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

Decomposição e Reúso de Componentes baseados em Metadados para Interfaces Gráficas do Usuário em Aplicações Corporativas Web

Rodrigo de Almeida Vilar de Miranda

Tese submetida ao Programa 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 Doutor em Ciência da Computação.

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

Hyggo Oliveira de Almeida e Angelo Perkusich
(Orientadores)

Campina Grande, Paraíba, Brasil
©Rodrigo de Almeida Vilar de Miranda, 04/09/2017

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

M672d Miranda, Rodrigo de Almeida Vilar de.

Decomposição e reúso de componentes baseados em metadados para interfaces gráficas do usuário em aplicações corporativas web / Rodrigo de Almeida Vilar de Miranda. – Campina Grande, 2017.

177 f. : il. color.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2017.

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

Referências.

1. Componentes – Interface Gráfica. 2. Metadados. 3. Interface Gráfica. 4. Computação – Aplicações Corporativas. I. Almeida, Hyggo Oliveira de. II. Perkusich, Angelo. III. Título.

CDU 004.5(043)

COMPOSIÇÃO E REÚSO DE COMPONENTES BASEADOS EM META DADOS PARA
ITERFACES GRÁFICAS DO USUÁRIO EM APLICAÇÕES CORPORATIVAS WEB"

RODRIGO DE ALMEIDA VILAR DE MIRANDA

TESE APROVADA EM 04/09/2017

HYGGO OLIVEIRA DE ALMEIDA, Dr., UFCG
Orientador(a)

ANGELO PERKUSICH, Dr., UFCG
Orientador(a)

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

TIAGO LIMA MASSONI, Dr., UFCG
Examinador(a)

UIRA KULESZA, Dr., UFRN
Examinador(a)

EDUARDO MARTINS GUERRA, Dr., INPE
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Metadados podem ser manipulados em aplicações corporativas a fim de reduzir o acoplamento entre a camada da interface gráfica do usuário e a camada do modelo do domínio. Dessa forma, os componentes da interface gráfica são substituídos por meta componentes, que podem ser facilmente reutilizados em domínios diferentes, aumentando a produtividade no desenvolvimento desse tipo de aplicação. Todavia, nas abordagens baseadas em metadados existentes, os meta componentes de interface são complexos, entrelaçam responsabilidades em um mesmo artefato e possuem baixa manutenibilidade. Neste trabalho propõe-se uma abordagem que organiza os meta componentes através de artefatos pequenos e com responsabilidades bem definidas, com a finalidade de aumentar o potencial de reúso e facilitar a sua customização. A abordagem proposta foi desenvolvida conceitualmente por meio de uma linguagem de padrões, contendo sete padrões de projeto e dois padrões arquiteturais, que foram catalogados a partir de abordagens existentes, abstraindo as boas práticas e propondo correções para as deficiências. Concretamente a abordagem foi implementada em dois arcabouços de código aberto, *Geneguis* e *Angular M*, para o desenvolvimento de interface gráfica em aplicações corporativas. *Angular M* teve a produtividade comparada com *Ruby on Rails* em um experimento com 15 desenvolvedores de software que atuaram sobre 12 cenários de customização de interface gráfica. Após uma análise da significância estatística dos resultados, foi observado que: nos seis cenários propostos onde havia reúso de componentes, *Angular M* foi mais produtivo em todos; e nos seis cenários onde não havia reúso de componentes, *Angular M* foi mais produtivo em dois, *Ruby on Rails* foi mais produtivo em um e não houve diferença significante nos outros três cenários. O ganho médio de produtividade com *Angular M*, em relação a *Ruby on Rails*, foi 20% nos cenários gerais e 64% ao considerar apenas os cenários com reúso de componentes.

Abstract

Developers can use metadata in enterprise applications in order to reduce coupling between graphical user interface and domain model layers. Therefore, user interface components are replaced by meta components, which can be easily reused in different domains, enhancing productivity of application development. However, the current metadata-based approaches define complex components for graphical interface, mixing responsibilities in an artifact and reducing maintainability. In this work, we propose an approach to implement meta components into small artifacts with encapsulated responsibilities, increasing its reuse and easing its customization. This approach has been developed as a pattern language with seven design patterns and two architectural patterns, which were defined based on existing approaches, abstracting good practices and proposing solutions for weaknesses. Two open source frameworks, *Geneguis* and *Angular M*, were implemented using the pattern language concepts in order to support User interface development for enterprise applications. The productivity of *Angular M* was compared to *Ruby on Rails* in an experiment with 15 software developers that performed 12 user interface customization tasks. The statistical analysis of experiment results has shown that: for the six customization tasks where components were reused, *Angular M* was significantly more productive in all tasks; and for the six tasks without reused components, *Angular M* was significantly more productive in two tasks, *Ruby on Rails* was significantly more productive in one task and the other three tasks did not show significant difference. The average increase of productivity for *Angular M*, in comparison with *Ruby on Rails*, was 20% for all 12 scenarios and 64% for the six scenarios with component reuse.

Agradecimentos

O doutorado tem sido para mim uma experiência transformadora, principalmente por causa das pessoas com quem tenho interagido. Pensava que seria apenas mais uma etapa da minha carreira acadêmica, mas toda minha vida tem mudado.

Agradeço aos meus orientadores, Hyggo Almeida e Angelo Perkusich, pela objetividade e pela capacidade de fazer as coisas acontecerem. Vocês são agentes de transformação social, formando e empregando centenas de pessoas. Que nunca lhes falte força para acordar a cada dia e fazer o que está posto diante de vocês. Muitas vezes me embaracei e não sabia como prosseguir. Poucos minutos de conversa com Hyggo eram suficientes para resolver meus problemas e me dar a direção. Muito obrigado mesmo!

Aos parceiros no mesmo tema de pesquisa, minha profunda gratidão. Como crescemos juntos! O que era um tema novo de pesquisa, sem muita literatura descoberta, hoje está bem mais claro e executa! Espero que nossas ideias cheguem no mercado e mudem um pouco a forma como as pessoas trabalham. Será o nosso sonho realizado. Enquanto estudávamos, Delano terminou o mestrado e passou no concurso do IFPE. Sinval se tornou um desenvolvedor sênior e conversou pessoalmente com Ralph Johnson e Joe Yoder sobre nossos padrões. Anderson foi quem mais trabalhou e publicou comigo e Fernando quem mais fez *pair programming* comigo.

Não posso esquecer de agradecer aos meus colegas do Departamento de Ciências Exatas na UFPB – Campus Rio Tinto, que me cederam o afastamento de longa duração para doutorado, viabilizando a evolução deste trabalho. Em especial, Yuska Paola e Juliana Saraiva, obrigado por assumirem minha carga horária de aulas. E, Ayla Rebouças, obrigado por orientar os alunos que deixei para trás, sempre ler meus textos e apontar onde eu estava errando.

Aos voluntários que participaram do experimento, muito obrigado pelas horas dedicadas. Espero que essa ajuda retorne a vocês. Quem sabe através do próprio *Angular M* ou através de qualquer demanda em que vocês precisem de alguma solução baseada em metadados. Ao Gerente de programa do Virtus, Sérgio Augusto, obrigado por ceder o espaço físico para o experimento e pelas várias dicas que me ajudaram a ter foco no fim deste doutorado.

Estou a caminho de conseguir realizar um sonho da minha mãe, Élvia Almeida, professora de Engenharia Química, que *só* caminhou até o mestrado, mas sempre manteve o sonho de ter um filho doutor. Obrigado, mainha, por fazer das tripas coração e pagar aquele curso de informática, quando eu tinha 13 anos. Naqueles dias, descobri o talento que Deus me deu para a computação. E hoje tenho a profissão na qual mais poderia me realizar. Ao meu pai, Orlando Vilar, a gratidão pela referência de caráter, pelo amor não fingido, pela disponibilidade de sempre. Diogo e Camila, meus irmãos, queridos companheiros, estamos juntos em amor.

Às minhas duas princesas, que mais pagaram o preço pelas noites, madrugadas e fins de semana ausente, pendurado no computador, com a cabeça quente. Está quase acabando! Mariana, esposa dedicada e que amadureceu tanto nesses anos, entendeu minha luta e cresceu comigo. Marina, minha filha amada, que tantas vezes quis digitar no computador para dividir o trabalho comigo, mesmo tendo apenas 5 anos e não sabendo ler. Recebo o amor de vocês e quero recuperar o tempo em que precisei estar ausente. Grande beijo!

Meu último e mais importante agradecimento a Deus. Creio que a ciência não me afasta de ti. Pelo contrário, me habilita a cada vez mais contemplar a beleza da tua criação e redenção. Obrigado por me dar família, amigos, irmãos, habilidades e força. Guia meus passos a cada dia para cumprir o teu propósito.

Conteúdo

1 Introdução **1**

1.1 Problemática 5

1.2 Objetivos 10

1.3 Metodologia 10

1.4 Contribuições 14

1.5 Organização do documento 15

2 Fundamentação teórica **17**

2.1 Arquitetura monolítica 17

2.2 Padrão arquitetural Camadas 20

2.3 Metadados em GUI 22

2.4 Acoplamento entre meta componentes 25

3 Trabalhos relacionados **30**

3.1 Programação orientada a aspectos 30

3.2 Desenvolvimento dirigido por modelos 32

3.3 *Scaffolding* 33

3.4 *Adaptive object model* 35

3.5 Arcabouços baseados em metadados 37

3.6 Considerações finais do capítulo 38

4 Linguagem de padrões - Catálogo **39**

4.1 Visão geral dos padrões 39

4.2 Padrão de projeto DOMAIN WIDGET 41

4.3 Padrão de projeto DOMAIN MODEL X-RAY 46

4.4 Padrão de projeto INDIRECT WIDGET 50
4.5 Padrão de projeto WIDGET ENGINE 58
4.6 Padrão arquitetural SERVER RENDERING 61
4.7 Padrão arquitetural CLIENT RENDERING 63
4.8 Padrão de projeto WIDGET SCOPE 66
4.9 Padrão de projeto PARAMETRIZABLE WIDGET 72
4.10 Padrão de projeto RELATIONSHIP RENDERER 76
4.11 Considerações finais do capítulo 78

5 Linguagem de padrões - Relacionamentos e arquitetura de referência 79
5.1 Pesquisa sobre relacionamentos em linguagens de padrões 80
5.2 Relações entre os padrões 82
5.3 Arquitetura de referência 84
5.4 Mapa de soluções 89

6 Viabilidade técnica da linguagem de padrões 92
6.1 Requisitos dos arcabouços 92
6.2 Visão arquitetural 94
6.3 Detalhes de implementação 100
6.4 Uso dos padrões da linguagem 101
6.5 Exemplos de reúso de GUI 104

7 Avaliação 114
7.1 Design do experimento 114
7.2 Preparação do experimento 118
7.3 Execução do experimento 123
7.4 Análise dos resultados 126
7.5 Análise de ameaças à validade 142
7.6 Avaliação da hipótese do trabalho 144

8 Conclusões e Trabalhos futuros 146

A Visão dos desenvolvedores de software sobre GUI para aplicações corporativas 161

B Projeto de tela complexa utilizando a abordagem proposta **167**

B.1 Estudo de viabilidade da linguagem de padrões 167

B.2 Análise do impacto da linguagem de padrões no acoplamento GUI–Domínio 171

C Script R para análise estatística **175**

Lista de Símbolos

API - *Application Programming Interface*

CRM - *Customer Relationship Management*

CRUD - *Create, Read, Update and Delete*

ERP - *Enterprise Resource Planning*

GQM - *Goal–Question–Metric*

GUI - *Graphical User Interface*

HTML - *Hyper Text Markup Language*

Lista de Figuras

2.1 Marcação de responsabilidades de GUI e Domínio em códigos monolíticos 19
2.2 Marcação de responsabilidades de GUI e Domínio em códigos com camadas 22
2.3 Exemplos reais de componentes de GUI que utilizam metadados 24
2.4 Análise das responsabilidades de metacomponentes reais 26
2.5 Análise das responsabilidades de metacomponentes hipotéticos 29

3.1 Principais conceitos da Programação orientada a aspectos 31
3.2 Exemplos de *property renderers* e *entity views* 35

4.1 *Widgets* e Componentes 44
4.2 Metadados em GUI 44
4.3 Exemplo de uso do padrão DOMAIN MODEL X-RAY 49
4.4 Exemplo de composição de GUI em uma árvore de componentes 52
4.5 Ligando *widgets* indiretamente através de portas 55
4.6 Ligando componentes e portas do tipo `Property` 56
4.7 Arquitetura do padrão CLIENT RENDERING 65
4.8 Aplicando a dimensão escopo nas ligações entre portas e componentes . . . 71
4.9 Usando dados de configuração para parametrizar componentes de GUI . . . 74

5.1 Linguagem de padrões para *widgets* em aplicações corporativas 83
5.2 Linguagem de padrões para *widgets* em aplicações corporativas 85
5.3 Mapa de soluções para a Linguagem de padrões 90

6.1 Diagrama de casos de uso dos arcabouços 93
6.2 Diagrama arquitetural com os principais componentes do *Geneguis* 94
6.3 Diagrama arquitetural com os principais componentes do *Angular M* 95

6.4 Comunicação entre os tipos de *widgets* 97
6.5 Reúso de componentes em tabelas de listagem 105
6.6 Reúso de componentes em formulários de criação 107
6.7 Exemplo de GUI com componente de relacionamento 109
6.8 GUI resultante de componentes baseados em *Bootstrap* 109
6.9 GUI resultante de componentes baseados em *Material* 111

7.1 Histogramas com o perfil dos participantes do experimento 120
7.2 Comparações dos resultados agrupados 128
7.3 Resultados individuais das tarefas do experimento 132

A.1 Perfil do desenvolvedores que responderam à pesquisa 162
A.2 *Ranking* das aplicações corporativas mais relevantes na pesquisa 163
A.3 Influência da GUI no desenvolvimento de aplicações corporativas 166

B.1 Criação de tarefas no *Jira* 168
B.2 Modelo do domínio para criação de tarefas no *Jira* 169
B.3 *Widgets* para criação de tarefas no *Jira* 170
B.4 Dependência entre os componentes de uma aplicação dividida em camadas 171
B.5 Dependência entre os componentes de uma aplicação orientada a recursos . 172
B.6 Dependência entre os componentes da arquitetura proposta nesta tese . . . 173

Lista de Tabelas

4.1 Exemplos de componentes de GUI por tipo de dados manipulados 51
4.2 Dados de configuração para ligar *widgets* através de portas 60
4.3 Dados de configuração para ligar *widgets* do tipo `Property` 60
4.4 Estrutura dos dados para ligações com escopo 70
4.5 Estrutura e exemplos de dados de configuração para montar a Figura 4.9 . . 74

7.1 Telas utilizadas nas tarefas de customização de GUI 117
7.2 Lista de aulas de Ruby on Rails: goo.gl/j5uYHk 121
7.3 Lista de aulas de Angular: goo.gl/Yj0KUS 121
7.4 Lista de aulas de Angular M: goo.gl/pz3swS 122
7.5 Tarefas de customização de GUI para Aplicações corporativas 125

A.1 Análise das ferramentas mais relevantes da pesquisa 164

B.1 Matriz de dependências para uma aplicação dividida em camadas 172
B.2 Matriz de dependências para uma aplicação orientada a recursos 173
B.3 Matriz de dependências para os componentes da arquitetura proposta nesta tese 174

Lista de Códigos Fonte

1.1 Processo de construção de GUI a partir de um *widget* 9
2.1 Código monolítico com GUI e Domínio de negócio 18
2.2 Código com GUI e Domínio de negócios separados em camadas 21
2.3 Metacomponentes de GUI com responsabilidades bem definidas 25
4.1 Exemplos de componentes de GUI simples 41
4.2 Exemplos de componentes de GUI complexos 42
4.3 Exemplo de um metacomponentes de GUI complexo 43
4.4 Exemplo de componente com obtenção de metadados 46
4.5 Exemplo de componentes ligados diretamente 51
4.6 Exemplo de componentes indiretos 56
4.7 Exemplo de compositor com referências diretas 58
4.8 Desenhando caixas de texto baseado em metadados 67
4.9 Exemplo de *widgets* muito parecidos 73
4.10 Exemplo de *widget* parametrizável 75
6.1 Exemplos de componentes *Geneguis* `EntityTypeSet` e `EntityType` . 97
6.2 Ex. de componentes *Angular M* `EntityTypeSet` e `EntityType` . . . 99
6.3 Componentes *Geneguis* formando uma tabela de listagem 105
6.4 Componentes *Angular M* formando uma tabela de listagem 106
6.5 Componentes *Geneguis* formando um formulário de criação 107
6.6 Componentes *Angular M* formando um formulário de criação 108
6.7 Componente que usa um relacionamento para povoar a tabela de listagem . 108
6.8 Melhoria do *design* da GUI com componentes baseados em *Bootstrap* . . . 110
6.9 Melhoria do *design* da GUI com componentes baseados em *Material* 112
C.1 Script R utilizado para analisar os resultados do experimento 175

Capítulo 1

Introdução

As aplicações corporativas são um tipo específico de software que resolve problemas para organizações [12]. Em geral, essas aplicações são tão complexas quanto o domínio onde estão inseridas e permitem que vários usuários executem tarefas do negócio simultaneamente. Consultorias financeiras globais estimam que o mercado de aplicações corporativas movimentou US$ 135,9 bilhões em 2013 e crescerá para US$ 182,7 bilhões em 2018 e US$ 213 bilhões em 2020 [21, 82, 55, 77]. Portanto, esse é um setor relevante no mercado de software, onde possíveis melhorias são úteis para o aumento da produtividade e redução de custo.

Alguns exemplos de aplicações corporativas [31] são sistemas para folha de pagamento, prontuário, rastreamento de entregas, análise de custo, análise de crédito, contabilidade, seguro, cadeia de suprimento, atendimento ao usuário, comércio eletrônico e sistemas integrados de gestão empresarial (ERP). Por outro lado, software para injeção eletrônica, processamento de textos, controle de elevadores, controle de plantas químicas, roteamento telefônico, sistemas operacionais, compiladores, jogos, simuladores e análise de dados não são considerados como aplicações corporativas.

Antigamente, os termos mais utilizados para identificar as aplicações corporativas eram Sistemas de Informação e Processamento de Dados. Independente da nomenclatura utilizada, esse tipo de software possui características bem específicas [31, p. 2]: dados persistentes, que devem resistir a falhas de hardware, sistema operacional, máquina virtual e rede; grandes volumes de dados, chegando a gigabytes de dados em milhões de registros; estrutura complexa para organização dos dados, geralmente utilizando dezenas ou centenas de tabelas relacionais e chaves estrangeiras, mas também podendo utilizar novas estruturas

NoSQL [86]; acesso concorrente a dados, com uma sessão concorrente para cada usuário ativo; regras de negócio que, muitas vezes, são dinâmicas e de difícil compreensão; uma enorme quantidade de telas para interface gráfica com o usuário, que servem distintamente a perfis de usuários diferentes; e interfaces para integração com outros sistemas.

Interfaces Gráficas de Usuário (GUI, do inglês *Graphical User Interface*) tendem a consumir a maior parte do esforço no desenvolvimento de aplicações corporativas. Há quase três décadas, estudos já indicavam que o desenvolvimento de GUI representa aproximadamente 50% do esforço total no desenvolvimento de software [70]. Esse montante tende a crescer pois as formas de interação com o usuário têm evoluído, com sistemas *web* 2.0 e reconhecimento de imagem e voz [62].

Uma pesquisa foi realizada com 83 desenvolvedores de software a fim de conferir o tamanho do esforço no desenvolvimento de GUI no contexto atual (Apêndice A). Foi observado um resultado abaixo da literatura, uma vez que os entrevistados estimam o esforço no desenvolvimento da GUI em média 31%. Porém, esse resultado está alinhado ao que a literatura indica sobre a relevância da GUI no desenvolvimento de aplicações corporativas e justifica a pesquisa e o desenvolvimento de novas abordagens que aumentem a sua produtividade.

Os modelos arquiteturais para aplicações corporativas têm evoluído nos últimos anos impulsionados pelo aumento da complexidade das aplicações e pela necessidade de separação entre GUI e o restante do projeto e código da aplicação. Por exemplo, a arquitetura monolítica [73] já é considerada uma abordagem rudimentar para o desenvolvimento de GUI, pois coloca no mesmo artefato códigos com responsabilidades diferentes. Dado que a separação das responsabilidades misturadas é muito difícil, o reúso de código é praticamente inviável.

Nesta pesquisa, foram encontramos exemplos reais de código monolítico em projetos de código aberto que estão hospedados em repositórios públicos, tais como *Tuleap*[1] e *Dolibarr*[2]. Ao analisar parte do código fonte desses sistemas (seção 2.1), foi verificado o entrelaçamento de responsabilidades distintas e o consequente acoplamento. Desse modo, o reúso de código é inviável e muitos componentes precisam ser refeitos completamente, mesmo que possuam muito código semelhante.

O padrão arquitetural Camadas [31] representa uma evolução para reduzir o acoplamento

[1] http://www.tuleap.org

[2] http://www.dolibarr.org

entre GUI e domínio do negócio, isolando código com responsabilidades diferentes em artefatos distintos. Nesse caso, a camada de GUI só pode conhecer a camada imediatamente inferior, que é a camada de domínio [16]. O relacionamento no sentido oposto deve ser evitado e o domínio se torna totalmente independente da GUI. As telas da GUI referenciam explicitamente as entidades do domínio, suas propriedades, relacionamentos e operações. Por outro lado, a camada do domínio de negócio pode ser reutilizada sem alterações em telas diferentes. Pode-se concluir que uma arquitetura orientada a camadas desacopla o domínio e o torna reutilizável, porém o mesmo não ocorre com a GUI.

Uma das formas de eliminar o acoplamento entre componentes de software é utilizar metadados para referenciar componentes externos [34, p. 13] [48, pp. 144-149]. Por exemplo, em vez de construir um componente GUI para representar apenas a matrícula de um funcionário, o desenvolvedor pode criar um meta componente de GUI a fim de representar qualquer propriedade de qualquer entidade do domínio.

A literatura acadêmica possui diversos exemplos de abordagens baseadas em metadados que podem ser aplicadas no desenvolvimento de aplicações corporativas. A Programação Orientada a Aspectos [54] pode utilizar metadados para selecionar os pontos de uma aplicação que serão estendidos por funcionalidades transversais. No Desenvolvimento Dirigido por Modelos [8], são criadas transformações baseadas nos metadados de modelos independentes de plataforma e modelos de plataformas específicas, com o intuito de gerar código executável, inclusive para aplicações corporativas [94].

Na indústria, diversas tecnologias baseadas em metadados para o desenvolvimento de sistemas são utilizadas. A técnica de reflexão permite acessar os metadados de sistemas orientados a objetos em tempo de execução e está presente em linguagens populares, como Java [22] e C# [66]. Os *templates* de geração de código [45] possuem estrutura baseada em metadados e são utilizados em plataformas de *Scaffolding* como *Ruby on Rails*[3], *Grails*[4], *Spring Roo*[5] e *Yeoman*[6]. Além disso, o uso de metadados pode ser encontrado no padrão arquitetural AOM - *Adaptive Object Model* [103, 99] e nos padrões para arcabouços baseados em metadados [38, 42].

[3] http://rubyonrails.org
[4] http://grails.org
[5] http://projects.spring.io/spring-roo
[6] http://yeoman.io

Ao analisar quatro sistemas reais de código aberto que utilizam metadados na sua construção — *AndroMDA*[7], *Redmine*[8], *SwingBean*[9] e *Ink*[10] — identificamos alguns artefatos complexos de GUI (Figura 2.3) que não possuem referências diretas ao domínio, substituindo-as por referências a metadados. Esses artefatos atuam como meta componentes de GUI e, em teoria, poderiam ser reutilizados em outros domínios que publiquem os seus metadados, diferentemente dos componentes comuns de GUI, que são dependentes de um domínio específico. No entanto, identificamos quatro características, nas abordagens analisadas, que inibem o efetivo reúso de GUI: generalidade excessiva, complexidade, dificuldade para manutenção e má divisão de responsabilidades entre os meta componentes de GUI.

As abordagens de programação orientada a aspectos, desenvolvimento dirigido por modelos, reflexão e arcabouços baseados em metadados são genéricas e possuem aplicabilidade horizontal. Portanto, não oferecem artefatos baseados em metadados para lidar com as especificidades do relacionamento entre a GUI e o domínio de aplicações corporativas. No caso específico do desenvolvimento dirigido por modelos, as transformações geralmente são baseadas em modelos UML complexos, que lidam com muitos elementos das plataformas genéricas e específicas. Assim sendo, os desenvolvedores de GUI precisariam conhecer muitos detalhes de UML para implementar meta componentes de GUI. Além disso, identificamos dificuldades para alteração dos *templates* que muitas vezes ficam embutidos ou inacessíveis nos geradores de código.

No entanto, a maior limitação encontrada foi a mistura das responsabilidades dos meta componentes de GUI. A despeito dos metadados das aplicações corporativas serem simples e bem definidos — entidades, propriedades e relacionamentos [74, 75] — as abordagens analisadas não fatoram as responsabilidades de GUI segundo esses metadados. Sendo assim, é comum encontrar um único artefato que implementa várias responsabilidades diferentes. Exemplos práticos desse problema são encontrados em plataformas utilizadas na indústria, tais como um *template Rails* adaptado[11] que gera formulários para entidades e também implementa a geração dos campos para propriedades e relacionamentos; e outro *template* do

[7] http://www.andromda.org
[8] http://www.redmine.org
[9] http://swingbean.sourceforge.net
[10] http://code.google.com/a/eclipselabs.org/p/ink
[11] http://goo.gl/HDVwXj

sistema de gestão de conteúdo *BrowserCMS*[12] que possui um *case* para diferenciar vários tipos de campo dentro do código de geração de formulários. A abordagem AOM [99] também mistura as responsabilidades, pois os meta componentes de propriedades podem ser utilizados para produzir GUI de relacionamentos. Além disso, em AOM, os meta componentes de entidades são responsáveis por compor os meta componentes de propriedades.

Essa mistura de responsabilidades torna os meta componentes mais acoplados entre si e, embora não estejam acoplados ao domínio, reduz a reusabilidade. Dessa forma, é importante que os meta componentes de GUI possuam responsabilidades bem encapsuladas e exclusivamente referentes a entidades, propriedades ou relacionamentos do modelo do domínio. Assim, mantêm-se os meta componentes pequenos, específicos e reutilizáveis, de modo que se um desenvolvedor necessitar alterar um detalhe específico da GUI, não precisará reescrever um *template* todo, pois bastará substituir um artefato pequeno e específico.

É neste contexto de suporte ao desenvolvimento de GUI para aplicações corporativas que se insere este trabalho, mais especificamente na proposição de mecanismos que viabilizem o reúso e facilitem a manutenção de GUI para tais aplicações.

1.1 Problemática

Ao analisar o estado da arte no desenvolvimento de GUI para aplicações corporativas, não encontramos uma abordagem que forneça simultaneamente reusabilidade e manutenibilidade. As abordagens baseadas em metadados são eficazes em relação ao reúso, uma vez que são independentes do domínio e podem ser reutilizadas em diversos projetos, porém são de difícil manutenção devido à generalidade excessiva, complexidade e má divisão de responsabilidades. Por outro lado, as abordagens mais utilizadas na indústria são voltadas para manutenibilidade, mas não utilizam meta componentes de GUI e fazem referências diretas ao domínio, o que inviabiliza o reúso de GUI.

Com a finalidade de facilitar a compreensão do problema de pesquisa, listamos os seguintes exemplos de problemas, no desenvolvimento de GUI para aplicações corporativas, que seriam facilmente resolvidos se houvesse uma abordagem viabilizando tanto reusabilidade quanto manutenibilidade:

[12] http://goo.gl/t5Q8QB

1. *Dado que existem meta componentes padrão para todas as entidades, propriedades e relacionamentos do domínio, o desenvolvedor precisa definir uma especificidade de GUI e deseja reutilizá-la automaticamente.*

   Por exemplo, todas as propriedades do tipo inteiro são, por padrão, representadas por um meta componente chamado `NumberField` e o desenvolvedor decide que as propriedades inteiras cujo nome contenha `cpf` sejam representadas pelo meta componente `CpfField`.

   Numa plataforma focada em reúso de GUI, esta modificação deveria ser realizada apenas uma vez, podendo ser reutilizada automaticamente em outros pontos do mesmo projeto e em outros projetos posteriores.

2. *Em um sistema que define uma transição padrão para todas a telas, o desenvolvedor quer definir um novo comportamento de GUI para algumas telas e reutilizar esse comportamento posteriormente.*

   Por exemplo, em um determinado sistema, a transição da tela de listagem para os formulários de cadastro sempre abre uma nova página. O desenvolvedor deseja otimizar a interação com o usuário, no caso das entidades com poucos campos, criando uma janela *pop-up* para os formulários de cadastro sobre a tabela de listagem.

   Esse novo comportamento da GUI se mostra mais ágil e interessante para o usuário e deve ser facilmente reutilizado em algumas entidades do mesmo ou de outros domínios.

3. *Em uma tela composta a partir da interação de vários meta componentes de GUI, deve ser fácil substituir qualquer um deles, reutilizando os demais.*

   Em um sistema, todas as telas de listagem, por exemplo, possuem um meta componente principal, uma tabela de listagem paginada (`PaginatedListingTable`). Esse artefato delega o desenho do cabeçalho da tabela para o meta componente `TablePropertyHead` e o desenho das linhas para `TableInstanceLine`, que, por sua vez, delega o desenho de cada célula para `TablePropertyCell`.

   Em algum momento, o cliente deseja substituir todas as tabelas paginadas do sistema por tabelas infinitas, que são carregadas assincronamente à medida que o usuário rola

a tela para baixo. Porém o desenho dos cabeçalhos, das linhas e células das tabelas não deve ser alterado. Apenas os botões de paginação serão substituídos pela barra de rolagem e por um indicador de que as linhas seguintes estão sendo carregadas.

Para realizar essa alteração facilmente e com foco em reúso, o desenvolvedor deve poder trocar apenas o meta componente `PaginatedListingTable` por um `InfiniteListingTable`, mantendo toda a estrutura restante das páginas de listagem.

De modo semelhante, os meta componentes de cabeçalho, linhas e células também devem ser substituíveis de forma fácil e isoladamente.

4. *Um desenvolvedor criou uma tela que envolve dados de várias entidades relacionadas e deseja reutilizar a mesma estrutura em entidades semelhantes.*

   Em um sistema de controle acadêmico hipotético, foi necessário um grande esforço para criar um meta componente de relatório tabular (chamado `OneToManyAndManyToOneReport`), onde as colunas representam os dias de aula, as linhas são os alunos de uma turma e as células marcam a presença ou a ausência do aluno naquela aula. Nesse cenário, o domínio pode ter sido modelado com três entidades, onde um `Aluno` se relaciona `um-para-muitos` com `Presença`, que se relaciona `muitos-para-um` com `Aula`.

   Como esse relatório é baseado em metadados, poderia ser reutilizado em quaisquer tabelas `A`, `B` e `C` que mantivessem o relacionamento `A um-para-muitos B muitos-para-um C`. Por exemplo, considere um sistema de folha de pagamento para representar as presenças dos trabalhadores de um departamento nos dias de trabalho. Isso seria facilmente reaproveitado, dado que as entidades `Trabalhador`, `Presença` e `DiaTrabalho` se relacionariam na forma `Trabalhador um-para-muitos Presença muitos-para-um DiaTrabalho`.

   Devido à divisão clara de responsabilidades, `OneToManyAndManyToOneReport` poderia ser reutilizado em um cenário ainda mais complexo. No mesmo sistema de controle acadêmico, haveria um relatório das notas dos alunos de uma turma. As colunas seriam as avaliações, as linhas os alunos e as células teriam um número decimal

relativo à nota do aluno naquela avaliação. Nesse caso, o relacionamento entre `Aluno`, `Nota` e `Avaliação` também segue a estrutura `Aluno um-para-muitos Nota muitos-para-um Avaliação`. A única mudança necessária seria no meta componente responsável por desenhar as células do relatório. Em vez de ler um booleano presença/ausência, ele interpretaria a nota, inclusive pintando-a de vermelho se for abaixo da média.

O pseudocódigo da Listagem 1.1 representa um exemplo hipotético, onde os meta componentes de GUI possuem responsabilidades bem definidas, de acordo com os metadados do modelo do domínio. Esse exemplo não segue nenhuma das abordagens existentes, pois o seu objetivo é sugerir um método realmente efetivo para o reúso de GUI. Os três meta componentes de GUI, que estão definidos entre as linhas 2 e 13, podem ser facilmente reutilizados e modificados em domínios diferentes.

O uso de metadados fica evidente, por exemplo, na linha 8, onde método `caixa_texto_simples` (que é um meta componente de GUI) atribui valor ao atributo `input.name` com o nome de qualquer propriedade recebida por parâmetro. Em contrapartida, as linhas 29 a 31 mostram componentes comuns, que atribuem valor ao mesmo atributo com referências diretas ao domínio da aplicação.

A Listagem 1.1 também sugere um **método** para construção de GUI baseada em metadados com responsabilidades bem definidas. Os meta componentes de GUI (linhas 2 a 13) e os metadados do domínio (linhas 16 a 19) podem ser combinados (linhas 22 a 25) a fim de produzir componentes de GUI (linhas 28 a 32). Com essa abordagem, os desenvolvedores focariam seus esforços na definição do domínio e os meta componentes de GUI criados para projetos anteriores poderão ser automaticamente reutilizados.

Por exemplo, as linhas 23 e 24 configuram meta componentes de GUI para todas as propriedades numéricas e para todas as propriedades dos demais tipos, respectivamente. Essa configuração e esses meta componentes podem ser reutilizados, do modo como estão, em sistemas de domínios totalmente diferentes. Caso exista alguma demanda específica de GUI em um projeto, um novo meta componente de GUI pode ser desenvolvido e armazenado numa biblioteca para reúso futuro. Dessa forma, o esforço repetitivo de GUI pode ser reduzido e os desenvolvedores precisarão implementar apenas a GUI que seja específica e originada em cada projeto.

Código Fonte 1.1: Processo de construção de GUI a partir de um *widget*

```
1  # Definir uma biblioteca de meta componentes de GUI
2  def formulario_simples(entidade):
3    return "<form action='/api/" + entidade.nome + "'>"
4      + GUI.desenharPropriedades(entidade.propriedades) + "</form>"
5
6  def caixa_texto_simples(propriedade):
7    return propriedade.etiqueta + ":<br>" +
8      "<input type='text' name='" + propriedade.nome + "'><br>"
9
10 def caixa_texto_numero(propriedade):
11   return propriedade.etiqueta + ":<br>"+
12     "<input type='number' name='" + propriedade.nome + "' min='" +
13     propriedade.minimo + "' max='" + propriedade.maximo + "'><br>"
14
15 # Publicar os Metadados do Dominio
16 entidade Produto
17   propriedade string nome
18   [minimo: 0, maximo: 120] propriedade int idade
19   [etiqueta: "Data de admissao"]propriedade Date admissao
20
21 # Configurar e iniciar a Plataforma de GUI
22 GUI.adicionarMetaComponente("Entidade", formulario_simples)
23 GUI.adicionarMetaComponente("Propriedade", "number", caixa_texto_numero)
24 GUI.adicionarMetaComponente("Propriedade", caixa_texto_simples)
25 GUI.desenhar(Produto)
26
27 # Resultado (indentacao artificial)
28 <form action='/api/Produto'>
29   Nome:<br><input type='text' name='nome'><br>
30   Idade:<br><input type='number' name='idade' min='0' max='120' <br>
31   Data de admissao:<br><input type='text' name='admissao'><br>
32 </form>
```

No entanto, entre as abordagens baseadas em metadados que podem ser utilizadas no desenvolvimento de GUI para Aplicações corporativas (ver detalhes no capítulo 3), não foram encontradas técnicas que viabilizem o reúso como definido na Listagem 1.1.

Diante desse contexto, formula-se a hipótese de que *é possível aumentar o reúso sem prejudicar a manutenibilidade no desenvolvimento de GUI para aplicações corporativas web, através do encapsulamento das responsabilidades de GUI em artefatos de granularidade fina com base nos metadados do modelo do domínio.*

1.2 Objetivos

O objetivo geral neste trabalho é propor, implementar e avaliar uma abordagem para aumentar o reúso no desenvolvimento de GUI para aplicações corporativas de forma que não prejudique a manutenibilidade da GUI, resultando em aumento de produtividade.

Como objetivos específicos, tem-se:

1. Analisar os pontos fortes e as limitações das abordagens relacionadas para o desenvolvimento de GUI baseadas nos metadados do modelo do domínio;

2. Documentar a síntese do conhecimento empírico acumulado nesta pesquisa, sobre o desenvolvimento de GUI baseadas nos metadados do modelo do domínio, através de uma linguagem de padrões (de projeto e arquiteturais) [4, 13, 60];

3. Disponibilizar plataformas de código aberto que implementem concretamente a linguagem de padrões documentada;

4. Avaliar experimentalmente o impacto dessa abordagem na produtividade e na reusabilidade do desenvolvimento de aplicações corporativas em ambiente controlado.

1.3 Metodologia

O desenvolvimento deste trabalho foi conduzido através de quatro procedimentos, detalhados a seguir.

1. *Análise do estado da arte no desenvolvimento de GUI para Aplicações corporativas, identificando as características importantes para o amplo reúso de código;*

Neste procedimento, foi elaborada uma linguagem de padrões, contendo sete padrões de projetos e dois padrões arquiteturais relacionados entre si, a fim de formalizar o conhecimento adquirido sobre componentes de GUI com responsabilidades bem definidas, aumentando, portanto, o potencial de reúso em GUI para Aplicações corporativas.

A Linguagem de padrões foi avaliada experimentalmente através dos procedimentos detalhados nos itens 3 e 4 a seguir.

2. *Estudo da viabilidade da linguagem de padrões proposta para implementação de aplicações corporativas e análise do seu impacto sobre o acoplamento GUI–Domínio;*

   Após a definição da linguagem de padrões, foi realizado um estudo teórico (Apêndice B), que demonstrou a viabilidade do uso de uma tecnologia baseada nos padrões propostos para implementar uma tela complexa de uma aplicação corporativa. Além disso, esse estudo mostrou que a abordagem reduz o acoplamento entre as camadas de GUI e domínio. Em vez do acesso direto das telas da GUI às classes do domínio, apenas dados de configuração, que podem ser facilmente alterados, conhecem o domínio.

   Assim sendo, a linguagem de padrões foi considerada madura bastante para ser implementada através de tecnologias concretas.

3. *Projeto e desenvolvimento de Plataformas que implementam a linguagem de padrões definida no procedimento 1 para aplicações corporativas com arquitetura web;*

   Neste procedimento, dois arcabouços de software (*frameworks*) foram implementados em linguagens diferentes para demonstrar a viabilidade técnica da linguagem de padrões proposta nesta tese.

   O primeiro arcabouço, chamado Geneguis, implementa o *frontend* na linguagem CoffeeScript[13] e o *backend* em Java e Spring. O nome do segundo arcabouço é Angular M e permite desenvolvimento de componentes TypeScript[14], baseados em metadados de entidades e propriedades, na tecnologia de *front-end* Angular 4[15]. Os dois projetos

[13] coffeescript.org/

[14] www.typescriptlang.org

[15] angular.io

são de código aberto e podem ser utilizados para o desenvolvimento de aplicações corporativas reais.

O Geneguis possui uma arquitetura mais invasiva e toda a GUI da aplicação é gerada a partir do arcabouço. Por outro lado, o Angular M é mais flexível, pois pode ser utilizado apenas nas telas que precisam ser montadas baseadas em metadados. Enquanto o Geneguis define as tecnologias que devem ser utilizadas tanto no *frontend* como no *backend*, Angular M pode interagir com qualquer tecnologia de *backend* que implemente REST. A última diferença entre os arcabouços é que o Geneguis implementa toda a linguagem de padrões proposta, enquanto o Angular M não possui os metadados de relacionamentos entre entidades.

4. *Realização de um experimento para a customização de trechos de GUI em aplicações corporativas, com profissionais e estudantes, em ambiente controlado.*

   Na indústria de software, novas tecnologias surgem constantemente, prometendo, dentre outras coisas, enormes ganhos de produtividade. Todavia não é comum haver avaliações quantitativas dessas promessas e as decisões arquiteturais de projeto de software acabam sendo tomadas considerando suposições em vez de fatos. Diante desse contexto, Juristo e Moreno [52] reforçam que a Engenharia de Software precisa se tornar uma ciência madura, através do uso do método científico para compreendermos a construção de software. Os pesquisadores devem utilizar o método experimental para iterativamente montar uma base de conhecimento confiável acerca do desenvolvimento de software, auxiliando a indústria na decisão de quais abordagens e tecnologias adotar.

   Nesse sentido, um experimento foi planejado a fim de gerar resultados, para a abordagem aqui proposta, que possam ser utilizados para decisões de projeto na indústria de software.

   Os objetivos deste trabalho foram avaliados seguindo a técnica GQM do inglês *Goal–Question–Metric* [89], pela qual foram definidas as seguintes questões:

   - A abordagem proposta melhora a produtividade no desenvolvimento de GUI de aplicações corporativas?

- A abordagem proposta permite o reúso de código no desenvolvimento de GUI de aplicações corporativas?

As métricas que foram utilizadas para responder a essas questões são: o *tempo gasto* para realizar tarefas de customização de GUI e o *reúso* de componentes de GUI.

As comparações foram feitas em 12 pares de telas, onde cada par possui uma implementação em Ruby on Rails e outra em Angular M, ambas com código HTML final idêntico. Cada par foi customizado da mesma forma e um mesmo teste Selenium [16] validou cada implementação customizada, antes de aceitar a submissão do código final pelo participante. Procurou-se distribuir diversos e variados tipos de customização entre as 12 telas do experimento.

15 profissionais desenvolvedores de software participaram voluntariamente do experimento. Inicialmente eles foram capacitados nas tecnologias Ruby on Rails, Angular 4 e Angular M. Passaram por exames para confirmar o nivelamento de conhecimento necessário para o experimento. Por fim, para cada participante foram sorteadas seis questões para serem implementadas em Ruby on Rails e seis em Angular M.

O tempo de início e fim de cada atividade foi coletado automaticamente por *Shell Scripts* que foram escritos exclusivamente para este experimento. O *script* de inicialização baixa o código inicial de cada tarefa tanto para Ruby on Rails como para Angular M, demarcando o tempo de início da tarefa quando todas as dependências necessárias estavam resolvidas. E o *script* de finalização da tarefa executa um teste Selenium e, se for finalizado com sucesso, o tempo final da tarefa é marcado e o código final do participante é submetido para o repositório online Github, de onde foi coletado para posterior avaliação.

Em seis telas, já havia algum componente Angular M disponível para auxiliar na realização da customização, sendo, portanto, reutilizado. Nas outras seis telas, o código Angular M precisava ser desenvolvido do zero. Dessa forma, os resultados desses dois grupos de seis tarefas puderam ser comparados a fim de verificar se o reúso de componentes, que é mais fácil no Angular M, impactou a produtividade no contexto do experimento.

[16] http://www.seleniumhq.org/projects/webdriver/

Nas seis telas em que havia componentes prontos para serem reutilizados, o tempo médio para a customização em Angular M foi menor do que em Ruby on Rails em todas, porém com significância estatística (para negar a hipótese nula de igualdade) em quatro telas.

Por outro lado, nas seis telas em que não havia componentes prontos, houve um empate com três tempos médios menores em cada tecnologia. Sendo que houve significância estatística apenas em uma das telas em Ruby on Rails demandando menos tempo para a customização.

Os comentários e os códigos resultantes dos participantes foram analisados a fim de identificar quais as características que conferiram bons resultados ao Angular M e que fragilidades podem ser eliminadas.

1.4 Contribuições

A principal contribuição deste trabalho foi identificada após a constatação de um problema recorrente nas abordagens existentes para construção de GUI em aplicações corporativas a partir de metadados: o entrelaçamento de responsabilidades, que dificulta o reúso de código. Portanto foi proposta uma nova abordagem que encapsula os componentes baseados em metadados, facilitando o seu reúso e aumentado a produtividade.

Essa abordagem foi documentada conceitualmente através de uma linguagem de padrões [98] e foi implementada concretamente através de dois arcabouços de código aberto: Geneguis[17] e Angular M[18]. Tanto a linguagem de padrões como os arcabouços são contribuições importantes deste trabalho, uma vez que os arcabouços podem ser utilizados para a implementação de aplicações corporativas reais, e a linguagem pode guiar a criação de novos arcabouços para GUI de aplicações corporativas.

Por fim, o experimento realizado também representa uma contribuição importante, não apenas por causa do resultado final demonstrado, mas porque vários problemas recorrentes em experimentos similares foram resolvidos. Dessa forma, experimentos futuros podem ser beneficiados ao adotar o *modus operandi* descrito no capítulo 7. Especificamente, os *Shell*

[17] github.com/rodrigovilar/geneguis

[18] github.com/rodrigovilar/angularm www.npmjs.com/package/angularm

scripts de suporte aos participantes preparam o ambiente para cada atividade que precisa ser feita, além de coletar automaticamente o horário de início e fim da atividade, reduzindo um problema comum nesse tipo de experimento, que é o esquecimento de marcar manualmente os horários. Além disso, foram utilizados testes automatizados para verificar se as atividades tinham sido concluídas com sucesso antes que o participante as submetesse para análise. A última contribuição do experimento foi a preparação de material, vídeo-aulas e exercícios, a fim de nivelar e conferir o conhecimento dos participantes, antes das atividades do experimento propriamente ditas.

1.5 Organização do documento

No capítulo 2 apresenta-se a fundamentação teórica deste trabalho, que consiste do histórico de abordagens para o desenvolvimento de GUI em aplicações corporativas com enfoque no desacoplamento entre GUI e domínio: arquitetura monolítica, camadas e metadados.

No capítulo 3 analisam-se os trabalhos relacionados à abordagem proposta: programação orientada a aspectos, desenvolvimento dirigido por modelos, geração de código baseada em *templates*, *adaptive object model* e arcabouços baseados em metadados.

No capítulo 4 apresentam-se os padrões obtidos a partir da análise dos trabalhos relacionados, acadêmicos e da indústria, no contexto do desenvolvimento de GUI desacoplada do domínio.

No capítulo 5 analisa-se uma nova abordagem para o desenvolvimento de GUI desacoplada do domínio de aplicações corporativas, através de uma linguagem de padrões para meta componentes de GUI.

No capítulo 6 descreve-se o projeto de alto nível e os resultados da implementação de duas plataformas de código aberto, chamadas Geneguis e Angular M, que contemplam os padrões descritos nos capítulos anteriores.

No capítulo 7 apresenta-se o projeto do experimento realizado para a avaliação deste trabalho, bem como a análise dos seus resultados.

Por fim, no capítulo 8, são apresentadas as conclusões deste trabalho e os caminhos para trabalhos futuros são discutidos.

Neste documento, algumas convenções de formatação são adotadas para facilitar a com-

preensão dos elementos citados. Os nomes dos padrões definidos no capítulo 4 aparecem como links em caixa alta, como em WIDGET ENGINE. Quando são citados padrões já definidos na literatura, o texto está em caixa alta, mas sem links, como em COMPOSITE. Os componentes ou sistemas citados aparecem em *itálico*, assim como os demais termos em inglês. Por fim, classes, arquivos e trechos de código aparecem em `fonte monoespaçada`.

Capítulo 2

Fundamentação teórica

Neste capítulo analisam-se abordagens para o desenvolvimento de GUI em aplicações corporativas com foco no reúso de código. A arquitetura monolítica é uma abordagem rudimentar, que mistura as responsabilidades de GUI e domínio nos mesmos artefatos de código. Todavia o desacoplamento é importante para fomentar o reúso dos componentes de software, por isso duas formas de desacoplar GUI e domínio são apresentadas: a arquitetura orientada a camadas, que torna o domínio independente da GUI; e o uso de metadados, que provê a independência no sentido inverso (GUI > Domínio). No final do capítulo, o acoplamento entre os próprios componentes de GUI baseados em metadados também é analisado.

2.1 Arquitetura monolítica

Uma forma rudimentar de implementar GUI é escrevê-la no mesmo artefato de código onde estão definidas as regras do domínio da aplicação corporativa. Essa abordagem — denominada como monolítica [73] — gera um altíssimo acoplamento entre a GUI e o domínio, pois o código entrelaça suas responsabilidades. Assim sendo, os elementos de software não possuem existência autônoma e não podem ser reutilizados individualmente em outros pontos da aplicação.

Por exemplo, a Listagem 2.1 implementa um relatório com destaque visual dos produtos com crescimento de vendas maior que 10% no mês atual. O código escrito na linguagem *Python*[1] mistura as regras de negócio do relatório com a concatenação das strings HTML que

[1] https://www.python.org/

formam a GUI. Se a funcionalidade de relatório com destaque também for necessária em outro módulo do sistema, ela precisará ser escrita novamente a partir do zero. Da mesma forma, se existir um relatório de produtos sem o destaque de vendas, o código de concatenamento de strings HTML precisa ser repetido em outro artefato.

Código Fonte 2.1: Código monolítico com GUI e Domínio de negócio

```
1  def destaque_laranja_vendas_dez():
2    resultado = "<table>" #Nova tabela HTML
3    mes_atual = datetime.datetime.now().month
4    for produto in RepositorioProdutos.listar_todos() :
5      resultado += "<tr>" #Nova linha de tabela HTML
6      resultado += "<td>" + produto.nome + "</td>" #Celula Nome
7      delta_vendas =
8        RepositorioVendas.calcular_delta_vendas(produto, mes_atual)
9      cor = "white"
10     if delta_vendas > 0.1 : #Destaque laranja para Delta vendas > 10%
11       cor = "orange"
12     #Celula Delta vendas
13     resultado += "<td color=" + cor + ">" + delta_vendas + "</td></tr>"
14   return resultado + "</table>" #Fecha tabela e retorna o HTML
```

Esse exemplo demonstra que, quando há acoplamento forte, é preciso duplicar código. Todavia, a recíproca também é verdadeira, pois o código duplicado representa um acoplamento conceitual [32]. Por exemplo, se houver dois relatórios como o da Listagem 2.1, que destacam produtos por suas vendas, e o cliente decide alterar a forma de destaque em todo o sistema, substituindo a cor laranja das células de tabelas por fontes em negrito. Nesse caso, os dois blocos de código duplicado precisarão ser alterados. Por essa razão, o acoplamento e a duplicação de código estão entre as razões para o aumento no esforço na manutenção de sistemas [59, p. 37], incluindo aplicações corporativas.

Com o intuito de encontrar exemplos reais de código monolítico, foi realizada uma breve pesquisa no repositório de projetos de código aberto *Github*[2]. Dois exemplos de código monolítico foram rapidamente encontrados nos projetos *Tuleap*[3], uma ferramenta para gestão

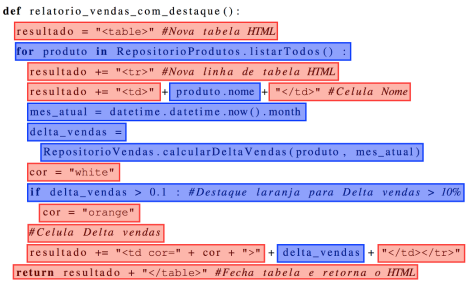
[2]http://www.github.com

[3]http://www.tuleap.org, http://goo.gl/vEieYA

de projetos, e *Dolibarr*[4], um sistema de ERP e CRM (*Customer Relationship Management*).

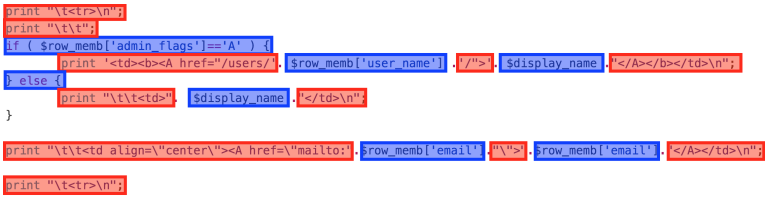
O código PHP desses dois projetos foi analisado juntamente com o código da Listagem 2.1, marcando as responsabilidades de GUI em vermelho e as responsabilidades do domínio do negócio em azul. O resultado desta análise está na Figura 2.1, que expõe claramente o entrelaçamento de código com responsabilidades distintas e o consequente *acoplamento*, que é a força do relacionamento entre os módulos de um sistema [1, p. 3-2]. Quanto mais trechos marcados com azul, pior é a dependência da GUI em relação ao domínio. Essa dependência pode variar de um simples acoplamento de dados, no qual um componente conhece a estrutura dos dados de outro componente, até o mais prejudicial acoplamento de conteúdo, onde um componente altera os dados de outro componente ou quando a execução salta de um componente para o interior de outro [2, p. 200].

A Figura 2.1 mostra que os componentes de GUI e do domínio do negócio possuem acoplamento de conteúdo, pois a execução se alterna várias vezes nos detalhes internos destes dois componentes. Desse modo, o reúso de código é inviável e muitos componentes precisam ser refeitos completamente, mesmo que possuam muito código semelhante.



(a) Código da listagem 2.1

(b) Relatório no Dolibarr



(c) Listagem de usuários no Tuleap

Figura 2.1: Marcação de responsabilidades de GUI e Domínio em códigos monolíticos

[4]http://www.dolibarr.org, http://goo.gl/uMxF7d

2.2 Padrão arquitetural Camadas

Uma solução simples para reduzir o acoplamento entre a interface gráfica e o domínio do negócio é a divisão do software em camadas [31], que isolam os trechos de código com responsabilidades diferentes em componentes distintos. Uma camada superior só pode conhecer e invocar a camada imediatamente inferior. O relacionamento no sentido oposto, das camadas inferiores para as superiores, deve ser evitado. Brown et al. [16] propuseram a divisão do código de aplicações corporativas nas seguintes camadas e responsabilidades: apresentação, para exibir informações e tratar as requisições do usuário; domínio, responsável pelas regras de negócio do sistema; e persistência, em bancos de dados ou sistemas externos. Com essa arquitetura, a camada do domínio de negócio pode ser reutilizada em várias telas sem sofrer alterações. E em um cenário extremo, a GUI pode até mesmo ser totalmente removida, quando a camada de domínio é invocada diretamente por sistemas externos.

A Listagem 2.2 representa um refatoramento [33, p. 9] da Listagem 2.1, que isolou quase todas as responsabilidades de GUI em apenas um artefato (método `relatorio_vendas_com_destaque`). O restante do código lida com o domínio de negócio, como, por exemplo, o método `produtos_vendas`, que pode ser reutilizado por outras funcionalidades do sistema.

Ao marcar visualmente as responsabilidades da Listagem 2.2 (como foi feito na Figura 2.1), pode-se ver os códigos de GUI e domínio do negócio separados em blocos bem definidos, que podem ser entendidos como camadas de um sistema (Figura 2.2a). Nesse caso, a camada vermelha (GUI) invoca a camada azul (domínio). Repetimos o mesmo procedimento sobre código real encontrado no repositório *Github* do projeto *ERPNext*[5] (um ERP de código aberto), e obtivemos um resultado visual parecido (Figura 2.2b), com camadas bem definidas.

No caso da Listagem 2.2, o código do domínio não está acoplado à GUI, portanto pode ser quase totalmente reutilizado, do jeito que está, em outras telas ou em chamadas de sistemas externos. No entanto, ainda existe acoplamento no caminho inverso. Em três pontos da Figura 2.2a, a GUI conhece a estrutura dos dados do domínio do negócio, portanto não pode ser reutilizada para outros domínios. O mesmo acoplamento ocorre na Figura 2.2b, cuja GUI

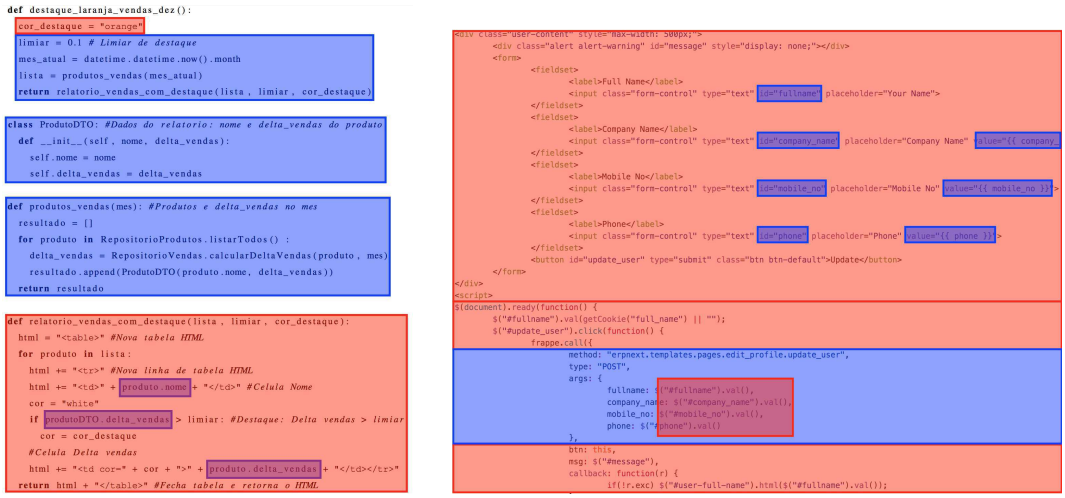
[5] http://www.erpnext.com, http://goo.gl/ELyxls

conhece os detalhes dos dados do domínio em sete pontos.

Código Fonte 2.2: Código com GUI e Domínio de negócios separados em camadas

```
1  def destaque_laranja_vendas_dez():
2    cor_destaque = "orange"
3    limiar = 0.1 # Limiar de destaque
4    mes_atual = datetime.datetime.now().month
5    lista = produtos_vendas(mes_atual)
6    return relatorio_vendas_com_destaque(lista, limiar, cor_destaque)
7
8  class ProdutoDTO: #Dados do relatorio: nome e delta_vendas do produto
9    def __init__(self, nome, delta_vendas):
10     self.nome = nome
11     self.delta_vendas = delta_vendas
12
13 def produtos_vendas(mes): #Produtos e delta_vendas no mes
14   resultado = []
15   for produto in RepositorioProdutos.listar_todos() :
16     delta_vendas = RepositorioVendas.calcular_delta_vendas(produto, mes)
17     resultado.append(ProdutoDTO(produto.nome, delta_vendas))
18   return resultado
19
20 def relatorio_vendas_com_destaque(lista, limiar, cor_destaque):
21   html = "<table>" #Nova tabela HTML
22   for produto in lista:
23     html += "<tr>" #Nova linha de tabela HTML
24     html += "<td>" + produto.nome + "</td>" #Celula Nome
25     cor = "white"
26     if produto.delta_vendas > limiar: #Destaque: Delta vendas > limiar
27       cor = cor_destaque
28     #Celula Delta vendas
29     html += "<td color="+cor+">"+produto.delta_vendas+"</td></tr>"
30   return html + "</table>" #Fecha tabela e retorna o HTML
```

Podemos concluir que uma arquitetura orientada a Camadas desacopla os elementos de um sistema e permite que as camadas inferiores possam ser reutilizadas por diversas camadas superiores. Porém, é proibitivo reutilizar uma camada superior, pois essa está acoplada aos



(a) Código da listagem 2.2 (b) Edição de perfil no ERPNext

Figura 2.2: Marcação de responsabilidades de GUI e Domínio em códigos com camadas

dados da camada ligeiramente inferior.

Neste trabalho, focamos no uso do padrão Camadas com o domínio do negócio sendo uma camada inferior e a GUI sendo a camada ligeiramente superior [16]. Uma vez que a GUI representa uma grande parte do esforço no desenvolvimento de aplicações corporativas, é importante reutilizar componentes de GUI para aumentar a produtividade na construção desse tipo de sistema.

Desse modo, a arquitetura em camadas, embora tenha contribuído bastante para o desenvolvimento de aplicações corporativas, não é uma abordagem efetiva em relação ao reúso de GUI, pois não remove o acoplamento de dados da camada de GUI para o domínio e não permite que ela seja amplamente reutilizada. Do mesmo modo que houve com a camada de domínio, é importante prover o desacoplamento da camada de GUI, a fim de torná-la reutilizável em diversos contextos diferentes.

2.3 Metadados em GUI

Uma das formas de eliminar o acoplamento entre componentes de software é utilizar metadados para referenciar componentes externos [34, p. 13] [48, pp. 144-149].

Fowler sugere duas vertentes principais para o uso de metadados: as APIs[6] de *reflexão*,

[6] *Application Programming Interfaces*

que permitem acessar os metadados dos elementos de software em tempo de execução; e a *geração de código fonte*, a partir da combinação de modelos do domínio e geradores, a fim de produzir funcionalidade semelhante a sistemas que foram implementados manualmente [34, p. 15]. No entanto, também é possível encontrar exemplos de uma terceira abordagem na literatura que substitui os geradores de código por *interpretadores* [103]. Desse modo, os metadados do modelos de domínio são combinados com componentes de alto nível, em tempo de execução, e também produzem funcionalidade semelhante a implementações manuais.

A técnica de reflexão está disponível em diversas linguagens de programação, como Java [22] e C# [66]. Do mesmo modo, a geração de código é comum em diversas abordagens e tecnologias, por exemplo: *templates* [45] nas tecnologias de *Scaffold* (*Ruby on Rails* [10], *Grails* [84] e *Spring Roo* [17]); e MDA - *Model driven architecture* [8, 24, 91]. Na abordagem de interpretação de metadados, encontramos exemplos que, embora não tenham obtido o sucesso mercadológico das demais abordagens, possuem implementações concretas: AOM - *Adaptive Object Model* [103, 99]; e *SwingBean* [38, 42].

Ao investigar projetos de código aberto, encontramos quatro exemplos reais de meta-componentes, que utilizam metadados para se desacoplar do domínio específico de cada aplicação. A Figura 2.3 mostra a marcação das responsabilidades de GUI em vermelho e as referências a metadados em amarelo. Nota-se que não há referências diretas ao domínio (que estariam marcadas em azul, conforme as figuras anteriores).

O projeto *AndroMDA* é uma ferramenta MDA para o desenvolvimento de aplicações baseado em modelos. A Figura 2.3a mostra um *template Velocity*[7] para geração de código no *AndroMDA* a partir dos metadados do domínio[8]. O *Redmine*, que é um sistema web para gestão de projetos feito em *Ruby on Rails*, também utiliza *templates*[9] de geração de código com metadados (Figura 2.3b). O *SwingBean* utiliza componentes Java baseados em metadados[10] para definir componentes de GUI de alto nível (Figura 2.3c). Por fim, o *Ink*, que implementa a abordagem AOM, também utiliza componentes Java baseados em metadados[11] para definir componentes de GUI (Figura 2.3d).

[7] http://velocity.apache.org/
[8] http://goo.gl/vGKbZ2
[9] http://goo.gl/q8zpUq
[10] http://goo.gl/KGiXSQ
[11] http://goo.gl/4Tn6GB

```
<deployment-report generation-date="$date">
  #foreach ($node in $nodes)
  <node name="$node.name">
    #foreach ($component in $node.deployedComponents)
    <component name="$component.name">
      #foreach ($artifact in $component.manifestingArtifacts)
      <artifact name="$artifact.name">
        #foreach ($package in $artifact.wrappedPackages)<package
          name="$package.fullyQualifiedName"></package>
        #end
      </artifact>
      #end
    </component>
    #end
  </node>
  #end
</deployment-report>
```

(a) *Template* de geração de código no *AndroMDA*

```
class <%= @controller_class %>Controller < ApplicationController
  unloadable

<% actions.each do |action| -%>

  def <%= action %>
  end
<% end -%>
end
```

(b) *Template* de geração de código no *Redmine*

```
class NumericWrapper implements ComponentWrapper {

  private JFormattedTextField numericText;
  private String type;
  private Double minValue;
  private Double maxValue;
  private String numberFormat;

  public Object getValue() {
          if((numericText).getText().equals(""))
                  return minValue != null ? minValue : 0;
          try {
                  if(type.equals(TypeConstants.INTEGER)){
                  return ((Number)numericText.getFormatter().stringToValue(numericText
                  if(type.equals(TypeConstants.LONG)){
                  return ((Number)numericText.getFormatter().stringToValue(numericText
                  if(type.equals(TypeConstants.FLOAT)){
                  return ((Number)numericText.getFormatter().stringToValue(numericText
                  if(type.equals(TypeConstants.DOUBLE)){
                  return ((Number)numericText.getFormatter().stringToValue(numericText
          } catch (Exception e) {
                  return minValue;
          }
          return minValue;
  }

  public String getStringValue(){
          return numericText.getText();
  }

  public void setValue(Object value) {
          try {
                  numericText.setText(numericText.getFormatter().valueToString(value));
          } catch (ParseException e) {
                  cleanValue();
```

(c) Componente de GUI no *SwingBean*

```
private void setExpandedKioskBars(boolean expanded) {
        int heightMark =(expanded ? 1 : (-1) );

        if( !hideInkNotations_ ) {
                if(mainExpBar_.getItem(0).getExpanded() != expanded ) {
                        mainExpBar_.getItem(0).setExpanded(expanded);
                }
        }

        if(inkstoneModel_==null) {return;}

        for (InkstoneProject project: inkstoneModel_) {
                if( project.getExpandItem().getExpanded() != expanded ) {
                        project.getExpandItem().setExpanded(expanded);
                }
                for(InkstoneLibrary library : project.getDslLibs()) {
                        if( library.getExpandItem().getExpanded() != expanded ) {
                                project.setDisplayHeight(library.getDisplayHeight() * heigh
                                library.getExpandItem().setExpanded(expanded);
                        }
                        for(InkstoneElementKind kind : library.getInkTypes()) {
                                if( kind.getExpandItem().getExpanded() != expanded ) {
                                        library.setDisplayHeight(kind.getDisplayHeight() *
                                        kind.getExpandItem().setExpanded(expanded);
                                }
                        }
                }
        }
}
```

(d) Componente de GUI no *Ink* (AOM)

Figura 2.3: Exemplos reais de componentes de GUI que utilizam metadados

2.4 Acoplamento entre meta componentes

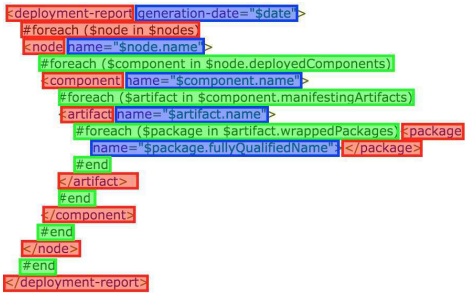
As abordagens que utilizam metadados para desacoplar a GUI do domínio buscam fomentar o reúso, uma vez que os metacomponentes são independentes do domínio e podem ser reutilizados em diversos projetos. Porém a má divisão de responsabilidades nesses meta-componentes dificulta sua manutenção e reduz sua reusabilidade.

A Figura 2.4 demonstra o resultado de uma análise do código fonte de seis sistemas reais que utilizam meta componentes: *templates* de geração de código no *AndroMDA*, no *Redmine*, no *BrowserCMS* e no *Rails*; e componentes de GUI no *SwingBean* e no *Ink*. Nessa análise, as responsabilidades referentes a entidades, propriedades, tipos de propriedades e relacionamentos foram demarcadas com cores diferentes. Pode-se ver como essas responsabilidades estão entrelaçadas e, consequentemente, acopladas.

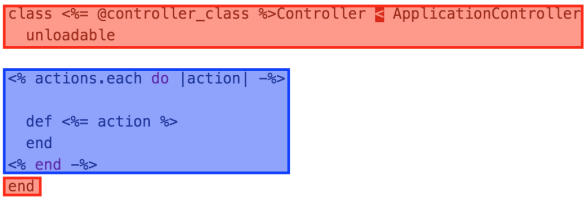
Embora os meta componentes não estejam acoplados ao domínio, pode-se notar o acoplamento entre eles mesmos, portanto sua reusabilidade é comprometida. Dessa forma, é importante que os meta componentes de GUI possuam responsabilidades bem encapsuladas e exclusivamente referentes a entidades, propriedades ou relacionamentos do modelo do domínio.

A Listagem 2.3 exibe metacomponentes de GUI hipotéticos que possuem responsabilidades pequenas e bem definidas. Esse código implementaria a mesma funcionalidade de GUI das Listagens 2.1 e 2.2, isolando a GUI do domínio e encapsulando os próprios meta-componentes. Dessa forma, a GUI se tornaria facilmente reutilizável como, por exemplo, no código das linhas 22 a 31, que mostra uma funcionalidade diferente — relatório de destaque de serviços pelo valor absoluto de vendas — que pôde reutilizar a GUI do relatório anterior sem nenhuma alteração.

Nesse exemplo, o código de GUI foi dividido e organizado em artefatos baseados nos metadados do Modelo do domínio, que é composto por [75, p. 29] [74, p. 54]: entidades ou classes, por exemplo, `ProdutoDTO` e `ServicoDTO`; instâncias ou objetos dessas entidades; propriedades, como `nome`, `delta_vendas`, `titulo`, `tipo` e `vendas`; relacionamentos entre entidades; e operações de entidades. As responsabilidades de relacionamentos e operações não foram cobertas no exemplo da Listagem 2.3, mas serão consideradas no restante deste trabalho.



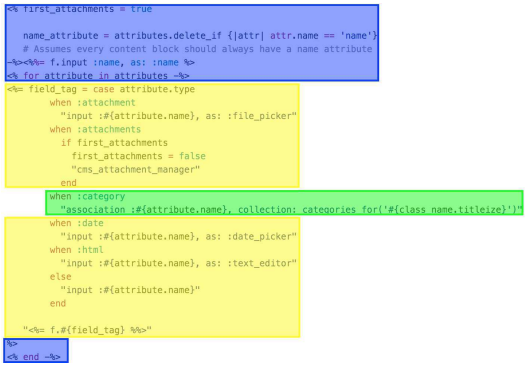
(a) *AndroMDA*



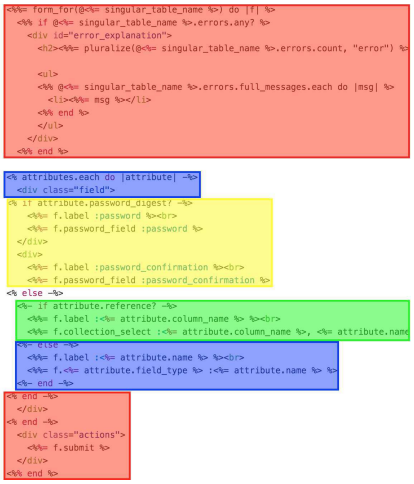
(b) *Redmine*

(c) *SwingBean*

(d) *Ink*



(e) *BrowserCMS*



(f) *Rails*

Figura 2.4: Análise das responsabilidades de metacomponentes reais

Código Fonte 2.3: Metacomponentes de GUI com responsabilidades bem definidas

```
1  def desenha_entidade_em_tabela(lista):
2    resultado = "<table>"
3    for item in lista:
4      resultado +=  GUI.desenha_instancia(item)
5    return resultado + "</table>"
6
7  def desenha_instancia_em_linha(instancia):
8    return "<tr>" + GUI.desenha_propriedades(instancia) + "</tr>"
9
10 def desenha_propriedade_simples_em_celula(propriedade):
11   return "<td>" + propriedade.valor + "</td>"
12
13 def desenha_propriedade_com_cor_em_celula(propriedade):
14   return "<td color="+propriedade.cor+">"+propriedade.valor+"</td>"
15
16 def produtos_destaque_laranja_delta_vendas_dez() :
17   mes_atual = datetime.datetime.now().month
18   lista = produtos_vendas(mes_atual)
19   return GUI.desenhar_lista(lista,
20     [delta_vendas: cor = (delta_vendas > 0.1) ? "orange" : "white"])
21
22 class ServicoDTO:
23   def __init__(self, titulo, tipo, vendas):
24     self.titulo = titulo
25     self.tipo = tipo
26     self.vendas = vendas
27
28 def servicos_destaque_vermelho_vendas_menor_milhao(ano) :
29   lista = servicos_vendas(ano)
30   return GUI.desenhar_lista(lista,
31     [vendas: cor = (vendas < 1000000) ? "red" : "white"])
```

Para esse exemplo se tornar real, seria necessário utilizar o conceito de Inversão de Controle (IoC do inglês *Inversion of Control* [30]), pois os metacomponentes de GUI não seriam invocados diretamente pela aplicação. Em vez disso, deveria existir um arcabouço [83] de GUI para decidir que metacomponente deveria ser invocado a fim de desenhar cada trecho

das telas da aplicação. No código, esse arcabouço é representado pela classe `GUI`, que invoca os métodos:

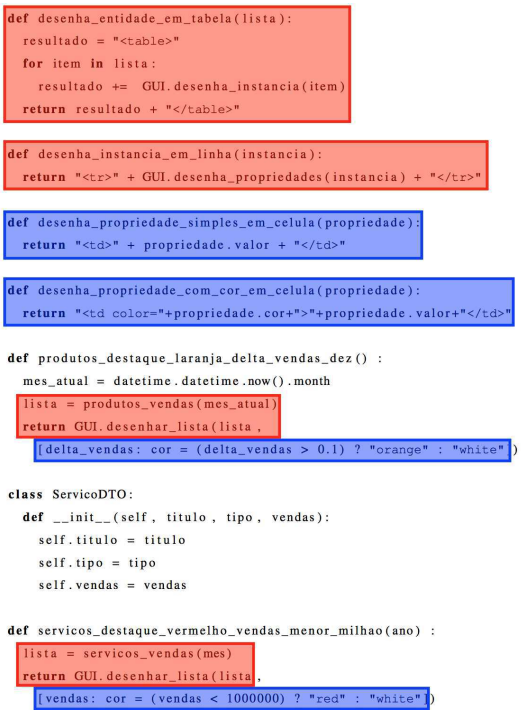
- `desenha_entidade_em_tabela(lista)` para desenhar coleções de objetos em uma tabela;
- `desenha_instancia_em_linha(instancia)` para desenhar cada objeto em uma linha da tabela;
- `desenha_propriedade_simples_em_celula(propriedade)` para mostrar propriedades simples nas células da linha;
- `desenha_propriedade_com_cor_em_celula(propriedade, cor)` para destacar a cor de fundo da célula das propriedades que forem anotadas com uma cor específica (por exemplo, `delta_vendas` e `vendas` nas linhas 20 e 31 respectivamente).

As responsabilidades da Listagem 2.3 foram anotadas visualmente, resultando na Figura 2.5, que apresenta duas visões para o mesmo código. A Figura 2.5a demarca quatro metacomponentes de GUI totalmente independentes do domínio, que são exatamente os quatro métodos citados no parágrafo anterior. Isso foi possível pois o acoplamento que havia da GUI para o domínio, na Figura 2.2a, foi substituído pelo uso de metadados, que estão destacados em amarelo. Sob outro ponto de vista, a Figura 2.5b demonstra que os quatro metacomponentes de GUI possuem responsabilidades bem definidas, sendo que dois deles lidam apenas com responsabilidades de entidade e os outros dois tratam apenas dos aspectos de propriedades.

Com essa abordagem, os artefatos de GUI se tornaram independentes do domínio de Produto e podem ser facilmente reutilizados. Tanto que o código entre as linhas 22 a 31 define uma nova entidade `ServicoDTO`, que reutiliza os artefatos de GUI como eles já estão implementados. Dessa vez, a GUI monta um relatório totalmente diferente, que destaca em vermelho os serviços com venda maior do que um milhão.

No entanto, a Figura 2.5 ainda demonstra acoplamento entre GUI e domínio nos dois pontos onde configurações de GUI específicas são atreladas a duas propriedades do domínio. Uma forma de reduzir esse acoplamento é extrair a configuração do código fonte e externá-la

(a) Código da listagem 2.3 - Marcação de responsabilidades de GUI e Domínio em códigos com metadados



(b) Código da listagem 2.3 - Marcação de responsabilidades bem definidas de entidade e propriedade

Figura 2.5: Análise das responsabilidades de metacomponentes hipotéticos

em arquivos de configuração ou bancos de dados. Elimina-se, dessa forma, completamente o acoplamento GUI – domínio do negócio e se viabiliza o reúso de GUI.

Neste capítulo, várias abordagens para implementação de GUI em aplicações corporativas foram analisadas, com o intuito de reduzir o acoplamento de entre o código de GUI e domínio, aumentando o reúso de código de GUI. Após demarcar visualmente as responsabilidades no código de sistemas feitos com essas abordagens, verificou-se que o reúso eficaz de GUI pode ser obtido se o código de GUI atender a dois requisitos: utilizar referências para metadados do domínio e possuir responsabilidades pequenas e bem definidas. No próximo capítulo descrevem-se as características das abordagens mais relacionadas à solução proposta nesta tese, dentro do escopo do desenvolvimento de GUI baseada em metadados para aplicações corporativas.

Capítulo 3

Trabalhos relacionados

Neste capítulo apresentam-se várias abordagens para geração de GUI a partir de metadados, a fim de compará-las com a solução proposta nesta tese. Para cada abordagem, foram pesquisados trabalhos e tecnologias utilizadas para geração de GUI baseada em metadados no contexto das aplicações corporativas. Para as abordagens em que não se encontrou exemplos de geração específica para GUI, foram propostas e analisadas soluções hipotéticas.

As abordagens analisadas neste capítulo são: programação orientada a aspectos, desenvolvimento dirigido por modelos, *Scaffolding*, *Adaptive object model* e arcabouços baseados em metadados.

3.1 Programação orientada a aspectos

A Programação orientada a aspectos (POA) [54, 53] é uma técnica utilizada para lidar com conceitos transversais (*crosscutting concerns*), que são responsabilidades cujo código se espalharia em muitas classes de um sistema orientado a objetos. POA provê um mecanismo para encapsular uma responsabilidade transversal em apenas um artefato que pode ser costurado (*weaving*) em um código base já existente.

Como pode ser visto na Figura 3.1, POA assume que o código base de um sistema possui pontos de junção (*join point*), que são locais onde o código base pode ser estendido para executar as funcionalidades transversais. As estruturas de *pointcut* selecionam alguns pontos de junção. Por fim, os aspectos (*aspect*) descrevem as funcionalidades transversais (encapsuladas nos *advices*) que serão executadas para cada *pointcut*.

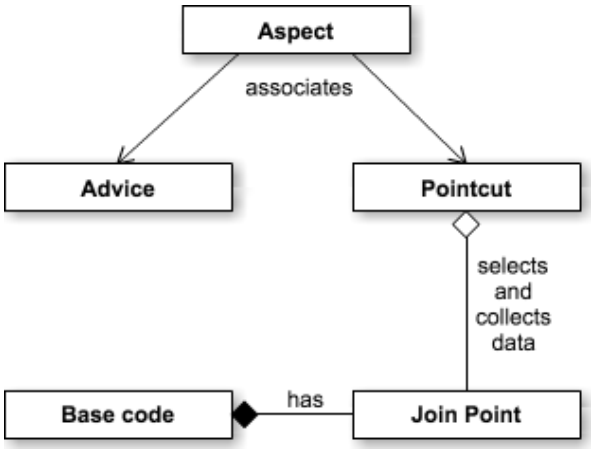


Figura 3.1: Principais conceitos da Programação orientada a aspectos

Um dos exemplos clássicos de POA é o encapsulamento da funcionalidade de *logging*. Nesse caso, pode-se definir um aspecto para que: antes de todos os métodos (*join point*), de todas as classes do pacote de domínio (*pointcut*), seja escrito um *Log* no nível de `trace` contendo o nome do método que será executado (*advice*). Com apenas esse artefato, um desenvolvedor pode implementar a funcionalidade de *logging* em todas as classes da camada de domínio.

Dessa forma, POA é uma abordagem poderosa para melhorar o projeto de sistemas. Porém, no contexto do acoplamento entre as camadas de GUI e domínio, não visualizamos grandes melhorias apenas com o uso de Aspectos. Obtivemos essa conclusão após projetar três soluções baseadas POA para reduzir esse acoplamento.

Na primeira tentativa, definimos que o código base representaria a camada de domínio e os *advices* implementariam a GUI. Dessa forma, o acoplamento GUI–domínio estaria apenas nas definições dos aspectos e *pointcuts*. Porém, essa solução não é viável, pois o código base deve ser uma aplicação executável. Os *advices* não têm vida própria, dado que são invocados ao se atingir os *pointcuts*. Portanto, a GUI não pode ser completamente implementada em *advices*. Mesmo que seja extensível, a GUI precisa fazer parte do código base, portanto estará acoplada ao domínio.

Outra alternativa seria definir o código duplicado da GUI através de aspectos. Por exemplo, um aspecto estabeleceria que todos as propriedades do tipo *string* devem ser representadas por `TextFields`. No entanto, a maior parte das soluções de GUI utilizam *templates* ou composição dinâmica de componentes para implementar telas. A definição de pontos de

junção é difícil nessas abordagens, pois POA geralmente usa métodos ou tratamento de exceções para *join points*. Até onde compreendemos POA, não encontramos formas de definir pontos de junção em *templates* ou *composites* de GUI.

Por fim, dado que exista uma GUI implementada através de um código base, através de métodos que possuam pontos de junção facilmente identificáveis, pode-se usar POA para estender as funcionalidades em GUI normais (sem metadados). Porém, a GUI normal já é acoplada ao domínio e os aspectos não removeriam esse acoplamento.

Além de projetar o uso de POA para reduzir o acoplamento GUI–Domínio, executamos diversas buscas no *Google Scholar*, por artigos sobre o uso de Aspectos para implementar GUI, e não encontramos resultados relevantes que pudessem ser comparados com a solução proposta nesta tese.

3.2 Desenvolvimento dirigido por modelos

O Desenvolvimento dirigido por Modelos, mais conhecido como MDD do inglês *Model driven development*, é uma abordagem baseada na transformação automática de modelos em código fonte [63]. MDD parte do princípio que existem muitas tecnologias e plataformas com as quais pode-se implementar um mesmo sistema. Portanto é preciso ter um modelo abstrato da aplicação independente da plataforma (PIM - *Platform Independent Model*), que é transformado em um modelo específico de plataforma (PSM - *Platform Specific Model*), o qual, por usa vez, é transformado em código fonte.

Em vez de tratar os modelos como meros artefatos de documentação, MDD os utiliza como entidades de primeira classe no desenvolvimento de software [69]. Assim sendo, as transformações PIM > PSM e PSM > código podem ser compreendidas como metacomponentes reutilizáveis, inclusive para gerar código de GUI em aplicações corporativas [94, 95, 61]. Todavia, os componentes do PIM e do PSM são baseados em muitos elementos da linguagem UML, o que os torna demasiadamente gerais e complexos. Dessa forma, a implementação e a reutilização de transformações difícil.

Diversos trabalhos baseados em MDD tratam a construção de GUI baseada em modelos [68, 50, 94, 5, 88]. Alguns dos conceitos para reusabilidade definidos por esses trabalhos são parecidos com a abordagem proposta nesta tese, sendo inclusive citados nos usos conhe-

cidos dos padrões propostos no capítulo 5. Todavia, a complexidade dos modelos de PIM e PSM dificulta a manutenibilidade das técnicas baseadas em MDD.

Silva apontou lacunas na customização do processo de geração UI baseada em transformações [23]. Porém existem abordagens, como *TransformiXML* [90], que expõem o seu processo de transformação explicitamente. No entanto, a customização desse processo de transformação é muito complexo, exigindo que o desenvolvedor de GUI seja um especialista em MDD [7].

Alguns trabalhos em MDD focaram no reúso de regras ou padrões, que são elementos pequenos [56, 11, 58]. Porém, em sistemas reais é necessário haver mecanismos para reúso de elementos maiores.

A solução proposta nesta tese é mais mais simples que MDA, pois não tem o intuito de gerar artefatos independentes de tecnologia. O modelo conceitual é mapeado, pelos metacomponentes de GUI, diretamente para telas executáveis em alguma tecnologia específica. Nessa caso, perde-se em generalidade em relação a MDA, mas ganha-se em facilidade de implementação dos metacomponentes.

3.3 *Scaffolding*

O crescimento das aplicações *web* tornou o desenvolvimento de software mais complexo, devido ao surgimento de novos requisitos não-funcionais como: escalabilidade, segurança, responsividade nas telas, internacionalização, tolerância a falhas e balanceamento de carga. Dessa forma, o desenvolvedor precisa escrever muito código e definir muita configuração antes de conseguir executar algum pedaço da aplicação.

Na última década, diversos arcabouços surgiram para auxiliar o desenvolvimento de aplicações *web*. Uma técnicas utilizada por essas ferramentas é chamada de *Scaffolding* [92] e consiste em gerar automaticamente o código e a configuração inicial da aplicação, a partir dos metadados do modelo conceitual. Desse modo, o programador pode executar alguma funcionalidade da aplicação, mesmo que primitiva, após poucos minutos de desenvolvimento.

Além da geração de código, os arcabouços de *Scaffolding* geralmente utilizam a abordagem *Convention over configuration* [18], fazendo com que parte do código gerado siga uma convenção padrão e funcional que pode ser alterada através de configuração. Por exemplo,

convenciona-se que o nome de uma tabela relacional deve ser o mesmo da sua respectiva classe de domínio. Como, na maioria dos casos, os nomes serão iguais, o desenvolvedor pode ignorar a tarefa de mapeamento entre domínio e o banco de dados. Porém, se os nomes forem diferentes, o desenvolvedor deve usar um arquivo de configuração para informar o nome da tabela.

Esse mesmo conceito é utilizado diversos pontos das aplicações *web*, portanto muitas das tarefas braçais no desenvolvimento de software podem ser ignoradas. Essa abstração não representa perda na flexibilidade dos arcabouços *web*, dado que o código ou a configuração que divergir do normal pode ser alterada através de configuração.

No entanto, o código de interface gráfica e lógica do domínio que é gerado pelos arcabouças de *Scaffolding* costuma ser apenas um ponto inicial para a implementação de aplicações *web*, sendo substituído no decorrer dos projetos de desenvolvimento [92].

Alguns arcabouços de *Scaffolding* geram internamente o código das aplicações *web* (como *Spring Roo* [81, 17] e *Play*[1]), enquanto outras ferramentas expõem os *templates* de geração de código — *Ruby on Rails* [78, 79], *Grails*[2] [84] e *Django*[3] [47, 28] — permitindo a alteração do processo de geração de código.

Nesses caso, os *templates* atuam como metacomponentes de GUI, que são independentes dos domínios de cada aplicação *web*. Porém, como pode ser visto em alguns exemplos reais[4], os *templates* misturam diversas responsabilidades diferentes na geração de GUI, se tornando acoplados entre si e reduzindo sua reusabilidade.

Outra opção seria a alteração do código após a sua geração. No entanto foi observado que a quantidade de código de GUI duplicado, para *Ruby on Rails* e *Spring Roo*, cresce linear ou quadraticamente em relação ao tamanho do modelo do domínio [6], o que reduz a qualidade do código e dificulta a sua manutenção.

[1] *Play framework* - Geração de código para CRUD: https://goo.gl/KQJ0wT

[2] Manual do comando `grails install-templates`: http://goo.gl/9oRSNE

[3] Geração de código de formulários em *Django*: https://goo.gl/G36Jpt

[4] *Templates* padrão de Rails: https://goo.gl/CxVzPw e https://goo.gl/u60tjM e https://goo.gl/p0ryDt

3.4 *Adaptive object model*

O estilo arquitetural *Adaptive Object Model* (AOM) [103] é uma solução para sistemas com intensa demanda para adaptação, pois permite alterar a estrutura do software (metadados) em tempo de execução, uma vez que se trata de uma solução baseada na interpretação de modelos em vez de geração de código.

Welicki et al. definam três padrões de renderização a fim de organizar o código de interface gráfica em sistemas AOM [99]. Esses padrões encapsulam as responsabilidades de GUI baseados nos metadados do domínio. O padrão de granularidade mais fina, `Property Renderer`, define classes para desenhar GUI referente tipos específicos de propriedades em certos contextos. Por exemplo, pode existir uma classes para desenhar todos os campos do tipo *String* no contexto de edição e outras classe para desenhar todos os campos do tipo *Date* no contexto de visualização. As classes dos padrões `Entity View` e `Dynamic View` coordenam um conjunto de `Property Renderers` com a finalidade de desenhar GUI para entidades completas. `Entity Views` produz GUI para apenas uma instância, por exemplo, formulários de criação, e `Dynamic Views` desenha GUI para várias instâncias, como tabelas de listagem.

A Figura 3.2 [99] apresenta alguns exemplos de fragmentos de GUI desenhados por `Entity Views` e `Property Renderers`. As classes participantes desses padrões podem ser compreendidas como metacomponentes de GUI, totalmente independentes do domínio de cada aplicação onde são utilizados, pois referenciam apenas os metadados do domínio. Sendo, portanto, facilmente reutilizadas em aplicações diferentes.

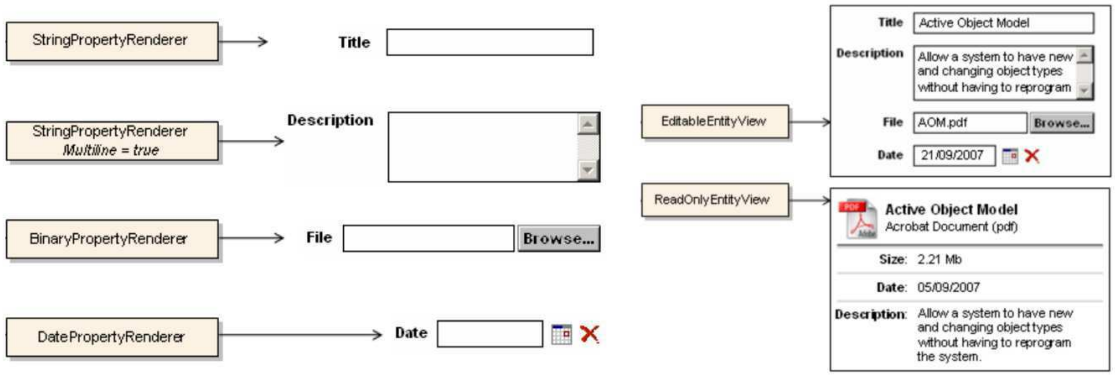


Figura 3.2: Exemplos de *property renderers* e *entity views*

De fato, os padrões de renderização AOM são a abordagem que mais se aproxima da solução proposta nesta tese, pois, além de definir metacomponentes de GUI, estabelece responsabilidades bem delimitadas para GUI de propriedades e entidades. No entanto, existem algumas características nesses padrões que dificultam a evolução da GUI.

Em primeiro lugar, os novos componentes de GUI devem estender as classes bases de `Property Renderer`, `Entity View` e `Dynamic View`. Dessa forma, os padrões de renderização definem um arcabouço caixa branca, que, segundo Roberts e Johnson [83], demanda mais esforço para introduzir novos elementos do que os arcabouças caixa preta. Isso ocorre porque os arcabouços caixa branca são baseados em herança, enquanto arcabouços caixa preta podem ser facilmente configurados e estendidos por regras de composição.

Além disso, nos padrões de renderização as classes `Dynamic View` possuem a responsabilidade de compor `Entity Views` e `Property Renderers`. Semelhantemente, as classes de `Entity View` devem compor as de `Property Renderers`. Essa responsabilidade gera acoplamento entre os metacomponentes de GUI e reduz sua reusabilidade. Para evitar esse problema, a composição dos metacomponentes de GUI deveria ser feita em classes externas, possivelmente fábricas ou interpretadores.

Outra característica limitante dos padrões de renderização é a ausência de um padrão específico para lidar com os metadados dos relacionamentos entre as entidades do domínio. Em vez disso, os autores sugerem que se reutilize `Property Renderers` para desenhar os relacionamentos na GUI. Novamente, essa solução gera acoplamento de responsabilidades distintas e reduz as possibilidades de reúso dos metacomponentes de GUI.

Por fim, não foram encontrados trabalhos acadêmicos ou exemplos reais de sistemas que detalhassem a implementação dos padrões de renderização. Esses detalhes são importantes para compreender claramente a composição e a comunicação entre as classes que participam dos padrões. O exemplo mais concreto de AOM encontrado nesta pesquisa foi o arcabouço *ModelTalk* [44, 43], posteriormente renomeado para *Ink* e disponibilizado com código aberto[5]. Todavia, esse projeto não implementa a geração de código de GUI.

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

3.5 Arcabouços baseados em metadados

Guerra desenvolveu um vasto trabalho com objetivo de catalogar padrões para arcabouços baseados em metadados. Esses arcabouços utilizam os metadados das classes em tempo de execução a fim de processar sua lógica [39].

Para definir uma linguagem de padrões para arcabouços baseados em metadados, foram analisados 11 desses arcabouços que eram utilizados na indústria [38]: *Hibernate*, *Swing-Bean*, *JAXB API*, *XapMap*, *ACE*, *MentalLink*, *MetadataSharing*, *Hibernate Validator*, *JColtrane*, *JBoss Application Server* e *Esfinge*. Dessa forma, foram identificados oito padrões de projeto que servem como referência para implementação da estrutura interna de novos arcabouços baseados em metadados.

Em um trabalho posterior, um arcabouço baseado em metadados foi criado para validar um modelo de geração de interface gráfica a partir dos metadados de classes [9]. Esse modelo, chamado MAGIU, busca resolver três tipos de inconsistências no código de GUI: inconsistência entre o código de GUI e o código do domínio; inconsistência na forma como os dados são exibidos; e inconsistência entre telas diferentes.

O modelo MAGIU define *templates* específicos para classes, coleções de objetos e propriedades. Dessa forma, permite a criação de metacomponentes de GUI com responsabilidades bem definidas. Todavia, esse modelo gera código apenas para as telas de edição e visualização de objetos. Faltam metacomponentes para desenhar, por exemplo, tabelas de listagem, relatórios e relacionamentos.

Guerra et. al. também definiram quatro padrões arquiteturais para arcabouços baseados em metadados [40], dos quais dois tem algo a ver com a solução proposta nesta tese. O padrão *Entity Mapping* permite o mapeamento de classes do domínio em outras formas de representá-las, por exemplo, nas telas da GUI. E o padrão *Metadata-based Graphical Component* usa metadados para gerar componentes gráficos.

De fato, os arcabouços baseados em metadados podem ser uma forma concreta para implementar a solução que é proposta nesta tese. Alguns dos trabalhos atuais, nesta abordagem, possuem as características importantes para reúso de GUI — independência de domínio e responsabilidades bem definidas — porém não geram todo o código necessário para GUI de aplicações corporativas, como, por exemplo, tabelas de listagem, relatórios e relacionamen-

tos.

No capítulo 6, um arcabouço baseado em metadados do domínio da aplicação foi implementado para validar a linguagem de padrões proposta nesta tese. Dessa forma, muitas das contribuições relatadas nesta Sub seção foram reutilizadas nesta tese.

3.6 Considerações finais do capítulo

Após analisar cinco abordagens para implementar interface gráfica baseada em metadados para aplicações corporativas, foram encontradas contribuições relevantes que podem ser combinadas para produzir a solução proposta nesta tese. De *Scaffolding*, pode-se aproveitar o conceito de *Convention over Configuration* e a geração de código baseada em *templates*. AOM define padrões com responsabilidades bem definidas para GUI de entidades e propriedades. Por fim, os arcabouços baseados em metadados representam uma estrutura concreta para implementar a solução proposta nesta tese.

A partir do estado-da-arte descrito neste capítulo, inovações serão propostas a fim de criar uma abordagem que melhore a reutilização de GUI, de forma que seja totalmente baseada nos metadados do domínio, gere todo o código da GUI e possua responsabilidades bem definidas.

Capítulo 4

Linguagem de padrões - Catálogo

Após analisar diversas abordagens para criação de GUI independente do domínio, foi construída uma linguagem de padrões, composta por sete padrões de projetos e dois padrões arquiteturais, que reforça as características positivas das abordagens analisadas no capítulo 3, mas também propõe soluções para as limitações dessas abordagens. Essa linguagem de padrões representa a maior contribuição deste trabalho de pesquisa e a sua descrição está dividida entre o capítulo 4, que cataloga cada padrão isoladamente, e o capítulo 5, que detalha os relacionamentos entre os padrões e apresenta uma arquitetura de referência. Uma versão preliminar desse catálogo de padrões foi publicada no EuroPLoP 2015 [98].

Neste trabalho, a estrutura utilizada para descrever os padrões se baseia na forma utilizada por Gamma et. al. para os padrões de projeto em sistemas orientados a objetos [35]. Portanto, os padrões definidos aqui possuem a seguinte estrutura: título e sinônimos em inglês; texto inicial com contexto e exemplos do problema; definição do problema em forma de pergunta; lista de forças presentes no contexto do problema; detalhamento da solução; detalhes de implementação (presente apenas quando mais detalhes são necessários); lista de consequências do padrão; usos conhecidos do padrão; e padrões relacionados.

4.1 Visão geral dos padrões

A abordagem proposta neste trabalho viabiliza o encapsulamento das responsabilidades de interface gráfica de aplicações corporativas, em artefatos de granularidade fina com base nos metadados do modelo do domínio. A listagem a seguir apresenta os nove padrões que

cooperam a fim de cumprir o objetivo da abordagem proposta:

1. Padrão de projeto DOMAIN WIDGET: representa a generalização de um componente comum de GUI. Em vez de referenciar elementos do Modelo do domínio, os *widgets* manipulam apenas metadados;

2. Padrão de projeto DOMAIN MODEL X RAY: fornece os metadados do domínio em uma fachada única para os widgets montarem a GUI;

3. Padrão de projeto INDIRECT WIDGET: remove o acoplamento entre *widgets*, utilizando portas em vez de referências diretas entre eles;

4. Padrão de projeto WIDGET ENGINE: implementa inversão de controle nos *indirect widgets*, decidindo quando instanciá-los, como ligar suas portas e quando invocá-los para desenhar a GUI;

5. Padrão arquitetural SERVER RENDERING: executa os *widgets* na camada servidor *web* para construir a GUI de aplicações corporativas;

6. Padrão arquitetural CLIENT RENDERING: executa os *widgets* na camada cliente (navegadores *web*) para construir a GUI de aplicações corporativas;

7. Padrão de projeto WIDGET SCOPE: diferencia *widgets* padrão e *widgets* que são específicos para alguns fragmentos da GUI;

8. Padrão de projeto PARAMETRIZABLE WIDGET: modifica o comportamento de um *widget* toda vez que ele é ligado a alguma porta ou ramo de porta;

9. Padrão de projeto RELATIONSHIP RENDERER: define a estrutura de componentes de GUI baseado em metadados de relacionamentos;

Os nove padrões serão detalhados nas próximas seções, e os seus relacionamentos dentro de uma liguagem de padrões serão descritos no capítulo 5.

4.2 Padrão de projeto DOMAIN WIDGET

Sinônimos: METADATA COMPONENT, METACOMPONENT

Em aplicações corporativas, os componentes de GUI mais simples podem ser facilmente reutilizados [29], pois o seu código geralmente aponta para apenas uma propriedade do modelo do domínio, sendo necessário apenas alterar a referência à propriedade cada vez que o componente é reutilizado. Na Listagem 4.1 descreve-se um exemplo de reutilização de propriedades simples no sistema para comércio eletrônico de código aberto *Magento*[1].

Nesse exemplo, dois componentes de GUI — `label` e `input` — foram reutilizados três vezes cada um. O que muda em cada um dos usos é apenas a propriedade que é referenciada (`name`, `email` e `telephone`).

Código Fonte 4.1: Exemplos de componentes de GUI simples

```
1  <label class="label" for="name"><span><?php echo __('Name') ?>
2    </span></label>
3  <input name="name" id="name" title="<?php echo __('Name') ?>" />
4  <label class="label" for="email"><span><?php echo __('Email') ?>
5    </span></label>
6  <input name="email" id="email" title="<?php echo __('Email') ?>" />
7  <label class="label" for="telephone"><span>
8    <?php echo __('Phone Number') ?></span></label>
9  <input name="telephone" id="telephone"
10   title="<?php echo __('Phone Number') ?>" />
```

Porém os componentes de GUI mais complexos, como formulários, tabelas de listagem e relatórios, possuem diversas referências ao domínio e se tornam mais acoplados e menos reutilizáveis. Para ilustrar esse cenário, na Listagem 4.2 repete-se o exemplo no *Magento*, mostrando o código do formulário como um todo. Nesse caso, o formulário trata os dados da entidade `contact` fazendo referências diretas ao domínio nas linhas 1, 3, 5, 6, 8 - 14 e 16. Se houver a demanda de criação de um formulário semelhante para outra entidade, o código deve ser copiado e todas essas referências precisam ser alteradas.

Em resumo, os componentes de GUI simples, que lidam apenas com propriedades podem ser facilmente reutilizados, porém os componentes mais complexos, que lidam com entida-

[1] https://goo.gl/DO2ph0

des ou relacionamentos, possuem muito acoplamento ao domínio e são menos reutilizáveis.

Problema

Como eliminar o acoplamento da camada de GUI para o modelo do domínio, a fim de construir componentes de GUI complexos e reutilizáveis?

Código Fonte 4.2: Exemplos de componentes de GUI complexos

```
1  <form class="form contact" method="post"
2        action="<?php echo $block->getFormAction(); ?>">
3    <label class="label" for="name"><span><?php echo __('Name') ?>
4      </span></label>
5    <input name="name" id="name" title="<?php echo __('Name') ?>" />
6    <label class="label" for="email"><span><?php echo __('Email') ?>
7      </span></label>
8    <input name="email" id="email"
9      title="<?php echo __('Email') ?>" />
10   <label class="label" for="telephone"><span>
11     <?php echo __('Phone Number') ?></span></label>
12   <input name="telephone" id="telephone"
13     title="<?php echo __('Phone Number') ?>" />
14   <label class="label" for="comment"><span>
15     <?php echo __('What is on your mind?') ?></span></label>
16   <textarea name="comment" id="comment"
17     title="<?php echo __('What is on your mind?') ?>"></textarea>
18   <button type="submit" title="<?php echo __('Submit') ?>">
19     <?php echo __('Submit') ?>
20   </button>
21 </form>
```

Forças

- Referências diretas ao modelo do domínio inibem a reutilização de componentes de GUI;
- Referências baseadas em metadados eliminam o acoplamento ao domínio, porém são mais complexas que as referências diretas.

Solução

Utilizar referências baseadas em metadados, limitando os metadados do domínio a entidades, propriedades e relacionamentos, a fim de simplificar a criação de metacomponentes de GUI.

Na Listagem 4.3 descreve-se um exemplo de código para desenhar o formulário do *Magento* que foi apresentado nas duas listagens anteriores, substituindo as referências diretas ao modelo de domínio (`contact`, `name`, `email`, `telephone` e `comment`) por referências baseadas em metadados. Observar as *tags* começando com `<?meta`, que utilizam elementos como `Entity`, `Property`, `Name`, `Label` e `IsLong`. Dessa forma, formulários semelhantes podem ser desenhados para entidades diferentes através desse mesmo metacomponente de GUI.

Código Fonte 4.3: Exemplo de um metacomponentes de GUI complexo

```
1  <form class="form <?meta Entity.Name ?>" method="post"
2        action="<?php echo $block->getFormAction(); ?>">
3  <?meta for Property in Entity.Properties ?>
4    <label class="label" for="<?meta Property.Name ?>"><span>
5      <?meta Property.Label ?></span></label>
6    <?meta if Property.IsLong ?>
7      <textarea name="comment" id="comment"
8        title="<?php echo __('What is on your mind?') ?>"> </textarea>
9    <?meta else ?>
10     <input name="<?meta Property.Name ?>" id="<?meta Property.Name ?>"
11       title="<?meta Property.Label ?>" />
12   <?meta endif ?>
13 <?meta endfor ?>
14   <button type="submit" title="<?php echo __('Submit') ?>">
15     <?php echo __('Submit') ?>
16   </button>
17 </form>
```

Detalhes de implementação

Na Figura 4.1 evidencia-se a diferença entre os componentes comuns de GUI (`aComponent`), que referenciam diretamente o domínio específico de cada aplicação corporativa, e os metacomponentes de GUI (`aWidget`), que conhecem apenas os metadados dos domínios. Nesse caso, o objeto `aComponent` pode ser utilizado apenas para a entidade `OrderItem`, enquanto o objeto `aWidget` pode desenhar diversas propriedades até de domínios diferentes. Assim sendo, os metacomponentes de GUI podem, em teoria, ser reutilizados mais facilmente do que os componentes comuns de GUI.

Com a finalidade de simplificar os nomes dos padrões e os diagramas nesta tese, nomeamos os metacomponentes de GUI como *Widgets*, um termo que já é comumente utilizado para elementos de GUI [64].

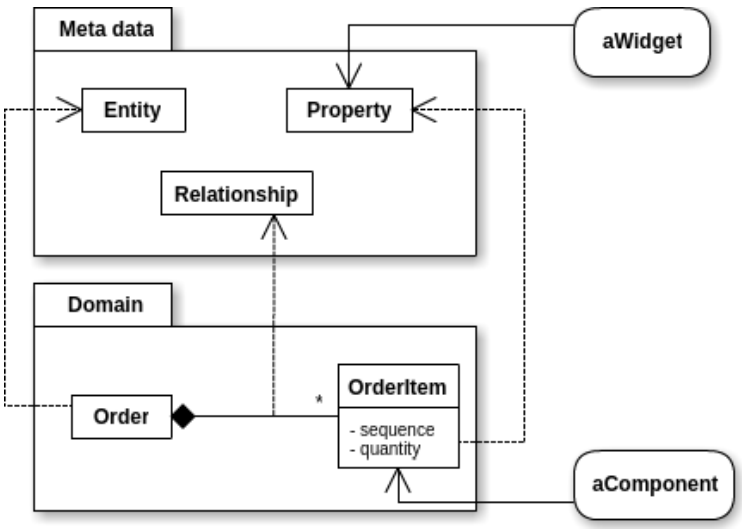


Figura 4.1: *Widgets* e Componentes

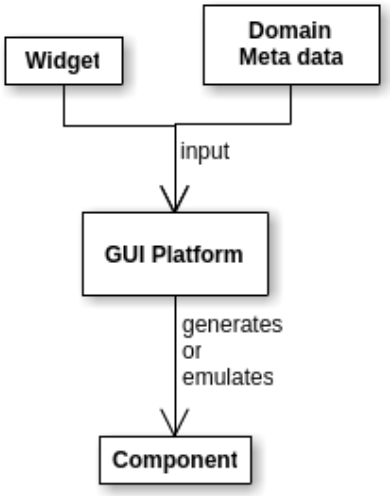


Figura 4.2: Metadados em GUI

O processo de funcionamento de um *widget* está descrito na Figura 4.2 e no pseudocódigo da Listagem 1.1. Inicialmente, a plataforma de GUI é invocada para iniciar a renderização de uma tela (linhas 22 a 25), recebendo como parâmetros de entrada os *widgets* (definidos nas linhas 2 a 13) e os metadados do domínio (descritos entre as linhas 16 e 19). O resultado desse processo assume a forma de um componente de GUI (linhas 28 a 32), que pode ter o código gerado ou a funcionalidade emulada (interpretada em tempo de execução).

Segundo esse processo, os *widgets* podem ser reutilizados em domínios diferentes, sem modificação no seu código fonte, produzindo a GUI necessária para aplicações corporativas

distintas. É importante que os *widgets* possuam responsabilidades bem definidas e sejam facilmente alteráveis para que o reúso de GUI seja efetivo. Esse é o objetivo da linguagem de padrões que é proposta neste trabalho.

Consequências

- Ao eliminar as referências diretas ao modelo do domínio, *widgets* complexos podem ser reutilizados em várias telas de aplicações corporativas;
- Para evoluir a GUI de aplicações corporativas, o desenvolvedor precisará compreender as estruturas baseadas em metadados, que são complexas. Parte desse trabalho pode ser reduzido se a plataforma de GUI disponibilizar ferramentas para suporte ao desenvolvedor, como pré visualizador de *widgets*, depurador, testes automatizados, auto complementação e biblioteca para publicação e reúso de *widgets*.

Usos conhecidos

Diversas abordagens utilizam metadados, em vez de referências diretas ao domínio, na interface gráfica:

- Arcabouços de *Scaffold* (*Ruby on Rails*, *Grails* e *Django*) utilizam *templates* baseados em metadados na GUI;
- Os padrões de renderização [99] definem renderizadores que interpretam metadados em tempo de execução a fim de desenhar as telas de aplicações AOM;
- O arcabouço baseado em metadados TERESA [68] usa metacomponentes de GUI baseados nos tipos das propriedades.

Padrões relacionados

DOMAIN WIDGET é uma especialização do padrão METADATA-BASED GRAPHICAL COMPONENT [40], na qual se trata apenas os metadados do domínio relativos a entidades, propriedades e relacionamentos.

Os *widgets* propostos neste padrão podem ser implementados como TEMPLATE VIEWS [31], embarcando marcadores em trechos de páginas HTML.

PROPERTY RENDERER, ENTITY VIEW e DYNAMIC VIEW [100] são especializações de DOMAIN WIDGET, que tratam com metadados de propriedades, instância única de uma entidade e lista de instâncias de uma entidade, respectivamente.

RELATIONSHIP RENDERER também é uma especialização de DOMAIN WIDGET, que pode ser usada para encapsular as responsabilidades de GUI referentes aos relacionamentos das entidades do domínio.

INDIRECT WIDGET reduz o acoplamento entre os metacomponentes de GUI propostos neste padrão, além de corrigir a má divisão de responsabilidades entre eles.

4.3 Padrão de projeto DOMAIN MODEL X-RAY

Sinônimos: METADATA SERVICE, DOMAIN DESCRIPTOR

Quando a interface gráfica de aplicações corporativas utiliza metadados em vez de referências diretas ao domínio, o reúso de GUI pode ser incrementado. Como pode-se ver na Figura 4.2, os *widgets* precisam ser combinados com os metadados do domínio para produzir os componentes concretos de GUI.

No entanto, a extração de metadados do domínio não é trivial nem padronizada, podendo variar dependendo da tecnologia utilizada para implementar cada aplicação corporativa. Se os *widgets* conhecerem o mecanismo como os metadados são obtidos, se tornarão acoplados a esses mecanismos.

Na Listagem 4.4, a lógica do *widget* `formulario_simples` está entrelaçada com a extração de metadados do domínio através da API de introspecção da linguagem Python[2]. Consequentemente, se a forma de obtenção de metadados mudar, esse *widget* será impactado.

Código Fonte 4.4: Exemplo de componente com obtenção de metadados

```
1 def formulario_simples(entidade):
2   return "<form action='/api/" + entidade.__class__.__name__ + "'>"
3     + GUI.desenharPropriedades( dir(entidade) ) + "</form>"
```

[2]http://www.diveintopython.net/power_of_introspection/

Problema

Como fornecer os metadados do modelo do domínio para a camada de interface gráfica?

Forças

- O modelo de domínio pode ser implementado em diversas abordagens diferentes: Orientação a objetos, TYPE SQUARE [102], TRANSACTION SCRIPT [31], REST [26], entre outros;
- O acesso aos metadados do modelo de domínio pode ser feito de várias formas diferentes: através de reflexão direta nas classes do modelo; com auxílio de arquivos de configuração; com auxílio de anotações;
- Uma plataforma para o desenvolvimento de GUI não deve conhecer os detalhes do modelo de domínio nem do acesso aos metadados, a fim de que seus artefatos possam ser reutilizados em aplicações com domínio implementado em diferentes tecnologias.

Solução

Expandir a interface da camada de domínio, adicionando serviços padrão para consulta dos metadados do modelo do domínio. Essa interface deve esconder a tecnologia utilizada para implementar o domínio e o mecanismo usado para obtenção de metadados.

A camada de domínio das aplicações corporativas deve prover serviços para as camadas clientes. Esses serviços, em geral, atendem às demandas das interfaces gráficas e dos sistemas externos. Porém, para viabilizar o processo de geração de GUI a partir de metacomponentes, o domínio também deve prover serviços para consulta dos metadados do domínio, fornecendo informações sobre as entidades (nome), as propriedades (tipo, nome e validação) e os relacionamentos (nome e cardinalidade) do domínio.

Esses serviços devem atuar como uma fachada que encapsula a tecnologia na qual o domínio foi implementado e a tecnologia de acesso aos metadados.

Detalhes de implementação

A consulta de metadados do domínio pode ser feita através de duas funções. A primeira retornaria os metadados básicos de todas as entidades pertencentes ao modelo do domínio, viabilizando a construção de GUI para a listagem dos módulos do sistema. Por exemplo, se uma aplicação possui três módulos (Cliente, Produto e Venda), no menu principal da GUI haveria três itens, um para cada abrir cada módulo desses.

A segunda função retornaria todos os metadados detalhados de uma entidade, com a estrutura das propriedades e dos relacionamentos. Essa informação poderia ser usada para desenhar os detalhes do módulo relativo a essa mesma entidade, quando o usuário final requisitar sua abertura.

Se essas duas funções estiverem definidas na mesma fachada, pode-se abstrair os detalhes das tecnologias utilizadas para implementar o domínio e o acesso aos metadados. Todavia, não se indica juntar a fachada de metadados com a fachada de dados operacionais, pois o usuário final e a camada GUI são clientes diferentes e é melhor que cada um possua sua própria fachada.

A Figura 4.3 demonstra os participantes do padrão DOMAIN MODEL X-RAY, marcados em amarelo. Dado que uma plataforma de GUI baseada em metadados combina *widgets* e metadados do domínio a fim de produzir a GUI que vai invocar o domínio. A interface `DomainMetadataReader` define dois métodos, para listar as entidades do domínio e detalhar cada entidade. Além disso, essa interface encapsula a estratégia que está sendo utilizada para leitura dos metadados do domínio e a própria tecnologia utilizada para implementar o domínio.

Desse modo, a classe `ReflectionReader`, que utilizaria reflexão para ler os metadados do domínio implementado em uma linguagem orientada a objetos, pode ser facilmente substituída por outra classe que, por exemplo, leria os metadados em arquivos de configuração.

Após a leitura dos metadados, é preciso utilizar uma forma invariante para enviar esses dados à plataforma de GUI. Para tanto, pode-se aplicar o padrão DATA TRANSFER OBJECT [31], como pode ser visto no pacote `Metadata transfer objects`.

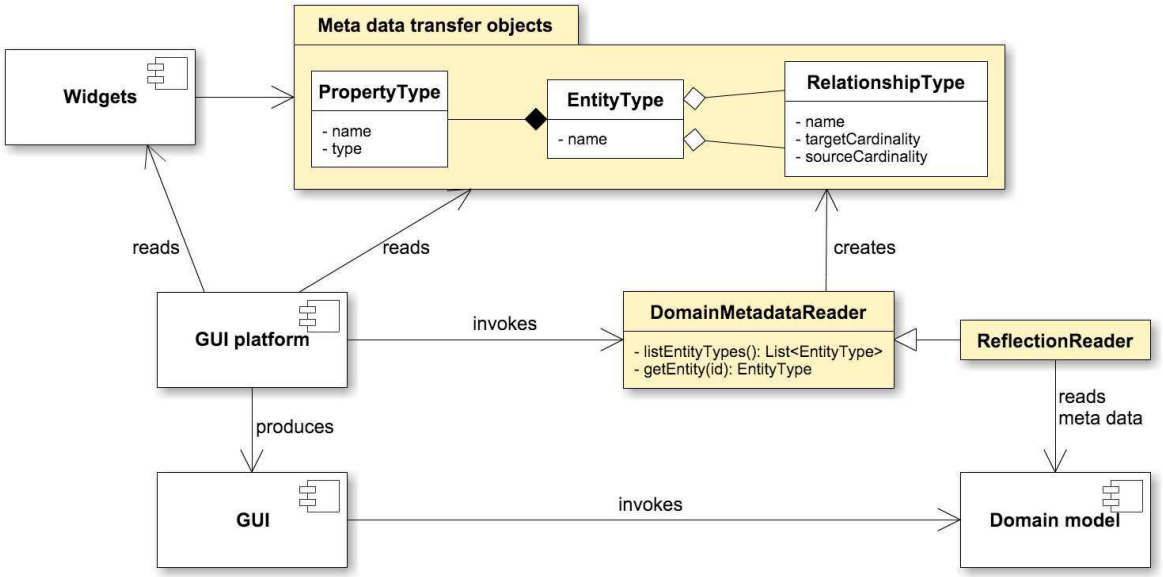


Figura 4.3: Exemplo de uso do padrão DOMAIN MODEL X-RAY

Consequências

- A camada de GUI se torna independente da tecnologia utilizada para implementar o domínio;
- A camada de GUI se torna independente da tecnologia utilizada para ler os metadados do domínio;
- Os *widgets* e a plataforma de GUI, desacoplados do domínio, são mais reutilizáveis.
- Será preciso implementar mais duas camadas de classes (interface do Reader e sua implementação) para ter acesso aos metadados do domínio.

Usos conhecidos

Os arcabouços de *Scaffold* que expõem *templates* para geração de código — *Ruby on Rails*, *Grails* e *Django* — utilizam marcadores para representar os metadados do domínio para o qual o código está sendo gerado. Por exemplo, para se acessar todas as propriedades de uma entidade, Rails usa a coleção `attributes`, que, junto a outros metadados, é provida em tempo de geração de código pela própria plataforma *Rails*. Porém a estratégia de leitura dos metadados fica no interior da plataforma *Rails* e sua substituição não é trivial.

A leitura dos metadados de domínios orientados a objeto pode ser implementada através dos mecanismos nativos de reflexão, presente em linguagens populares como Java [22] e C# [66]. Porém diversas outras formas de leitura podem ser implementadas. Por exemplos, os metadados do domínio podem ser expressos em arquivos de configuração XML ou até em bancos de dados como AOM faz.

A linguagem WSDL [19] descreve os metadados dos tipos e procedimentos que compõem a interface de um *Web service*. Desse modo, o código cliente pode ser gerado automaticamente ao se consultar os metadados WSDL, de modo semelhante ao que a plataforma de GUI fas ao consultar a interface `DomainMetadataReader`.

Padrões relacionados

O padrão DOMAIN WIDGET deve ser usado em conjunto com DOMAIN MODEL X RAY para que uma plataforma de GUI baseada em metadados viabilize o reúso de código.

Este padrão pode ser implementado com uma FACADE [35] para agregar encapsular todos os serviços de consulta de metadados do domínio.

A forma de obtenção dos metadados pode ser trocada facilmente se for implementada através de STRATEGY [35], METADATA READER STRATEGY [38] ou METADATA CONTAINER [38].

INSTANCE PRESENTATION e SERVICE PRESENTATION apresentam os metadados de instâncias e serviços respectivamente [67].

4.4 Padrão de projeto INDIRECT WIDGET

Sinônimos: WIDGET PORTS, FINE-GRAINED WIDGETS

O código da GUI não deve ser implementado em apenas uma classe, pois seria muito complexo e agregaria responsabilidades demais. Portanto, quebrar a GUI em componentes pequenos é uma boa prática, uma vez que cada artefato de GUI seria simples e possuiria poucas responsabilidades coesas. Esse princípio também se aplica aos metacomponentes de GUI.

Uma forma de organizar as responsabilidades dos componentes de GUI, em aplicações corporativas, é separá-los pelo tipo de dados com o qual eles lidam, como pode ser visto nos

exemplos da Tabela 4.1.

Tabela 4.1: Exemplos de componentes de GUI por tipo de dados manipulados

Tipo de dado	**Exemplos de componentes de GUI**
Conjunto de entidades	Tabelas de listagem, listas (*bullets*), relatórios e gráficos
Uma entidade	Linha de tabela, formulário, visualização de detalhes
Propriedade	Caixa de texto, *check box*, campo data, célula de tabela
Relacionamento	*Combo box*, *link*, caixa de seleção múltipla

Outra forma viável para organizar as responsabilidades é usar o padrão de projetos COMPOSITE [35], que permite criar componentes genéricos e compô-los numa árvore flexível de objetos. No contexto de GUI, os desenvolvedores podem usar o COMPOSITE, criando uma árvore de componentes bem encapsulados, que implementa as diversas telas de uma aplicação. Na Figura 4.4 ilustra-se um exemplo de composição de GUI através de uma árvore de componentes genéricos de GUI, onde os componentes superiores delegam responsabilidades para alguns componentes inferiores. Nos casos onde um componente não possui filhos, todas as responsabilidades alocadas para aquele componente são implementadas no seu próprio código e não são delegadas para outros componentes.

Os padrões de renderização AOM, por exemplo, definem *widgets* com delegação de responsabilidades [99] no estilo COMPOSITE. Dois desses padrões (PROPERTY RENDERER e ENTITY VIEW) se ligam através do código fonte, pois as classes `EntityView` conhecem diretamente as classes `PropertyRenderer` que utilizarão. No entanto, essa ligação direta entre os componentes é um problema, pois causa acoplamento forte entre eles, reduzindo as possibilidades de reúso.

Na Listagem 4.5, pode-se ver um *widget* de formulário que referencia diretamente os *widgets* dos seus campos. Embora as responsabilidades da GUI tenham sido divididas entre os*widges*, o fato de haver referências diretas entre eles mantém o código complexo, dificultando a customização da GUI. Por exemplo, para adicionar um novo *widget* específico para desenhar os campos inteiros com nome igual a CEP ou outro para desenhar todos os campos de data do sistema, seria necessário expandir ainda mais a já complexa estrutura *if* entre as linhas 5 e 14.

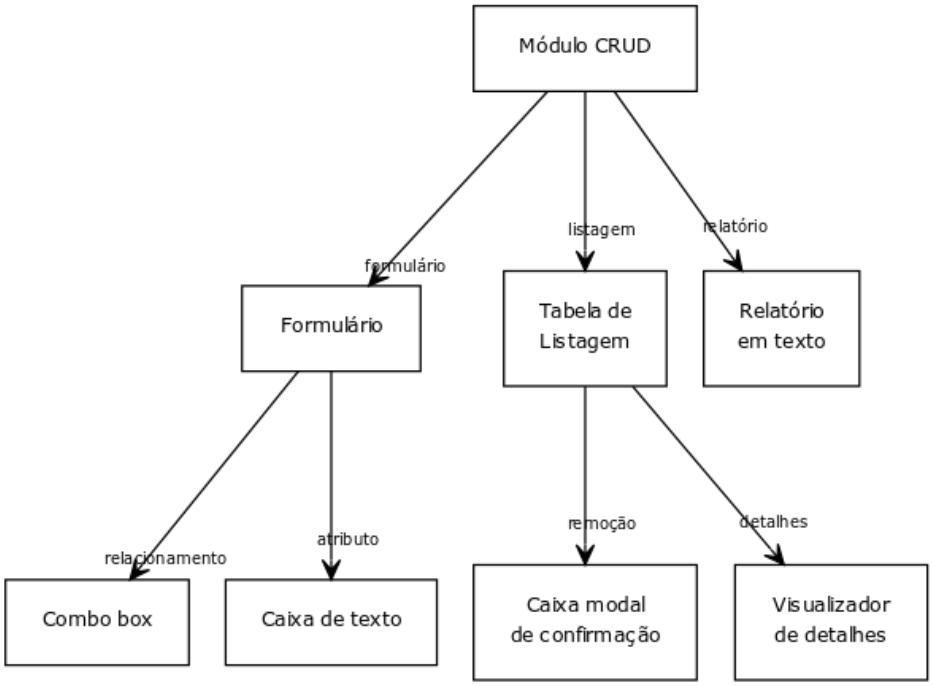


Figura 4.4: Exemplo de composição de GUI em uma árvore de componentes

Código Fonte 4.5: Exemplo de componentes ligados diretamente

```
1  class FormularioSimples
2    desenhar(entidade):
3      res = "<form action='/api/" + entidade.nome + "'>"
4      for propriedade in entidade.propriedades
5        if propriedade.tipo == 'integer'
6          res += new CampoInteiro().desenhar(propriedade)
7        else if propriedade.tipo == 'boolean'
8          res += new CheckBox().desenhar(propriedade)
9        else if propriedade.nome == 'cpf'
10         res += new CampoCPF().desenhar(propriedade)
11       else if propriedade.tipo == 'string' and propriedade.tamanho > 100
12         res += new TextArea().desenhar(propriedade)
13       else
14         res += new CaixaTexto().desenhar(propriedade)
15     return res + "</form>"
```

Problema

Ao reduzir a quantidade de responsabilidades dos *widgets*, quebrando-os em componentes menores, como evitar o acoplamento forte entre eles?

Forças

- Quando os *widgets* se conhecem diretamente, a verificação erros de tipagem pode ser feita em tempo de compilação, porém ocorre um fortemente acoplamento;
- Quanto mais flexível o projeto dos *widgets* for, mais fácil será a sua composição;
- *Widgets* podem usar polimorfismo [15] para se comunicar indiretamente, através de um super tipo;
- Se os *widgets* realizarem interfaces baseadas nos metadados do modelo de domínio, poderão ser reutilizados em diversos pontos de uma aplicação corporativa;
- A estrutura dos metadados do modelo do domínio é padrão mesmo em domínios diferentes, sendo composta por entidades, propriedades e relacionamentos. Portanto, a interface desses metadados pode ser congelada e utilizada para substituir referências diretas entre os *widgets*;
- Referências indiretas são menos legíveis que as diretas.

Solução

Criar portas bem definidas para a comunicação entre widgets, compostas por um nome único e uma interface baseada nos metadados do modelo do domínio. Utilizar um objeto externo para ligar (compor) os widgets.

Quando se divide as responsabilidades de GUI entre dois *widgets*, o *widget* original se transforma em um *widget pai P* que invoca o *widget filho F*. Dessa forma, uma árvore de *widgets* pode ser montada para representar toda a GUI de uma aplicação corporativa, desde *widgets* complexos como `CrudModule` (um módulo que desenha telas para realização de

todas as operações de CRUD[3] de uma entidade) até *widgets* bem simples, por exemplo, `TextField` (um campo Caixa de Texto em um formulário).

Embora *P* precise invocar *F* para delegar responsabilidades, esses *widgets* não devem se conhecer, a fim de evitar o acoplamento forte, que reduz a reusabilidade do código. Portanto, o *widget P* deve definir **portas** pelas quais delegará responsabilidades para seus filhos, de forma que *P* só conheça o *nome* e o *tipo* das portas que declara. O nome da porta deve ser único em toda a aplicação e o tipo da porta define a interface dos *widgets* que podem ser ligados na porta. Para cada porta definida em *P*, um *widget* filho deve ser provido.

Existem três tipos de metadados no modelo do Domínio: entidades, propriedades e relacionamentos. Assim sendo, a solução deve prover quatro interfaces para os componentes de GUI baseados em metadados: `Property` para desenhar fragmentos de GUI para um atributo de uma entidade, da mesma forma como o padrão PROPERTY RENDERER de AOM faz; `Entity` que desenha fragmentos para uma instância de uma entidade, semelhante ao padrão ENTITY VIEW de AOM; `EntitySet` trabalha como o padrão DYNAMIC VIEW de AOM, desenhando fragmentos para várias instâncias de uma entidade; e `Relationship` encapsula o código de GUI que representa relacionamentos entre entidades do domínio e está detalhado no padrão RELATIONSHIP RENDERER.

Na Figura 4.5 ilustra-se o exemplo de um *widget* pai, chamado `CrudModule`, delegando responsabilidades de GUI para outros dois *widgets* filhos: `VerticalLayoutForm` e `TxtReport`. A GUI executa o *widget* `CrudModule` para desenhar telas genéricas de um módulo de cadastro como, por exemplo, tabelas de listagem, formulários de criação e atualização, opções de consulta, caixas de confirmação de remoção e relatórios. `CrudModule` pode criar portas livremente, conforme a necessidade de delegar responsabilidades de GUI para outro *widget*. Nessa figura, `CrudModule` criou duas portas: *form*, para desenhar os formulários de criação e atualização; e *report*, para desenhar as telas de relatórios.

É importante ressaltar que o tipo definido numa porta deve casar com o tipo do *widget* provido para ela. Na Figura 4.5, essa associação está representada pelas cores iguais entre o nome das portas e os *widgets* que as servem. Por esse motivo, *form* e `VerticalLayoutForm` são verdes (tipo `Entity`), e *report* e `TxtReport` são azuis (tipo `EntitySet`).

[3]CRUD, do inglês *Create*, *Read*, *Update* e *Delete*.

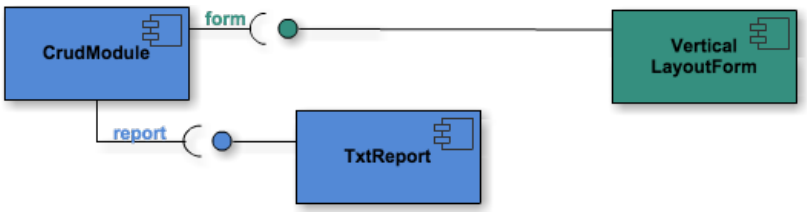


Figura 4.5: Ligando *widgets* indiretamente através de portas

Dado que os *widgets* não se conhecem diretamente, eles podem ser trocados facilmente, até em tempo de execução. Se o *widget* da porta *report* for trocado para um `HtmlReport` ou o *widget* da porta *form* for substituído por um `TabbedLayoutForm`, o *widget* pai, `CrudModule`, não será afetado.

Embora seja simples, esse mecanismo de portas e interfaces pode organizar responsabilidades de telas complexas, pois não há limites nessa abordagem para a altura da árvore de *widgets*. Assim sendo, qualquer *widget* pode definir novas portas, mesmo que já tenha sido provido para uma porta de outro *widget* (que é seu pai), repetindo o processo de composição livremente.

Na Figura 4.6 ilustra-se um exemplo mais refinado da composição de *widgets*. No caso do tipo `Property`, mais uma informação pode ser utilizada no momento de ligar portas e *widgets*. Trata-se do Tipo da propriedade, que pode modificar o *widget* ligado, a partir do tipo do atributo que está sendo desenhado na tela. Nesse exemplo, um *widget* genérico de Tabela de listagem (`ListingTable`) criou uma porta para delegar o desenho de células (*cell*) para outros *widgets*, mas a ligação foi expandida para envolver vários *widgets* na mesma porta. Isso ocorre porque nas propriedades o comportamento dos *widgets* pode variar completamente dependendo do tipo de dado que está sendo manipulado.

Para ilustrar essa situação, o *widget* `SimpleCell` escreve o dado de uma célula da mesma forma como ele é carregado do banco de dados, enquanto que, em colunas do tipo *Data*, o dado bruto (possivelmente um inteiro representando milissegundos) deve ser convertido num formato de data legível, o que é feito pelo *widget* `DateFormatterCell`. Dessa forma, na Figura 4.6 ilustra-se uma bifurcação na porta *cell*. Enquanto o braço *Date* conecta a porta *cell* ao *widget* `DateFormatterCell`, o braço padrão (*), que serve para todos os demais tipos, conecta a porta *cell* ao *widget* `SimpleCell`. Nessa figura, as portas e os *widgets* do tipo `Property` aparecem em vermelho.

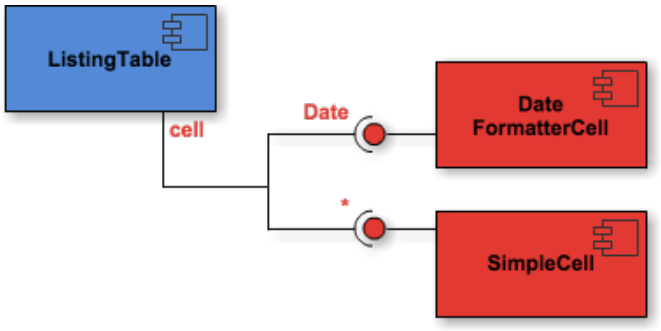


Figura 4.6: Ligando componentes e portas do tipo `Property`

A Listagem 4.6 representa o código da Listagem 4.5 após a aplicação do padrão INDIRECT WIDGET. Em vez de referenciar os *widgets* filhos diretamente, o *widget* pai `FormularioSimples` abre uma porta chamada `campos_de_formulario` e espera que algum objeto ou configuração externa decida que campos devem ser selecionados para cada uma das suas propriedades. A lógica para a seleção dos *widgets* filhos, para essa porta, está declarada entre as linhas 8 e 12, no formato `condicao => Widget filho`.

Código Fonte 4.6: Exemplo de componentes indiretos

```
1  class FormularioSimples
2    desenhar(entidade):
3      res = "<form action='/api/" + entidade.nome + "'>"
4      res += abrirPorta("campos_de_formulario", entidade.propriedades)
5      return res + "</form>"
6
7  Configuracoes da porta "campos_de_formulario":
8    tipo == 'integer' => CampoInteiro
9    tipo == 'boolean' => CheckBox
10   nome == 'cpf' => CampoCPF
11   tipo == 'string' and tamanho > 100 => TextArea
12   default => CaixaTexto
```

Consequências

- A lógica de escrita da GUI em HTML foi desacoplada na seleção de componentes filhos, tornando o código mais simples para manter;

- A ligação indireta entre os *widgets* dificulta a depuração e a legibilidade no desenvolvimento da GUI, portanto a plataforma de GUI deve prover ferramentas de suporte à visualização e depuração;
- Os *widgets* não se conhecem diretamente, portanto seus relacionamentos são flexíveis para modificações até em tempo de execução;
- As interfaces para os quatro tipos de metadados do modelo do domínio isolam os *widgets* e permitem a sua reutilização em diversos pontos da GUI;
- Os *widgets* podem se combinar em uma árvore flexível, composta por elementos simples mas que pode representar estruturas complexas.

Usos conhecidos

O motor de *templates StringTemplate* quebra os widgets, gerando artefatos menores e com responsabilidades mais coesas [76, p. 9];

Os *templates* de geração de código de *Ruby on Rails* podem chamar outros *templates* pelo nome, através do comando `generate` [79, 78];

O arcabouço TERESA [68] usa objetos do tipo `connection` para ligar os *widgets*, que são representados pelo tipo `presentation`.

Padrões relacionados

Os *widgets* que forem projetados segundo o padrão DOMAIN WIDGET podem ser facilmente ligados com o uso de INDIRECT WIDGET.

O objeto WIDGET ENGINE pode ser o responsável por ligar os *widgets* através das portas.

INDIRECT WIDGET desacopla componentes que seguem o padrão de projeto COMPOSITE [35].

Um conjunto de *widgets* pode ser classificado pelos seus tipos e publicado numa biblioteca de componentes (COMPONENT LIBRARY [83]).

4.5 Padrão de projeto WIDGET ENGINE

Sinônimos: WIDGET CONTAINER, METACOMPONENT ENGINE

Os *widgets* feitos segundo o padrão INDIRECT WIDGET são desacoplados, pois conhecem apenas a interface uns dos outros. Porém, é preciso que algum objeto conheça os tipos desses *widgets*, a fim de instanciá-los e ligá-los, num processo que chamamos de *Composição da GUI*. Segundo os padrões EXPERT e CREATOR [57], os *widgets* pais poderiam criar e compor os *widgets* filhos nas suas portas, porém isso acoplaria fortemente os pais aos tipos concretos dos seus respectivos filhos.

Uma forma de desacoplar totalmente os *widgets* é colocar a responsabilidade de composição em um objeto externo, chamado *Compositor*, que seria o único a conhecer os tipos concretos dos *widgets*. Porém, essa abordagem inibe o reúso do código do compositor, demandando que sejam criados diversos compositores dentro de uma mesma aplicação. A Listagem 4.7 mostra um exemplo de compositor que conhece diretamente os *widgets*.

Código Fonte 4.7: Exemplo de compositor com referências diretas

```
1  import CampoInteiro, CheckBox, CampoCPF, TextArea, CaixaTexto
2
3  class Compositor
4    configurarPortaCamposDeFormulario:
5      GUI.configurarPorta('campos_de_formulario')
6        .addCondition(tipo == 'integer', CampoInteiro)
7        .addCondition(tipo == 'boolean', CheckBox)
8        .addCondition(nome == 'cpf', CampoCPF)
9        .addCondition(tipo == 'string' and tamanho > 100, TextArea)
10       .addDefault(CaixaTexto)
```

Problema

Como reduzir o acoplamento entre compositores e *widgets*?

Forças

- Ao compor os *widgets* diretamente em código fonte, erros de tipagem podem ser verificados em tempo de compilação, porém o compositor fica fortemente acoplado aos

widgets;

- Quanto mais flexível o projeto de *widgets* for, mais fácil será compor *widgets* para produzir telas complexas;
- Compositores podem manipular *widgets* desconhecidos polimorficamente, através de um super tipo;
- Para instanciar *widgets* sem conhecer o seu tipo concreto, os compositores precisam usar técnicas como reflexão [37];
- Dados sobre os tipos dos *widgets* podem ser carregados a partir de fontes externas.

Solução

Implementar um compositor de GUI com Inversão de Controle, lendo dados externos para instanciar widgets e compô-los numa árvore flexível, a fim de produzir as telas de uma aplicação corporativa. O compositor, que neste padrão é chamado de motor ou engine, só deve conhecer a interface dos widgets.

O motor de GUI deve atuar como um arcabouço para desenhar as telas de aplicações corporativas. Ele coordena a composição de GUI utilizando o contexto e os dados externos, a fim de escolher e invocar os melhores *widgets* para cada fragmento da GUI que precisa ser desenhado.

Se os dados da composição de GUI estiverem externalizados em arquivos de configuração ou bancos de dados, o motor de GUI poderá: carregar os dados em tempo de execução, instanciar os *widgets* através de reflexão e uní-los dinamicamente. Dessa forma, a árvore de *widgets* da GUI pode ser alterada sem afetar o código fonte do compositor e o acoplamento será mínimo.

No contexto do padrão INDIRECT WIDGET, o motor de GUI pode ler as informações de composição de *widgets* e portas a partir de dados externos, inclusive os tipos concretos dos componentes, removendo esse acoplamento totalmente do código da GUI.

A Tabela 4.2 representa a estrutura dos dados externos que podem ser usados para compor os *widgets* da Figura 4.5. Quando um *widget* estiver desenhando a GUI e repassar a execução para alguma porta, o motor de GUI entra em cena para procurar uma linha da tabela com o

mesmo nome da porta. Daí, o motor usa reflexão para carregar o *widget* definido na mesma linha e repassa a execução para ele. Dessa forma, cada linha da tabela representa uma ligação entre *widget* pai – porta – *widget* filho.

Tabela 4.2: Dados de configuração para ligar *widgets* através de portas

Porta		**Widget**
Nome	Tipo	
form	Entity	`VerticalLayoutForm`
report	EntitySet	`TxtReport`

Semelhantemente a Tabela 4.3 contém a estrutura dos dados externos para compor a Figura 4.6. Nesse caso, além de procurar pelo nome da porta na tabela, o motor de GUI precisa verificar se há algum *widget* específico para o tipo da propriedade que está sendo desenhada, por exemplo, *Date* – `DateFormatterCell`. Caso contrário, o *widget* padrão (* – `SimpleCell`) deve ser utilizado.

Tabela 4.3: Dados de configuração para ligar *widgets* do tipo `Property`

Porta			**Widget**
Nome	Tipo	*Tipo de Propriedade*	
cell	Property	*Date*	`DateFormatterCell`
cell	Property	*	`SimpleCell`

Consequências

- A ligação indireta dificulta a depuração e a legibilidade da GUI, portanto o motor de GUI deve prover ferramentas de suporte à visualização e depuração;
- O compositor não conhece os *widgets* diretamente e os seus relacionamentos são flexíveis até em tempo de execução;
- As interfaces para os quatro tipos de metadados do Modelo de Domínio isolam o compositor dos *widgets* e permite a criação de um compositor genérico capaz de desenhar diversos tipos de GUI;

- O compositor pode combinar os *widgets* em uma árvore flexível, composta por elementos simples mas que pode representar estruturas complexas.

Usos conhecidos

Ferramentas de inversão de controle, como *Ninject*[4] e *Spring IoC*[5], podem ler dados de arquivos de configuração, instanciar componentes por reflexão e ligar esses objetos no código original da aplicação.

Padrões relacionados

O relacionamento entre o motor de GUI e os DOMAIN WIDGETS pode ser implementado através dos padrões TEMPLATE METHOD ou STRATEGY [35].

O motor de GUI atua como uma fábrica de *widgets* (ABSTRACT FACTORY [35]).

Os *widgets* instanciados pelo motor de GUI podem ser do tipo PROPERTY RENDERER, ENTITY VIEW, DYNAMIC VIEW [100] ou RELATIONSHIP RENDERER.

INDIRECT WIDGETS podem ser ligados pelo motor de GUI.

4.6 Padrão arquitetural SERVER RENDERING

Sinônimos: BACKEND RENDERING, SERVER WIDGET

Atualmente a maior parte das aplicações corporativas *web* utiliza a arquitetura em camadas [31], com a finalidade de organizar e encapsular o código de GUI, lógica de negócio e persistência. Uma das principais abordagens para implementar a camada de GUI com interface *web* é produzir o código HTML no servidor e enviá-lo para o navegador *web* através do protocolo HTTP [27].

Os padrões INDIRECT WIDGET e WIDGET ENGINE, que definem *widgets* genéricos e reutilizáveis, também podem ser utilizados em aplicações *web*.

[4] http://www.ninject.org/extensions.html

[5] http://www.tutorialspoint.com/spring/spring_ioc_containers.htm

Problema

Como produzir GUI, a partir de componentes baseados em metadados e dados de configuração, nos servidores de aplicações corporativas *web*?

Forças

- O navegador *web* precisa carregar os recursos e as páginas HTML do servidor para exibir a GUI ao usuário da aplicação;
- Nas abordagens com montagem de HTML no lado servidor, os navegadores não são capazes de executar a lógica de montagem de telas baseados nos metadados do domínio;
- *Widgets* e dados de composição são recursos necessários para o servidor desenhar as páginas HTML e dependem da tecnologia na qual o servidor foi implementado.

Solução

Implementar o código dos widgets na mesma tecnologia do servidor, armazenar os dados de composição e executar o arcabouço de GUI e os widgets no servidor, e transferir o HTML resultante para o navegador.

A principal restrição na abordagem com montagem de HTML no lado servidor, é o fato dos navegadores não serem responsáveis por executar código, podendo apenas exibir o código HTML que recebem do servidor. Portanto a lógica de montagem de GUI baseada nos metadados do domínio deve ser executada no servidor, na mesma tecnologia em que ele foi implementado.

Os dados de composição dos *widgets* podem ser facilmente armazenados no servidor, em bancos de dados ou arquivos de configuração. Sendo, portanto, carregados em memória pelo arcabuoço de GUI para configurar os *widgets* e gerar o código HTML que será enviado para o navegador através do protocolo HTTP.

Consequências

- O nó Servidor utiliza uma interface bem definida (HTML e HTTP) para enviar a GUI ao navegador;
- Plataformas de GUI baseadas no padrão INDIRECT WIDGET podem desenhar a GUI no nó servidor;
- O tráfego de rede é maior e o tempo de resposta da aplicação é menor devido à necessidade de fazer requisições ao servidor para todas as interações do usuário com a aplicação.

Padrões relacionados

Uma das primeiras decisões arquiteturais que deve ser tomada, ao criar um arcabouço para o desenvolvimento de GUI para aplicações corporativas baseada nos metadados domínio, é o uso de SERVER RENDERING ou CLIENT RENDERING, pois esses padrões arquiteturais são mutuamente exclusivos.

SERVER RENDERING implementa DOMAIN WIDGETS e INDIRECT WIDGETS no servidor *web*.

O padrão DOMAIN MODEL X-RAY pode ser utilizado para carregar os metadados do domínio no servidor.

SERVER RENDERING executa o WIDGET ENGINE no servidor.

4.7 Padrão arquitetural CLIENT RENDERING

Sinônimos: FRONTEND RENDERING, FRONTEND WIDGET

Atualmente a maior parte das aplicações corporativas *web* utiliza a arquitetura em camadas [31], com a finalidade de organizar e encapsular o código de GUI, lógica de negócio e persistência. Uma das principais, e mais modernas, abordagens para implementar a camada de GUI em aplicações corporativas com interface *web* é enviar código executável (geralmente em *JavaScript*) do servidor para o navegador [36, 65]. Essa abordagem viabiliza a construção da GUI perto do usuário final, reduzindo a carga de processamento no servidor e o tráfego de rede.

Os padrões INDIRECT WIDGET e WIDGET ENGINE, que definem *widgets* genéricos e reutilizáveis, também podem ser utilizados em aplicações *web*. Para esses padrões serem implementados no navegador, os *widgets* e os dados de composição precisam ser disponibilizados para esse nó.

Problema

Como produzir GUI, a partir de componentes baseados em metadados e dados de configuração, nos navegadores de aplicações corporativas *web*?

Forças

- O navegador *web* precisa carregar os recursos do servidor para desenhar e exibir as páginas HTML;
- *Widgets* e dados externos também são recursos que precisam ser baixados do servidor para o navegador *web*;
- Os *widgets* precisam executar dentro do navegador, portanto devem ser escritos numa linguagem que navegadores possam executar;
- O código de linguagens dinâmicas (como *JavaScript*) pode ser armazenado como *strings*, recuperado e interpretado em tempo de execução.

Solução

Armazenar o código dos widgets e os seus dados de composição no servidor, baixá-los para o navegador sob demanda, e executar o arcabouço de GUI e os widgets no navegador.

Este padrão afeta a arquitetura das aplicações corporativas que o utilizam, como está descrito na Figura 4.7. Dois nós da aplicação — *Client* (Navegador *web*) e *Server* (Servidor *web*) — devem conter alguns componentes para viabilizar a solução. Assim sendo, CLIENT RENDERING pode ser considerado um padrão arquitetural, diferentemente dos padrões de projeto, que são mais simples.

Como em qualquer aplicação corporativa comum, o nó Servidor provê uma API de Serviço operacionais (*Operational service*), que atendem às requisições da GUI, executando a

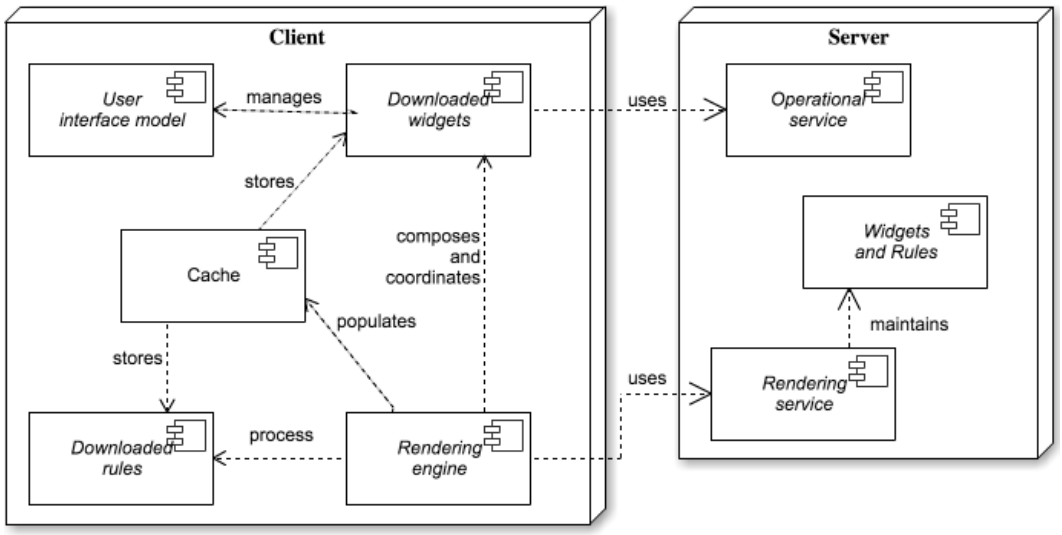


Figura 4.7: Arquitetura do padrão CLIENT RENDERING

lógica do domínio e acessando um banco de dados. A principal inovação deste padrão é o Serviço de renderização (*Rendering service*), um componente responsável por armazenar o código dos *widgets* e as suas ligações. Além disso, o serviço de renderização deve enviar todos esses recursos ao navegador.

O serviço de renderização pode utilizar um banco de dados ou um arquivo de configuração para persistir os dados da GUI, cuja estrutura pode ser semelhante a definida nas Tabelas 4.2, 4.3, 4.4 e 4.5.

O nó cliente mantém um Modelo da Interface com o Usuário (*User interface model*), que nos navegadores recebe o nome de *DOM - Document Object Model*. Além disso, o nó cliente é capaz de executar código que altera o Modelo e as mudanças são refletidas na GUI.

O código do cliente é organizado em quatro módulos. A *Rendering engine* implementa o padrão WIDGET ENGINE e realiza inversão de controle nos *widgets* que executam no cliente. Para tanto, esse componente é responsável por decidir quais *widgets* devem ser baixados do servidor, para cada tela aberta na GUI.

Uma vez baixados, os *widgets* são executados na tela que os demandou e podem ser guardados para reúso em outras telas. Por isso, é importante manter um *Cache* dos *widgets* e das suas ligações no nó cliente. As ligações entre os *widgets* são representadas através de dados externos por causa do padrão WIDGET ENGINE. Dessa forma, o padrão CLIENT RENDERING trata as ligações como regras (*rules*) que são baixadas do servidor e analisadas

no cliente, podendo, inclusive, ser reutilizadas em várias telas.

Consequências

- O nó Servidor deve prover uma interface para enviar ao cliente: os metadados do domínio; código e as ligações dos *widgets*;
- Plataformas de GUI baseadas no padrão INDIRECT WIDGET podem desenhar a GUI no nó cliente;
- Uma *Cache* de *widgets* e das suas ligações deve ser disponibilizada a fim de reduzir o tráfego de rede e o tempo de resposta;
- Os *widgets* são armazenados no Servidor, mas são implementados em uma linguagem diferente do Servidor, que é executável no Navegador.

Padrões relacionados

Uma das primeiras decisões arquiteturais que deve ser tomada, ao criar um arcabouço para o desenvolvimento de GUI para aplicações corporativas baseada nos metadados domínio, é o uso de SERVER RENDERING ou CLIENT RENDERING, pois esses padrões arquiteturais são mutuamente exclusivos.

CLIENT RENDERING transfere DOMAIN WIDGETS e INDIRECT WIDGETS do servidor *web* para os navegadores.

O componente *rendering service* implemeta o padrão DOMAIN MODEL X-RAY.

CLIENT RENDERING executa o WIDGET ENGINE no navegador.

Pode-se utilizar os padrões IDENTITY MAP e OPTIMISTIC OFFLINE LOCK para implementar o componente *Cache* [31].

4.8 Padrão de projeto WIDGET SCOPE

Sinônimos: WIDGET RULE, WIDGET SELECTOR

Em aplicações corporativas, partes da GUI podem ser implementadas através de componentes genéricos, que possuem o mesmo comportamento independente do elemento re-

presentado na GUI. Por exemplo, as propriedades `Nome`, `RG` e `Cargo` de uma entidade `Cliente` podem ser representadas por caixas de texto simples.

Algumas abordagens baseadas em metadados para GUI, como os arcabouços de *Scaffold* e os padrões de renderização *AOM*, provêm **artefatos padrão** para desenhar certos elementos da GUI sempre da mesma forma. Com essa funcionalidade, o desenvolvedor não precisa configurar ou implementar um *widget* para cada entidade, propriedade e relacionamento da aplicação, reduzindo bastante o esforço no desenvolvimento de GUI.

No entanto, em algumas situações, é preciso substituir os *widgets* padrão por outros específicos. Isso ocorre, por exemplo, nos campos que possuem máscara, como `CPF`, `CEP` e `telefone`, que precisam de caixas de texto especiais. Na Listagem 4.8 descrevem-se duas funções baseadas em metadados. A primeira desenha caixas de texto simples e a segunda utiliza uma expressão regular (*regex*) para implementar máscaras nas caixas de texto.

Além do caso dos *widgets* genéricos *versus* específicos, é preciso considerar que, numa versão inicial do sistema, os *widgets* genéricos podem ser utilizados para dar um *feedback* visual rápido ao desenvolvedor e ao cliente, sendo, posteriormente, substituídos pelos *widgets* definitivos.

Código Fonte 4.8: Desenhando caixas de texto baseado em metadados

```
1 def desenha_propriedade_simples_em_caixa_texto(propriedade):
2   resultado = propriedade.etiqueta + ":<br>"
3   resultado += "<input type='text' name='"
4   return resultado + propriedade.nome + "'><br>"
5
6 def desenha_propriedade_em_caixa_texto_mascara(propriedade, regex):
7   resultado = propriedade.etiqueta + ":<br>"
8   resultado += "<input type='text' name='" + propriedade.nome + "' "
9   return resultado + "pattern='" + regex + "'><br>"
```

Podemos concluir que os *widgets* padrão, providos pela plataforma de GUI, não são suficientes para atender a todas as especificidades das telas de aplicações corporativas, portanto é preciso definir um mecanismo para substituí-los facilmente em escopos pré-determinados.

Problema

Como prover *widgets* genéricos, baseados em metadados, para a GUI de uma aplicação corporativa e como substituí-los facilmente?

Forças

- *Widgets* genéricos podem ser usados para boa parte da GUI e em estágios iniciais da GUI;
- *Widgets* genéricos não atendem a todas as especificidades da GUI;
- *Widgets* genéricos de propriedades podem ser disponibilizados de acordo com o tipo da propriedade, por exemplo, Caixas de texto para *strings*, *Check boxes* para *booleans*.

Solução

Estender o padrão INDIRECT WIDGET*, adicionando uma dimensão, chamada* ***escopo****, às ligações entre portas e widgets. O valor do escopo pode ser* ***padrão (*)****, que estabelece os widgets para casos gerais; ou referências a entidades, propriedades ou relacionamentos* ***específicos*** *onde as ligações têm validade.*

O diagrama da Figura 4.8 apresenta um exemplo extenso do uso da dimensão *escopo* para definir *widgets* gerais e específicos. O *widget* principal, `CrudModule`, define duas portas — *form* e *report* — e cada uma possui uma regra para definir seu *widget* padrão. `TxtReport` desenha relatórios para todas as entidades, exceto para `OrderItems` cujo relatório é feito por `CsvReport`. Semelhantemente, `VerticalLayoutForm` desenha formulários para todas as entidades, exceto para `Product` cujo formulário é dividido em abas (`TabbedLayoutForm`).

Esse exemplo demonstra que, nas portas dos tipos *Entity* e *EntitySet*, os valores possíveis são padrão (*) ou o nome específico de uma entidade (como `OrderItems` ou `Product`). Ademais, as plataformas de GUI podem facilmente implementar valores intermediários para o escopo, baseados em caracteres coringa. Por exemplo, o valor de escopo `*Product*` determina que um *widget* deve ser aplicado em todas as entidades cujo nome possua o

texto `Product` em alguma posição, como `ProductItem`, `ProductSpecification` e `MyProduct`.

Para portas de *relacionamentos*, os desenvolvedores podem definir o escopo dos *widgets* a partir da cardinalidade do relacionamento (ver padrão RELATIONSHIP RENDERER). Por exemplo, cada `Product` (Produto) tem UM `Salesman` (Vendedor) que é representado em formulários através de um `Combobox`, e `Products` (Produtos) são armazenados em MUITAS `Shelves` (Prateleiras) que são desenhadas nos formulários em uma `Listbox` com opção de seleção múltipla.

Na Figura 4.8 ilustram-se duas regras gerais: todos os relacionamentos com cardinalidade UM devem ser desenhados por `Combobox`; e todos os relacionamentos com cardinalidade MUITOS devem ser desenhados por `Listbox`. Portanto, o desenvolvedor não precisa escrever GUI específica para os relacionamentos entre `Product`, `Salesman` e `Shelf`. Por outro lado, na entidade `Supplier` (Fornecedor), todo os relacionamentos devem ser representados como painéis `MasterDetails`, portanto deve haver uma regra restringindo o escopo desse *widget* aos relacionamentos daquela entidade específica, independente da cardinalidade.

Por fim, nas portas do tipo *Property*, a dimensão escopo precisa considerar o fato de que as propriedades pertencem a entidades. Portanto, um mesmo escopo pode envolver propriedades de entidades diferentes ou todas as propriedades de uma entidade. Por exemplo, todos os campos `CPF` em qualquer entidade devem ser desenhados por `CpfField` ou todos os campos de `Customer` serão desenhados por `SpecialTextField`.

Na Figura 4.8, a porta de propriedade *line* possui três ligações. A primeira aplica o *widget* `NumberField` para todas as propriedades do tipo *real* na entidade `Customer`. A segunda seleciona o *widget* `Checkbox` em todas as propriedades booleanas cujo nome é `active` em qualquer entidade. A última ligação define que o *widget* padrão, para todas as demais propriedades de todas as entidades, é `TextField`.

A Tabela 4.4 mostra a estrutura dos dados de configuração externos que podem representar ligações entre portas e *widget* que utilizam escopo. Todas as ligações da Figura 4.8 estão representadas na Tabela 4.4.

WIDGET SCOPE é um padrão que facilita a composição de *widgets*, porém pode causar inconsistência na sua configuração, se não forem definidas ligações para o escopo padrão.

Tabela 4.4: Estrutura dos dados para ligações com escopo

Porta					**Widget**
Nome	Tipo	Tipo de Propriedade	Cardinalidade	*Escopo*	
report	EntitySet			*Entidade OrderItems*	`CsvReport`
report	EntitySet			* *Todas as outras entidades*	`TxtReport`
form	Entity			*Entidade Product*	`TabbedLayoutForm`
form	Entity			* *Todas as outras entidades*	`VerticalLayotForm`
relation	Relationship		* (UM ou MUITOS)	*Entidade Supplier*	`MasterDetails`
relation	Relationship		UM	* *Todas as outras entidades*	`Combobox`
relation	Relationship		MUITOS	* *Todas as outras entidades*	`Listbox`
line	Property	real		*Customer.* (Todas as propriedades da entidade Customer)*	`NumberField`
line	Property	boolean		**.active (Propriedades active de todas as entidades)*	`Checkbox`
line	Property	Todos os outros tipos		*.* (Todas as propriedades de todas as entidades)	`TextField`

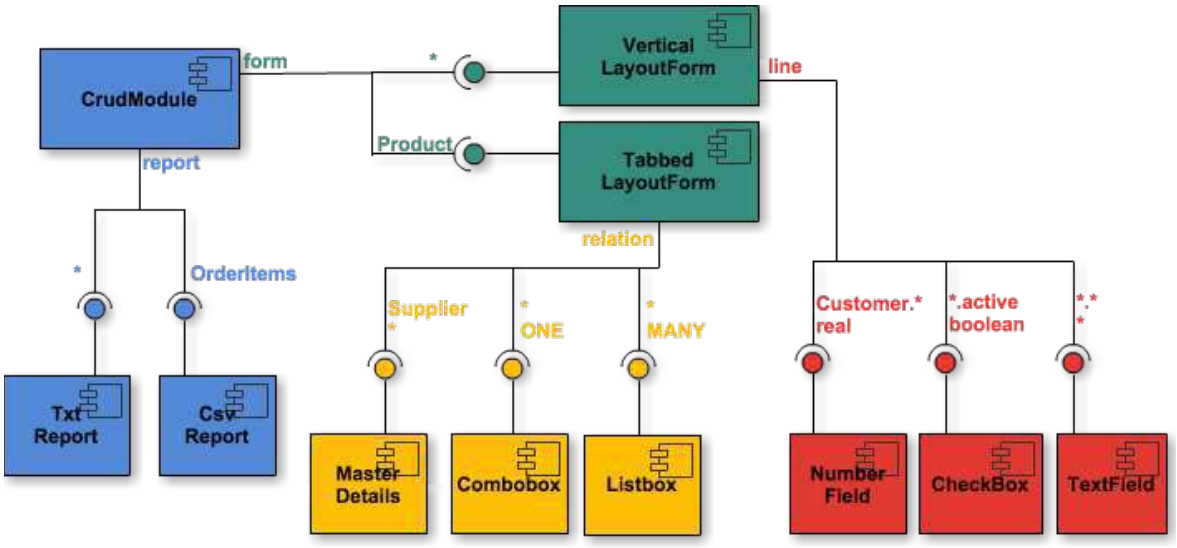


Figura 4.8: Aplicando a dimensão escopo nas ligações entre portas e componentes

Logo, a plataforma de GUI deve verificar e notificar a ausência das regras padrão para não correr o risco de não poder escolher *widgets* para alguma entidade, propriedade ou relacionamento.

Consequências

- O escopo padrão define os *widgets* genéricos para boa parte da GUI e para estágios iniciais da GUI;
- Escopos específicos servem a necessidades específicas da GUI e têm prioridade sobre o escopo padrão;
- Os desenvolvedores podem alterar facilmente os *widgets* padrão e específicos, selecionando novos *widgets* a partir dos metadados de entidades, propriedades e relacionamentos;
- O uso da dimensão escopo adiciona complexidade à ligação indireta entre *widgets*, dificultando a compreensão do código.

Usos conhecidos

Os *templates* padrão de *Ruby on Rails* e *Grails* podem ser alterados. Para tanto, deve-se exportar os *templates* em uma pasta específica, alterá-los manualmente e gerar o código novamente para as entidades onde a mudança se aplica.

ASP.NET Dynamic Data tem *Fields* padrão que podem ser customizados[6].

Padrões relacionados

WIDGET SCOPE flexibiliza a ligação de INDIRECT WIDGETS.

WIDGET ENGINE pode facilitar a configuração do escopo ao exportar os dados da Tabela 4.4 para arquivos de configuração ou bancos de dados.

4.9 Padrão de projeto PARAMETRIZABLE WIDGET

Sinônimos: PLUGGABLE WIDGET, WIDGET PARAMETERS

A GUI de aplicações corporativas tem muitas peculiaridades que se repetem em várias telas. Alguns *widgets* são praticamente iguais, diferindo em características pequenas, tais quais, cor, visibilidade, *layout*, validação, etc.

Uma forma rápida para implementar essas peculiaridades é estender os *widgets* existentes ou realizar interfaces definidas pelo arcabouço de GUI. Todavia, a implementação de muitas peculiaridades de GUI pode levar a uma explosão de classes que se diferenciam de maneira trivial, como está exemplificado na Listagem 4.9.

Por outro lado, o comportamento que diverge um pouco do existente em outro componente pode ser implementado através de estruturas de seleção e parâmetros, em vez de cópia ou herança.

Problema

Como evitar a criação de *widgets* triviais cada vez que se implementa peculiaridades de GUI?

[6] https://msdn.microsoft.com/en-us/library/cc488533.aspx

Forças

- Quando se cria novas classes, a complexidade do sistema ou do arcabouço aumenta;
- Estruturas de seleção e parâmetros reduzem a legibilidade do código e aumentam o acoplamento de controle.

Código Fonte 4.9: Exemplo de *widgets* muito parecidos

```
1  class CelulaSimples
2    desenhar(propriedade):
3      return "<td>" + propriedade.valor + "</td>"
4
5  class CelulaNegrito
6    desenhar(propriedade):
7      return "<td><b>" + propriedade.valor + "</b></td>"
8
9  class CelulaItalico
10   desenhar(propriedade):
11     return "<td><i>" + propriedade.valor + "</i></td>"
12
13 class CelulaAzul
14   desenhar(propriedade):
15     return "<td><font color='blue'>" + propriedade.valor + "</font></td>"
```

Solução

Projetar widgets que possam ser configurados, cada vez que são usados, a fim de selecionar o comportamento que deve ser executado.

Este padrão pode ser aplicado sobre INDIRECT WIDGETS, adicionando parâmetros de configuração que definem o comportamento dos *widgets* cada vez que eles são ligados às portas. O exemplo da Figura 4.9 usa o mesmo *widget* `TextField` três vezes na mesma porta *field*. Cada vez que um *widget* é ligado a uma porta, seu comportamento pode ser modificado através dos parâmetros de configuração. Nesse exemplo, os parâmetros de configuração adicionam uma máscara e alteram a cor do campo em escopos específicos.

A Tabela 4.5 estende a estrutura da Tabela 4.4, adicionando a coluna *Configuração* para montar a árvore de *widgets* da Figura 4.9.

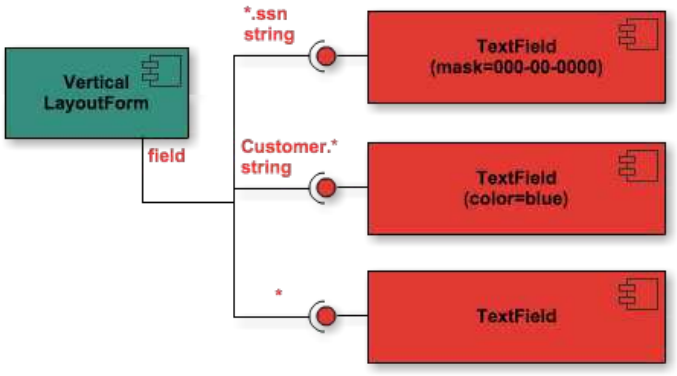


Figura 4.9: Usando dados de configuração para parametrizar componentes de GUI

Tabela 4.5: Estrutura e exemplos de dados de configuração para montar a Figura 4.9

Porta				**Widget**	***Configuração***
Nome	Tipo	Tipo de Propriedade	Escopo		
field	Property	string	*.ssn	`TextField`	*mask=000-00-0000*
field	Property	string	Customer.*	`TextField`	*color=blue*
field	Property	*	*	`TextField`	

Esse padrão causa acoplamento de controle nos *widgets*, devido às estruturas de seleção que decidem o comportamento a ser executado a partir da configuração dos parâmetros. Para diminuir esse problema, os desenvolvedores podem refatorar [33] o código dos *widgets*, isolando as estruturas de seleção em métodos para reduzir o acoplamento.

O padrão PARAMETRIZE WIDGETS viabiliza o uso de abordagens poderosas para a adaptação de GUI em tempo de execução. Por exemplo, um *widget* pode definir uma URL na configuração que apontará para uma biblioteca (por exemplo, em *JavaScript*), que pode ser carregada e aplicada sobre o próprio *widget*, modificando a sua apresentação e comportamento sem demandar a alteração do código fonte da GUI.

Na Listagem 4.10, pode-se ver o código da Listagem 4.9 refatorado após se aplicar o padrão PARAMETRIZE WIDGETS. O código do *widget* se tornou mais complexo, mas, para o programador que vai utilizá-lo ele se tornou mais simples de configurar (apenas por parâmetros) e capaz de combinar as configurações, como se pode ver no *widget w5*.

Consequências

- Este padrão reduz o número de classes na GUI e facilita o reúso de *widgets* em diferentes escopos e portas;
- No entanto, gera acoplamento e demanda refatoramento.

Código Fonte 4.10: Exemplo de *widget* parametrizável

```
1  class CelulaParametrizavel
2    constructor(private configuration)
3
4    desenhar(propriedade):
5      pre = "<td>"
6      pos = "</td>"
7      if (configuration.negrito)
8        pre = pre + "<b>"
9        pos = "</b>" + pos
10     if (configuration.italico)
11       pre = pre + "<i>"
12       pos = "</i>" + pos
13     if (configuration.cor)
14       pre = pre + "<font color='" + configuration.cor + "'>"
15       pos = "</font>" + pos
16     return pre + propriedade.valor + pos
17
18 Widget w1 = new CelulaParametrizavel( {} )
19 Widget w2 = new CelulaParametrizavel( { negrito: true } )
20 Widget w3 = new CelulaParametrizavel( { italico: true } )
21 Widget w4 = new CelulaParametrizavel( { cor: 'blue' } )
22 Widget w5 = new CelulaParametrizavel(
23   { negrito: true, italico: true, cor: 'blue' } )
```

Usos conhecidos

Quando desenvolvedores usam os PADRÕES DE RENDERIZAÇÃO AOM [99], parâmetros podem ser passados para diferenciar os *widgets*.

Em *Ruby on Rails*, um gerador pode chamar outro passando parâmetros [78].

Padrões relacionados

PARAMETRIZE WIDGETS é uma especialização de PLUGGABLE OBJECTS [83].

Cada ramo de porta criado pelo uso do padrão WIDGET SCOPE pode definir uma configuração diferente de *widget*.

Os *widgets* projetados conforme o padrão INDIRECT WIDGET podem ser diferenciados com PARAMETRIZABLE WIDGET.

4.10 Padrão de projeto RELATIONSHIP RENDERER

Sinônimos: RELATION WIDGET, METADATA RELATIONS

Neste trabalho, investigamos diversas plataformas de GUI para definir os padrões. Embora muitas delas definam artefatos genéricos para desenhar GUI de propriedades e entidades, em apenas três casos encontramos componentes baseados em metadados para relacionamentos entre entidades. Em muitas abordagens, como os padrões de renderização AOM e a maior parte dos arcabouços de *Scaffold*, não existe esse tipo de *widget*.

No entanto, os relacionamentos entre entidades são uma parte importante das aplicações corporativas, sendo representados em diversos pontos da GUI por componentes como *Combo box*, Caixas de seleção múltiplas, painéis *master-details*, entre outros. Portanto, também há duplicação de código de relacionamentos na GUI, que também pode ser abstraído em artefatos baseados em metadados.

Problema

Como definir componentes genéricos de GUI baseados em metadados de relacionamentos?

Forças

- Nas plataformas que ignoram a geração de código para relacionamentos, os desenvolvedores precisam implementar GUI de relacionamentos manualmente;
- PROPERTY RENDERERS [99] podem ser adaptados para desenhar relacionamentos, porém não podem usar todos os metadados dos relacionamentos, como cardinalidade e navegabilidade;

- Os *widgets* para relacionamentos aumentam a complexidade da plataforma de GUI, mas viabilizam o amplo reúso de código de relacionamentos.

Solução

*Estender os **padrões de renderização AOM** a fim de definir componentes baseados em metadados para relacionamentos.*

Em AOM, alguns PROPERTY RENDERERS podem ser adaptados para tratar relacionamentos, porém o padrão DYNAMIC RELATIONS [71] mostrou que existem dados específicos de relacionamentos que não são utilizados pelos *widgets* de propriedades. Dessa forma, este padrão indica que devem existir *widgets* específicos para desenhar GUI relativa a relacionamento de entidades.

Dentre os metadados exclusivos de relacionamento, a Cardinalidade tem um papel importante, pois define a quantidade de instâncias que podem ser selecionadas num relacionamento entre objetos de duas entidades, alterando as características visuais da GUI. Por exemplo, em um mesmo formulário de *Pedido* pode haver dois componentes de relacionamento totalmente distintos: um `Combo box` para selecionar o *Cliente* e um `List box` para selecionar os vários *Vendedores* que participaram do pedido. Um outro exemplo de componentes para relacionamento pode ser visto na Figura 4.8 (destacados na cor amarela).

Consequências

- Desenvolvedores podem criar *widgets* específicos para desenhar relacionamentos na GUI, utilizando todos os metadados dos relacionamentos presentes no modelo do domínio;
- O código da plataforma de GUI deve ser mais complexo para suportar o processamento dos metadados de relacionamentos.

Usos conhecidos

Django define *fields* para gerar GUI de relacionamentos[7].

[7] http://www.djangobook.com/en/2.0/chapter10.html

O arcabouço TERESA [68] usa *widgets* — `object`, `single_choice`, `multiple_choice` e `object_edit`— para tratar os relacionamentos entre entidades na GUI.

O arcabouço *SwingBean* utiliza componentes para desenhar listas para relacionamentos[8].

Padrões relacionados

RELATIONSHIP RENDERER estende os padrões de renderização AOM e podem ser encapsulados como DOMAIN WIDGETS.

O padrão DYNAMIC RELATIONS [71] confirma a necessidade de tratar especificamente os metadados de relacionamentos.

MASTER-DETAIL PRESENTATION [67] é um RELATIONSHIP RENDERER específico para relacionamentos representados através de componentes *Master-details*.

4.11 Considerações finais do capítulo

Neste capítulo, sete padrões de projeto e dois padrões arquiteturais foram descritos a partir da análise de soluções de mercado ou acadêmicas para o desenvolvimento de GUI em aplicações corporativas. Os padrões foram catalogados individualmente no formato *Gang of Four* e serão inter-relacionados no próximo capítulo.

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

Capítulo 5

Linguagem de padrões - Relacionamentos e arquitetura de referência

No capítulo 4, apresentou-se um catálogo com nove padrões para metacomponentes de GUI em aplicações corporativas. Todavia, catálogos se limitam ao conhecimento sobre padrões isolados. Segundo Alexander, padrões isolados são apenas boas ideias não relacionadas [3]. Salingaros [87] afirma que palavras sem relacionamento não formam uma linguagem, portanto padrões devem ser combinados a fim de formar padrões de mais alto nível, que são chamados de linguagens de padrões. Dado que as combinações possíveis entre os padrões, dentro de um catálogo, são geralmente infinitas, uma das utilidades de uma linguagem de padrões é delimitar quais combinações devem ser feitas. Além disso, uma linguagem de padrões pode ser representada visualmente através de um grafo, no qual os padrões são os nós e os relacionamentos são as arestas. Dessa forma o conhecimento adquirido sobre um determinado domínio pode ser documentado através das combinações possíveis dentro do conjunto de soluções para os problemas desse domínio.

Neste capítulo, apresenta-se uma linguagem de padrões que enriquece a compreensão e a utilidade dos padrões já catalogados. O público alvo da linguagem de padrões aqui proposta são os arquitetos de software que projetam arcabouços de GUI para aplicações corporativas. Para desenvolver essa linguagem, foram utilizados os métodos descritos no trabalho de Dearden e Finlay [25] no escopo de linguagem de padrões para GUI e interação

homem-máquina. Ademais, outras referências citadas nesse trabalho foram consultadas para basear o método de construção e aplicação da linguagem de padrões. O resultado dessa pesquisa está na próxima seção.

O restante deste capítulo se desenvolve apresentando os relacionamentos da linguagem de padrões proposta, uma arquitetura de referência para implementação de arcabouços de GUI baseados nessa linguagem e um mapa de soluções que representa visualmente como os padrões podem ser aplicados.

5.1 Pesquisa sobre relacionamentos em linguagens de padrões

Segundo Dearden e Finlay [25], o principal trabalho sobre padrões na engenharia de software — o livro cujos autores ficaram conhecidos como *Gang of Four* [35] — era apenas um catálogo de padrões classificados por seu tipo, de modo que os autores esperavam que o catálogo fosse evoluído para se tornar uma linguagem de padrões. No entanto, como esses padrões são genéricos, podem se relacionar a padrões de várias linguagens de padrões em diversos domínios diferentes. Por exemplo, neste trabalho o padrão COMPOSITE é citado no padrão INDIRECT WIDGET, e os padrões ABSTRACT FACTORY, TEMPLATE METHOD e STRATEGY se relacionam com o padrão WIDGET ENGINE.

Van Welie and van de Veer [93] propuseram três relacionamentos possíveis entre padrões, no contexto de uma linguagem de padrões: especialização, na qual um padrão herda elementos de um padrão de mais alto nível; agregação, onde um padrão é contido em outro padrão; e associação, quando um padrão utiliza outro. Buschmann (2001) define o termo *completa* para expressar o relacionamento entre padrões.

No artigo *User Interface Software* [20], Jens Coldewey define alguns padrões para GUI e como eles se relacionam: USER INTERFACE LAYER é um padrão arquitetural e contém alguns padrões menores (SEPARATE TRANSFORMATION, WIDGET MODEL, CONTEXT SUPPORT e DOMAIN LAYER ACCESS); o padrão CONTEXT SUPPORT pode ser implementado através de vários outros padrões (APPLICATION, DOCUMENT, SELECTION, SESSION MEMORY e COOKIE); e DOMAIN LAYER ACCESS também pode ser implementado através de outros padrões (OBSERVER, COMMAND, AVAILABILITY METHOD e DOMAIN LEVEL

TYPE).

No trabalho *Synthesizer, A Pattern Language for Designing Digital Modular Synthesis Software* [51], Judkins e Gill definem uma linguagem de padrões com relacionamentos temporais na aplicação dos padrões. Por exemplo, se houver a relação A -> B, então o padrão A deve ser aplicado previamente para depois se aplicar B. Berczuk et. al. [14] definiram uma linguagem de padrões com um relacionamento no estilo A -> B, onde o padrão A precisa do padrão B para ser completo. Além disso, eles representaram visualmente a diferença dos padrões definidos no próprio trabalho e os padrões oriundos de outros trabalhos. De fato, Beck e Cunningham já afirmavam que a conexão e a interdependência entre os padrões é importante pois uma decisão tomada na aplicação de um padrão influencia nos padrões que serão aplicados posteriormente [13].

Por fim, Richardson [80] definiu uma linguagem de padrões para componentes *Web J2EE*, detalhando a forma como ela pode ser aplicada. Nesse caso, pode-se fazer pequenos refatoramentos, aplicando padrões isolados, ou pode-se mudar toda a arquitetura do sistema, aplicando todos os padrões numa sequência específica.

Em resumo, as lições aprendidas nesses trabalhos sobre o desenvolvimento de linguagens de padrões para GUI foram:

- Os padrões definidos neste trabalho podem se relacionar com padrões já existentes, e deve-se diferenciar esses dois tipos de padrões na representação visual da linguagem;
- Os relacionamentos possíveis entre os padrões são: especialização, agregação, associação, complementação, implementação e dependência;
- Padrões com granularidades diferentes podem se relacionar, por exemplo, padrões arquiteturais e padrões de projeto;
- Os padrões podem ser aplicados isolada ou completamente (neste caso, seguindo uma sequência predeterminada pela linguagem).

Essas lições foram aplicadas na linguagem de padrões proposta neste trabalho, a fim de definir o relacionamento entre os padrões e guiar os cenários onde a sua aplicação é realizada.

5.2 Relações entre os padrões

Na Figura 5.1 sumariza-se a abordagem proposta neste trabalho para o aumento do reúso de código em interfaces gráficas de aplicações corporativas. Nesse diagrama, estão presentes os nove padrões que foram propostos no capítulo 4 (caixas com fundo branco), a forma como esses padrões se relacionam, e os principais padrões presentes na literatura (caixas com o fundo cinza) com os quais os padrões propostos aqui se relacionam.

O padrão basilar dessa linguagem de padrões é DOMAIN WIDGET, que é uma especialização de METADATA-BASED GRAPHICAL COMPONENT e pode ser implementado como um TEMPLATE VIEW. Além disso, DOMAIN WIDGET representa uma generalização dos padrões de renderização de AOM (PROPERTY RENDERER, ENTITY VIEW e DYNAMIC VIEW) e do padrão RELATIONSHIP RENDERER (e sua especialização MASTER-DETAILS PRESENTATION).

Para gerar GUI, os DOMAIN WIDGETS precisam ter acesso aos metadados do domínio, utilizando o padrão DOMAIN MODEL X RAY, que, por sua vez, pode implementar diversas formas de leitura de metadados utilizando os padrões METADATA READER STRATEGY ou STRATEGY. O padrão FACADE pode ser utilizado para unificar a interface dos serviços que provêm acesso aos metadados. INSTANCE PRESENTATION e SERVICE PRESENTATION são especializações de DOMAIN MODEL X RAY. DYNAMIC RELATIONS pode ser utilizado para definir os metadados de relacionamentos que serão consumidos pelos RELATIONSHIP RENDERERS.

O padrão INDIRECT WIDGET remove o acoplamento entre DOMAIN WIDGETS concretos, utilizando portas em vez de referências diretas entre eles. Essas portas devem ser povoadas através de INVERSION OF CONTROL pelo padrão especializado WIDGET ENGINE, que também se torna responsável por decidir quando os DOMAIN WIDGETS devem ser instanciados e invocados para desenhar a GUI. Uma biblioteca de INDIRECT WIDGETS pode ser disponibilizada aplicando-se o padrão COMPONENT LIBRARY. O relacionamento da WIDGET ENGINE com os DOMAIN WIDGETS pode ser implementado através de STRATEGY ou TEMPLATE METHOD. Além disso, a WIDGET ENGINE pode ser uma ABSTRACT FACTORY.

Dois padrões podem flexibilizar ainda mais os INDIRECT WIDGETS: WIDGET SCOPE permite diferenciar *widgets* padrão e *widgets* que são específicos para alguns fragmentos

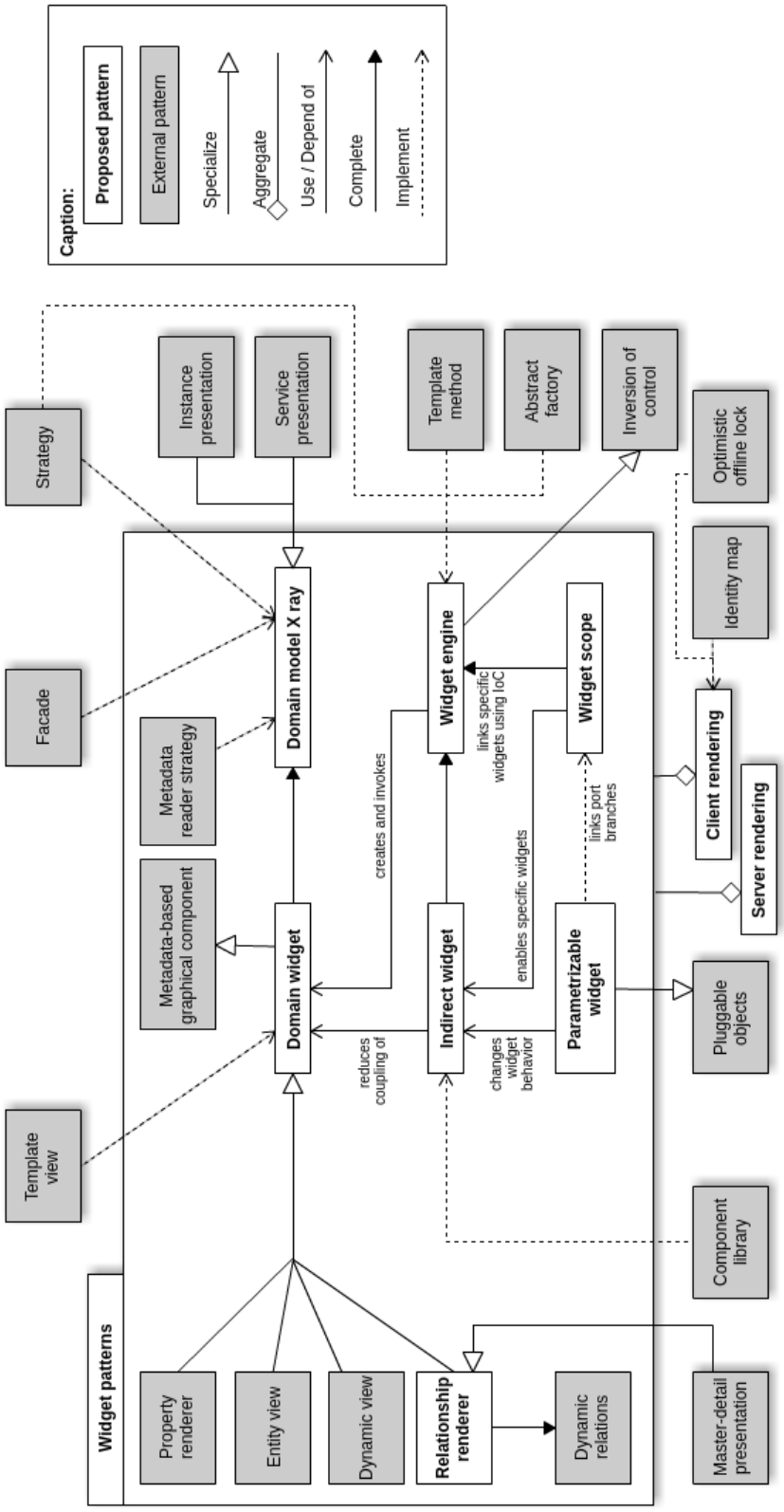


Figura 5.1: Linguagem de padrões para *widgets* em aplicações corporativas

da GUI; e PARAMETRIZABLE WIDGET (especialização de PLUGGABLE OBJECTS) permite modificar o comportamento de um *widget* toda vez que ele é ligado a alguma porta ou ramo de porta.

Porém, a primeira decisão que se deve tomar ao projetar o arcabouço diz respeito a qual dos padrões arquiteturais — SERVER RENDERING ou CLIENT RENDERING — utilizar. Essa decisão é importante, pois definirá onde os padrões do pacote *Widget patterns* serão implementados: no servidor *web* ou no navegador, respectivamente.

Para melhorar o desempenho e diminuir o tráfegos de rede, o padrão CLIENT RENDERING deve implementar uma *cache*, que pode utilizar os padrões IDENTITY MAP ou OPTIMISTIC OFFLINE LOCK.

5.3 Arquitetura de referência

As linguagens de padrões são artefatos de alto nível que documentam soluções recorrentes em um domínio e não precisam, obrigatoriamente, ser completas nem seguir algum conjunto de requisitos. Por outro lado, as arquiteturas de referência são artefatos mais concretos, com uma estrutura unificada que pode suprir as lacunas das linguagens de padrões com elementos inovadores, a fim de definir soluções completas para um domínio.

Guerra e Nakagawa propuseram um processo para construir arquiteturas de referência a partir de linguagens de padrões [41], que consiste de: unificar os componentes participantes dos padrões da linguagem em uma mesma estrutura, mesclando os componentes que aparecem em padrões diferentes; propor soluções novas ou particulares para os padrões que ainda não estiverem definidos na linguagem; escolher apenas um padrão entre padrões padrões mutuamente exclusivos da linguagem.

Seguindo esse processo, a arquitetura de referência apresentada na Figura 5.2 foi criada para guiar a construção de arcabouços de GUI, em aplicações corporativas, renderizada no cliente *web* através de artefatos com granularidade fina baseados nos metadados do modelo do domínio.

O elemento central desta arquitetura é a classe `WidgetEngine`, que deve encapsular a complexidade do arcabouço e expor apenas a sua interface para criação de GUI. Essa classe é responsável por coordenar os cinco passos que formam o fluxo de trabalho do arcabouço

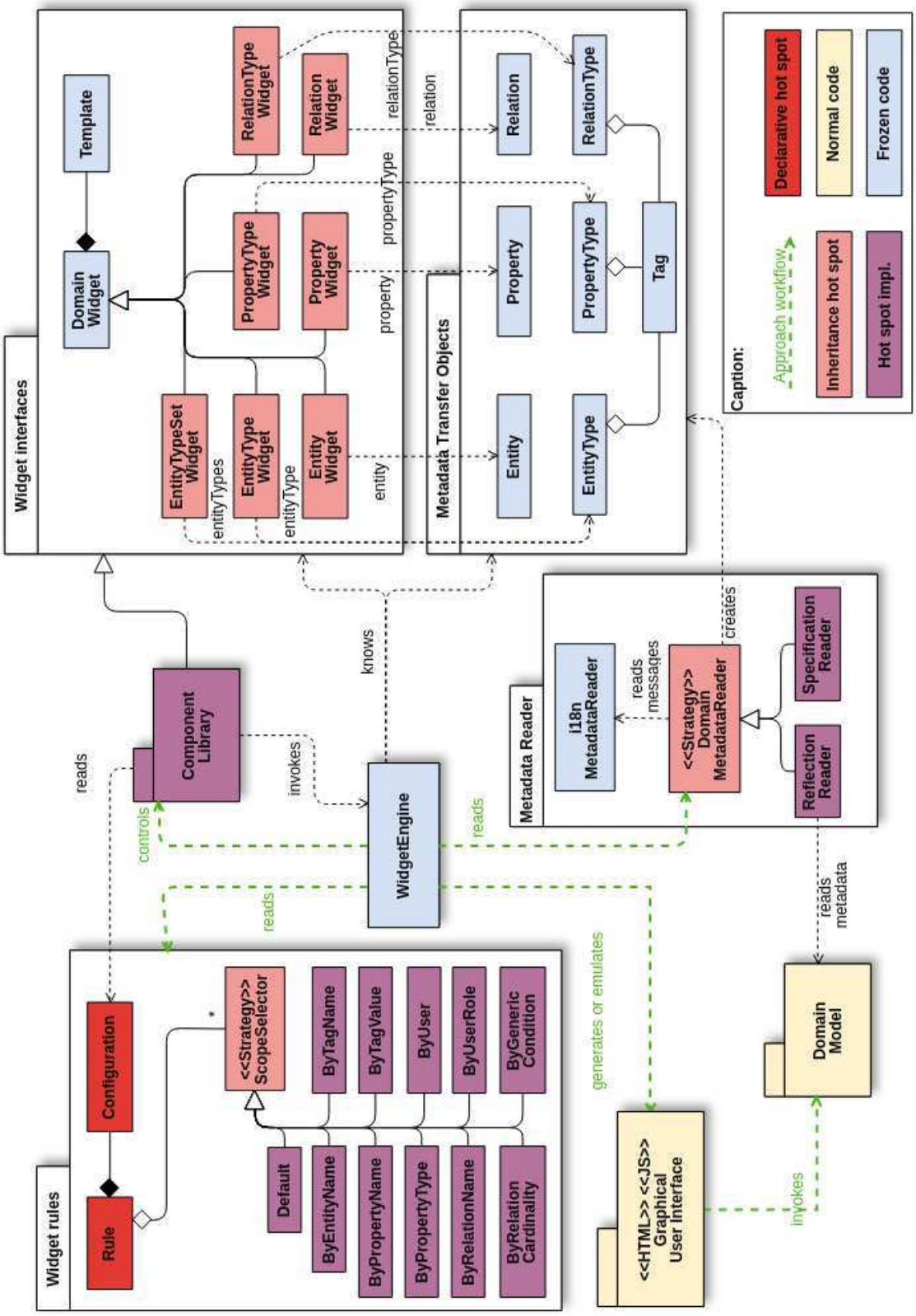


Figura 5.2: Linguagem de padrões para *widgets* em aplicações corporativas

(e estão destacados em verde):

1. Ler os metadados do modelo do domínio, sob responsabilidade do pacote `Metadata Reader`, que cria e retorna `Metadata transfer objects`;
2. Ler as regras de configuração dos componentes, sob responsabilidade do pacote `Widget rules`;
3. Controlar a criação e execução de componentes de GUI na `Component Library`, todavia sem conhecê-los diretamente, utilizando apenas as interfaces definidas pelo pacote `Widget interfaces`;
4. Gerar o código HTML e *JavaScript* da GUI comum (com referências diretas ao domínio) ou emulá-lo;
5. Controlar a invocação da GUI ao domínio.

Na arquitetura de referência, os elementos em azul claro são fixos e não precisam ser estendidos para construir a GUI. Os elementos em amarelo claro representam o código comum da aplicação que não utiliza metadados. Por outro lado, os elementos em vermelho são os *hot spots* do arcabouço, ou seja, devem ser expandidos a fim de criar e customizar a GUI. Os elementos vermelhos mais claro são estendidos através da herança orientada a objetos e os vermelhos mais escuros são construções declarativas que podem ser implementadas mais facilmente, por exemplo, com linguagens de domínio específico. As classes em roxo representam implementações dos pontos de extensão que já estão definidas na arquitetura de referência.

O pacote `Metadata Transfer Object` (*MTO*) se baseia no padrão DATA TRANSFER OBJECT (*DTO*) para trafegar dados entre as camadas do sistema. Porém, como a estrutura dos metadados não muda (sempre terá entidades, propriedades e relacionamentos) o pacote *MTO* é congelado, diferente do *DTO*, que varia para toda entidade do sistema. O pacote MTO implementa o padrão TYPE SQUARE para representar entidades e propriedades e implementa o padrão DYNAMIC RELATIONS para os metadados de relacionamentos.

Além disso, foi adicionada a funcionalidade de etiquetas (*tags*) aos tipos de entidades, propriedades e relacionamentos com o propósito de anotar o domínio com informações de

negócio que podem influenciar a GUI. Embora não tenha sido documentado um padrão para essa funcionalidade na linguagem de padrões, as *tags* são importantes para o arcabouço de GUI, pois podem ser usadas para tipificar metadados. Por exemplo, demarcando os campos como CPF, senha, máscara ou chave primária; auxiliando na validações dos dados (campos obrigatórios ou com limites); ou definindo que uma entidade representa o usuário e os papeis do sistema (para autenticação e autorização).

No pacote `Metadata Reader`, pode-se substituir facilmente o mecanismo de obtenção dos metadados do domínio, pois foi aplicado o padrão STRATEGY na interface `DomainMetadataReader`. Por exemplo, dois mecanismos de leitura de metadados implementam essa interface: `ReflectionReader`, que utiliza reflexão para extrair metadados a partir do modelo orientado a objetos do domínio; e `SpecificationReader` que permite ao desenvolvedor descrever os metadados programaticamente para o arcabouço. Novos mecanismos para leitura de metadados podem ser implementados ao realizar a interface `DomainMetadataReader`. Independente da estratégia utilizada, essa interface define que devem ser criados e retornados, para a *engine*, metadados no formato do pacote `Metadata Transfer Object`.

Internacionalização é um aspecto importante das aplicações corporativas que não foi coberto pela linguagem de padrões. A estruturação padrão de metadados auxilia a criação de mensagens internacionalizadas, pois os elementos do domínio podem receber chaves de mensagens internacionalizáveis automaticamente. Por exemplo, para toda entidade do domínio, podem ser criadas as chaves: `${entidade}.nome.singular` e `${entidade}.nome.plural`. O mesmo se aplica às propriedades e aos relacionamentos do domínio, que também possuirão chaves baseadas nos metadados. As chaves devem ser traduzidas para as línguas utilizadas na aplicação, através de arquivos de configuração ou no próprio banco de dados. E, ao requisitar os metadados, a *engine* pode informar o *locale* vigente para o usuário e a classe `i18nMetadataReader` povoa os metadados com as chaves já traduzidas.

O pacote `Widget rules` é responsável por fornecer uma estrutura para a configuração dos componentes de GUI através de regras declarativas. Basicamente, uma regra possui um tipo, pertence a uma porta, é vigente em um escopo e seleciona qual *widget* deve ser aplicado na porta. A seguir, será descrito como implementar essa funcionalidade.

Cada tipo de metadado possui um conjunto de regras separado; portanto existe um mapa com as regras de cada porta para `EntityTypes`, outro para `Entities`, outro para `PropertyTypes` e assim por diante.

As regras são aplicadas em escopos gerais ou específicos de acordo com o escopo das regras. A seleção do escopo pode ser implementada como um STRATEGY nas regras. Para o escopo padrão, pode-se utilizar o `ScopeSelector Default`. Também existem seletores por nome da entidade, nome da propriedade, tipo da propriedade, nome do relacionamento e cardinalidade do relacionamento.

A Figura 5.2 apresenta mais alguns seletores de escopo que não foram previstos na linguagem de padrões, mas são importantes para a implementação de sistemas concretos. Os seletores por nomes ou valores de *tags* podem utilizar anotações do domínio para, por exemplo, aplicar validações de campos na GUI. Os seletores por usuário ou papel do usuário podem ser utilizado para customizar a GUI de acordo com o nível de permissão do usuário logado na aplicação. Por fim, o seletor `ByGenericCondition` pode ser utilizado para implementar um tipo de escopo não previsto, por exemplo, que os campos string com tamanho maior que 100 devem utilizar *textareas*.

O padrão PARAMETRIZABLE WIDGET define parâmetros de configuração para *widgets* que podem ser alterados para cada regra. Essa funcionalidade está implementada na arquitetura de referência através do objeto `Configuration` que é lido pelo *widget* selecionado por cada regra.

A regras representam o principal ponto de customização do sistema. Daí a importância do projeto dessas classes que foram flexibilizadas a fim de viabilizar a customização até por linguagens de domínio específico, sem demandar a alteração de código fonte.

O pacote `Widget interfaces` implementa vários padrões de projetos para definir classes abstratas para componentes baseados em metadados: DOMAIN WIDGET (na classe `Domain widget`), TEMPLATE VIEW (`Template`), DYNAMIC VIEW (`EntityTypeSetWidget` e `EntityTypeWidget`), ENTITY VIEW (`EntityWidget`), PROPERTY RENDERER (`PropertyTypeWidget` e `PropertyWidget`) e RELATIONSHIP RENDERER (`RelationTypeWidget` e `RelationWidget`). Essas classes possuem referências para os seus respectivos metadados no pacote `Metadata Transfer Object`.

As classes definidas no pacote `Component library` herdam dos `widgets` abstratos definidos no pacote `Widget interfaces` e representam os *widgets* que desenharão trechos da GUI da aplicação corporativa baseados em metadados do modelo do domínio. Apenas as classes desse pacote podem ser selecionadas pelas regras de `Widget rules`.

Após a combinação de metadados, regras e *widgets*, a `Widget engine` pode comandar a geração de código ou emulação da GUI (`Graphical User Interface`), que atua normalmente, invocando o domínio diretamente, como se tivesse sido feita sem auxílio de metadados.

Completa-se, portanto, o ciclo do arcabouço de GUI, que a despeito de possuir componentes baseados em metadados e com granularidade fina, é capaz de executar a mesma funcionalidade de GUI normais.

5.4 Mapa de soluções

Na Figura 5.3, se apresenta um mapa para guiar a aplicação das soluções propostas pela Linguagem de padrões. No mapa, os padrões estão representados pelos retângulos e as setas representam os problemas que são solucionados pelos padrões.

Pode-se concluir que não existe uma sequência linear para aplicação dos padrões. Portanto é possível aplicar os padrões total ou parcialmente, a partir de pontos iniciais distintos, de acordo com os problemas que forem encontrados pelos arquitetos de software ao construirem arcabouços de GUI.

Por exemplo, ao criar um arcabouço de GUI o arquiteto de software deseja reduzir o acoplamento da camada de GUI para a camada de domínio, de modo que os componentes de GUI possam ser reutilizados em diferentes telas. Esse problema é solucionado pelo padrão DOMAIN WIDGET, que define componentes de GUI com referências a metadados em vez de referências diretas ao modelo do domínio.

Paralelamente o arquiteto de software pode decidir expor os metadados do modelo do domínio em uma interface de serviço, utilizando o padrão DOMAIN MODEL X RAY.

A junção desses dois primeiros padrões permite a criação de componentes de GUI que utilizam uma interface para consulta dos metadados do domínio. Essa é a estratégia utilizada pela maioria dos arcabouços que geram código de GUI, como *Ruby on Rails*, *Grails* e *Spring*

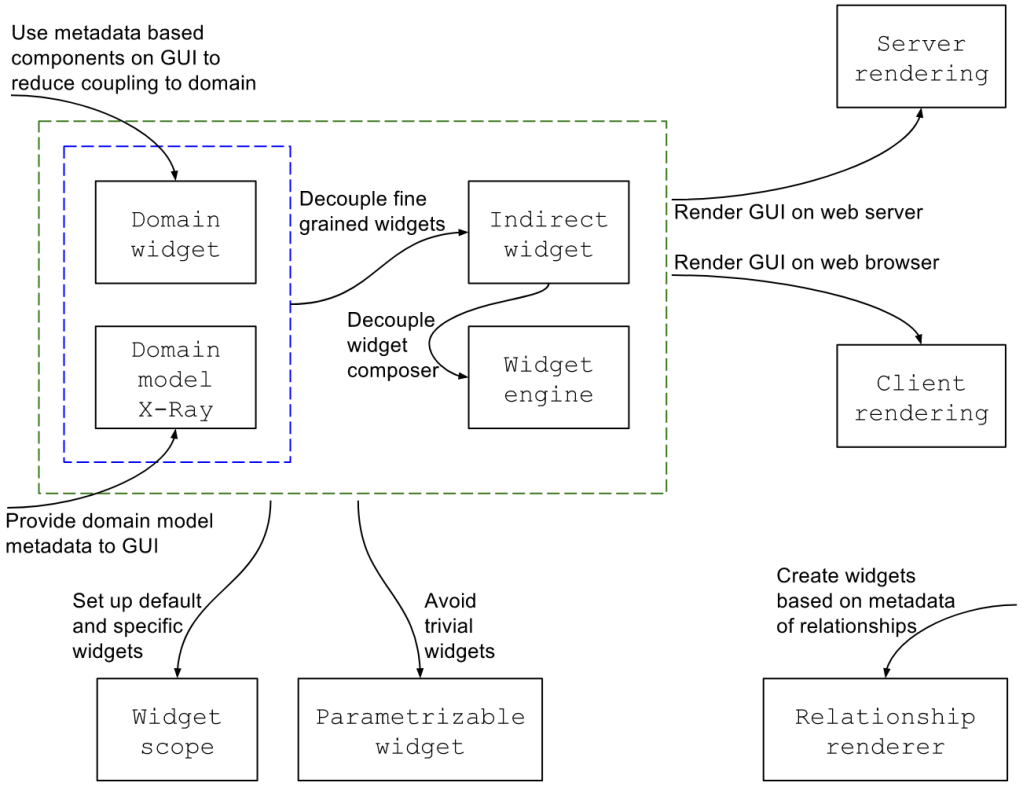


Figura 5.3: Mapa de soluções para a Linguagem de padrões

Roo. Por esse motivo, os dois padrões estão envolvidos por um retângulo azul, representando o estado da prática da maior parte das ferramentas e abordagens estudadas nesta pesquisa.

A partir desse ponto, o arquiteto de software pode decidir utilizar a abordagem proposta nesta tese, reduzindo o tamanho dos componentes de GUI e encapsulando as suas responsabilidades baseado nos metadados do domínio. Primeiramente deve-se aplicar o padrão INDIRECT WIDGET para remover o acoplamento entre os componentes de GUI. Depois disso pode-se aplicar o padrão WIDGET ENGINE padrão implementar inversão de controle e desacoplar o código que compõe os componentes (compositor) dos componentes propriamente ditos.

Os quatro padrões demarcados pelo retângulo verde representam o cerne da solução proposta nesta tese. Desse modo, é obrigatório que um arcabouço aplique esses quatro padrões para que seja classificado como uma solução que adota a abordagem proposta nesta tese. Isso ocorre, por exemplo, nos arcabouços descritos no capítulo 6.

Uma das principais decisões na construção de um arcabouço que implementa o padrão

WIDGET ENGINE é em que nó executar o processo de construção da GUI. Se o arquiteto decidir montar as páginas HTML no servidor web, deve então aplicar o padrão arquitetural SERVER RENDERING. Todavia, nas aplicações web mais modernas, os arcabouços desenham a GUI no navegador e o servidor possui apenas uma API REST. Neste caso, indica-se a aplicação do padrão arquitetural CLIENT RENDERING.

Se o arquiteto do arcabouço identificar a necessidade de customizar facilmente os componentes padrão da GUI para telas específicas, ele pode aplicar o padrão WIDGET SCOPE. Assim sendo, as regras de composição da GUI podem utilizar seletores para definir componentes customizados em fragmentos específicos da GUI.

Se os componentes de GUI criados por um arcabouço possuírem diferenças mínimas e houver a criação de uma grande quantidade de classes para representar a biblioteca de componentes de GUI, o arquiteto pode aplicar o padrão PARAMETRIZABLE WIDGET. Consequentemente os componentes podem ser parametrizados e a quantidade de classes pode diminuir.

Por fim, o padrão RELATIONSHIP RENDERER deve ser aplicado nos arcabouços que permitem a geração de código de GUI baseada nos metadados de relacionamentos do modelo do domínio. Esse padrão é totalmente independente da abordagem proposta nesta tese, mas pode ser combinado com ela para produzir arcabouços com componentes pequenos e específicos para GUI de relacionamentos.

No próximo capítulo, será descrita a implementação de dois arcabouços que possuem os padrões da linguagem proposta neste trabalho e seguem a arquitetura de referência da Figura 5.2.

Capítulo 6

Viabilidade técnica da linguagem de padrões

Com o propósito de verificar a viabilidade da linguagem de padrões proposta nos Capítulos 4 e 5, dois arcabouços de código aberto para desenvolvimento de aplicações corporativas foram implementados. Neste capítulo, apresentam-se os requisitos, a arquitetura e os detalhes de implementação desses arcabouços, chamados *Geneguis*[1] e *Angular M*[2], e se descreve como cada padrão foi aplicado neles. Além disso, algumas telas e trechos de código exemplificarão o reúso de GUI.

6.1 Requisitos dos arcabouços

O diagrama de casos de uso da Figura 6.1 apresenta os atores e as funcionalidades do *Geneguis* e do *Angular M*.

A primeira funcionalidade disponível para o ator Desenvolvedor é a criação de componentes (*widgets*) baseados em metadados do domínio. À medida que o conjunto de componentes cresce, eles podem ser reutilizados na maior parte da GUI e a necessidade de criar novos componentes diminuirá.

O próximo passo deve ser a adaptação da GUI (*Customize GUI*), mais especificamente o desenvolvedor deve definir quais são os componentes padrão para desenhar as telas para

[1] https://github.com/rodrigovilar/geneguis
[2] https://github.com/rodrigovilar/angularm

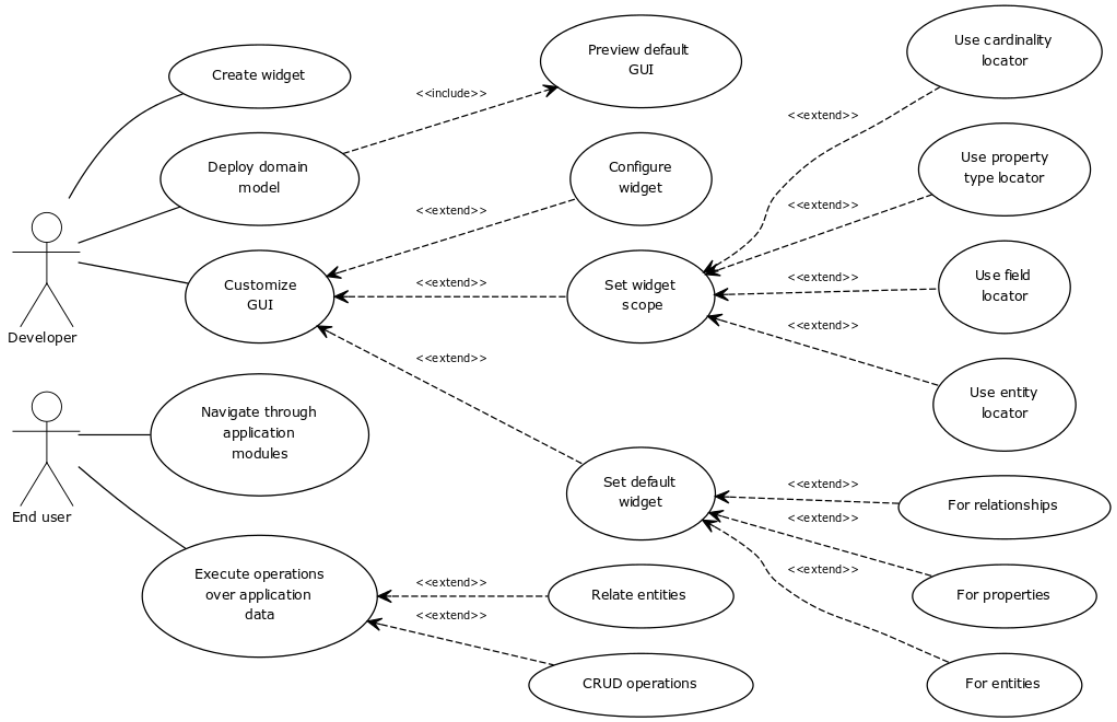


Figura 6.1: Diagrama de casos de uso dos arcabouços

entidades, propriedades e relacionamentos. Os requisitos de relacionamentos foram implementados apenas no *Geneguis*.

Nesse ponto, quando o desenvolvedor implantar o Modelo do domínio, o arcabouço deve ser capaz de construir as telas da aplicação automaticamente. Essa GUI padrão pode ser prévisualizada pelo próprio desenvolvedor, a fim de que ele possa realizar e avaliar as adaptações na GUI.

A partir da construção automática da GUI, o usuário final pode navegar pelos módulos da aplicação e executar operação (CRUD e relacionamentos) sobre os dados da aplicação.

A qualquer momento, o desenvolvedor pode realizar novas adaptações na GUI que podem ser: modificações nos componentes padrão; modificações nos componentes com escopo delimitado por entidade, campo (termo genérico para propriedade e relacionamento dentro de uma entidade), tipo de propriedade e cardinalidade de relacionamento; modificações dos comportamentos dos componentes através da configuração de parâmetros.

Todas essas adaptações são refletidas na GUI do usuário final e visam atender às demandas desse ator com usabilidade otimizada.

Dentre os requisitos não funcionais do *Geneguis* e do *Angular M*, destacam-se o baixo tempo de resposta da GUI e o baixo tráfego de recursos na rede. Por isso, os arcabouços devem construir a GUI no navegador *web*.

6.2 Visão arquitetural

Os principais componentes da arquitetura do *Geneguis* estão distribuídos nos três nós da Figura 6.2: *browser*, que executa a camada de GUI (ou *frontend*) da aplicação; *server*, responsável pela lógica e modelo do domínio; e o banco de dados (*database*). Os *widgets* são componentes importantes em relação ao reúso de GUI, pois representam os metacomponentes de GUI que são armazenados no *server* e devem ser baixados pelo *browser* a fim de executar a construção da GUI.

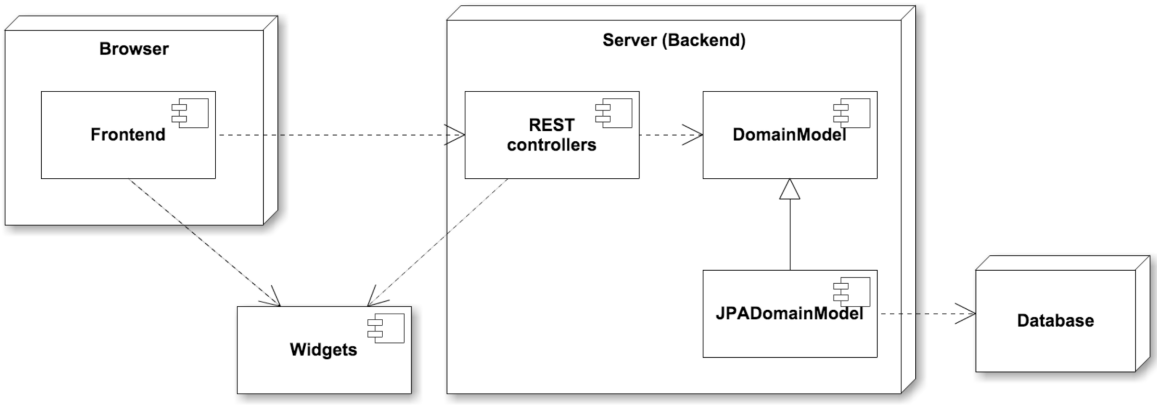


Figura 6.2: Diagrama arquitetural com os principais componentes do *Geneguis*

A comunicação entre *frontend* e *backend* segue a abordagem REST [26]. Portanto o *frontend* atua como um cliente REST e o *server* possui *controllers* que fornecem serviços REST.

O componente *DomainModel* é uma interface para acesso aos dados e metadados do modelo do domínio. Atualmente, existe uma implementação dessa interface que utiliza a tecnologia JPA [49] para acessar o banco de dados. Porém, outras implementações do modelo do domínio, como AOM [103] ou baseada em NoSQL [86] podem ser facilmente adicionadas, ao se realizar novamente a interface *DomainModel*.

Na Figura 6.3 pode-ser a arquitetura do arcabouço *Angular M*, que executa majoritariamente no *Browser* e podendo se conectar a *Backends REST* independente da tecnologia.

Diferentemente do *Geneguis*, que é configurado no *Backend* e utiliza o *Frontend* para desenhar as telas da GUI, *Angular M* utiliza a versatilidade da plataforma *Angular 2+* para implementar a lógica da linguagem de padrões apenas no *Frontend*.

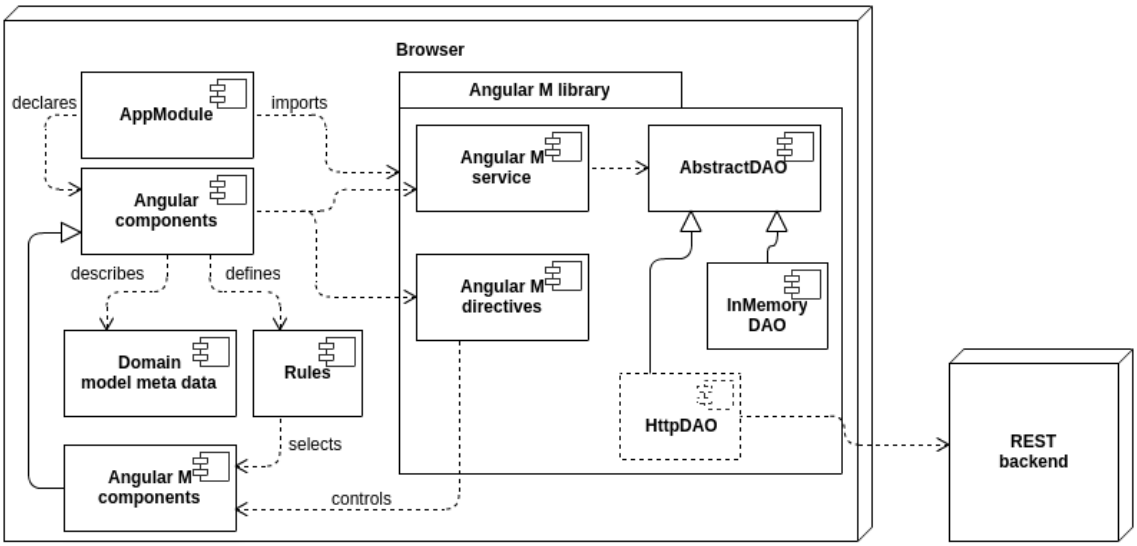


Figura 6.3: Diagrama arquitetural com os principais componentes do *Angular M*

Os elementos `AppModule` e `Angular components` pertencem à plataforma *Angular* e são responsáveis por importar e utilizar a biblioteca `angularm`, que está publicada no NPM[3].

Uma aplicação pode conter seus próprios componentes Angular M, que são especializações dos componentes comuns de *Angular*. Os `Angular components` são capazes de desenhar as telas da GUI a partir de metadados (`Domain model metadata`) e regras (`Rules`), sem se acoplarem diretamente ao Modelo do domínio. Sendo, portanto, facilmente reutilizáveis.

As diretivas *Angular M* são invocadas sempre que se delega a construção de um fragmento da GUI para um componente *Angular M*.

Quando um componente precisa manipular dados ou metadados, deve invocar a classe `AngularMService` que é a fachada do *Angular M*. A manipulação dos dados das Aplicações está encapsulada na interface `AbstractDAO`, que atualmente possui apenas uma

[3] www.npmjs.com/package/angularm

implementação que guarda os dados em memória. No entanto, para se comunicar com `Backends REST`, seria preciso apenas implementar um `DAO` que utilizasse requisições HTTP para invocar o *Backend*.

Uma diferença importante na arquitetura dos arcabouços *Geneguis* e *Angular M* é a relação de dependência entre o arcabouço e a aplicação. *Geneguis* é mais invasivo, pois a aplicação precisa ser desenvolvida totalmente sob a sua arquitetura. Por outro lado, *Angular M* pode ser enxertado em qualquer parte de uma aplicação que precise ser flexível, de um simples detalhe em uma tela à toda a aplicação podem ser feitos com esta tecnologia.

A divergência nas arquiteturas dos arcabouços Geneguis e *Angular M* demonstra a versatilidade da linguagem de padrões proposta nesta tese.

Interface entre os componentes

Uma vez que os componentes devem ser desacoplados, possuindo responsabilidades pequenas e bem definidas, os arcabouços *Geneguis* e *Angular M* provê um esquema de portas para implementar a comunicação indireta e flexível entre componentes. Ao declarar uma porta, um componente pai deve informar o nome da porta e o tipo dos componentes filhos esperados para atender a essa porta.

No entanto, a comunicação entre componentes deve respeitar a granularidade dos tipos dos componentes pai e filhos. Por exemplo, um componente de entidade pode invocar um componente de propriedade, mas o caminho inverso não é possível. Na Figura 6.4 são representadas as invocações possíveis entre tipos de componentes diferentes. Os comandos iniciados com `for` aparecem no código fonte dos componentes e abrem portas para a comunicação entre os componentes.

Inicialmente, o arcabouço define a porta *root* como o ponto de partida da GUI. Essa porta provê uma lista (`entityTypes`) com todos os tipos de entidades cadastradas nos metadados, e espera *widgets* do tipo `EntityTypeSet` para desenhá-los. Os tipos de entidades possuem os metadados necessários para, por exemplo, desenhar a lista dos módulos disponíveis na aplicação corporativa.

Os componentes do tipo `EntityTypeSet` podem invocar componentes do mesmo tipo, através do comando `forEntitytypes` que mantém o acesso dos componente filhos à lista `entityTypes` cadastradas. Ou podem invocar componente filhos do tipo `EntityType`

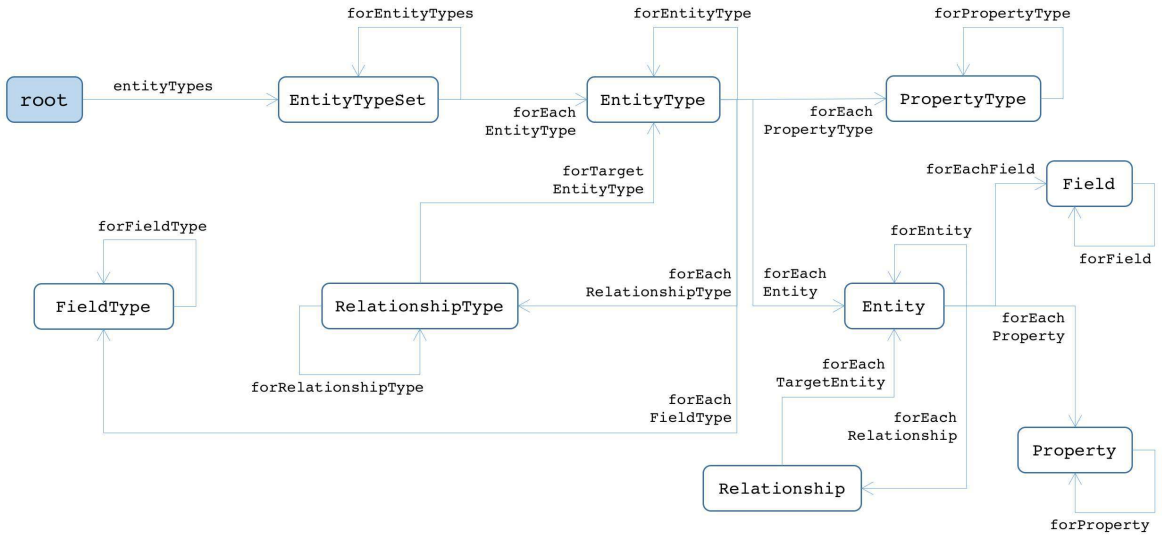


Figura 6.4: Comunicação entre os tipos de *widgets*

com o comando `forEachEntityType`. Nesse caso, o arcabouço realiza um laço sobre a lista `entityTypes` e invoca os componente filhos em cada iteração, fornecendo o índice `entityType` que contém os metadados de um tipo de entidade. Todos os comandos iniciados com `forEach` realizam um laço sobre alguma coleção de metadados.

Na Listagem 6.1, apresenta-se o código fonte de três componentes *Geneguis*: `EntityTypeList`, `EntityTypeItem` e `EntityTitle`. Além disso, demonstra-se a utilização dos comandos `for` para a comunicação entre os componentes. Na linha 4, pode-se ver o uso de um comando `for`, que possui a sintaxe: `{{'nome_porta' | comandoFor}}`. A linha 9 apresenta um comando `for` iniciado com `newPage`. Nesse caso, além de executar o comando `for`, abre-se conceitualmente uma nova página para desenhar a GUI do componente filho (`EntityTitle`), quando o elemento marcado com `newPage` for clicado. Nas linhas 8, 9, 12 e 13, o comando `{{name}}` é utilizado para acessar o nome do tipo da entidade que está sendo desenhada. Uma vez que essa informação e os comandos `for` são metadados, os três componentes da Listagem 6.1 podem ser reutilizados em qualquer aplicação corporativa.

A Listagem 6.2 possui código fonte de três componentes *Angular M* que produzem o mesmo resultado da Listagem 6.1. A diferença na sintaxe dos componentes se dá pelas linguagens utilizadas por cada arcabouço, cujos detalhes de implementação serão exibidos na seção 6.3.

Código Fonte 6.1: Exemplos de componentes *Geneguis* `EntityTypeSet` e `EntityType`

```
1  //Componente EntityTypeList do tipo EntityTypeSet:
2  <h2>Entities</h2>
3  <ul>
4    {{'entity_type_item' | forEachEntityType}}
5  </ul>
6
7  //Componente EntityTypeItem do tipo EntityType:
8  <li {{'entity_type_page' | newPageForEntityType}}>{{name}}</li>
9
10 //Componente EntityTitle do tipo EntityType:
11 <h2>{{name}}</h2>
12
13 //Regras utilizadas para ligar os componentes e as portas:
14   Ligar a porta 'entity_type_item' ao componente EntityTypeItem
15     no escopo padrao (*)
16   Ligar a porta 'entity_type_page' ao componente EntityTitle
17     no escopo padrao (*)
```

Os componentes do tipo `EntityType` podem iterar sobre quatro coleções ao invocar componentes filhos: `forEachPropertyType` itera sobre os tipos de propriedades do tipo de entidade atual; `forEachRelationshipType` itera sobre os tipos de relacionamentos do tipo de entidade; `forEachFieldType` itera genericamente sobre os tipos de propriedades e de relacionamentos; e `forEachEntity` itera sobre as entidades (instâncias) do tipo de entidade. Na Figura 6.4, o tipo genérico `Field` é utilizado quando o componente pai precisa iterar sobre propriedades e relacionamentos em uma mesma porta. Por fim, o comando `forEntityType` invoca componentes do mesmo tipo, mantendo o acesso dos componentes filhos à `entityType` atual.

Os componentes dos tipos `PropertyType` e `FieldType` só podem invocar componentes do mesmo tipo, através dos comandos `forPropertyType` e `forFieldType`, respectivamente, que repassam os tipo de propriedade e o tipo de campo atual para o componente filho. Por outro lado, os componentes do tipo `RelationshipType`, além de invocar componentes semelhantes com `forRelationshipType`, podem invocar componentes `EntityType` para desenhar a GUI relativa ao tipo de entidade no lado oposto do

relacionamento (*target*). Para tanto, deve-se utilizar o comando `forTargetEntityType`.

Código Fonte 6.2: Ex. de componentes *Angular M* `EntityTypeSet` e `EntityType`

```
1  //Trechos do componente EntityTypeList:
2  <h2>Entities</h2>
3  <ul>
4    <div [mgForeachEntityType]="'entity_type_item'"
5         [entityType]="entityType"></div>
6  </ul>
7  class EntityTypeListComponent extends EntityTypesComponent
8
9  //Trechos do componente EntityTypeItem do tipo EntityType:
10 <li (click)="open('entity_type_page')"> {{entityType.singular}} </li>
11 class EntityTypeItemComponent extends EntityTypeComponent {
12   open(port: string): void { }
13 }
14
15 //Trechos do componente EntityTitle do tipo EntityType:
16 <h2>{{entityType.singular}}</h2>
17 class EntityTitleComponent extends EntityTypeComponent
18
19 //Regras utilizadas para ligar os componentes e as portas:
20   entityTypeRule('entity_type_item', '*', EntityTypeItemComponent)
21   entityTypeRule('entity_type_page', '*', EntityTitleComponent)
```

Os demais componentes possuem acesso aos dados operacionais da GUI. Por exemplo, os componentes `Property` podem utilizar o comando `{{value}}` para imprimir o valor de uma propriedade na GUI. Além disso, esses componentes têm acesso ao objeto que define o tipo dos seus dados. Componentes `Entity` podem acessar o objeto *entityType*, componentes `Property` acessam *propertyType*, componentes `Relationship` acessam *relationshipType* e componentes `Field` acessam o tipo genérico *fieldType*.

Os componentes `Entity` podem iterar sobre três coleções ao invocar componentes filhos: `forEachProperty` itera sobre as propriedades da entidade atual; `forEachRelationship` itera sobre os relacionamentos da entidade; e `forEachField` itera sobre as propriedades e relacionamentos juntamente. Os componentes dos tipos `Property` e `Field` só podem invocar componentes do mesmo tipo, através dos coman-

dos `forProperty` e `forField`, respectivamente, que repassam a propriedade e o campo atual para o componente filho. Por outro lado, os componentes do tipo `Relationship`, além de invocar componentes semelhantes com `forRelationship`, podem invocar componentes `Entity` para desenhar a GUI relativa às entidades no lado oposto do relacionamento. Para tanto, deve-se utilizar o comando `forEachTargetEntity`.

O projeto *widgets*[4], disponível no *Geneguis*, possui exemplos de todos os tipos de componentes e de como eles se relacionam para montar telas funcionais de Aplicações corporativas.

Como a versão atual de *Angular M* (0.4.21) não possui metadados de relacionamentos, os comandos `forEachRelationshipType`, `forRelationshipType`, `forEachRelationship`, `forRelationship`, `forEachFieldType`, `forFieldType`, `forEachField` e `forField` ainda não foram implementados para essa tecnologia.

6.3 Detalhes de implementação

Geneguis

Na implementação do componente *server* (Figura 6.2) foram utilizadas as tecnologias: *Spring Boot*[5] para embarcar um servidor *web*, definir um processo padrão para automação de *build* com *Maven*[6], configurar as bibliotecas *Spring* (segurança, persistência, mapeamento de URI, etc.); e *JPA* para o mapeamento objeto-relacional no acesso ao banco de dados. Para ler os metadados do domínio, o componente `JPADomainModel` utiliza a API de reflexão de Java para ler as anotações de JPA, que demarcam as entidades do domínio e a forma como elas se relacionam.

No *frontend* a linguagem utilizada foi *CoffeeScript*[7] que é compilada para *JavaScript*, portanto pode ser executada em *browsers*. *CoffeeScript* possui um estilo orientado a objetos mais parecido com linguagens de *backend*. Dessa forma, *CoffeeScript* facilita a aplicação de padrões de projeto no código do *frontend* [72].

[4] https://github.com/rodrigovilar/geneguis/widgets
[5] http://projects.spring.io/spring-boot/
[6] https://maven.apache.org/
[7] http://coffeescript.org/

Os componentes de GUI foram implementados como *templates* da linguagem *Nunjucks*[8], cujo motor de *templates* pode executar em navegadores *web* como *JavaScript* nativo. Porém, a principal característica que diferencia *Nunjucks* das demais linguagens de *templates* existentes é a capacidade de gerar código HTML de forma assíncrona, demanda real das páginas que utilizam *Ajax* [36].

No repositório do *Geneguis* também existe um projeto chamado *e2e*, que contém os testes automatizados fim a fim (end-to-end) de todos os componentes da arquitetura. Esses testes utilizam *Selenium*[9] para invocar a interface gerada para as aplicações de teste e *Apache Http Client*[10] para realizar as invocações REST, e para configurar a GUI no *backend*.

Angular M

Angular M é uma tecnologia baseada fortemente no arcabouço *Angular 2+*[11], de modo que os componentes *Angular M* são especializações (herança orientada a objetos) de componentes *Angular*. A linguagem *TypeScript*[12] é utilizada no código executável dos componentes *Angular* e os *templates* para geração de HTML possuem sintaxe própria com invocação de componentes, diretivas e *pipes*.

6.4 Uso dos padrões da linguagem

A listagem a seguir descreve como os padrões, contidos na linguagem de padrões aqui proposta, foram utilizados na implementação do *Geneguis* e do *Angular M*.

- DOMAIN WIDGET

  Os componentes do *Genesis* e do *Angular M* não possuem referências diretas ao modelo do domínio da aplicação corporativa. Em vez disso, são utilizados elementos baseados em metadados, conforme está determinado neste padrão.

[8] http://mozilla.github.io/nunjucks/
[9] http://www.seleniumhq.org/
[10] https://hc.apache.org/
[11] angular.io
[12] www.typescriptlang.org/

Como consequência, os componentes podem ser reutilizados na mesma aplicação ou em aplicações diferentes.

- TEMPLATE VIEW

  Em vez de concatenar *strings* para formar a GUI em HTML, os componentes do *Geneguis* são *templates* que aceitam comandos demarcados no seu código. Semelhantemente os componentes do *Angular M* possuem *templates* que aceitam diretivas no seu código. Assim sendo, nos dois arcabouços os desenvolvedores utilizam uma linguagem mais alto nível e produtiva para criar componentes.

- DOMAIN MODEL X-RAY

  O arcabouço *Geneguis* fornece objetos, a partir do *backend*, com os metadados para serem consumidos pelos componentes, escondendo a forma e o momento em que esses metadados foram obtidos. Desse modo, o arcabouço pode implementar uma *cache* de metadados sem afetar o código dos componentes.

  Em *Angular M*, os desenvolvedores descrevem os metadados através de código *TypeScript* no *frontend*.

- INDIRECT WIDGET

  Os comandos `for` implementam este padrão no *Geneguis*, como também fazem as diretivas de *Angular M*, criando portas para os componentes pai se comunicarem com os seus componentes filhos. O código do *frontend*, nos dois arcabouços, atua como o objeto externo que liga os componentes às portas, conforme as configurações definidas em regras.

- WIDGET ENGINE

  O *frontend*, nos arcabouços *Geneguis* e *Angular M*, representa o motor que utiliza inversão de controle para instanciar, compor e coordenar a comunicação dos componentes. Para executar suas funcionalidades, o motor consulta as regras que estão armazenadas no *backend* do *Geneguis* e no *frontend* do *Angular M*.

  Nos dois arcabouços, o motor não referencia os componentes diretamente. Em vez disso, conhece apenas as interfaces ou superclasses dos tipos de componen-

tes: `EntityTypeSet`, `EntityType`, `PropertyType`, `RelationshipType`, `FieldType`, `Entity`, `Property`, `Relationship` e `Field`.

- WIDGET SCOPE

  O uso do padrão WIDGET SCOPE, nos dois arcabouços, afetou a estrutura das regras e o mecanismo de inversão de controle no *frontend*. As regras ganharam uma dimensão a mais, chamada escopo, que pode ser padrão (*), estabelecendo os componentes para casos gerais, ou pode referenciar as entidades, propriedades ou relacionamentos específicos onde as ligações têm validade.

  Com o uso deste padrão, os desenvolvedores podem facilmente substituir os componentes padrão por componentes específicos em alguns pontos da GUI da aplicação corporativa.

- PARAMETRIZABLE WIDGET

  Este padrão também adicionou uma dimensão nas regras, chamada configuração, que pode ser diferente em cada associação porta–componente. Dessa forma, o mesmo componente pode ser reutilizado em vários pontos da GUI, modificando o seu comportamento conforme a configuração definida em cada regra.

- DYNAMIC VIEW

  Este padrão de renderização AOM é implementado, no *Geneguis* e no *Angular M*, pelos componentes do tipo `EntityType`, a fim de desenhar fragmentos de GUI baseados em metadados para conjuntos de entidades.

- ENTITY VIEW

  Este padrão de renderização AOM é implementado, no *Geneguis* e no *Angular M*, pelos componentes do tipo `Entity`, a fim de desenhar fragmentos de GUI baseados em metadados para uma entidade.

- PROPERTY RENDERER

  Este padrão de renderização AOM é implementado, no *Geneguis* e no *Angular M*, pelos componentes dos tipos `PropertyType` e `Property`, a fim de desenhar fragmentos de GUI baseados em metadados para uma propriedade.

- RELATIONSHIP RENDERER

  Este padrão é implementado apenas no *Geneguis* pelos componentes dos tipos `RelationshipType` e `Relationship`, a fim de desenhar fragmentos de GUI baseados em metadados para um relacionamento.

- CLIENT RENDERING

  Ao aplicar este padrão no *Geneguis*, o código dos componentes e os seus dados de composição são armazenados no *server*. O *frontend*, que executa no navegador *web*, atua como um arcabouço responsável por baixar e executar os componentes.

  O arcabouço *Angular M* executa totalmente no *frontend*, portanto sua arquitetura implementa naturalmente este padrão.

- COMPONENT LIBRARY

  No momento, o projeto `Geneguis/widgets` representa uma implementação simplista de uma biblioteca com os *widgets* já desenvolvidos para o Geneguis.

  Por padrão, uma biblioteca de componentes que implementa *CRUDs* semelhantes ao *Scaffolding* de *Ruby on Rails* está disponível no *Angular M*. Essa biblioteca de componentes pode ser estendida.

6.5 Exemplos de reúso de GUI

Nesta seção, lista-se alguns exemplos de GUI implementada com os arcabouços *Geneguis* e *Angular M*. Esses exemplos mostram o código de GUI sendo reutilizado e alterado para domínios diferentes e os respectivos resultados visuais na GUI.

O código da Listagem 6.3 exibe os componentes necessários para montar uma tabela de listagem independente do Modelo de domínio. A aplicação desse código sobre duas entidades diferentes (`CustomerDetails` e `Product`) pode ser vista na Figura 6.5.

A mesma funcionalidade implementada em *Angular M* pode ser vista na Listagem 6.4.

Código Fonte 6.3: Componentes *Geneguis* formando uma tabela de listagem

```
1  //Componente ListingTable do tipo EntityType:
2  <h2>{{name}}</h2>
3  <table border=1>
4    <thead>
5      <tr> {{'table_head' | forEachFieldType}} </tr>
6    </thead>
7    <tbody> {{'table_line' | forEachEntity}} </tbody>
8  </table>
9
10 //Componente TableHead do tipo FieldType:
11 <th> {{name}} </th>
12
13 //Componente TableLine do tipo Entity:
14 <tr> {{'line_cell' | forEachField}} </tr>
15
16 //Componente TableCell do tipo Field:
17 <td> {{value}} </td>
18
19 Regras utilizadas para ligar os componentes e as portas:
20   Ligar a porta 'table_head' ao componente TableHead
21     no escopo padrao (*)
22   Ligar a porta 'table_line' ao componente TableLine
23     no escopo padrao (*)
24   Ligar a porta 'line_cell' ao componente TableCell
25     no escopo padrao (*)
```

CustomerDetails

id	ssn	name	credit
1	ssn1	name1	1
2	ssn2	name2	2

(a) Listagem de clientes

Product

id	barCode	name	description	price
1	00123451	Product 1	Description of product 1	1
2	00123499	Product 2	Description of product 2	2
3	00123777	Product 3	Description of product 3	3.5

(b) Listagem de produtos

Figura 6.5: Reúso de componentes em tabelas de listagem

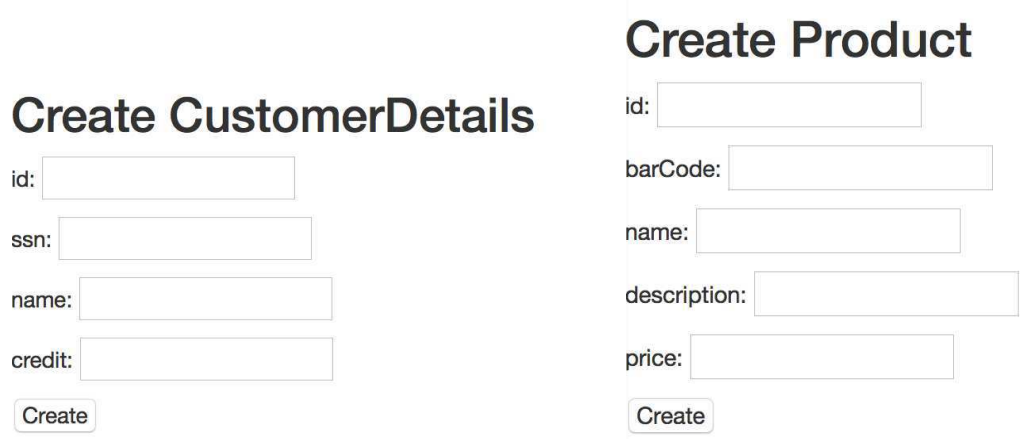
Código Fonte 6.4: Componentes *Angular M* formando uma tabela de listagem

```
1  //Trechos do componente ListingTable:
2  <h2> {{entityType.plural}} </h2>
3  <table border=1>
4    <thead>
5      <div [mgForeachPropertyType]="'table_head'"
6           [entityType]="entityType"></div>
7    </thead>
8    <tbody>
9      <div [mgForeachEntity]="'table_line'"
10          [entityType]="entityType"></div>
11   </tbody>
12 </table>
13 class ListingTableComponent extends EntityTypeComponent
14
15 //Trechos do componente TableHead:
16 <th> {{propertyType.name}} </th>
17 class TableHeadComponent extends PropertyTypeComponent
18
19 //Trechos do componente TableLine:
20 <tr>
21   <div [mgForeachProperty]="'line_cell'" [entity]="entity">
22 </tr>
23 class TableLineComponent extends EntityComponent
24
25 //Trechos do componente TableCell:
26 <td> {{property.value}} </td>
27 class TableCellComponent extends PropertyComponent
28
29 Regras utilizadas para ligar os componentes e as portas:
30   defaultPropertyTypeRule('table_head', TableHeadComponent)
31   entityRule('table_line', '*', TableLineComponent)
32   defaultPropertyRule('line_cell', TableCellComponent)
```

O segundo exemplo de reúso de componentes *Geneguis* pode ser visto no código da Listagem 6.5, que monta um formulário para criação de registros de qualquer entidade. A GUI resultante aparece na Figura 6.6. A mesma funcionalidade implementada em *Angular M* pode ser vista na Listagem 6.6.

Código Fonte 6.5: Componentes *Geneguis* formando um formulário de criação

```
1  //Componente CreateForm do tipo EntityType:
2  <h2> Create {{name}} </h2>
3  <form>
4    {{'form_line' | forEachPropertyType}}
5    <button {{'backPage' | post}}>Create</button>
6  </form>
7
8  //Componente FormLine do tipo PropertyType:
9  <p>{{name}}: <input type="text"/></p>
10
11 Regra utilizada para ligar o componente e a porta:
12   Ligar a porta 'form_line' ao componente FormLine
13     no escopo padrao (*)
```



(a) Formulário de clientes (b) Formulário de produtos

Figura 6.6: Reúso de componentes em formulários de criação

Código Fonte 6.6: Componentes *Angular M* formando um formulário de criação

```
1  //Trechos do componente CreateForm:
2  <h2> Create {{entityType.singular}} </h2>
3  <form [formGroup]="myForm"(ngSubmit)="onSubmit(myForm.value)">
4    <div [mgForeachPropertyType]="'form_line'"
5         [entityType]="entityType"></div>
6    <input type="submit" value="Create">
7  </form>
8  class CreateFormComponent extends EntityTypeComponent {
9    onSubmit(form: any): void { }
10 }
11
12 //Trechos do componente FormLine do tipo PropertyType:
13 <p>{{name}}: <input type="text"/></p>
14 class FormLineComponent extends PropertyTypeComponent
15
16 Regra utilizada para ligar o componente e a porta:
17   defaultPropertyTypeRule('form_line', FormLine)
```

A partir do terceiro exemplo, os componentes serão alterados para atender a novas demandas da GUI. Essas alterações também são facilmente reutilizadas em entidades diferentes. A Listagem 6.7 evolui o código dos componentes *Geneguis* das tabelas de listagem (Listagem 6.3), diferenciando o tratamento de células de propriedades (`TableCell`) e células de relacionamentos (`NamedRelation`). O resultado desta alteração pode ser visto na Figura 6.7. Na listagem de dependentes, o componente `NamedRelation` imprime o nome dos clientes (*Client* é a entidade *target* do relacionamento).

Código Fonte 6.7: Componente que usa um relacionamento para povoar a tabela de listagem

```
1  //Componente NamedRelation do tipo Relationship:
2  <td> {{target.name}} </td>
3
4  Regras utilizadas para ligar os componentes e as portas:
5    Ligar a porta 'line_cell' ao componente TableCell
6      no escopo de Property
7    Ligar a porta 'line_cell' ao componente NamedRelation
8      no escopo de Relationship
```

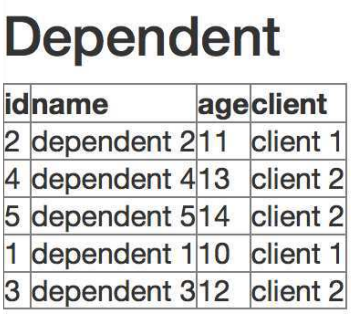
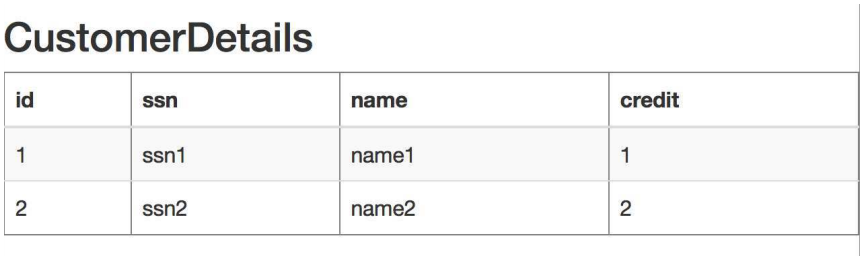


Figura 6.7: Exemplo de GUI com componente de relacionamento

O próximo exemplo demonstra a alteração dos componentes com a finalidade de adicionar a biblioteca *Bootstrap*[13] para melhoria do *design* da GUI. O código *Geneguis* alterado pode ser visto na Listagem 6.8 e a GUI resultante está na Figura 6.8.



(a) Listagem de clientes com *Bootstrap*

(b) Formulário de clientes com *Bootstrap*

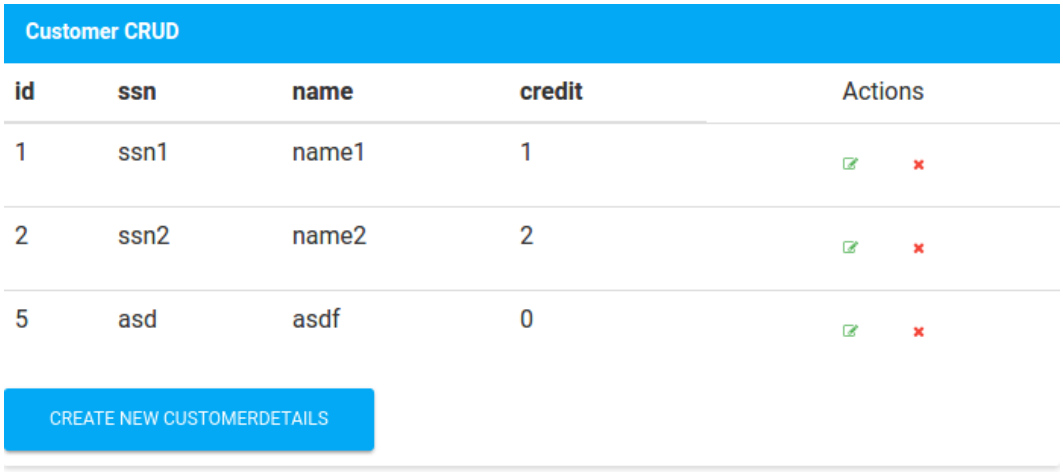
Figura 6.8: GUI resultante de componentes baseados em *Bootstrap*

[13] http://getbootstrap.com/components/

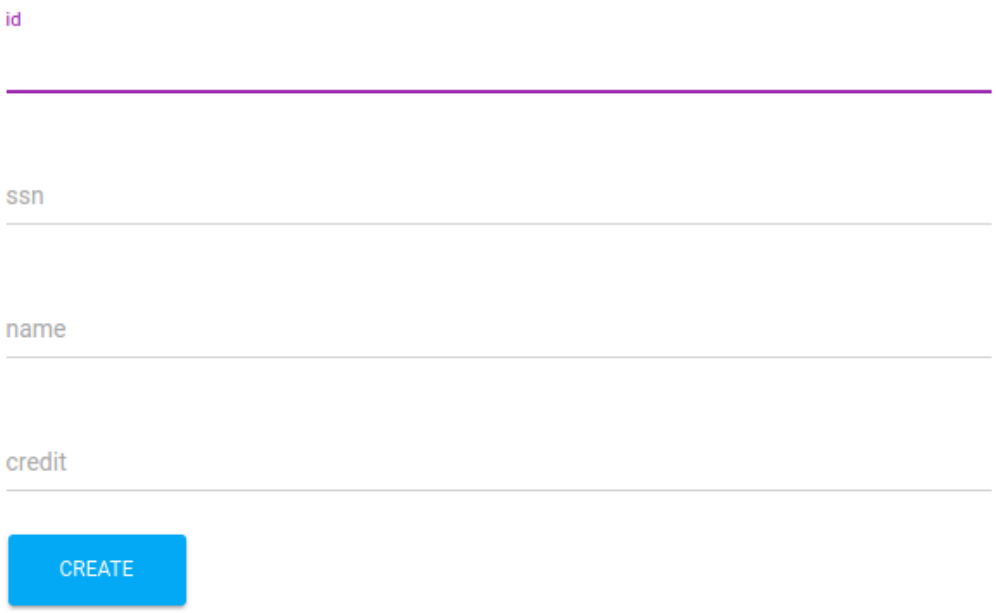
Código Fonte 6.8: Melhoria do *design* da GUI com componentes baseados em *Bootstrap*

```
1  //Componente ListingTableBootstrap do tipo EntityType:
2  <h2 id='title_{{name}}'>
3    {{name}}
4  </h2>
5  <table class="table table-striped" id='table_{{name}}' border=1>
6    <thead>
7      <tr>
8        {{'table_head' | forEachPropertyType}}
9      </tr>
10   </thead>
11   <tbody>
12     {{'table_line' | forEachEntity}}
13   </tbody>
14 </table>
15
16 //Componente CreateFormBootstrap do tipo EntityType:
17 <div class="container" style="max-width:600px;
18     max-height: 500px;float: none;
19     margin-left: auto;margin-right: auto;">
20   <form id='create_{{name}}'>
21     {{'form_line' | forEachPropertyType}}
22     <button type="submit" class = "btn btn-primary"
23       {{'backPage' | post}}>Create</button>
24   </form>
25 </div>
26
27 //Componente FormLineBootstrap do tipo PropertyType:
28 <div class = "form-group">
29   <label for="{{name}}">{{name}}</label>
30   <input type="text" name="create_form_field_{{name}}"
31     class="form-control" placeholder="Enter {{name}}">
32  </div>
```

O último exemplo também demonstra a alteração dos componentes, para adicionar a biblioteca *Material design*[14] na GUI. O código alterado pode ser visto na Listagem 6.9 e a GUI resultante está na Figura 6.9.



(a) Listagem de clientes com *Material design*



(b) Formulário de clientes com *Material design*

Figura 6.9: GUI resultante de componentes baseados em *Material*

[14]https://www.google.com/design/spec/material-design/introduction.html

Código Fonte 6.9: Melhoria do *design* da GUI com componentes baseados em *Material*

```
1  //Componente ListingTableMaterial do tipo EntityType:
2  <div class="panel panel-info">
3    <div class="panel-heading"><strong>Customer CRUD</strong></div>
4    <table class="table" id='table_{{name}}'>
5      <thead>
6        <tr>
7          {{'table_head' | forEachPropertyType}}
8          <td colspan="2" class="text-center">Actions</td>
9        </tr>
10     </thead>
11     <tbody>
12       {{'table_line' | forEachEntity}}
13     </tbody>
14   </table>
15   <button type="submit" class="btn btn-raised btn-info"
16     id='button_new_{{name}}' {{'creation_form' | newPageForEntityType}}>
17       Create new {{name}}
18   </button>
19 </div>
20
21 //Componente CreateFormMaterial do tipo EntityType:
22 <div class="container" style="max-width:600px;
23     max-height: 500px;float: none;
24     margin-left: auto;margin-right: auto;">
25   <form id='create_{{name}}'>
26     {{'form_line' | forEachPropertyType}}
27     <button type="submit" class="btn btn-raised btn-info"
28       {{'backPage' | post}}>Create</button>
29   </form>
30 </div>
31
32 //Componente FormLineMaterial do tipo PropertyType:
33 <div class="form-group label-floating is-empty">
34   <label class="control-label" for="{{name}}">{{name}}</label>
35   <input type="text" name="create_form_field_{{name}}" class="form-
       control">
```

```
36    <span class="material-input"></span>
37  </div>
```

Neste capítulo, foi confirmada a viabilidade da linguagem de padrões proposta nesta tese. Os arcabouços de código aberto *Geneguis* e *Angular M* implementam a linguagem de padrões e facilita a criação, evolução e reúso de componentes de GUI.

No próximo capítulo, se detalha a avaliação necessária para medir o impacto da linguagem de padrões no desenvolvimento de GUI em aplicações corporativas.

Capítulo 7

Avaliação

Após a proposição de uma linguagem de padrões e a sua implementação concreta no arcabouço *Angular M*, a avaliação deste trabalho foi realizada através de um experimento de customização de GUI em aplicações corporativas, com o intuito de detectar as diferenças quando a abordagem adotada é *Scaffolding* ou a linguagem de padrões aqui proposta.

Os conceitos da Engenharia de software experimental [52] foram utilizados para projetar, executar e analisar o experimento. As lições aprendidas pelo autor no estudo experimental sobre a Evolução de software não-antecipada [96, 97] serviram como base para este experimento e foram adaptadas a fim de medir o impacto real do *Angular M* na produtividade do desenvolvimento de GUI, sob o ponto de vista de desenvolvedores de software, no contexto de aplicações corporativas *web*.

No restante deste capítulo, são apresentados o design do experimento, sua preparação e execução. Os resultados são analisados estatisticamente e interpretados. Por fim, apresenta-se a análise das ameaças à validade do experimento.

7.1 Design do experimento

O design deste experimento está dividido em quatro partes:

1. Uma variável de entrada controlada – **Tecnologia**

   Ao avaliar o impacto do arcabouço *Angular M* na produtividade do desenvolvimento de GUI para aplicações corporativas *web*, foi decidido utilizar uma tecnologia real (e

de uso amplo na indústria) como referência para a métrica de produtividade. Para tanto, o arcabouço de *Scaffolding Ruby on Rails* foi escolhido como grupo de controle.

Em termos formais, a variável de entrada **Tecnologia** possui dois níveis (valores possíveis) — *Angular M* e *Ruby on Rails* — que serão distribuídos igualmente nas unidades experimentais.

2. Variáveis não desejadas – **Participante** e **Ambiente**

   O intuito deste experimento não é avaliar a produtividade dos participantes individualmente, nem por classes de acordo com sua experiência. No entanto, esse fator influencia diretamente no resultado de cada unidade experimental. Foi necessário, portanto, bloquear o efeito dessa variável não desejada através da seleção aleatória de tecnologias para cada participante em cada unidade experimental. Além disso, os participantes receberam um treinamento para nivelar o conhecimento sobre as duas tecnologias avaliadas.

   O design do experimento também buscou bloquear o efeito da variável **Ambiente**, ao utilizar um mesmo laboratório para executar o experimento e ao disponibilizar uma Máquina Virtual com as ferramentas de software necessárias para executar as atividades de customização de GUI, tanto em *Ruby on Rails* como em *Angular M*.

3. Unidades experimentais – **Tarefas de customização de tela**

   Foram elaboradas 12 tarefas de customização de GUI com características distintas a fim de comparar as duas tecnologias em diversos cenários diferentes. As tarefas possuíam três tipos possíveis de alteração de GUI: customização da GUI referente à Entidade, por exemplo, na tabela de listagem ou no formulário; customização da GUI referente à Propriedade, por exemplo, na célula da tabela de listagem que expõe o valor de uma propriedade ou em um campo de formulário; e customização da GUI referente à Entidade e Propriedade simultaneamente. Não foram testadas alterações referentes aos metadados de relacionamentos no modelo do domínio pois *Ruby on Rails* não provê *templates* padrão para esse tipo de metadado.

   Para as tarefas de customização da GUI referente à Entidade houve variação do escopo da customização: alterar a GUI para todas as entidades (customização default) ou

alterar apenas para algumas entidades (customização específica). Para as tarefas de customização da GUI referente à Propriedade, houve mais opções de escopo: alterar a GUI para todas as propriedades (customização default), alterar a GUI para apenas algumas propriedades (customização específica) ou alterar a GUI para algum tipo de propriedade (customização por tipo).

Outro aspecto considerado nas unidades experimentais foi o reúso de componentes. Devido à sua arquitetura, *Angular M* facilita a criação e o reúso de componentes referentes a trechos da GUI. O mesmo não acontece nos *templates* de *Ruby on Rails*, que são mais difíceis de entender, decompor e reutilizar. Por essa razão, em metade das tarefas, os participantes precisavam escrever os componentes *Angular M* do zero. Na outra metade, eles recebiam os componentes prontos para serem reutilizados.

A fim de simular as tarefa de customização em sistemas reais, as 12 telas listadas na Tabela 7.1 foram selecionadas e populadas nos metadados das versões *Ruby on Rails* e *Angular M*. A combinação de metadados com *templates* ou componentes foi realizada de forma que o código HTML final, para as 12 telas, foi idêntico independente de tecnologia. Sete dessas telas foram selecionadas a partir de uma pesquisa realizada com desenvolvedores sobre telas complexas de aplicações corporativas (Apêndice A). Todavia, como essa pesquisa foi realizada com programadores, as sugestões de telas complexas foram limitadas a ferramentas de apoio ao desenvolvimento de software. Por esse motivo o autor sugeriu outras cinco telas de outros tipos de aplicações corporativas, como por exemplo, sistemas de comércio eletrônico e ensino à distância.

Devido às características do experimento, o metamodelo das telas selecionadas sofreu as seguintes simplificações e adaptações:

- Foram adicionadas propriedades `code:string` para representar os campos de id, que estavam ocultos nas telas selecionadas;
- Campos referentes a Relacionamentos e *enumerations* foram transformados em *String* pois *Ruby on Rails* não fornece *templates* padrão para esse tipo de GUI;
- Campos binários foram ignorados por limitação de tempo para realizar o experimento;

Tabela 7.1: Telas utilizadas nas tarefas de customização de GUI

Aplicação	**Tela**	**Referência**	**Fonte**
Bugzilla	Bug	https://goo.gl/wwaQfH	Pesquisa com desenvolvedores
	User	https://goo.gl/FMEEDB	
	Product	https://goo.gl/9Nnfuw	
Jira	Issue	https://goo.gl/1Dep7B	
Magento	Product	https://goo.gl/ZBZuxS	
Redmine	Project	https://goo.gl/GoxvRC	
TestLink	Test Case	https://goo.gl/Lpnkrh	
Salesforce	User	https://goo.gl/ENM2A1	Sugestão do autor
	Lead	https://goo.gl/x3b5HS	
	Campaign	https://goo.gl/Tr8Eje	
Moodle	Quiz	https://goo.gl/q7xpTB	
	Course	https://goo.gl/TLwX5Z	

- Apenas customizações simples, sem *JavaScript*, foram realizadas, devido à recomendação de segurança de não injetar *scripts* em *templates* de componentes *Angular*.

Os recursos de suporte à customização (cópia dos *templates*, comandos de *Scaffold*, arquivos `entities.ts` e `rules.ts`) foram providos para as tarefas de customização, a fim de não influenciar no tempo da sua realização. Dessa forma, tentou-se reduzir o ruído que a manipulação desses recursos teriam sobre o experimento.

Em cada tarefa de customização de GUI do experimento, as telas da Tabela 7.1 deveriam ser alteradas da mesma forma, tanto em *Ruby on Rails* como em *Angular M*. De acordo com o escopo da tarefa, a alteração atuou sobre todas as telas ou apenas sobre um subconjunto específico. Assim sendo, um mesmo teste automatizado, escrito em *Selenium*[1], foi utilizado para verificar customizações nas duas tecnologias, antes de aceitar a submissão do código final pelo participante.

[1] http://www.seleniumhq.org/projects/webdriver/

4. Variáveis de saída – **Tempo gasto**, **Inspeção manual do código alterado** e **Opinião dos participantes**

   O tempo de início e fim de cada atividade foi coletado automaticamente por *Shell Scripts* escritos exclusivamente para este experimento. O *script* de inicialização baixa o código inicial de cada tarefa para *Ruby on Rails* ou para *Angular M*, demarcando o tempo de início da tarefa quando todas as dependências necessárias estavam resolvidas. E o *script* de finalização da tarefa executa um teste *Selenium* e, se for finalizado com sucesso, o tempo final da tarefa é marcado e o código final do participante é submetido para o repositório online *Github*, de onde foi coletado para posterior avaliação.

   Ao coletar e analisar a diferença de tempo gasto para realizar as mesmas tarefas com as duas tecnologias estudadas, pôde-se inferir o impacto do uso de *Angular M* na produtividade para o contexto deste experimento. Consequentemente, pôde-se extrair indícios para a comparação mais abstrata entre: tecnologias baseadas em metadados que utilizam componentes com responsabilidades entrelaçadas (*Scaffolding/Ruby on Rails*) *versus* tecnologias que encapsulam as responsabilidades dos metacomponentes (linguagem de padrões proposta aqui/*Angular M*).

   Também foi inspecionado manualmente o código resultante das alterações realizadas pelos participantes, com o intuito de auxiliar a compreensão dos resultados sobre o tempo gasto. A inspeção de código pôde, por exemplo, indicar quantas linhas de código e quantos arquivos precisaram ser alterados em cada tarefa do experimento.

   Além disso, foram feitas entrevistas com os participantes, no final das tarefas, a fim de coleta suas opiniões e sugetões sobre as tecnologias utilizadas e sua aplicação nas tarefas do experimento, o que também auxiliou na interpretação dos resultados.

7.2 Preparação do experimento

Nesta seção, estão detalhados os procedimentos realizados antes da execução das unidades experimentais com a finalidade de atender ao design do experimento.

Os participantes do experimento foram convidados através de uma chamada por email

na lista de colaboradores do Virtus/UFCG[2], que é um laboratório de Pesquisa & Desenvolvimento voltado para projetos da Lei de Informática. Os cerca de 200 profissionais do Virtus atendem clientes nacionais e multinacionais em projetos de Computação Móvel, Computação Embarcada, Internet das Coisas, Aplicações corporativas *desktop* e web, Jogos, entre outros.

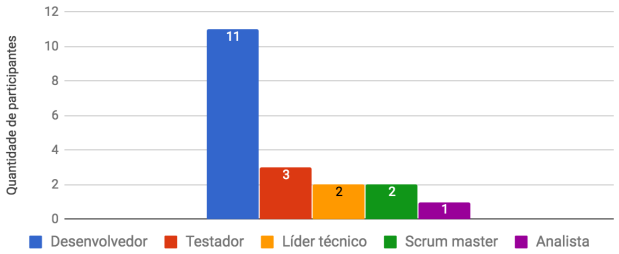
O principal atrativo para participar voluntariamente do experimento foi a oportunidade de adquirir experiência com *Angular 4*, uma das tecnologias que mais estava sendo demandada pelos clientes do Virtus desde o fim de 2016. Tanto que inicialmente 42 pessoas se inscreveram para o experimento.

A interferência da experiência do participante no experimento foi nivelada através de um curso semipresencial sobre *Ruby on Rails*, *Angular 4* e *Angular M*. 14 vídeo aulas foram preparadas e disponibilizadas no *Youtube*, somando cerca de 5 horas de conteúdo das três tecnologias. As Tabelas 7.2, 7.3 e 7.4 apresentam os detalhes das vídeo aulas. Essas informações foram enviadas para os inscritos no experimento, que puderam assisti-las no momento que lhes foi mais conveniente.

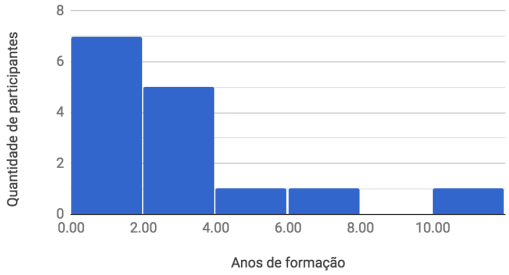
O conhecimento sobre cada aula foi avaliado através de exercícios presenciais. No momento da correção, que foi individual, houve a resolução de diversas dúvidas e a transmissão do conhecimento que não foi coberto pelas vídeo aulas. A conclusão correta de todos os exercícios foi pré-requisito para a participação no experimento propriamente dito. Dos 42 inscritos, 25 iniciaram o curso e 15 concluíram todos os exercícios, tornando-se aptos para participar do experimento.

O perfil dos participantes pode ser compreendido na Figura 7.1, cujos dados foram extraídos a partir de um formulário online utilizado para inscrição no experimento. As informações mais importantes sobre os participantes são: a maior parte atua como desenvolvedor (Figura 7.1a); 80% dos participantes possui menos de 4 anos de formação, o que configura uma população com experiência baixa ou intermediária (Figuras 7.1b, 7.1c e 7.1e); metade já conhecia a tecnologia *Angular* nas versões JS e 2+ (Figuras 7.1d e 7.1h); apenas dois participantes possuíam experiência com *Ruby on Rails* (Figura 7.1g); apenas um terço já tinha alguma experiência com metadados (Figura 7.1f).

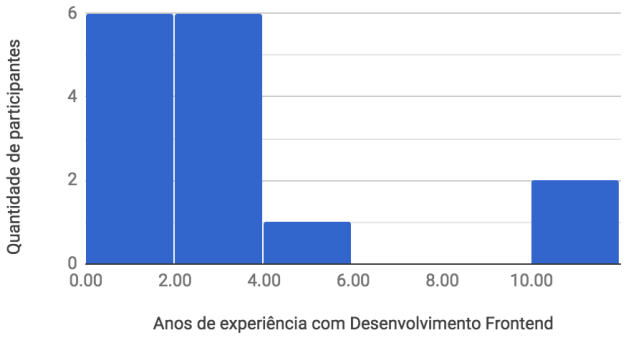
[2]http://virtus.ufcg.edu.br



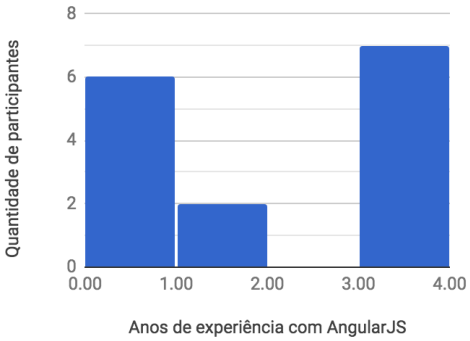
(a) Cargo



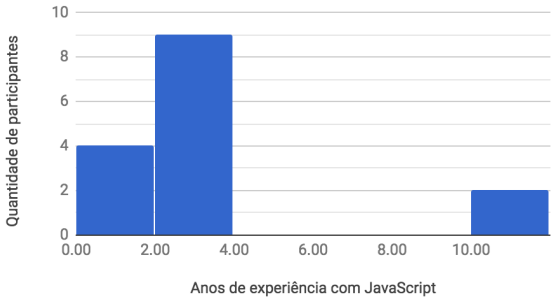
(b) Anos de formação



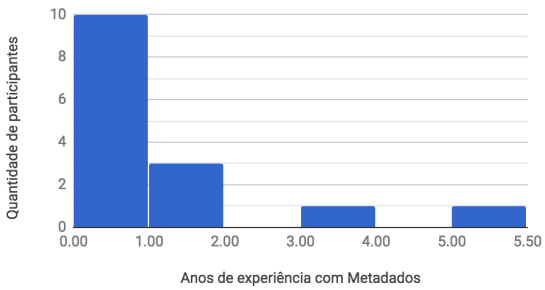
(c) Frontend



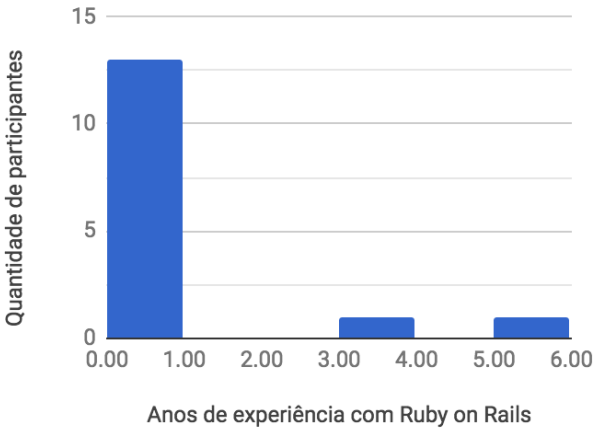
(d) AngularJS



(e) JavaScript



(f) Metadados



(g) Ruby on Rails

(h) Angular 2+

Figura 7.1: Histogramas com o perfil dos participantes do experimento

Tabela 7.2: Lista de aulas de Ruby on Rails: goo.gl/j5uYHk

Aula	**Vídeo**	**Slides**	**Exercícios**
Hello World	youtu.be/wgxaixyGU7I	goo.gl/zUfRvm	goo.gl/KPsZWM
CRUD app	youtu.be/JhyJlerLV-w	goo.gl/9mWr7i	goo.gl/7rXb3f
CRUD E2E Tests	youtu.be/SuKvXHrj-fI	goo.gl/ijmkQ4	goo.gl/qgyv8O
Code generation	youtu.be/U2rnorNKWzE		goo.gl/6h962o

Tabela 7.3: Lista de aulas de Angular: goo.gl/Yj0KUS

Aula	**Vídeo**	**Slides**	**Exercícios**
Hello World	youtu.be/f5Y4f2PUBfk	goo.gl/HgiRu0	goo.gl/cl4dux
Architecture	youtu.be/tPgd7-ggPkk	goo.gl/QXwRlq	goo.gl/yCq4Y6
Routing	youtu.be/rWBESxG-khE	goo.gl/VeI41L	goo.gl/7gTh77
TypeScript	youtu.be/JzcgF3ZDeDU	goo.gl/fjvtqX	goo.gl/s6KaXd
Built-in Directives	youtu.be/YGCpQduzsvs	goo.gl/uYRPi9	goo.gl/H44Jjx
CRUD app	youtu.be/GixOKkQL1vA	goo.gl/CPE0Wf	goo.gl/K5WPa2

Uma máquina virtual foi criada com a tecnologia *VirtualBox*[3], publicada para os participantes do experimento no endereço na Internet e instalada nos computadores do Virtus, a fim de facilitar o estudo das três tecnologias e a execução de alguns exercícios que envolviam codificação. A mesma máquina virtual foi utilizada para homogeneizar o ambiente de execução para as tarefas do experimento.

Especificação da Máquina virtual:

- Sistema operacional Linux Mint 18.1 Cinnamon 64 bits
- 4 GB de Memória RAM
- 20 GB Espaço em disco
- Ruby 2.3.1
- Node 6.11.0

[3] www.virtualbox.org

Tabela 7.4: Lista de aulas de Angular M: goo.gl/pz3swS

Aula	**Vídeo**	**Slides**	**Exercícios**
Hello World	youtu.be/XdJDLGINySg	goo.gl/9SJq8G	goo.gl/xmnqVV
Metadata	youtu.be/Bfybg3dtyrg	goo.gl/xTvwOq	goo.gl/vSTNhJ
Ports	youtu.be/KY_ixjKKjhE	goo.gl/xTvwOq	goo.gl/88YBBP
CRUD app	youtu.be/trhICy7_omo	goo.gl/xTvwOq	goo.gl/pCUngY

- Rails 4.2.6
- Angular 4.0.0
- Angular CLI 1.1.2
- Visual Studio Code 1.13.1

Após a conclusão de todos os exercícios, cada participante recebeu uma lista com as tarefas que deveria realizar no experimento propriamente dito. A lista continha 13 tarefas, numeradas de 0 a 12. Cada tarefa possui uma *Tag* (*Git*), representando o código inicial específico daquela tarefa, que também contém um teste automático *Selenium* para validar se a customização foi concluída com sucesso. O código das 12 tarefas foi disponibilizado no *Github* para *Ruby on Rails*[4] e *Angular M*[5], sendo que para uma tarefa qualquer N, o código HTML inicial após a execução do código *ERB* ou *Angular* era idêntico, e o teste Selenium também era idêntico para as duas tecnologias. Dessa forma, pôde-se garantir que o escopo da tarefa N foi igual independente da tecnologia escolhida para o participante. Todos os participantes deveriam copiar (*fork*) os repositórios das tarefas no *Github*, a fim de viabilizar a submissão de suas respostas no seus próprios repositórios *Github*.

Antes de iniciar as tarefas, o participante deveria executar o *script* `setup.sh`[6], responsável por configurar os dados do participante do *Git* e salvar o endereço dos repositórios *Github* do participante no arquivo `github.account.data`.

[4] github.com/rodrigovilar/rortasks

[5] github.com/rodrigovilar/mgtasks

[6] github.com/rodrigovilar/experiment_scripts/blob/master/setup.sh

A tarefa 0 representava uma customização simples, apenas adicionar uma classe de estilo *CSS* na tabela de listagem[7], e serviu de treinamento para o ambiente do experimento: *script* de montagem do ambiente da tarefa, *IDE*, *script* para teste da tarefa e envio dos resultados para o *Github*. Dessa forma, ela deveria ser executada para as duas tecnologias e seu resultado (tempo necessário para conclusão da alteração) não foi considerado para análise.

As tarefas 1 a 12 foram associadas individualmente a apenas uma das duas tecnologias, para cada participante, através de um sorteio aleatório realizado no site *Random.org*[8], que sorteou 6 números de 1 a 12 aleatoriamente. Para as tarefas cujo número foi sorteado, o participante a realizou com *Ruby on Rails*. Caso contrário, com *Angular M*.

Dessa forma, cada participante recebeu um email com as instruções de execução do experimento contendo a lista de 13 tarefas de customização a realizar, associadas às tecnologias. Também foram enviados os links para todo material utilizado nas vídeo aulas e para as instruções das customizações a fazer em cada tarefa.

7.3 Execução do experimento

Para iniciar a execução de cada uma das 12 tarefas do experimento, um participante deveria executar o *script* `start.sh`[9], que recebe como parâmetro o número da tarefa que deve ser realizada e a tecnologia a ser utilizada. O pseudocódigo desse script é:

- Verificar a quantidade de parâmetros e os valores dos parâmetros 1 (número da tarefa: de 0 a 12) e 2 (tecnologia: `ror` para *Ruby on Rails* ou `mg` para *Angular M*);
- Verificar a existência do arquivo `github.account.data`, que possui os dados de acesso aos repositórios no *Github* do participante;
- Salvar o número da tarefa no arquivo `task.data`;
- Salvar a tecnologia utilizada no arquivo `technology.data`;
- Clonar (baixar) o repositório da tecnologia utilizada a partir do *Github* do participante;

[7] Instruções para tarefa 0: goo.gl/RA9GXZ

[8] www.random.org/integer-sets

[9] github.com/rodrigovilar/experiment_scripts/blob/master/start.sh

- Mover o código para a *Tag* (*Git*) referente à tarefa;
- Atualizar as dependências do projeto;
- Criar uma *Branch* (*Git*) para acomodar o código de resposta da customização;
- Salvar o horário de início da tarefa no arquivo `begin.data`.

Na Tabela 7.5, são listadas as 12 tarefas de customização projetadas para o experimento. A coluna **Tipo** representa o tipo de metadado que precisava ser alterado na tarefa: E para metadados de Entidade, P de Propriedade e E&P de ambos. Na coluna **Escopo**, está descrito o escopo da alteração utilizado em cada tarefa: D (*default*) significa em todas as Entidades ou Propriedades, E quando as alterações foram limitadas por Entidade; P para as limitadas por Propriedade e TP para as limitadas por Tipo de propriedade. A coluna **Reúso** indica se houve a provisão de componentes *Angular M* para reúso na tarefa. Por fim, a coluna **Detalhes** apresenta os links com os detalhes das tarefas que foram enviados para os participantes.

Durante a customização da GUI, cada participante foi acompanhado por um monitor que respondeu dúvidas apenas sobre a sintaxe dos arquivos ERB e dos componentes. Dessa forma, tentou-se reduzir o ruído dos bugs de sintaxe na variável de saída do experimento (tempo gasto). O monitor não estava autorizado a auxiliar nas decisões sobre que trechos dos arquivos deveriam ser alterados, para não influenciar o aspecto cerne do experimento.

A atividade de povoar os metadados do domínio não foi avaliada no experimento. Portanto os comandos de *Scaffolding* de *Ruby on Rails*, que criavam os CRUDs para as 12 telas do experimento, foram disponibilizados no arquivo `entities.sh`, que pôde ser executado facilmente após a alteração dos *templates*. Além disso, o arquivo `entities.ts` possuía todos os metadados necessários para as mesmas 12 telas na versão *Angular M* do código. Assim sendo, o ruído da atividade de povoar os metadados também foi reduzido no experimento.

Ao término de cada tarefa, o participante deveria executar o *script* `stop.sh`[10], que possui o seguinte pseudocódigo:

- Ler o número da tarefa do arquivo `task.data`;

[10] github.com/rodrigovilar/experiment_scripts/blob/master/stop.sh

Tabela 7.5: Tarefas de customização de GUI para Aplicações corporativas

	Resumo da Tarefa	**Tipo**	**Escopo**	**Reúso**	**Detalhes**
1	Formatar os formulários de criação e edição como cartões	E	D	Sim	goo.gl/pP6U8y
2	Formatar o formulário, modificar a posição das etiquetas e formatar os campos no formulário de criação	E&P	D	Não	goo.gl/GniG9H
3	Aplicar Material design na tela de listagem, transformando as linhas da tabelas em cartões	E&P	D	Sim	goo.gl/Z7Pc38
4	Centralizar as colunas da tabela de listagem para quatro entidades	E&P	E	Não	goo.gl/5F6jbu
5	Aplicar um efeito de formatação nas linhas da tabela de listagem para quatro entidades	E	E	Sim	goo.gl/ancZWU
6	Transformar a tabela de listagem em cartões e representar os valores das propriedades como tags para três entidades	E&P	E	Não	goo.gl/bjPk9X
7	Aplicar tags coloridas nos valores da tela de show para três propriedades	P	P	Sim	goo.gl/o7bd9L
8	Centralizar as colunas da tabela de listagem para dois tipos de propriedade	P	TP	Não	goo.gl/PrVJYP
9	Criar um menu para as ações na tabela de listagem	E	D	Não	goo.gl/bpmn49
10	Definir valores mínimo e máximo permitidos para os campos inteiros de formulários	P	TP	Sim	goo.gl/rjsYT4
11	Utilizar campos textarea para as propriedades description e summary no formulário de criação	P	P	Sim	goo.gl/qkjbwu
12	Utilizar uma lista de seleção para representar o campo status no formulário de criação	P	P	Não	goo.gl/nnzZh6

- Ler a tecnologia do arquivo `technology.data`;
- Ler os dados do repositório *Github* do participante do arquivo `github.account.data`;
- Executar o arquivo de testes *Selenium* `test/task-e2e.js`;
  - Se o teste falhar, avisar o motivo da falha e parar a execução deste *script*;
- Salvar o horário de término da tarefa no arquivo `end.data`.
- Fechar uma versão (*git commit*) com as alterações realizadas e enviá-las (*git push*) para o repositório *Github* do participante;
- Apagar os arquivos utilizados na tarefa.

Os horários de início e término para cada participante e tarefa foram coletados manualmente e armazenados em uma planilha, juntamente com as demais informações sobre os participantes (Figura 7.1) e as tarefas (Tabela 7.5), para a realização da análise detalhada na próxima seção.

7.4 Análise dos resultados

Os resultados do experimento foram processados por um *script* escrito na linguagem R[11] (Apêndice C) e são apresentados nesta seção através de uma abordagem *top-down*. Inicialmente, são analisados os resultados gerais (Figura 7.2), agrupados em três formas: todas as tarefas, apenas tarefas com reúso de componente *Angular M* e apenas tarefas sem reúso de componente *Angular M*. Depois são analisados os resultados de cada tarefa individualmente (Figuras **??** e **??**), com a finalidade de observar as peculiaridades de cada parte do experimento.

Análise de todos os resultados juntos

Na Figura 7.2a, pode-se ver o *Boxplot* de todas as observações feitas no experimento. O eixo x representa a tecnologia utilizada: *mg* para *Angular M* e *ror* para *Ruby on Rails*. O eixo

[11] www.r-project.org

y representa o tempo gasto para realizar cada tarefa de customização de GUI. Ao executar o teste de normalidade de Shapiro-Wilk [85], viu-se que os valores observados para as duas tecnologias não era normal, portanto foi utilizado o teste não paramétrico Wilcoxon [46] para comparar os dois conjuntos.

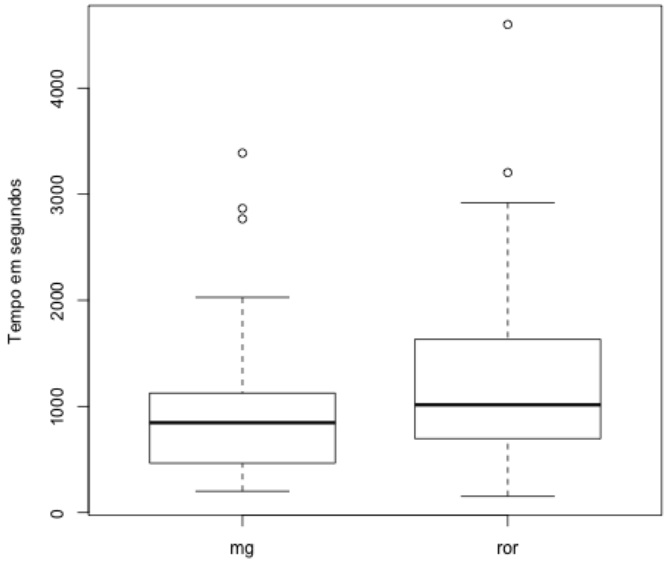
O valor mediano para *Angular M* foi $846,5$ segundos e para *Ruby on Rails* 1015 segundos. De acordo com o teste estatístico, para o nível de confiança de 95%, a probabilidade da hipótese nula — o tempo necessário para realização das tarefas em *Angular M* é maior do que em *Ruby on Rails* para todas as tarefas do experimento — ser observada é 0.014 (*p-value*). Portanto deve-se rejeitar moderadamente a hipótese nula, indicando que, na média de todo o experimento, *Angular M* influenciou positiva e significativamente a produtividade no desenvolvimento de GUI para Aplicações corporativas. De acordo com as médias observadas, nesse cenário, *Angular M* reduziu em 17% o tempo médio para realização das tarefas, o que representa um aumento de 20% na produtividade.

No intuito de confirmar ou refutar esse resultado em cenários mais detalhados, a análise estatística foi repetida separadamente para os resultados de dois grupos de tarefas, mantendo-se o nível de confiança em 95% e a hipótese nula como o tempo necessário para realização das tarefas em *Angular M* é maior do que em *Ruby on Rails*. Conforme a Tabela 7.5, as tarefas 1, 3, 5, 7, 10 e 11 fazem reúso de componentes *Angular M* e nas tarefas 2, 4, 6, 8, 9 e 12 os componentes Angular M precisaram ser escritos do zero.

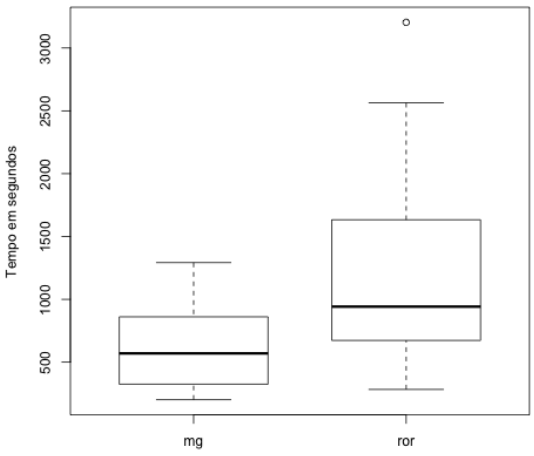
Análise dos resultados com reúso de componentes

Para o grupo de tarefas com reúso (Figura 7.2b), o tempo médio observado foi 570 e 942 segundos para *Angular M* e *Ruby on Rails*, respectivamente, o que significa um aumento de 64% na produtividade ao utilizar *Angular M* para realizar as tarefas. Devido à não normalidade dos resultados, foi utilizado um teste estatístico não paramétrico que calculou o *p-value* como 0.00007, o que sugere fortemente a rejeição da hipótese nula.

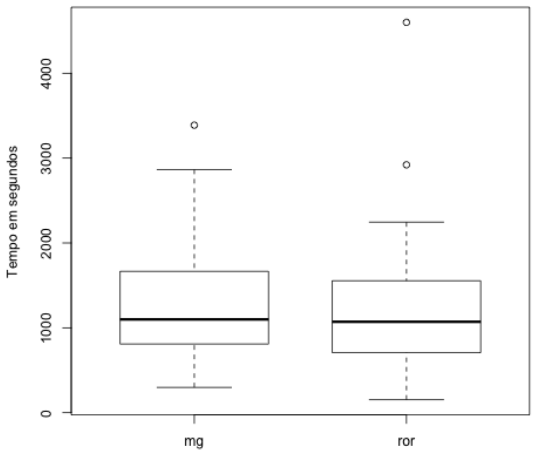
Esse resultado sugere que o aumento de produtividade proporcionado pelo *Angular M* é potencializado à medida que se disponibiliza uma biblioteca de componentes prontos para o reúso. Dessa forma, para se desenvolver futuramente a tecnologia aqui proposta, pode-se criar mecanismos para publicação, documentação e instalação de componentes *Angular M*,



(a) Todas as tarefas



(b) Tarefas com Reúso



(c) Tarefas sem Reúso

Figura 7.2: Comparações dos resultados agrupados

nos moldes do *Node Package Manager*[12], por exemplo.

Outra forma de enriquecer o reúso, é prover um mecanismo de pré-visualização de componentes com indicação visual de onde cada parâmetro do *configuration* atua. Durante a execução das tarefas, observou-se que os participantes demoravam para identificar esses parâmetros nos componentes.

Análise dos resultados sem reúso de componentes

Por outro lado, no grupo de tarefas sem reúso (Figura 7.2c), as médias foram próximas, 1099 e 1071 segundos para *Angular M* e *Ruby on Rails*, respectivamente. E o *p-value* foi alto (0.52, obtido por teste não paramétrico) indicando que a hipótese nula não pode ser rejeitada. Também foi realizado outro teste não paramétrico com a hipótese nula de que o tempo necessário para realização das tarefas em *Angular M* é **menor** do que em *Ruby on Rails*, para o qual se obteve *p-value* igual a 0.48. Assim sendo, não pode se afirmar que nenhuma das tecnologias é mais produtiva. Embora haja uma pequena vantagem de 3% para *Ruby on Rails* neste cenário, esse resultado pode ser fruto do acaso.

Esse resultado indica que *Angular M* manteve os mesmos níveis de produtividade de *Ruby on Rails*, mesmo em cenários onde os componentes precisaram ser construídos do zero. Deve-se considerar que a criação de componentes *Angular M* é, em teoria, mais complexa do que a alteração de templates *Rails*. Como pode ser visto nas análises a seguir, algumas alterações simples na GUI afetavam apenas um arquivo de *template Rails*, enquanto que quatro arquivos *Angular M* foram alterados (componente pai, componente filho, `AppModule` e `rules.ts`). Além disso, a inversão de controle, implementada por *Angular M*, dificulta a compreensão da interação entre os componentes. Essa dificuldade foi observada para todos os participantes e seria minimizada através de uma ferramenta visual para compor e pré-visualizar os componentes.

Assim sendo, seria esperado que, nos cenários sem reúso, *Ruby on Rails* fosse mais produtivo. Todavia, nesse ponto, foi observada nitidamente a problemática levantada no capítulo 1: o entrelaçamento de responsabilidades no mesmo *template* dificultou a sua compreensão, de maneira que alterações de, por exemplo, 6 linhas em um *template Rails* foram realizadas no mesmo tempo em que alterações de 50 linhas em 4 arquivos *Angular M*.

[12] www.npmjs.com/

Os *templates* já estavam copiados no projeto para as tarefas feitas em *Ruby on Rails*. Os participantes observaram que isso pode ter melhorado o tempo dessas tarefas em relação às de *Angular M* sem reúso de componentes. Por outro lado, a sintaxe complicada dos *templates* pode ter influenciado negativamente *Rails*, tanto quanto o entrelaçamento de responsabilidades. Em trabalhos futuros, pode-se projetar um experimento onde cada página do CRUD seria desenhada por um componente *Angular M* com entrelaçamento de responsabilidades. Dessa forma, poderia se reduzir os ruídos nos resultados e observar com mais clareza os resultados obtidos para o grupo de tarefas sem reúso de componentes.

A partir da análise agrupada das tarefas, de acordo com o reúso ou não de componentes, pode-se observar que a vantagem de *Angular M* no resultado geral foi fruto do grupo de tarefas onde havia reúso de componentes. Portanto, conclui-se que o impacto positivo de *Angular M* é indireto. O simples fato de utilizar *Angular M* não melhora a produtividade. É preciso fomentar a comunidade para povoar uma biblioteca de componentes prontos para reúso, o que tende a melhorar a produtividade em cerca de 64%.

Análise dos resultados individuais

O próximo passo da análise estatística foi a comparação dos resultados para cada tarefa individualmente. Os dados analisados estão resumidos na Figura 7.3, onde pode-se ver:

- O *Boxplot* dos tempos das tarefas para as duas tecnologias;
- O número da tarefa, à esquerda dos respectivos *Boxplots*;
- O impacto médio de *Angular M* na produtividade de cada tarefa, em relação a *Ruby on Rails*:
  - Esta informação (em percentual) está representada em retângulos no interior dos *Boxplots*;
  - Os retângulos com seta para cima representam tarefas em que *Angular M* foi mais produtivo. Caso contrário, a seta aponta para baixo;
  - Devido à pequena quantidade de amostras por tarefa, neste ponto da análise, o autor foi mais flexível em relação ao *p-value* dos resultados, considerando valores menos significantes (entre 0.05 e 0.1);

  - Os retângulos azuis representam análises estatísticas que não obtiveram impacto significante (*p-value* maior que 0.1);
  - Os retângulos laranja claro representam análises estatísticas com impacto significativamente fraco (*p-value* entre 0.05 e 0.1);
  - Os retângulos laranja escuro representam análises estatísticas com impacto significativamente forte (*p-value* menor que 0.05);

- A quantidade média de linhas alteradas por tarefa e tecnologia pode ser vista nos retângulos verdes abaixo de cada par de *Boxplots*, no formato: linhas alteradas em *Angular M* x linhas alteradas em *Ruby on Rails*;
- As tarefas onde houve reúso de componentes em *Angular M* estão indicadas com uma peça de quebra cabeça verde, no canto superior esquerdo do *Boxplot*.

Novamente, nesta fase da análise, manteve-se o nível de confiança em 95% e a hipótese nula como o tempo necessário para realização das tarefas em *Angular M* é maior do que em *Ruby on Rails*.

Análise individual das tarefas com reúso de componentes

Tarefa 1

Na tarefa 1, os tempos médios foram 240 e 640 segundos para *Angular M* e *Ruby on Rails*, respectivamente, o que indica um aumento de produtividade por volta de 166% para *Angular M*. O *p-value* obtido através de um teste não paramétrico foi 0.04, o que preconiza moderadamente que a hipótese nula deve ser rejeitada e que de fato há um ganho de produtividade com *Angular M* para esse cenário.

A tarefa 1 demanda a alteração da *tag* `form` nas telas de criação e edição, adicionando uma classe CSS. Essa alteração deveria ser aplicada nas telas de todos os *CRUDs*. Portanto a tarefa 1 representa uma alteração do tipo Entidade e com escopo *Default*. Além disso, para *Angular M* foi providenciado um componente que possibilitava esta alteração através do objeto *configuration*. Nesse cenário, *Angular M* auxilia o programador ao viabilizar a customização do formulário através do arquivo `rules.ts`.

Figura 7.3: Resultados individuais das tarefas do experimento

Ao analisar as respostas submetidas pelos participantes, observou-se que, para *Angular M*, duas linhas do arquivo `rules.ts` foram alteradas, uma para o formulário de criação e outra para o formulário de edição. Enquanto que em *Rails*, apenas uma linha do *template* de formulários precisou ser alterada. Um exercício semelhante a essa tarefa fora praticado na vídeo aula de *Code generation* de *Rails*, mesmo assim os participantes reportaram dificuldades para entender que ponto do formulário deveria ser alterado para incluir a classe CSS.

Diante dos resultados da tarefa 1, pode-se sugerir que, em *Angular M*, é mais fácil parametrizar as configurações de um componente de GUI baseado em metadados do que em *Ruby on Rails*. Desse modo, os componentes podem ser facilmente reutilizados e customizados, o que explicaria o aumento de 166% da produtividade nessa tarefa, mesmo tendo que alterar mais linhas de código.

Tarefa 3

Os resultados da **tarefa 3** foram: tempos médios de 1005 e 1618 segundos para *Angular M* e *Ruby on Rails*, respectivamente; aumento de produtividade de 61% para *Angular M*; e *p-value* igual a 0.076) obtido através de um teste paramétrico. Devido ao *p-value* um pouco alto, pode-se apenas sugerir fracamente a rejeição da hipótese nula.

Na tarefa 3, os participantes deveriam aplicar o *Material design*[13] na tela de listagem, o que envolveu responsabilidades de Entidade e Propriedade para o escopo *Default*. Alguns componentes *Angular M* foram providos para reúso nessa tarefa, de modo que apenas seis linhas do arquivo `rules.ts` precisavam ser alteradas. Por outro lado, para *Ruby on Rails*, 42 linhas precisaram ser alteradas no *template* de listagem.

Embora o tamanho do efeito tenha sido expressivo para a tarefa 3 (ganho de produtividade de 61% para *Angular M*), o *p-value* levemente alto enfraquece as conclusões para a tarefa 3 individualmente. Possivelmente, dois resultados muito baixos obtidos para *Rails* (para dois participantes muito experimentes) influenciaram o *p-value*. Além disso, os participantes demoraram para entender os sete parâmetros do *configuration* que precisaram ser definidos em *Angular M*. Pode-se, portanto, reforçar a importância de uma ferramenta visual para compreensão dos componentes *Angular M*.

[13] material.io/guidelines/material-design/introduction.html

Tarefa 5

Os resultados da **tarefa 5** foram: tempos médios de 523.5 e 995 segundos para *Angular M* e *Ruby on Rails*, respectivamente; aumento na produtividade de 90% para *Angular M*; e *p-value* igual a 0.09 obtido através de um teste paramétrico. Devido ao *p-value* um pouco alto, pode-se apenas sugerir fracamente a rejeição da hipótese nula.

Na tarefa 5, os participantes deveriam aplicar uma classe CSS nas linhas da tela de listagem, o que envolveu responsabilidades de Entidade para o escopo limitado a quatro entidades. Alguns componentes *Angular M* foram providos para reúso nessa tarefa, de modo que apenas cinco linhas do arquivo `rules.ts` precisavam ser alteradas. Por outro lado, para *Ruby on Rails*, foi necessário limitar o escopo da customização através de uma estrutura *if* dentro do *template*. Embora apenas quatro linhas novas precisassem ser criadas, o tempo da tarefa em *Ruby on Rails* foi quase o dobro de *Angular M*. Isso sugere que há maior complexidade e menor manutenibilidade nos *templates Rails* em relação aos componentes *Angular M*.

Tarefa 7

Os resultados da **tarefa 7** foram: tempos médios de 604.5 e 997.4 segundos para *Angular M* e *Ruby on Rails*, respectivamente; aumento na produtividade de 65% para *Angular M*; e *p-value* igual a 0.06 obtido através de um teste paramétrico. Devido ao *p-value* um pouco alto, pode-se apenas sugerir fracamente a rejeição da hipótese nula.

Na tarefa 7, os participantes deveriam aplicar uma classe CSS nas linhas da tela de *show*, o que envolveu responsabilidades de Propriedade para o escopo limitado a três propriedades. Um componente *Angular M* foi provido para reúso nessa tarefa, de modo que apenas quatro linhas do arquivo `rules.ts` precisavam ser alteradas. Por outro lado, para *Ruby on Rails*, foi necessário limitar o escopo da customização através de três estruturas *if* dentro do *template* e nove linhas precisaram ser criadas.

Essa tarefa representa um dos cenários onde a estrutura de portas de *Angular M* mais pode ser útil. O uso de três regras associadas a três objetos *configuration* diferentes substituiu elegantemente os três *ifs* no *template* de *Rails*. Provavelmente por esse motivo, os resultados sugerem um aumento na produtividade em 65% para *Angular M*.

Tarefa 10

Os resultados da **tarefa 10** foram: tempos médios de 582.3 e 1410.8 segundos para *Angular M* e *Ruby on Rails*, respectivamente; aumento na produtividade de 142% para *Angular M*; e *p-value* igual a 0.05 obtido através de um teste paramétrico, o que preconiza moderadamente que a hipótese nula deve ser rejeitada e que de fato há um ganho de produtividade com *Angular M* para esse cenário.

Nesta tarefa, onde houve alto ganho de produtividade para *Angular M*, foi disponibilizado um componente *Angular M* para reúso, no qual o objeto *configuration* possuía parâmetros bem parecidos com os atributos que precisavam ser customizados na *tag* `input`. Esse resultado reforça a suposição de que a boa compreensão sobre o objeto *configuration* é um dos pontos chave para o ganho de produtividade quando se utiliza componentes *Angular M*.

Tarefa 11

Os resultados da **tarefa 11** foram: tempos médios de 857 e 1130.9 segundos para *Angular M* e *Ruby on Rails*, respectivamente; aumento na produtividade de 32% para *Angular M*; e *p-value* igual a 0.09 obtido através de um teste paramétrico. Devido ao *p-value* um pouco alto, pode-se apenas sugerir fracamente a rejeição da hipótese nula.

Na tarefa 11, os participantes deveriam aplicar o elemento `textarea` em campos do formulário, o que envolveu responsabilidades de Propriedade para o escopo limitado a duas propriedades. Um componente *Angular M* foi provido para reúso nessa tarefa, de modo que apenas três linhas do arquivo `rules.ts` precisavam ser alteradas. Por outro lado, para *Ruby on Rails*, foi necessário limitar o escopo da customização através de estruturas *if* dentro do *template* e seis linhas precisaram ser criadas.

Essa tarefa representa novamente um cenário onde a estrutura de portas de *Angular M* permite o uso de duas regras associadas a dois objetos *configuration*, substituindo elegantemente os *ifs* no *template* de *Rails*.

Análise individual das tarefas sem reúso de componentes em Angular M e com poucas linhas de código alteradas em Ruby on Rails

Tarefa 2

Na tarefa 2, os tempos médios foram 1419.4 e 1564.2 segundos para *Angular M* e *Ruby on Rails*, respectivamente, o que indicaria um aumento de produtividade por volta de 11% para *Angular M*. Todavia, devido ao valor alto para o *p-value* (0.41) obtido através de um teste paramétrico, a hipótese nula original não pode ser rejeitada.

Também foi realizado outro teste paramétrico com a hipótese nula de que o tempo necessário para realização da tarefa 2 em *Angular M* é **menor** do que em *Ruby on Rails*, para o qual se obteve *p-value* igual a 0.59. Assim sendo, não se pode afirmar que nenhuma das tecnologias é mais produtiva para a tarefa 2. Embora haja uma ligeira vantagem de 11% para *Angular M* neste cenário, esse resultado pode ser fruto do acaso.

A tarefa 2 requer a alteração da *tag* `form` e também modifica a disposição dos elementos da linha do formulário, por isso o tipo de alteração dessa tarefa envolve metadados de Entidade e Propriedade. A customização dessa tarefa foi aplicada em todas as telas (escopo *Default*). Não foi disponibilizado componente *Angular M* para reúso na tarefa.

Nas respostas dos participantes, foram alteradas cerca de 7 linhas no *template* de formulário *Rails* ou 47 linhas em 4 arquivos de *Angular M*. Dado que não foram providos componentes reutilizáveis dois componentes precisaram ser feitos do zero. Em teoria, *Ruby on Rails* deveria ter sido mais produtivo nesse cenário. No entanto, como os resultados foram similares, pode-se supor que houve dificuldade na compreensão do *template* de formulário.

Tarefa 4

Os resultados da **tarefa 4** foram: tempos médios de 1595.1 e 1074.8 segundos para *Angular M* e *Ruby on Rails*, respectivamente; redução na produtividade de 33% para *Angular M*; e *p-value* igual a 0.88 obtido através de um teste paramétrico. Ao testar a hipótese nula como o tempo médio em *Angular M* é menor do que em *Ruby on Rails*, o *p-value* foi 0.12. Devido ao *p-value* alto nos dois testes, os resultados são inconclusivos para a tarefa 4.

Na tarefa 4, os participantes deveriam centralizar as colunas na tela de listagem, manipulando metadados de Propriedade e limitando a customização para o escopo de apenas quatro

entidades. Não foi disponibilizado componente *Angular M* para reúso na tarefa. Apenas duas linhas de código precisavam ser alteradas para *Ruby on Rails*, enquanto que para *Angular M* 46 linhas novas precisaram ser criadas em 4 arquivos.

A alteração era muito mais simples de fazer para *Ruby on Rails*, o que explica os resultados ligeiramente a favor dessa tecnologia. Todavia, para os dois testes paramétricos, o *p-value* foi alto e as conclusões não têm validade estatística.

Os participantes sugeriram a criação de ferramentas para automatizar a criação de componentes *Angular M*, da mesma forma como a ferramenta *Angular CLI*[14] faz com componentes *Angular 2+*. Outra sugestão foi a criação de ações de refatoramento de componentes *Angular M* que transformasse elementos do *template* em parâmetros de *configuration*. Houve confusão para alguns participantes sobre se seria possível selecionar várias entidades com apenas uma regra, o que não estava implementado no *Angular M* na época.

Tarefa 8

Os resultados da **tarefa 8** foram: tempos médios de 1489.4 e 1011.2 segundos para *Angular M* e *Ruby on Rails*, respectivamente; redução na produtividade de 32% para *Angular M*; e *p-value* igual a 0.84 obtido através de um teste paramétrico. Ao testar a hipótese nula como o tempo médio em *Angular M* é menor do que em *Ruby on Rails*, o *p-value* foi 0.16. Devido ao *p-value* alto nos dois testes, os resultados são inconclusivos para a tarefa 8.

Na tarefa 8, os participantes deveriam centralizar as colunas na tela de listagem, manipulando metadados de Propriedade e limitando a customização para o escopo de apenas dois tipos de propriedade. Não foi disponibilizado componente *Angular M* para reúso na tarefa. Apenas duas linhas de código precisavam ser alteradas para *Ruby on Rails*, enquanto que para *Angular M* oito linhas precisaram ser alteradas em três arquivos.

A alteração era mais simples de fazer para *Ruby on Rails*, o que explica os resultados ligeiramente a favor dessa tecnologia. Todavia, para os dois testes paramétricos, o *p-value* foi alto e as conclusões não têm validade estatística.

[14] cli.angular.io

Tarefa 12

Os resultados da **tarefa 12** foram: tempos médios de 1378.4 e 818.7 segundos para *Angular M* e *Ruby on Rails*, respectivamente; redução na produtividade de 41% para *Angular M*; e *p-value* igual a 0.97 obtido através de um teste paramétrico. Ao testar a hipótese nula como o tempo médio em *Angular M* é menor do que em *Ruby on Rails*, o *p-value* foi 0.03. Dessa forma, pode-se afirmar moderadamente que *Ruby on Rails* foi mais produtivo que *Angular M* para a tarefa 12.

Na tarefa 12, os participantes deveriam transformar o campo `status` em uma caixa de seleção, manipulando metadados de Propriedade e limitando a customização para o escopo de apenas uma propriedade. Não foi disponibilizado componente *Angular M* para reúso na tarefa. 11 linhas de código precisavam ser alteradas para *Ruby on Rails*, enquanto que para *Angular M* 34 linhas precisaram ser alteradas em três arquivos.

Esta tarefa pode ser considerada de complexidade média, pois envolve apenas uma estrutura *if* dentro de uma propriedade. Das 11 linhas alteradas em *Rails*, sete eram praticamente estáticas por representar a montagem da caixa de seleção. Além disso, todos os participantes que utilizaram *Angular M* tiveram dificuldades para utilizar a diretiva `[attr.list]`, pois ela utilizou uma forma para concatenar *strings* que não tinha sido explicada no treinamento.

Esse resultado indica moderadamente que, em customizações mais simples e sem reúso de componentes *Angular M*, *Ruby on Rails* é mais produtivo.

Análise individual das tarefas sem reúso de componentes em Angular M e com muitas linhas de código alteradas em Ruby on Rails

Tarefa 6

Os resultados da **tarefa 6** foram: tempos médios de 1283.5 e 1683 segundos para *Angular M* e *Ruby on Rails*, respectivamente; aumento na produtividade de 31% para *Angular M*; e *p-value* igual a 0.10 obtido através de um teste não paramétrico. Devido ao *p-value* um pouco alto, pode-se apenas sugerir fracamente a rejeição da hipótese nula.

A tarefa 6 representa uma customização complexa, parecida com a da tarefa 3. No entando, desta vez não havia componentes *Angular M* prontos para reúso. Em *Ruby on Rails*, o *template* de listagem deveria ser alterado em 15 linhas, entrelaçando uma estrutura *if*, para

limitação do escopo a três entidades, com dois *loops* para desenhar as entidades e suas propriedades. Além disso, para as outras nove entidades fora do escopo, deveria ser mantida a listagem original em tabela. Em *Angular M*, foi necessário criar 45 linhas em 4 arquivos.

Embora o *p-value* levemente alto enfraqueça as conclusões da tarefa 6, pode-se supor que a organização dos componentes *Angular M* é mais fácil de manter do que *Ruby on Rails* em cenários complexos. Pois, mesmo precisando alterar o triplo de linhas de código, os participantes que utilizaram *Angular M* foram 31% mais produtivos.

Tarefa 9

Os resultados da **tarefa 9** foram: tempos médios de 437.4 e 775 segundos para *Angular M* e *Ruby on Rails*, respectivamente; aumento na produtividade de 77% para *Angular M*; e *p-value* igual a 0.03 obtido através de um teste paramétrico, o que preconiza moderadamente que a hipótese nula deve ser rejeitada e que de fato há um ganho de produtividade com *Angular M* para esse cenário.

Na tarefa 9, os participantes deveriam criar um menu para todas as linhas da tela de listagem, o que envolveu responsabilidades de Entidade com escopo *Default*. Não foi disponibilizado componente *Angular M* para reúso na tarefa. 15 linhas de código precisavam ser alteradas no *template* de listagem para *Ruby on Rails*, enquanto que para *Angular M* 30 linhas deviam ser alteradas em três arquivos.

Semelhante à tarefa 6, que demandou a alteração de 15 linhas no *template Rails*, na tarefa 9, *Angular M* foi bem mais produtivo (31% na tarefa 6 e 77% na tarefa 9) do que *Ruby on Rails*, mesmo demandando a alteração do dobro de linhas de código. A similaridade desses resultados pode indicar que, à medida que a quantidade de linhas alteradas nos *templates* cresce, ou seja, quando se tem cenários de customização mais complexos, a produtividade de *Angular M* diverge positivamente em relação a *Ruby on Rails*, até quando não há reúso de componentes *Angular M*.

Em resumo, pode-se considerar que, após a análise individual das tarefas:

- Ao se considerar apenas o efeito de melhora ou piora da produtividade para *Angular M*, em nove tarefas *Angular M* foi mais produtivo (1, 2, 3, 5, 6, 7, 9, 10 e 11). Ao passo que *Ruby on Rails* foi mais produtivo em três cenários (4, 8 e 12);

- *Angular M* foi mais produtivo sempre que houve reúso de componentes, mesmo quando mais linhas de código precisaram ser alteradas nessa tecnologia;
- Nas tarefas sem reúso de componentes, o resultado dependeu da complexidade das alterações em *Ruby on Rails*:
  - Quando houve poucas linhas de código alteradas em *Ruby on Rails*, três resultados foram inconclusivos (2, 4 e 8) e um apontou melhor produtividade para *Ruby on Rails* (12);
  - Quando houve mais que 11 linhas de código alteradas em *Ruby on Rails*, *Angular M* foi significativamente mais produtivo, o que indica que a complexidade dos *templates* tende a dificultar a manutenção do seu código.
- Ao se considerar os cenários que possuam, pelo menos, validade estatística fraca ($p - value <= 0.10$), *Angular M* foi mais produtivo em oito tarefas (1, 3, 5, 6, 7, 9, 10 e 11) e *Ruby on Rails*, em uma (12):
  - Em todos os seis cenários com reúso de componentes, *Angular M* foi mais produtivo;
  - Nos cenários sem reúso, houve uma ligeira vantagem para *Angular M* ($2x1$).
- Ao se considerar os cenários que possuam validade estatística forte ($p - value <= 0.05$), *Angular M* foi mais produtivo em três tarefas (1, 9 e 10) e *Ruby on Rails*, em uma (12):
  - Em dois cenários com reúso de componentes, *Angular M* foi mais produtivo.
  - Nos cenários sem reúso, houve um empate ($1x1$).

Esses resultados, embora que nem todos possuam validade estatística forte, corroboram com a Figura 7.2 ao indicar que o reúso de componentes foi o fator que causou a melhora de produtividade para *Angular M* neste experimento.

Comentários e sugestões dos participantes

Além dos comentários dos participantes citados na análise individual das tarefas, as seguintes observações foram feitas após a conclusão do experimento:

- Os seis participantes que não tinham experiência com HTML apresentaram dificuldades para relacionar as *tags* nas tarefas de customização;
- A sintaxe do método `for_form` e `presence?` em *Ruby on Rails* e da diretiva `[ngClass]` de *Angular* é confusa;
- Na tarefa 12, o `datalist` é estático mas alguns participantes tentaram gerá-lo dinamicamente;
- Um participante não reutilizou o componente *Angular M* em uma tarefa pois não notou que ele existia;
- Os testes *Selenium* comparavam os textos considerando os espaços, por isso quebrava em cenários onde poderia ser mais tolerante, acrescentando tempo desnecessário a algumas tarefas;
- Houve confusão entre as regras de *property* e *property type* de *Angular M*;
- Houve confusão entre as regras de *entity* e *entity type* de *Angular M*;
- A documentação dos parâmetros das regras de *entity type* e *property type* foi deficiente em relação aos seus parâmetros;
- Houve confusão entre as *tags erb* (`<% %>`) e as *tags* do *template* (`<%% %>`);
- Para *Angular M*, é difícil encontrar o componente responsável por desenhar uma parte específica da GUI.

As sugestões a seguir foram providas pelos participantes a fim de melhorar o arcabouço *Angular M*:

- Nas regras, o seletor de entidades e propriedades poderia ser *case insensitive*;

- Nas regras, aceitar uma lista nos seletores para reduzir a quantidade de regras necessárias;
- Implementar um assistente para criar um componente a partir de outro já existente;
- Utilizar algoritmos para montar blocos de regras facilmente, por exemplo, passar um parâmetro para o método `describeRules` que modificaria as regras retornadas. No primeiro caso, configuraria todos os componentes para utilizar *Bootstrap* e, no segundo caso, *Material*. Para o usuário final, bastaria a troca do valor deste parâmetro para modificar diversas regras e seus componentes.

Lições aprendidas no experimento

Por fim, três lições aprendidas foram documentadas com a finalidade de melhorar futuros experimentos:

- Na finalização da tarefa, dividir o *commit* em dois: um para as alterações nos *templates* e outro para as demais. Com isso seria mais fácil identificar o esforço do participante;
- Pode-se sabotar os testes automatizados das tarefas fazendo-os sempre passar. Esse truque pode ser anulado se, antes de executar os testes, somente o código de teste for retornado para o estado inicial da tarefa;
- Houve perda de tempo por erro na senha do *Github*. A solução pode ser marcar o tempo de término da tarefa logo após o teste passar. Se houver erro de senha, o *script* de finalização pode ser invocado novamente e deve avançar imediatamente para o *push* no *Github*.

7.5 Análise de ameaças à validade

A primeira, e principal, ameaça à validade deste experimento é a de generalização dos resultados, da amostra escolhida — profissionais desenvolvedores de software com experiência baixa ou intermediária — para toda a população de desenvolvedores.

De fato, a população global de desenvolvedores pode divergir da amostra experimentada, porém deve-se considerar o alto custo para envolver profissionais neste experimento. Em um

cálculo superficial, supondo que o custo da hora de um programador médio custa 50 reais e que os 15 programadores precisaram de 40 horas em média para treinamento e execução do experimento, pode-se concluir que, se o experimento não tivesse sido voluntário, seu custo total seria em torno de trinta mil reais. Assim sendo, entende-se porque a maior parte das pesquisas de Engenharia de Software Experimental utiliza alunos de graduação ou pós-graduação em seus experimentos, apresentando problemas de generalidade maiores do que o presente neste trabalho.

Também existe uma ameaça pelo fato dos participantes trabalharem na mesma empresa, o que pode configurar um viés devido à homogeneidade de práticas e processos de desenvolvimento. No entanto, pôde-se verificar algumas características heterogêneas nos quinze participantes: trabalham em onze equipes diferentes que desenvolvem projetos para nove clientes do Virtus; possuem um portfólio variado de tipos de projeto, a saber *desktop*, *frontend web*, *backend web*, móvel Android e iOS, sistemas embarcados e bancos de dados; atuam em cargos variados, como pode ser visto na Figura 7.1a; foram formados por seis cursos universitários diferentes; e possuem experiência com as linguagens Java, Python, JavaScript, C++, C#, PHP, Pascal, Swift, Ruby e R. Dessa forma, supõe-se que a amostra do experimento se aproxima das características da população de desenvolvedores de software.

Pequenos incidentes aleatórios podem ter interferido nos resultados do experimento, como por exemplo: queda de conexão com a Internet, queda de energia, problemas de hardware e software nas máquinas, entre outros.

A fim de dirimir esta ameaça, duas ações foram realizadas. Primeiro o experimento foi dividido em 12 tarefas, de modo que a incidência das pequenas interferências pôde ser distribuída mais homogeneamente nas unidades experimentais. De fato, a própria quebra do experimento (de uma tarefa grande para 12 tarefas pequenas) reduz a probabilidade de ocorrência de um incidente durante uma tarefa. Em segundo lugar, as mesmas condições de trabalho foram providas para todos os participantes, em termos de espaço físico, hardware e software.

Outra ameaça é referente a sujeitos experimentais humanos que podem facilmente alterar seu comportamento ao passar do tempo gerando ruído aos dados. Essa ameaça também foi combatida através da quebra do experimento em 12 tarefas, a fim de reduzir o impacto nas mudanças de comportamento a pequenas tarefas. Ademais, todos os participantes fizeram o

experimento presencialmente e acompanhados de um monitor.

Um dos aspectos que mais pode ter influenciado o experimento, e que pode acometer qualquer pesquisa que meça produtividade, foi o tempo gasto pelos participantes para resolver problemas que não pertenciam ao cerne da pesquisa, por exemplo, erros de sintaxe de *Ruby* ou *Angular*. Em ambiente industrial, esse problema tende a desaparecer à medida que os desenvolvedores se tornam mais experientes com as linguagens e plataformas.

Um monitor foi disponibilizado para auxiliar os participantes na resolução desse tipo de problema periférico. Todavia, pôde-se ver em alguns resultados *outliers* que nem todos os participantes requisitaram ajuda do monitor.

Por fim, a comunidade de Engenharia de Software Experimental relata recorrentemente problemas na coleta do tempo das tarefas de experimento. O principal motivo é o esquecimento dos participantes anotarem o horário e término das tarefas, mesmo quando são supervisionados por monitores presenciais. Esse problema foi resolvido através da criação de dois *Shell scripts* para montar o ambiente de cada tarefa e para submeter os resultados. Respectivamente, esses *scripts* marcavam e salvavam em arquivo os horários de início e término de cada tarefa. Por essa razão, todos os horários relativos ao experimento foram salvos e coletados com sucesso.

7.6 Avaliação da hipótese do trabalho

No início deste trabalho, levantou-se a hipótese de que é possível aumentar o reúso sem prejudicar a manutenibilidade no desenvolvimento de GUI para aplicações corporativas *web*, através do encapsulamento das responsabilidades de GUI em artefatos de granularidade fina com base nos metadados do modelo do domínio.

No decorrer da avaliação deste trabalho, foram utilizadas duas premissas. Primeiramente o uso de metadados na GUI de aplicações corporativas web torna o seu código reutilizável ao desacoplá-la do Modelo do domínio, o que foi demonstrado na seção 2.3.

Em segundo lugar, a abordagem proposta nesse trabalho — encapsulamento das responsabilidades de GUI em artefatos de granularidade fina com base nos metadados do modelo do domínio — foi documentada através de uma linguagem de padrões (capítulo 5) e foi implementada através do arcabouço *Angular M* (capítulo 6).

Assim sendo, é preciso observar o aspecto da manutenibilidade ao se utilizar ou não a abordagem proposta. Especificamente para este trabalho, escolheu-se, como métrica de manutenibilidade, o tempo gasto para realização de tarefas de customização de GUI. Quanto maior o tempo demandado para realizar a tarefa, mais fácil seria manter o software e mais produtivo seria o time de desenvolvedores.

No entanto, foi preciso escolher um grupo de controle para comparar os resultados obtidos para *Angular M*. Se tivesse sido escolhida uma tecnologia sem manipulação de metadados, a comparação poderia ser deturpada pelo ganho de produtividade que o uso de metadados por si só confere. Portanto, a tecnologia escolhida para as tarefas do grupo de controle foram os *templates* de *Ruby on Rails*, que manipulam metadados mas não possuem responsabilidades encapsuladas.

Consequentemente, a hipótese nula testada concretamente neste trabalho foi:

A realização de tarefas de customização de GUI, em aplicações corporativas web, é mais produtiva com templates de Ruby on Rails do que com Angular M.

Essa hipótese pôde ser rejeitada a partir dos resultados do experimento relatado neste trabalho. Em termos gerais, *Angular M* foi 20% mais produtivo do que *Ruby on Rails* e a hipótese nula foi rejeitada com nível de confiança de 95% e *p-value* igual a 0.014.

Os resultados também foram analisados em contextos menores, onde se observou que *Angular M* é mais produtivo nos cenários com componentes prontos para serem reutilizados. Esse reúso de metacomponentes não pode ser realizado em *Ruby on Rails* e só foi possível de ser feito em *Angular M* porque esse arcabouço segue a abordagem proposta neste trabalho.

Consequentemente, pode-se afirmar que o encapsulamento das responsabilidades de GUI em artefatos de granularidade fina com base nos metadados do modelo do domínio permite o reúso de código e melhora a manutenibilidade, pelo menos em termos do tempo gasto para realização de tarefas de customização.

Capítulo 8

Conclusões e Trabalhos futuros

Uma abordagem para decomposição e reúso de componentes baseados em metadados foi proposta neste trabalho, com o objetivo de aumentar a produtividade no desenvolvimento de interfaces gráficas para aplicações corporativas *web*.

Tal abordagem foi desenvolvida e catalogada [98] através de uma linguagem de padrões de projeto e arquiteturais, a partir da análise do estado da arte e das tecnologias presentes na indústria para o desenvolvimento de interfaces gráficas para aplicações corporativas. As principais características da linguagem de padrões são:

- Formalização das boas práticas e sugestão de melhorias para as deficiências recorrentes nas tecnologias de mercado e no estado da arte;
- Desacoplamento entre o código da interface gráfica e do modelo de domínio das Aplicações corporativas por meio do uso de metadados em vez de referências diretas;
- Decomposição dos componentes de GUI, encapsulando suas responsabilidades de acordo com os metadados do modelo de domínio, o que potencializa o seu reúso e facilita a manutenção de GUI;
- Inversão de controle nos componentes de GUI baseados em metadados, facilitando sua substituição, parametrização e customização em telas específicas de aplicações corporativas;
- Uso de regras declarativas para compor, customizar e parametrizar facilmente a GUI de aplicações corporativas;

- Criação de um mecanismo para publicação e reúso de componentes e regras de Interface gráfica para Aplicações corporativas baseados em metadados, que podem ser reaproveitados até em aplicações distintas.

Dois arcabouços foram implementados, com linguagens de programação e arquiteturas diferentes, no intuito de verificar a viabilidade da implementação total ou parcial da linguagem de padrões.

Para fins de avaliação da abordagem proposta, um dos arcabouços criados (*Angular M*) foi comparado com uma tecnologia amplamente utilizada no mercado para gerar GUI a partir dos metadados do modelo de domínio, o arcabouço de *Scaffolding Ruby on Rails*. Para tanto, foi realizado um experimento com 15 profissionais, que executaram 12 tarefas de customização de GUI cada um, totalizando 180 unidades experimentais que foram analisadas estatisticamente. Dessas tarefas, metade provia componentes prontos para reúso (opção viável apenas para *Angular M*) e, na outra metade, o código da customização precisava ser criado do zero.

Em termos gerais, nas tarefas implementadas com *Angular M* os participantes foram 20% mais produtivos do que nas tarefas feitas com *Ruby on Rails*. Esse resultado tem validade estatística pois apresenta *p-value* igual a 0.014, com nível de confiança de 95%, para a hipótese nula de que *Ruby on Rails* é mais produtivo que *Angular M*.

Ao analisar separadamente os grupos de tarefas com reúso de componentes, o ganho de produtividade cresce para 64% e o *p-value* se torna ainda mais significativo (0.00007). Enquanto que, nas tarefas sem reúso, não existe diferença significativa de produtividade. Assim sendo, pode-se sugerir que, no contexto desse experimento, *Ruby on Rails* e *Angular M* possuem produtividade semelhante no início dos projetos de desenvolvimento de GUI para aplicações corporativas. E, à medida que uma biblioteca de componentes de GUI vai sendo montada, *Angular M* possivelmente apresentará um ganho de produtividade alto e estatisticamente significante. Se uma empresa adotar uma política de reúso dos componentes de GUI em projetos diferentes, *Angular M* poderá apresentar ganho de produtividade desde o início dos novos projetos.

Uma análise estatística ainda mais detalhada foi realizada para cada grupo de 12 tarefas. Considerando uma validade estatística mais fraca (*p-value* < 0.10), *Angular M* foi mais produtivo em oito tarefas, enquanto que *Ruby on Rails* o foi em uma. Ao restringir os por uma

validade mais forte (*p-value* < 0.05), o resultado muda para $3x1$ a favor de *Angular M*. Portanto, o resultado mais detalhado apresenta uma leve consistência em relação aos resultados gerais, dado que o ganho de produtividade de *Angular M* não acontece em picos isolados de tarefas. Pelo contrário, esse efeito pode ser observado em dois terços dos cenários do experimento.

Os resultados obtidos neste trabalho confirmam a hipótese de que: o encapsulamento das responsabilidades de GUI em artefatos de granularidade fina, com base nos metadados do modelo do domínio, permite o reúso de código e melhora a manutenibilidade, pelo menos em termos do tempo gasto para realização de tarefas de customização.

Trabalhos futuros

Por limitações de escopo, a linguagem de padrões proposta não possui todas as características das ferramentas existentes na indústria nem no estado da arte. Portanto, pode-se continuar adicionando padrões na linguagem para lidar com aspectos tais quais: renderização da GUI no servidor (em contraponto ao padrão CLIENT RENDERING); geração de código final para GUI *versus* interpretação de metadados em tempo de execução; especialização do padrão PROPERTY RENDERER para tratar propriedades que representam *enumerations*; autenticação e autorização baseados em metadados; uso de *tags* no Modelo do domínio e nas regras para delimitar o escopo nas customizações de GUI.

Neste trabalho definiu-se uma referência na comparação de produtividade entre *Angular M* e *Ruby on Rails*. Portanto, em experimentos futuros, a evolução de *Angular M* pode ser comparada com a versão utilizada aqui ($0.4.21$). Além disso, o mesmo experimento executado nesta ser repetido apenas com *Angular M*, ao substituir *Ruby on Rails* por uma implementação com componentes *Angular M* grandes, que entrelaçam responsabilidades de modo semelhante a *Ruby on Rails*. Esta repetição do experimento pode, inclusive, remover o efeito da sintaxe complexa dos *templates* e avaliar com mais nitidez o efeito da abordagem proposta neste trabalho: decompor e reutilizar os componentes de GUI baseados em metadados.

Os componentes de GUI baseados em metadados de relacionamentos do Modelo do domínio não foram avaliados neste trabalho devido à sua ausência nos *templates* de *Ruby on*

Rails. Quando for encontrada uma tecnologia relevante no mercado que supra essa lacuna, *Angular M* pode ser comparado a ela em um novo experimento. Para isso, seria necessário implementar o suporte a relacionamentos no *Angular M*, baseado no código já existente no *Geneguis*.

Os participantes do experimento fizeram as seguintes sugestões de melhoria para o *Angular M*, que podem ser implementadas em trabalhos futuros: criar uma ferramenta de suporte para compor visualmente os componentes e as regras de *Angular M*, facilitando a compreensão da interação entre os componentes; melhorar a API do *Angular M* com listas em seletores, documentação mais clara e seletores *case insensitive*; implementar assistentes para refatoramento e criação de componentes específicos para *Angular M*; compor regras e componentes utilizando lógica de decisão complexa a fim de facilitar customização de grandes trechos de código de GUI para o usuário final, viabilizando, por exemplo, a troca de 20 componentes *Bootstrap* por outros 20 componentes *Material*, bastando mudar o valor de um parâmetro no arquivo `rules.ts`. Pode-se especular que essa última melhoria pode gerar um ganho de produtividade muito maior do que o observado nesta tese.

Uma perspectiva totalmente diferente para este trabalho seria o seu uso em artefatos de documentação no processo de desenvolvimento de aplicações corporativas. Por exemplo, muito texto repetitivo pode ser encontrado em requisitos e casos de teste e uma questão importante seria: quanto conteúdo pode ser abstraído nesses documentos ao se utilizar componentes geradores de texto baseados em metadados e com responsabilidades bem definidas? *Angular M* poderia ser facilmente adaptado para essa pesquisa ao gerar texto em vez de HTML.

Por fim, após evoluir e robustecer a biblioteca *Angular M*, pode-se avaliá-la em estudos de caso industriais. No Virtus, por exemplo, existe um histórico de aplicações corporativas *web* que podem ser comparadas com novas aplicações implementadas em *Angular M*, em relação ao esforço necessário para implementar módulos ou sistemas completos. Desse modo, pode-se disponibilizar, para a indústria de software, resultados ainda mais claros sobre o impacto da abordagem proposta aqui na produtividade do desenvolvimento de interfaces gráficas para aplicações corporativas *web*.

Bibliografia

[1] Alain Abran e Pierre Bourque. *SWEBOK: Guide to the software engineering Body of Knowledge*. IEEE Computer Society, 2004 (ver p. 19).

[2] K. K. Aggarwal e Yogesh Singh. *Software Engineering*. New Age International Pvt Ltd Publishers, 2008. ISBN: 978-8122423600 (ver p. 19).

[3] Christopher Alexander. "The origins of pattern theory: The future of the theory, and the generation of a living world". Em: *Software, IEEE* 16.5 (1999), pp. 71–82 (ver p. 79).

[4] Christopher Alexander, Sara Ishikawa e Murray Silverstein. *A pattern language: towns, buildings, construction*. Vol. 2. Oxford University Press, 1977 (ver p. 10).

[5] Alexandre Cláudio de Almeida, Glauber Boff e Juliano Lopes De Oliveira. "A framework for modeling, building and maintaining enterprise information systems software". Em: *Software Engineering, 2009. SBES'09. XXIII Brazilian Symposium on*. IEEE. 2009, pp. 115–125 (ver p. 32).

[6] Andre M Andrade, Rodrigo A Vilar, Anderson A Lima, Hyggo Almeida e Angelo Perkusich. "Analyzing duplication on code generated by Scaffolding frameworks for Graphical user interfaces". Em: *Proceedings of 29th Int. Conf. Software Engineering and Knowledge Engineering*. 2017 (ver p. 34).

[7] Nathalie Aquino, Jean Vanderdonckt e Oscar Pastor. "Transformation templates: adding flexibility to model-driven engineering of user interfaces". Em: *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM. 2010, pp. 1195–1202 (ver p. 33).

[8] Colin Atkinson e Thomas Kühne. "Model-driven development: a metamodeling foundation". Em: *Software, IEEE* 20.5 (2003), pp. 36–41 (ver pp. 3, 23).

[9] Luiz Azevedo, Clovis Torres Fernandes e Eduardo Martins Guerra. "Architectural Model for Generating User Interfaces Based on Class Metadata". Em: *Computational Science and Its Applications–ICCSA 2013*. Springer, 2013, pp. 230–245 (ver p. 37).

[10] Michael Bächle e Paul Kirchberg. "Ruby on Rails." Em: *IEEE Software* 24.6 (2007), pp. 105–108 (ver p. 23).

[11] András Balogh e Dániel Varró. "Pattern composition in graph transformation rules". Em: *1st European Workshop on Composition of Model Transformations (CMT'06)[10]*. Citeseer. 2006, pp. 33–37 (ver p. 33).

[12] Vangie Beal. *enterprise application*. URL: `http://www.webopedia.com/TERM/E/enterprise_application.html` (acedido em 05/12/2015) (ver p. 1).

[13] Kent Beck e Ward Cunningham. "Using pattern languages for object-oriented programs". Em: *OOPSLA-87 Workshop on the Specification and Design for Object-Oriented Programming*. 1987 (ver pp. 10, 81).

[14] Steve Berczuk, Brad Appleton e Ralph Cabrera. "Getting ready to work: Patterns for a developer's workspace". Em: *Proceedings of the 7th Conference on Pattern Languages of Programs*. ACM. 2000 (ver p. 81).

[15] Grady Booch. *Object oriented analysis & design with application*. Pearson Education India, 2006 (ver p. 53).

[16] Kyle Brown, Gary Craig, Greg Hester, Jim Amsden, David Pitt, Peter M Jakab, Russell Stinehour e Mark Weitzel. *Enterprise java programming with IBM webSphere*. Addison-Wesley Professional, 2003 (ver pp. 3, 20, 22).

[17] Juan Carlos Castrejón, Rosa López-Landa e Rafael Lozano. "Model2Roo: A model driven approach for web application development based on the Eclipse Modeling Framework and Spring Roo". Em: *Electrical Communications and Computers (CONIELECOMP), 2011 21st International Conference on*. IEEE. 2011, pp. 82–87 (ver pp. 23, 34).

[18] Nicholas Chen. "Convention over configuration". Em: *http://softwareengineering,vazexqi. com/files/pattern.html* (2006) (ver p. 33).

[19] Erik Christensen, Francisco Curbera, Greg Meredith e Sanjiva Weerawarana. *Web Services Description Language (WSDL) 1.1*. URL: https://www.w3.org/TR/wsdl (acedido em 12/03/2016) (ver p. 50).

[20] Jens Coldewey et al. "User interface software". Em: *Proceedings of the 5th Conference on Pattern Languages of Programs*. 1998 (ver p. 80).

[21] Louis Columbus. *IDC Predicts SaaS Enterprise Applications Will Be A 50.8B Market By 2018*. URL: http://www.forbes.com/sites/louiscolumbus/2014/12/20/idc-predicts-saas-enterprise-applications-will-be-a-50-8b-market-by-2018/ (acedido em 07/12/2015) (ver p. 1).

[22] Danny Coward. "JSR 175: A Metadata Facility for the JavaTM Programming Language". Em: *Java Community Process. https://www. jcp. org/en/jsr/detail* (2004) (ver pp. 3, 23, 50).

[23] Paulo Pinheiro Da Silva. "User interface declarative models and development environments: A survey". Em: *Interactive Systems Design, Specification, and Verification*. Springer, 2000, pp. 207–226 (ver p. 33).

[24] Frankel S David. *Model driven architecture: applying MDA to enterprise computing*. 2003 (ver p. 23).

[25] Andy Dearden e Janet Finlay. "Pattern languages in HCI: A critical review". Em: *Human–computer interaction* 21.1 (2006), pp. 49–102 (ver pp. 79, 80).

[26] Roy Thomas Fielding. "Architectural styles and the design of network-based software architectures". Tese de doutoramento. University of California, Irvine, 2000 (ver pp. 47, 94).

[27] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach e Tim Berners-Lee. *Hypertext transfer protocol–HTTP/1.1*. Rel. téc. 1999 (ver p. 61).

[28] Django Software Foundation. *Built-in class-based generic views*. URL: https://docs.djangoproject.com/en/1.9/topics/class-based-views/generic-display/ (acedido em 06/03/2016) (ver p. 34).

[29] Martin Fowler. *GUI Architectures*. URL: http://www.martinfowler.com/eaaDev/uiArchs.html (acedido em 10/03/2016) (ver p. 41).

[30] Martin Fowler. *Inversion of control containers and the dependency injection pattern*. URL: http://martinfowler.com/bliki/InversionOfControl.html (acedido em 26/12/2015) (ver p. 27).

[31] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002 (ver pp. 1, 2, 20, 46, 47, 48, 61, 63, 66).

[32] Martin Fowler. "Reducing coupling". Em: *IEEE Software* 4 (2001), pp. 102–104 (ver p. 18).

[33] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999 (ver pp. 20, 74).

[34] Martin Fowler. "Using metadata". Em: *Software, IEEE* 19.6 (2002), pp. 13–17 (ver pp. 3, 22, 23).

[35] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994 (ver pp. 39, 50, 51, 57, 61, 80).

[36] Jesse J Garrett. *Ajax: A New Approach to Web Applications*. 2005. URL: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications (acedido em 14/03/2016) (ver pp. 63, 101).

[37] Dale Green. "The reflection api". Em: *Java Tutorial Continued, The: The Rest of the JDK* (2004), pp. 699–732 (ver p. 59).

[38] Eduardo M Guerra, Jerffeson T de Souza e Clovis T Fernandes. "A pattern language for metadata-based frameworks". Em: *Proceedings of the 16th Conference on Pattern Languages of Programs*. ACM. 2009, p. 3 (ver pp. 3, 23, 37, 50).

[39] Eduardo Martins Guerra e Clovis Torres Fernandes. "A Metadata-Based Components Model". Em: *18th ECOOP Doctoral Symposium and PhD Student Workshop*. 2008, p. 9 (ver p. 37).

[40] Eduardo Guerra, Clovis Fernandes e Fábio Fagundes Silveira. “Architectural patterns for metadata-based frameworks usage”. Em: *Proceedings of the 17th Conference on Pattern Languages of Programs*. ACM. 2010, p. 4 (ver pp. 37, 45).

[41] Eduardo Guerra e Elisa Yumi Nakagawa. “Relating patterns and reference architectures”. Em: *Proceedings of the 22nd Conference on Pattern Languages of Programs*. The Hillside Group. 2015, p. 8 (ver p. 84).

[42] Eduardo Guerra, Jerffeson de Souza e Clovis Fernandes. “Pattern Language for the Internal Structure of Metadata-Based Frameworks”. English. Em: *Transactions on Pattern Languages of Programming III*. Ed. por James Noble, Ralph Johnson, Uwe Zdun e Eugene Wallingford. Vol. 7840. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 55–110. ISBN: 978-3-642-38675-6. DOI: 10.1007/978-3-642-38676-3_3. URL: http://dx.doi.org/10.1007/978-3-642-38676-3_3 (ver pp. 3, 23).

[43] A. Hen-Tov, D. H. Lorenz e L. Schachter. “ModelTalk: A Framework for Developing Domain Specific Executable Models”. Em: *Proc. 8th Ann. OOPSLA Workshop Domain-Specific Modeling*. 2009 (ver p. 36).

[44] Atzmon Hen-Tov, David H Lorenz, Assaf Pinhasi e Lior Schachter. “ModelTalk: When everything is a domain-specific language”. Em: *Software, IEEE* 26.4 (2009), pp. 39–46 (ver p. 36).

[45] Jack Herrington. *Code generation in action*. Manning Publications Co., 2003 (ver pp. 3, 23).

[46] Myles Hollander, Douglas A Wolfe e Eric Chicken. *Nonparametric statistical methods*. John Wiley & Sons, 2013 (ver p. 127).

[47] Adrian Holovaty e Jacob Kaplan-Moss. *The Definitive Guide to Django*. Apress, 2009 (ver p. 34).

[48] Andrew Hunt e David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000 (ver pp. 3, 22).

[49] Eric Jendrock. *The Java EE 5 Tutorial*. Prentice Hall Professional, 2006 (ver p. 94).

[50] Jacob W Jespersen e Jesper Linvald. “Investigating user interface engineering in the model driven architecture”. Em: *Proceedings of the Interact 2003 Workshop on Software Engineering and HCI*. 2003 (ver p. 32).

[51] Thomas V Judkins e Christopher D Gill. “Synthesizer, A Pattern Language for Designing Digital Modular Synthesis Software”. Em: (2000) (ver p. 81).

[52] Natalia Juristo e Ana M Moreno. *Basics of software engineering experimentation*. Springer Science & Business Media, 2013 (ver pp. 12, 114).

[53] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm e William G Griswold. “An overview of AspectJ”. Em: *ECOOP 2001—Object-Oriented Programming*. Springer, 2001, pp. 327–354 (ver p. 30).

[54] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier e John Irwin. *Aspect-oriented programming*. Springer, 1997 (ver pp. 3, 30).

[55] Bhawna Kohli. *Enterprise Application Market - Opportunities and Forecasts, 2013 - 2020*. URL: https://www.alliedmarketresearch.com/enterprise-application-market (acedido em 07/12/2015) (ver p. 1).

[56] Ivan Kurtev, Klaas van den Berg e Frédéric Jouault. “Rule-based modularization in model transformation languages illustrated with ATL”. Em: *Science of computer programming* 68.3 (2007), pp. 138–154 (ver p. 33).

[57] Craig Larman. *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. Pearson Education India, 2005 (ver p. 58).

[58] Michael Lawley e Kerry Raymond. “Implementing a practical declarative logic-based model transformation engine”. Em: *Proceedings of the 2007 ACM symposium on Applied computing*. ACM. 2007, pp. 971–977 (ver p. 33).

[59] Matthew Lee, Ben-Zion Barta e Peter Juliff. *Software quality and productivity: theory, practice, education and training*. Springer, 2013 (ver p. 18).

[60] Helmut Leitner. *Pattern Theory: Introduction and Perspectives on the Tracks of Christopher Alexander*. CreateSpace Independent Publishing, 2015. ISBN: 978-1505637434 (ver p. 10).

[61] Theo Dirk Meijler, Jan Pettersen Nytun, Andreas Prinz e Hans Wortmann. "Supporting fine-grained generative model-driven evolution". Em: *Software & Systems Modeling* 9.3 (2010), pp. 403–424 (ver p. 32).

[62] Gerrit Meixner, Gaëlle Calvary e Joëlle Coutaz. *Introduction to Model-Based User Interfaces - W3C Working Group Note 07 January 2014*. URL: http://www.w3.org/TR/mbui-intro/ (acedido em 07/12/2015) (ver pp. 2, 166).

[63] Stephen J Mellor, Tony Clark e Takao Futagami. "Model-driven development: guest editors' introduction." Em: *IEEE software* 20.5 (2003), pp. 14–18 (ver p. 32).

[64] Atif Memon, Ishan Banerjee e Adithya Nagarajan. "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing". Em: *Proceedings of the 10th Working Conference on Reverse Engineering*. IEEE Computer Society. 2003, p. 260 (ver p. 44).

[65] Michael S Mikowski e Josh C Powell. "Single Page Web Applications". Em: *B and W* (2013) (ver p. 63).

[66] James S Miller e Susann Ragsdale. *The common language infrastructure annotated standard*. Addison-Wesley Professional, 2004 (ver pp. 3, 23, 50).

[67] Pedro J Molina, Santiago Meliá e Oscar Pastor. "Just-ui: A user interface specification model". Em: *Computer-Aided Design of User Interfaces III*. Springer, 2002, pp. 63–74 (ver pp. 50, 78).

[68] Giulio Mori, Fabio Paterno e Carmen Santoro. "Design and development of multidevice user interfaces through multiple logical descriptions". Em: *Software Engineering, IEEE Transactions on* 30.8 (2004), pp. 507–520 (ver pp. 32, 45, 57, 78).

[69] Chokri Mraidha, Yann Tanguy, Christophe Jouvray, François Terrier e Sébastien Gérard. "An execution framework for MARTE-based models". Em: *Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on*. IEEE. 2008, pp. 222–227 (ver p. 32).

[70] Brad A Myers e Mary Beth Rosson. “Survey on user interface programming”. Em: *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM. 1992, pp. 195–202 (ver pp. 2, 166).

[71] Sinval Vieira Mendes Neto, Rodrigo Almeida Vilar e Ayla Dantas. “The Dynamic Relations Pattern”. Em: *10th Latin American Conference on Pattern Languages of Programs - SugarLoaf PLoP*. Ilha bela, Sao Paulo, Brazil, 2014 (ver pp. 77, 78).

[72] Mozilla Developer Network. *Introduction to Object-Oriented JavaScript*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript (acedido em 16/03/2016) (ver p. 100).

[73] Sam Newman. *Building Microservices*. "O’Reilly Media, Inc.", 2015 (ver pp. 2, 17).

[74] OMG. *Meta Object Facility (MOF) Core Specification Version 2.4.1*. Rel. téc. 2013 (ver pp. 4, 25).

[75] OMG. *Unified Modeling Language (UML) Version 2.4.1 Superstructure specification*. Rel. téc. 2011 (ver pp. 4, 25).

[76] Terence John Parr. “Enforcing strict model-view separation in template engines”. Em: *Proceedings of the 13th international conference on World Wide Web*. ACM. 2004, pp. 224–233 (ver p. 57).

[77] The Statistics Portal. *Global enterprise application software revenue from 2011 to 2019 (in billion U.S. dollars)*. URL: http://www.statista.com/statistics/247554/global-enterprise-application-software-revenue/ (acedido em 07/12/2015) (ver p. 1).

[78] Ruby on Rails Guides. *Creating and Customizing Rails Generators & Templates*. URL: http://guides.rubyonrails.org/generators.html (acedido em 29/12/2015) (ver pp. 34, 57, 75).

[79] Ruby on Rails Guides. *Rails Application Templates*. URL: http://guides.rubyonrails.org/rails_application_templates.html (acedido em 30/12/2015) (ver pp. 34, 57).

[80] Chris Richardson. “A pattern language for J2EE web component development”. Em: *Proceedings of the 8th Conference on Pattern Languages and Programming*. ACM. 2001 (ver p. 81).

[81] Ken Rimple, Srini Penchikala e Ben Alex. *Spring Roo in action*. Manning, 2012 (ver p. 34).

[82] Janessa Rivera. *Worldwide Enterprise Application Spending to Grow 7.5 Percent to 149.9 Billion in 2015*. URL: http://www.gartner.com/newsroom/id/3119717 (acedido em 07/12/2015) (ver p. 1).

[83] Don Roberts e Ralph Johnson. “Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks”. Em: *Proceedings of the Third Conference on Pattern Languages and Programming*. 1996 (ver pp. 27, 36, 57, 76).

[84] Garme Keith Rocher. *The Definitive Guide to Grails*. Dreamtech Press, 2007 (ver pp. 23, 34).

[85] Patrick Royston. “Remark AS R94: A Remark on Algorithm AS 181: The W-test for Normality”. Em: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 44.4 (1995), pp. 547–551. ISSN: 00359254. DOI: 10.2307/2986146. URL: http://dx.doi.org/10.2307/2986146 (ver p. 127).

[86] Pramod J. Sadalage e Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. 1st. Addison-Wesley Professional, 2012. ISBN: 0321826620, 9780321826626 (ver pp. 2, 94).

[87] Nikos A Salingaros. “The structure of pattern languages”. Em: *Architectural Research Quarterly* 4.02 (2000), pp. 149–162 (ver p. 79).

[88] Wilane Carlos da Silva e Juliano Lopes de Oliveira. “Gerência de Interface Homem-Computador para Sistemas de Informação Empresariais: uma abordagem baseada em modelos”. Em: *iSys-Revista Brasileira de Sistemas de Informação* 2.1 (2009) (ver p. 32).

[89] Rini van Solingen, Vic Basili, Gianluigi Caldiera e H. Dieter Rombach. “Goal Question Metric (GQM) Approach”. Em: *Encyclopedia of Software Engineering*. John

Wiley & Sons, 2002. ISBN: 9780471028956. DOI: 10.1002/0471028959.sof142. URL: http://dx.doi.org/10.1002/0471028959.sof142 (ver p. 12).

[90] Adrian Stanciulescu, Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte e Francisco Montero. "A transformational approach for multimodal web user interfaces based on UsiXML". Em: *Proceedings of the 7th international conference on Multimodal interfaces*. ACM. 2005, pp. 259–266 (ver p. 33).

[91] Gabriele Taentzer, Dirk Müller e Tom Mens. "Specifying domain-specific refactorings for andromda based on graph transformation". Em: *Applications of Graph Transformations with Industrial Relevance*. Springer, 2008, pp. 104–119 (ver p. 23).

[92] Dave Thomas e David Hansson. *Agile Web Development with Rails*. Pragmatic Bookshelf, 2006. ISBN: 0-9766940-0-X (ver pp. 33, 34).

[93] Martijn Van Welie e Gerrit C Van der Veer. "Pattern languages in interaction design: Structure and organization". Em: *Proceedings of interact*. Vol. 3. 2003, pp. 1–5 (ver p. 80).

[94] Jean Vanderdonckt. "A MDA-compliant environment for developing user interfaces of information systems". Em: *Advanced Information Systems Engineering*. Springer. 2005, pp. 16–31 (ver pp. 3, 32).

[95] Jean Vanderdonckt. "Model-driven engineering of user interfaces: Promises, successes, failures, and challenges". Em: *Proceedings of ROCHI* 8 (2008) (ver p. 32).

[96] Rodrigo A Vilar, Anderson A Lima, Hyggo O Almeida e Angelo Perkusich. "Impact of unanticipated software evolution on development cost and quality: An empirical evaluation". Em: *Proceedings of 27th Int. Conf. Software Engineering and Knowledge Engineering*. 2015, pp. 128–133 (ver p. 114).

[97] Rodrigo A Vilar, Anderson A Lima, Hyggo O Almeida e Angelo Perkusich. "Unanticipated Software Evolution: Evaluating the Impact on Development Cost and Quality". Em: *International Journal of Software Engineering and Knowledge Engineering* 25.09n10 (2015), pp. 1727–1731 (ver p. 114).

[98] Rodrigo Vilar, Delano Oliveira e Hyggo Almeida. "Rendering patterns for enterprise applications". Em: *Proceedings of the 20th European Conference on Pattern Languages of Programs*. ACM. 2015, p. 22 (ver pp. 14, 39, 146).

[99] León Welicki, Joseph W Yoder e Rebecca Wirfs-Brock. "Rendering patterns for adaptive object-models". Em: *Proceedings of the 14th Conference on Pattern Languages of Programs*. ACM. 2007 (ver pp. 3, 5, 23, 35, 45, 51, 75, 76).

[100] León Welicki, Joseph W Yoder, Rebecca Wirfs-Brock e Ralph E Johnson. "Towards a pattern language for adaptive object models". Em: *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM. 2007, pp. 787–788 (ver pp. 46, 61).

[101] A Yassine. "An introduction to modeling and analyzing complex product development processes using the design structure matrix (DSM) method". Em: *Urbana* 51.9 (2004), pp. 1–17 (ver p. 171).

[102] Joseph W Yoder, Federico Balaguer e Ralph Johnson. "Architecture and design of adaptive object-models". Em: *ACM Sigplan Notices* 36.12 (2001), pp. 50–60 (ver p. 47).

[103] Joseph W. Yoder e Ralph Johnson. "The Adaptive Object-Model Architectural Style". English. Em: *Software Architecture*. Ed. por Jan Bosch, Morven Gentleman, Christine Hofmeister e Juha Kuusela. Vol. 97. IFIP — The International Federation for Information Processing. Springer US, 2002, pp. 3–27. ISBN: 978-1-4757-6538-0. DOI: 10.1007/978-0-387-35607-5_1. URL: http://dx.doi.org/10.1007/978-0-387-35607-5_1 (ver pp. 3, 23, 35, 94).

Apêndice A

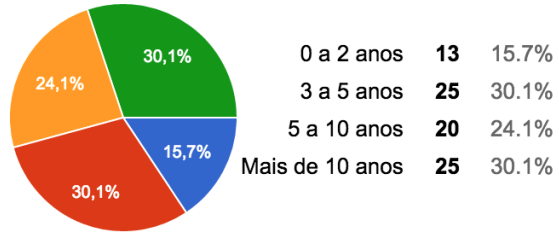
Visão dos desenvolvedores de software sobre GUI para aplicações corporativas

Esta pesquisa foi realizada com a finalidade de observar as características do desenvolvimento de GUI para Aplicações corporativas. Foi disponibilizado um formulário online, que foi respondido por 83 desenvolvedores de software, participantes de grupos temáticos de programação no Estado da Paraíba/Brasil, entre os dias 22/01/2016 e 02/02/2016. Dessa forma, pôde-se corroborar a relevância deste trabalho de tese, além de guiar alguns aspectos da sua metodologia.

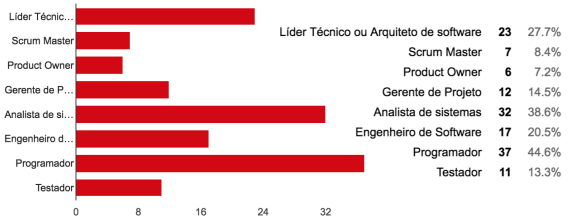
O grupo de desenvolvedores de software foi escolhido como público alvo da pesquisa posto que eles são capazes de opinar sobre a complexidade das telas das aplicações, não apenas pelo seu aspecto visual, mas também avaliando quão difícil seria desenvolver cada tela. Essa decisão influenciou no resultado da pesquisa, tanto que a maior parte das aplicações mais citadas foi ferramentas de auxílio à gestão de projetos de desenvolvimento. No entanto, esse resultado pode ser considerado válido, pois identificadas telas complexas de sistemas reais, que foram reproduzidas a fim de avaliar o uso do Angular M em cenários complexos.

As questões desta pesquisa foram divididas em três partes principais: perfil do desenvolvedor, ver sumário das respostas na Figura A.1; perfil do usuário de Aplicações corporativas; e a influência da GUI no desenvolvimento de Aplicações corporativas.

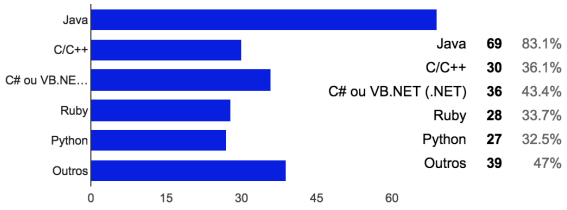
Perfil do desenvolvedor. Na Figura A.1a, pode-se ver que 54% dos desenvolvedores que responderam à pesquisa possuem mais de 5 anos de experiência no desenvolvimento de Aplicações corporativas, possuindo provavelmente perfil sênior nessa atividade. Esse



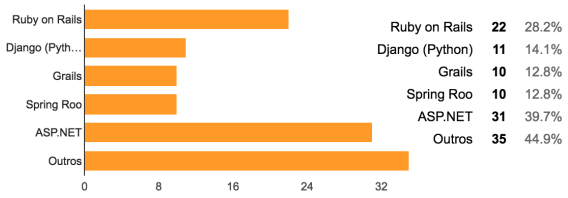
(a) Anos de experiência no desenvolvimento de aplicações corporativas



(b) Papéis já exercidos pelo desenvolvedor no time



(c) Experiência com linguagens de programação



(d) Experiência com arcabouços *web*

Figura A.1: Perfil do desenvolvedores que responderam à pesquisa

valor é confirmado pelos dados da Figura A.1b, onde se vê que 28% dos desenvolvedores se declararam arquitetos ou líderes técnicos. Desse modo, a pesquisa conseguiu envolver um número relevante de desenvolvedores experientes que opinaram sobre a complexidade da GUI em aplicações corporativas. Dentre as linguagens e arcabouços utilizados para o desenvolvimento de aplicações corporativas, destacam-se *Java*, *.NET* e *Ruby on Rails*.

Para compreender o **perfil do** desenvolvedor como **usuário de Aplicações corporativas**, foi elaborada uma grade de perguntas, onde cada linha representava uma aplicação corporativa sugerida na pesquisa e as colunas representavam o nível da experiência do usuário. As aplicações sugeridas foram classificadas em: comércio eletrônico (*Magento, Shopify, WooCommerce*); ERP (*iDempiere, Protheus, SAP, Salesforce*); CRM (*SuiteCRM, Fat Free CRM, Odoo*); gestão de projetos (*Trac, Jira, Redmine, ProjectLibre, Tuleap, dotProject*); *bug tracking* (*Bugzilla, Mantis*); gestão de testes (*TestLink*); gestão de processos de negócio (*Bonita BPM*); *Business intelligence* (*Pentaho*); e redes sociais (*Discourse*). A experiência do usuário foi graduada em sete níveis – não conhece, nunca utilizou, uso superficial, uso básico, uso intermediário, uso avançado e contribui no projeto – onde o primeiro nível tem peso zero e o último nível tem peso seis. Alem dessa grade, foi disponibilizado um espaço onde o usuário pôde sugerir outras Aplicações corporativas com as quais tinha experiência, porém nenhuma

dessas sugestões apareceu em mais de uma resposta e todas foram descartadas.

A Figura A.2 apresenta um *ranking* das Aplicações corporativas mais relevantes no contexto desta pesquisa. A pontuação de cada ferramenta foi calculada a partir dos anos de experiência de cada desenvolvedor de software multiplicados pelo nível de envolvimento do desenvolvedor com cada ferramenta (de 0 para experiência no nível *não conhece* a 6 para experiência no nível *contribui no projeto*).

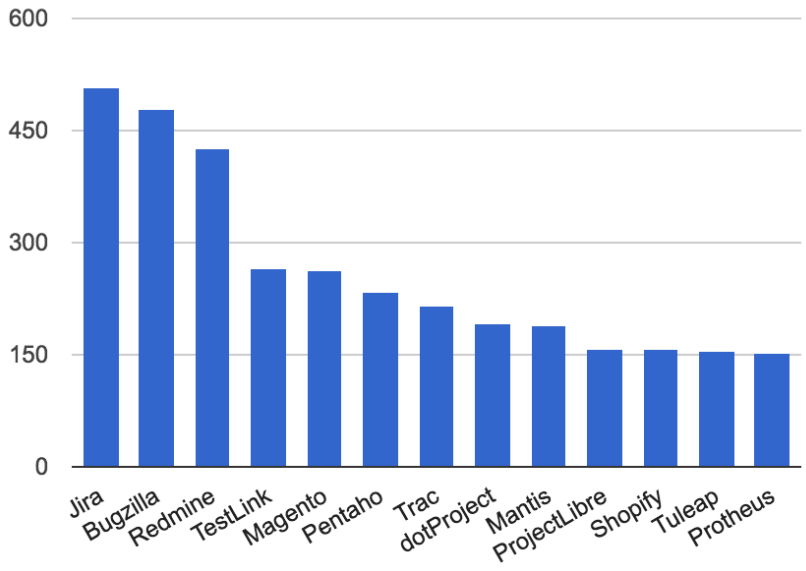


Figura A.2: *Ranking* das aplicações corporativas mais relevantes na pesquisa

A Tabela A.1 apresenta as características das aplicações mais relevantes na pesquisa: tipo da aplicação, abertura do código fonte, tecnologia utilizada e site. Cerca de 50% das aplicações apresentadas são do tipo Gestão de projetos. Esse valor já era esperado, pois esse tipo de ferramenta é bastante utilizado nos projetos de desenvolvimento de software, bem como as ferramentas de *Bug tracking* e gestão de testes. Os outros tipos de aplicações relevantes listados foram comércio eletrônico, *business intelligence* e ERP.

Também foi perguntado aos desenvolvedores quais seriam as telas mais complexas das Aplicações corporativas que eles conheciam. Obteve-se uma lista das telas mais complexas, ordenada pela relevância da ferramenta:

- Cadastro de atividades no *Jira* (citada 3 vezes);
- Pesquisa de atividades no *Jira*;

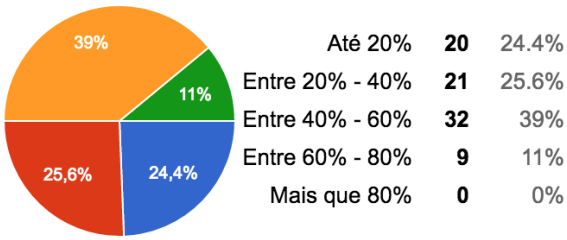
Tabela A.1: Análise das ferramentas mais relevantes da pesquisa

Ranking	**Ferramenta**	**Tipo**	**Código aberto?**	**Tecnologia**	**Site**
1	Jira	Gestão de projetos	Não	Java	www.atlassian.com/software/jira
2	Bugzilla	*Bug tracking*	Sim	Perl	github.com/bugzilla/bugzilla
3	Redmine	Gestão de projetos	Sim	Rails	github.com/redmine/redmine
4	TestLink	Gestão de testes	Sim	PHP	sourceforge.net/projects/testlink
5	Magento	Comércio eletrônico	Sim	PHP	github.com/magento/magento2
6	Pentaho	*Business intelligence*	Sim	Java	github.com/pentaho/pentaho-platform
7	Trac	Gestão de projetos	Sim	Python	svn.edgewall.org/repos/trac/
8	dotProject	Gestão de projetos	Sim	PHP	sourceforge.net/projects/dotproject/
9	Mantis	*Bug tracking*	Sim	PHP	github.com/mantisbt/mantisbt
10	ProjectLibre	Gestão de projetos	Sim	Java	sourceforge.net/projects/projectlibre/
11	Shopify	Comércio eletrônico	Não	Rails	www.shopify.com
12	Tuleap	Gestão de projetos	Sim	PHP	github.com/Enalean/tuleap
13	Protheus	ERP	Não	?	www.totvs.com/software-de-gestao

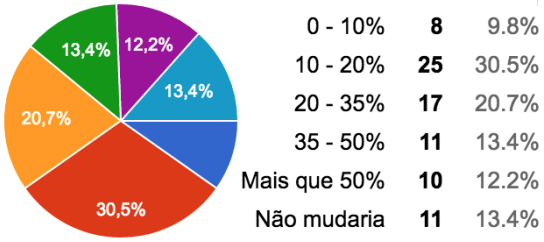
- Tela de planejamento de *sprint* no *Jira*;
- Pesquisa de bugs no *Bugzilla* (citada 6 vezes);
- Cadastro de atividades no *Redmine* (citada 2 vezes);
- Pesquisa de atividades no *Redmine* (citada 2 vezes);
- Relatório de progresso no *Redmine*;
- Cadastro de casos de teste no *Test Link* (citada 2 vezes);
- Busca de casos de testes e testes no *TestLink*;
- Configuração de plano de teste no *TestLink*;
- Cadastro de produtos agrupados no *Magento*;
- Tela do carrinho de compras no *Magento*;
- Tela de pagamento no *Magento*;
- Cadastro de imagens no *Magento*;
- Cadastro de atividades e estórias no *Trac*;
- Cadastro de *bugs* no *Trac*;
- Pesquisa de bugs por tipo no *Mantis*;
- Administração de usuários e grupos em vários sistemas;
- Criação de consultas e relatórios customizados em vários sistemas;
- Telas de variação de imposto, classificação fiscal e que exijam informações contábeis em módulos fiscais de diversos sistemas.

Algumas telas dessa lista foram utilizadas em um experimento a fim de emular o uso da biblioteca Angular M em sistemas reais, avaliando o seu impacto na produtividade do desenvolvimento de GUI em Aplicações corporativas.

Na parte final do formulário, os desenvolvedores responderam duas questões gerais sobre a **influência da GUI no desenvolvimento de Aplicações corporativas**. Essas respostas foram resumidas na Figura A.3.



(a) Percentual do tempo gasto com a GUI no desenvolvimento de aplicações corporativas



(b) Percentual de economia que motivaria a mudança na tecnologia de GUI

Figura A.3: Influência da GUI no desenvolvimento de aplicações corporativas

Ao se calcular a média dos valores da Figura A.3a, pode-se ver que, segundo os desenvolvedores, a GUI demanda por volta de 31% do esforço no desenvolvimento de aplicações corporativas. Esse valor é um pouco menor do que os valores obtidos em pesquisas anteriores [70, 62], porém ainda se trata de um montante significante no custo dos projetos.

Conforme a Figura A.3b, se houver uma redução de 35% no esforço de desenvolvimento de GUI, 60% dos desenvolvedores mudariam da tecnologia que usam atualmente para a mais econômica.

Pode-se concluir, com esses resultados, que a pesquisa e o desenvolvimento de novas tecnologias que possam aumentar a produtividade no desenvolvimento de GUI para Aplicações corporativas é relevante. Assim sendo, justifica-se o estudo da nova abordagem que foi proposta por esta tese.

Apêndice B

Projeto de tela complexa utilizando a abordagem proposta

B.1 Estudo de viabilidade da linguagem de padrões

Este estudo consiste do projeto de componentes de GUI conforme a linguagem de padrões proposta nesta tese, que foram combinados com o modelo do domínio de uma aplicação corporativa, a fim de simular a criação de uma tela complexa e avaliar teoricamente a viabilidade dos padrões.

Por limitações de tempo, este estudo utilizou artefatos de projeto (diagramas) em vez de artefatos de implementação e o escopo foi restrito à tela de criação de tarefas no *Jira* (Figura B.1). Uma vez que essa ferramenta não possui código aberto, o modelo do domínio que envolve a tela em questão foi obtido através da análise da tela e foi modelado na Figura B.2.

O passo mais importante, neste estudo, foi o projeto dos metacomponentes capazes de construir a tela de criação de tarefas do *Jira*, utilizando os conceitos de porta, escopo e configuração, oriundos da linguagem de padrões proposta nesta tese. Na Figura B.3, pode-se ver que esses conceitos podem, em teoria, ser combinados com os metadados do domínio do *Jira* produzindo a tela esperada.

A solução projetada designou que a estrutura do formulário será desenhada pelo componente `VerticalLayoutForm`, que coloca os nomes dos campos na área amarelada à esquerda e os campos propriamente ditos no lado direito. Esse componente define duas portas: *line* para as propriedades e *relation* para os relacionamentos.

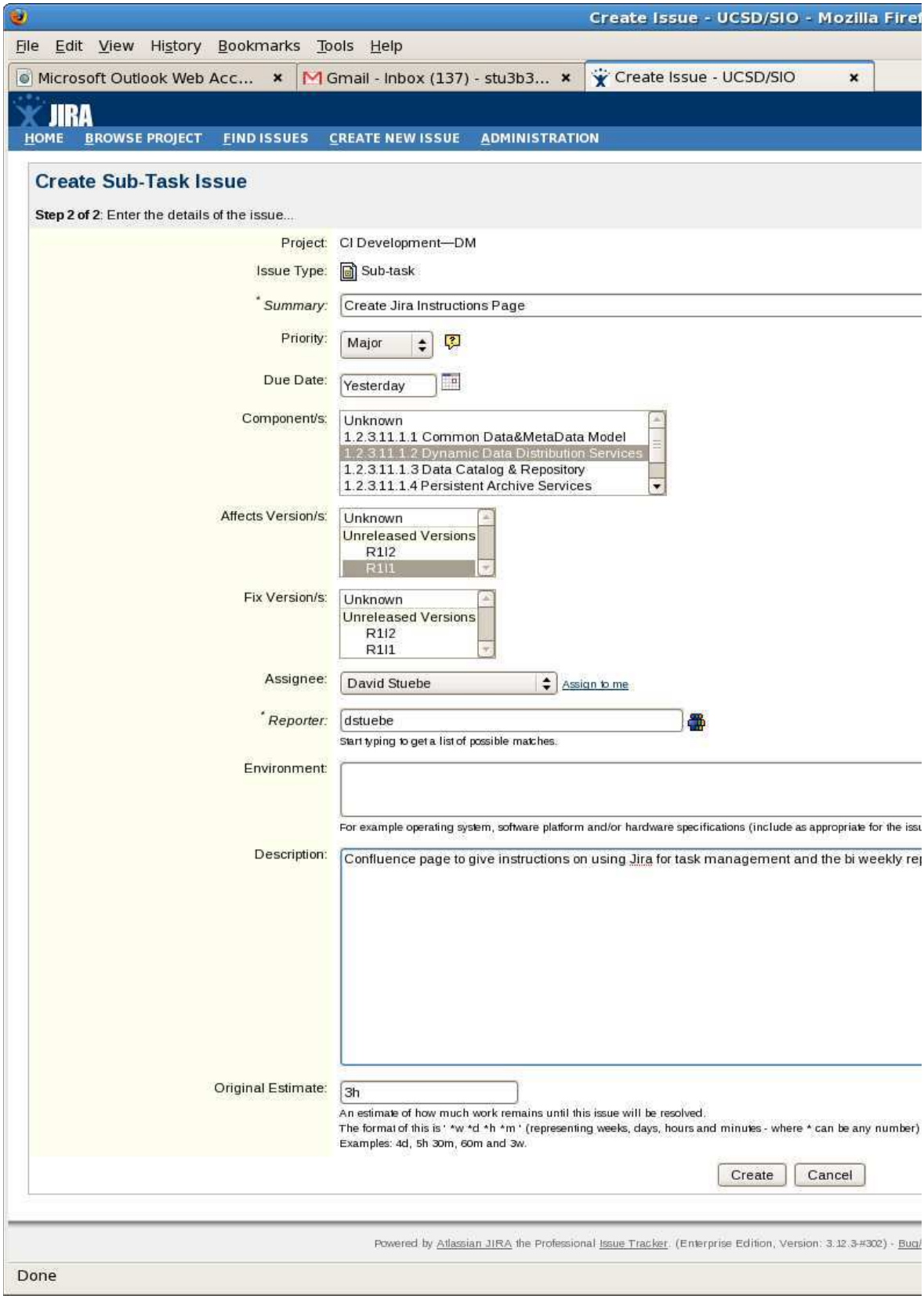


Figura B.1: Criação de tarefas no *Jira*

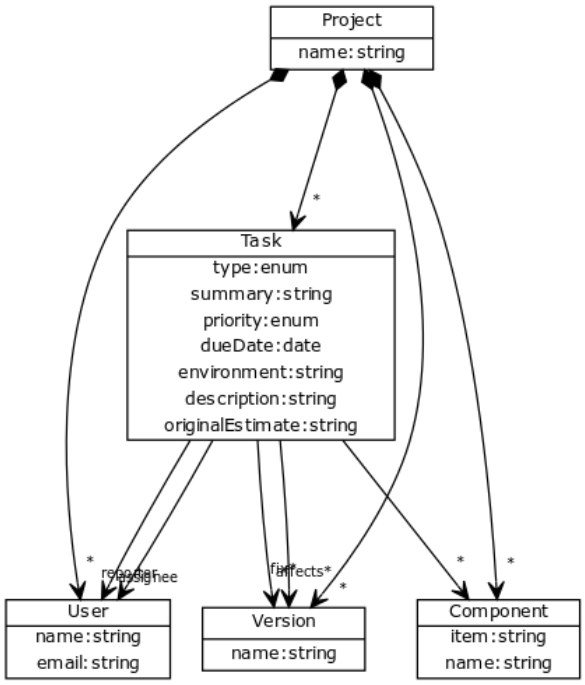


Figura B.2: Modelo do domínio para criação de tarefas no *Jira*

O campo *Project* representa o relacionamento MUITOS-PARA-UM de `Task` para `Project`. Como a cardinalidade oposta do relacionamento é UM e a entidade oposta é `Project`, o componente selecionado para desenhar este campo é `RelationLabel`, que escreve o nome do projeto, sem a opção de editá-lo.

O campo *Issue type* representa a propriedade *type* da entidade `Task`, à qual é associado especificamente o componente `IconLabelField`, que escreve o tipo da tarefa sem opção de edição e com um ícone associado a cada tipo. Esse comportamento é obtido por causa da configuração desse componente na porta *line*, que define uma classe CSS (`iconClass=task`) para buscar os ícones e define o campo como não editável.

O campo *Summary* é desenhado pelo componente padrão para todas as propriedades: `TextField`. O campo *Priority* é associado ao componente padrão para todos os *enumerations*, `EnumCombobox`, com um ícone de ajuda (*tip*) habilitado. *Due date* é tratado pelo componente padrão para os campos do tipo Data (`DateField`).

Os campos *Components*, *Affects Versions* e *Fix Versions* são desenhados pelo mesmo componente, `ListBox`, que é padrão para todos os relacionamentos onde a cardinalidade oposta é MUITOS. O campo *assignee* é associado especificamente ao componente *User-Combobox* que lista os usuários e fornece um link para selecionar o usuário logado na sessão.

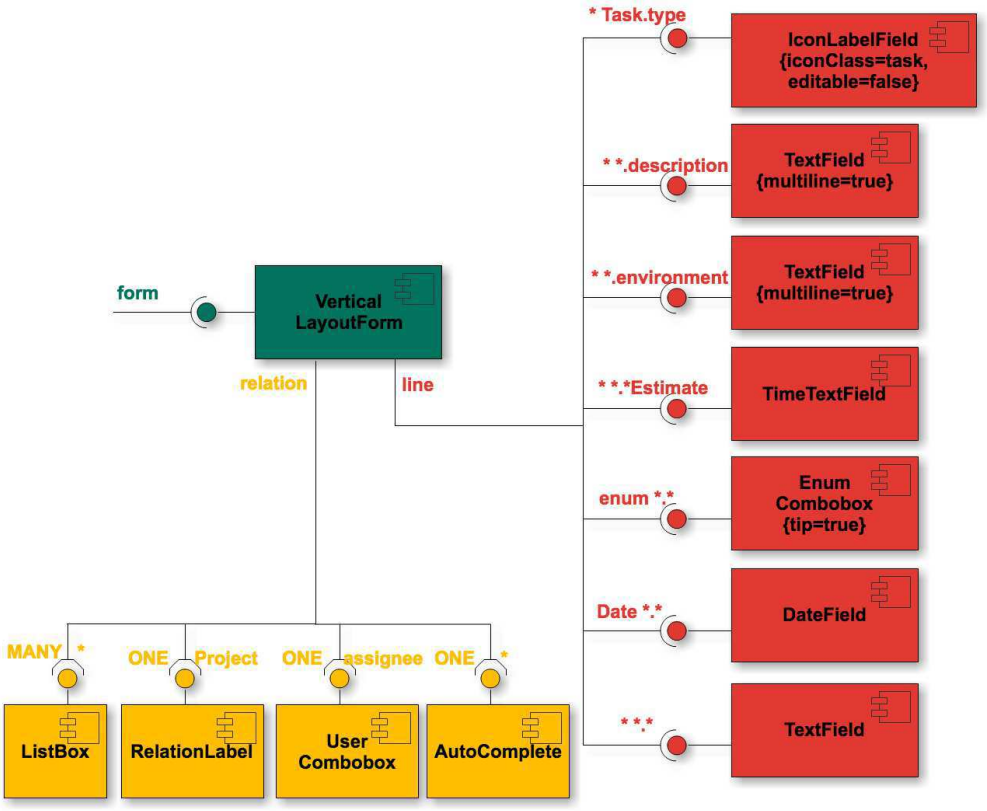


Figura B.3: *Widgets* para criação de tarefas no *Jira*

O campo *Reporter* não possui regra de escopo específico, portanto é tratado pelo componente padrão dos relacionamentos com cardinalidade oposta UM: `AutoComplete`.

Os campos *Environment* e *Description* de quaisquer entidades serão montados pelo componente *TextField* com configuração de linhas múltiplas, gerando, dessa forma, elementos *textarea* de HTML. Por fim, todos os campos cujo nome termine com *Estimated* serão desenhados por `TimeTextField`, que permite a entrada de valores representando minutos, horas, dias e semanas.

Ao avaliar essa tela, conclui-se que a linguagem de padrões proposta nesta tese é capaz de implementá-la, a partir do modelo de domínio sugerido na Figura B.2 e dos componentes projetados na Figura B.3. Dessa forma, pôde-se prosseguir com a pesquisa implementando um arcabouço com os conceitos contidos na linguagem de padrões.

B.2 Análise do impacto da linguagem de padrões no acoplamento GUI–Domínio

Além do estudo de viabilidade da linguagem de padrões, foi realizada uma análise teórica do seu efeito no acoplamento GUI–Domínio de Aplicações corporativas. A dependência entre os componentes arquiteturais de Aplicações corporativas foi analisada através de matrizes de dependência [101], e as características que diferiram de uma arquitetura usual em camadas foram evidenciadas a seguir.

A Figura B.4 representa a arquitetura de uma aplicação dividida em camadas. A camada de GUI conhece diretamente o modelo do domínio (entidades, propriedades e relacionamentos) que são exibidos na API da lógica de negócio. A lógica também é referenciada diretamente pela GUI e invoca a camada de persistência.

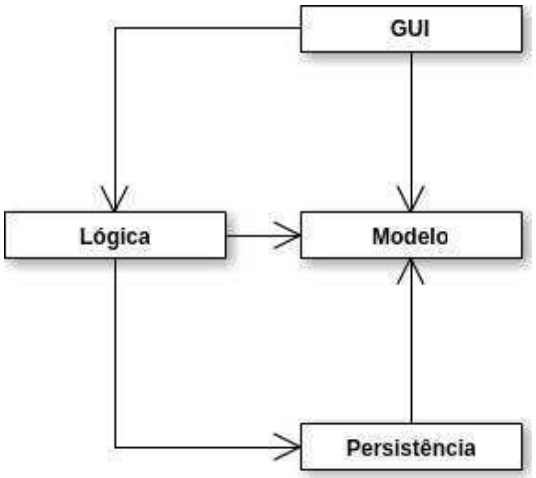


Figura B.4: Dependência entre os componentes de uma aplicação dividida em camadas

A Tabela B.1 apresenta uma matriz de dependências entre os componentes da Figura B.4. Cada célula marcada com ✓significa que o componente representado na coluna conhece o componente da linha.

Na Figura B.5 e na Tabela B.2, foi representado um sistema com arquitetura orientada a recursos, como REST. Nesse caso, as ações são padronizadas, como por exemplo os verbos HTTP (POST, PUT, GET e DELETE), para todos os recursos do sistema. Consequentemente, a GUI não precisa conhecer a lógica diretamente e o acoplamento é menor. Todavia, ainda existe o acoplamento forte entre a GUI e o modelo do domínio. A célula em vermelho,

Tabela B.1: Matriz de dependências para uma aplicação dividida em camadas

	1. GUI	2. Lógica	3. Persistência	4. Modelo
1. GUI				
2. Lógica	✓			
3. Persistência		✓		
4. Modelo	✓	✓	✓	

nessa tabela, mostra que a dependência que havia da GUI para a Lógica foi removida.

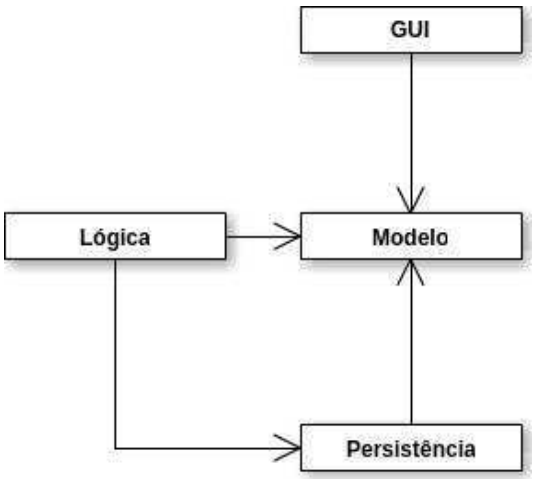


Figura B.5: Dependência entre os componentes de uma aplicação orientada a recursos

Em um cenário hipotético, a partir da arquitetura orientada a recursos, se aplica os padrões definidos nesta tese, e pode-se obter uma arquitetura semelhante à da Figura B.6. A GUI original foi dividida em três componentes: GUI, composta por um conteiner de componentes e pelos componentes propriamente ditos; as estruturas de metadados (`EntityType`, `PropertyType`, etc.), que não variam, portanto não geram problemas para os componentes que dependem delas; e os dados de configuração, que podem ser definidos em bancos de dados ou arquivos externos e podem ser alterados facilmente.

Em vez de referenciar o domínio diretamente, a GUI conhece apenas os seus metadados. O modelo pode ser anotado com instruções para auxiliar na construção da GUI, por exemplo,

Tabela B.2: Matriz de dependências para uma aplicação orientada a recursos

	1. GUI	2. Lógica	3. Persistência	4. Modelo
1. GUI				
2. Lógica				
3. Persistência		✓		
4. Modelo	✓	✓	✓	

para dizer o nome reverso dos relacionamentos, por isso o modelo também pode conhecer os metadados.

Os dados de configuração, que implementam as regras de ligação entre portas e componentes, são o acoplamento remanescente entre a GUI e o domínio, nessa arquitetura. Todas as regras selecionam algum componente, portanto existe um acoplamento fraco (facilmente alterável) entre a configuração e a GUI. Além disso, quando um componente é aplicado em um escopo específico, o nome da entidade, da propriedade ou do relacionamento precisa ser descrito na regra. Logo, também existe um acoplamento fraco da configuração para o modelo do domínio.

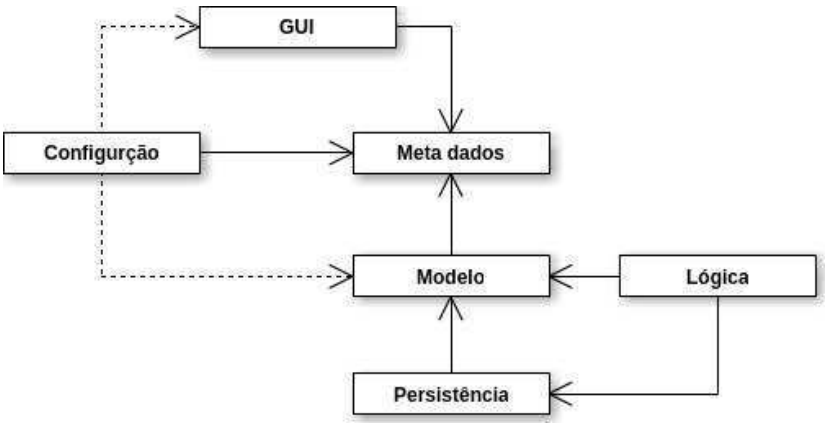


Figura B.6: Dependência entre os componentes da arquitetura proposta nesta tese

Na Tabela B.3, nota-se que o acoplamento da GUI para o Modelo do domínio foi removido (célula vermelha). Porém houve o surgimento de um acoplamento mais fraco da

Tabela B.3: Matriz de dependências para os componentes da arquitetura proposta nesta tese

	1. Configuração	2. GUI	3. Lógica	4. Persistência	5. Modelo	6. Metadados
1. Configuração						
2. GUI	•					
3. Lógica						
4. Persistência			✓			
5. Modelo	•		✓	✓		
6. Metadados	✓	✓			✓	

configuração para a GUI e para o modelo do domínio (células verdes). As dependências em relação aos metadados não serão consideradas (células azuis), pois esse componente possui uma estabilidade forte e não deve sofrer alterações. Desse modo, não deve haver impacto nos componentes que dependem dos metadados: configuração, GUI e modelo do domínio.

Este estudo teve caráter teórico e precisa ser confirmado com a análise de matriz de dependências em sistemas reais. A continuação desse estudo foi realizado no experimento descrito no capítulo 7, onde se confirmou que a remoção da dependência forte da GUI para o domínio é mais benéfica, na produtividade do desenvolvimento, do que a criação das dependências da configuração para a GUI e para o domínio.

Apêndice C

Script R para análise estatística

Código Fonte C.1: Script R utilizado para analisar os resultados do experimento

```
1  pValue<- function(data) {
2    round(as.numeric(data['p.value']), digits = 5)
3  }
4
5  isSig<- function(data) {
6    pValue(data) < 0.05
7  }
8
9  printNotSig <- function(type, res, text) {
10   if (isSig(res)) {
11     print(paste(type, 'is not', text, pValue(res)))
12   } else {
13     print(paste(type, 'is', text, pValue(res)))
14   }
15 }
16
17 printSig <- function(type, res, text) {
18   if (isSig(res)) {
19     print(paste(type, 'is', text, pValue(res)))
20   } else {
21     print(paste(type, 'is not', text, pValue(res)))
22   }
23 }
24
```

```
25 analise <- function(data, arquivo) {
26   dataMg = subset(data, data$Tecn == "mg")
27   dataRoR = subset(data, data$Tecn == "ror")
28   cat("Mg: ", dataMg$Tempo, "\n")
29   cat("RoR: ", dataRoR$Tempo, "\n")
30   mgNorm = shapiro.test(dataMg$Tempo)
31   rorNorm = shapiro.test(dataRoR$Tempo)
32   printNotSig('mg', mgNorm, 'normal')
33   printNotSig('ror', rorNorm, 'normal')
34   if (!isSig(mgNorm) && !isSig(rorNorm)) {
35     print('t-test')
36     tt = t.test(dataMg$Tempo, dataRoR$Tempo)
37     printSig('mg', tt, 'different of ror')
38     meanMg = tt$estimate[[1]]
39     meanRoR = tt$estimate[[2]]
40     print(paste(meanMg, meanRoR))
41     tl = t.test(dataMg$Tempo, dataRoR$Tempo, alternative = 'l')
42     printSig('mg', tl, 'less than ror')
43     tg = t.test(dataMg$Tempo, dataRoR$Tempo, alternative = 'g')
44     printSig('mg', tg, 'greater than ror')
45   } else {
46     print('wilcox-test')
47     wt = wilcox.test(dataMg$Tempo, dataRoR$Tempo, alternative = 't')
48     printSig('mg', wt, 'different of ror')
49     medianMg = median(dataMg$Tempo)
50     medianRoR = median(dataRoR$Tempo)
51     print(paste(medianMg, medianRoR))
52     wl = t.test(dataMg$Tempo, dataRoR$Tempo, alternative = 'l')
53     printSig('mg', wl, 'less than ror')
54     wg = t.test(dataMg$Tempo, dataRoR$Tempo, alternative = 'g')
55     printSig('mg', wg, 'greater than ror')
56   }
57   data$Tecn = droplevels(data$Tecn)
58   png(filename=paste("Documents/10327780xmrqqbbbwdfw/figuras/validacao/",
         arquivo, ".png", sep=''))
59   boxplot(Tempo ~ Tecn, data = data, ylab = "Tempo em segundos", boxwex
       =0.7)
```

```
60    dev.off()
61  }
62
63  tarefa <- function(data, num) {
64    subset(data, data$Tarefa == num)
65  }
66
67  reuso <- function(data, val) {
68    subset(data, data$ReusaComponente == val)
69  }
70
71  data = read.csv("Downloads/Dados.csv")
72  data = data[-c(1), ]
73  data = subset(data, !is.na(data$NTempo))
74
75  analise(data, 'boxplot')
76
77  analise(reuso(data, TRUE), 'boxplotReuso')
78  analise(reuso(data, FALSE), 'boxplotNaoReuso')
79
80  analise(tarefa(data, 1), 'boxplot1')
81  analise(tarefa(data, 2), 'boxplot2')
82  analise(tarefa(data, 3), 'boxplot3')
83  analise(tarefa(data, 4), 'boxplot4')
84  analise(tarefa(data, 5), 'boxplot5')
85  analise(tarefa(data, 6), 'boxplot6')
86  analise(tarefa(data, 7), 'boxplot7')
87  analise(tarefa(data, 8), 'boxplot8')
88  analise(tarefa(data, 9), 'boxplot9')
89  analise(tarefa(data, 10), 'boxplot10')
90  analise(tarefa(data, 11), 'boxplot11')
91  analise(tarefa(data, 12), 'boxplot12')
```