *widgets* necessários.

Entity View. O padrão *Entity View* é responsável por coordenar os vários *Property Rendererers* de uma entidade para gerar uma interface gráfica mais complexa. O contexto simboliza o propósito da interface. Por exemplo, pode-se gerar um formulário com o propósito de editar as informações de um objeto. Nesse caso, o contexto seria de edição, como ilustrado no exemplo na Figura 2.2. A sequência e a composição dos *Property Rendererers* podem ser especificadas no código ou utilizando metadados. O *Entity View* deve estar ciente do contexto onde os *widgets* serão aplicados, e o contexto pode conter informações adicionais que serão utilizadas durante o processo de renderização gráfica.

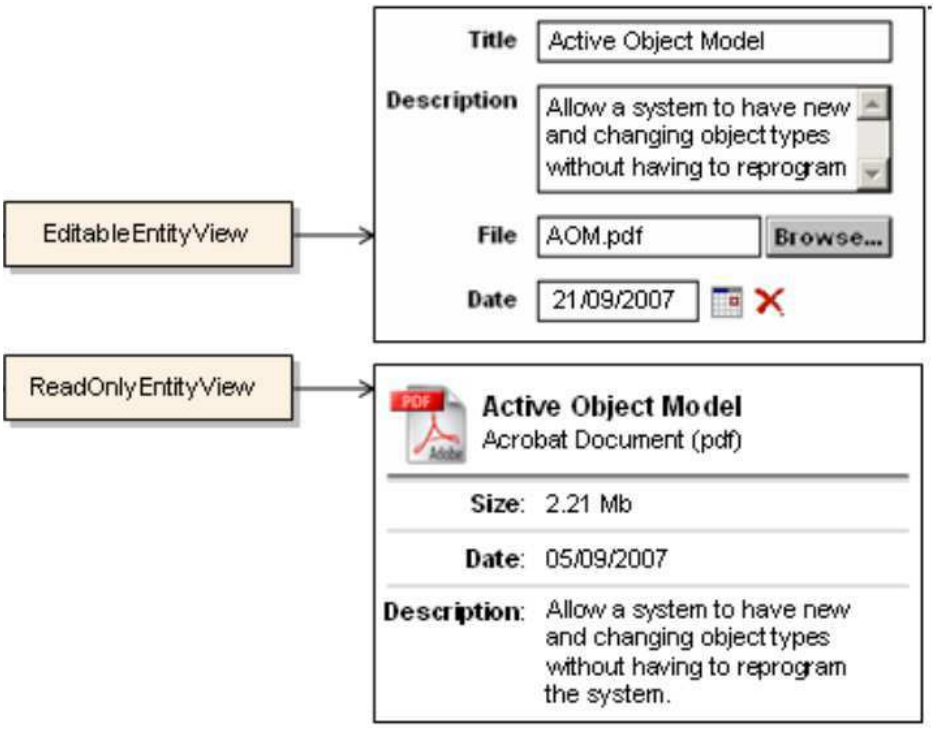

Figura 2.2: Exemplos de Entity View

Dynamic View. O padrão *Dynamic View* é especializado em gerar interface gráfica para um conjunto de entidades (ou instâncias do mesmo tipo de entidade). Ele recebe como entrada um conjunto de entidades e renderiza a interface gráfica de acordo com o propósito da tela, podendo apresentar telas para o mesmo modelo de diferentes formas. Esse padrão

é capaz de gerar o código da interface de um conjunto de entidades sem acoplamento ou sem precisar referenciar nada da interface no modelo. O *Dynamic View* pode gerar as telas a partir dos outros padrões como o *Property Renderer* e o *Entity View*. A Figura 2.3 mostra um exemplo da aplicação desse padrão.

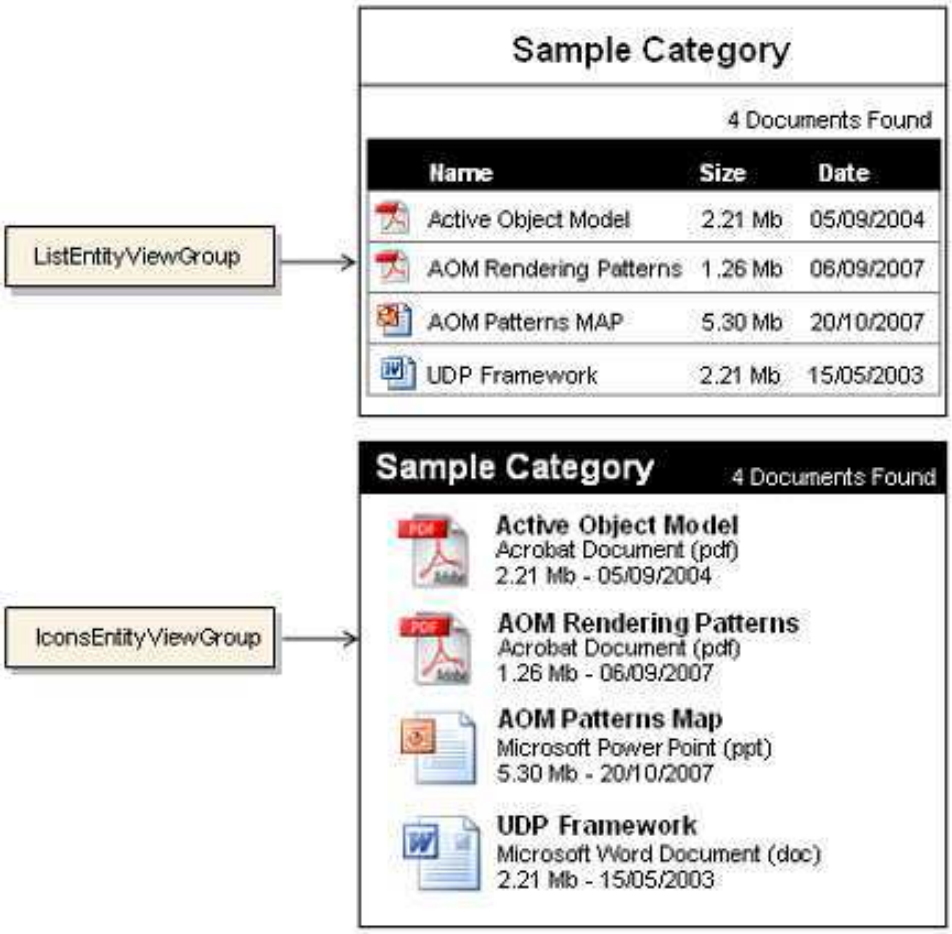

Figura 2.3: Exemplos de Dynamic View

Capítulo 3

Cenários que evidenciam as limitações de Scaffold

Neste capítulo são descritos alguns cenários onde as plataformas de *Scaffold* apresentam dificuldades para fatorar o código de GUI, com a finalidade de destacar as limitações dessa abordagem.

A presente análise não considerará o esforço necessário para desenvolver *widgets* do zero. Em vez disso, considera-se que os *widgets* já estão implementados em JavaScript e apenas precisam ser referenciados no código da GUI da aplicação corporativa. De fato, há muitas bibliotecas JavaScript, tais como Bootstrap[1], jQuery UI[2] e AngularJS[3], cujos componentes podem ser anexados nos elementos de uma página HTML. Alguns dos *widgets* usados nesta análise são hipotéticos, mas representam situações reais que podem acontecer com *widgets* concretos.

3.1 Cenário 1 - Widgets Não Antecipados

Quando o cliente demanda um widget específico que não foi previsto na tecnologia de desenvolvimento da GUI.

[1] disponível em: http://getbootstrap.com/components/

[2] disponível em: https://jqueryui.com/

[3] disponível em: https://angularjs.org/

3.1.1 Exemplos

Em um sistema de comércio eletrônico, o cliente requisita a funcionalidade de cadastro de fornecedores, com os campos "nome fantasia", "razão social", "CNPJ", "endereço" e "CEP". Inicialmente todos esses campos são implementados como *Strings* com formatação livre, todavia o cliente logo nota que a GUI pode ser melhorada nos seguintes pontos:

1. O *widget* que recebe os dados de "CNPJ" pode possuir uma máscara do tipo **##.###.###/####-##**, que insere os pontos, a barra e o traço automaticamente, além de só aceitar números nas posições marcadas com #;

2. A mesma ideia se aplica ao campo "CEP", cuja máscara é **#####-###**;

3. O algoritmo para validação dos dígitos verificadores do "CNPJ", poderia ser implementado no navegador, a fim de apontar o erro rapidamente quando o usuário inserir um CNPJ inválido, sem precisar enviar os dados para serem processados no servidor;

4. O *widget* do "CEP" pode conectar o navegador com um serviço web remoto, com o intuito de validar o CEP e recuperar a informação do restante do endereço, populando outros campos do formulário automaticamente.

3.1.2 Solução com Scaffold

Inicialmente o desenvolvedor precisa definir uma entidade de domínio e descrever os seus atributos. Para este cenário, ele deve implementar uma entidade **Fornecedor** com os respectivos campos. A Listagem 3.1 apresenta um exemplo de código que pode ser submetido para uma plataforma de *Scaffold* genérico, a fim de gerar código para GUI da funcionalidade de cadastro de fornecedores.

Código Fonte 3.1: Entidade de domínio para o Cenário 1

```
1 class Fornecedor {
2   String nomeFantasia
3   String razaoSocial
4   @Mask("###.###.###-##") String cnpj
5   String endereco
6   @Mask("#####-###") String cep
```

```
7 }
```

O desenvolvedor utilizou uma anotação para aplicar as respectivas máscaras nos campos "CNPJ" e "CEP", pois a funcionalidade de máscara é comum e genérica o suficiente para ser aplicada em qualquer campo do tipo *String*. No entanto, a funcionalidade de validação do dígito verificador e o uso do servidor de endereços remoto são específicos deste cenário e dificilmente seriam generalizados. Dessa forma, a plataforma de *Scaffold* não pode prevê-los e o código gerado precisa ser alterado manualmente pelo desenvolvedor.

Dado que o código de validação de CNPJ já está implementado em alguma função JavaScript, uma das formas mais simples de alterar o *widget* de "CNPJ", para validar os dígitos verificadores do valor inserido, é usar uns seletores de atributos, como **id** ou **class** [3]. Dessa forma, ao carregar um script, por exemplo *validador.cnpj.js*, uma função pode buscar todos os elementos marcados com o campo **class="cnpj"** e ativar o código de validação de CNPJ neles.

Para implementar a consulta do CEP em um servidor remoto, a solução também se baseia em seletores (**class="consultaCep"**), mas é preciso usar um atributo para indicar o endereço do servidor: **servidor-consulta-cep="url"**. Para que o campo de "endereço" seja preenchido automaticamente no retorno da consulta, o seu campo deve ser marcado com outro seletor, por exemplo, **class="consultaCepCallBack"**.

A Listagem 3.2 apresenta um trecho de código de GUI que foi gerado por *Scaffold* e alterado manualmente para implementar as funcionalidades requisitadas pelo cliente nesse cenário.

Código Fonte 3.2: Resolvendo o Cenário 1 com *Scaffold*

```
1 <form action="/fornecedores" method="POST">
2   <!-- ... -->
3   <input type="text" name="cnpj" value="${fornecedor.cnpj}" class="cnpj
        " />
4   <input type="text" name="cep" value="${fornecedor.cep}"  class="
        consultaCep"
5     servidor-consulta-cep="http://validadorcep.gov.br" />
6   <input type="text" name="endereco" value="${fornecedor.endereco}"
7       class="consultaCepCallBack" />
8   <!-- ... -->
```

```
9  </form>
10 <!-- ... -->
11 <script src="validador.cnpj.js" />
12 <script src="consulta.cep.js" />
```

3.1.3 Análise

Embora a alteração no campo "CNPJ" seja simples, ela não pode ser feita automaticamente, pois a lógica dos dígitos verificadores é específica de cada domínio de problema, e pode ser implementada de várias formas diferentes na linguagem JavaScript. Além disso, essa mesma alteração precisará ser feita em todos os pontos da aplicação onde há *widgets* para "CNPJ".

A alteração do campo "CEP" é maior, pois envolve mais atributos e também afeta o campo "endereço". Novamente, essa é uma funcionalidade com domínio específico (endereçamento postal brasileiro), que não pode ser implementada automaticamente por *Scaffold*.

Em resumo, não há metadados disponíveis para geração de *widgets* específicos para "CNPJ" ou "CEP". Estes e quaisquer *widgets* não antecipados pelas plataformas de *Scaffold* demandam que o desenvolvedor altere manualmente o código gerado, que possui violações do princípio DRY. Dessa forma, o desenvolvedor terá que lidar com os problemas de código duplicado, anteriormente mencionados.

3.2 Cenário 2 - Alterações transversais em Propriedades

Quando o cliente decide mudar o widget padrão de uma propriedade em pontos específicos ou em todo o sistema.

3.2.1 Exemplos

A mesma aplicação de comércio eletrônico do cenário anterior precisa ser atualizada em vários cadastros, para implementar as seguintes alterações:

1. Todos os campos de edição de datas do sistema devem exibir um calendário *pop-up*, permitindo que o usuário selecione datas de forma mais intuitiva;

2. Nas telas de visualização, todos os campos de e-mail, ao serem clicados, devem direcionar o usuário para sua aplicação padrão de e-mail;

3. Ao clicar no campo de e-mail do Funcionário, a aplicação padrão de e-mail deve ser aberta passando, além do e-mail do funcionário, o e-mail da equipe de RH da empresa como cópia da mensagem.

3.2.2 Solução com *Scaffold*

Em aplicações corporativas, os campos do tipo *Date* podem ser usados com diversos propósitos: data de nascimento de pessoas (clientes, funcionários, fornecedores), data de realização de operações (compras, vendas, contratações, pagamentos), início e término de períodos, etc. Se o proprietário da aplicação decidir alterar a forma como todas as datas são manipuladas no sistema, o código de GUI deve ser alterado em todos os pontos nos quais são utilizadas datas. Isso ocorre por causa da duplicação de código fruto das tecnologias atuais para GUI, inclusive *Scaffold*.

Semelhantemente, os dados de e-mails podem ser usados em diversas entidades de uma aplicação corporativa, e as alterações transversais, que envolvem os campos de e-mail, precisam ser alterados em todos os trechos duplicados.

A Listagem 3.3 apresenta um exemplo de código que pode ser submetido para uma plataforma *Scaffold*, a fim de gerar código para cadastro de algumas entidades com datas e e-mails.

Código Fonte 3.3: Trechos de entidades do Cenário 2

```
1  class Cliente {
2    Date dataDeNascimento
3    String email
4  }
5  class Funcionario {
6    Date dataDeNascimento
7    String email
8  }
9  class Fornecedor {
10   Date dataDeNascimento
11 }
```

```
12 class Venda {
13   Date dataRealizacao
14   Funcionario vendedor
15 }
16 class Departamento {
17   Date dataAbertura
18   String emailListaDeDiscussao
19 }
20 class Promocao {
21   Date inicio
22   Date fim
23 }
```

A primeira alteração deste cenário modifica os campos de data para que eles usem um calendário *pop-up*. De forma semelhante ao cenário anterior, pode-se marcar os campos data com um seletor **class="calendario"** e invocar uma função JavaScript que busca e altera o comportamento dos campos marcados.

A segunda alteração afeta os campos textuais que representam e-mail, adicionando um link no formato **"mailto:email"**. A terceira alteração é similar à segunda, porém há diferenças importantes: essa funcionalidade é exclusiva para o e-mail de funcionários; e deve ser adicionado mais um parâmetro no link desse campo. Assim sendo, especificamente na tela de visualização de funcionário, o campo textual de e-mail terá um link referenciando **"mailto:email?cc=emailDaEquipeDeRH"**.

Para implementar as alterações deste cenário, o desenvolvedor depende da abordagem utilizada na plataforma de *Scaffold* escolhida: geração de código passiva ou ativa [12].

Na geração de código ativa, as mudanças são feitas nos *templates* de geração de código. Esses *templates* são combinados com as entidades de domínio para gerar o código das demais camadas da aplicação. Dessa forma, após alterar um *template* como na Listagem 3.4, o código que for gerado para as propriedades do tipo data vai usar um *widget* de calendário e os campos de e-mail possuirão um link *mailto*.

Código Fonte 3.4: Alterando um template de geração de código

```
1 <!-- Template para formularios de edicao -->
2 <form action="/#{resource(entidade)}" method="PUT">
3   <g:renderFieldsByType entity="#{entidade}">
```

```
4  </form>
5  <!-- Template para edicao de data -->
6  <input name="#{campo.nome}" class="calendario"
7    value='<fmt:formatDate
8      value="${#{campo.entidade.nome}.#{campo.nome}}" />'/>
9  <!-- Template para visualizacao de campos String -->
10 <g:if contains="#{campo.nome};mail">
11   <g:if equals="#{campo.entidade.nome};Funcionario">
12     <a href="mailto:${#{campo.entidade.nome}.#{campo.nome}}?
13       cc=emailEquipeRH">
14         ${#{campo.entidade.nome}.#{campo.nome}}</a>
15   </g:if>
16   <g:else>
17     <a href="mailto:${#{campo.entidade.nome}.#{campo.nome}}">
18       ${#{campo.entidade.nome}.#{campo.nome}}</a>
19   </g:else>
20 </g:if>
```

Por outro lado, na geração de código passiva, os desenvolvedores precisam alterar manualmente o código gerado. A Listagem 3.5 apresenta trechos de códigos resultante, que deve ser obtido independente da abordagem de geração de código

Código Fonte 3.5: Resolvendo o Cenário 2 com *Scaffold*

```
1  <!-- Trechos dos formularios de edicao -->
2  <form action="/clientes" method="PUT">
3    <input name="dataDeNascimento" class="calendario"
4      value='<fmt:formatDate
5        value="${cliente.dataDeNascimento}" />'/>
6  </form>
7  <form action="/funcionarios" method="PUT">
8    <input name="dataDeNascimento" class="calendario"
9      value='<fmt:formatDate
10       value="${funcionario.dataDeNascimento}" />'/>
11 </form>
12 <form action="/fornecedores" method="PUT">
13   <input name="dataDeNascimento" class="calendario"
14     value='<fmt:formatDate
15       value="${fornecedor.dataDeNascimento}" />'/>
```

```
16 </form>
17 <!-- Trecho da visualizacao de Cliente -->
18 <a href="mailto:${cliente.email}">
19   ${cliente.email}</a>
20 <!-- Trecho da visualizacao de Funcionario -->
21 <a href="mailto:${funcionario.email}?
22   cc=emailEquipeRH">
23     ${funcionario.email}</a>
24 <!-- Trecho da visualizacao de Departamento -->
25 <a href="mailto:${departamento.emailListaDeDiscussao}">
26   ${departamento.emailListaDeDiscussao}</a>
```

3.2.3 Análise

Embora seja mais persistente, a geração ativa de código pode ser impraticável. O código dos *templates* é muito complexo, pois lida com duas formas de representação: dados e metadados.

Na abordagem de geração passiva de código, o desenvolvedor precisa revisar todas as páginas da GUI que possuem campos de data ou e-mail, a fim de garantir que esses campos foram implementados de forma consistente. Considerando que uma aplicação corporativa pode ter dezenas ou centenas de entidades e que para cada entidade é gerado um conjunto de páginas, implementar essas alterações pode se tornar muito custoso ou até inviável.

Esses problemas são consequências das muitas duplicações que há ao gerar código de GUI com frameworks *Scaffold*.

3.3 Cenário 3 - Alterações transversais em Entidades

Quando o cliente decide mudar as telas de apresentação das entidades de uma aplicação corporativa

3.3.1 Exemplos

Em um sistema de comércio eletrônico fictício, o cliente requisitou as funcionalidades de cadastro do tipo CRUD para várias entidades do sistema. Nessa primeira versão, após as

operações que alteram os dados, a GUI é redirecionada para a tela de listagem de registros do respectivo cadastro. Sendo assim, muda-se a linha de código que faz esse redirecionamento em todos os *controllers* do sistema trocando a página que será redirecionada. Em um segundo momento, o cliente decide que:

1. A GUI deve exibir a tela de visualização de detalhes após a realização de qualquer cadastro no sistema. Só após essa tela deve aparecer a opção de voltar à listagem;

2. Haverá um relatório de detalhes de um departamento em formato de pdf.

3.3.2 Solução com Scaffold

A versão inicial do sistema possui diversas entidades de domínio, que foram utilizadas para gerar, dentre outros artefatos, *controllers* para cuidar da transição de telas. A Listagem 1.2 mostra o código fonte de dois *controllers* que são semelhantes aos *controllers* gerados por frameworks de *Scaffoding* e evidenciam a duplicação de código que existe nestas classes.

Para implementar a primeira alteração, é preciso modificar todos os *controllers* gerados, no ponto que determina a página que deve ser apresentada logo após concluir a persistência dos dados. A Listagem 3.6 apresenta o código modificado de dois *controllers* nas linhas 6 e 17. As linhas 5 e 16 estão comentadas e representam o código original.

Algumas plataformas *Scaffold* permitem implementar a alteração do relatório em PDF com o uso de *plugins*. Para tal, o desenvolvedor precisa instalar o *plugin*, configurar e alterar o código para usá-lo. Quando não houver um *plugin* disponível, o desenvolvedor terá que alterar a página gerada para usar alguma biblioteca JavaScript que permita apresentar essa página no formato PDF[4]. Nesse cenário, será apresentada a solução com o uso de *plugin* por ser mais simples para o desenvolvedor.

Com o *plugin* instalado e configurado, o desenvolvedor precisa adicionar o método *report* no *controller* e criar uma página que será usada como *template* para o *plugin* poder gerar o PDF (Listagem 3.7).

Código Fonte 3.6: Controllers para Funcionário e Departamento do Cenário 3

```
1 class FuncionarioController extends Controller {
```

[4]Um exemplo é o jsPDF, disponível em: https://parall.ax/products/jspdf

```
2     public void doPost(HttpRequest request, HttpResponse response) {
3       Result result = facade.save(funcionario);
4       if(result.isOk()) {
5         //response.redirectTo(LISTING, facade.findAll());
6         response.redirectTo(SHOW, funcionario);
7       } else {
8         response.redirectTo(FORM, result.getErrors());
9       }
10    }
11  }
12  class DepartamentoController extends Controller {
13    public void doPost(HttpRequest request, HttpResponse response) {
14      Result result = facade.save(departamento);
15      if(result.isOk()) {
16        //response.redirectTo(LISTING, facade.findAll());
17        response.redirectTo(SHOW, departamento);
18      } else {
19        response.redirectTo(FORM, result.getErrors());
20      }
21    }
22    public void report(HttpRequest request, HttpResponse response) {
23          def args = [template:"pdf", model:[departamento:
                departamentoInstance]]
24          pdfRenderingService.render(args+[controller:this],response)
25    }
26  }
```

Código Fonte 3.7: Template para geração de PDF para o Cenário 3

```
1  <g:each var="funcionario" in="${departamento.getFuncionarios()}">
2      <rendering:inlineJpeg bytes="${form?.getPhoto()}" height="200px"/>
3      <p>Nome: ${funcionario.fullName}</p>
4      <g:formatDate date="${funcionario.getDataDeNascimento()}" format="
           dd/MM/yyyy"/>
5  </g:each>
6  <!-- ... -->
```

3.3.3 Análise

Para implementar uma funcionalidade transversal em código gerado por plataforma de *Scaffold*, é preciso muito esforço. De forma análoga ao cenário anterior, para implementar alteração na transição de páginas, o desenvolvedor analisou todos os *controllers* com o intuito de alterar a página de redirecionamento. Novamente isso ocorre pois o código dos *controllers* gerados por *Scaffold* possui muita repetição, obrigando o desenvolvedor a alterar o mesmo trecho de código em diversos pontos do sistema.

3.4 Cenário 4 - Alterações transversais em Relacionamentos

Quando o cliente decide mudar widgets de relacionamentos em uma aplicação corporativa

3.4.1 Exemplos

No sistema de comércio eletrônico que tem sido utilizado como exemplo em todos os cenários, os relacionamentos entre entidades aparecem na GUI, por padrão, como *combo boxes* ou caixas de múltipla escolha, dependendo da cardinalidade **um** ou **muitos** da relação, respectivamente.

Em algum momento o proprietário do sistema decide que, por padrão, a aplicação deve representar relacionamentos na GUI como campos de texto com *auto complete*, a fim de pesquisar e selecionar objetos da outra ponta do relacionamento.

3.4.2 Solução com Scaffold

A implementação de relacionamentos em plataformas *Scaffold* é simples, pois as entidades de domínio podem ter referências únicas ou coleções de referências para objetos de outras entidades. A Listagem 3.8 apresenta um exemplo de código com um relacionamento entre Departamento e Funcionario.

Código Fonte 3.8: Entidades de domínio para o Cenário 4

```
1 class Funcionario {
```

```
2    String nome
3    Departamento departamento
4  }
5  class Departamento {
6    String nome
7    List<Funcionario> funcionarios
8  }
```

Para implementar a alteração pedida pelo proprietário, será preciso fazer modificações no código gerado, semelhante às modificações feitas nos dois cenários anteriores. Nesse cenário, devem ser substituídos os *widgets* que representam um relacionamento de um-para-muitos com Funcionário por uma caixa de texto de múltiplos valores. Não há uma solução nativa que permita fazer isso, mas há várias bibliotecas que podem ser usadas para isso. Assim, o desenvolvedor poderia adicionar o atributo **class="multipleField"** e atributos que definam os dados que podem ser utilizados para pesquisa. Neste caso seriam adicionados os atributos **data="Funcionarios.all()"** e **searchField="nome"**, ao campo de texto com o intuito de torná-lo um campo de múltipla escolha.

A Listagem 3.9 apresenta trechos de códigos de páginas geradas pela plataforma *Scaffold* com as modificações necessárias.

Código Fonte 3.9: Resolvendo o Cenário 3 com *Scaffold*

```
1  <form action="/departamento" method="POST">
2    <input type="text" name="name" value="${departamento.name}" />
3    <input type="text" name="funcionarios" class="multipleField" data="${
         Funcionarios.all()}" searchField="${nome} />
4  </form>
```

3.4.3 Análise

Assim como nos cenários anteriores, as plataformas *Scaffold* não oferecem uma solução em alto nível para implementar as alterações transversais, impondo que o desenvolvedor altere o código gerado. Além das dificuldades apresentadas nos cenários anteriores, tais como problemas de duplicação de código, o desenvolvedor precisará aplicar mais esforço já que os *widgets* de relacionamento são mais complexos que os citados em outros cenários.

Capítulo 4

Arcabouço Proposto

Neste capítulo apresenta-se uma solução para o problema de duplicação de código de GUI em aplicações corporativas: um arcabouço baseado em metadados e nos Padrões de Renderização.

4.1 Fatorando Widgets

Em um *widget* é possível encontrar muitas referências diretas aos elementos do modelo conceitual. Por exemplo, a Listagem 4.6 apresenta um trecho de código de um formulário para Cliente que apresenta um campo de texto para o nome e um calendário para selecionar a data de nascimento. Nas linhas 1-4 têm algumas referências diretas para a entidade Cliente e suas propriedades. Seria preciso alterar essas referências para reusar esse mesmo formulário ou partes dele em outras entidades.

Código Fonte 4.1: Exemplo de *widget*

```
1 <form action="/cliente" method="POST">
2   <input type="text" name="nome"/>
3   <input name="dataDeNascimento" class="datePicker" value=
4     '<fmt:formatDate pattern="dd/MM/yyyy" />'/>
5 </form>
```

De fato, para reusar um *widget* é preciso remover todas as referências diretas aos elementos do modelo conceitual. O uso de **metadados** é uma solução para fazer referências indiretas a esses elementos. A Listagem 4.2 apresenta um suposto exemplo de como seria

fatorar a Listagem 4.6 com o uso de metadados.

Código Fonte 4.2: Exemplo de um *widget* fatorado

```
1 <form action="/${entidade.nome}" method="POST">
2   <g:renderProperties in="${entidade}" context="form">
3 </form>
```

Nesse caso, todas as referências a elementos do modelo conceitual foram substituídas por metadados. Por exemplo, na linha 1 havia uma referência à entidade cliente que foi substituída por um elemento que representa o nome de uma entidade. Da mesma forma foi feito para as propriedades, todas as referências a propriedades foram substituídas por um código que constrói todos os *widgets* das propriedades de uma entidade. Esse código precisa conhecer a entidade e o contexto (listagem, formulário, relatório, etc.) em que os *widgets* serão apresentados. Logo, a Listagem 4.2 poderia ser reusada por qualquer entidade do modelo conceitual.

Na solução proposta, os tipos de *widgets* se baseiam em cada um dos Padrões de Renderização (ver Seção 2.3) para se tornarem independentes do modelo conceitual e utilizarem metadados. Assim sendo, a classe **PropertyWidget** atua como um *Property Renderer*; a classe **EntityWidget** representa *Entity View*; e a classe **EntitySetWidget** é uma *Dynamic View*. Além desses, mais um *widget* será considerado neste trabalho: a classe **RelationshipWidget** que trata da renderização de GUI para relacionamentos entre entidades do modelo do domínio.

Uma vez que, em códigos como o da Listagem 4.2, os *widgets* não conhecem diretamente os elementos do modelo conceitual pelos quais estão responsáveis por renderizar, é preciso utilizar um elemento externo ao *widget* para conectá-lo aos elementos do modelo conceitual. A solução proposta neste trabalho utiliza regras de renderização para conectar os widgets a elementos do modelo conceitual em cada um dos contextos existentes.

4.2 Regras

Regras permitem determinar o comportamento de um sistema. Assim, um conjunto de regras pode ser criado para decidir que *widget* deve ser usado para representar um elemento do modelo conceitual. Por exemplo, uma regra pode determinar que as propriedades do tipo

Date serão representadas por um calendário em uma página de formulário. Logo, essa regra permite associar um *PropertyTypeType* (o tipo *Date*) com um *widget* (o calendário) em um determinado contexto (a página de formulário). As regras não só permitem associar *PropertyTypeType* aos *widgets* como também é possível associar outros elementos do modelo conceitual (tipos de entidades, propriedades e relacionamentos) a *widgets*. Assim, para cada tipo de *widget* existe um tipo de regra.

Regras para Entity ou EntitySet. Um *Entity* representa uma instância de uma entidade do modelo conceitual, enquanto que o EntitySet representa um conjunto de instâncias. Esses dois tipos de regra permitem associar um *EntityType* a um *EntityWidget* ou *EntitySetWidget*, respectivamente, em um determinado contexto. Estes são alguns exemplos:

1. Definindo regras para *Entity*:
    - Em linha de relatório (contexto), para Pessoa (*EntityType*), use PdfTableLine (*widget*)
    - Em formulário, para * (todos as entidades), use FlowLayoutLine
2. Definindo regras para *EntitySet*:
    - Em relatório, para Pessoa, use PdfTableGenerator
    - Em listagem, para * (todos as entidades), use HtmlTableWidget

Regras para Property. Esse tipo de regra permite associar um *PropertyType* ou um tipo de *PropertyType* a um *widget PropertyWidget* em um determinado contexto. Estes são alguns exemplos:

1. Regras para definir *widgets* padrões para um tipo específico de *PropertyType*:
    - Em um formulário (contexto), para todas as propriedades do tipo String (tipo de *PropertyType*), use TextField (*widget*)
    - Em um relatório, para todas as propriedades do tipo String, use Label
2. Regras para definir *widgets* para uma entidade e/ou propriedade específica:
    - Em um formulário, para Client.cpf (*EntityType*.*PropertyType*), use CpfField.

- Em um formulário, para *.cpf (propriedade cpf de qualquer entidade), use AdvancedCpfField.

3. Regras que combinam os dois propósitos anteriores:

   - Em um relatório, para todas as String em *.cpf (propriedade cpf de qualquer entidade), use AdvancedCpfField.

Regras para Relationship. Esse tipo de regra permite associar um *Relationship* a um *RelationshipWidget* em um determinado contexto. Ele é similar às regras para *Property*, entretanto é necessário especificar na regra o tipo de cardinalidade da entidade alvo (um ou muitos) do relacionamento quando não estiver explícito. Estes são alguns exemplos:

- Em um formulário (contexto), todos os relacionamentos de cardinalidade alvo muitos, use MultiSelectField

- Em um formulário, para Cliente.dependentes (*EntityType.relationship*), use MasterDetailsPanel

A partir dos exemplos apresentados para cada tipo de regra é possível extrair um padrão na definição das regras. Por exemplo, o contexto é primordial, logo a regra tem que estar inserida em um contexto específico. Entretanto os outros dados podem variar dependendo do tipo de regra. A Tabela 4.1 apresenta os dados que são aplicados aos tipos de regras e suas variações.

Alguns *widgets* permitem ser configurados para melhor se adaptar a um contexto. Por exemplo, um *TextField* deve ser desabilitado para edição no campo Cliente.CPF. Assim sendo, é possível aplicar os parâmetros de configurações em uma regra. E as configurações serão aplicadas na renderização do *widget* no contexto da regra.

4.3 Arquitetura da Solução

A arquitetura do arcabouço é composta por quatro nós: *Browser*, *Server*, *Operational database* e *Rule repository*. A Figura 4.1 descreve as relações entre eles. O ***Browser*** carrega regras e *widgets* do ***Server*** e as coordena para construir a GUI. O *Server* tem uma API para

Tabela 4.1: Campos de descrição das regras e suas variações

Regra para:	**Entidade**	**Propriedade**	**Tipo de Propriedade**	**Cardinalidade**
Entity	específica ou todas	N/A	N/A	N/A
EntitySet	específica ou todas	N/A	N/A	N/A
Property	específica ou todas	específica ou todas	específica ou todas	N/A
Relationship	específica ou todas	específica ou todas	específica ou todas	um ou muitos

processar as requisições da GUI e outra para disponibilizar e instalar/definir *widgets* e regras. Para isso, o *Server* usa o **Operational database** e o **Rule repository**.

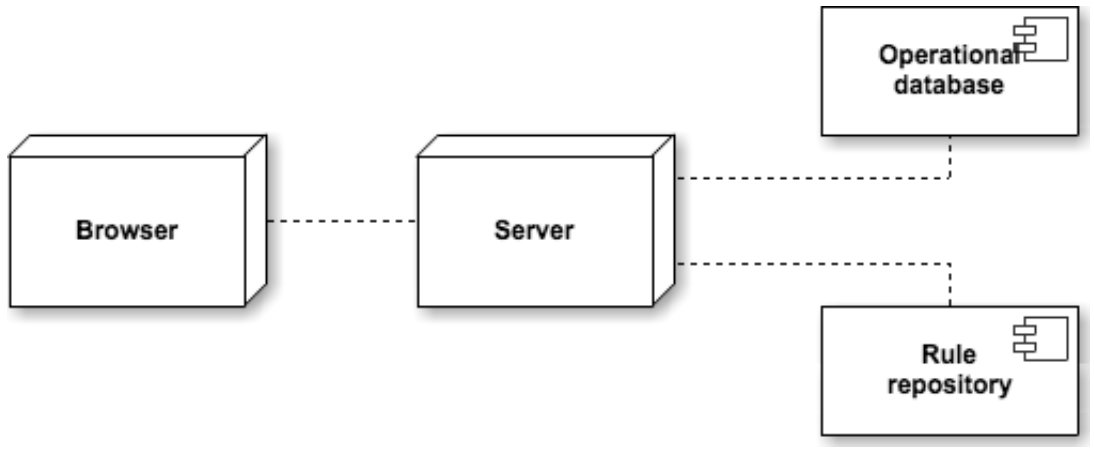

Figura 4.1: Diagrama arquitetural *blackbox*

4.3.1 Browser

O *Browser* é o nó que fica do lado cliente. Ele é executado por um navegador *web* que mantém um *Document Object Model* (DOM) de páginas HTML e executa código JavaScript. Ele é responsável por converter os elementos do modelo conceitual e construir a GUI. Para isso, esse nó é composto pelos seguintes módulos: *Rendering engine*, *Cache*, *Downloaded widgets*, *Downloaded rules*. A Figura 4.2 descreve como os módulos estão interligados.

O *Rendering engine* é o principal módulo do arcabouço. Ele é responsável por carregar os elementos da GUI (regras e *widgets*), povoar a *cache* e coordenar a apresentação da GUI. O DOM é o módulo que representa a página HTML do usuário. O *Rendering engine* pode manipular o DOM em casos especias, tal como na passagem de uma tela para outra. Todavia os *widgets* são os principais componentes que manipulam o DOM para construir a GUI da Aplicação Corporativa.

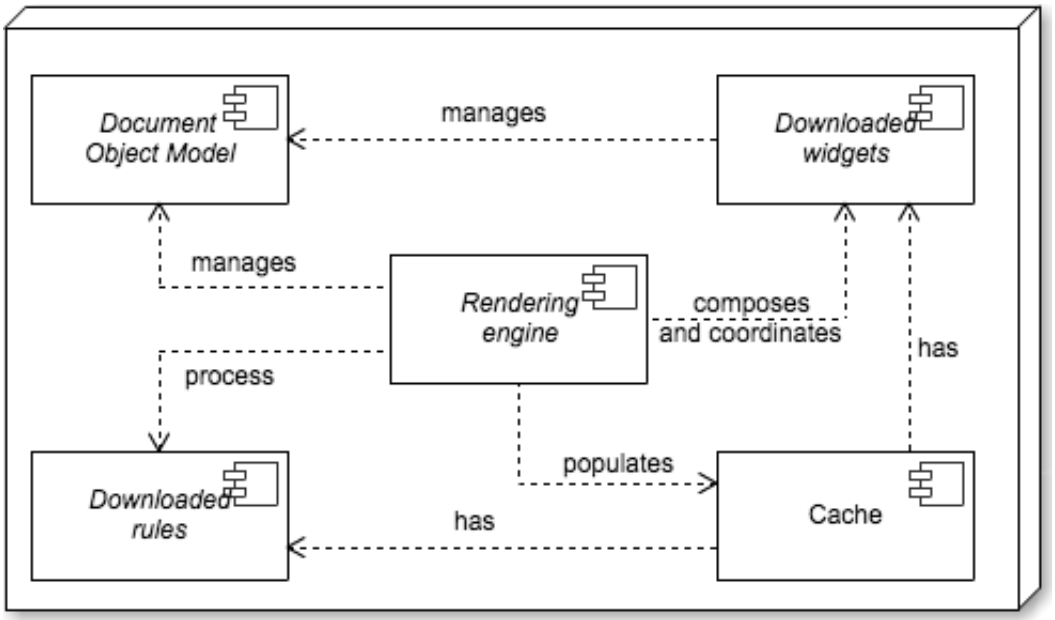

Figura 4.2: Módulos do nó *Browser*

4.3.2 Server

Como em qualquer Aplicação Corporativa comum, o *Server* tem uma API para processar requisições da GUI, que executa as regras de negócio e acessa o *Operational database*. A principal novidade nessa arquitetura é o módulo *Rendering service*, que é responsável por instalar os *widgets* e manter as regras de GUI. Os *widgets* instalados são armazenados internamente, enquanto que os dados do modelo conceitual e as regras são persistidos em componentes externos, o *Operational database* e o *Rule repository* respectivamente. A Figura 4.3 descreve como esses módulos se relacionam.

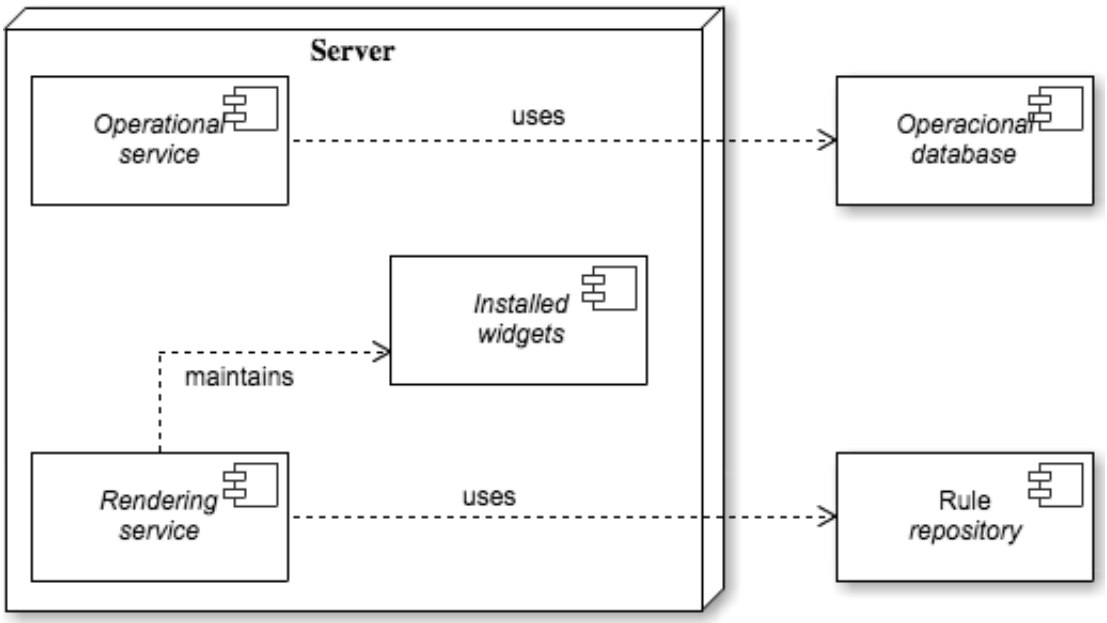

Figura 4.3: Módulos do nó *Server*

4.3.3 Operational database e Rule repository

O *Operational database* e o *Rule repository* são responsáveis por armazenar os dados do modelo conceitual e das regras de GUI respectivamente. O *Rule repository* pode ser um banco de dados ou até um arquivo, enquanto que o *Operational database* é um banco de dados modelado para as entidades de domínio. Esses nós são independentes de tecnologia e podem estar hospedados remotamente, entretanto é necessário configurar o *Server* para que ele possa ter acesso a esses nós.

4.4 Projeto Detalhado

Um arcabouço foi construído baseado na arquitetura descrita na seção anterior. Os módulos do *Browser* foram implementados com CoffeeScript e compiladas para JavaScript; o *Server* foi implementado usando a plataforma Spring Boot; o *Operational database* e o *Rule repository* foram definidos automaticamente pelo Spring Boot. Mais detalhes dessas tecnologias podem ser encontradas na Seção 4.5.

A Figura 4.4 apresenta a arquitetura do arcabouço, seus componentes internos e suas interligações. Nesta seção são apresentados detalhes do projeto do arcabouço e a mecânica dos módulos para prover o seu funcionamento.

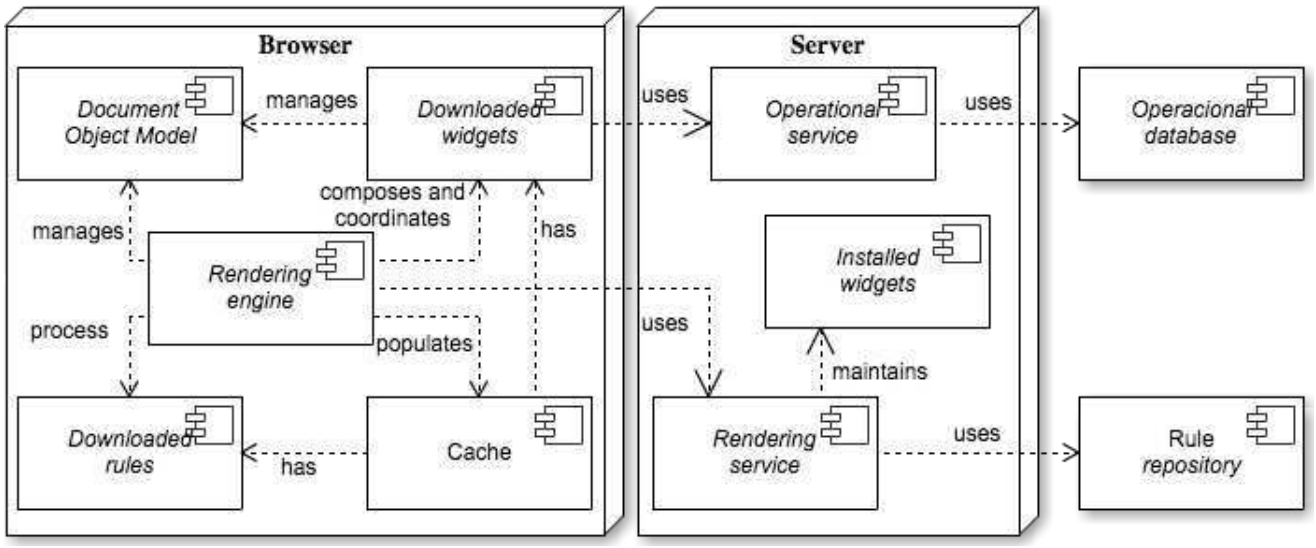

Figura 4.4: Diagrama arquitetural com os componentes internos do arcabouço

4.4.1 Estrutura

O diagrama de classes apresentado na Figura 4.5 descreve os elementos do arcabouço e como eles se relacionam. Mais detalhes são apresentados a seguir:

- **Rendering engine**. O *Rendering engine* é o principal componente do arcabouço, ele é responsável por processar as *Rules*, recuperar os *WidgetsSpecification* e interpretá-los para criar e renderizar *Widgets*.

- **Widgets**. Os *Widgets* possuem métodos para executar a renderização, validar os dados submetidos e converter os dados em um JSON para serem enviados ao *Operational service*.

- **Rules**. Uma *Rule* é composta por um *Scope* e associada a um *Context* e um *WidgetSpecification*. Assim é possível referenciar um *widget* a partir do contexto e do escopo de uma regra.

- **Scope**. O *Scope* é uma representação dos elementos do modelo conceitual para o qual a sua *Rule* está atribuída.

- **Context**. O *Context* define uma situação onde um *widget pai* delega alguma responsabilidade para outro *widget filho*. Para cada contexto, deve haver regras que proveja *widgets* de tal maneira que satisfaça todos os escopos.

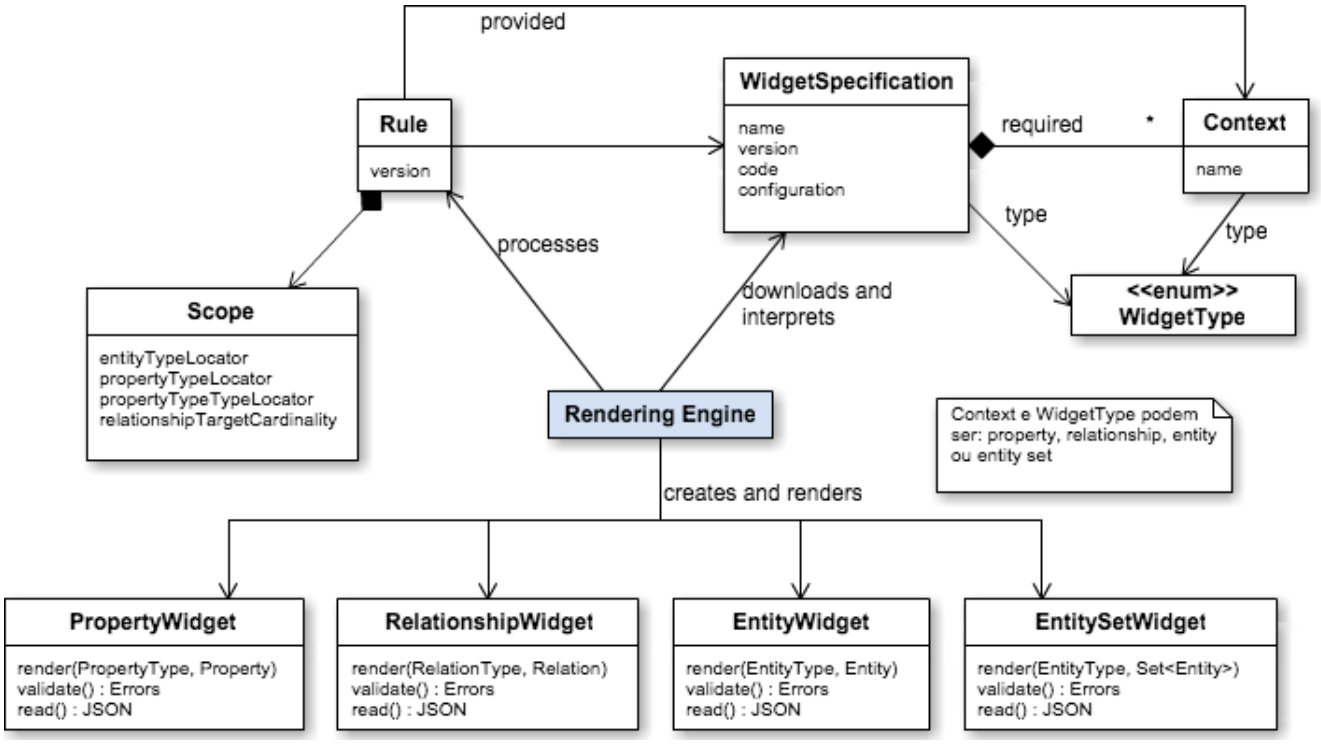

Figura 4.5: Diagrama de classe dos elementos do arcabouço

- **WidgetSpecification**. O *WidgetSpecification* é uma descrição de um *widget*, onde é definido o nome, versão, o código-fonte e os contextos necessários para completar a renderização.

4.4.2 Mecânica

Com relação à mecânica do arcabouço, o fluxo do sistema inicia quando o usuário acessa a aplicação através do navegador *web*. Logo, o *Rendering engine* é carregado e executado, que por sua vez recupera todas as regras de GUI, carrega e executa o RootRenderer, que é um *widget* especial responsável por apresentar todos os *EntityTypes* (ver Seção 2.3). O RootRenderer carrega os EntityType do *Rendering Service* e manipula o DOM para apresentá-los na página. A Figura 4.6 mostra um diagrama de sequência que descreve esse fluxo.

Quando o usuário clica em um EntityType, o *Browser* envia um evento para o *RootRenderer*. Esse solicita ao *RenderingEngine* que procure pelo *widget* de listagem (contexto="listing") para aquela *EntityType* e o execute. O *RederingEngine* processa as regras e descobre qual o *widget* deve ser usado para aquela *EntityType* naquele contexto. Depois,

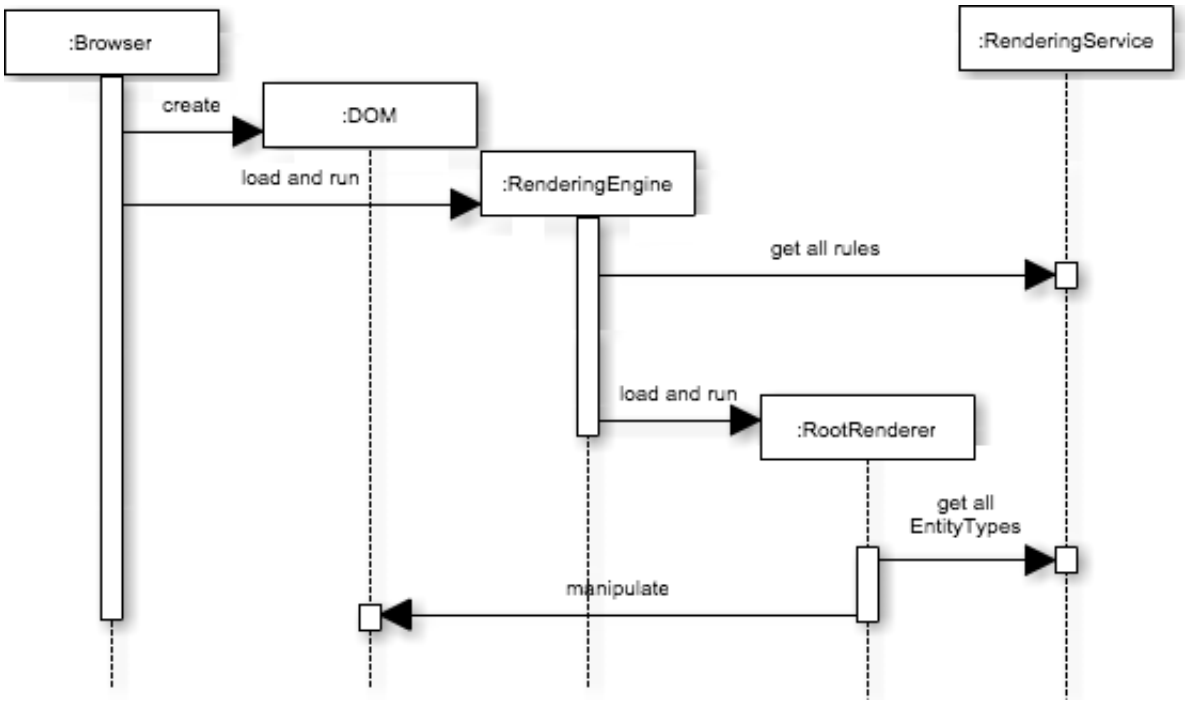

Figura 4.6: Diagrama de sequência do fluxo inicial do arcabouço

o *RenderingEngine* recupera o *widget* do *RederingService*, caso ele não esteja disponível no cache, e em seguida limpa a tela e executa o *widget*. Por sua vez, o *widget* carrega todos os *Entity* daquele *EntityType* e manipula o DOM para apresentar uma lista de *Entity*. A Figura 4.7 apresenta um diagrama de sequência que descreve esse fluxo.

Opcionalmente um *widget* pode delegar parte da construção da GUI para outros *widgets*. Para isso, o desenvolvedor precisa declarar os contextos requeridos pelo *widget* no momento de sua instalação no sistema e definir regras para esses contextos, assim quando esse *widget* estiver executando, ele poderá solicitar ao *RenderingEngine* que procure por *widgets* para esses contextos. No diagrama da Figura 4.7 há um exemplo disso. Após o *EntitySetWidget* manipular o DOM, ele solicita ao *RenderingEngine* que procure um *widget* de listagem para uma propriedade (contexto="propertyListing"), logo o *widget* é recuperado e executado, que por sua vez manipula o DOM. Esse fluxo será executado para cada propriedade do *EntityType*.

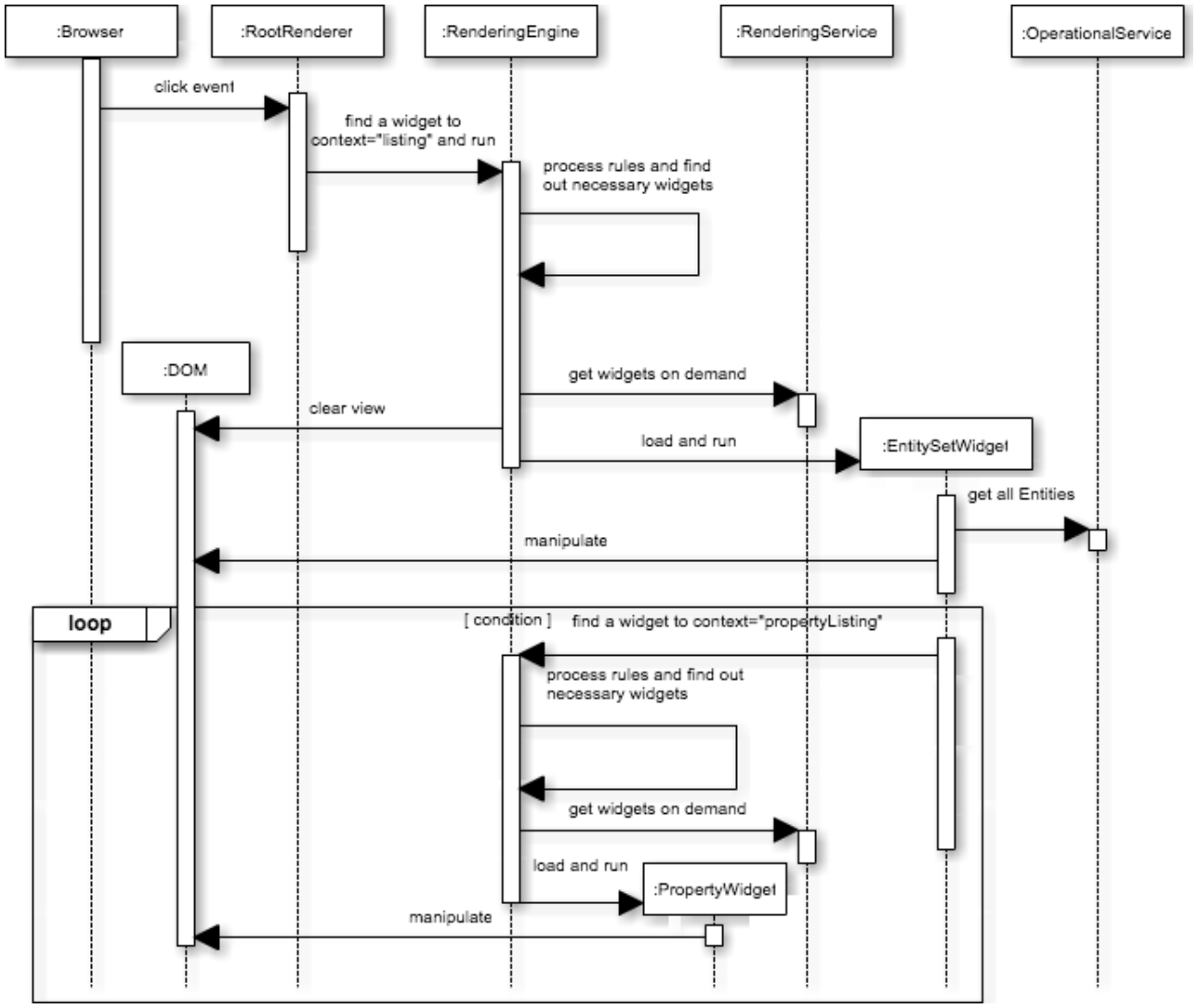

Figura 4.7: Diagrama de sequência do fluxo de um *widget*

4.5 Implementação

Uma AC web moderna, na maioria das vezes, é composta por duas partes: *front-end* e *back-end*. Essa composição permite modularizar a aplicação com o intuito de separar o domínio de negócio da GUI.

De forma análoga, o arcabouço foi dividido em dois artefatos: *meta-gui-web* e *meta-gui-server*. Os códigos fontes estão disponíveis, respectivamente, em https://github.com/delanohelio/meta-gui-web e https://github.com/rodrigovilar/meta-gui-server.

4.5.1 Meta-gui-server

O propósito deste artefato é permitir que o desenvolvedor defina os elementos do modelo conceitual e da GUI para que sejam disponibilizados por uma API sob o protocolo HTTP usando REST. Esses elementos são armazenados em banco de dados, repositório ou arquivo. Esse artefato foi dividido em três camadas: domínio, *controller* e persistência.

A plataforma Spring Boot foi usada para implementar este artefato pois ela promove uma rápida prototipagem, convenção sobre configuração e requisitos não funcionais pré-configurados.

Camada de Domínio

Essa camada é responsável pela lógica de negócio da aplicação, onde o desenvolvedor deve especificar o modelo de negócio. Para isso, foi implementado um *container* que permite publicar as classes do modelo conceitual, a fim de que o arcabouço possa conhecer os seus metadados.

No entanto, algumas informações do modelo conceitual não estão explícitas nos metadados de uma classe, como por exemplo informações sobre a persistência do modelo. Nesse sentindo, foi adotado o uso de anotações para complementar as informações do modelo conceitual. Algumas anotações de JPA são usadas para especificar dados referentes à persistência. Além disso, outras anotações foram implementadas para explicitar informações necessárias para o arcabouço.

A Listagem 4.3 apresenta um trecho de código de uma classe anotada que poderia ser submetida para o *container*. Algumas anotações de JPA foram usadas neste exemplo. Na linha 1, a anotação "@Entity" é usada para determinar que essa classe representa uma entidade a ser persistida. Isso obriga o desenvolvedor a usar a anotação "@Id" para definir qual propriedade representa a chave primária no banco de dados. Na linha 6 a anotação "@GeneratedValue" define a estratégia para gerar a chave primária. Por fim, na linha 12, a anotação "@OneToMany" é usada para especificar o tipo de relacionamento e, se ele for bidirecional, qual a propriedade da outra classe que representa esse relacionamento. Outros tipos de anotações JPA podem ser usados para especificar mais detalhes de persistência.

Apesar das anotações JPA expandirem os metadados dessa classe, ainda é necessário des-

crever mais informações. A seguir, serão detalhadas as anotações que foram implementadas para complementar esses dados. Na linha 2, a anotação "@EntityType" descreve dados referentes aos serviços operacionais desse domínio, onde o "resource" é o nome usado para acessar a API REST oferecida pelo *Operacional service* para esse domínio, e o "repositoryType" é uma referência para uma interface que abstrai o acesso a tabela desse domínio no banco de dados. Na linha 11, a anotação "RelationshipType" define que este relacionamento é do tipo composição.

Código Fonte 4.3: Exemplo de uma classe anotada

```
1  @Entity
2  @EntityType(resource="clients", repositoryType=ClientRepository.class)
3  public class Client {
4    @Id
5    @GeneratedValue
6    private Long id;
7
8    private String name;
9
10   @RelationshipType(composition=true)
11   @OneToMany(mappedBy="client")
12   private List<Dependent> dependents;
13 }
```

Camada de Persistência

A camada de persistência é responsável por armazenar os dados e metadados do modelo conceitual, as regras de renderização, os *widgets* e as suas especificações. Todos esses dados são persistidos em um banco de dados, exceto os *widgets* que são persistidos em arquivos.

O Spring Boot permite modelar e acessar um banco de dados de forma abstrata. Quando se quer criar um modelo no banco de dados, o desenvolvedor cria uma interface referenciando uma classe anotada (conforme a Listagem 4.3) que representa esse modelo. A partir disso, automaticamente o Spring Boot cria o modelo no banco de dados e provê as operações de CRUD através da interface implementada.

A Listagem 4.4 apresenta uma interface implementada que permite criar e acessar o

modelo da classe da Listagem 4.3 no banco de dados.

Código Fonte 4.4: Exemplo de uma classe anotada

```
1 public interface ClientRepository extends JpaRepository<Client, Long> {
2
3 }
```

Camada de *Controller*

A camada de *controller* é responsável por intermediar o acesso à camada de negócio. Ela disponibiliza métodos REST do protocolo HTTP para transferir os elementos do modelo conceitual e da GUI. O *controller* converte os elementos da camada de negócio para o formato JSON ou vice-versa para enviar ou receber dados pelo protocolo HTTP. O Spring Boot permite que o desenvolvedor crie *controllers* usando anotações em classes e métodos.

A Listagem 4.5 apresenta um pedaço de código do *controller* que compõe o *Operational service* implementado para processar as requisições da GUI no modelo conceitual de forma genérica. As anotações das linhas 1 e 2 definem que essa classe é um *controller* e que ela responde as requisições feitas na url "/api/resource", onde *resource* é o nome definido na classe do modelo conceitual. As anotações das linhas 4 e 9 especificam que aquele método responde a um tipo específico de requisição HTTP e possibilita, se necessário, definir uma url para compor a url do controller. Enquanto que as anotações das linhas 5-6 e 10-11 são usadas para que o Spring Boot tente converter automaticamente os dados da requisição em objetos e vice-versa.

Dessa forma, ao inserir as classes no *container*, o desenvolvedor não precisa implementar o seu *controller*, pois o *OperationalController* já vai implementar a API REST daquela classe de domínio.

Código Fonte 4.5: Exemplo de uma classe anotada

```
1 @Controller
2 @RequestMapping(value = "/api/{resource}")
3 public class OperationalController {
4   @RequestMapping(method = RequestMethod.POST)
5   @ResponseBody
```

```
6     public <T> ResponseEntity<T> create(@PathVariable String resource,
          @RequestBody String input) {
7       //...
8     }
9     @RequestMapping(value = "{instanceId}", method = RequestMethod.GET)
10    @ResponseBody
11    public <T> ResponseEntity<T> get(@PathVariable String resource,
          @PathVariable Long instanceId) {
12      //...
13    }
14  }
```

Por sua vez, o *Rendering service* é composto por três *controllers*: um *controller* para disponibilizar os metadados do modelo conceitual; e outros dois *controllers* para consultar e alterar os widgets e as regras de renderização.

4.5.2 Meta-gui-web

O propósito deste artefato é apresentar os elementos do modelo conceitual em páginas da *web* permitindo que o usuário final interaja com eles. Para isso, o *Rendering engine* processa as regras de renderização com o propósito de combinar os *widgets* para formar as telas, de forma análoga ao que o padrão de projeto Composite [9] permite fazer em software orientado a objetos.

Diante do exposto, é necessário que a linguagem de programação de desenvolvimento deste artefato possua características de linguagens orientada a objetos para que permita a aplicação dos padrões de projeto.

JavaScript é uma linguagem de programação dinâmica baseada em protótipos, que é amplamente usada no desenvolvimento de GUI para web, e que permite controlar o navegador, realizar comunicação assíncrona e alterar o conteúdo do documento exibido. Apesar dela usar protótipos em vez de classes, é possível simular muitas características de orientação a objetos baseada em classes com protótipos.

No entanto, o modelo orientado a objetos do JavaScript é controverso e amplamente criticado [2]. Por essa razão, linguagens alternativas têm ganhado espaço na indústria, onde muitas delas geram código JavaScript. Uma dessas linguagens é CoffeeScript, que adici-

ona elementos de sintaxe inspirados no Ruby e Python para aprimorar a leitura e concisão do JavaScript, agregando características mais similares a de linguagens orientada a objetos. Portanto, a linguagem CoffeeScript[1] foi escolhida para implementar este artefato.

Para simplificar o desenvolvimento deste artefato e facilitar a manipulação do DOM, a biblioteca jQuery [4] foi amplamente usada nos *widgets* e na *Rendering Engine*.

Widgets

Os *widgets* são responsáveis por manipular o DOM para construir uma página web ou parte dela. Cada tipo de *widget* tem um escopo específico que define o tipo de elemento conceitual que ele referencia. A Listagem 4.6 apresenta um PropertyWidget que renderiza datas formatadas. É possível perceber que sua sintaxe é muito similar à de linguagens orientada a objetos. Por exemplo, nas linhas 1 e 3 são definidos respectivamente uma classe que herda de uma superclasse e um método. É na linha 7 que a data é formatada e impressa na *view*, que é uma parte da página neste caso.

Código Fonte 4.6: Exemplo de um PropertyWidget

```
1 class DateFormatterWidget extends PropertyWidget
2
3   render: (view) ->
4     format = "yy-mm-dd"
5     if(@configuration)
6       format = @configuration.format
7     view.append $.datepicker.formatDate(format, new Date(@property))
8
9 return new DateFormatterWidget
```

A Listagem 4.7 apresenta trechos de código de um EntityWidget que renderiza uma página de formulário. Ele delega parte de sua responsabilidade para outros *widgets*. Na linha 7 é solicitado ao *Renderer engine* um *widget* para renderizar o campo (o contexto "field") de uma propriedade específica da entidade a que se refere esse formulário.

Código Fonte 4.7: Exemplo de um EntityWidget

```
1 class SimpleFormWidget extends EntityWidget
```

[1] http://coffeescript.org

```
2
3     render: (view) ->
4       #...
5       entityType.propertiesType.forEach (propertyType) ->
6         td  = $("<td>");
7         widget = RenderingEngine.getPropertyWidget 'field', entityType,
              propertyType
8         widget.propertyType = propertyType
9         widget.render td
10      #...
11
12  return new SimpleFormWidget
```

Engine

A *Engine* é responsável por carregar os elementos da GUI do *back-end*, processar as regras de renderização, coordenar os *widgets* e auxiliá-los no acesso aos serviços operacionais do *back-end*. Para melhor organizar as responsabilidades da *engine*, ela foi modularizada nos seguintes *scripts*: *RenderingEngine*, *RulesManager*, *WidgetManager*, *DataManager* e *View*.

Os detalhes de cada *script* são apresentadas a seguir:

- **RenderingEngine**. É o principal *script* da *engine* e o ponto inicial da GUI. Ele é responsável por disponibilizar uma API para auxiliar os *widgets* no processo de renderização.

- **RulesManager**. É o *script* responsável por gerenciar as regras de renderização e processá-las. Ele recupera as regras do *back-end*, os armazena no *localStorage* do navegador e disponibiliza uma função, que é usada pelo *RenderingEngine*, para recuperar a regra mais adequada aos elementos do modelo conceitual passado no argumento.

- **WidgetManager**. É o *script* responsável por gerenciar os *widgets*. De forma análoga ao *RulesManager*, esse *script* carrega os *widgets* do *back-end*, os armazena no *localStorage* do navegador e disponibiliza uma função para recuperar um *widget* a partir de seu nome e versão.

- **DataManager**. É o *script* responsável por gerenciar o acesso aos dados e metadados do modelo conceitual.

- **View**. É o *script* responsável por auxiliar o *RenderingEngine* a preparar uma página da *web* para ser manipulada pelos *widgets*.

A Listagem 4.8 apresenta trechos de código do *RenderingEngine* referentes ao ponto inicial da GUI. A execução do *script* começa nas linhas 8-11, onde as regras e os *widgets* são carregados, a página web é preparada e logo depois é executada a função para apresentar a tela inicial.

Código Fonte 4.8: Trechos de código do *RenderingEngine*

```
1  window.RenderingEngine = {}
2
3  RenderingEngine.openApp = (view) ->
4    WidgetManager.getRootRenderer (rootRenderer) =>
5      DataManager.getAllEntitiesTypes (allEntitiesTypes) =>
6        rootRenderer.render view, allEntitiesTypes
7
8  $ ->
9    RulesManager.downloadAllRules()
10   WidgetManager.downloadAllWidgets()
11   RenderingEngine.openApp View.emptyPage()
```

Capítulo 5

Validação

Neste capítulo apresenta-se um estudo de caso para validar o arcabouço proposto no Capítulo 4. Neste estudo de caso, compara-se a duplicação de código gerada pelo arcabouço e por uma plataforma *Scaffold*, no desenvolvimento de uma aplicação corporativa.

5.1 Projeto do Estudo de Caso

Este estudo de caso consiste no desenvolvimento de dois projetos "irmãos" (equivalentes) de uma parte de um sistema de comércio eletrônico. Um projeto é desenvolvido com o arcabouço proposto e o outro com uma plataforma *Scaffold*. Depois, é realizada uma análise no código fonte desses projetos para comparar a quantidade de duplicações de código que existem em cada projeto. Com essa análise pretende-se responder a seguinte questão de pesquisa:

- **Questão de pesquisa:** *A solução proposta é capaz de reduzir quanto da duplicação de código de GUI dos arcabouços de Scaffold?*

Duas hipóteses foram formuladas para responder a questão supracitada, são elas:

- H0: *O arcabouço proposto não reduz a duplicação de código no desenvolvimento de aplicações corporativas em relação a uma plataforma Scaffold*
- H1: *O arcabouço construído reduz a duplicação de código no desenvolvimento de aplicações corporativas em relação a uma plataforma Scaffold*

Para testar essas hipóteses, será calculada a proporção de linhas de código duplicados dos projetos desenvolvidos com o arcabouço proposto em relação ao do projeto de uma plataforma *Scaffold*. Essa proporção é determinada através da divisão da quantidade de linhas de código duplicados pela quantidade total de linhas de código. Para extrair essa métrica, será usada uma ferramenta de análise estática de código.

5.1.1 Plataforma Scaffold

A plataforma de *Scaffold* escolhida para realizar este estudo de caso foi Spring Roo [17]. Essa tecnologia provê um *Shell* para o desenvolvedor, com o qual ele pode utlizar comandos intuitivos para incluir e configurar bibliotecas, tais quais: JPA para persitência de dados, Spring MVC para implementar a *View*, Spring Security para autenticação e autorização, Log4j para auditoria, JUnit para testes de unidade, Selenium para testes de interface gráfica web, JMS para serviço de mensagens, etc.

Através do *Shell*, o desenvolvedor pode usar comandos para descrever um modelo de domínio, que será utilizado pelo Spring Roo para gerar código Java e JSP para as camadas de apresentação e persistência. Spring Roo também gera código orientado a aspectos em AspectJ, com declarações *inter-type*, também conhecidas como *mixins*. Este modelo segue o conceito de *separation of concerns*, que basicamente separa o código de *Scaffold* (isolando o em aspectos) e o código do modelo do domínio. Quando é necessário alterar o código gerado automaticamente pelo *Scaffold*, seu respectivo método deve ser movido do aspecto para uma classe Java e alterado normalmente.

Uma vez que todo o código gerado pelo *Scaffold* pode ser alterado, esse estudo de caso moveu todo o codigo dos aspectos para as classes Java, sem alterá-lo, e realizou a análise apenas no código Java.

5.1.2 Ferramenta de análise estática

Para extrair a quantidade de duplicação que existe no código-fonte de um sistema foi utilizado a ferramenta PMD[1]. Ela é amplamente usada no desenvolvimento de software para analisar o código fonte Java e procurar por possíveis problemas, tais como: possíveis *bugs*,

[1]Disponível em: http://pmd.sourceforge.net/

código que nunca é executado, código não otimizado, expressões muito complexas e **duplicação de código**.

O PMD disponibiliza um subproduto, o CPD, para fazer análise apenas da duplicação de código. Ele usa um algoritmo de *string matching*, o Karpin-Rabin [14], que compara os valores *hash* dos *tokens* utilizados no projeto. A seguir, são descritas as entradas e saídas do CPD:

- **Entradas**:
  - **minimum-tokens (obrigatório)**: o tamanho mínimo do *token* que deve ser reportado como duplicado. Um *token* para a ferramenta pode ser um símbolo léxico ou um conjunto de símbolos léxicos seguidos.
  - **ignore-literals (opcional)**: quando ativado ignora os valores de números e o conteúdo de *strings* na comparação dos textos.
  - **ignore-identifiers (opcional)**: quando ativado ignora o nome das constantes e variáveis na comparação dos textos.
  - **ignore-annotations (opcional)**: quando ativado ignora as anotações da linguagem na comparação dos textos.
- **Saída**. Como saída é gerado um relatório com a quantidade de *tokens* e linhas que foram duplicadas e os locais onde eles foram usados. É possível exportar esse relátorio nos seguintes formatos: texto, xml e csv.

5.1.3 Projeto Alvo

A Figura 5.1 apresenta o modelo conceitual de parte de um sistema de comércio eletrônico desenvolvido para este estudo de caso. Ela descreve um sistema onde clientes fazem pedidos de produtos e o vendedor os lança no sistema.

5.2 Resultados

Os projetos gerados na execução desse estudo de caso foram os seguintes:

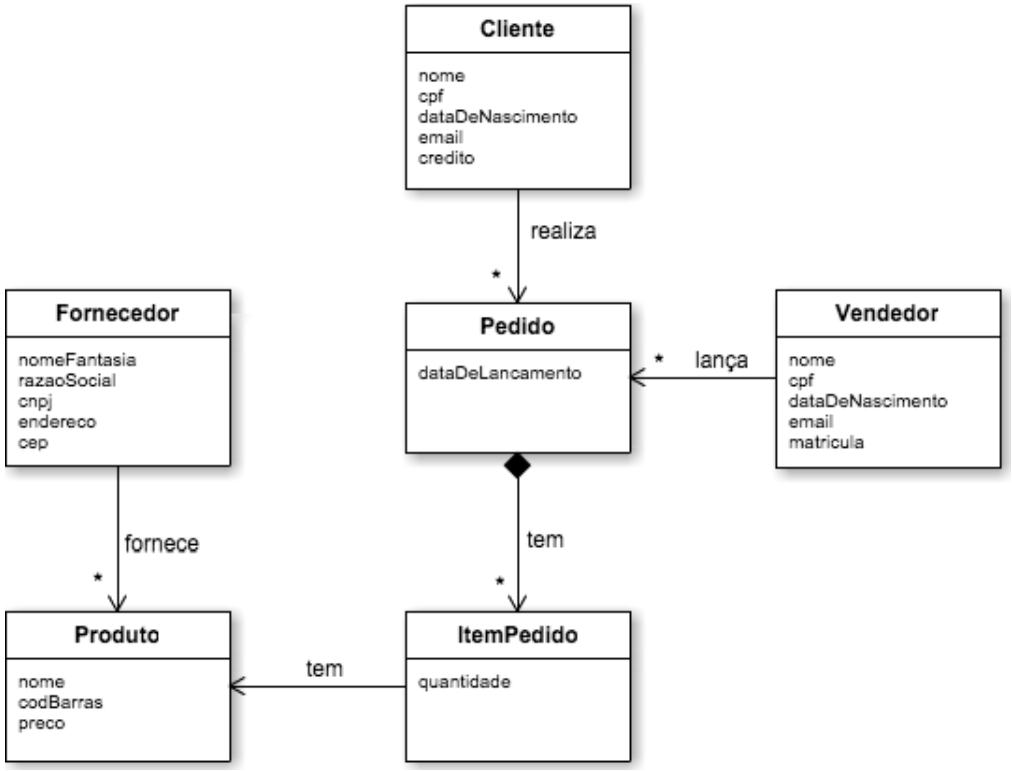

Figura 5.1: Diagrama de classes de parte de um sistema de comércio eletrônico

- **Meta-GUI (Arcabouço Proposto)**:
  https://github.com/delanohelio/ComercioEletronicoMetaGUI

- **Spring Roo**: https://github.com/rodrigovilar/ComercioEletronicoSpringRoo

O projeto do Meta-GUI é composto pelas classes do modelo conceitual e *widgets*. Enquanto que o projeto do Spring Roo é composto pelas classes do modelo conceitual, as classes do *controllers* e os arquivos JSP da *view*.

Na análise realizada, as classes do modelo conceitual foram ignoradas pois elas não estão inseridas no contexto da GUI. Desse modo, foram analisados os *controllers* e as *views* desses projetos. É importante resaltar que os *controllers* do Meta-GUI são interpretados automaticamente em tempo de execução pelo *Operational service*, ou seja, não há código fonte de *controller* no Meta-GUI para ser analisado.

Para este estudo de caso, foram definidos quatro tipos de configurações dos parâmetros do CPD. A Tabela 5.1 descreve essas configurações. Por padrão a ferramenta define o parâmetro *minimum-tokens* em 75, entretanto não foi possível encontrar duplicações de código no Meta-GUI com esse número. Assim, foi feito uma avaliação no código para identificar

a quantidade média de tokens que representa uma ou duas linhas de código em ambos os projetos, como resultado foi encontrado o valor de 8 tokens. Esse valor foi adotado neste estudo de caso.

Tabela 5.1: Tabela com os tipos de configurações para execução do CPD

Conf.	**minimum-tokens**	**ignore-literals**	**ignore-identifiers**	**ignore-annotations**
C1	8	desativado	desativado	desativado
C2	8	ativado	desativado	desativado
C3	8	ativado	ativado	desativado
C4	8	ativado	ativado	ativado

A Tabela 5.2 apresenta os resultados por camada e configuração obtidos através do uso da ferramenta CPD. É importante destacar que a quantidade de linhas duplicadas pode ser maior que o total de linhas pois primeiro o CPD compara um bloco de linhas de código para verificar se esse bloco foi duplicado no projeto, e depois compara cada linha de código desse bloco separadamente. Desse modo uma linha de código pode ser considerada duplicada mais de uma vez, uma como parte de um bloco e outra como uma linha independente.

Não foi possível executar a análise nas configurações C2, C3 e C4 para as *views* dos projetos, pois a ferramenta não permite variar os outros parâmetros na análise de código JSP e JavaScript. A Figura 5.2 apresenta os dados da Tabela 5.2 em um gráfico de barras.

A Tabela 5.3 agrupa os resultados da configuração C1 por projeto. Também foi calculada a proporção de linhas duplicadas em relação ao total de linhas de cada projeto. A Figura 5.3 apresenta os dados da Tabela 5.3 em um gráfico de barras.

Tabela 5.2: Tabela de resultados da duplicação de código por camada e configuração

Projeto	**Camada**	**Configuração**	**Linhas duplicadas**	**Total de linhas**
Spring Roo	Controller	SR Controllers C1	3851	2961
		SR Controllers C2	4274	2961
		SR Controllers C3	8816	2961
		SR Controllers C4	7725	2961
	View	SR View C1	2259	479
Meta-GUI	Widgets	MG View C1	16	424

Tabela 5.3: Tabela de resultados da duplicação por projeto para a C1

Projeto	**Linhas duplicadas (LD)**	**Total de linhas (TL)**	**Proporção (LD/TL)**
Spring Roo	6110	3440	1,776
Meta-GUI	16	424	0,037

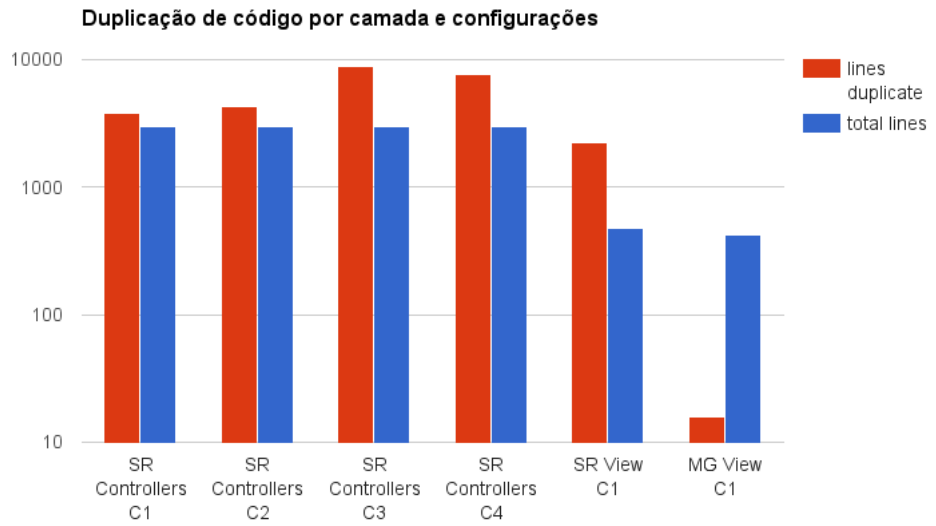

Figura 5.2: Gráfico de barras com os resultados da duplicação de código por camada e configurações

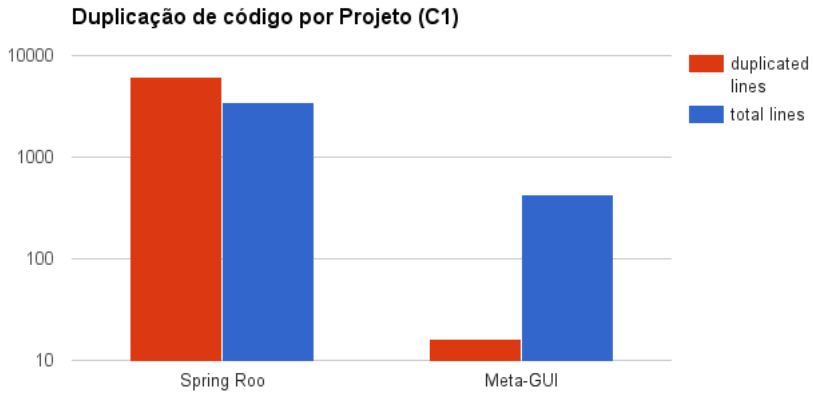

Figura 5.3: Gráfico de barras com os resultados da duplicação de código da C1 por projeto

5.3 Análise

Na Tabela 5.2 é possível identificar uma ampla diferença dos resultados da C3 para as outras configurações, principalmente em relação às duplicações de *tokens*. Isso indica que há muita duplicação de código nos *controllers* gerados por uma plataforma *Scaffold* quando não são considerados os nomes e valores de literais, variáveis e constantes. Esse tipo de configuração é equivalente à forma que foi analisada no Capítulo 3. Apesar disso, a comparação entre os projetos foi feita com os resultados da C1, devido às limitações da ferramenta de análise sintática.

O gráfico da Figura 5.3 apresenta a comparação por projetos. Nesse gráfico é visível a disparidade que há ao comparar a proporção de linhas duplicadas no Spring Roo com o Meta-GUI. Por sua vez, para responder a questão de pesquisa é analisado o valor da Proporção de linhas duplicadas na Tabela 5.3. Para o Spring Roo a proporção é maior que um, enquanto que para o Meta-GUI a proporção é próxima de zero, isso indica que a hipótese zero é falsa. A propósito, ao ser analisada as linhas de códigos que estão duplicadas no Meta-GUI notou-se que se trata de linhas de código que é usada sempre que uma classe herda de outra em JavaScript, desse modo pode-se considerar que não há duplicação de código funcional nos *widgets* do Meta-GUI. De fato, o arcabouço construído reduz a duplicação de código de GUI no desenvolvimento de AC em relação a uma plataforma *Scaffold*.

5.4 Ameaças à validade do Estudo de Caso

Foram identificadas as seguintes ameaças à validade deste estudo de caso:

- O Meta-GUI foi comparado com apenas uma plataforma *Scaffold*, que é uma amostra pequena e pode não refletir na população alvo;
- O autor deste trabalho realizou este estudo de caso, que representa uma amostra pequena dos desenvolvedores e sua experiência com o desenvolvimento do arcabouço pode ter influenciado os resultados.
- Este estudo de caso foi feito em um laboratório, que não representa um ambiente real de desenvolvimento.

- As limitações da ferramenta de análise sintática não permitiram realizar uma análise com todos os critérios definidos neste estudo de caso.

Capítulo 6

Conclusão

A primeira contribuição deste trabalho foi a identificação de quatro cenários que evidenciam os problemas no desenvolvimento de GUI para aplicações corporativas com plataformas *Scaffold*. Nesses cenários foram demonstradas situações em que as plataformas *Scaffold* não conseguem gerar os requisitos da GUI por completo, induzindo o desenvolvedor a alterar o código fonte gerado pela plataforma, que possui muita duplicação de código.

Outras abordagens foram estudadas. Dentre elas, os Padrões de Renderização se mostraram promissores quanto ao problema de duplicação de código de GUI. Foram estudados os detalhes desses padrões e ficou evidenciado que eles direcionam o programador a criar elementos de GUI com responsabilidades bem definidas. No entanto, não foram encontrados na literatura nem na indústria fundamentos que mostrem de fato a eficácia desses padrões para reduzir a duplicação de código de GUI.

Diante do exposto, foi proposto e implementado um arcabouço baseado nos Padrões de Renderização para reduzir a duplicação de código de GUI em aplicações corporativas modernas para web. Esse arcabouço permite fatorar os *widgets* através do uso dos metadados do modelo conceitual, e definir regras de renderização para conectar os *widgets* aos elementos do modelo conceitual. Além disso, foram apresentados os detalhes da arquitetura do arcabouço, dos componentes internos e da sua mecânica.

Este trabalho foi validado a partir de um estudo de caso que avaliou a duplicação de código de GUI gerada pelo arcabouço proposto (Meta-GUI) e por uma plataforma *Scaffold* (Spring Roo) no desenvolvimento de uma aplicação corporativa. Nesse estudo de caso, um pequeno projeto foi definido e desenvolvido pelas duas abordagens. Após isso, foi extraída a

quantidade de duplicação de códigos de GUI dos projetos usando uma ferramenta de análise estática.

Na análise dos resultados, foi demonstrada uma redução na quantidade de duplicações de código de GUI no projeto desenvolvido com o Meta-GUI, em relação ao mesmo projeto desenvolvido com Spring Roo. Desse modo foi concluído de fato que o Meta-GUI reduz a duplicação de código de GUI no desenvolvimento de AC em relação às plataformas Scaffold.

Como trabalhos futuros são propostos: (i) realizar um estudo de caso em um ambiente real que permita avaliar se o uso desse arcabouço aumenta a produtividade de desenvolvimento e manuntenção, (ii) realizar uma avaliação empírica para identificar o impacto na produtividade do desenvolvedor, (iii) realizar um experimento que avalie o desempenho do acabouço, (iv) construir uma ferramenta que permita adicionar novos *widget* e criar regras de renderização de forma visual.

Bibliografia

[1] Eli Acherkan, Atzmon Hen-Tov, David H Lorenz, and Lior Schachter. The ink language meta-metamodel for adaptive object-model frameworks:[extended abstract]. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 181–182. ACM, 2011.

[2] Aansa Ali. Evaluation and comparison of alternate programming languages to javascript. In *Proceedings in Research Conference in Technical Disciplines*, volume 1, pages 90–95. EDIS - Publishing Institution of the University of Zilina, 2013.

[3] Scott Boag, Michael Kay, Don Chamberlin, Jerome Simeon, Mary Fernandez, Jonathan Robie, and Anders Berglund. XML path language (XPath) 2.0 (second edition). W3C recommendation, W3C, December 2010. http://www.w3.org/TR/2010/REC-xpath20-20101214/.

[4] Jonathan Chaffer and Karl Swedberg. *Learning jQuery Fourth Edition*. Packt Publishing, 2013.

[5] Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Design for an adaptive object-model framework. In *Proceedings of the 4th Workshop on Models@ run. time, held at the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*, 2009.

[6] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[7] Jeff Forcier, Paul Bissex, and Wesley Chun. *Python Web Development with Django*. Addison-Wesley Professional, 2008.

[8] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[10] Jesse J Garrett. Ajax: A New Approach to Web Applications, 2005.

[11] Eduardo Guerra, Clovis Fernandes, and Fábio Fagundes Silveira. Architectural patterns for metadata-based frameworks usage. In *Proceedings of the 17th Conference on Pattern Languages of Programs*, PLOP '10, pages 4:1–4:25, New York, NY, USA, 2010. ACM.

[12] Jack Herrington. *Code Generation in Action*. Manning Publications, 2003.

[13] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

[14] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[15] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall, 2004.

[16] Peter Ledbrook and Glen Smith. *Grails in Action*. Manning Publications, 2014.

[17] Ken Rimple and Srini Penchikala. *Spring Roo in Action*. Manning Publications, 2012.

[18] Sam Ruby, Dave Thomas, and David Heinemeier Hansson. *Agile Web Development with Rails 4 (Facets of Ruby)*. Pragmatic Bookshelf, 2013.

[19] Ralph R Swick and Mark S Ackerman. The x toolkit: More bricks for building user-interfaces or widgets for hire. In *Usenix Winter*, pages 221–228. Citeseer, 1988.

[20] León Welicki, Joseph W Yoder, and Rebecca Wirfs-Brock. Rendering patterns for adaptive object-models. In *Proceedings of the 14th Conference on Pattern Languages of Programs*. ACM, 2007.

[21] Joseph W Yoder, Federico Balaguer, and Ralph Johnson. Architecture and design of adaptive object-models. *ACM Sigplan Notices*, 36(12):50–60, 2001.

[22] Joseph W Yoder and Ralph Johnson. The adaptive object-model architectural style. In *Software Architecture*, pages 3–27. Springer, 2002.