UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

OBCRUD: UM FRAMEWORK PARA CONSTRUÇÃO DINÂMICA DE INTERFACES GRÁFICAS PARA PERSISTÊNCIA DE DADOS.

Ivan Paulino Pereira

UNIFEI
Itajubá
Julho, 2017

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

Ivan Paulino Pereira

OBCRUD: UM FRAMEWORK PARA CONSTRUÇÃO DINÂMICA DE INTERFACES GRÁFICAS PARA PERSISTÊNCIA DE DADOS.

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

Área de Concentração: Sistemas de Computação

Orientador: Prof. Dr. Enzo Seraphim

UNIFEI
Itajubá
Julho, 2017

Dados Internacionais de Catalogação na Publicação (CIP)

Pereira, Ivan Paulino.

P436o ObCrud: Um Framework para construção dinâmica de Interfaces Gráficas para Persistência de Dados. / Ivan Paulino Pereira. – Itajubá: UNIFEI, 2017.

xv + 146 f. : il. ; 29.7

Orientador: Enzo Seraphim.

Dissertação (Mestrado em Ciência e Tecnologia da Computação) – Universidade Federal de Itajubá, Itajubá, 2017.

1. Interface de Usuário. 2. Desenvolvimento de Interfaces do Usuário Baseadas em Modelo. 3. Engenharia Dirigida a Modelos. 4. Scaffoldings. 5. Metaprogramação. 6. CRUD. I. Seraphim, Enzo, orient. II. Universidade Federal de Itajubá. III. Título.

CDU 004.4‘2

Banca Examinadora:

Dr. Enzo Seraphim _____

Orientador

Universidade Federal de Itajubá

Dr. Marcelo Zanchetta do Nascimento _____

Membro da Banca

Universidade Federal de Uberlândia

Dr. Edmilson Marmo Moreira _____

Membro da Banca

Universidade Federal de Itajubá

A meus pais Nairton (in memoriam) e Neide e à minha amada esposa e filha, Cássia e Lorena

Agradecimentos

Agradeço a Deus por tudo, pois sem ele nada seria possível ou faria sentido.

Agradeço a minha família que sempre me apoiou e me compreendeu quando não pude estar presente. Agradeço, em especial, a meu Pai, que apesar de ser padastro, sempre me tratou como um filho e me ensinou os valores da vida. Agradeço também minha à mãe que sempre me protegeu e por muito tempo ocupou as posições de pai e mãe.

Agradeço à minha esposa por me incentivar, por estar comigo nos momentos difíceis e me acalmar quando acreditava que nada daria certo, por me desculpar quando ficava bravo e por gerar nossa linda filha que com seu sorriso me dá fôlego novo para seguir em frente.

Agradeço ao professor Enzo, por me orientar neste trabalho e por servir de inspiração para ser um profissional e pessoa melhor.

Agradeço a todos os professores da UNIFEI que compartilharam seus conhecimentos e me fizeram crescer.

Agradeço aos colegas da UNIFEI, em especial a Helaine, o João e a professora Thatyana, que sempre estiveram dispostos a ajudar e a conversar.

Agradeço à UNIFEI pela disponibilização dos recursos físicos e pela oportunidade de realizar o curso de mestrado.

Agradeço ao IFSULDEMINAS pela flexibilização de horário.

Agradeço também aos participantes da banca pela disponibilidade de avaliar este trabalho.

Agradeço a todos que contribuíram para que eu pudesse concluir mais essa etapa importante da minha vida. Obrigado!

Talvez não tenha conseguido fazer o melhor, mas lutei para que o melhor fosse feito. Não sou o que deveria ser, mas Graças a Deus, não sou o que era antes.

Martin Luther King

Resumo

As Interfaces do Usuário (User Interfaces - UI) são um dos artefatos mais importantes de um *software*, muitas vezes o sucesso ou fracasso de um sistema está associado a qualidade da UI. Não obstante, o desenvolvimento das UIs é um processo pesado, que contribui para a baixa produtividade do desenvolvimento de sistemas. Ao longo dos últimos anos diversas pesquisas e ferramentas, baseadas no paradigma da Engenharia Dirigida por Modelo (Model-Driven Engineering - MDE), foram desenvolvidas com o intuito de simplificar e aumentar a produtividade no desenvolvimento de UIs. Apesar dos diversos estudos, as ferramentas, disponíveis atualmente, ainda apresentam algumas limitações ou suporte reduzido para gerar UIs baseadas em contexto de uso, para adicionar novas tarefas e para configurar a aparência e o comportamento da UI. Além dessas limitações, faz-se necessário que as UIs geradas e as ferramentas de automatização possuam boa usabilidade e que os modelos utilizados no desenvolvimento estejam sempre sincronizados com as UIs. Tendo em vista as características e limitações das ferramentas atuais, este trabalho apresenta a implementação do *framework ObCrud*, que faz uso das abordagens dirigidas por modelo, para automatizar o processo de criação de UIs, em sistemas orientados a negócio. O desenvolvimento do *framework* buscou implementar as características de qualidade descritas na literatura e as observadas nas diversas ferramentas de automatização avaliadas. Para validar o *framework ObCrud*, foram realizados experimentos que mostraram que sua produtividade é semelhante as ferramentas de *scaffoldings* e que as UIs geradas e o próprio *framework* possuem bons índices de usabilidade. As ideias e as técnicas utilizadas para o desenvolvimento da *ObCrud* podem contribuir para a evolução e melhoria das ferramentas atuais de geração de UIs CRUD.

Palavras-chave: Interface de Usuário, Desenvolvimento de Interfaces do Usuário Baseadas em Modelo, Engenharia Dirigida a Modelos, Scaffoldings, Metaprogramação, CRUD.

Abstract

ObCrud: A Framework for Dynamic Building of Graphical Interfaces for Data Persistence

UI Interfaces are one of the most important artifacts of a software, often the success or failure of a system is associated with UI quality. Nevertheless, the development of IUs is a cumbersome process, which contributes to the low productivity of systems development. Throughout the last years, several researches and tools, based on the paradigm of the model-driven engineering (MDE), have been developed with the purpose of simplifying and increasing the productivity in the development of IUs. Despite the various studies, the currently available tools still have some limitations or reduced support for generate UI based on usage context, adding new tasks, and configuring the appearance and behavior of the UI. In addition to these limitations, it is necessary that generated UIs and automation tools have good usability and that the models used in the development are always synchronized with UIs. Considering the characteristics and limitations of current tools, this study presents the implementation of the ObCrud framework, which uses model-driven approaches to automate the UI creation process, in business-oriented systems. The development of the framework sought to implement the quality characteristics described in the literature and those observed in the various automation tools evaluated. In order to validate the ObCrud framework, experiments were performed that showed that its productivity is similar to the scaffolding tools and that the UIs generated and the framework itself have good usability indexes. The ideas and techniques used for the development of ObCrud can contribute to the evolution and improvement of the current tools of generation of CRUD UIs.

Keywords: *User Interface, Model-Based User Interface Development, Model-Driven Engineering, Scaffoldings, Metaprogramming, CRUD.*

Lista de Figuras

2.1 Metamodelo simplificado do diagrama de classes da UML. 9
2.2 Relação entre elementos do metamodelo, modelo e domínio 10
2.3 Exemplos dos modelos CIM, PIM e PSM 11
2.4 Exemplo de níveis de plataforma do MDA 12
2.5 Sucessivas transformações de PIM em PSM 12
2.6 Extensão da UML com perfis e a criação de marcas 14
2.7 As ideias básicas por trás de *Model-Driven Software Development* 16
2.8 Modelo de Domínio 17
2.9 Modelo de Tarefas 18
2.10 Simplificação do Processo MBUID com CRF 20
2.11 Interface gráfica para cadastrar livros gerada pelo *scaffolding* do *Grails* 28
2.12 Esquema de funcionamento da metaprogramação 31
2.13 Digrama de classes: Sistema de autuação de trânsito 46

3.1 Arquitetura lógica do *ObCrud* 61
3.2 Diagrama de Classes do *ObCrud* 63
3.3 Processo de desenvolvimento do *ObCrud* 79
3.4 UIs CRUD básicas geradas pelo *ObCrud* 81
3.5 UIs CRUD notícia com anotações do modelo de apresentação 83
3.6 Exemplo de UIs com validação de dados 84
3.7 Diagrama de classes com associações 85
3.8 UIs CRUD cirurgia: Renderização dos *widgets* data e hora 88
3.9 UIs CRUD cirurgia: Renderização dos *widgets* de associação 89
3.10 UIs CRUD de listagem de associação 89
3.11 UIs de edição com e sem configurações de contexto de uso 91
3.12 UIs CRUD notícia com anotações do modelo de apresentação 92
3.13 UIs CRUD com novas tarefas 93
3.14 UIs CRUD antes e após a execução de uma precondição 95

4.1 Fases e atividades do experimento 98
4.2 Modelo de Domínio utilizado na 1ª atividade da fase de treinamento 99
4.3 Diagrama de objetos utilizado na 2ª atividade da fase de treinamento 100
4.4 Modelo de Domínio utilizado na 1ª atividade da fase de execução 101
4.5 Diagrama de objetos utilizado na 2ª atividade da fase de execução 101

4.6 Tempos *T1* gastos para gerar as UIs CRUDs 105
4.7 Tempos *T2* gastos para gerar as UIs CRUDs 106

A.1 Experimento 1 - Treinamento - Diagrama de classes 1 126
A.2 Experimento 1 - Treinamento - Diagrama de objetos 1 128
A.3 Diagrama de classes da fase de execução 129
A.4 Diagrama de objetos da fase de execução 129
A.5 Diagrama de classes da fase de treinamento 131
A.6 Diagrama de classes da fase de treinamento 133
A.7 Diagrama de objetos da fase de treinamento 134
A.8 Diagrama de classes da fase de execução 134
A.9 Diagrama de objetos da fase de execução 135

Lista de Tabelas

2.1 Rotas para `LivroController.groovy` 28
2.2 Métodos de introspecção da classe `Class` para membros públicos 35
2.3 Métodos de introspecção da classe `Class` para membros declarados 36
2.4 Métodos de intercessão das classes `Field`, `Method` e `Constructor` 36

3.1 Rotas geradas para os métodos da classe `AlunoController` 64

4.1 Problemas de usabilidade identificados no *ObCrud* 107
4.2 Resultado do teste estatístico para comparação das UIs 108
4.3 Pontução SUS dos *frameworks ObCrud* e *Grails* 109

B.1 Tempo gasto na execução das tarefas dos Experimentos 1 e 2 (em segundos) 137
B.2 Respostas dos participantes: Comparação de usabilidade das UIs 137
B.3 Respostas dos participantes e a pontuação SUS do sistema *ObCrud* 138
B.4 Respostas dos participantes e a pontuação SUS do *framework Grails* 138

E.1 Heurísticas de Nielsen 142
E.2 Grau de severidade 142

Lista de Códigos

2.1 Exemplo de *template* para criação de formulário HTML . . . 25
2.2 Exemplo de Contexto em Java . . . 25
2.3 Código gerado por meio do *Template* e do Contexto . . . 25
2.4 Classe de Domínio do *Grails* . . . 26
2.5 Classe `LivroController` gerada pelo comando `generate-all` . . . 27
2.6 *Scaffolding* dinâmico: Usando do nome da classe de Domínio . . . 29
2.7 *Scaffolding* dinâmico: Usando do nome da classe *Controller* . . . 29
2.8 Obtenção de objetos de `Class` . . . 34
2.9 Classe de modelo `Aluno` . . . 35
2.10 Uso de reflexão para obter os atributos da classe `Aluno` . . . 35
2.11 Exemplo de reflexão . . . 37
2.12 Declaração de tipo de anotação . . . 38
2.13 Sintaxe para anotação de elementos . . . 38
2.14 Reflexão: Obtendo elementos anotados . . . 39
2.15 Exemplo de classe de *controller* do VRaptor . . . 42
2.16 Exemplo de arquivo de visão do VRaptor . . . 42
2.17 Fábrica de conexões sem CDI . . . 43
2.18 Classe de acesso a dados sem CDI . . . 43
2.19 Classe de acesso a dados com CDI . . . 44
2.20 Fábrica de conexões com CDI . . . 45
2.21 Classe `SistemaAutuacao` sem uso de eventos . . . 47
2.22 Classe `SistemaNotificacao` sem evento . . . 48
2.23 Classe `SistemaAutuacao` com uso de eventos . . . 48
2.24 Classe `SistemaNotificacao` com evento . . . 49
2.25 Exemplo de Interceptador . . . 50
2.26 Anotação para vinculação de interceptador . . . 51
2.27 Classe `UsuarioController`: definição de um ponto de interceptação . . . 51
2.28 Tipo de anotação para validação de dados . . . 52
2.29 Classe para validação de dados . . . 53
2.30 Classe `Aluno` com restrições de validação . . . 53
2.31 Exemplo de validação usando a *Bean Validation API* . . . 54

3.1 Classe `AlunoController`: Métodos com e sem anotação `@Crud` . . . 64
3.2 Implementação da Classe `MethodCrudExecuted` . . . 66
3.3 Implementação simplificada da classe `SaveObject` . . . 67
3.4 Classe `FactoryMetaModel` . . . 69

3.5 Classe `RepositoryMetaModels` 69
3.6 Classe `BasicMetaModel`: A implementação de `MetaModel` 70
3.7 Classe `BasicMetaField` simplificada 71
3.8 Classe `BasicContext` simplificada 74
3.9 Classe `ShowAddForm` simplificada 75
3.10 *Template* da UI de inserção 76
3.11 Apresentando a UI na camada de visão 77
3.12 Classe de domínio `Medico` 80
3.13 Classe de controle `Medico` 80
3.14 Classe de domínio `Noticia` 82
3.15 Classe *Controller* `Noticia` 82
3.16 Classe para validação de dados 84
3.17 Classe domínio `Paciente` com associações 86
3.18 Classe Domínio `Medico` com associações 87
3.19 Classe domínio `Cirurgia` com associações 87
3.20 Usando configurações para gerar UIs baseadas em contexto de uso 91
3.21 Classe de domínio `Cirurgia` 93
3.22 *Controller* com Interceptador 94

Abreviaturas e Siglas

API	–	Application Programming Interface
CDI	–	Contexts and Dependency Injection
CGI	–	Common Gateway Interface
CIM	–	Computation Independent Model
CPF	–	Cadastro de Pessoa Física
CRF	–	Cameleon Reference Framework
CRUD	–	Create Read Update Delete
CSS	–	Cascading Style Sheets
CSV	–	Comma Separated Values
DAO	–	Data Access Object
DDR	–	Double Data Rate
DI	–	Dependency Injection
DSL	–	Domain-Specific Language
ECC	–	Error-Correcting Code
GUI	–	Graphical User Interface
HTML	–	HyperText Markup Language
HTTP	–	HyperText Transfer Protocol
IDE	–	Integrated Development Environment
IoC	–	Inversion of Control
ISBN	–	International Standard Book Number
JDBC	–	Java Database Connectivity
JPA	–	Java Persistence API
JSF	–	JavaServer Faces
JSP	–	JavaServer Pages
JSR	–	Java Specification Request
LASER	–	Laboratório de Segurança e Engenharia de Redes

MBUID	–	Model-Based User Interface Development
MBUIDE	–	Model-Based User Interface Development Environment
MDA	–	Model-Driven Architecture
MDD	–	Model-Driven Development
MDE	–	Model-Driven Engineering
MDSD	–	Model-Driven Software Development
MOF	–	Meta Object Facility
MVC	–	Model View Controller
OCL	–	Object Constraint Language
OMG	–	Object Management Group
ORM	–	Object-Relational Mapping
PDF	–	Portable Document Format
PIM	–	Platform Independent Model
PSM	–	Platform Specific Model
RAM	–	Random Access Memory
REST	–	Representational State Transfer
SGDB	–	Sistema de Gerenciamento de Banco de Dados
SOA	–	Service-Oriented Architecture
SQL	–	Structured Query Language
SUS	–	System Usability Scale
UI	–	User Interface
UML	–	Unified Modeling Language
UNIFEI	–	Universidade Federal de Itajubá
URL	–	Uniform Resource Locator
WYSIWYG	–	What You See Is What You Get
XMI	–	XML Metadata Interchange
XML	–	eXtensible Markup Language

Sumário

Lista de Figuras **x**

Lista de Tabelas **xi**

Lista de Códigos **xiii**

Abreviaturas e Siglas **xiv**

1 Introdução **1**
1.1 Motivação . . . 3
1.2 Objetivos . . . 4
1.3 Organização do trabalho . . . 4

2 Fundamentação Teórica **6**
2.1 Model-Driven Engineering (MDE) . . . 6
2.1.1 Model-Driven Architecture (MDA) . . . 7
2.1.1.1 Modelo e Metamodelo . . . 8
2.1.1.2 Ponto de Vista . . . 10
2.1.1.3 Plataformas . . . 11
2.1.1.4 Transformação . . . 12
2.1.1.5 Marcas . . . 13
2.1.2 Model-Driven Development (MDD) . . . 14
2.1.3 Model-Based User Interface Development (MBUID) . . . 16
2.1.3.1 Modelo de Domínio . . . 17
2.1.3.2 Modelo de Tarefas . . . 18
2.1.3.3 Modelo de Apresentação . . . 19
2.1.3.4 Modelo de Diálogo . . . 19
2.1.3.5 CAMELEON Reference Framework (CRF) . . . 19
2.1.3.6 Contexto de Uso . . . 20
2.1.4 Scaffolding . . . 21
2.1.4.1 Tipos de Scaffolding . . . 22
2.1.4.2 Scaffolding com templates . . . 24
2.1.4.3 Scaffolding estático e dinâmico no framework Grails . . . 26
2.2 Metaprogramação . . . 30

2.2.1 Metaprogramação em Java . . . 33
2.2.1.1 Java Reflections API . . . 33
2.2.1.2 Java Annotations API . . . 37
2.3 Desenvolvimento Web em Java . . . 40
2.3.1 VRaptor . . . 41
2.3.2 Contexts and Dependency Injection . . . 42
2.3.2.1 Mecanismo de Injeção de Dependências . . . 42
2.3.2.2 Modelo de notificação de eventos . . . 46
2.3.2.3 Interceptadores . . . 49
2.3.3 Bean Validation API . . . 51
2.4 Trabalhos Similares . . . 55
2.5 Considerações Finais . . . 58

3 O framework ObCrud 60
3.1 Arquitetura lógica . . . 60
3.1.1 Rotas (routes) . . . 64
3.1.2 Processador de requisição (processor) . . . 65
3.1.3 Repositório (repository) . . . 67
3.1.4 Validador de objetos (validator) . . . 68
3.1.5 Interceptadores (interceptor) . . . 68
3.1.6 Uploads . . . 69
3.1.7 Metamodelo (meta) . . . 69
3.1.7.1 Metadados (annotations) . . . 72
3.1.8 Template Engine (engine) . . . 73
3.1.9 Formulários (form) . . . 74
3.1.9.1 Widget . . . 78
3.2 Processo de desenvolvimento . . . 78
3.2.1 Geração de UIs básicas . . . 80
3.2.2 Personalização das UIs com anotações do ObCrud . . . 82
3.2.3 Validação de dados . . . 84
3.2.4 Geração de UIs com associações entre classes de domínio . . . 85
3.2.5 Geração de UIs baseadas em contexto de uso . . . 90
3.2.6 Aplicação do modelo de tarefas . . . 92
3.2.6.1 Adicionando tarefas . . . 92
3.2.6.2 Adicionando Interceptadores . . . 94
3.3 Considerações finais . . . 95

4 Experimentos e Resultados 96
4.1 Perfil dos participantes e do ambiente de testes . . . 96
4.2 Metodologia . . . 97

4.2.1 Fase de Treinamento 98
4.2.2 Fase de Execução 100
4.2.3 Fase de Avaliação 101
4.2.3.1 Avaliação heurística da UI 102
4.2.3.2 Comparação da usabilidade entre as UIs 103
4.2.3.3 Avaliação da usabilidade dos frameworks 103
4.3 Análise e resultados 103
4.3.1 Avaliação da produtividade 104
4.3.2 Avaliação heurística da UI 106
4.3.3 Comparação da usabilidade da UI 107
4.3.4 Avaliação de usabilidade dos frameworks 108
4.4 Considerações finais 110

5 Conclusão **111**
5.1 Contribuições 112
5.2 Dificuldades Encontradas 113
5.3 Trabalhos Futuros 115

Referências Bibliográficas **117**

Apêndice A – Roteiro Experimental **124**

Apêndice B – Dados Experimentais **137**

Apêndice C – Formulário de Registro de Tempo **139**

Apêndice D – Comparação das UIs CRUD **140**

Apêndice E – Avaliação heurística de usabilidade das UIs CRUD **141**

Apêndice F – Questionário SUS **143**

Apêndice G – Questionário do perfil dos participantes **145**

Capítulo 1

Introdução

Nos últimos anos a sociedade pôde experimentar o surgimento de inúmeras tecnologias, mas sem dúvida, uma das maiores invenções, foi a Internet. De acordo com Leiner et al. (2009), a Internet é um mecanismo que possibilita a disseminação de informações, além de permitir a colaboração e a interação entre indivíduos e seus computadores, independentemente da localização geográfica.

A Internet revolucionou as relações e a forma de vida da sociedade, possibilitou novas formas de comunicação, de estudo e trabalho, além de criar novas oportunidades de negócios. Isso somente foi possível, além de outros fatores, com a invenção e a popularização dos microcomputadores. Os primeiros computadores, os *mainframes*, ocupavam muito espaço, consumiam muita energia, possuíam baixo poder de processamento e eram difíceis de operar e configurar, eram utilizados apenas por órgãos governamentais e por universidades.

Para que a Internet evoluísse, nos moldes de hoje, os computadores tiveram que reduzir de preço e tamanho, tiveram que ampliar a capacidade de armazenamento e processamento e o mais importante, tiveram que se tornar de fácil utilização. Neste quesito, o surgimento das Interfaces Gráficas do Usuário (Graphical User Interface - GUI) contribuiu, e muito, para popularização dos computadores. A Interface do Usuário (User Interface - UI) de um programa de computador é a parte que permite que o usuário realize a entrada de dados e visualize a saída gerada pela aplicação (MYERS, 1995).

As UIs são tão fundamentais que Walker (1990) redefiniu as gerações dos computadores sob o ponto de vista da interação com o usuário. Para Walker, a Primeira Geração de interfaces era constituída de painéis com botões, mostradores e funcionamento dedicado. A Segunda Geração era formada por lotes de cartões de dados perfurados e entrada de dados remota. A Terceira Geração era de teletipo (linha de comando) de tempo compartilhado. A Quarta Geração era composta por sistemas de menus. Por fim, a Quinta Geração era constituída de controles gráficos e janelas. Nos dias atuais, as UIs continuam a evoluir, contando agora com recursos de toque, voz e gestos.

As UIs são consideradas um dos artefatos mais importantes no desenvolvimento de *software* (MYERS, 1995; MYERS et al., 2000; SILVA; PATON, 2003), alguns usuários acreditam que

a interface é o próprio sistema. Nesse sentido, a usabilidade de uma UI é imprescindível, visto que, o sucesso ou fracasso de uma aplicação está intimamente relacionada à quanto a interface é amigável, de fácil aprendizagem e utilização, sendo que essas características definem o nível de usabilidade de uma interface (NIELSEN, 1990; THIMBLEBY, 1990; MOLINA et al., 2012).

Dada a importância das UIs, não poderia ser diferente, seu desenvolvimento é uma atividade complexa, difícil de implementar, testar e modificar (MYERS, 1995) que contribui para a baixa produtividade no desenvolvimento de *software* (SILVA, 2010). Conforme as UIs se tornam mais fáceis de utilizar, também se tornam mais difíceis de criar (MYERS, 1994). Myers e Rosson (1992) apresentam que aproximadamente 48% do código-fonte, 50% do tempo de implementação e 37% do tempo de manutenção são destinados à criação de UI. Isso se deve ao fato de que as modificações nas UIs são mais frequentes do que em qualquer outro artefato de *software* (SHNEIDERMAN; PLAISANT, 1987). De acordo com Meixner et al. (2011), essas estimativas de tempos ainda são válidas, devido a interatividade dos novos sistemas, o aumento de requisitos nos *softwares*, a variedade de linguagens e tipos de dispositivos.

Com o objetivo de reduzir a complexidade e aumentar a produtividade no desenvolvimento de UIs, diversas pesquisas e ferramentas foram desenvolvidas. Um dos tipos de ferramentas, desenvolvidos para este fim e amplamente aceita pelo mercado, são os construtores de interface (*Interface Builders*). Estas ferramentas utilizam editores visuais que possibilitam que o projetista selecione um *widget* e o coloque na tela, utilizando o ponteiro do *mouse*. Esses editores também possibilitam que o *widget* seja posicionado, alinhado e que sua aparência seja modificada, por meio de uma janela de propriedades. Essas ferramentas são de fácil uso e aumentam a produtividade do desenvolvimento, no entanto, não fornecem orientações sobre construir boas UIs, uma vez que fornecem total liberdade aos projetistas (MYERS, 1995).

Quando se trata de UIs altamente interativas para aplicações orientadas a negócio, as abordagens dirigidas por modelo apresentam certas vantagens. De acordo com Vanderdonckt (1994), aplicações orientadas a negócio são aquelas que manipulam banco de dados, encontradas em sistemas de informação, como as de automação de escritório e as de controle financeiro. Além disso, ele define o termo "altamente interativo" como os mecanismos avançados, que essas interfaces fornecem aos usuários, para navegação, manipulação e visualização de dados.

Uma prática já incorporada no ciclo de desenvolvimento de *software* é a modelagem de sistemas. De acordo com (BEZERRA, 2015), a modelagem de sistemas possibilita diversas vantagens como o gerenciamento da complexidade, a comunicação entre as pessoas envolvidas, a redução dos custos de desenvolvimento e a previsão do comportamento futuro do sistema. No que tange ao desenvolvimento de UIs, os modelos, comumente desenvol-

vidos, possuem diversas informações que estão atreladas às UIs, como por exemplo os Modelos de Domínio que possuem atributos, com tipos de dados e regras de validação, que em uma UI correspondem a determinados tipos de *widgets* (PUERTA; EISENSTEIN, 1999).

Abordagens dirigidas por modelo concentram esforços na especificação de modelos, que posteriormente são transformados automaticamente em artefatos de *software*. Artefatos de *software* podem ser desde uma documentação até o próprio sistema completo. Na área das UIs diversas pesquisas têm sido realizadas, objetivando a construção de UIs baseadas em modelos (MOLINA et al., 2012; SILVA, 2010; MRACK et al., 2006; PUERTA, 1996). As ferramentas de geração automática de UIs, baseadas nas abordagens dirigidas por modelo, ao contrário dos *Interfaces Builders*, que não fornecem orientações sobre construir boas UIs, empregam heurísticas, padrões e modelos de projetos bem-sucedidos para gerar UIs com boa usabilidade.

O paradigma da Engenharia Dirigida por Modelos (*Model-Driven Engineering* – MDE), que cobre as diversas abordagens dirigidas por modelos, tem ganhado *status* de nova geração de linguagem de programação (DAVID, 2009). Isso porque ele possibilita que modelos, mais abstratos que as linguagens de programação atuais, sejam transformados em artefatos reais, almejando ainda que essas transformações possam criar sistemas completos. Por ser uma área ainda em expansão, existe certa dúvida sobre essa possibilidade. Assim como foi no surgimento do primeiro compilador, muitos desenvolvedores pensaram que ele seria incapaz de gerar código de máquina, tão bem quanto aos criados por eles próprios.

Este trabalho visa explorar e aplicar os conceitos das abordagens dirigidas por modelo na construção de uma ferramenta para geração automática de UIs para aplicações orientadas a negócio, com isso espera-se reduzir a complexidade do desenvolvimento de UIs e ao mesmo tempo aumentar a produtividade e a qualidade das UIs geradas.

1.1 Motivação

Ao longo dos últimos anos, diversas pesquisas foram realizadas para construir ambientes de desenvolvimento de UIs utilizando modelos, sendo a principal abordagem utilizada o Desenvolvimento de Interfaces do Usuário Baseadas em Modelo (*Model-Based User Interface Development* - MBUID) (SILVA; PATON, 2003). Outras abordagens como a Arquitetura Dirigida por Modelos (*Model-Driven Architecture* - MDA) e o Desenvolvimento Dirigido por Modelos (*Model-Driven Development* - MDD) possuem conceitos semelhantes ao MBUID e são frequentemente utilizados para o desenvolvimento de outros artefatos de *software*, podendo incluir também as UIs.

Essas abordagens da MDE vêm ganhando espaço nos últimos anos, mas para a consolidação deste paradigma é necessário o surgimento de novas ferramentas de automatização

de modelos, bem como o amadurecimento das já existentes, de forma que elas possuam boa usabilidade, alta curva de aprendizagem e alta produtividade (FOWLER, 2014).

Tendo vista que as UIs são um dos elementos mais importantes de um sistema, que seu desenvolvimento é uma atividade complexa que exige muito tempo e mão de obra para ser realizada, que as abordagens baseadas em modelo apresentam vantagens para a construção de UIs orientadas a negócios, fica evidente a necessidade do estudo do tema, e da proposição de uma ferramenta que possa atender as expectativas da área.

1.2 Objetivos

Este trabalho de mestrado propõe o projeto e a implementação do *framework ObCrud*, que possibilita a geração automática de interfaces gráficas do usuário, para sistemas orientados a negócio. Para geração das UIs, o *framework* emprega os conceitos das abordagens dirigidas por modelo, faz uso dos Modelos de Domínio, de Tarefa e de Apresentação e utiliza técnicas de Metaprogramação.

Durante a realização do trabalho foram identificados os principais pontos fortes e fracos das atuais ferramentas de geração automática de GUI, com isso foi proposto o *framework ObCrud*, que busca mitigar os problemas encontrados em outras ferramentas, ao mesmo tempo que busca manter as características de qualidade presentes nelas.

O *framework* utiliza como entrada os Modelos de Domínio, de Tarefas e de Apresentação, para gerar automaticamente, em tempo de execução, as GUIs. Por meio da abordagem *Model-Driven Development* (MDD) as interfaces são evoluídas e refinadas, conforme os modelos são construídos e modificados.

O principal objetivo desse trabalho é fornecer ao desenvolvedor uma ferramenta de fácil utilização e aprendizagem, que aumente a produtividade na construção de sistemas web e que construa automaticamente GUIs, mantendo elevado padrão de usabilidade. Os estudos realizados podem auxiliar na evolução de outras ferramentas disponíveis no mercado.

1.3 Organização do trabalho

Os demais Capítulos dessa Dissertação estão organizados da seguinte maneira.

- *Capítulo 2 - Fundamentação Teórica*: Apresenta um levantamento bibliográfico sobre os principais paradigmas e abordagens baseadas em modelos para o desenvolvimento de UIs. Este Capítulo apresenta ainda a tecnologia de metaprogramação

e metadados, além do *framework VRaptor* e das especificações Java, fundamentais para a construção do *framework* proposto neste trabalho. Ao final do Capítulo são apresentadas algumas das ferramentas desenvolvidas ao longo dos últimos anos.

- *Capítulo 3 - O framework ObCrud*: Este é o Capítulo central da Dissertação, nele é apresentada a arquitetura do *framework ObCrud* e cada uma de suas características. Ao final é realizada uma comparação do *framework* com outras ferramentas apresentadas na literatura.

- *Capítulo 4 - Experimentos e Resultados*: explica a metodologia dos experimentos realizados, apresenta os dados e resultados obtidos.

- *Capítulo 5 - Conclusões*: apresenta as conclusões obtidas da análise dos resultados, as contribuições do trabalho, as dificuldades encontradas e sugere trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este Capítulo apresenta a fundamentação teórica necessária para o desenvolvimento deste trabalho. A Seção 2.1 apresenta a Engenharia Orientada por Modelos (*Model-Driven Engineering* - MDE) e suas abordagens orientadas a desenvolvimento. O MDE é um paradigma que considera que os modelos são mais do que artefatos de documentação, eles são utilizados em diversas disciplinas da engenharia de *software*, como na análise, projeto, implementação e testes. Nas abordagens orientadas a desenvolvimento, os modelos abstratos são transformados em implementações concretas. A Seção 2.2 apresenta a técnica de metaprogramação, considerada uma evolução das linguagens de programação. Esta técnica é comumente utilizada para criar mecanismos de transformação/automatização de modelos abstratos, da abordagem MDE, em sistemas concretos. A Seção 2.3 discorre sobre a evolução do desenvolvimento de sistemas web utilizando a linguagem Java e suas subseções apresentam as tecnologias empregadas no desenvolvimento do *framework ObCrud*. Finalmente, a Seção 2.4 apresenta trabalhos similares, que também utilizam as abordagens baseadas em MDE para o desenvolvimento de Interfaces Gráficas. As ideias básicas de cada trabalho, suas vantagens e desvantagens foram consideradas na implementação deste trabalho.

2.1 Model-Driven Engineering (MDE)

O *Model-Driven Engineering* (MDE) é um paradigma de engenharia de *software*, no qual os modelos abstratos são responsáveis por conduzir todo o processo de desenvolvimento do *software*. Os modelos criados sofrem transformações sistemáticas, que permitem a criação e/ou a execução de sistemas de *softwares* (MOLINA et al., 2012).

No MDE os modelos são considerados artefatos centrais, no campo da Engenheira de *Software* (SILVA, 2015). Desta forma, eles têm a mesma importância que o próprio código fonte, isso porque, eles são os responsáveis por gerá-lo. Essa abordagem é interessante, visto que, muitos modelos são desenvolvidos nas fases iniciais do desenvolvimento, entretanto, na fase de implementação, as modificações realizadas em código são repassadas para os modelos.

Nas abordagens não dirigidas a modelos, os modelos são utilizados principalmente para documentar os sistemas. O problema é que, enquanto ocorre a implementação, os sistemas evoluem, mas os modelos não são atualizados. Isso faz com que muitos desenvolvedores acreditem que realizar a modelagem de sistemas seja uma perda de tempo (VÖLTER et al., 2013). A discrepância entre os modelos e os códigos produzidos dificultam a manutenção e a evolução dos sistemas.

Um dos principais objetivos do MDE é aumentar a abstração no desenvolvimento dos sistemas, retirando os detalhes referentes às plataformas de execução. Isso permite, além de outras coisas, focar na análise do problema. Após a construção dos modelos, que atendam às necessidades dos *"stakeholders"*, diferentes transformações são realizadas, levando em consideração as diversas plataformas e tecnologias, empregadas no desenvolvimento do sistema. Assim, diferentes transformações nos modelos geram novos sistemas, para plataformas específicas.

As tecnologias evoluem rapidamente, e muitas vezes é necessário portar sistemas inteiros para outras tecnologias para atender novos requisitos dos clientes. Essa portabilidade nas abordagens tradicionais exige o retrabalho das fases de análise, projeto e principalmente da implementação. Para criar sistemas para plataformas e tecnologias diferentes utilizando o MDE, basta possuir os transformadores específicos para as plataformas de destino.

O paradigma MDE se desdobra em diversas abordagens. As principais abordagens e conceitos envolvidos no paradigma MDE e em suas respectivas abordagens são apresentados nas subseções seguintes.

2.1.1 Model-Driven Architecture (MDA)

O *Model-Driven Architecture* (MDA) é uma abordagem para o desenvolvimento de sistemas, proposta pela *Object Management Group* (OMG), no ano de 2000. A abordagem é orientada a modelos, pois os coloca no centro do desenvolvimento e fornece mecanismos para que eles direcionem todas as fases de desenvolvimento do sistema (MILLER; MUKERJI, 2003). Os principais objetivos do MDA são permitir a redução da complexidade e possibilitar a interoperabilidade, a portabilidade e a independência de plataforma de *software* (MOLINA et al., 2012; VÖLTER et al., 2013).

O termo *Architecture* em MDA possui dois significados. No primeiro, *Architecture* é visto como um conjunto de modelos com a finalidade de representar um sistema de interesse. No segundo, *Architecture* representa toda atividade e/ou prática de criação dos conjuntos de modelos, que representam o sistema (SIEGEL, 2014). O MDA defende o uso da modelagem e a utilização de diversos padrões de propriedade da OMG, tais como: a *Unified Modeling Language* (UML); a *Object Constraint Language* (OCL); o *Meta Object Facility* (MOF) e o *XML*

Metadata Interchange (XMI). Com isso, espera-se que a construção e melhoria dos sistemas sejam menos onerosa e arriscada (BÉZIVIN, 2005; SIEGEL, 2014).

2.1.1.1 Modelo e Metamodelo

Os modelos são as peças fundamentais para o paradigma MDE e para suas abordagens. Em MDA, o modelo é um conjunto de informações que descreve e representa um sistema, sobre determinado ponto de vista e atende a um conjunto específico de preocupações, sendo frequentemente representado por uma combinação de desenhos e textos (MILLER; MUKERJI, 2003; SIEGEL, 2014).

No MDA os modelos são produzidos utilizando a Linguagem de Modelagem Unificada (UML). A UML é uma linguagem visual, logo ela define elementos gráficos que possuem sintaxe e semântica. A sintaxe corresponde a forma predeterminada de se desenhar um elemento, enquanto que a semântica corresponde ao significado do elemento e de como ele deve ser utilizado (BEZERRA, 2015).

A UML é formada por uma família de notações de modelagem unificadas e por um metamodelo que cobre diversos aspectos da modelagem de negócio e de sistemas. A UML possui notações para: modelagens de classes orientadas a objeto, processos e atividades, Casos de Uso, Máquinas de Estado, Sequenciamento e trocas de mensagens, modelagem de componentes, estruturas compostas e comunicação e colaboração (SIEGEL, 2014).

O metamodelo da UML é responsável por descrever todos os conceitos que podem ser utilizados na linguagem de modelagem. O termo conceito refere-se às partes ou elementos que compõem a linguagem de modelagem, por exemplo, classe é um conceito da UML, construtores e métodos são conceitos da linguagem Java, chave primária e estrangeira são conceitos da linguagem SQL. Esses conceitos por vezes são denominados de *meta-Classes* ou *metaTypes*, sendo que o conjunto de todos eles constituem o metamodelo de uma linguagem (WARMER; KLEPPE, 2003). A Figura 2.1 apresenta um metamodelo simplificado do diagrama de classes da UML. O metamodelo apresenta que tudo em um modelo é um `ModelElement`, assim ele é o elemento mais abstrato e é a superclasse de todas as outras metaclasses. O elemento `Classifier` é a superclasse abstrata de todos os tipos presentes em um modelo, suas especializações são `Class`, `DataType` e `Interface`. O `Classifier` possui recursos (`Feature`) que são a superclasse de `Atributte`, `Operation` e `AssociationEnd`. Cada recurso possui um tipo, no caso dos `Attributes`, o tipo é o próprio tipo do atributo, de `Operation` o tipo é o tipo de retorno e em `Association` o tipo é a classe que o dono do final da Associação possui. Os recursos possuem uma visibilidade (público, privado e protegido). O elemento `Attribute` corresponde aos atributos ou características de uma classe, o elemento `Operation` corresponde aos métodos e `Parameter` aos parâmetros dos métodos, por fim `Association` corresponde a associação com outro elemento `AssociationEnd`.

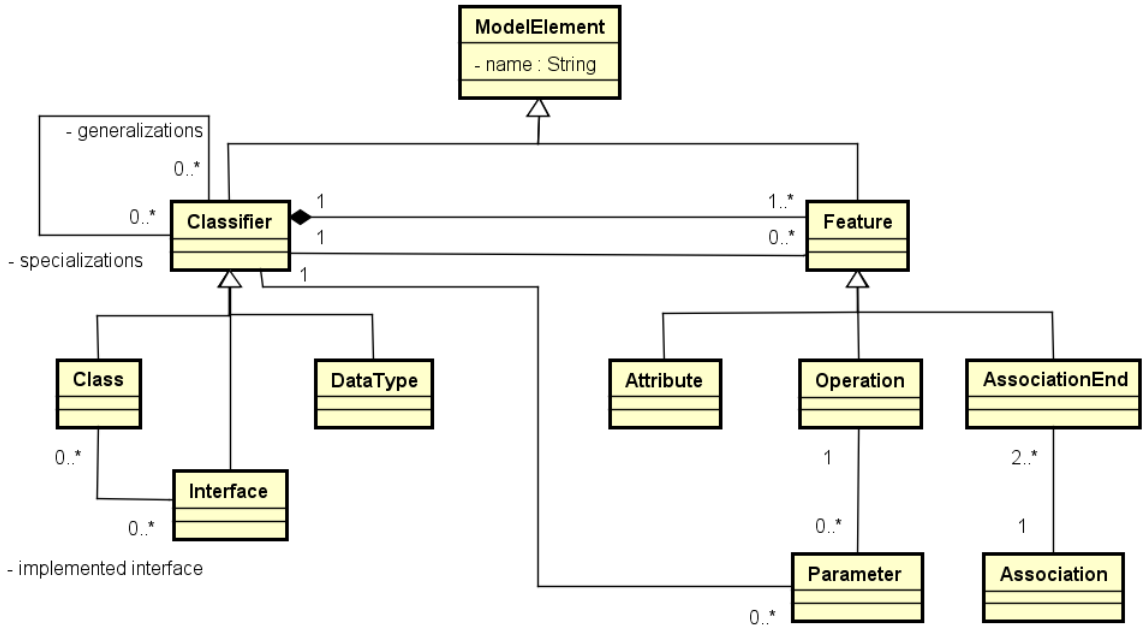

Figura 2.1: Metamodelo simplificado do diagrama de classes da UML.
Fonte: Adaptado de Warmer e Kleppe (2003)

Um metamodelo é um modelo utilizado para expressar outro modelo e é responsável por garantir que a linguagem de modelagem, que o utiliza, seja estruturada e possua notações, sintaxe, semântica e as regras de integridade (SIEGEL, 2014). Desta forma, o metamodelo possibilita a validação dos modelos gerados por uma linguagem de modelagem. Para exemplificar, a Figura 2.2 apresenta a relação entre metamodelo e modelo, onde cada elemento de um modelo é um exemplar ou instância de um conceito presente em uma linguagem de modelagem, assim como cada objeto é uma instância de uma Classe.

O elemento `Aluno`, do modelo, é uma instância da *metaClass* `UML Class`, definida no metamodelo. Os atributos `ra` e `nome` dos tipos `long` e `String` são instâncias da *metaClass* `UML Attribute`, como os modelos representam um sistema, os objetos que existem no domínio do problema também são instâncias da classe `Aluno`.

É importante salientar que modelo e diagrama são coisas distintas. Os modelos representam e descrevem sistemas, por outro lado, um diagrama ou conjunto de diagramas podem ser utilizados para descrever visualmente um modelo. A diversidade de notações e diagramas da UML, bem como seu formalismo, definido no seu metamodelo, possibilitam realizar a modelagem sob diferentes pontos de vista, bem como realizar a transformação de modelos em outros artefatos de *software*, como por exemplo, o código fonte.

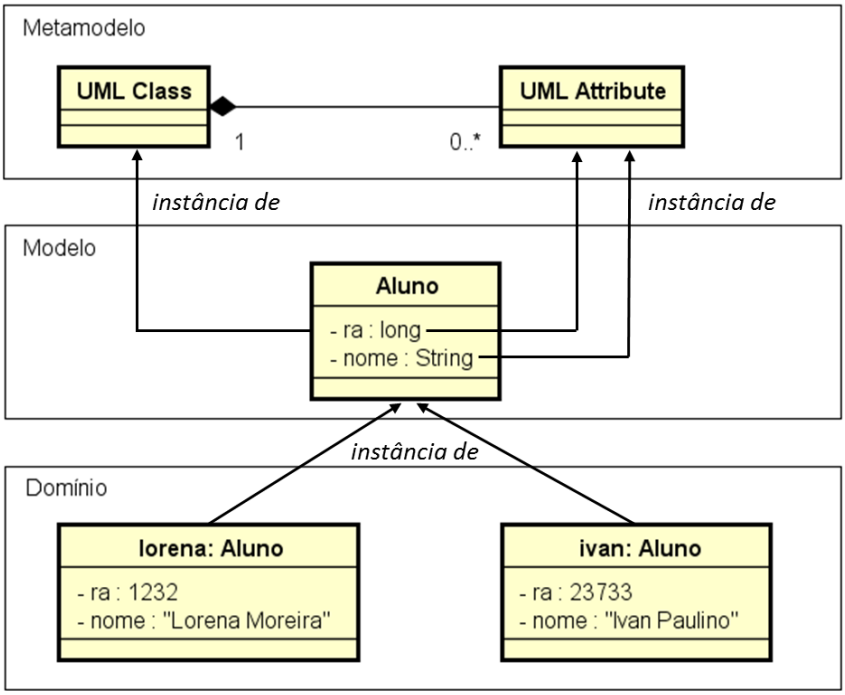

Figura 2.2: Relação entre elementos do metamodelo, modelo e domínio

2.1.1.2 Ponto de Vista

Em MDA, ponto de vista é uma técnica de abstração, que se concentra em partes específicas do sistema, selecionando conceitos importantes e suprimindo detalhes irrelevantes para a solução do problema. Isso permite que o modelo seja simplificado, reduzindo a complexidade do desenvolvimento (SIEGEL, 2014). O MDA especifica três pontos de vista ou níveis de abstração em um sistema, esses pontos de vistas são representados respectivamente por três tipos de modelo: o Modelo Independente de Computação (*Computation Independent Model* - CIM), o Modelo Independe de Plataforma (*Platform Independent Model* - PIM) e o Modelo Dependente de Plataforma (*Platform Specific Model* - PSM).

O CIM é o modelo construído a partir dos requisitos elicitados para o sistema, com objetivo de descrever como o sistema será utilizado ou irá operar, sem apresentar exatamente o que o sistema deve fazer (SIEGEL, 2014). Esse modelo possui alto nível de abstração, uma vez que, oculta as informações sobre como o sistema será automatizado e como os dados serão processados. Ele é utilizado para entender o problema e serve de vocabulário para a construção de outros modelos e também é conhecido como Modelo de Domínio ou Modelo de Negócios.

O PIM é o modelo construído para descrever, sob determinado ponto de vista, o que o sistema deve fazer, com base no CIM. O PIM é um modelo menos abstrato que o CIM, pois descreve computacionalmente o que o sistema deverá realizar, no entanto, o PIM abstrai as informações sobre a plataforma onde o modelo será implementado.

O PSM é um modelo construído a partir do PIM, ele descreve a solução de um problema sobre determinado ponto de vista, igualmente ao PIM, contudo, neste modelo, é considerado a plataforma e as tecnologias que serão utilizadas para construir o sistema. Dessa forma, o PSM possuir menor abstração que o PIM. A Figura 2.3 apresenta um exemplo dos 3 modelos.

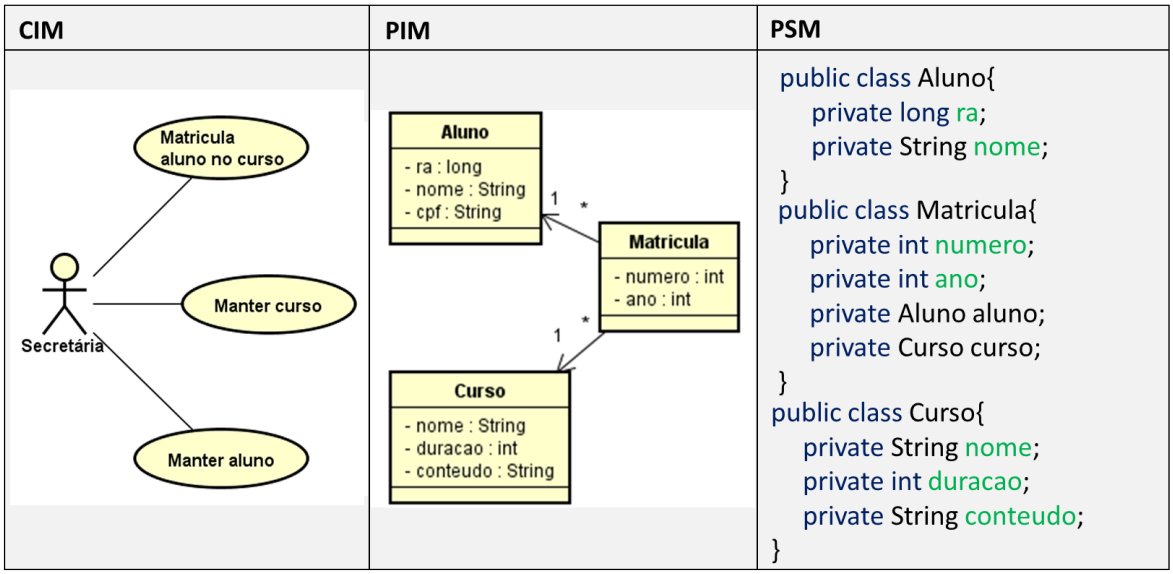

Figura 2.3: Exemplos dos modelos CIM, PIM e PSM

2.1.1.3 Plataformas

Em MDA, plataforma é um conjunto de recursos sobre o qual um sistema é construído. Esse conjunto de recursos são utilizados para construir aplicações, sendo que a plataforma somada as aplicações constitui o sistema. Para MDA existe diversos tipos e níveis de plataformas, como por exemplo: a plataforma AJAX, que é composta de outras plataformas, e que permite a construção de sistemas web interativos; a plataforma VRaptor que permite a construção de sistemas Java Web e aplicações orientadas a serviço SOA, utilizando o padrão *Model-View-Controller* (MVC); a plataforma JAVA, que permite a implementação de aplicativos, a plataforma do processador Intel i5, entre outras (SIEGEL, 2014).

Por existir diversos tipos e níveis de plataforma, em MDA, um PIM pode em determinados momentos ser considerado um PSM e vice e versa. No entanto, o PSM sempre será um modelo mais específico do que o PIM, no que tange a utilização de plataformas. Por exemplo, um modelo de classe criado por uma ferramenta de modelagem pode ser exportado para XMI, que é dependente do padrão XML para troca de dados entre aplicações. O XMI é escrito em XML, que é independente de linguagens de programação, como por exemplo as linguagens C e Java. A linguagem Java é independente do tipo de plataforma Web ou Desktop, assim como a linguagem C é independente do tipo de processador na qual será executada, como por exemplo intel ou arm. A plataforma Java

Web, por sua vez, é independente da especificação ou *framework* MVC utilizado, como por exemplo o JSF ou o VRaptor (SIEGEL, 2014). Pelo exposto, observa-se que, dependendo do ponto de vista, um modelo poder ser em determinados momentos PIM e em outros momentos PSM. A Figura 2.4 apresenta um exemplo dos níveis de plataforma, onde em determinado momento um modelo pode ser considerado PIM e em outro momento PSM, dependendo da plataforma selecionada.

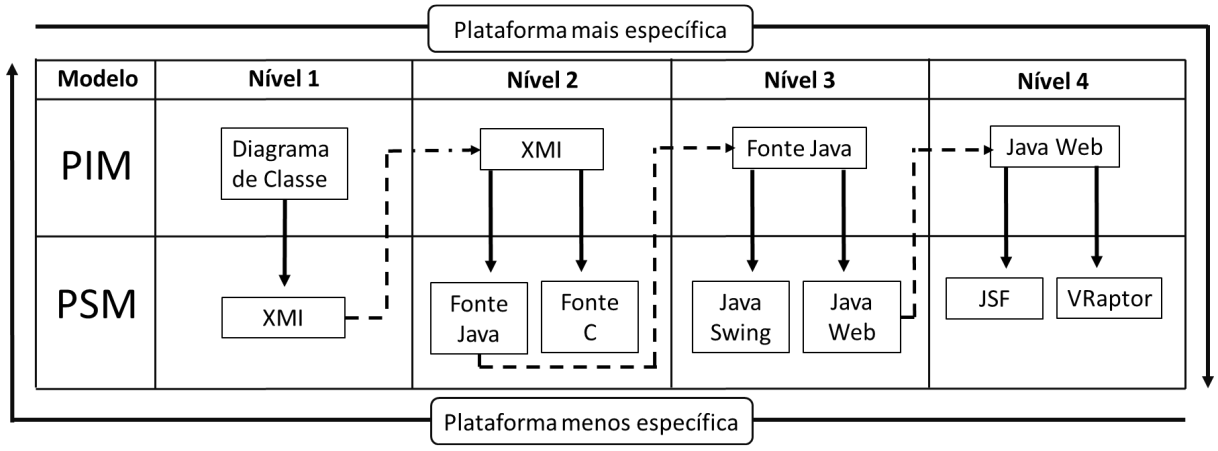

Figura 2.4: Exemplo de níveis de plataforma do MDA

2.1.1.4 Transformação

Um dos conceitos chave do MDA é a transformação. A transformação é processo responsável por converter um modelo em outro modelo do mesmo sistema. Basicamente o processo de transformação ocorre ao converter o PIM em PSM, para isso, são utilizados padrões e parâmetros que auxiliam na transformação dos modelos. Assim como existe diversos níveis de plataforma, sucessivas transformações de PIM em PSM podem construir um sistema automaticamente. A Figura 2.5 ilustra o processo de sucessivas transformações de PIM em PSM. As marcas correspodem a outros conjuntos de informações que são associados ao PIM para produzir um PSM.

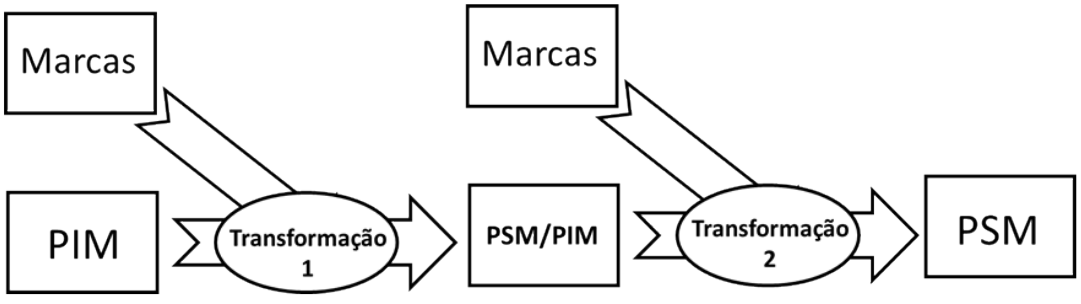

Figura 2.5: Sucessivas transformações de PIM em PSM
Fonte: Adaptada de Miller e Mukerji (2003)

De acordo com Siegel (2014), a automatização permite a transformação de modelos de alto nível para sistemas executáveis, esse processo de automação reduz o tempo, o custo e os riscos de produzir e manter os sistemas, ao mesmo tempo que melhora sua qualidade. Existe duas abordagens para automatizar modelos para sistemas executáveis:

- *Transformação:* Os modelos passam por um processo de transformação que produz artefatos ou modelos específicos da tecnologia alvo, como por exemplo, um modelo de domínio pode ser transformado em XML ou em código Java.
- *Execução:* Os modelos são executados diretamente por um mecanismo de execução de modelos, construído especificamente para a plataforma alvo, nesta abordagem o modelo é tratado como código-fonte interpretável.

2.1.1.5 Marcas

Dá-se o nome de marcas ao conjunto informações associadas ao PIM no processo de transformação para produzir um PSM, elas são utilizadas para marcar elementos do PIM e indicar como esses elementos devem ser transformados em PSM. As marcas podem ser definidas de diferentes formas, dentre elas:

- Tipos de um modelo, como por exemplo, classes e associações;
- Elementos de um modelo especificado em um metamodelo;
- Estereótipos de um perfil UML

Os perfis UML são o mecanismo padrão de extensão da linguagem UML. Por meio dos perfis é possível ampliar o vocabulário da UML, estendendo seu metamodelo. Os perfis UML contém conceitos da linguagem, definidos por meio de construções básicas da UML, como por exemplo: as Classes; as Associações; os estereótipos; os valores e as regras de modelagem. Os modelos criados, a partir da extensão do metamodelo, devem atender as regras de modelagem e podem utilizar os novos conceitos, definidos pelo perfil UML criado. Dessa forma, o perfil UML oferece uma notação concreta para determinar se um modelo é valido ou não (VÖLTER et al., 2013).

A possibilidade de extensão da UML é interessante, pois permite ampliar a possibilidade de modelar diversos sistemas e pontos de vista, levando em consideração os diversos requisitos para o sistema. Além disso, para que seja possível automatizar a geração de modelos para sistemas é necessário que os modelos sejam válidos, ou seja, que atenda as restrições imposta pelo metamodelo e ainda que possuam informações adicionais, as marcas. A Figura 2.6 ilustra a extensão do metamodelo, a modelagem produzida com a extensão e o exemplo de código gerado por meio da transformação.

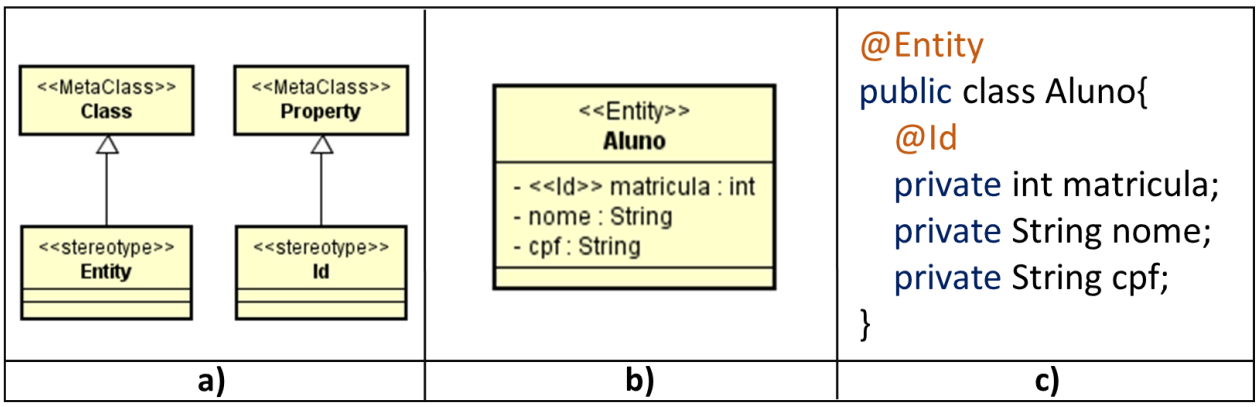

Figura 2.6: Extensão da UML com perfis e a criação de marcas. (a) Extensão do MetaModelo UML. (b) Uso de estereótipos no diagrama de classe. (c) Transformação do diagrama de classe e dos estereótipos em código Java

A Figura 2.6 (a) ilustra a extensão do metamodelo UML, por meio da criação de um perfil UML. Os estereótipos «`Entity`» e «`Id`» estendem respectivamente as *metaClasses* `Class` e `Property`. A Figura 2.6 (b) apresenta a criação de um diagrama de classe, onde a classe `Aluno` foi marcada com o estereótipo «`Entity`» e seu atributo `matricula` foi marcado com o estereótipo «`Id`». Caso a classe `Aluno` fosse marcada com o estereótipo «`Id`» e o atributo `matricula` fosse marcado com o estereótipo «`Entity`», o diagrama não seria válido, visto que o metamodelo exige que o estereótipo «`Entity`» seja aplicado somente a elementos `Class` e o estereótipo «`Id`» somente aplicado a elementos `Property`. Na Figura 2.6 (c) a classe Java `Aluno` foi criada, por meio de um processo de transformação, nela foram adicionadas as anotações `@Entity` e `@Id` que correspondem as marcas do perfil UML. A classe Java `Aluno` com suas anotações pode ser processada posteriormente por outra ferramenta de transformação gerando outros artefatos de *software*.

2.1.2 Model-Driven Development (MDD)

O *Model-Driven Development* (MDD) ou também conhecido com *Model-Driven Software Development* (MDSD) é uma abordagem de desenvolvimento de *software* voltada, principalmente, para as necessidades de análise, projeto e implementação (MELLOR et al., 2002; ATKINSON; KUHNE, 2003; SELIC, 2008; SILVA, 2015). O MDD possui uma abordagem semelhante ao MDA, os modelos também são considerados peças centrais no desenvolvimento de *software*, ou seja, são mais do que simples documentação. Os modelos têm a mesma importância do código-fonte de um sistema, visto que é possível, por meio da automatização e de regras de mapeamento, realizar a transformação dos modelos em sistemas (VÖLTER et al., 2013).

O MDD difere-se do MDA pelo fato de não estar associado a nenhum conjunto específico de padrões, por outro lado, o MDA foi concebido utilizando os padrões da OMG (FOWLER,

2017). Em MDD, o desenvolvedor é livre para utilizar qualquer padrão, modelo ou regras de transformação, inclusive os próprios padrões da OMG. Segundo David (2009), a principal contribuição do MDD é a flexibilidade para definir processos de desenvolvimento, já o MDA tende a ser mais restritivo, focando em linguagens de modelagem baseadas na UML (VÖLTER et al., 2013).

Mellor et al. (2003) definem o MDD como a simples noção de que é possível construir um modelo de um sistema, que então pode ser transformado em uma coisa real. O autor apresenta ainda que todos os desenvolvedores de linguagens de terceira geração são desenvolvedores MDD, uma vez que, o código-fonte produzido em linguagens, como Smalltalk, Java ou C# é considerado um modelo. Esse modelo, ao ser compilado, é transformado em outra linguagem, que por sua vez também é um modelo, que é passível de ser interpretado e executado por uma máquina virtual.

Por meio desta visão, Mellor et al. (2003) apresentam que, apesar dos modelos serem frequentemente equiparados a simples imagens (diagramas) retiradas do desenvolvimento de sistemas reais, eles são muito mais que isso. Völter et al. (2013) apresentam que em MDD os modelos gráficos são frequentemente utilizados, mas que os modelos textuais são uma opção igualmente viável, o importante é que os modelos sejam um conjunto coerente de elementos formais que descrevam algo, com um determinado propósito e sejam passíveis de serem analisados (MELLOR et al., 2003).

Outra diferença entre o MDD e o MDA repousa na importância dada aos objetivos de cada abordagem. Os objetivos das duas abordagens são semelhantes, no entanto o MDA atribui maior valor a reusabilidade dos modelos, a interoperabilidade e a portabilidade dos sistemas de *software* gerados, enquanto que o MDD, valoriza a produtividade do desenvolvimento, bem como a qualidade do *software* produzido (VÖLTER et al., 2013).

A Figura 2.7 apresenta as ideias básicas por trás do MDD. A Figura 2.7 (a) representa o código-fonte de uma aplicação, que ao ser analisado pode ser decomposto nas seguintes partes: *1) Código Genérico:* O código genérico representa as API e *frameworks* que podem ser utilizados para construir e executar diversos sistemas, esse código está associado a plataforma, na qual o sistema será executado; *2) Código Repetitivo:* É comum no desenvolvimento de sistemas haver código que se repete em diversos lugares, com poucas ou nenhuma alteração, esse código repetido recebe o nome de *boilerplate*. Por meio de técnicas de metaprogramação, esse tipo de código pode ser gerado automaticamente. *3) Código Individual:* É o código específico da aplicação, que resolve um determinado problema do domínio, conforme mostrado na Figura 2.7(b).

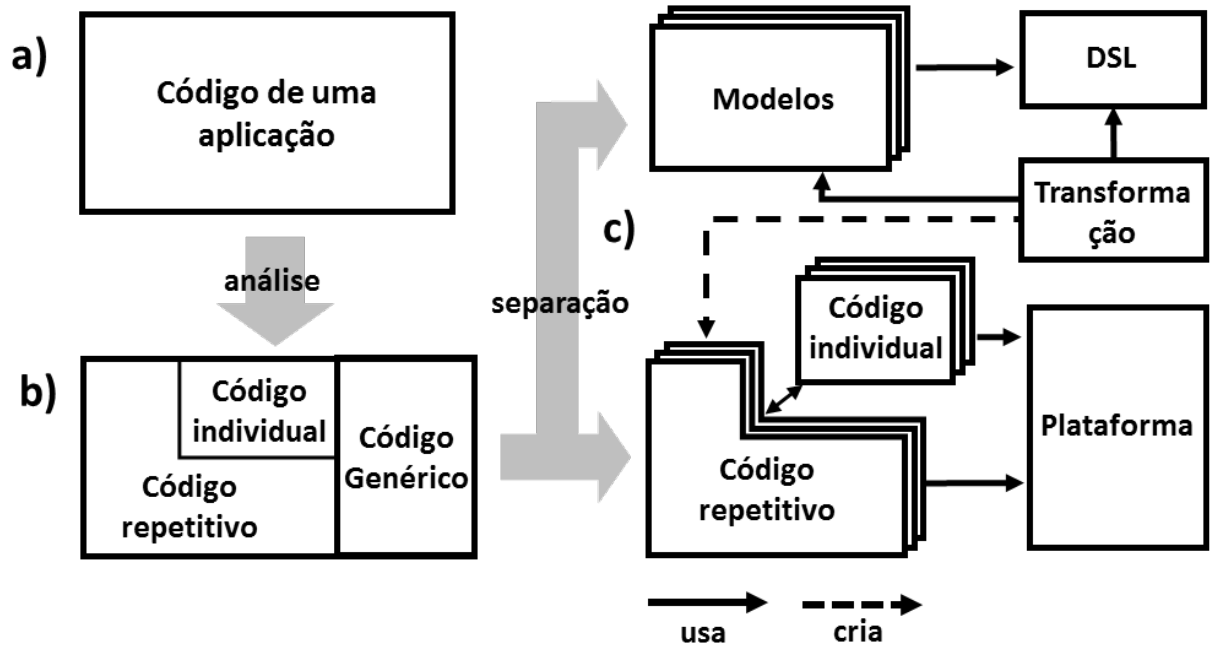

Figura 2.7: As ideias básicas por trás de *Model-Driven Software Development*
Fonte: Adaptada de Völter et al. (2013)

A Figura 2.7 (c) ilustra a proposta do MDD para a organização da estrutura de desenvolvimento do sistema. Os modelos que descrevem a aplicação são construídos utilizando uma linguagem de modelagem formal, podendo ser uma linguagem de domínio específico. Os modelos e o formalismo da linguagem são utilizados pelo transformador que irá produzir o código repetitivo e esquemático. O código repetitivo é associado ao código individual da aplicação, sendo que os dois utilizam os códigos genéricos das API e dos *frameworks*, que neste caso constituem a plataforma sobre a qual o sistema será executado.

2.1.3 Model-Based User Interface Development (MBUID)

O *Model-Based User Interface Development* (MBUID) é uma abordagem para o desenvolvimento de Interfaces do Usuário, (*User Interface - UI)*, baseada em modelos, que possibilita a redução de esforços empregados na criação de UI (MEIXNER et al., 2011).

O MBUID é a evolução do *User Interface Management Systems* (UIMS). De acordo com Meixner et al. (2011), as UIMS são ferramentas que aumentam a produtividade do programador, podendo ser comparada as linguagens de quarta geração, uma vez que, se concentram na especificação ao invés da codificação. A mudança de terminologia ocorreu com a evolução das linguagens de modelagem e de programação, principalmente com o surgimento do paradigma de orientação a objetos, que permitiram a criação de UI mais sofisticadas e complexas (SILVA, 2000; MEIXNER et al., 2011; SZEKELY, 1996).

De acordo com Schlungbaum (1996), as ferramentas baseadas no MBUID, também conhecidas como *Model-Based User Interface Development Environment* (MBUIDEs), devem atender a dois critérios:

1) Conter um modelo abstrato de alto nível, capaz de representar a interatividade do sistema a ser desenvolvido, podendo ser um modelo de tarefas, ou de domínio ou ambos.

2) Realizar algum tipo de transformação automática ou de execução do modelo em UI.

Diversos tipos de modelos têm sido empregados para criação de UI, sendo os principais: o Modelo de Domínio, o Modelo de Tarefas, o Modelo de Diálogo e o Modelo de Apresentação (MEIXNER et al., 2011; PUERTA; EISENSTEIN, 1999).

2.1.3.1 Modelo de Domínio

O modelo de domínio representa as classes conceituais do mundo real, geralmente na UML, esse modelo é representado por um diagrama de classes. Ele é considerado um dos mais importantes modelos no MBUID, visto que ele possui uma estreita relação com a Interface Final com o Usuário (*Final User Interface* - FUI). Os atributos com seus tipos e restrições, as associações, as heranças e as operações permitem determinar como será a UI para o objeto (PUERTA; EISENSTEIN, 1999). A Figura 2.8 apresenta um exemplo de modelo de domínio.

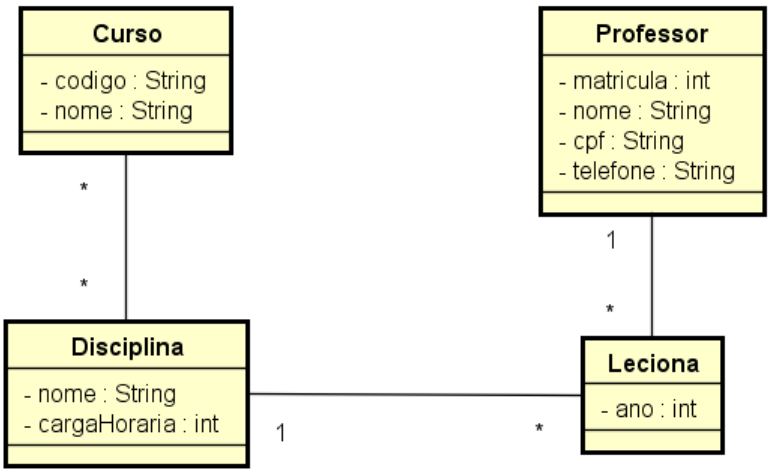

Figura 2.8: Modelo de Domínio

O modelo de domínio representa uma escola. Em uma escola existe diversos cursos, cada curso possui diversas disciplinas e uma disciplina pode estar na grade de vários cursos. Uma disciplina é lecionada, em um ano, por um professor e o professor pode lecionar diversar diciplinas no ano.

2.1.3.2 Modelo de Tarefas

O modelo de tarefas representa a relação das atividades que devem ser realizadas, para que o usuário realize seu objetivo no sistema. Esse modelo pode ser visto como uma árvore de tarefas e subtarefas. Cada tarefa, do modelo, possui uma ordem de execução e condições de pré e pós execução. O modelo de tarefa pode ser representado sobre diferentes níveis de abstração (PATERNÒ, 2004). A Figura 2.9 ilustra um exemplo de modelo de tarefas.

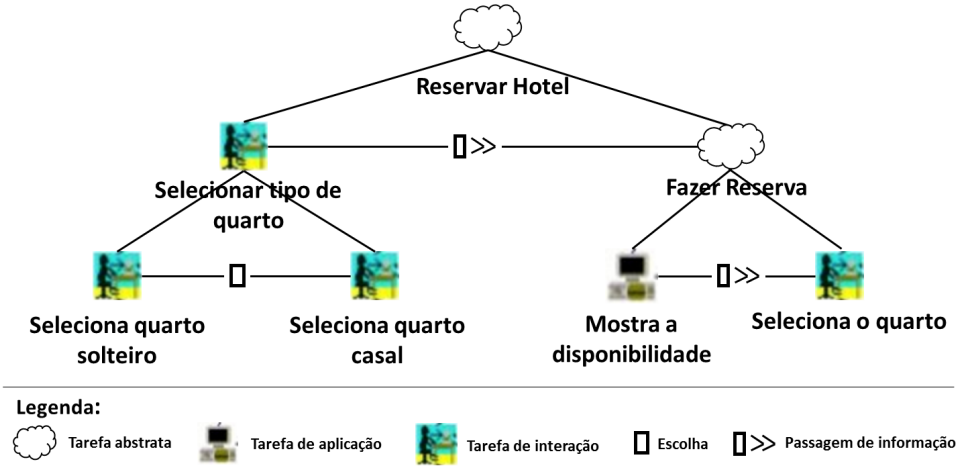

Figura 2.9: Modelo de Tarefas
Fonte: Adaptada de (PATERNÒ, 2004)

No primeiro nível hierárquico é definida a tarefa abstrata `Reservar Hotel`. Uma tarefa abstrata é uma tarefa complexa que pode ser decomposta em outras tarefas simples. No segundo nível a tarefa `Reservar Hotel` é decomposta em outras duas tarefas, sendo uma do tipo de interação `Selecionar tipo de quarto` e outra tarefa abstrata `Fazer reserva`. Tarefa de interação é aquela que o usuário realiza interagindo com o sistema. As tarefas `Selecionar tipo de quarto` e `Fazer reserva` estão conectadas com um operador de passagem de informação, que indica que a segunda tarefa `Fazer Reserva` não pode ser executada, enquanto a primeira tarefa, `Selecionar tipo de quarto`, não for realizada e que os dados gerados pela primeira tarefa devem servir de entrada para a segunda tarefa. A tarefa de interação `selecionar tipo de quarto`, pode ser decomposta em outras duas tarefas de interação `seleciona quarto de solteiro` e `seleciona quarto de casal`. As duas tarefas estão conectadas por um operador de escolha, que indica que apenas uma tarefa deve ser escolhida, após realizar a escolha outra tarefa não pode ser mais realizada. A tarefa abstrata `fazer reserva` pode ser decomposta em outras duas tarefas, uma de aplicação `mostra a disponibilidade` e outra de interação `seleciona quarto`. Uma tarefa de aplicação é aquela realizada pelo sistema, como por exemplo a exibição de dados. As tarefas de `mostrar a disponibilidade` e de `selecionar quarto` estão conectadas pelo operador de passagem de informação, assim o quarto só será selecionado após o sistema apresentar a disponibilidade do quarto.

2.1.3.3 Modelo de Apresentação

O modelo de apresentação descreve o conjunto de elementos, gráficos, gestuais e auditivos, suas características de *layout* e dependência visual, que a UI fornece ao usuário (PUERTA; EISENSTEIN, 1999). Schlungbaum (1996) apresenta que as aplicações são compostas de duas partes, uma estática que corresponde aos *widgets*, como botões, menus, campos de entrada, e a outra dinâmica que exibe os dados manipulados pela aplicação.

2.1.3.4 Modelo de Diálogo

O modelo de diálogo é uma descrição abstrata das ações e das possíveis relações temporais entre elas, que os usuários e o sistema executam na UI (PATERNO, 2012). Esse modelo é utilizado para descrever a comunicação entre o homem e a máquina, descrevendo, por exemplo, quando o usuário pode invocar comandos e realizar entradas de dados e quando o sistema pode apresentar as informações (SCHLUNGBAUM, 1996). O modelo de diálogo pode ser visto como uma derivação do modelo de tarefas, possibilitando a conexão das tarefas com os elementos de interação, ou seja, a conexão do modelo de tarefas com o modelo de apresentação.

2.1.3.5 CAMELEON Reference Framework (CRF)

Ao longo dos últimos anos diferentes estruturas foram criadas para definir processos e para identificar os elementos conceituais importantes para a abordagem MBUID (MEIXNER et al., 2011). Uma das estruturas que tem sido amplamente aceita pela comunidade de *Human-Computer Interaction* (HCI) é o *CAMELEON Reference Framework* (CRF) (CALVARY et al., 2002).

O CRF é um *framework* conceitual que serve de referência para definição de ferramentas de criação de UI, que suportam múltiplos contextos de uso, com base na abordagem baseada em modelos. Ele define uma abordagem abstrata para construir UI, ao invés de descrever formas, métodos e linguagens para realizar as diversas etapas de desenvolvimento (MEIXNER et al., 2011). A Figura 2.10 apresenta a simplificação do processo MBUID utilizando o *framework* CRF.

A estrutura do *framework* é composta por quatro camadas de abstração. A primeira camada é de *Tarefas e Conceitos* que descreve uma hierarquia de tarefas executadas em ordem temporal, a fim de atender os objetivos do usuário. A segunda camada é a *Interface Abstrata com o Usuário* (AUI) que expressa a interface através de objetos abstratos de interação (VANDERDONCKT; BODART, 1993). Essa camada é independente de plataforma, ou seja, de tecnologia (linguagem de programação) e de modalidade (gráfica, vocal, gestual, etc.).

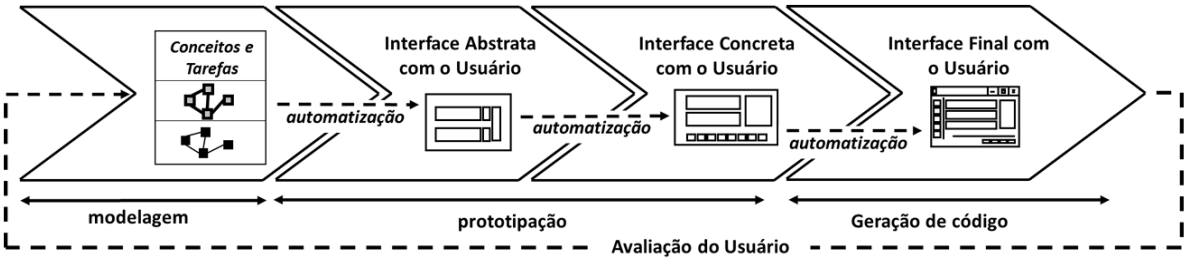

Figura 2.10: Simplificação do Processo MBUID com CRF
Fonte: Adaptado de (MOLINA et al., 2012)

A terceira camada é a *Interface Concreta com Usuário* (CUI) que define como a interface é percebida pelo usuário. Essa camada modela a interface para uma determinada plataforma, como por exemplo, TV, Smartphone, Tablet, PC, entre outras (VANDERDONCKT; BODART, 1993). Por fim, a última camada é a *Interface Final com o Usuário* (FUI) que representa a UI para um dispositivo específico, como um *Tablet Android* ou um *Browser*. A FUI é criada com qualquer linguagem de programação ou marcação, como por exemplo Java ou HTML, geralmente a FUI é criada automaticamente por ferramentas de autoria (CALVARY et al., 2002; MEIXNER et al., 2011). Os modelos são construídos nas camadas mais abstratas e por meio de ferramentas de transformação MBUIDE eles são sucessivamente refinados até a geração da UI.

Apesar de possuir 4 camadas não é necessário passar por todas as etapas de refinamento, pode-se iniciar o desenvolvimento em qualquer nível de abstração e realizar o refinamento ou abstração dependendo do projeto (MEIXNER et al., 2011).

2.1.3.6 Contexto de Uso

O objetivo das UIs é ser o meio, no qual usuários e sistemas se comunicam e interagem. No entanto, essa comunicação não é tão bem-sucedida quanto a comunicação entre os seres humanos. Dey (2001) apresenta que os seres humanos são bem-sucedidos em transmitir ideias uns aos outros e a responder a estímulos adequadamente e isso se deve a influência de diversos fatores, como por exemplo: a riqueza da língua que compartilham; a compreensão de como o mundo funciona e o entendimento das atividades cotidianas.

A comunicação entre os seres humanos é bem-sucedida, pois outras circunstâncias estão associadas a emissão da mensagem, como o lugar, o tempo e a cultura do emissor e do receptor. A essas circunstâncias dá-se o nome de contexto. Diversas pesquisas têm sido realizadas para melhorar a experiência do usuário no uso de sistemas, por meio de UIs baseadas em contexto de uso (DEY, 2001; PATERNO et al., 2009; CALVARY et al., 2002).

Para Dey (2001), contexto de uso, no desenvolvimento de UI, é "qualquer informação que pode ser usada para caracterizar a situação de uma entidade. Uma entidade é uma pessoa, lugar ou objeto que é considerado relevante para a interação entre um usuário e uma aplicação, incluindo o próprio usuário e as próprias aplicações ".

O *framework* CRF é sensível ao contexto de uso, assim a geração das UIs leva em conta os tipos de usuários e suas preferências, os diferentes tipos de plataformas e as propriedades do ambiente, onde a aplicação será executada, como por exemplo: o tamanho da tela, a disposição do *layout* (vertical ou horizontal), a luz ambiente e o nível de ruídos, entre outros. Os contextos de uso influenciam na forma como uma interface é apresentada ao usuário.

Para possibilitar a criação de UIs sensível ao contexto de uso, o CRF, na fase de projeto, cria diferentes projetos de UIs, para atender aos diferentes contextos de uso e dispositivos. Na fase de execução, quando há modificações no contexto de uso, traduções são realizadas entre as camadas de abstração do *framework*, para apresentar uma nova UI que atenda ao novo contexto de uso (CALVARY et al., 2002).

2.1.4 Scaffolding

Scaffolding foi o termo criado pelo *framework Rails* para descrever a geração de artefatos que compõem uma (UI), que permite realizar operações CRUD (ROCHER; BROWN, 2009). *Scaffolding*, em inglês cimbramento, na engenharia civil representa as estruturas temporárias, geralmente formadas por escoras e andaimes, que recebem a cargas da lajes, vigas e pilares e as transferem para o solo ou para o andar inferior, enquanto o concreto dessas peças passa pelo processo de curagem. O cimbramento ao mesmo tempo que suporta as cargas das estruturas que estão sendo construídas também fornece uma plataforma onde operários e máquinas podem trabalhar, para construir os andares superiores de um edifício.

No desenvolvimento de *software* o termo *scaffolding* tem significado semelhante ao da engenharia civil. A técnica de *scaffolding* cria automaticamente as estruturas mais significativas de um sistema de informação, incluindo as Interfaces do Usuário - UIs CRUD. Essas estruturas, assim como os andaimes, fornecem uma base temporária (plataforma) onde os desenvolvedores se apoiam para construir, os andares superiores do *software*, ou seja, para desenvolver os requisitos essenciais, que agregam valor ao sistema.

Apesar do *scaffolding* criar estruturas consideradas temporárias, o que se observa é que essas estruturas são mantidas ao longo do desenvolvimento dos sistemas, no entanto, é comum que algumas modificações sejam realizadas para atender as especificidades de cada projeto.

O acrônimo CRUD se origina das palavras *Create* (criar), *Read* (ler), *Update* (atualizar) e *Delete* (excluir), as 4 operações básicas utilizadas em banco de dados relacionais (LUMERTZ et al., 2016). As UIs CRUD possibilitam que os usuários possam manipular e persistir dados. Os sistemas de informação procuram padronizar essas interfaces, de forma a propiciar uma melhor aprendizagem por parte do usuário.

Em sistemas de informação, aproximadamente 65% das UIs são responsáveis por realizar as operações CRUD (MRACK et al., 2006). Essas UIs possuem códigos semelhantes, uma vez que, possuem o mesmo padrão de interface e realizam as mesmas operações CRUD. A codificação manual dessas UI causam a baixa produtividade e elevação dos custos de desenvolvimento, bem como causa a baixa qualidade do projeto (COHEN-ZARDI, 2013).

Atualmente a maioria dos *frameworks* para desenvolvimento web implementam alguma técnica de *scaffolding*. Uma característica em comum entre esses *frameworks* é o uso do padrão de projeto *Model-View-Controller* (MVC). O MVC é o padrão mais popular para o desenvolvimento de aplicações UIs (SHARAN, 2015), pois realiza a divisão de responsabilidades do sistema em 3 camadas, que dão nome ao padrão.

A camada de modelo *Model*, implementa o modelo de domínio, que é uma representação de classes conceituais, que modelam os problemas do mundo real (LARMAN, 2002). Objetos de domínio possuem uma estreita relação com as UI, uma vez que possuem atributos, tipos de dados e definem restrições como: valores mínimos, máximos e regras de validação (PUERTA; EISENSTEIN, 1999). A camada de visão *(View)* é responsável por definir a interface com o usuário, por meio dela, o usuário pode realizar a entrada e visualizar a saída dos dados. A camada de controle *(Controller)* é responsável receber as requisições do usuário e decidir por executar uma regra de negócio ou selecionar uma *View* para exibição.

A separação de responsabilidades provida pelo padrão MVC favorece a aplicação da técnica de *Scaffolding*, isso porque, o *scaffolding* utiliza o modelo de domínio, a camada *Model*, para gerar os artefatos que possuem código repetitivo, ou seja, as camadas *View* e *Controller*.

2.1.4.1 Tipos de Scaffolding

A técnica de *scaffolding* possui duas abordagens diferentes para geração de artefatos de *software* e de UIs CRUDs, essas abordagens são conhecidas como *scaffolding* estático e *scaffolding* dinâmico e estão respectivamente relacionadas às abordagens de transformação e execução de modelos do paradigma MDE.

O *Scaffolding* estático utiliza a transformação como abordagem para automatizar modelos para sistemas executáveis. Nessa abordagem, utilizando o padrão MVC, uma ferramenta de geração de código utiliza, como entrada, as classes do modelo de domínio (*Model*) para produzir, como saída, os códigos das camadas de controle (*Controller*) e de visão (*View*).

O *Scaffolding* dinâmico, por sua vez, utiliza a abordagem de execução para automatizar modelos para sistemas executáveis, essa abordagem, diferentemente da transformação, não gera código-fonte. O *Scaffolding* dinâmico utiliza técnicas de metaprogramação para interpretar as classes do modelo de domínio (*Model*) e gerar em tempo de execução as UIs CRUD.

As duas abordagens, estática e dinâmica, possuem suas vantagens e desvantagens. A abordagem estática, que gera os artefatos de código-fonte, tem como vantagem, a possibilidade de o desenvolvedor customizar o código gerado para atender as especificidades do projeto, no entanto, essa vantagem em alguns casos pode ser caracterizada como uma desvantagem. Caso seja necessário realizar diversas modificações no código-fonte, para atender os requisitos do projeto, pode ser mais produtivo criar manualmente os artefatos do que utilizar a técnica de *scaffolding* e depois realizar as modificações manualmente.

Outra desvantagem da abordagem estática está relacionada a modificação/evolução das classes de domínio. É comum que, durante o desenvolvimento, as classes de domínio sofram alterações, como por exemplo, adição de novos atributos ou alterações de tipos de dados, inclusão ou exclusão de associações, modificações nas regras de validação e integridades. Na abordagem estática, após realizar essas modificações a ferramenta de transformação deve ser executada novamente, esse processo sobrescreve os arquivos gerados na execução anterior, caso o desenvolvedor houvesse customizado os códigos gerados pela ferramenta, eles serão perdidos.

As vantagens e desvantagens do *scaffolding* dinâmico são inversamente proporcionais as vantagens e desvantagens do *scaffolding* estático. Por não gerar código-fonte, o *scaffolding* dinâmico, introduz limitações, ao desenvolvedor, para customizar as UIs produzidas. As modificações realizadas nas classes de domínio, no entanto, são refletidas automaticamente na UIs CRUDs geradas, não sendo necessário realizar novas transformações.

Nos *scaffolding* dinâmicos há maior gasto de memória e processamento, pois as UIs CRUD são geradas em tempo de execução. Para evitar esse consumo de recurso, os *scaffoldings* dinâmicos passam por diversas melhorias e otimizações visando reduzir o consumo de memória e de processamento.

Além dessas vantagens e desvantagens, todo *scaffolding* possui os seguintes benefícios:

- *Produtividade:* Talvez o benefício mais importante das ferramentas de *scaffolding*. Os artefatos produzidos por essas ferramentas são construídos quase que instantemente, ao passo que, se fossem desenvolvidos manualmente poderiam gastar horas de desenvolvimento. O trabalho de Magno (2015) apresenta que a aplicação da técnica de *scaffolding* para gerar UIs CRUD é cerca de 91% mais produtiva do que a codificação manual.

- *Personalizações:* Grande parte das ferramentas de *scaffolding* possibilitam que os desenvolvedores customizem as UIs CRUD geradas. As modificações podem ocorrer diretamente no código produzido, por meio da customização de *templates* ou por meio da extensão e sobrescrita da ferramenta de *scaffolding*. Apesar dessas possibilidades, o desenvolvedor ao escolher uma ferramenta de *scaffolding* deve avaliar as formas de personalização ofertadas por cada ferramenta, além de verificar o grau de complexidade para realizar a personalização.

- *Padrão de qualidade:* Os códigos gerados e as UI CRUDs produzidas pelas ferramentas de *scaffolding* geralmente não possuem erros, possuem alto desempenho e boa usabilidade, isso porque os artefatos produzidos utilizam, como modelo, códigos-fonte considerados padrões para diversas aplicações. Esses códigos são testados exaustivamente, além disso, quando erros ocorrem, as ferramentas são corrigidas de forma que os erros encontrados não se repitam nas futuras automatizações.

2.1.4.2 Scaffolding com templates

Independentemente do tipo de abordagem utilizada pelos *scaffoldings* e de outras ferramentas MDD, é comum que elas façam o uso de *templates* para realizar a automatização de modelos em código-fonte. Os *templates* são arquivos de texto, que utilizam como modelo o código-fonte de projetos bem-sucedidos, ou seja, de projetos que foram testados exaustivamente pela comunidade (MAGNO, 2015).

Os *templates* são modelos de código-fonte incompletos, que realizam determinada funcionalidade. Esses modelos possuem diversos parâmetros, que indicam onde e quais informações devem ser inseridas para que o *template* se transforme efetivamente em código-fonte funcional.

Os parâmetros de um *template* são preenchidos com dados extraídos dos modelos criados na fase de análise e projeto. Esses modelos podem ser, um conjunto de diagramas, arquivos de textos ou um modelo de domínio. O último pode ser representado por classes Groovy, no caso do *framework Grails*, ou por classes Java, no caso do *framework Play*. Esses *frameworks* web implementam o padrão MVC e a técnica de *scaffolding*.

O Código 2.1, apresenta um exemplo simplificado de um arquivo de *template* para a criação de campos de um formulário HTML. Todos os valores entre `${...}` são parâmetros e serão substituídos por dados extraídos da classe de modelo. A sintaxe para representação de parâmetros pode variar de acordo com a ferramenta de automatização de código.

```
<form method="${method}" action="${action}">
<label>${label}</label>
   <input type="${type}" name="${name}" value="${value}" />
   <button type="submit">Cadastrar</button>
</form>
```

Código 2.1: Exemplo de *template* para criação de formulário HTML

Comumente os dados são extraídos das classes de domínio, por meio da técnica de metaprogramação, uma vez extraídos, os dados são armazenados em estruturas chamadas de Contexto.

Todo parâmetro presente no arquivo de *template* está relacionado a um dado armazenado no contexto, desta forma, a ferramenta de automatização é capaz de realizar o preenchimento do *template* com os dados presentes no Contexto. Um Contexto pode possuir mais dados do que um arquivo de *template*, isso deve-se ao fato de que um contexto poder ser utilizado para o preencher diferentes *templates*. O Código 2.2 apresenta um exemplo de contexto em Java.

```
Contexto c = new Contexto();
c.setMethod("post");
c.setAction("cliente/insert");
c.setLabel("Foto:");
c.setType("file");
c.setName("cliente.foto");
c.setValue("");
```

Código 2.2: Exemplo de Contexto em Java

O código-fonte gerado pela ferramenta de automatização é o resultado da substituição dos parâmetros dos *templates* pelos respectivos dados presentes no Contexto. O Código 2.3 apresenta um exemplo de código-fonte gerado utilizando o *template* apresentado no Código 2.1 e o contexto apresentado no Código 2.2.

```
<form method="post" action="cliente/insert">
   <label>Foto:</label>
   <input type="file" name="cliente.foto" value="" />
   <button type="submit">Cadastrar</button>
</form>
```

Código 2.3: Código gerado por meio do *Template* e do Contexto

2.1.4.3 Scaffolding estático e dinâmico no framework Grails

O Grails é um *framework* web, de código aberto e licença Apache, desenvolvido para a plataforma Java e escrito na linguagem Groovy, ele utiliza diversos recursos e paradigmas como: convenção sobre configuração, Mapeamento Objeto Relacional (*Object-relational mapping* - ORM), Linguagem específica de Domínio (*Domain-Specific Language* - DSL), metaprogramação em tempo de execução e compilação e programação assíncrona. O Grails tem por objetivo aumentar a produtividade no desenvolvimento em aplicações web (PIVOTAL, 2014).

O Grails tem ganhando destaque entre os *frameworks* de desenvolvimento web, atualmente diversas empresas como a NETFLIX, Disney, Sky e LinkedIn utilizam o Grails em seus projetos (WEISSAMANN, 2015). Por ter uma boa aceitação da comunidade de desenvolvimento e por implementar as duas abordagens de *scaffolding*, a estática e a dinâmica, o Grails foi escolhido para demonstrar a aplicação da técnica de *scaffolding* e exemplificar a diferença entre as duas abordagens de automatização. A aplicação da técnica inicia com a criação da classe de domínio, apresentada no Código 2.4.

```
1  class Livro{
2    String titulo
3    String isnb
4    String edicao
5    String ano
6    Integer num_pag
7    static constraints = {
8      titulo nullable:false
9      isbn unique:true
10   }
11 }
```

Código 2.4: Classe de Domínio do *Grails*

Uma classe de domínio representa a estrutura de um objeto manipulado pelo sistema. Por meio dela, é possível descrever quais características (atributos), restrições, comportamentos/trocas de mensagens (métodos) e as associações que um objeto possui. As duas abordagens, estática e dinâmica, necessitam que o modelo de domínio seja criado para aplicar a técnica de *scaffolding*. No Grails, o modelo de domínio corresponde as classes Groovy, Código 2.4. No exemplo, a classe de domínio representa um `Livro`, que possui como características o título (`titulo`), o número do livro no padrão internacional (`isnb`), o número da edição (`edicao`), o ano de publicação (`ano`) e o número de páginas (`num_pag`), conforme mostrado nas linhas 2 a 6. As linhas de 7 a 10 definem as restrições de validação, no caso o título do livro não pode ser nulo e o isbn deve ser único.

Abordagem Estática

Para aplicar a técnica de *scaffolding* estático é necessário executar, no terminal, o comando `generate-all`. A execução deste comando faz a análise e a extração de dados da classe de domínio. Os dados, então, são vinculados a arquivos de *templates*, que são renderizados, gerando os artefatos de código-fonte. Como resultado da execução do comando, foram gerados os arquivos de código-fonte: a classe controladora `LivroController.groovy`; a classe de testes unitários `LivroControllerSpec.groovy` e as interfaces gráficas: `edit.gsp`; `create.gsp`; `index.gsp` e `show.gsp`. O arquivo `LivroController.groovy`, Código 2.5, é a classe controladora para a classe de domínio `Livro.groovy`.

```
1  @Transactional(readOnly = true)
2      class LivroController {
3
4      static allowedMethods = [save: "POST", update: "PUT", delete: "DELETE"]
5
6      def create() {
7          respond new Livro(params)
8      }
9      @Transactional
10         def save(Livro livro) { //...
11     }
12     @Transactional
13         def update(Livro livro) {//...
14     }
15     // outros métodos da classe
16 }
```

Código 2.5: Classe `LivroController` gerada pelo comando `generate-all`

A linha 1, do Código 2.5, é utilizada para definir que, por padrão, os métodos da classe `LivroController` não realizaram transações, que modifiquem a base de dados. Esse comportamento é modificado, nas linhas 9 e 12, para os métodos `save` e `update`, com uso do tipo de anotação `@Transactional`, uma vez que eles devem modificar a base de dados. A anotação `@Transactional` implementa transações e garante as propriedades ACID. A linha 4 associa quais métodos do protocolo HTTP devem ser utilizados para acessar um método da classe controladora. O método `save` somente pode ser acessado com o método `POST` do HTTP, o `updade` com o método `PUT` e o `delete` com o método `DELETE`. A linha 12 define o método `create` que é responsável por criar um novo livro, que será exibido em um formulário para que o usuário possa preencher seus atributos.

No modelo MVC, a classe controladora é responsável por expor métodos de acesso para as funcionalidades do sistema. Em sistemas web o acesso aos métodos do controlador é

realizado por meio das *Uniform Resource Locator* (URL), que geralmente estão associadas a métodos do protocolo *HiperText Transfer Protocol* (HTTP). Ao conjunto das URLs e dos métodos do protocolo HTTP, para acesso aos métodos da classe controladora, dá-se nome de rotas.

A tabela Tabela 2.1 apresenta as rotas para a classe controladora `LivroControllers.groovy`, como também os arquivos de visão associados a cada método do controlador, bem como a sua descrição.

Tabela 2.1: Rotas para `LivroController.groovy`

Método HTTP	Caminho	Página na Camada de Visão	Método do controlador no controlador	Descrição
GET	/livro	index.gsp	index	Lista todos os objetos `livro`
GET	/livro/create	create.gsp	create	Formulário HTML para criar novos `livros`
POST	/livro/save		save	Persiste um novo `livro`
GET	/livro/show/:id	show.gsp	show	Mostra o livro com o id `:id`
GET	/livro/edit/:id	edit.gsp	edit	Formulário HTML para alterar `livro` com id :id
PUT	/livro/update		update	Altera o`livro` com id `:id`
DELETE	/livro/delete/:id		delete	Exclui o `livro` com id `:id`

Os métodos `index`, `create`, `show` e `edit` da classe `LivroControllers.groovy` quando chamados exibem UIs, para que os usuários possam manipular os objetos, já os métodos `save`, `update` e `delete` não estão associados a UIs, eles são utilizados pelo *framework* para realizar operações de persistência na base de dados.

A Figura 2.11 apresenta o acesso da rota `/livro/create` que executa o método `create` da classe `LivrosController`, que por sua vez exibe a página `create.gsp`, utilizada para cadastrar novos livros.

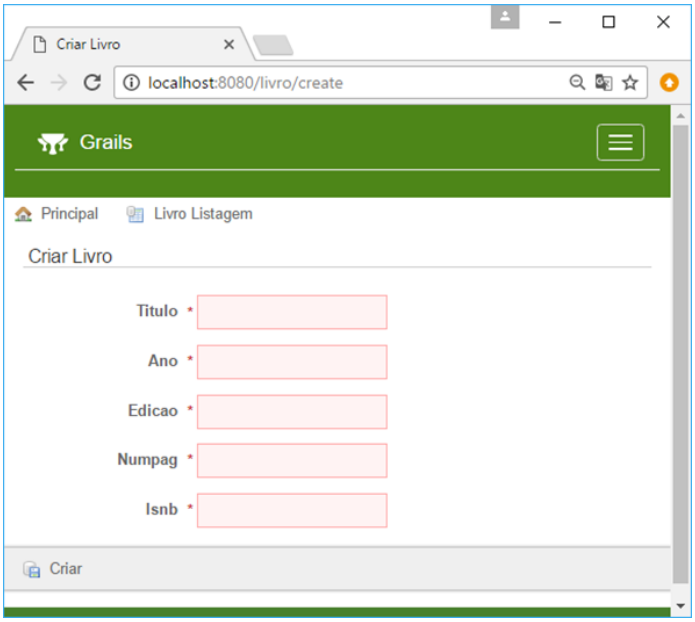

Figura 2.11: Interface gráfica para cadastrar livros gerada pelo *scaffolding* do *Grails*

Abordagem Dinâmica

Para aplicar a técnica de *scaffolding* dinâmico é necessário criar uma classe controladora e nessa classe adicionar o atributo estático `scaffold`. O atributo `scaffold` deve ser inicializado com nome da classe de domínio, para a qual se deseja criar as UIs CRUD, conforme mostrado no Código 2.6, ou deve ser inicializado com o valor `true`, no caso de a classe controladora possuir o mesmo nome que a classe de domínio, acrescida do sufixo *Controller*, conforme mostrado Código 2.7.

```
1 class BookController{
2     static scaffold = Livro;
3 }
```

Código 2.6: *Scaffolding* dinâmico: Usando do nome da classe de Domínio

```
1 class LivroController{
2     static scaffold = true;
3 }
```

Código 2.7: *Scaffolding* dinâmico: Usando do nome da classe *Controller*

Na abordagem dinâmica, a classe controladora em tempo de desenvolvimento, geralmente não possui métodos, tampouco rotas de acesso. Entretanto, em tempo de execução, o *framework* Grails, por meio da classe `ScaffoldingControllerInjector.groovy`, procura por classes controladoras que possuam o atributo estático `scaffold`, ao localizar essas classes, o método `performInjectionOnAnnotatedClass()` é chamado para tornar a classe localizada filha da classe `RestfulController.groovy`.

A classe `ResfulController` possui todos os métodos para manipulação de dados da classe de domínio, bem como as rotas para acesso das UIs CRUD. Ela possui muita semelhança com a classe controladora criada pela abordagem estática, isso porque, ela é o *template* utilizado pelo *scaffolding* estático.

Quando a rota `/livro/create` é acessada, o método `create` da classe `LivroController`, que foi herdado da classe `RestfulController`, é executado. O método `create` cria uma nova instância da classe, que foi indicada no atributo estático `scaffold`, a instância é então enviada para um arquivo de visão, que será responsável por exibir, neste caso, um formulário para preenchimento dos atributos, conforme mostrado na Figura 2.11.

O arquivo de visão, que irá apresentar o formulário, é selecionado em tempo de execução, por meio da classe `ScaffoldingViewResolver` do *framework Grails*. Os arquivos de visão utilizados, tanto pela abordagem estática, quanto pela abordagem dinâmica, são *templates*, ou seja, são modelos de códigos-fonte com parâmetros que devem ser substituídos por dados das classes de domínio.

2.2 Metaprogramação

O termo metaprogramação tem sido definido de diferentes maneiras ao longo dos anos. Para Bartlett (2005), metaprogramação é escrever programas que geram código e, para Rideau (1999), metaprogramação é a arte de programar programas que leem, transformam ou escrevem outros programas. De acordo com essas definições, a metaprogramação consiste na construção de programas que são capazes de analisar ou de gerar outros programas.

Cordy e Shukla (1992) definem metaprogramação como a "técnica de especificar modelos de código de *software* genérico a partir dos quais as classes de componentes de *software*, ou partes deles, podem ser instanciadas automaticamente para produzir novos componentes de *software*"e Štuikys e Damaševičius (2012) definem metaprogramação como "um paradigma de programação de nível superior que visa estender os limites da programação na construção de programas automaticamente". Para esses autores, a metaprogramação é vista como o surgimento de uma nova geração de linguagens de programação, capaz de ampliar o poder de abstração das linguagens atuais, por meio da especificação de modelos genéricos, que possibilitam estender os limites da programação ao gerar *softwares* automaticamente.

Štuikys e Damaševičius (2012) afirmam que a metaprogramação e a programação são assuntos da mesma área de conhecimento e que a metaprogramação não pode ser concebida sem conhecer os fundamentos da programação. Para Sheard (2001), na metaprogramação, os programas são dados, essas assertivas corroboram para a definição de metaprogramação dada por Magno (2015), que afirma que "programas de computador analisam e transformam dados de entrada (*input*) em dados de saída (*output*), na metaprogramação os dados de entrada e saída são programas".

Na área de metaprogramação, outros conceitos e definições são considerados importantes para compreensão da matéria, sendo eles: a *metalinguagem*, a *linguagem-objeto*, o *programa-objeto*, o *metaprograma* e *sistemas de metaprogramação*.

- *Linguagem-objeto:* É a linguagem utilizada para codificar o programa-objeto.
- *Programas-objetos:* São programas comuns, ou melhor, seu código-fonte (MAGNO, 2015). Sheard (2001) define programa-objeto como "qualquer sentença de uma linguagem formal".
- *Metalinguagem:* É qualquer linguagem ou sistema simbólico que é utilizado para discutir, descrever ou analisar outra linguagem de programação ou sistema simbólico (BATORY, 1998).

- *Metaprograma:* É um programa escrito em uma metalinguagem que manipula programa-objeto e pode gerar novos programas-objetos, a partir de fragmentos de códigos já existentes (CZARNECKI; EISENECKER, 2000; ŠTUIKYS; DAMAŠEVIČIUS, 2012).
- *Sistemas de metaprogramação:* Sistemas de metaprogramação são o conjunto do metaprogramas e programas-objetos.

A Figura 2.12 apresenta o esquema de funcionamento da metaprogramação elucidando os conceitos definidos. Um metaprograma recebe como entrada um programa-objeto, o metaprograma faz o processamento da entrada, que dependendo do seu tipo pode ser uma análise, uma transformação ou uma geração. A saída do metaprograma pode ser: o resultado da análise, ou o mesmo programa-objeto transformado, ou ainda um novo programa-objeto.

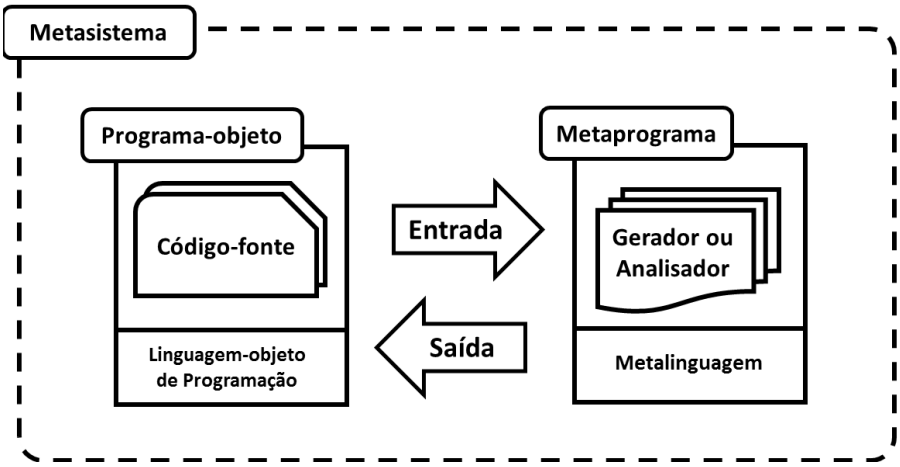

Figura 2.12: Esquema de funcionamento da metaprogramação

A literatura apresenta diversas taxonomias para metaprogramação (DAMAŠEVIČIUS; ŠTUIKYS, 2015; SHEARD, 2001). Algumas das classificações mais utilizadas são: quanto a função (geradores ou analisadores de programas); quanto ao tempo de execução (estáticos ou dinâmico) e quanto ao tipo de metalinguagem (homogêneo e heterogêneo).

Os geradores de programa são utilizados para gerar programas-objetos que resolvem problemas específicos e possuem soluções similares, exemplos de geradores são compiladores, interpretadores e transformadores. Os analisadores de programa são capazes de analisar e de extrair características importantes do código-fonte de um programa-objeto, exemplos dos resultados produzidos pelos analisadores são: fluxogramas, diagramas, otimizações, refatorações, sistemas de avaliação, etc. (SHEARD, 2001).

Os programas-objetos podem ser estáticos ou dinâmicos. Os geradores estáticos geram código que são posteriormente processados por compiladores normais que constroem o programa-objeto. Os geradores dinâmicos ou de tempo de execução escrevem ou constroem programas-objetos que são executados imediatamente (SHEARD, 2001).

O tipo de metalinguagem utilizada para construir um sistema de metaprogramação possibilita classificá-lo em homogêneo ou heterogêneo. Nos sistemas de metaprogramação, onde a metalinguagem é diferente da linguagem-objeto, tem-se sistemas heterogêneos, por outro lado, quando a metalinguagem utilizada é igual a linguagem-objeto tem-se sistemas homogêneo. Apenas em sistemas homogêneos é possível aplicar a técnica de reflexão (SHEARD, 2001).

A reflexão é a capacidade que um programa tem de observar a si próprio e de possivelmente modificar a sua estrutura e comportamento (FORMAN, 2005; MALENFANT et al., 1996). A reflexão é expressa em termos de introspeção e intercessão. A introspeção é ação de olhar para dentro de si, do programa observar suas estruturas, como por exemplo, suas classes, atributos, métodos e associações. A intercessão é a ação de intervenção, ou seja, o programa intervém em sua execução, podendo modificar sua estrutura e seu comportamento.

Normalmente a reflexão ocorre dinamicamente, em tempo de execução, e com uso de linguagens interpretadas, isso porque, as informações sobre a estrutura do código e dos metadados geralmente são perdidas no processo de compilação (OLIVEIRA, 2012; ŠTUIKYS; DAMAŠEVIČIUS, 2012). Sistemas que possibilitam a reflexão preservam essas informações e, por meio da metaprogramação, é possível alterar a estrutura e o comportamento do sistema (ATTARDI; CISTERNINO, 2001).

Os metadados são dados estruturados que descrevem as características das entidades emissoras de informação, são utilizados para auxiliar na identificação, descoberta, avaliação e gestão dessas entidades (ŠTUIKYS; DAMAŠEVIČIUS, 2012). Em metaprogramação, os metadados são marcações adicionadas ao código-fonte, que são obtidas em tempo de execução ou compilação, para indicar quais ações devem ser realizadas. Muitas especificações Java e *frameworks* utilizam metadados e reflexão para realizar suas funções, alguns exemplos são: *Java Persistence API* e os *framewoks Hibernate*, *EclipseLink* e *TopLink*, que utilizam os metadados para realizar o mapeamento de classes em tabelas de banco de dados; *Java Server Faces – JSF*, que utiliza os metadados para indicar quais classes são gerenciáveis e como os dados, manipulados por elas, serão compartilhados; e *Context Dependency Injection* (CDI), que utiliza os metadados para fazer a injeção de dependências e a inversão de controle.

A literatura apresenta diversos motivos para uso da metaprogramação, dentre eles podemos destacar:

- *Produtividade:* Uma vez que, os metaprogramas possibilitam a criação automática de outros programas, ela possibilita alcançar maior produtividade no desenvolvimento de sistemas (ŠTUIKYS; DAMAŠEVIČIUS, 2012).
- *Redução da Complexidade:* A metaprogramação contribui para a gestão da complexi-

dade do desenvolvimento de *software*, uma vez que, ao utilizar geradores de programas, menos código precisa ser escrito manualmente (ŠTUIKYS; DAMAŠEVIČIUS, 2012).

- *Redução de custos e prazo de entrega:* Por permitir a geração automática de programas, a metaprogramação consequentemente possibilita a redução dos custos de desenvolvimento, uma vez que, os códigos não serão codificados manualmente, além disso, o prazo para codificação é menor, pois é realizado automaticamente, assim o produto é entregue ao cliente em um tempo menor (ŠTUIKYS; DAMAŠEVIČIUS, 2012).

- *Desempenho e Qualidade:* Uma vez que os geradores produzem programas para problemas específicos que possuem soluções similares, os programas gerados tendem a possuir melhor qualidade e desempenho do que os programados manualmente. Além disso, os analisadores possibilitam que otimizações e fatorações sejam realizadas nos programas analisados (ŠTUIKYS; DAMAŠEVIČIUS, 2012; SHEARD, 2001).

- *Foco no projeto de sistemas:* Com menos código sendo executado manualmente, os desenvolvedores podem concentrar menos esforços na programação e mais esforços na integração de tarefas do domínio e do projeto do sistema.

2.2.1 Metaprogramação em Java

A linguagem de programação Java é uma linguagem de propósito geral, concorrente, baseada em classes e orientada a objetos, foi projetada para ser simples, permitindo que muitos programadores obtenham fluência na linguagem (GOSLING et al., 2013). A linguagem Java permite a criação de metassistemas homogêneos, uma vez que, ela é ao mesmo tempo metalinguagem e linguagem-objeto.

2.2.1.1 Java Reflections API

Por meio da *Java Reflection API* é possível inspecionar classes, interfaces, atributos e métodos, é possível ainda, criar objetos, recuperar valores de atributos e executar métodos de objetos em tempo de execução. Essas funcionalidades dão mais flexibilidade ao *software* construído, que se adapta melhor às mudanças. Além disso, componentes reflexivos tendem a ser mais reutilizados em outras aplicações (FORMAN, 2005).

A reflexão em Java tem início com a obtenção de um objeto `Class`, este objeto possibilita extrair informações sobre a classe, tais como nome do pacote, seus modificadores (`public`, `abstract`, `final`, etc), a lista completa de membros (atributos, métodos, construtores) e seus tipos, como por exemplo, as interfaces que implementa e as classes que estende (ARNOLD; GOSLING, 2007). Existem 4 maneiras de se obter um objeto `Class`.

- *Instância da classe:* Todos os objetos em Java são filhos da classe `Object`, que implementa o método `getClass`, esse método retorna o objeto `Class` associado ao objeto em tempo de execução. A linha 7 do Código 2.8 exibe um exemplo de utilização do método `getClass`.
- *Nome plenamente qualificado:* A classe `Class` possui o método estático `forName` que permite obter o objeto `Class` fornecendo o nome plenamente qualificado da classe, ou seja, incluindo o nome do pacote onde a classe está localizada. A linha 8 do Código 2.8 exibe um exemplo de utilização do método `forName`.
- *Literal classe:* É possível obter o objeto `Class` utilizando o nome da classe seguido do literal `.class`. A linha 9 do Código 2.8 exibe um exemplo de utilização do literal de classe.
- *Reflexão:* Outra possibilidade é utilizar métodos de reflexão que retornam objetos `Class`. A linha 10 do Código 2.8 demostra a utilização do método de reflexão `getClasses` que retorna as classes internas da classe Exemplo.

```
1  package model;
2
3  public class Exemplo {
4
5      public static void main(String[] args) throws ClassNotFoundException {
6          Exemplo e = new Exemplo();
7          Class clazz1 = e.getClass(); //Instância da classe
8          Class clazz2 = Class.forName("mode.Exemplo"); //Nome plenamente qualificado
9          Class clazz3 = Exemplo.class; //Literal classe
10         Class[] clazz4 = clazz1.getClasses(); //Reflexão
11     }
12     class ClasseInterna{ }
13 }
```

Código 2.8: Obtenção de objetos de `Class`

Conforme exposto, um objeto de `Class` permite consultar e extrair todas as informações sobre sua classe, dentre as principais informações, que podem ser recuperadas, estão os Membros da Classe. Faz parte dos Membros de uma classe os atributos, os métodos e os construtores que são encapsulados, em Java, nas respectivas classes `Field`, `Method` e `Constructor`. Os membros de uma classe podem ser considerados públicos ou declarados.

Os membros públicos são aqueles que possuem o modificador `public`, incluindo os membros públicos herdados. Os membros declarados são todos os membros da classe, exceto os membros herdados. O Código 2.9 apresenta a classe de modelo `Aluno` que possui os atributos declarados `matricula`, `nome`, `dataNascimento` e `cpf`.

```
1 public class Aluno {
2     private Integer matricula;
3     private String nome;
4     private Calendar dataNascimento;
5     private String cpf;
6         //getter e setter
7 }
```

Código 2.9: Classe de modelo `Aluno`

O Código 2.10 demostra o uso da técnica de reflexão na obtenção de atributos declarados da classe Aluno. A linha 3 obtém uma instância de `Class` do objeto Aluno. A linha 4 faz uma chamada ao método `getDeclaredFields()` que retorna um vetor de atributos (`Field`) da classe aluno Código 2.9. A linha 5 imprime o nome de todos os atributos, por meio da estrutura de repetição `for`.

```
1 public class ShowFields {
2     public static void main(String[] args) {
3         Class clazz = Aluno.class;
4         for (Field atributo : clazz.getDeclaredFields()) {
5             System.out.println(atributo.getName());
6         }
7     }
8 }
```

Código 2.10: Uso de reflexão para obter os atributos da classe `Aluno`

A Tabela 2.2 apresenta os métodos de introspecção, da *Java Reflection API*, utilizados para obter membros públicos de uma classe. Os membros públicos são todos os membros que possuem o modificador de acesso público, inclusive os membros que foram herdados de outra classe.

Tabela 2.2: Métodos de introspecção da classe `Class` para membros públicos

Elemento	Método de Introspecção
`Field`	`getField(String nome)` `getFields()`
`Method`	`getMethod(String nome, Class<?> ... parametros)` `getMethods()`
`Constructor`	`getConstructor(Class<?> ... parametros)` `getConstructors()`

A Tabela 2.3 apresenta os métodos de introspecção, da *Java Reflection API*, utilizados para obter membros declarados de uma classe. Os membros declarados são todos aqueles declarados dentro na classe, não importando seu modificador de acesso (`public`, `private` ou `protected`).

Tabela 2.3: Métodos de introspecção da classe `Class` para membros declarados

Elemento	Método de Introspecção
`Field`	`getDeclaredField(String nome)`
	`getDeclaredFields()`
`Method`	`getDeclaredMethod(String nome, Class<?> ... parametros)`
	`getDeclaredMethods()`
`Constructor`	`getDeclaredConstructor(Class<?> ... parametros)`
	`getDeclaredConstructors()`

Os atributos de uma classe (Objetos `Field`) podem ter seus valores lidos ou alterados por reflexão, assim como os métodos (Objetos `Method`) podem ser invocados, passando parâmetros e recebendo o valor retornado e os construtores (Objetos `Constructor`) podem instanciar novos objetos. Essas alterações de estado e comportamento, também conhecidas por intercessão, são realizadas pelos métodos apresentados na Tabela 2.4.

Tabela 2.4: Métodos de intercessão das classes `Field`, `Method` e `Constructor`

Elemento	Método de Intercessão
`Field`	`set(Object obj, Object v)`
	`setBoolean(Object obj, boolean v)`
	`setByte(Object obj, byte v)`
	`setChar(Object obj, char v)`
	`setDouble(Object obj, double v)`
	`setFloat(Object obj, float v)`
	`setInt(Object obj, int v)`
	`setLong(Object obj, long v)`
	`setShort(Object obj, short v)`
`Method`	`invoke(Object obj, Object... argumentos)`
`Constructor`	`newInstance(Object... argumentos)`

O objeto `Field` possui um método específico, para realizar a atribuição de valor, para cada tipo primitivo. O parâmetro `obj` é o objeto que terá o valor do atributo (`Field`) modificado e o parâmetro `v` é o valor que será atribuído.

O objeto `Method` possui o método `invoke` que possibilita invocar o método de um objeto. O método `invoke` possui o parâmetro `obj` que corresponde ao objeto que terá o método

invocado, o segundo parâmetro corresponde a lista de parâmetros do método que será invocado.

O objeto `Constructor` cria uma nova instância do objeto, ao qual o construtor pertence, com os argumentos de inicialização especificados.

O Código 2.11 apresenta um exemplo de código Java que faz intercessão. Na linha 5 um objeto da classe `Aluno` (Código 2.9) é criado. Na linha 7 o objeto `Class` associado ao objeto aluno é obtido, na linha 9 uma introspeção é realizada para obter o atributo `nome` da classe `Aluno`, na 11 uma intercessão é realizada para modificar o valor do atributo `nome` do objeto `aluno`.

```
1  public class ExemploIntercessao {
2
3      public static void main(String[] args) throws NoSuchFieldException,
           IllegalArgumentException, IllegalAccessException {
4          //cria um objeto novo
5          Aluno jose = new Aluno();
6          //Obtém o objeto Class de Aluno
7          Class clazzAluno = Aluno.class;
8          //Obtém o atributo(Field) nome da classe Aluno
9          Field nomeField = clazzAluno.getField("nome");
10         //Atribui o nome (Objeto String) José da Silva ao objeto jose
11         nomeField.set(jose, "José da Silva");
12         //Obtém o valor do atributo(Field) nome do objeto jose
13         String nome = nomeField.get(jose).toString();
14         //Exibe o valor do atributo nome do objeto jose
15         System.out.println("Nome " + nome);
16     }
17 }
```

Código 2.11: Exemplo de reflexão

2.2.1.2 Java Annotations API

Em Java, as anotações são as estruturas utilizadas para anexar metadados a elementos da linguagem. Os elementos que podem ser anotados são: pacote (`package`), classes (`Class`), enumeradores (`Enum`), interfaces (`interfaces`), atributos (`Field`), métodos (`Method`), construtor (`Constructor`), parâmetros (*formal parameter*) e variáveis locais (*local variable*) (GOSLING et al., 2013).

Para anotar um elemento é necessário criar um tipo de anotação ou utilizar um tipo já criado pelo Java ou por outro *framework*. Um tipo de anotação é um tipo especial de `interface` (GOSLING et al., 2013). Assim a declaração de um tipo de anotação é semelhante

a declaração de uma interface, as diferenças ficam por conta da adição do símbolo "@" antes da palavra reservada `interface` e no uso dos métodos. O símbolo "@" em inglês significa *at* um acrônimo para *annotation type*, ou tipo de anotação. O Código 2.12 apresenta uma declaração de tipo de anotação.

```
1 @Inherited
2 @Documented
3 @Retention(RetentionPolicy.RUNTIME)
4 @Target(Element.TYPE)
5 @interface Entity{
6    String value() default "";
7    boolean persistente() default true;
8 }
```

Código 2.12: Declaração de tipo de anotação

Os métodos `value()` e `persistence()` declarados nas linhas 6 e 7 são métodos de tipo de anotação, também conhecidos como elementos. Eles são responsáveis por guardar informações relacionadas a anotação, na forma de pares de elemento-valor (GOSLING et al., 2013). Nas interfaces tradicionais, os métodos devem ser implementados pela classe que implementa a interface.

Uma vez criado o tipo de anotação, para anotar um elemento basta adicionar um "@" com o nome do tipo de anotação, antes da declaração do elemento a ser anotado. O Código 2.13 apresenta exemplos de anotação de classes.

```
1 @Entity
2 public class SintaxeMarcadora{}
3
4 @Entity("valor")
5 public class SintaxeUnicoElemento{}
6
7 @Entity(value="valor", persistence = false)
8 public class SintaxeGeral{}
```

Código 2.13: Sintaxe para anotação de elementos

Quando, ao anotar um elemento, não são fornecidos valores para os métodos do tipo de anotação, os valores `default` são utilizados, como na linha 1 do Código 2.13. Neste caso e nos casos onde o tipo de anotação não possui métodos, a anotação é conhecida com marcadora (*marker*). No caso da anotação possuir apenas o método `value`, ela é considerada uma anotação de tipo único, neste caso, ao realizar a anotação do elemento Java, não é necessário fornecer o nome método `value`, conforme mostrado na linha 4 do Código 2.13.

No Código 2.12 é possível verificar que existem anotações no tipo de anotação, essas anotações são chamadas de metaanotações. O pacote `java.lang.annotation` contém as metaanotações que podem ser aplicadas a um tipo de anotação, as mais relevantes são: `@Inherited`, `@Retention`, `@Repetable` e `@Target`.

- *@Inherited:* Propaga a anotação da superclasse (mãe) para as subclasses (filha).
- *@Retention:* Determina se a anotação estará disponível em tempo de compilação ou em tempo de execução. Em tempo de execução, os metadados estarão disponíveis para consulta, por meio da reflexão.
- *@Target:* Especifica quais elementos Java podem ser anotados com o tipo de anotação. Por exemplo, é possível criar tipos de anotações restritos a Classes, Atributos ou Métodos.
- *@Repetable:* Essa metaanotação é utilizada para indicar que o tipo de anotação, anotado por ela, pode ser utilizado mais de uma vez para anotar um mesmo elemento Java.

As anotações podem estar disponíveis em tempo de compilação ou execução dependendo da configuração da metaanotação `@Retention`. Em tempo de execução, a anotação pode ser consultada, por meio da reflexão. O Código 2.14 demostra como realizar a leitura de um elemento Java anotado.

```
1  public class IntropecaoDeAnotacao {
2
3      public static void main(String[] args) {
4
5          Class sintaxe = SintaxeGeral.class;
6
7          if (sintaxe.isAnnotationPresent(Entity.class)){
8              Entity entity = (Entity) sintaxe.getAnnotation(Entity.class);
9              String vTable = entity.value();
10         }
11     }
12 }
```

Código 2.14: Reflexão: Obtendo elementos anotados

Na linha 5 do Código 2.14 é criado um objeto `Class` da classe `SintaxeGeral`. Na linha 7 é verificado se a classe `SintaxeGeral` possui o tipo de anotação `@Entity`, criada no Código 2.12, se ela possuir, a execução prossegue na linha 8 com a obtenção e armazenamento do tipo de anotação `@Entity` na variável `entity`. Na linha 9, o método `value` da anotação `@Entity` é chamado e o valor retornado é armazenado na variável `vTable`.

Gosling et al. (2013) afirmam que as anotações são recursos poderosos da linguagem Java, mas devem ser usados com cautela e sabedoria, o uso de muitas anotações pode poluir o código-fonte, além disso, anotações são adequadas para análise automática, por meio de ferramentas de processamento ou por reflexão, essas ferramentas geralmente definem os tipos de anotação zelando pela padronização. Em geral poucos programadores precisam definir seus próprios tipos de anotação.

2.3 Desenvolvimento Web em Java

O desenvolvimento de sistemas web, *server-side*, em Java, iniciou com o lançamento das *servlets* em 1997. As *servlets* processavam as requisições em *threads* paralelas, o que as tornavam mais rápidas, escaláveis e robustas, do que os modelos CGIs existentes na época, essas características contribuíram para a adoção desta tecnologia pelo mercado web (SILVEIRA, 2012).

Apesar de apresentar desempenho superior, as *servlets* não foram concebidas para criar páginas web, de forma a desacoplar a lógica da aplicação da camada de visão, isso tornava o desenvolvimento improdutivo. Então, em 1998, foi lançado o *JavaServer Pages* – JSP, com objetivo aumentar a produtividade e facilitar a criação de páginas.

O JSP utilizava as *servlets* como infraestrutura, assim ele possibilitava criar aplicações rápidas, escaláveis e robustas, ao mesmo tempo que aumentava a produtividade do desenvolvimento. No entanto, a forma como os desenvolvedores utilizaram o JSP para desenvolver aplicações, na época, foi considerado uma prática ruim. O desenvolvimento consistia na junção de código HTML com fragmentos de código Java, essa prática é considerada ruim, pois promove a mistura de responsabilidades (SILVEIRA, 2012).

Visando a separação de responsabilidade e tornar a utilização dos *servlets* mais simples, diversos *frameworks* MVC foram lançados, como por exemplo: o Struts, o WebWorks, Spring MVC e Tapestry (SILVEIRA, 2012). No entanto, a utilização desses *frameworks* era complexa, obrigando que diversas configurações fossem realizadas, por meio da escrita de código XML e arquivos de propriedades. Além disso, as classes deviam estender ou implementar as interfaces do *framework*, tornando a aplicação acoplada e dependente deles (CAVALCANTI, 2014).

Dada a dificuldade do desenvolvimento Web surgiu o *framework* VRaptor. O VRaptor é um *framework* web – MVC, de código aberto e licença Apache 2.0, escrito e desenvolvido para a plataforma Java, que conta com uma grande comunidade de desenvolvedores e usuários (VRAPTOR, 2017).

2.3.1 VRaptor

O VRaptor foi criado em 2004, na Universidade de São Paulo, pelos irmãos Paulo Silveira e Guilherme Silveira. Em 2016, foi lançada a primeira versão estável do *framework*, esta versão contou com a participação de diversos desenvolvedores, que empregaram ideias e boas práticas do *framework Ruby On Rails* (CAVALCANTI, 2014). A atual versão do *framework* utiliza as principais tecnologias da plataforma Java, dentre elas a especificação *Contexts and Dependency Injection – CDI* e *Bean Validator API*.

De acordo com VRaptor (2017), a documentação oficial do projeto, o *VRaptor* entrega alta produtividade no desenvolvimento de aplicações web, tornando o desenvolvimento uma tarefa mais simples e fácil; a curva de aprendizagem de *framework* é alta, de forma que, com pouco tempo, é possível aprender o necessário para construir aplicações; o código da aplicação construída é modularizado e desacoplado do *framework*, o que torna fácil a manutenção e os testes; o *framework* promove a economia de tempo e dinheiro, uma vez que a alta produtividade reduz o número de horas de trabalho para implementar novas funções; ele possui suporte a aplicações orientadas a serviços - SOA e a Transferência de Estado Representacional – RESTful, além de fazer uso das melhores práticas de desenvolvimento e possuir uma vasta documentação.

Cavalcanti (2014) apresenta que o *VRaptor* possibilita que o desenvolvedor tenha liberdade para escolher sua camada de visão, no entanto, isso implica que o *framework* não irá auxiliar muito no desenvolvimento do HTML das páginas. Assim, a construção das UIs deve ser realizada manualmente, utilizando HTML, CSS e JavaScript ou utilizando bibliotecas como *JQuery UI* e *Bootstrap*.

O Código 2.15 apresenta uma classe controladora do VRaptor. A anotação `@Controller` indica que a Classe `PrimeiroController` deve responder as requisições do usuário, a anotação `@Inject` indica que a classe `PrimeiroController` necessita de uma instância de `Result` para realizar suas ações. A instância de `Result` é injetada automaticamente pelo servidor de aplicação, utilizando a especificação CDI. A linha 7 implementa o método `home()` que na linha 8 e 9 utiliza a instância de `result` para incluir na camada de visão as variáveis `titulo` e `msg` com os respectivos valores "Título"e "Olá, VRaptor".

O Código 2.16 apresenta o código da camada de visão. O código por padrão deve estar no diretório `WEB-INF/jsp/primeiro/home.jsp`, onde `primeiro` corresponde ao nome da classe controladora, sem o sufixo `controller`, e o nome do arquivo `home.jsp` corresponde ao nome do método `home()`. Por meio da *Expression Language*, linha 4 e 7, o valor das variáveis `titulo` e `msg` são adicionados ao código HTML.

```
1  @Controller
2  public class PrimeiroController {
3
4   @Inject
5   private Result res;
6
7   public void home() {
8    res.include("titulo", "Título");
9    res.include("msg", "Olá, VRaptor");
10  }
11 }
```

Código 2.15: Exemplo de classe de *controller* do VRaptor

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>${titulo}</title>
5    </head>
6    <body>
7      ${msg}
8    </body>
9  </html>
```

Código 2.16: Exemplo de arquivo de visão do VRaptor

2.3.2 Contexts and Dependency Injection

A especificação JSR 299, *Contexts and Dependency Injection for Java EE plataform* – CDI, define como os *frameworks* de injeção de dependência devem funcionar na plataforma Java (CORDEIRO, 2014). Ela descreve um conjunto de serviços complementares que ajudam a melhorar a estrutura do código de aplicação (KING, 2009). Dentre os serviços definidos, na especificação, estão: mecanismo de injeção de dependências, um modelo de notificação de eventos e a capacidade de adicionar interceptadores a objetos.

2.3.2.1 Mecanismo de Injeção de Dependências

A injeção de dependência (*Dependency Injection* - DI) é uma forma de inversão de controle (*Inversion of Control* - IoC). No controle normal, um objeto é responsável por criar e/ou gerenciar suas dependências, na inversão de controle, o objeto recebe suas dependências inicializadas e prontas para uso e o gerenciamento destas dependências deixa de ser responsabilidade do objeto.

Para exemplificar os conceitos de inversão de controle e injeção de dependência o Código 2.17 e o Código 2.18 apresentam um exemplo, onde as classes não implementam esses conceitos. A classe `ConnectionFactory` é responsável por criar uma conexão com um SGDB MySQL, por meio do *Java Database Connectivity* - JDBC. A classe `AlunoDAO` é responsável por implementar a camada de acesso a dados, ou seja, as operações CRUD.

```
1  public class ConnectionFactory {
2
3      private static final String URL = "jdbc:mysql://localhost/mydb";
4      private static final String USER = "root";
5      private static final String PWD = "password";
6      private ConnectionFactory(){}
7
8      public static Connection getConnection() throws SQLException{
9       return DriverManager.getConnection(URL, USER, PWD);
10     }
11 }
```

Código 2.17: Fábrica de conexões sem CDI

```
1  public class AlunoDAO {
2      private Connection connection;
3
4      public AlunoDAO() throws SQLException{
5          connection = ConnectionFactory.getConnection();
6      }
7      public void adicionar(Aluno a){}
8      public void remover(Aluno a){}
9      public void buscar(Integer id){}
10     public void atualizar(Aluno a){}
11
12     public void fechar() throws SQLException{
13         connection.close();
14     }
15 }
```

Código 2.18: Classe de acesso a dados sem CDI

Para implementar as operações CRUD, a classe `AlunoDAO` depende de uma conexão, instância de `Connection`, assim seu método construtor, na linha 5, obteve uma conexão, por meio da classe `ConnectionFactory`.

Uma vez que, a classe `AlunoDAO` criou uma nova conexão, ela também deve ser responsável por fechá-la, liberando assim os recursos alocados. Como o Java não possui um método destrutor, não é possível determinar quando a conexão deve ser finalizada, dessa forma, foi criado um método `fechar()` na linha 12.

A implementação do método `fechar()` apresenta diversos problemas, conforme apresentado em Silveira (2012), sendo eles: 1) Espalhamento de Responsabilidade: A classe `AlunoDAO` criou a conexão, mas não sabe quando fechá-la, assim ela delega a função a quem a utiliza, espalhando a responsabilidade pelo sistema; 2) Não há garantia da invocação do método: Como a responsabilidade, de fechar a conexão, está espalhada pela

aplicação é possível que os utilizadores da classe `AlunoDAO` esqueçam de encerrar a conexão, isso não acarretaria erro de compilação, mas causaria diversos problemas, quando muitas conexões estivessem abertas; 3) Complexidade e tratamento de erros: O encerramento de uma conexão pode ser mais complexo do que a simples invocação do método `fechar()`, pode ser necessário tratar exceções e realizar operações de *rollback* e 4) Quebra de Encapsulamento: O método `fechar()` existe apenas nas implementações de DAO que utilizam banco de dados, caso a implementação de DAO fosse substituída por uma que utilizasse outra fonte de dados, como por exemplo XML, não seria necessário fechar a conexão, no entanto, ainda assim o método deveria ser implementado.

A classe `AlunoDAO` está no controle, visto que ela própria criou suas dependências, no entanto, isso gerou diversos problemas, pois a ela foi atribuída muitas responsabilidades. A responsabilidade de criar e fechar as conexões não deveria ser atribuição da classe `AlunoDAO`, essa responsabilidade deve ser atribuída a outra classe. O Código 2.19 e o Código 2.20 apresentam o mesmo exemplo utilizando a injeção de dependência, da especificação CDI.

A classe `AlunoDAO`, Código 2.19, não cria mais a conexão e não possui o método para fechá-la, ela simplesmente recebe a conexão pronta para uso. Essa injeção de dependência é realizada com o tipo de anotação `@Inject` da linha 3. A criação da instância de `Connection`, sua atribuição e destruição é realizada pelo `framework` de injeção de dependências. Qualquer `framework` de injeção de dependências pode ser utilizado, desde que ele atenda as especificações da CDI - JSR 299 (KING, 2009). A classe `AlunoDAO` agora está mais coesa, uma vez que suas responsabilidades são apenas acessar e persistir dados.

```
1  public class AlunoDAO {
2
3      @Inject
4      private Connection connection;
5
6      public void adicionar(Aluno a){}
7      public void remover(Aluno a){}
8      public void buscar(Integer id){}
9      public void atualizar(Aluno a){}
10 }
```

Código 2.19: Classe de acesso a dados com CDI

O Código 2.20 apresenta a classe `ConnectionFactory` com uso do CDI.

```
1  @ApplicationScoped
2  public class ConnectionFactory {
3
4      private final String URL = "jdbc:mysql://localhost/mydb";
5      private final String USER = "root";
6      private final String PWD = "vertrigo";
7
8      public ConnectionFactory(){}
9
10     @Produces
11     @RequestScoped
12     public Connection getConnection() throws SQLException{
13         return DriverManager.getConnection(URL, USER, PWD);
14     }
15     public void fechar(@Disposes Connection connection) throws SQLException {
16         if (!connection.isClosed()) {
17             connection.close();
18         }
19     }
20 }
```

Código 2.20: Fábrica de conexões com CDI

A classe `ConnectionFactory` agora é responsável por criar e fechar as conexões. Nela foram utilizados os tipos de anotação `@ApplicationScoped`, `@Produces`, `@RequestScoped` e `@Disposes`. Os tipos de anotação `@ApplicationScoped` e `@RequestScoped` definem o escopo de um objeto. O escopo é o tempo de vida de um objeto, ele é utilizado para definir quando um objeto será criado e descartado pelo *framework* de injeção de dependências. A classe `ConnectionFactory` foi anotada com `@ApplicationScoped`, assim será criado apenas um objeto, desta classe, para toda a aplicação, semelhante ao padrão de projeto *singleton*.

O tipo de anotação `@Produces` é utilizado para criar objetos complexos. Objetos complexos são aqueles que não são criados com um construtor padrão. No caso, o objeto `Connection` não é criado com um construtor padrão, para criá-lo é necessário invocar o método `getConnection`, de `DriverManager`, passando com parâmetro a localização do banco (`URL`), o usuário (`USER`) e a senha (`PWD`). O método `getConnection()`, linha 12, é responsável por criar esse objeto complexo, por esta razão ele foi anotado com `@Produces`, linha 10, que indica que esse método é responsável por produzir o objeto complexo.

O tipo de anotação `@RequestScoped`, linha 11, também foi aplicado ao método `getConnection()`, de forma que os objetos `Connection` possuam escopo de requisição. Escopo de requisição, ou `@RequestScoped`, indica que será criado um objeto para cada requisição, ao final da requisição o objeto é descartado.

O método `fechar()` foi retirado da classe `AlunoDAO` e adicionado a classe `ConnectionFactory`. Para que a classe `ConnectionFactory` saiba o momento correto de fechar a conexão, o parâmetro `Connection`, do método `fechar()`, linha 15, foi anotado com o tipo `@Disposes`. Como o escopo da instância `Connection` é `@RequestScoped` ele será descartado no final da requisição. O tipo `@Disposes` observa o momento que o descarte será realizado e antes que isso ocorra invoca o método `fechar()`.

A inversão de controle e a injeção de dependências possibilita a criação de códigos mais coesos e com baixo acoplamento. A coesão refere-se a responsabilidades de um objeto. Um objeto só deve implementar ações que são de sua responsabilidade. O acoplamento refere-se à interligação de objetos, quando um objeto utiliza outro e quando sua modificação afeta o comportamento do outro tem-se um acoplamento. A redução do acoplamento é conseguida com encapsulamento e fazendo o bom uso de interfaces (SILVEIRA, 2012). Além disso, o modelo de notificação de eventos, do CDI, também favorece a redução de acoplamento.

2.3.2.2 Modelo de notificação de eventos

O modelo de notificação de eventos do CDI é utilizado para reduzir o acoplamento entre objetos, ele possibilita que um aviso seja disparado para todo o sistema quando uma ação ocorrer. O aviso pode ser observado por outros elementos do sistema, que então, podem realizar suas tarefas.

A Figura 2.13 apresenta um diagrama de classes para um sistema de autuação de trânsito.

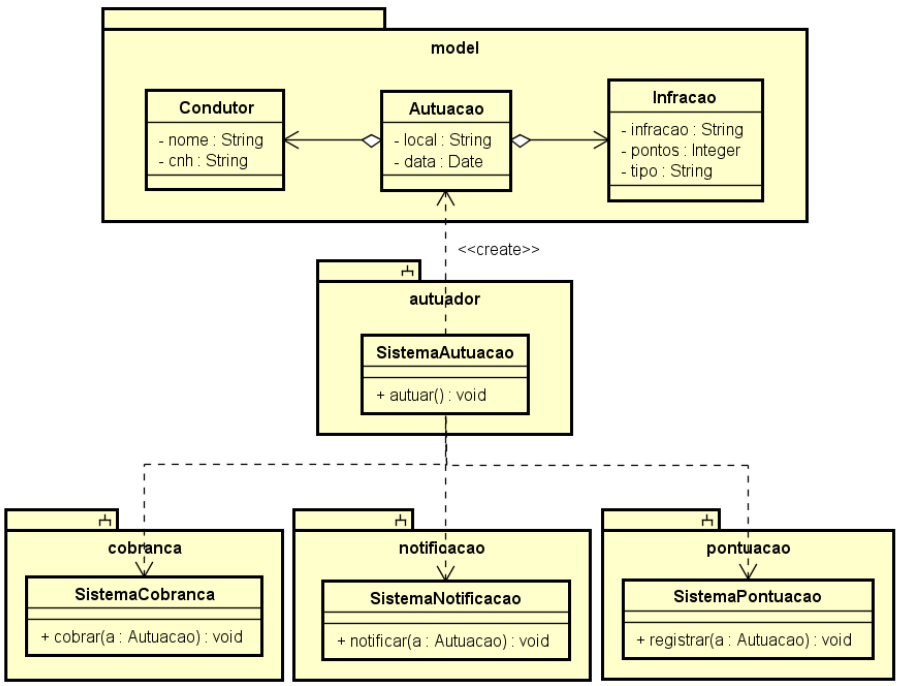

Figura 2.13: Digrama de classes: Sistema de autuação de trânsito

Uma autuação é realizada a um `Condutor` que cometeu uma `Infracao` de trânsito. Após uma `Autuacao` ser registrada pelo subsistema de autuação, os subsistemas de cobrança, de notificação e de pontuação devem ser chamados para, respectivamente, realizar a cobrança da multa, enviar uma notificação ao condutor e para registrar os pontos na Carteira Nacional de Habilitação - CNH. Para simplificar o exemplo, estamos considerando que a autuação já é a própria multa, dessa forma o condutor não possui direito de recurso.

A classe `SistemaAutuacao` depende das instâncias de `SistemaCobranca`, `SistemaNotificacao` e `SistemaPontuacao`. O Código 2.21 apresenta um exemplo de implementação da classe `SistemaAtuacao`, sem o uso de eventos. Nas linhas de 4 a 9 são injetadas as dependências. No método `autuar`, linhas 12 e 13, são criadas as instâncias de `Condutor` e de `Infracao`. Na linha 14 uma instância de `Autuacao` é criada e as linhas de 29 a 31 os subsistemas de `SistemaCobranca`, `SistemaNotificacao` e `SistemaPontuacao` são chamados passando a instância de `Autuacao`.

```
1   @Controller
2   public class SistemaAutuacao {
3
4   @Inject
5   private SistemaNotificacao sistemaNotificacao;
6   @Inject
7   private SistemaCobranca sistemaCobranca;
8   @Inject
9   private SistemaPontuacao sistemaPontuacao;
10
11  public void autuar(){
12      Condutor c = new Condutor("Ivan");
13      Infracao i = new Infracao("Transitar acima da velocidade", 4, "média");
14      Autuacao autuacao = new Autuacao(c, i);
15      sistemaNotificacao.notificar(autuacao);
16      sistemaCobranca.cobrar(autuacao);
17      sistemaPontuacao.pontuar(autuacao);
18  }
19 }
```

Código 2.21: Classe `SistemaAutuacao` sem uso de eventos

O Código 2.22 apresenta uma implementação ingênua da classe `SistemaNotificacao`, sem o uso de eventos. Basicamente essa classe deve possuir um método `notificar`, linha 3, que recebe por parâmetro uma autuação. Esse método deve implementar a lógica para expedir uma notificação ao condutor.

```
1  public class SistemaNotificacao {
2
3      public void notificar(Autuacao autuacao){
4          Condutor condutor = autuacao.getCondutor();
5          Infracao infracao = autuacao.getInfracao();
6          System.out.println("Condutor: " + condutor.getNome());
7          System.out.println("Infração: " + infracao.getInfracao());
8      }
9  }
```

Código 2.22: Classe `SistemaNotificacao` sem evento

Como é possível observar, a classe `SistemaAutuacao` depende das classes `SistemaCobranca`, `SistemaNotificacao` e `SistemaPontuacao`, dessa forma, existe um acoplamento entre elas. Modificações nessas classes podem interferir no funcionamento da classe `SistemaAutuacao`, causando erros ou inconsistências.

Utilizando o notificador de eventos do CDI essas dependências poderiam ser removidas, reduzindo o acoplamento entre as classes. O Código 2.23 apresenta a implementação da classe `SistemaAutuacao` utilizando eventos. Para criar o evento, na linha 17, foi injetada uma instância que implementa a interface `Event`, essa instância é parametrizada com o tipo de evento a ser gerado, neste caso `Autuacao`. A invocação do método `fire`, linha 23, faz a notificação, para os demais subsistemas, de que uma autuação foi criada, ou seja, de que um evento ocorreu. É possível observar que a classe `SistemaAutuacao` não está mais acoplada as classes de `SistemaCobranca`, `SistemaNotificacao` e `SistemaPontuacao`, uma vez que não possui mais essas dependências.

```
1  @Controller
2  public class SistemaAutuacao {
3
4      @Inject
5      private Event<Autuacao> event;
6
7      public void autuar(){
8          Condutor c = new Condutor("Ivan");
9          Infracao i = new Infracao("Transitar acima da velocidade", 4, "média");
10         Autuacao autuacao = new Autuacao(c, i);
11         event.fire(autuacao);
12     }
13 }
```

Código 2.23: Classe `SistemaAutuacao` com uso de eventos

Para que os subsistemas possam observar os eventos disparados, é necessário adicionar o tipo de anotação `@Observes` ao parâmetro do método que irá tratar o evento. O Código 2.24 apresenta a implementação da classe `SistemaNotificação`, com uso de eventos do CDI. Para observar as autuações geradas pela classe `SistemaAutuacao`, o parâmetro `Autuacao autuacao`, do método `notificar`, da linha 3, foi anotado com `@Observes`, com isso, toda vez que um evento for disparado, o método `notificar` será executado.

```
1  public class SistemaNotificacao {
2
3  public void notificar(@Observes Autuacao autuacao){
4          Condutor condutor = autuacao.getCondutor();
5          Infracao infracao = autuacao.getInfracao();
6          System.out.println("Condutor: " + condutor.getNome());
7          System.out.println("Infração: " + infracao.getInfracao());
8      }
9  }
```

Código 2.24: Classe `SistemaNotificacao` com evento

Além de reduzir o acoplamento, o sistema de notificação de eventos do CDI possibilita realizar a extensão do código e das funcionalidades do sistema, sem ter que modificar as classes que já estão criadas, e que algumas vezes não se tem acesso. Por exemplo, caso fosse necessário criar um novo subsistema, que fosse chamado toda vez que uma autuação fosse gerada, não seria necessário modificar a classe `SistemaAutuacao`, bastaria observar os eventos disparados por ela.

2.3.2.3 Interceptadores

Um *Interceptor*, ou interceptador, permite executar tarefas antes e/ou depois de uma lógica de negócio. Para isso, ele utiliza os conceitos do paradigma de Programação Orientada a Aspectos.

Três conceitos são relevantes para a compreensão do paradigma de Orientação a Aspectos: os *joinpoints*, *pointcut* e *advices*. Um *joinpoint* corresponde a pontos bem definidos no fluxo de execução de um sistema, podendo ser, por exemplo, a execução de um método, o acesso a membros de uma classe ou criação de objetos. Esses pontos bem definidos podem ser marcados para que uma ação seja realizada antes, durante ou depois daquele ponto. Essa marcação é conhecida como *pointcut*, ou ponto de corte, onde a execução normal é interrompida para executar outra ação, essa outra ação é conhecida como *advice*.

O Código 2.25 apresenta um exemplo de interceptador para criar *logs*, quando um método for invocado.

```
1  @Interceptor
2  @Log
3  @Priority(Interceptor.Priority.APPLICATION)
4  public class Logs {
5
6      @Inject
7      private Logger logger;
8
9      @AroundInvoke
10     public Object makeLog(InvocationContext context) throws Exception{
11         logger.info("Executa uma tarefa antes");
12         //executa o método interceptado
13         Object retorno = context.proceed();
14         logger.info("Executa uma tarefa após");
15         return retorno;
16     }
17 }
```

Código 2.25: Exemplo de Interceptador

Para criar um interceptador é necessário criar uma classe e anotá-la com o tipo de anotação `@Interceptor`, conforme mostrado na linha 1. O tipo de anotação `@Log`, linha 2, foi criado no Código 2.26, ele é utilizado para vincular o interceptador aos pontos de interceptação. No CDI essa anotação é conhecida como *interceptor bindings*, seu objetivo é definir quem será interceptado (*pointcut*) e qual interceptador irá responder pela interceptação. A classe que trata a interceptação e os *pointcuts* devem estar anotados com `@Log`.

O tipo de anotação `@AroundInvoke`, linha 9, é utilizado para anotar o método responsável por executar uma ação antes e/ou depois de uma interceptação. O método anotado com `@AroundInvoke` corresponde ao *advice* da Programação Orientada a Aspectos.

O parâmetro `context`, passado para o método `makeLog`, linha 10, possibilita recuperar qual método foi interceptado, quais parâmetros foram passados ao método, a qual classe o método pertence, além de prosseguir com a execução do método interceptado.

Tendo em vista a possibilidade de criar diversos interceptadores o tipo de anotação `@Priority`, linha 3, é utilizada para definir a ordem na qual os interceptadores serão carregados.

O Código 2.26 apresenta a definição do tipo de anotação `@Log`, utilizado para vincular os *pointcuts* a um interceptador. A definição deste tipo de anotação é semelhante a outros tipos, exceto pelo uso da metaanotação `@InterceptorBinding`, linha 4.

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target({ElementType.TYPE, ElementType.METHOD})
3 @Inherited
4 @InterceptorBinding
5 public @interface Log {
6
7 }
```

Código 2.26: Anotação para vinculação de interceptador

O Código 2.27 apresenta a utilização do tipo de anotação `@Log`, utilizado para interceptar um método e gerar *logs*. Para realizar a interceptação de um método, basta utilizar o tipo de anotação `@Log`, conforme mostrado na linha 4. Quando o método `login` for invocado, a execução será interrompida para que o método `makeLog`, linha 10, do Código 2.25 seja executado.

```
1  @Controller
2 public class UsuarioController {
3
4     @Log
5     public void login(){
6         System.out.println("Faz o login");
7     }
8 }
```

Código 2.27: Classe `UsuarioController`: definição de um ponto de interceptação

Os interceptadores normalmente são utilizados para implementação de requisitos transversais. Requisitos transversais, também conhecidos como requisitos não funcionais, são aqueles relacionados a características de qualidade, como segurança, desempenho e escalabilidade.

2.3.3 Bean Validation API

A especificação JSR 303, *Bean Validation API*, é o mecanismo padrão para validação de JavaBean, da plataforma Java EE 6. Ela define um modelo de metadados e uma API para validação (BERNARD; PETERSON, 2009).

O modelo de metadados corresponde aos tipos de anotações disponíveis para impor as restrições de validação. Os tipos de anotações integrados para validação são:

- *@NotNull*: O elemento anotado não pode ser nulo.
- *@Size*: O elemento anotado deve estar entre os limites máximo e mínimo especificados.
- *@AssertFalse*: O elemento anotado deve ser falso (`false`).
- *@AssertTrue*: O elemento anotado deve ser verdadeiro (`true`).
- *@DecimalMax*: O elemento anotado deve ser um número, cujo valor deve ser igual ou menor o valor máximo especificado.
- *@DecimalMin*: O elemento anotado deve ser um número, cujo valor deve ser igual ou maior o valor mínimo especificado.
- *@Digits*: O elemento anotado deve ser um número dentro da faixa aceita.
- *@Future*: O elemento especificado deve ser uma data no futuro.
- *@Past*: O elemento especificado deve ser uma data no passado.
- *@Max*: O elemento anotado deve ser um número, cujo valor deve ser igual ou menor o valor máximo especificado.
- *@Min*: O elemento anotado deve ser um número, cujo valor deve ser igual ou maior o valor mínimo especificado.
- *@Null*: O elemento anotado deve ser nulo.
- *@Pattern*: O elemento anotado deve corresponder a uma expressão regular especificada.

Caso os tipos de anotação, pré-definidos na *Bean Validator API*, não atendam às necessidades de validação do desenvolvedor, a API permite que o desenvolvedor crie seus próprios tipos de anotações de validação. O Código 2.28 apresenta um exemplo de código para criar um tipo de anotação para validação de objetos.

```
1 @Target({ ElementType.FIELD })
2 @Retention(RetentionPolicy.RUNTIME)
3 @Constraint(validatedBy = {CPFValidator.class })
4 public @interface ValidCPF {
5
6     String message() default "O CPF é inválido";
7     Class<?>[] groups() default {};
8     Class<? extends Payload>[] payload() default {};
9 }
```

Código 2.28: Tipo de anotação para validação de dados

A linha 3 vincula a anotação com a classe que irá realizar o processamento da validação, neste caso será a classe `CPFValidator.class`. A linha 4 define o nome do tipo de anotação, a linha 6 define o elemento `message` que corresponde a mensagem de erro apresentada caso a validação falhe, a linha 7 possibilita agrupar a validação de elementos e a linha 8 permite que os desenvolvedores associem informações nos tipos de anotação de validação.

Uma vez criado o tipo de anotação é necessário criar uma classe, que irá realizar a validação dos dados, essa classe deve implementar a interface `ConstraintValidator`. O Código 2.29 apresenta a classe `CPFValidator.class` utilizada para realizar a validação do tipo de anotação `@ValidCPF`, criado no Código 2.28. O método `isValid` da linha 5 deve retornar verdadeiro (`true`) ou falso (`false`), se o valor informado, na criação do objeto, for válido ou não. No Código 2.29 não foi implementado um código funcional de validação de CPF, apenas foi verificado se o valor não era nulo, não estava vazio e se possuía 11 caracteres.

```
1 public class CPFValidator implements ConstraintValidator<ValidCPF, String>{
2     @Override
3     public void initialize(ValidCPF constraintAnnotation) { }
4     @Override
5     public boolean isValid(String cpf, ConstraintValidatorContext context) {
6         // código para validar o CPF
7         return cpf != null && !cpf.isEmpty() && cpf.length() == 11 ;
8     }
9 }
```

Código 2.29: Classe para validação de dados

O Código 2.30 apresenta a utilização dos tipos de anotação para validação de dados.

```
1  public class Aluno {
2      @NotNull
3      private Integer matricula;
4
5      @NotNull
6      @Size(min = 7, message = "O nome não pode ser menor do que 7 caracteres")
7      private String nome;
8
9      @Past
10     private Date dataNascimento;
11
12     @ValidCPF
13     private String cpf;
14     }
```

Código 2.30: Classe `Aluno` com restrições de validação

As linhas 2 e 3 informam que o atributo `matricula` não pode ser nulo, ou seja, seu preenchimento é obrigatório. O atributo `nome`, da linha 7, utiliza os tipos de anotação `@NotNull` e `@Size`, desta forma o valor do atributo não pode ser nulo e não pode ser inferior a 7 caracteres. O elemento `message` do tipo de anotação `@Size` foi configurado para alterar a mensagem exibida no caso de falha da validação. O atributo `dataNascimento`, linha 10, utiliza a anotação `@Past`, informando que a data informada deve ser anterior a data atual e o atributo `cpf`, linha 13, utiliza a anotação criada no Código 2.28.

A API de validação contém as interfaces que descrevem como validar um objeto. A interface `ConstraintViolation` descreve uma única falha de restrição, já a interface `Validator` é responsável por validar o objeto e retornar todas violações. O Código 2.31 apresenta um exemplo de uso da API de validação. Na linha 4 é injetada a dependência de `Validator`. As linhas de 10 a 14 criam o objeto aluno e definem valores para seus atributos. A linha 16 valida o objeto aluno e adiciona as violações de restrição na variável `restricoes`. Por fim, a linha 18 percorre o conjunto de violações, armazenados na variável `restrições` e a linha 19 imprime as mensagens de erro.

```
1  @Controller
2  public class AlunoViolationController {
3
4      @Inject private Validator validator;
5
6      public void valida(){
7          Calendar c = Calendar.getInstance();
8          c.set(1986, 4, 26);
9
10         Aluno a = new Aluno();
11         a.setMatricula(1);
12         a.setNome("Ivan");
13         a.setCpf("123");
14         a.setDataNascimento(c.getTime());
15
16         Set<ConstraintViolation<Aluno>> restricoes = validator.validate(a);
17
18         for(ConstraintViolation<Aluno> erro : restricoes){
19             System.out.println("Aconteceu" + erro.getMessage());
20         }
21     }
22 }
```

Código 2.31: Exemplo de validação usando a *Bean Validation API*

2.4 Trabalhos Similares

Ao longo dos últimos anos diversas abordagens para geração de UIs, baseadas em modelos, foram propostas na literatura, além disso, outras ferramentas foram produzidas e consumidas pelo mercado de desenvolvimento. O paradigma de Engenharia Dirigida por Modelo (MDE) define duas formas de automatização de modelos, para sistemas executáveis: a transformação e a execução.

Na transformação os modelos passam por processos de transformação/conversão que produzem novos artefatos ou modelos específicos. No caso da geração das UIs, os modelos são processados por ferramentas de transformação, que geram os código-fonte das UIs. Alguns dos trabalhos encontrados na literatura e ferramentas que pertencem a esta categoria são:

- *Metaffolder*: Em Magno (2015) foi implementado uma ferramenta que aplica a técnica de *scaffolding* para gerar as UIs CRUD para sistemas *web*. A ferramenta utiliza as classes de domínio e um conjunto de anotações para definir como os *widgets* serão apresentados na UI. Uma vez criado os modelos, a ferramenta é executada, por meio do terminal de comandos, neste momento ela faz análise do modelo e gera automaticamente os códigos das camadas de visão e controle. No desenvolvimento da ferramenta foram utilizadas as especificações Java: *Java Persistence API*, *Bean Validation API*, *Java Reflection API*, além das linguagens HTML, *JavaScript* (*JQuery*) e CSS. Para validação da ferramenta foi realizado um experimento com 10 participantes. No experimento foi verificado a produtividade e a usabilidade da ferramenta. Na produtividade foi medido o tempo gasto por cada participante para criar um sistema CRUD. Os tempos foram comparados com o desenvolvimento do mesmo sistema codificado manualmente e com a aplicação da técnica de *scaffolding* no *framework Rails*. Para avaliação da produtividade foi aplicado o questionário *System Usability Scale* - SUS. Os resultados dos experimentos mostraram que o *Metaffolder* é cerca de 91% mais produtivo do que a codificação manual, e que o *framework Rails* é cerca de 17% mais produtivo do que o *framework Metaffolder.* Quanto a usabilidade o *framework Metaffolder* obteve 83,7 no questionário SUS e o *Rails* obteve 86 pontos, o que demostrou que as duas ferramentas possuem boa usabilidade.

- *JavaWebCreator*: Em Silva et al. (2011) foi implementada uma ferramenta para geração de UIs CRUD para plataforma Java *web*. A ferramenta utiliza como modelo, os *scripts* de criação de tabelas de banco de dados relacionais (*Data Definition Language* - DDL). Ao executar a ferramenta, o desenvolvedor deve fornecer os códigos DDLs e realizar algumas configurações, configurações estas que possibilitam modificar poucas características relacionadas a apresentação da UI, como por exemplo, o rótulo (*label*) de um *widget*. Ao final a ferramenta gera os arquivos de códigos, que então

devem ser copiados para um projeto Java *web* MVC. Para o desenvolvimento das UIs foram utilizados os *servlets* para gerar a camada de controle e o JSP e o *JQuery* para construir a camada de visão, na camada de persistência foi utilizado *Hibernate* e a validação de dados é realizada apenas no *client-side* com *plugins* do *JQuery*. Este trabalho não realizou experimentos para demostrar o aumento de produtividade e para avaliar a usabilidade das UIs geradas pela ferramenta.

- *Rails*: é o *framework* precursor da técnica de *scaffolding*. Ele possibilita a geração de UIs CRUD para plataforma *web*, utilizando a linguagem Ruby. A geração do *scaffolding* ocorre apenas de forma estática. Neste *framework*, o desenvolvedor não cria, manualmente, os modelos. A criação dos modelos e a geração das camadas de visão e controle ocorrem diretamente pelo terminal de comandos. Assim como os demais *scaffoldings* estáticos, a personalização das UIs e a inclusão do código individual para aplicação são realizados manualmente (HANSSON, 2003).

- *Ambiente Integrado de Desenvolvimento – IDEs*: Grande parte das modernas IDE possibilitam a geração de UIs CRUD, com base em modelos. A IDE *Netbeans* possibilita a geração de UIs utilizando como modelo as tabelas de bancos de dados relacionais ou classes de domínio (entidades JPA). As UIs podem ser geradas para a plataforma Java Desktop (*Swing*) e para plataforma web (*JavaServe Faces*) (NETBEANS, 2010). A IDE *Intellij IDEA* possibilita a geração das UIs utilizando, como modelo, as classes de domínio (entidades JPA) (MALYSHEV, 2006). O Eclipse IDE não possibilita, nativamente, a construção de UIs CRUD, no entanto, com a instalação de *plugins* no IDE é possível realizar a geração (SANCHEZ, 2008).

Na abordagem de automatização de modelos por execução, os modelos são interpretados e as UIs são geradas dinamicamente. Alguns trabalhos relacionados a esta abordagem encontrados na literatura são:

- *Merlin*: O trabalho de Mrack et al. (2006) utiliza uma abordagem semelhante a proposta neste trabalho. A ferramenta desenvolvida no trabalho utiliza as classes de domínio e anotações para definir a apresentação de *widgets* e das UIs. O modelo de tarefas, utilizado pela ferramenta, consiste no uso de métodos da classe de modelo ou de outras classes, que podem ser executados. As tarefas, suas precondições e pós-condições estão relacionadas a eventos dos *widgets*, como por exemplo, quando um campo de texto perde o foco de uma validação é realizada. A ferramenta foi construída para gerar UIs para *desktop* na plataforma Java *Swing*. Um dos grandes pontos fracos, desta ferramenta, é a impossibilidade de gerar UIs para classes que possuam associações com multiplicidade um para muitos e de muitos para muitos. Este trabalho acadêmico também não realizou experimentos para validar a ferramenta desenvolvida.

- *GDIG*: Em Silva (2010) foi implementada uma ferramenta para construção de UIs para sistemas *desktop* com Java *Swing*. Este trabalho apresenta uma abordagem diferente dos demais, visto que os modelos são criados utilizando um editor, semelhante a um formulário de cadastro. A ferramenta utiliza dois tipos de modelo, o modelo de negócio que é formado pelas entidades, atributos e associações e o modelo de apresentação que define como será o aspecto visual da UI para um o modelo de negócio. Uma vez criado os modelos eles são armazenados em banco de dados e a geração das UIs é realizada automaticamente. Este trabalho atua como caixa preta, dessa forma, o desenvolvedor não possui controle sobre o que está sendo gerado, além de não poder estender as funcionalidades do *framework* e de associar o código individual da aplicação as UIs geradas. Neste trabalho foi realizado um experimento para medir o tempo de desenvolvimento e a produtividade da ferramenta, na geração de UIs de pequena, média e grande complexidade. Os resultados foram comparados com editores visuais (*Interfaces Builder*) e com a codificação manual. O experimento contou com a participação de 4 desenvolvedores. Os resultados dos experimentos apresentaram que a ferramenta GDIG é, no mínimo, quinze vezes mais rápida que a codificação manual e cerca de seis vezes mais rápida do que a geração auxiliada por editores visuais.

- *Grails*: como apresentado anteriormente, é um *framework* MVC que implementa os dois tipos de *scaffolding*, estático e dinâmico. Independentemente do tipo, a geração das UIs inicia com a construção das classes de domínio. Na transformação, *scaffolding* estático, a ferramenta é executada no terminal de comandos e os arquivos de controle e visão são gerados automaticamente. Na execução, *scaffolding* dinâmico, a classe de controle deve ser criada com o atributo estático `scaffold`, em tempo de execução o modelo é interpretado e a UI é gerada automaticamente (ROCHER et al., 2014).

- *Play*: É um *framework web* MVC, que dispõe de um *scaffolding* dinâmico que possibilita a geração de UIs CRUD. Este *framework* utiliza as classes Java, como modelo de domínio para geração das UIs. A personalização da aparência das UIs deve ser realizada com a edição dos arquivos de *template* do *framework* (TYPESAFE, 2014).

As ferramentas que fazem uso da abordagem de transformação para construir as UIs CRUD são mais flexíveis, pois possibilitam que o desenvolvedor personalize as UIs geradas e implementem novas funcionalidades. No entanto, toda vez que os modelos são evoluídos, é necessário executar novamente a ferramenta de transformação, que sobrescreve os arquivos de código da UIs, gerados anteriormente, que possivelmente haviam sido modificados manualmente pelo desenvolvedor. A modificação manual dos códigos gerados pela ferramenta tende a ser custosa e algumas vezes complexa.

As ferramentas que fazem uso da abordagem de execução são menos flexíveis, pois o desenvolvedor, geralmente, não pode adaptar ou reescrever os códigos das UIs, entretanto,

as modificações realizadas nos modelos são aplicadas instantaneamente nas UIs. De forma geral, as UIs produzidas pelos *scaffoldings* e pelas IDEs são básicas demais, de forma que o desenvolvedor deve realizar algum tipo de recodificação manual para atender as necessidades do sistema em desenvolvimento. As ferramentas MBUIDEs, como por exemplo o Merlin, por utilizar diferentes tipos de modelo, possibilitam construir UIs mais ricas, além de ser flexível e gerar as UIs em tempo de execução.

Das ferramentas elencadas nenhuma possibilita a geração de UIs baseadas em contexto de uso, utilizando modelos. Para gerar diferentes tipos de UIs, para grupos específicos de usuário, as ferramentas da abordagem de transformação exigem que os desenvolvedores codifiquem essas UIs manualmente.

2.5 Considerações Finais

Este Capítulo apresentou o paradigma de Engenharia Dirigida por Modelo (MDE), bem como suas principais abordagens, o *Model-Driven Architecture* (MDA), o *Model-Driven Development* (MDD) e o *Model-Based User Interface* (MBUID). De forma geral, essas abordagens utilizam diversos tipos de modelo para descrever o sistema a ser construído. Por meio das ferramentas de automatização, tais como os geradores de código ou interpretadores de modelos, os sistemas ou suas partes, são construídos automaticamente. Essas abordagens prometem trazer diversos benefícios para o desenvolvimento de sistemas, dentre eles a produtividade, a qualidade e o desempenho dos sistemas gerados.

Foi introduzida a técnica de *scaffolding* que possibilita a criação automática de artefatos de *software*, principalmente as UI para persistência de dados (CRUD). Essa técnica teve boa aceitação do mercado e tem sido empregada em diversos *frameworks* MVC. Ela pode ser classificada como uma abordagem *Model-Driven Development* (MDD), tendo em vista que ela aplica técnicas de automatização/transformação nos modelos de domínio, mas não pode ser considerada uma abordagem MDA, pois não utiliza os padrões da OMG e não visa a reusabilidade de modelos e a interoperabilidade e portabilidade de plataforma. Os *frameworks* de *scaffolding* também podem ser considerados MBUIDEs, visto que atendem os critérios definidos por Schlungbaum (1996), ao utilizar modelos abstratos de alto nível e por realizar a automatização desses modelos em UI. Entretanto, como essas ferramentas geralmente utilizam apenas o modelo de domínio, a representação da interação do usuário com a interface fica comprometida, com isso as UIs geradas acabam por serem básicas demais forçando o desenvolvedor a fazer ajustes manuais no código gerado.

Tendo em vista que um dos pontos chave do MDE são as ferramentas de automatização, foi introduzido o paradigma de metaprogramação que possibilita a construção desse tipo de ferramenta. As anotações possibilitam adicionar marcas nos modelos (código-fonte) e

por meio da metaprogramação e da reflexão é possível extrair esses metadados e gerar novos sistemas ou partes dele.

Este trabalho propõe o desenvolvimento uma ferramenta de automatização de modelos, baseada nas abordagens MDD e MBUID, implementada em Java e utilizando metaprogramação. A ferramenta possui similaridades com a técnica de *scaffolding*, no entanto, faz uso de mais tipos de modelo, fornecendo assim mais funcionalidades e flexibilidade para o desenvolvedor.

A ferramenta foi construída para gerar UI CRUDs para o *framework* MVC *VRaptor*. O *VRaptor* é um *framework* MVC brasileiro, bem aceito pela comunidade de desenvolvedores e atualmente está na versão 4. Nas versões anteriores, o *framework*, possuía o recurso de *scaffolding*, que foi removido dessa versão, uma vez que o *framework* foi reescrito para utilizar a especificação Java CDI. Apesar da ferramenta ter sido construída para o *VRaptor* a ideia por trás dela pode ser implementada em outros *frameworks* MVC.

Capítulo 3

O framework ObCrud

Este capítulo apresenta a principal contribuição deste trabalho, que é o desenvolvimento do *framework ObCrud*. O *framework* tem como objetivo realizar a geração de *UIs CRUD* para sistemas orientados a negócio. Para geração das UIs, o *ObCrud*, faz uso dos Modelos de Domínio, de Tarefa e de Apresentação. No desenvolvimento do *framework* foram empregados os principais conceitos das abordagens dirigidas por modelo e a técnica de metaprogramação apresentada na fundamentação teórica.

Este capítulo está organizado da seguinte forma: A Seção 3.1 apresenta a arquitetura do *framework ObCrud* apresentando seus subsistemas, responsabilidades e as principais classes e interfaces do *framework*. A Seção 3.2 apresenta o processo de desenvolvimento de UIs utilizando o *framework* e diversos casos de uso. A Seção 3.3 apresenta as considerações finais do capítulo.

3.1 Arquitetura lógica

A arquitetura lógica de um sistema corresponde a organização de suas classes em subsistemas (BEZERRA, 2015). Um subsistema fornece ou utiliza serviços de outros subsistemas, por meio de suas interfaces. O *framework ObCrud* foi organizado em 12 subsistemas ou pacotes, conforme apresentado na Figura 3.1.

O subsistema `route` é responsável por criar as rotas de acesso para as *UIs CRUD* e por recuperar e manipular as rotas requisitadas pelos usuários durante a execução do sistema.

O subsistema `processor` é responsável por processar a requisição do usuário. De acordo com a rota e com a requisição, uma das implementações de `RequestProcessor` é selecionada para exibir uma UI ou para realizar operações de persistência dos dados. Como esse subsistema é responsável por processar as requisições do usuário, ele possui o maior número de dependências, uma vez que necessita utilizar os serviços dos demais subsistemas para desempenhar suas funções. O subsistema `processor` utiliza os serviços de `route` para conhecer a rota e a requisição que o usuário efetuou, com base nessas informações uma ação é realizada. Uma das ações que o `processor` pode realizar é a persistência de

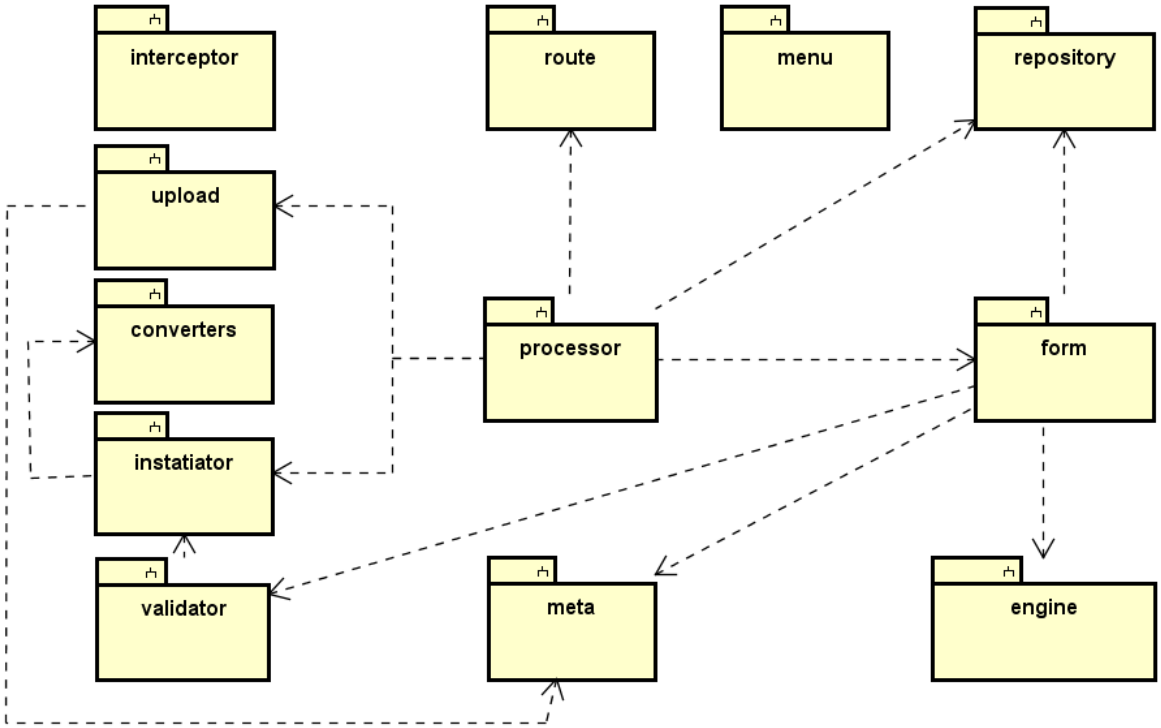

Figura 3.1: Arquitetura lógica do *ObCrud*

dados, para isso ele utiliza os serviços do pacote `repository`, que implementa a lógica de acesso a dados, ele também utiliza os serviços de `instatiator`, pois os dados enviados na requisição devem ser transformados em objetos, para que possam ser persistidos. Alguns dados enviados na requisição podem ser do tipo arquivo, assim o serviço do subsistema `upload` é utilizado para armazenar o arquivo no servidor de aplicação e associar seu caminho ao atributo do objeto ou para associar os `bytes` do arquivo ao atributo do objeto instanciado. Caso a ação do processador seja criar uma UI, então os serviços do subsistema `form` é utilizado.

O subsistema `repository` é responsável por isolar a lógica de acesso a dados do restante da aplicação, este pacote implementa o padrão `Repository` (EVANS, 2004). Evans (2004) apresenta que este padrão possui diversas vantagens, dentre elas: ele é um modelo simples de obter objetos persistentes e de gerenciar seu ciclo de vida, uma vez que ele abstrai a tecnologia empregada no desenvolvimento do aplicativo, permitindo a substituição ou uso de várias fontes de dados; ele permite substituir facilmente uma implementação *dummy* "ingênua", geralmente implementada em memória volátil para realização de testes e ele comunica regras de projeto para acesso aos objetos.

O subsistema `instatiator` é responsável por criar as instâncias que serão manipuladas e persistidas. As instâncias (objetos) são criadas com os dados enviados na requisição, esses dados são tipo `String`. Para que seja possível atribuir os dados a instância, é necessário realizar conversões para os respectivos tipos de atributos da instância. O pacote `converters` possui as classes que realizam a conversão de tipos, elas implementam a in-

terface `Converter` do *framework VRaptor*. A conversão de alguns tipos comuns, incluindo os tipos primitivos, são realizadas pelos próprios conversores do *VRaptor*.

O subsistema `validator` é responsável por recuperar os erros gerados pela implementação da especificação *Bean Validation API* (BERNARD; PETERSON, 2009). Quando uma instância é criada, pelo subsistema `instatiator` com os dados enviados na requisição, erros de validação podem ocorrer. Esses erros são capturados e apresentados nas UIs de forma que o usuário possa corrigi-los e prosseguir com a execução do sistema.

O subsistema `intercetor` é responsável por fornecer o serviço de execução de precondições e pós-condições, garantindo que o código individual da aplicação seja executado, antes ou após a execução de uma ação CRUD. Esse subsistema não depende dos serviços dos demais subsistemas, as classes controladoras dos usuários que utilizam seus serviços.

O subsistema `upload` é responsável por armazenar os arquivos enviados pelas UIs. É possível configurar o local (diretório no servidor) onde os arquivos serão armazenados, no caso do tipo do atributo do modelo ser uma `String`, no caso do tipo de ser um vetor de `bytes` o armazenamento é realizado diretamente na fonte de dados. Este subsistema depende do subsistema `meta`, para obter os atributos do tipo `InputType.FILE`, do objeto que será persistido.

O subsistema `meta` é responsável por aplicar a técnica de metaprogramação e inspecionar os modelos, utilizados pelo *framework* para gerar as UIs CRUD. Nele são definidas as marcas (metadados) que são aplicadas no modelo domínio e correspondem ao modelo de apresentação. Os dois modelos, em tempo de execução, são inspecionados, suas informações são extraídas e um metamodelo é criado e armazenado em um repositório.

O subsistema `engine` fornece uma interface para configuração e uso de *templates engine*. *Template engine*, ou motor de modelos, é a ferramenta que gera texto, ou código-fonte, por meio de arquivos de *templates* (modelos de código) e de dados (contexto). O *ObCrud* utiliza o *Apache Freemaker* como *template engine*. O *Apache Freemaker* é uma excelente ferramenta de *template engine*, no entanto, dadas as especificidades dos projetos e da equipe de desenvolvimento, é possível configurar o *ObCrud* para utilizar outras *engines*.

O subsistema *form* é responsável por construir as UIs e seus *widgets*. Para desempenhar suas funções, ele utiliza os metamodelos (`meta`), as configurações realizadas nos controladores, os possíveis erros de validação (`validator`) e os objetos armazenados no repositório (`repository`). Todas essas informações formam o contexto que, juntamente com os arquivos de *template*, são processados pelo subsistema `engine` que cria o código das UIs.

O pacote `menu` possibilita que o menu principal da aplicação seja criado dinamicamente, para cada método anotado com o tipo `@Crud` um novo *link* é criado no menu.

Para auxiliar na apresentação das características e do comportamento do *framework ObCrud*, a Figura 3.2 apresenta seu diagrama de classes simplificado, que é detalhado nas subseções seguintes.

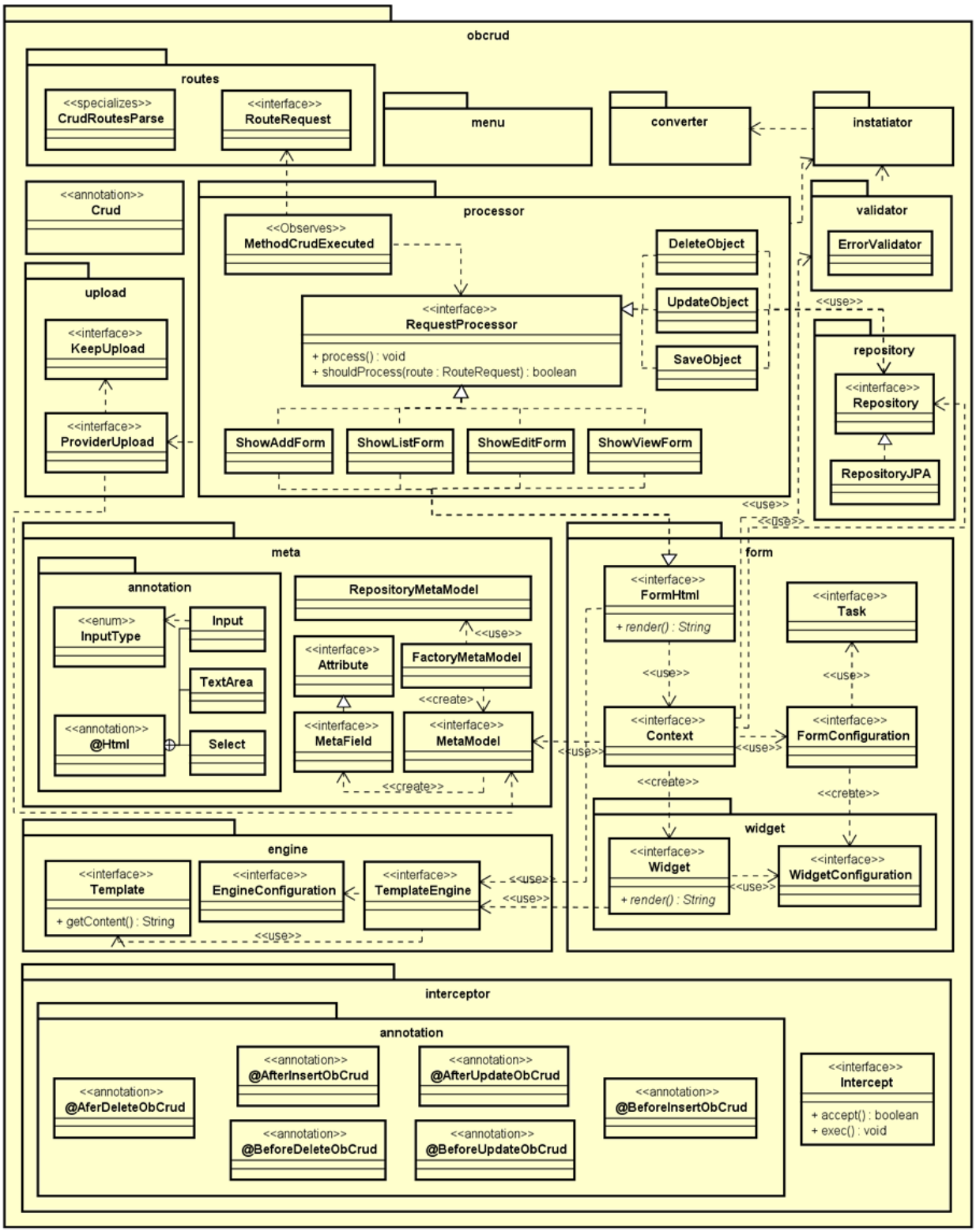

Figura 3.2: Diagrama de Classes do *ObCrud*

3.1.1 Rotas (routes)

O subsistema `routes` basicamente é composto pela classe `CrudRoutesParse` e pela interface `RouteRequest` (Figura 3.2), que expõe os serviços do subsistema. A classe `CrudRoutesParse` estende a classe `PathAnnotationRoutesParser` do *VRaptor*, que é responsável por criar as rotas para cada um dos métodos da classe controladora (*Controller*).

A classe `CrudRoutesParse` sobrescreve o método `registerRulesFor`, da classe `PathAnnotationRoutesParser`, modificando o seu comportamento ao gerar as rotas para os métodos, anotados com `@Crud`, da classe controladora. Neste caso, ao invés de ser criada apenas uma rota, são criadas diversas rotas com diferentes parâmetros e métodos do protocolo do HTTP.

Cada uma das rotas criadas pela classe `CrudRoutesParse` permite acessar uma funcionalidade CRUD específica. O Código 3.1 apresenta a classe `AlunoController` que possui os métodos `normal` e `especializado`. O método `especializado` foi anotado com tipo `@Crud`. A Tabela 3.1 apresenta as rotas geradas para os métodos `normal` e `especializado`.

```
1 @Controller
2 public class AlunoController {
3
4     public void normal(){       }
5
6     @Crud(forClass = Aluno.class)
7     public void especializado(){    }
8
9 }
```

Código 3.1: Classe `AlunoController`: Métodos com e sem anotação `@Crud`

Tabela 3.1: Rotas geradas para os métodos da classe `AlunoController`

Rota	Método HTTP	Caminho (path)	Ação
1	ALL	/aluno/normal	A ser definido pelo desenvolvedor
2	DELETE	/aluno/{aluno.idAluno}	Apaga um aluno com idAluno fornecido.
3	GET	/aluno	Exibe a listagem de todos os alunos.
4	POST	/aluno	Salva o aluno no banco de dados.
5	PUT	/aluno/{aluno.idAluno}	Atualiza o aluno que possuí o idMedico.
6	GET	/aluno/edit/{aluno.idAluno}	Exibe um formulário para editar os dados do aluno que possuí o idAluno informado.
7	GET	/aluno/{aluno.idAluno}	Exibe uma página HTML com os dados do aluno que possuí o aluno o informado.
8	GET	/aluno/add	Exibe um formulário HTML para adicionar um novo aluno.

A rota 1, apresentada na Tabela 3.1, foi gerada pelo `framework` *VRaptor*, uma vez que o método `normal` não está anotado com o tipo `@Crud`. O método `normal` pode ser acessado por todos (`ALL`) métodos do protocolo HTTP, seu acesso é efetuado pelo cami-

nho `/aluno/normal`, que corresponde ao nome da classe controladora, sem o sufixo `Controller` mais o nome do método. As rotas de 2 a 8 foram criadas pelo *framework ObCrud*. O mesmo método `especializado` possui 7 rotas diferentes. Uma rota se difere da outra, pois a combinação do método de acesso do protocolo HTTP e do caminho *path* são únicos para cada rota. Todas as rotas invocam o mesmo método, `especializado`, mas executam funções diferentes.

A utilização dos métodos `GET`, `POST`, `PUT` e `DELETE` do protocolo HTTP possibilita a implementação de *web services* utilizando o padrão *RESTful*. O *Representational State Transfer* (REST) é um estilo arquitetônico, baseado na arquitetura cliente/servidor, no qual as funcionalidades e dados são considerados recursos e são acessados por uma *Uniform Resource Identifiers* (URIs). Este estilo aplicado a serviços web provê diversas propriedades desejáveis, como por exemplo, o desempenho, a escalabilidade e modificabilidade (JENDROCK ERIC, 2013). A arquitetura, do *ObCrud*, foi projetada para suportar o padrão *RESTful*, entretanto, o *framework* atualmente disponibiliza apenas a execução das funcionalidades CRUD.

A interface `RouteRequest` possui os métodos para recuperar as informações da rota que foi requisitada pelo usuário do sistema, essas informações são utilizadas pelo subsistema de processamento de requisição.

3.1.2 Processador de requisição (processor)

O subsistema `processor` é composto pela classe `MethodCrudExecuted`, pela interface `RequestProcessor` e pelas classes que a implementam, conforme pode ser observado na Figura 3.2. O Código 3.2 apresenta a classe `MethodCrudExecuted`, que é uma das mais importantes do *framework*. Nessa classe são injetadas, como dependência, todas as classes que implementam a interface `RequestProcessor`, linhas 4 e 5, além da implementação de `RouteRequest`, linhas 7 e 8.

Toda vez que um método da classe controladora é executado, o VRaptor dispara um evento que é capturado pelo tipo de anotação `@Observes` do CDI. O método `observerMethods`, da linha 10, teve seu parâmetro `MethodExecuted` anotado com o tipo `@Observes`, assim este método será invocado toda vez que um método do controlador for executado. Na linha 11, o método `isCrudMethod` é invocado para verificar-se método executado na classe controladora possui a anotação `@Crud`, linha 17. Se o método possui a anotação `@Crud`, então o método `processRequest` é invocado, linha 12.

O método `processRequest` percorre a coleção de `RequestProcessor` executando os métodos `shouldProcess`, linha 22. A interface `RequestProcessor` possui dois métodos `process()` e `shoudProcess(RouteRequest route)`. O método `shoudProcess` verifica

se a classe, que o implementa, deve processar a requisição. Para isso, o método analisa o objeto de `RouteRequest` recebido por parâmetro, se a classe for responsável por processar a requisição, ou seja, se o método `shoudProcess` retornar `True`, então seu método `process` é executado, linha 23.

```
1  @RequestScoped
2  public class MethodCrudExecuted {
3
4      @Inject @Any Instance<RequestProcessor>
5      private Instance<RequestProcessor> requestprocessos;
6
7      @Inject
8      private RouteRequest routeRequest;
9
10     public void observerMethods(@Observes MethodExecuted method){
11         if (isCrudMethod(method)) {
12             processRequest();
13         }
14     }
15
16     public boolean isCrudMethod(MethodExecuted method){
17         return method.getControllerMethod().containsAnnotation(Crud.class);
18     }
19
20     public void processRequest(){
21         for (RequestProcessor r : requestprocessos) {
22             if (r.shouldProcess(routeRequest)) {
23                 r.process();
24             }
25         }
26     }
27 }
```

Código 3.2: Implementação da Classe `MethodCrudExecuted`

Atualmente, no *framework ObCrud*, as classes `DeleteObject`, `UpdateObject`, `SaveObject`, `ShowAddForm`, `ShowListForm`, `ShowEditForm` e `ShowViewForm` implementam a interface `RequestProcessor`. O desenvolvedor pode criar suas próprias classes que implementam essa interface, possibilitando que outras requisições sejam avaliadas e novas funções sejam adicionadas ao *framework*. O Código 3.3 apresenta a implementação simplificada da classe `SaveObject`.

```
1  @RequestScoped
2  public class SaveObject implements RequestProcessor {
3
4      @Inject private Instatiator instatiator;
5      @Inject private Repository repositorio;
6      @Inject private ProviderUpload providerUpload;
7
8      @Override
9      public boolean shouldProcess(RouteRequest route) {
10         return route.getHttpMethod() == HttpMethod.POST;
11     }
12
13     @Override
14     public void process() {
15         Object obj = instatiator.getInstiateOrRedirectOnError("add");
16         providerUpload.processUpload(obj);
17         repositorio.create(obj);
18     }
19 }
```

Código 3.3: Implementação simplificada da classe `SaveObject`

Na classe `SaveObject` são injetadas as dependências de `Instatiator`, `Repository` e `ProviderUpload`, linhas de 4 a 6. O método `shouldProcess`, linha 9, retorna verdadeiro se a rota foi acessada com o método `POST` do protocolo HTTP, neste caso a classe `MethodCrudExecuted` irá executar o método `process`, definido na linha 14. O método `process` invoca o método `getInstiateOrRedirectOnError`, linha 15, que tenta criar uma instância com os dados enviados na requisição, se não for possível ele efetua o redirecionamento para o formulário de inserção que irá exibir os erros de validação. Se for possível criar o objeto, o método `processUpload`, linha 16, é invocado. Este método irá realizar, se houver, o *upload* de arquivos do objeto instanciado (obj). A linha 17 invoca o método `create` do repositório para persistir o objeto instanciado.

3.1.3 Repositório (repository)

O subsistema de repositório é composto pela interface `Repository` e por sua implementação `RepositoryJPA`. As classes que implementam essa interface lidam com a tecnologia envolvida na persistência de dados, fornecendo aos subsistemas que utilizam seus serviços uma forma simples de persistir e recuperar objetos de uma fonte de dados.

As classes `DeleteObject`, `UpdateObject` e `SaveObject` respectivamente exclui, atualiza e salva um objeto no banco de dados. Para realizar essas ações, elas utilizam os servi-

ços do subsistema de repositório. O Repositório é utilizado também pelo subsistema de formulários (`form`), para exibir os objetos nas UIs.

3.1.4 Validador de objetos (validator)

Os serviços providos pelo subsistema de validação são expostos pela interface `ErrorValidator`, este subsistema é responsável por recuperar as mensagens de erro, que foram geradas durante a criação dos objetos. A criação dos objetos é realizada pelo subsistema `instatiator`, que é utilizado pelas classes `DeleteObject`, `UpdateObject` e `SaveObject`, do subsistema de processamento de requisição. A validação dos objetos é realizada pela *Bean Validation API* (BERNARD; PETERSON, 2009).

3.1.5 Interceptadores (interceptor)

O subsistema de interceptação (`interceptor`) utiliza os conceitos do paradigma de Programação Orientada a Aspectos para vincular os códigos individuais da aplicação ao código genérico gerado pelo *framework*.

O subsistema `interceptor` é formado pela interface `Intercept`, que possui as assinaturas dos métodos `accept` e `exec`, e pelos tipos de anotação: `@AfterDeleteObCrud`; `@AfterInsertObCrud`; `@AfterUpdateObCrud`; `@BeforeDeleteObCrud`; `@BeforeInsertObCrud`; e `@BeforeUpdateObCrud`, conforme apresentado na Figura 3.2. Estes tipos de anotação são utilizados para anotar métodos na classe *controller*. Quando utilizados, eles informam ao *framework* o *pointcut* e o *advice*, ou seja, qual ação CRUD do *framework* deve ser interrompida (*pointcut*) e qual método da classe *controller* (*advice*) deve ser executado.

No contexto do *Model-Driven Development* (MDD), as UIs CRUD com suas funcionalidades correspondem ao código genérico gerado pelo *ObCrud*, enquanto que os *advices* correspondem aos códigos individuais da aplicação. Nenhum dos *scaffolding* dinâmicos, estudados neste trabalho, possuem este tipo de recurso.

O método `accept`, das implementações de `Intercept`, é responsável por verificar se a classe *controller* possui um *pointcut* e se ela é a responsável por interceptar a execução de uma ação CRUD. Caso seja, o método `exec` obtém e executa o *advice*, método da classe *controller* que está anotado.

A implementação dos interceptadores é realizada utilizando a especificação JSR 299 - *Contexts and Dependency Injection* (CDI) (JSR-299 Expert Group, 2009) e o paradigma de metaprogramação. Esta funcionalidade possibilita a execução de precondições e pós-condições, que são conceitos presentes no modelo de tarefas.

3.1.6 Uploads

O subsistema `upload` é responsável por realizar o *upload* dos arquivos selecionados nas UIs CRUD. Seus serviços são expostos pelas interfaces `KeepFiles` e `ProviderUpload`, conforme pode ser visto na Figura 3.2. A interface `KeepFiles` define os métodos necessários para guardar o arquivo enviado. O armazenamento pode ser uma cópia do arquivo para um diretório do servidor web, ou a atribuição direta dos `bytes` do arquivo no atributo do objeto. A interface `ProviderUpload` possui os métodos destinados a processar o *upload* do objeto enviado pela UI. Para preencher o atributo correto com o nome ou com os bytes do arquivo enviado, a implementação de `ProviderUpload` utiliza as informações de `MetaModel`.

3.1.7 Metamodelo (meta)

O subsistema `meta` é responsável por extrair e armazenar todas as informações dos modelos utilizados para construir as UIs. As principais interfaces deste subsistema são `MetaModel` e `MetaField`, conforme mostrado na Figura 3.2. A interface `MetaModel` representa uma classe de domínio, enquanto que o `MetaField` representa todos os atributos da classe, incluindo suas associações e metadados. O Código 3.4 e o Código 3.5 apresentam as classes responsáveis por criar e armazenar os `MetaModel`.

```
1  public class FactoryMetaModel {
2    @Inject
3    private ClassForCrud clazzCrud;
4    @Inject
5    private RepositoryMetaModel
         repository;
6
7    @Produces
8    public MetaModel createMetaModel(){
9      return repository.get(clazzCrud.
           getClassForCrud());
10   }
11 }
```

Código 3.4: Classe `FactoryMetaModel`

```
1  @ApplicationScoped
2    public class RepositoryMetaModel{
3    private final Map<Class<?>,
         MetaModel> models;
4    public RepositoryMetaModel() {
5      models = new HashMap<>();
6    }
7    public MetaModel get(Class<?> key) {
8      MetaModel m = models.get(key);
9      if (m == null){
10       m = put(key);
11     }
12     return m;
13   }
14   private MetaModel put(Class<?> k) {
15     MetaModel m = new BasicMetaModel(k
           , this);
16     models.put(k, m);
17     return m;
18   }  //outros metodos
19 }
```

Código 3.5: Classe `RepositoryMetaModels`

Um `MetaModel` é um objeto complexo, logo não possui um construtor padrão e necessita de um método anotado com @Produces para criá-lo. Este objeto é criado pelo método `createMetaModel` da classe `FactoryMetaModel`. A classe `FactoryMetaModel` recebe como dependência as instâncias de `ClassForCrud` e de `RepositoryMetaModel`. A classe `ClassForCrud` é responsável por obter o valor do elemento `forClass` do tipo de anotação `@Crud`, ou seja, é responsável por retornar a classe de modelo de domínio que será utilizada para gerar a UI CRUD. A `RepositoryMetaModel` é uma estrutura de dados utilizada para armazenar as instâncias de `MetaModel`. O método `createMetaModel`, linha 9 do Código 3.4, simplesmente invoca o método `get` da classe `RepositoryMetaModel`, que retorna o `MetaModel` da classe de domínio.

O método `get`, linha 7 da classe `RepositoryMetaModel`, obtém o `MetaModel` associado a classe de domínio, se o `MetaModel` ainda não foi criado, ou seja, se ele não está armazenado no repositório, então a condição da linha 9 é verdadeira e o método `put`, da linha 14, é invocado para criar e adicionar o `MetaModel` no respositório. O `RepositoryMetaModel` possui escopo de aplicação `@ApplicationScope`, atuando de forma semelhante ao padrão *singleton*, assim existe apenas um objeto de `RepositoryMetaModel` para toda a aplicação. Isso é importante, pois a criação de um `MetaModel` e dos seus `MetaFields` são realizados uma única vez, evitando desperdício de processamento e, por conseguinte, melhorando o desempenho da aplicação. O Código 3.6 apresenta um trecho da classe que implementa a interface `MetaModel`.

```
1  public class BasicMetaModel implements MetaModel {
2    private final Class<?> clazz;
3    private final Map<String, MetaField> fields = new HashMap<>();
4    private final RepositoryMetaModels repository;
5
6    public BasicMetaModel(Class<?> clazz, RepositoryMetaModels repository) {
7      this.clazz = clazz;
8      this.repository = repository;
9      getAllFields().stream().forEach((f) -> {
10     if (!(Modifier.isStatic(f.getModifiers()) || Modifier.isTransient(f.
           getModifiers()))){
11       MetaField metaField = new BasicMetaField(this, f);
12       fields.put(f.getName(), metaField);
13     }
14    });
15   }
16   private Set<Field> getAllFields() {
17     return ReflectionUtils.getAllFields(clazz, (Predicate<? super Field>[]) null);
18   } //outros métodos
19 }
```

Código 3.6: Classe `BasicMetaModel`: A implementação de `MetaModel`

A classe `BasicMetaModel` recebe pelo construtor, linha 6, a classe de domínio e a referência de `RepositoryMetaModels`. Na linha 9, o construtor invoca o método `getAllFields` que retorna todos os atributos da classe de domínio. Para cada um dos atributos retornados é verificado se eles não possuem os modificadores `static` ou `transient`, linha 10. Atributos estáticos (`static`) são atributos da classe, compartilhados por todos os objetos e atributos transiente (`transient`) são temporários, nos dois casos os atributos com esses modificadores não são persistidos na base de dados, desta forma, também não fazem parte do metamodelo.

Na linha 11, as instâncias de `MetaField` são criadas, invocando o construtor da classe `BasicMetaField`, que recebe por parâmetro as referências de `MetaModel` (`this`) e de `Field` (`f`). Na linha 12, o `metaField` é armazenado no mapa `fields` para uso posterior. A interface `MetaModel` possui, ainda, os métodos para recuperar: o `MetaField` identificador da classe `@Id`, o nome da classe e o nome do objeto, os atributos que são do tipo associação e os atributos do tipo *upload*, que são comumente utilizados na construção de formulários *Web*, na geração de consultas e no processamento de *uploads*. O Código 3.7 apresenta a implementação simplificada de `MetaField`.

```
1  public class BasicMetaField implements MetaField {
2    private final MetaModel model;
3    private final Field field;
4    private final Html.Input input;
5
6    public BasicMetaField(MetaModel model, Field field) {
7      this.model = model;
8      this.field = field;
9      input = (field.isAnnotationPresent(Html.Input.class)) ? field.getAnnotation(
           Html.Input.class) : null;
10   }
11   @Override
12   public boolean isId() {
13     return field.isAnnotationPresent(Id.class);
14   }
15   @Override
16   public boolean isRequired() {
17     return field.isAnnotationPresent(NotNull.class);
18   }
19   @Override
20   public boolean isFileUpload() {
21     return (input != null && input.inputType() == InputType.FILE);
22   }
23     //outros métodos
24 }
```

Código 3.7: Classe `BasicMetaField` simplificada

A classe `BasicMetaField` implementa a interface `MetaField` que herda as assinaturas dos métodos da interface `Attribute`. A interface `Attribute` possui os métodos para extrair o nome do atributo e seu tipo, e verificar se o atributo é transiente, enumerador, associação ou coleção. No caso de coleção, ela também possui o método para recuperar o tipo genérico do atributo. A interface `MetaField` possui os métodos para a extração de metadados, do modelo de apresentação, que define como os controles (*widgets*) serão exibidos na UIs. O método construtor da classe `BasicMetaField`, linha 6, recebe como parâmetro, a instância de `MetaModel` e a instância de `Field`, para a qual será construída o `MetaField`. A linha 9, verifica se o atributo (`field`) possui a anotação `@Html.Input`, se ele possuir, a anotação é obtida e armazenada na variável `input`.

O método `isId`, linha 12, retorna verdadeiro se o atributo possui a anotação `@Id`, o método `isRequired` retorna verdadeiro se o atributo possui a anotaçao `@NotNull` e o `isFileUpload` retorna verdadeiro se o atributo foi anotado com o tipo `@Html.input` e possui o elemento `inputType` com valor `InputType.File`, ou seja, o atributo é do tipo `upload`.

Outros exemplos de métodos da interface `MetaField` são: `getDisplayName` utilizado para obter o rótulo (`label`) do *widget*; `getMin` e `getMax` utilizados para obter o tamanho mínimo e máximo do campo; `isInput` para verificar se o atributo vai ser mostrado como *widget* do tipo *input* e `getInputType` para obter qual tipo *widget* será utilizado para representar o atributo na UI. Todas essas informações são obtidas utilizando o paradigma de metaprogramação, que realiza a análise do modelo de domínio e das suas marcas. As marcas são metadados ou anotações, que corresponde ao modelo de apresentação utilizado no *framework*.

3.1.7.1 Metadados (annotations)

O pacote `annotations` define os tipos de anotação utilizados para anexar metadados às classes de domínio. Este pacote define 4 (quatro) tipos de anotação: `@HTML`; `@Input`; `@Select` e `@TextArea`, conforme pode ser visto na Figura 3.2. Os 3 (três) últimos tipos de anotação foram declarados dentro de `@HTML` e, portanto, só podem ser acessados por meio dele.

O tipo de anotação `@Html` é utilizado para definir como os atributos da classe de modelo serão apresentados nas UIs. Do ponto de vista do MDA, essas anotações correspondem as marcas, utilizadas para agregar informações aos modelos, possibilitando que eles possam ser automatizados. Do ponto de vista do MBUID, estas anotações juntamente com as configurações do formulário (`FormConfiguration`) correspondem ao modelo de apresentação, que especifica como a interface gráfica deve ser apresentada ao usuário.

O tipo de anotação `@Html` possui elementos para descrever as características comuns entre

os diversos *widgets*. Os elementos disponíveis no tipo de anotação `@Html` são:

- *editable*: Elemento utilizado para definir se o *widget* permite edição ou é de apenas leitura, seu valor padrão é verdadeiro, ou seja, possibilita edição.
- *enable*: Elemento utilizado para definir se o *widget* está habilitado, seu valor padrão é verdadeiro.
- *autoFocus*: Elemento utilizado para definir se o *widget* irá receber o foco (cursor) quando a UI for visualizada, seu valor padrão é falso.
- *placeHolder*: Elemento utilizado para definir um texto explicativo que será apresentado dentro do *widget*. Ao iniciar a digitação de dados, o texto explicativo é apagado, este elemento não possui valor padrão.
- *displayName*: Elemento utilizado para definir um texto que será utilizado como rótulo do *widget*, seu valor padrão é o nome do atributo.

Além dos elementos listados, o tipo de anotação `@Html` possui os tipos de anotação internos: `@Input`, `@TextArea` e `@Select`. Estes tipos de anotação são utilizados para definir que tipo de *widget*, respectivamente, será utilizado para representar o atributo, da classe, no formulário. Quando um atributo é anotado com o tipo `@Html.Input`, o elemento `inputType` pode ser configurado com uma constante do enumerador `InputType`, possibilitando que o *widget* seja apresentado como um campo de formulário do HTML5. As seguintes constantes estão disponíveis no enumerador `InputType`: `COLOR`; `DATE`; `DATETIME`; `EMAIL`; `FILE`; `HIDDEN`; `MONTH`; `NUMBER`; `PASSOWORD`; `RANGE`; `SEARCH`; `TEL`; `TEXT`; `TIME`; `URL` e `WEEK`. Além do elemento `inputType` o tipo de anotação `@Input`, possui os elementos `pattern` que possibilita validar os dados informados no *widget* e o elemento `mask` que possibilita definir uma máscara de entrada.

Além dos tipos de anotação do próprio *framework*, o *ObCrud* também utiliza os tipos de anotação de outras APIs, como por exemplo, os tipos de anotação `@Id`, `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany` e `@Temporal` da *Java Persistence API* (DEMICHIEL, 2009) e os tipos `@NotNull`, `@Size`, `@Min` e `@Max` da *Bean Validation API* (BERNARD; PETERSON, 2009), utilizados respectivamente para identificar qual é o atributo Id da classe, qual o tipo de associação entre as classes, qual o tipo de tempo (Data ou Hora) de um atributo, se o atributo é obrigatório, e qual é o seu tamanho ou valor mínimo e máximo.

3.1.8 Template Engine (engine)

O subsistema `engine` disponibiliza os serviços para configuração e uso de *templates engine*. Este subsistema fornece serviços por meio das interfaces: `TemplateEngine`;

`EngineConfiguration`, e `Template`, conforme pode ser visto na Figura 3.2.

A interface `EngineConfiguration` possibilita definir e recuperar informações relativas a *engine* utilizada para processar os arquivos de *template*. A interface `TemplateEngine` disponibiliza os métodos para utilizar um arquivo de *template*, com as configurações definidas na `EngineConfiguration`, por fim, a interface `Template` possibilita vincular valores para os parâmetros do arquivo de *template* e obter o resultado do seu processamento.

Este subsistema basicamente é utilizado pelo subsistema de formulários (`form`), apresentado na próxima seção.

3.1.9 Formulários (form)

O subsistema `form` é responsável por construir as UIs e os componentes que fazem parte dela, os *widgets*. Esse subsistema é formado pelas interfaces `Context`, `FormConfiguration`, `Task`, `FormHtml` e pelo pacote `widgets`, conforme pode ser visto na Figura 3.2.

A implementação de `Context` utiliza as informações de `MetaModel`, de `Repository`, de `FormConfiguration` e de `ErrorValidator` para criar os *widgets* e as tarefas. O Código 3.8 apresenta a implementação simplificada da interface `Context`.

```
1  @RequestScoped
2  public class BasicContext  implements Context{
3    private Map<String, Widget> widgets = new HashMap<>();
4    @Inject private MetaModel metaModel;
5    @Inject private FormConfiguration config;
6    @Inject private FormTaskCollection tasks;
7    @Inject private ErrorValidator errors;
8    @Inject private Repository repository;
9
10   @PostConstruct
11   public void process() {
12     tasks = new BasicFormTaskCollection(config);
13     this.metaModel.getFields().forEach((k, v) -> {
14       widgets.put(k, new BasicWidget(v, config.getWidgetConfiguration(k),
             repository, getEditObject(), errors));
15     });
16   }
17   //outros métodos
18 }
```

Código 3.8: Classe `BasicContext` simplificada

As linhas de 4 a 8 injetam as dependências de `Context`. O método `process`, linha 11, está anotado com o tipo `@PostConstruct`, da linha 10, para que o CDI o invoque, após o objeto ser construído. Esse método, na linha 12, cria as tarefas que serão adicionadas nas UIs e na linha 14, cria um `widget` para cada `MetaField` do `MetaModel`. Para criar os `widget`, o construtor de `BasicWidget` é invocado passando o `MetaField v`, as configurações do `widget` (`WidgetConfiguration`) obtidas de `FormConfiguration`, os possíveis erros de validação (`ErrorValidator`), o objeto que por ventura está sendo editado e o repositório. Todos os `widget` são armazenados em um mapa, para posterior uso nas classes que implementam a interface `FormHtml`. O Código 3.9 apresenta, de forma simplificada, a classe `ShowAddForm` que implementa as interfaces `FormHtml` e `RequestProcessor`.

```
1  @RequestScoped
2  public class ShowAddForm implements FormHtml, RequestProcessor{
3    @Inject private TemplateEngine template;
4    @Inject private Context context;
5    @Inject private Result result;
6    @Override
7    public boolean shouldProcess(RouteRequest route) {
8      return route.getHttpMethod() == HttpMethod.GET && route.getRequestedUri().
           startsWith(route.getRoute("add"));
9    }
10   @Override
11   public void process() {
12     result.include("crud", this.render("add"));
13   }
14   @Override
15   public String render(String template) {
16     return template.use(template).with("widgets", context.getWidgetAddForm())
17     .with("buttons", context.getButtonsVisible())
18     .with("path", context.getPathAction(""))
19     .with("title", context.getTitle()).getContent();
20  }
21 }
```

Código 3.9: Classe `ShowAddForm` simplificada

As linhas de 3 a 5 injetam as dependências da classe `ShowAddForm`. A linha 7 implementa o método `shouldProcess` da interface `RequestProcessor`, que verifica se a classe é responsável por processar a requisição do usuário, ela será responsável se a rota for `add` e o protocolo `HTTP` for `GET`. O método `process`, na linha 12, inclui na resposta da requisição uma variável `crud` com valor de retorno do método `render`. O método `render`, linha 15, recebe como parâmetro o nome do arquivo de *template* a ser renderizado "`add`". Este `Template` é utilizado, na linha 16, com um conjunto de dados, na forma de chave-valor. A chave corresponde a um parâmetro, do arquivo de *template*, que será substituído por valor do contexto (`Context`). A chamada do método `getContent`, na linha 19, executa o

`TemplateEngine` *Freemarker*, que processa o *template* com os dados do contexto, gerando o código do formulário de inserção. O Código 3.10 apresenta um trecho do *template* da UI de inserção `add.ftl`.

```
1  <div class="page-header1">
2    <h1>${title}</h1>
3  </div>
4
5  <form method="post" action="${path}" enctype="multipart/form-data">
6   <div class="form_crud">
7    <#list widgets as widget>
8      ${widget.render()}
9    </#list>
10
11   <div class="row">
12   <div class="col-md-12 top-space-10">
13     <#if buttons?seq_contains('SAVE')>
14       <button type='submit' class="btn btn-success">Salvar</button>
15     </#if>
16     <#if buttons?seq_contains('SAVE_BACK')>
17       <button type='submit' class="btn btn-default">Salvar e voltar para a listagem<
           /button>
18     </#if>
19     <#if buttons?seq_contains('CANCEL')>
20       <a href="${path}" class="btn btn-default voltar_list">Cancelar</a>
21     </#if>
22     </div>
23   </div>
24  </div>
25 </form>
```

Código 3.10: *Template* da UI de inserção

O *template* consiste em um modelo de código, neste caso um código HTML. Na linha 2 foi definido o parâmetro `${title}` que será substituído pelo valor de retorno do método `getTitle` do contexto, que por sua vez, verifica se o desenvolvedor atribuiu um título para o formulário, por meio da interface `FormConfiguration`. A linha 5 define o método de envio do formulário `POST` e substitui o parâmetro `${path}` pela URL da rota de inserção. As linhas de 7 a 9 percorrem uma lista de `widget` substuindo o parâmetro `${widget.render()}` pelo retorno do método `render` do `widget`. Os `widget` foram criados no `Context` e por meio do método `getWidgetAddForm`, os `widget` da UI de inserção, foram enviados ao `Template`. O método `render` do `widget`, também utiliza arquivos de *template* e informações do `MetaField` e do `WidgetConfigurantion`, para gerar `widget` apropriado para UI. As linhas de 13 a 21 verificam se o parâmetro `button`, do tipo vetor de `String`, possui os elementos `SAVE`, `SAVE_BACK` e `CANCEL`, para cada valor existente no vetor um tipo de botão é exibido na UI.

Conforme apresentado, na linha 12 do Código 3.9, o código da UI de inserção é incluído na variável `crud` do resultado da requisição. Essa variável pode, então, ser exibida na camada de visão, por meio da *Expression Language*. O Código 3.11 apresenta um exemplo de exibição da UI criada na camada de visão. A linha 7 exibe o valor da variável `menu`, criada pelo subsistema de `menu` e a linha 8 exibe a váriavel `crud`, que possui o código de uma UI e foi gerado pelo subsistema `form`.

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4         <!-- link dos arquivos CSS -->
5         </head>
6         <body>
7           ${menu}
8           ${crud}
9         </body>
10        <!-- JavaScript -->
11 </html>
```

Código 3.11: Apresentando a UI na camada de visão

O *ObCrud* modifica o comportamento do VRaptor, para que o retorno do método, anotado com `@Crud`, apresente o arquivo de visão disponibilizado pelo *framework*, assim a UI gerada vai utilizar as folhas de estilo em cascata - CSS e os arquivos JavaScript definidos pelo *framework*.

Ainda sobre a interface `FormConfiguration`, ela possibilita configurar a aparência e o comportamento das UIs. Ela possui métodos para configurar, entre outras coisas, quais *widgets* estarão nas interfaces de adição, edição e de visualização, quais colunas (atributos) estarão visíveis na UI de listagem e quais ações CRUD serão permitidas em cada UIs. Além disso, ela permite adicionar novas tarefas nas UIs, por meio da implementação da interface `Task`. A `FormConfiguration` possui escopo de requisição `@RequestScope`, dessa forma, uma única instância é criada por requisição, assim a instância de `FormConfiguration` utilizada do `Controller` é a mesma que é utilizada no `Context`.

A implementação de `Task` possui os métodos necessários para adicionar novas funcionalidades as UIs CRUD. Por padrão, as funcionalidades básicas de uma UI CRUD são: Criar (**C**reate); Recuperar (**R**etrive); Atualizar (**U**pdate) e Excluir (**D**elete) objetos do domínio. As tarefas juntamente com os interceptadores definem o modelo de tarefas utilizado no *ObCrud*.

A interface `FormConfiguration` atua também como uma fachada, *facade pattern*, para a configuração dos *Widgets* (`WidgetConfiguration`). O `WidgetConfiguration` pertence ao pacote `widget` apresentado na próxima seção.

3.1.9.1 Widget

O pacote `widget` disponibiliza os serviços para criação de *widgets*. *Widgets* são objetos visuais que possibilitam aos usuários executar ações ou informar valores, exemplos de *widgets* são: campos de texto, caixas de seleção, menus, caixa de alertas, etc. (FERREIRA; RODRIGUES, 2008).

Os *widgets* são criados e armazenados na implementação de `Context`. A implementação de `Context` fornece suas referências de `MetaModel`, de `Repository`, de `ErrorValidator` e de `WidgetConfiguration` para que a implementação de `Widget` seja construída, com essas informações a `Widget` define que tipo de objeto visual deve ser gerado e quais serão suas características. Ao final, os serviços do subsistema `engine` são requisitados para gerar o código do *widget*.

A interface `WidgetConfiguration` possibilita realizar as mesmas configurações que o tipo de anotação `@Html`, a diferença reside no fato de que as configurações realizadas com `@Html` são aplicadas a todas as UIs geradas, enquanto que as configurações realizadas com o `WidgetConfiguration` são específicas para cada UI. Isso possibilita que diferentes UIs sejam geradas para o mesmo modelo de domínio, atendendo a diferentes contextos de uso.

As implementações de `FormConfiguration` e `WidgetConfiguration`, fazem parte do modelo de apresentação do *ObCrud* e possibilitam maior flexibilidade na geração de UIs.

3.2 Processo de desenvolvimento

Um processo de desenvolvimento de *software* corresponde a "um conjunto de atividades, métodos, práticas e transformações que as pessoas empregam para desenvolver e manter *software* e produtos associados (por exemplo, planos de projeto, documentos de projeto, projeto de *software*, código, casos de testes e manual do usuário" (BRASIL, 1999). Esta seção apresenta o processo de desenvolvimento de UIs CRUD utilizando o *framework ObCrud*. Para isso, são descritas as atividades, práticas e transformações relacionadas, exclusivamente, a fase de implementação.

O *ObCrud* não impõe restrições quanto ao processo de desenvolvimento de *software* adotado pela equipe de desenvolvimento, no entanto, por se tratar de uma ferramenta de desenvolvimento dirigida por modelo (MDD), algumas atividades e modelos devem ser realizados para gerar as UIS. A Figura 3.3 apresenta um esquema do processo de desenvolvimento de UIs CRUD adotado pelo *framework*.

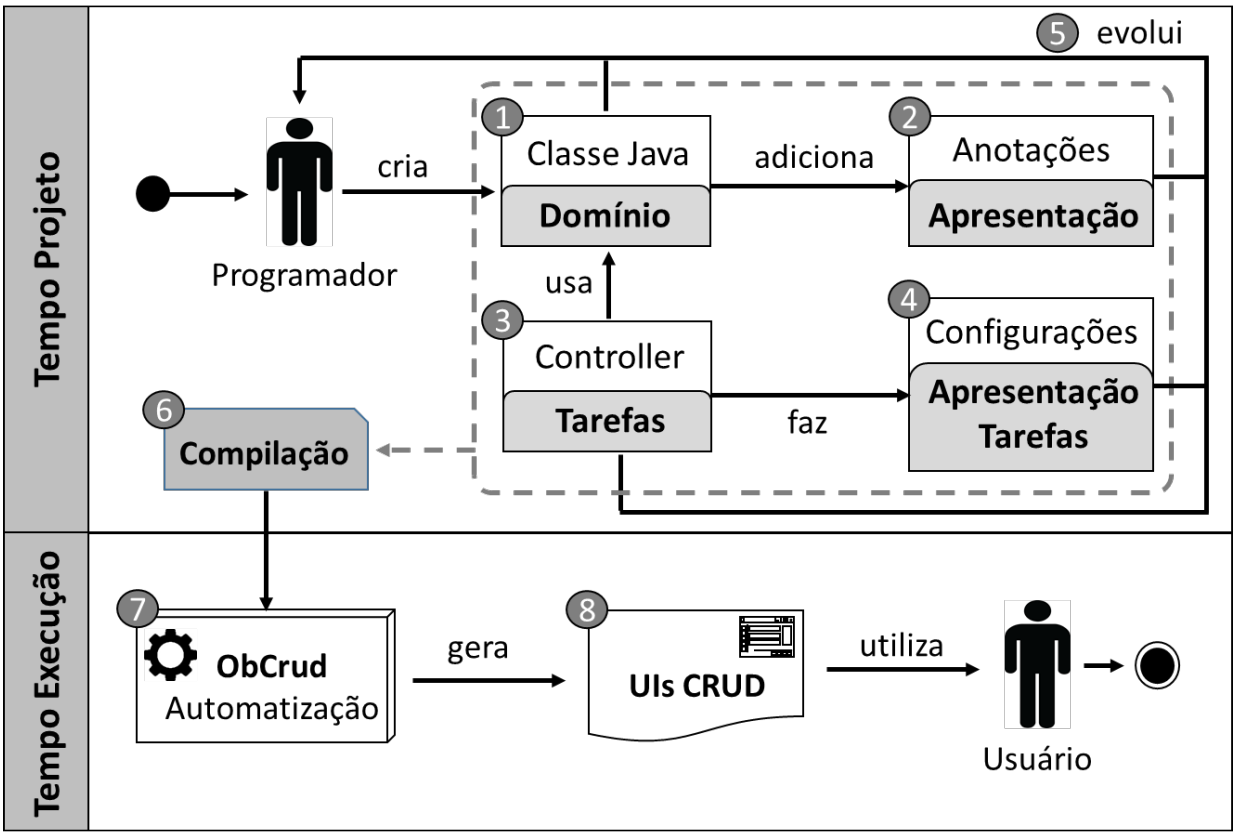

Figura 3.3: Processo de desenvolvimento do *ObCrud*

O processo de desenvolvimento das UIs CRUD é realizado em dois momentos: em tempo de projeto e em tempo de execução. Em tempo de projeto, o programador deve (1) criar as classes Java do modelo de domínio, a essas classes (2) são adicionadas anotações, que correspondem ao modelo de apresentação, essas anotações podem ser próprias do *ObCrud* ou de outras especificações como a *JPA* (DEMICHIEL, 2009) e a *Bean Validation* (BERNARD; PETERSON, 2009).

Uma vez criada as classes de modelo e adicionadas as anotações, o próximo passo (3) é criar a classe controladora. Esta classe deve possuir um método público, anotado com o tipo de anotação `@Crud`. No elemento `forClass`, do tipo de anotação `@Crud`, deve ser informado a classe de domínio, para qual se deseja criar a UI CRUD.

O uso de (4) configurações são opcionais. Elas possibilitam adicionar e modificar comportamento nas UIs, bem como permite modificar sua aparência. Por meio das configurações, é possível adicionar novas tarefas, habilitar ou desabilitar as ações CRUD, adicionar ou remover *widgets* e personalizar a aparência e comportamento das UIs. Além disso, nos *controllers*, é possível adicionar anotações de interceptação, possibilitando que funções do desenvolvedor sejam executadas antes ou após uma ação CRUD. As configurações e os interceptadores correspondem aos modelos de apresentação e de tarefa.

Todos esses modelos podem evoluir durante o desenvolvimento do sistema, sendo que as UIs acompanham essa evolução dinamicamente. Uma vez construído os modelos, (6) eles devem ser compilados e implantados em um servidor de aplicação, que tenha suporte ao CDI. Em tempo de execução, o *framework ObCrud* entra em ação. Neste momento, ele (7) inspeciona todos os modelos criados e extrai as informações necessárias para gerar as UIs CRUD, além de criar as rotas de acesso para as funcionalidades CRUD e analisar as requisições do usuário, definindo qual ação ou (7) UI deve ser apresentada.

Por padrão, o *ObCrud* tenta poupar o trabalho do desenvolvedor, utilizando as informações disponíveis no modelo de domínio para gerar as UIs. Dessa forma, usando apenas a anotação `@Crud`, é possível gerar uma UI CRUD básica, igual as geradas pelos *scaffoldings*. No entanto, o uso de configurações e de outras anotações possibilitam maior flexibilidade ao desenvolvedor, que pode gerar UIs mais complexas, capaz de atender as diferentes necessidades do projeto.

As próximas seções apresentam alguns casos de uso, demonstrando as possibilidades de geração de UIs CRUD, utilizando o *framework ObCrud*. Durante as demonstrações, os conceitos apresentados no trabalho são relacionados com a arquitetura do *framework*.

3.2.1 Geração de UIs básicas

A configuração mínima para a geração de uma UI CRUD consiste na criação de uma classe de domínio e de uma classe controladora. A classe de domínio deve possuir atributos e deve estar anotada com os tipos de anotação `@Entity`, `@Id`, `@GeneratedValue`, esses tipos pertencem a *Java Persistence API* (DEMICHIEL, 2009) e são utilizados, respectivamente, para mapear uma classe para uma tabela do banco de dados, mapear um atributo como chave primária de uma tabela e configurar o atributo para gerar valores automaticamente (*auto increment*). O Código 3.12 apresenta um exemplo de classe de domínio com as anotações da JPA e o Código 3.13 apresenta um modelo de classe controladora com a anotação do *ObCrud*.

```
1  @Entity
2  public class Medico {
3
4      @Id
5      @GeneratedValue
6      private Integer idMedico;
7      private String nome;
8      private String crm;
9      //getter e setter
10 }
```

Código 3.12: Classe de domínio `Medico`

```
1  @Controller
2  public class MedicoController {
3
4      @Crud(forClass = Medico.class)
5      public void index(){
6
7      }
8  }
```

Código 3.13: Classe de controle `Medico`

A classe controladora deve possuir um método público e deve estar anotada com o tipo de anotação `@Crud`, esse tipo de anotação foi definido no *framework ObCrud* e é utilizado para indicar que o método anotado deve gerar as UIs CRUD.

A anotação `@Crud` possui o elemento `forClass`, neste elemento deve ser informada a classe de domínio para qual se deseja criar as UIs CRUD. Ao implantar a aplicação e acessar a rota para a classe controladora, as UIs CRUD são criadas dinamicamente. A Figura 3.4 apresenta as UIs CRUD geradas usando o Código 3.12 e o Código 3.13. Para fins de demonstração, alguns dados foram inseridos previamente.

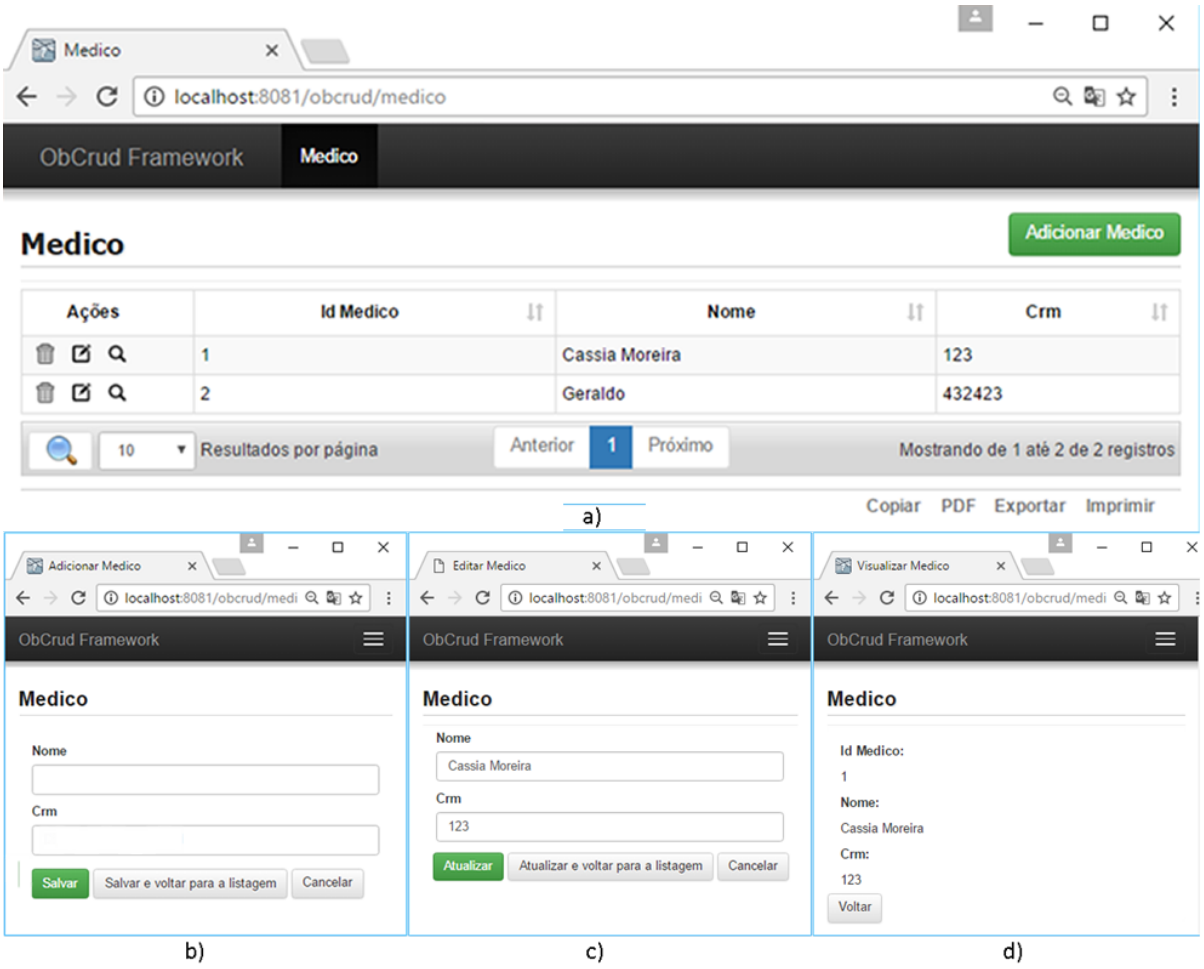

Figura 3.4: UIs CRUD básicas geradas pelo *ObCrud*

A Figura 3.4 (a) apresenta a interface de listagem de objetos, que é a interface principal para cada uma das classes de domínio. Ela possibilita chamar as demais UIs e realizar diversas ações. Na listagem de objetos, a primeira coluna possui 3 (três) ícones que permitem excluir, editar e visualizar um objeto. O botão verde na listagem, "Adicionar Médico", exibe uma UI para adicionar novos médicos, Figura 3.4 (b). O botão editar exibe uma UI para editar médicos, Figura 3.4 (c), e o botão visualizar exibe um UI para visualizar dados de um médico, Figura 3.4 (d).

A UI de listagem, Figura 3.4 (a), possui as ações para consultar médicos, exibir uma determinada quantidade de médicos por página, navegar entre as páginas de resultados (paginação), ordenar os médicos pelos valores das colunas (atributos), além de possibilitar que os objetos sejam exportados para os formatos CSV (*comma separated values*) e PDF, também é possível copiá-los para área de transferência ou imprimi-los. As UIs geradas são responsivas, ou seja, se adaptam a resolução dos dispositivos, no qual estão sendo visualizadas.

3.2.2 Personalização das UIs com anotações do ObCrud

O *ObCrud* dispõe de tipos de anotações, que aplicadas ao modelo de domínio, possibilitam modificar a forma de apresentação dos *widgets* nas UIs. Essas anotações correspondem ao modelo de apresentação do *ObCrud*. O Código 3.14 apresenta a classe de domínio `Noticia.java`, essa classe possui 5 atributos, `idNoticia`, `titulo`, `materia`, `imagem` e `dataPublicação`. O `idNoticia` foi anotado com os tipos de anotação `@Id` e `@GeneratedValue`, dessa forma este atributo é utilizado para identificar uma notícia, como seu valor é gerado automaticamente ele não cria um campo no formulário para que o usuário possa preenchê-lo.

```
1  @Entity
2  public class Noticia {
3
4      @Id
5      @GeneratedValue
6      private int idNoticia;
7
8      @Html(displayName = "Título",
9          placeHolder="Digite o título")
10     private String titulo;
11
12     @Html.TextArea
13     private String materia;
14
15     @Html.Input(inputType = InputType.
           FILE)
16     private String imagem;
17
18     @Temporal(TemporalType.DATE)
19     private Date dataPublicacao;
20 }
```

Código 3.14: Classe de domínio `Noticia`

```
1  @Controller
2  public class NoticiaController {
3
4      @Crud(forClass = Noticia.class)
5      public void index(){
6
7      }
8  }
```

Código 3.15: Classe *Controller* `Noticia`

A Figura 3.5 apresenta a UI, gerada pelos códigos 3.14 e 3.15.

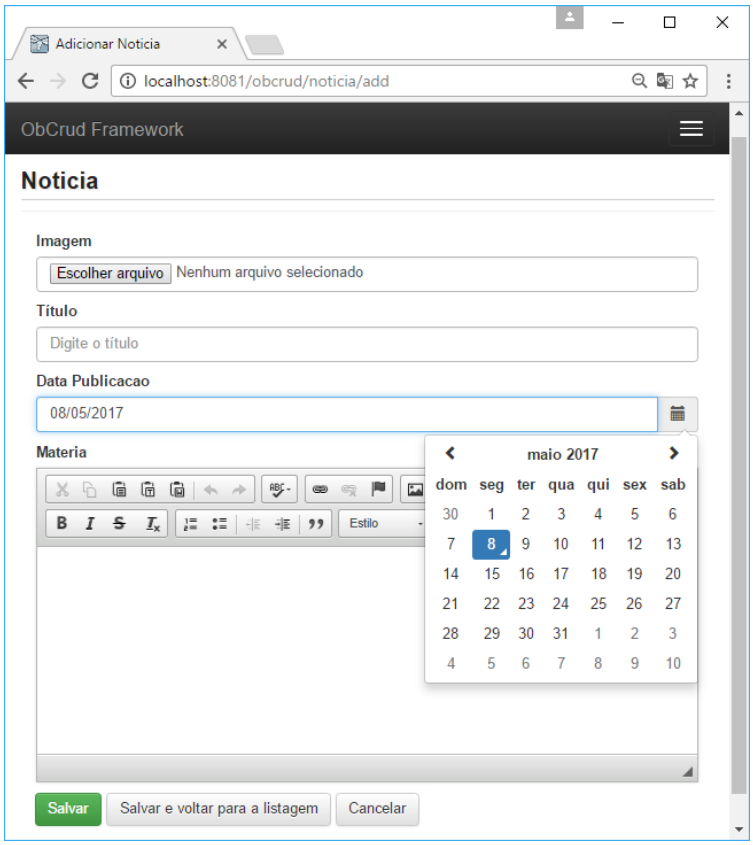

Figura 3.5: UIs CRUD notícia com anotações do modelo de apresentação

Os atributos `titulo`, `materia` e `imagem` são do tipo `String`, mas estão anotados com tipos diferentes de anotações. O atributo `titulo` foi anotado com o tipo `@Html`, que permite a modificação de propriedades compartilhadas por todos *widgets*. Foi atribuído ao elemento `displayName` o valor "Título", corrigindo assim, a acentuação do rótulo do `Widget`. Também foi aplicado, ao *Widget*, um texto de ajuda “digite o título”, com o uso do elemento `placeHolder`.

O atributo `materia` foi anotado com o tipo de anotação `@Html.TextArea`. Quando um atributo é anotado com `@Html.TextArea` seu `Widget` na UI é apresentado com um editor WYSIWYG (*what you see is what you get*), o que permite ao usuário aplicar estilos ao texto. Essa configuração pode ser removida, de forma que apenas um elemento `TextArea` seja exibido, para isso o elemento `editor`, do tipo de anotação `@Html.TextArea`, deve estar configurado com `false`.

O atributo `imagem` foi anotado com o tipo de anotação `@Html.Input`. Por padrão, todo atributo do tipo `String` é um elemento `Input` do tipo `text`. Quando o elemento é anotado com `@Html.Input` é possível alterar o tipo do elemento `Input`, por meio do elemento `inputType`. No exemplo, foi atribuído ao elemento `inputType` o enumerador `InputType.FILE`, desta forma, o campo será apresentado como um campo de `upload`, além disso, a UI ao ser gerada estará apta para realizar *upload* de arquivos. Os arqui-

vos serão armazenados em um diretório do servidor e seu caminho será armazenado no atributo.

O atributo `dataPublicacao` é do tipo `Date` que pode ser utilizado, em Java, para armazenar data e hora. Sempre que este tipo é utilizado, o padrão JPA exige que seja informado a forma de mapeamento para o banco de dados. É possível mapear o atributo como uma data, uma hora ou como uma data e hora. O mapeamento é realizado com o tipo de anotação `@Temporal`. O *framework ObCrud* utiliza essa informação para gerar um *widget* apropriado para atributo.

3.2.3 Validação de dados

O *ObCrud* utiliza a *Bean Validation API* (BERNARD; PETERSON, 2009) para realizar a validação de campos do formulário (*widgets*). O Código 3.16 apresenta a utilização dos tipos de anotação para validação de dados. A Figura 3.6 apresenta a UI gerada, por meio da classe `Medico`, após uma tentativa de cadastro com valores inválidos.

```
1  @Entity
2  public class Medico {
3
4      @Id
5      @GeneratedValue
6      private Integer idMedico;
7
8      @NotNull
9      private String nome;
10
11     @NotNull
12     @Size(min = 3, message="O número
           do CRM deve possuir no mínimo
           3 dígitos")
13     private String crm;
14
15     @ValidCPF
16     private String cpf;
17
18     //getter e setter
19
20 }
```

Código 3.16: Classe para validação de dados

Figura 3.6: Exemplo de UIs com validação de dados

A linha 8 utiliza o tipo de anotação `@NotNull` para informar que o atributo `nome` não pode ser nulo, ou seja, seu preenchimento é obrigatório. O atributo `crm`, linha 13, utiliza os tipos de anotação `@NotNull`, linha 11, e `@Size`, linha 12, desta forma o valor do atributo não pode ser nulo e seu comprimento não pode ser inferior a 3 caracteres, conforme definido no elemento `min`. O elemento `message` do tipo de anotação `@Size` foi configurado para alterar a mensagem exibida no caso de falha da validação. Por fim, o atributo `cpf` utiliza a anotação `@ValidCPF`, apresentada no Código 2.28. As anotações `@NotNull` e `@Size` são tipos incorporados a *Bean Validation API*, enquanto que a anotação `@ValidCPF` é um tipo criado pelo usuário.

O *ObCrud* realiza a validação de dados em dois momentos. No primeiro momento, a validação é realizada no lado cliente (*client-side*), utilizando os novos recursos do HTML 5. No segundo momento, a validação é realizada no lado do servidor (*server-side*), neste momento a *Bean Validation API* é acionada. A Figura 3.6 (a) apresenta a UI para adicionar médico, após a tentativa de salvar o registro sem os campos estarem preenchidos. Neste caso, foi realizada a validação no *client-side*. Na Figura 3.6 (b) o campo `nome` havia sido preenchido, o `cpf` não estava preenchido e o `crm` possuía apenas um caractere. No segundo foi realizada a validação no *server-side*.

O HTML 5 não possibilita realizar todos os tipos de validação, por isso a *Bean Validator API* é essencial. Além disso, o *ObCrud*, foi projetado para implementar serviços web (*RESTful*), neste caso, a validação pode ocorrer somente no *server-side*. A validação no *client-side* busca reduzir, quando possível, a carga de trabalho do servidor e agilizar o preenchimento dos dados, visto que a validação no *client-side* é mais rápida do que no *server-side*.

3.2.4 Geração de UIs com associações entre classes de domínio

Nos exemplos anteriores, as UIs foram geradas a partir de uma única classe do modelo de domínio. No desenvolvimento de sistemas reais é comum a existência de diversas classes inter-associadas. Uma associação de classes, em Java, consiste na definição de um atributo em uma classe, cujo tipo é outra classe. A Figura 3.7 apresenta um diagrama de classes, onde a classe `Cirurgia` está associada as classes `Medico` e `Paciente`.

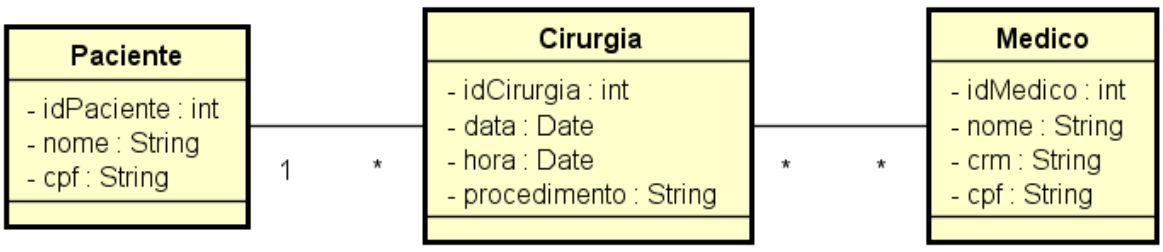

Figura 3.7: Diagrama de classes com associações

A associação apresentada na Figura 3.7 entre a classe `Paciente` e `Cirurgia` é uma associação de multiplicidade um para muitos, assim em código Java, a classe `Paciente` possuirá uma lista de atributos do tipo `Cirurgia`, enquanto que a classe `Cirurgia` possuirá um atributo do tipo `Paciente`. O Código 3.17 apresenta a implementação, em Java, da classe `Paciente` mostrada na Figura 3.7.

```
1  @Entity
2  public class Paciente {
3      @Id
4      @GeneratedValue
5      private int idPaciente;
6      private String nome;
7      private String cpf;
8
9      @OneToMany(mappedBy = "paciente", cascade = CascadeType.ALL)
10     private List<Cirurgia> cirurgias = new ArrayList<>();
11     //getter e setter
12 }
```

Código 3.17: Classe domínio `Paciente` com associações

A classe `Paciente` possui um atributo do tipo lista de `Cirurgia`, criado na linha 10. Este atributo está anotado com o tipo de anotação `@OneToMany`, que indica que um `Paciente` pode possuir várias cirurgias. Nesta anotação, foram informados dois elementos: o `mappedBy` e o `cascade`.

O elemento `mappedBy` indica que a chave-estrangeira, no mapeamento objeto-relacional, será criada na tabela `Cirurgia`. Ele também indica que a associação é bidirecional, caso não fosse utilizado este elemento, ao invés de uma associação bidirecional, seriam criadas duas associações unidirecionais.

O elemento `cascade` indica como as operações devem ser propagadas para os demais objetos associados. Neste caso, foi informado o valor `CascadeType.ALL`, assim qualquer modificação no objeto `Paciente` reflete na sua lista de `Cirurgia`, por exemplo, se o objeto `Paciente` for excluído, então todas as suas cirurgias serão excluídas, como previsto para uma associação de composição.

A associação entre `Medico` e `Cirurgia` possui a multiplicidade de muitos para muitos, assim a classe `Medico` tem uma lista de atributos do tipo `Cirurgia` e uma `Cirurgia` tem uma lista de atributos do tipo `Medico`. O Código 3.18 apresenta o código resultante da implementação da classe `Medico` apresentada na Figura 3.7.

A classe `Medico` possui um atributo do tipo lista de `Cirurgia`, anotado com o tipo `@ManyToMany`, que indica que um `Medico` pode realizar várias cirurgias e que uma

`Cirurgia` pode ser realizada por vários médicos. O elemento `mappedBy` indica que se trata de uma associação bidirecional e informa o local da criação da chave estrangeira. Neste tipo de anotação, não foi utilizado o elemento `cascade`, assim ao tentar excluir um `Medico`, que possua cirurgias, um erro será apresentado, solicitando que as associações sejam removidas, antes de realizar a exclusão do médico.

```
1  @Entity
2  public class Medico {
3      @Id
4      @GeneratedValue
5      private Integer idMedico;
6      private String nome;
7      private String crm;
8      private String cpf;
9      @ManyToMany(mappedBy = "medicos")
10     private List<Cirurgia> cirurgias = new ArrayList<>();
11     //getter e setter
12 }
```

Código 3.18: Classe Domínio `Medico` com associações

O Código 3.19 apresenta a implementação da classe `Cirurgia` apresentada na Figura 3.7.

```
1  @Entity
2  public class Cirurgia {
3      @Id
4      @GeneratedValue
5      private Integer idCirurgia;
6      @Temporal(TemporalType.DATE)
7      private Date data;
8      @Temporal(TemporalType.TIME)
9      private Date hora;
10     @ManyToOne
11     private Paciente paciente;
12     @ManyToMany
13     private List<Medico> medicos = new ArrayList<Medico>();
14     private String procedimento;
15     //getter e setter
16 }
```

Código 3.19: Classe domínio `Cirurgia` com associações

A classe possui um atributo do tipo `Paciente`, linha 11, anotado com o tipo de anotação `@ManyToOne`, que indica que muitas cirurgias podem ser realizadas em um `Paciente`, ela também possui um atributo do tipo lista de `Medico`, linha 13, anotado com tipo de

anotação `@ManyToMany`, que indica que muitas cirurgias podem ser realizadas por muitos médicos.

As anotações `@Entity`, `@Id`, `@GeneratedValue`, `@Temporal`, `@OneToOne`, `@OneToMany`, `@ManyToOne` e `@ManyToMany` pertencem a especificação *Java Persistence API* (DEMICHIEL, 2009), logo já são empregadas no desenvolvimento de sistemas orientados a objeto, que utilizam banco de dados relacionais. O `ObCrud` aproveita essas anotações, que já seriam utilizadas, para obter informações que possibilitam criar as UI CRUD com suporte a associações.

Para os atributos que correspondem a associações são criados *widgets* do tipo `select`. Quando o atributo é simples, é criado um `select` para selecionar uma única opção, quando o atributo é uma lista, é criado um `select` para selecionar multiplas (`multipe`) opções. Nos *widgets* `select` são mostrados os dados referentes ao tipo do atributo, armazenado no banco de dados, ou caso o tipo seja um enumerador, são apresentados seus valores.

O *ObCrud* suporta todos os tipos de associação e multiplicidade, incluindo a generalização. A Figura 3.8 e a Figura 3.9 apresentam a UI da classe `Cirurgia` gerada a partir do Código 3.19. A Figura 3.8 (a) apresenta a renderização do *widget* data e a Figura 3.8 (b) a renderização do *widget* hora.

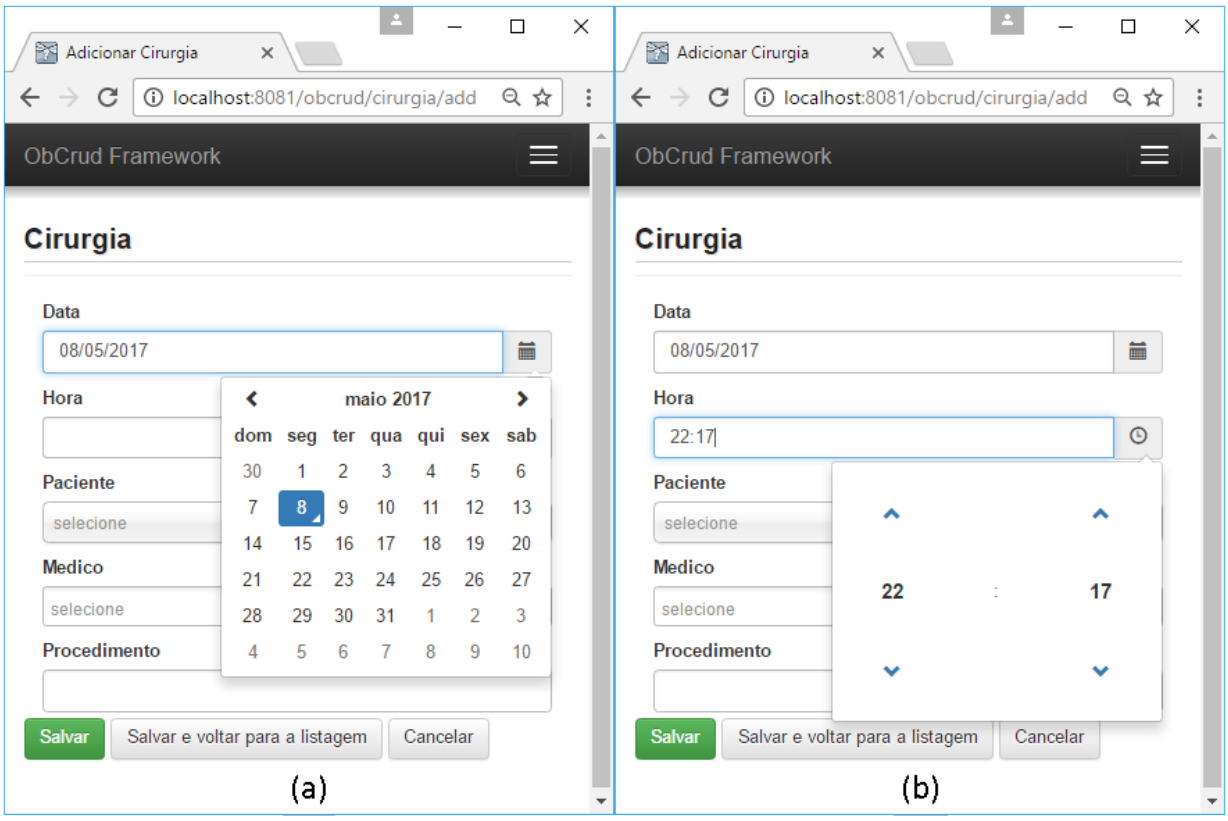

Figura 3.8: UIs CRUD cirurgia: Renderização dos *widgets* data e hora

A Figura 3.9 (a) apresenta o *widget* para a associação do tipo muitos para um (`@ManyToOne`) e a Figura 3.9 (b) apresenta o *widget* para a associações do muitos para muitos (`@ManyToMany`), presentes na classe `Cirurgia`, Código 3.19.

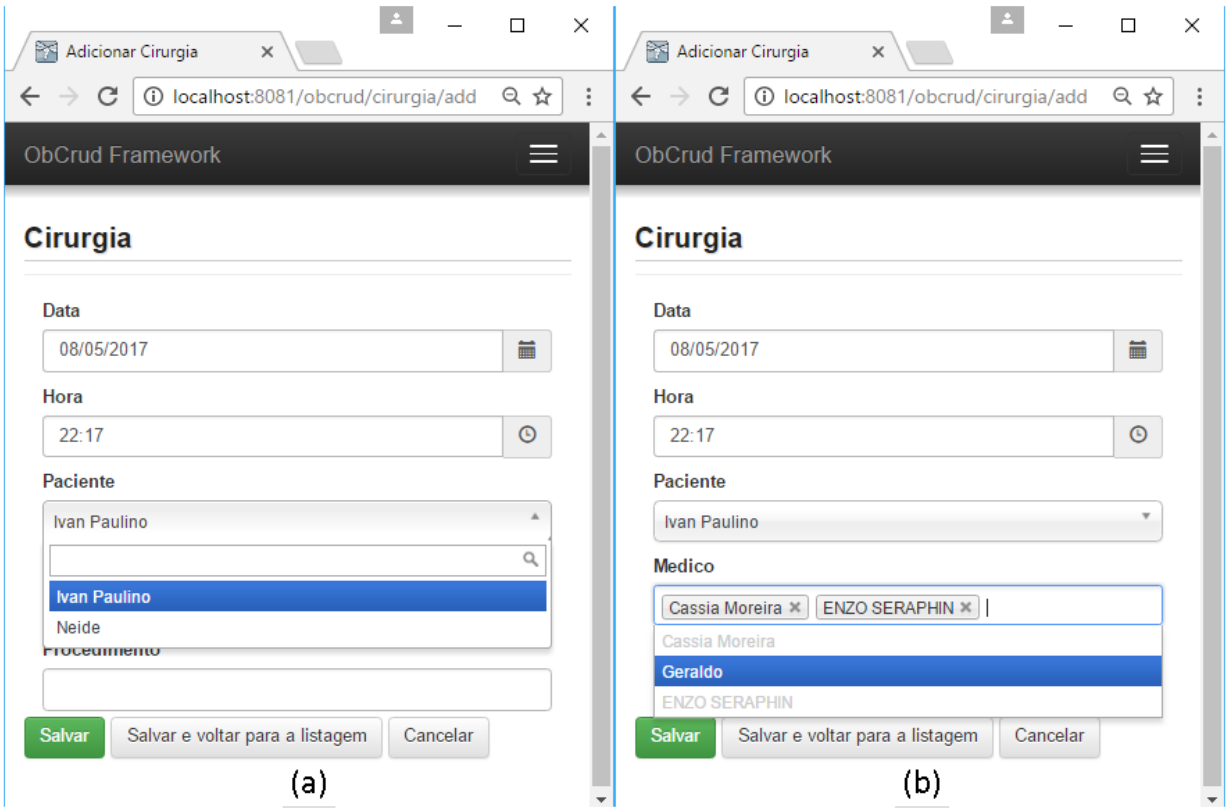

Figura 3.9: UIs CRUD cirurgia: Renderização dos *widgets* de associação

A Figura 3.10 apresenta a UI da listagem de cirurgias. Como o atributo `medicos` é uma lista, seus valores, no formulário de listagem, são separados por vírgula.

Figura 3.10: UIs CRUD de listagem de associação

3.2.5 Geração de UIs baseadas em contexto de uso

Considera-se contexto, nesta seção, as informações sobre os usuários que estão utilizando o sistema e quais dados e ações eles podem visualizar e executar nas UIs. O *framework ObCrud* não realiza o gerenciamento das informações dos usuários, como a autenticação e autorização, mas possibilita que diferentes UIs sejam geradas para diferentes grupos de usuários. Dessa forma, o *framework* utiliza as informações de contexto de uso para produzir interfaces que atendam as especificidades de cada usuário.

No *ObCrud*, a geração de UIs baseadas em contextos de uso, corresponde a configuração da aparência e de funcionalidades nas UIs. Essas configurações são realizadas nas classes de controle, por meio da implementação de `FormConfiguration`.

A interface `FormConfiguration` permite definir quais *widgets* estarão disponíveis para visualização, alteração ou inserção nas UIs. Além disso, ela permite: definir quais funcionalidades CRUD poderão ser executadas; adicionar precondições e pós-condições em cada uma das ações CRUD e incluir novas ações (tarefas) que os usuários poderão executar nas UIs.

O Código 3.20 apresenta o uso da dependência `FormConfigurantion` na classe `MedicoController`. Neste exemplo, foi modificado para todos os usuários, o título do formulário e os rótulos dos *widgets* `idMedico` e `crm`. Para simular o uso de usuários diferentes, na linha 15, foi um incluída uma instrução condicional. A condição verifica se um usuário `Enfermeiro` está utilizando a UI. Neste caso, o método `withColumns` recebe por parâmetro as colunas que devem ser apresentadas na UI de listagem, o método `withEditFields` recebe por parâmetro o nome dos atributos (*widgets*) que devem estar visíveis na UI de edição, os métodos `setVisibleAddButton` e `setVisibleDeleteButton` recebem por parâmetro o valor `false` (falso), indicando que a funcionalidade de inserção e remoção da UI estão inabilitadas para este usuário. Supondo que o usuário não fosse um `Enfermeiro`, então essas configurações não seriam aplicadas e o contexto de uso seria diferente.

A Figura 3.11 (a) apresenta a UI de edição gerada sem a aplicação de nenhuma configuração no controlador, como é possível observar, foram criados *widgets* para todos os atributos da classe. A Figura 3.11 (b) apresenta a mesma UI com as configurações definidas no Código 3.20. Neste caso, a utilização do método `withEditFields`, linha 17, criou na UI apenas um campo, para inserção do `Nome`.

```
1  @Controller
2  public class MedicoController {
3
4   @Inject
5   private FormConfiguration cfg;
6
7   @Crud(forClass = Medico.class)
8   public void index(){
9
10  cfg.setTitle("Cadastro de Médico");
11  cfg.setDisplayLabel("idMedico", "Có
        digo");
12  cfg.setDisplayLabel("crm", "CRM");
13
14  String user = "Enfermeiro";
15  if (user.equals("Enfermeiro")){
16    cfg.withColumns("nome");
17    cfg.withEditFields("nome");
18    cfg.setVisibleAddButton(false);
19    cfg.setVisibleDeleteButton(false);
20  }
21 }
22 }
```

Código 3.20: Usando configurações para gerar UIs baseadas em contexto de uso

Figura 3.11: UIs de edição com e sem configurações de contexto de uso

A Figura 3.12 apresenta as UI de listagem geradas sem configuração (a) e com as configurações definidas no Código 3.20 (b).

Na Figura 3.12 (a), a UI de listagem possui todas as ações CRUD (inserir, ver, atualizar e excluir), além disso, todas as colunas (`cpf`, `idMedico`, `nome` e `crm`) estão visíveis e o título da UI corresponde ao nome da classe de domínio. Na Figura 3.12 (b), a UI de listagem foi configurada com o Código 3.20. Nesta UI, as ações de inserir e remover foram removidas, as colunas visíveis agora são apenas o `nome` e o `crm`, e o título da UI foi alterado para "Cadastro de Médico".

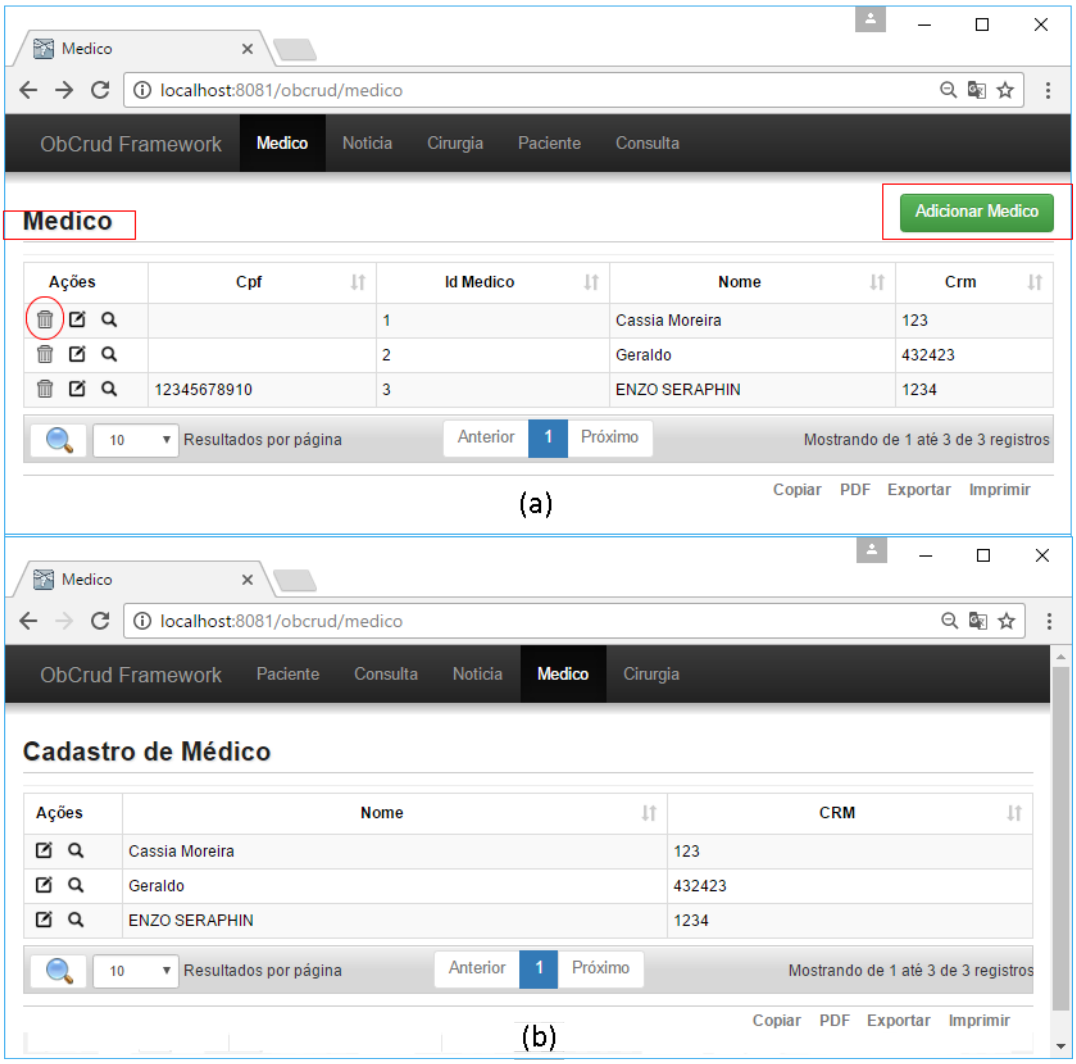

Figura 3.12: UIs CRUD notícia com anotações do modelo de apresentação

3.2.6 Aplicação do modelo de tarefas

Esta seção apresenta a geração de UIs utilizando o Modelo de Tarefas do *framework ObCrud*. Este modelo permite que UIs complexas sejam desenvolvidas e que os códigos individuais da aplicação sejam incorporados as UIs CRUD.

3.2.6.1 Adicionando tarefas

As tarefas básicas, de uma aplicação orientada a negócio, são as funcionalidades CRUD, entretanto, na maioria das vezes, essas tarefas são insuficientes para construir uma aplicação completa. Grande parte das ferramentas que implementam a técnica de *scaffolding*, quando permitem acoplar novas funcionalidades as UIs CRUD, exigem que alterações manuais sejam realizadas no código gerado. Como visto, essas alterações podem ser perdidas se a ferramenta de *scaffolding* for executada novamente. O *ObCrud* implementa o Modelo de

Tarefas, por meio da interface `FormConfiguration` e dos interceptadores. O Código 3.21 demostra a inclusão de uma nova tarefa na UI CRUD utilizando o `FormConfiguration`.

```
1  @Controller
2  public class MedicoController {
3
4      @Inject private FormConfiguration config;
5
6      @Crud(forClass = Medico.class)
7      public void index(){
8          config.setTitle("Cadastro de Médico");
9          config.addFormTask(new BasicTask("Inativar", "inactive"));
10     }
11     @Get("/medico/inactive/{medico.idMedico}")
12     public void inactive (Medico medico){
13         System.out.println("Id do médico " + medico.getIdMedico());
14     }
15 }
```

Código 3.21: Classe de domínio `Cirurgia`

A linha 9 cria uma nova tarefa utilizando o construtor da classe `BasicTask`, que implementa a interface `Task`. O construtor recebe por parâmetro um rótulo (Inativar) e o nome do método `inactive` a ser executado na classe controladora. A linha 12 define rota utilizada para acessar o método e o parâmetro a ser preenchido. A linha 14 imprime o valor do atributo `idMedico` do objeto Médico selecionado na UI.

A Figura 3.13 apresenta a UI CRUD gerada pelo Código 3.21. É possível observar que na coluna ações foi adicionado um novo *link* "Inativar" que corresponde a tarefa adicionada na linha 9 do Código 3.21. O *ObCrud* fornece outras sobrecargas para o construtor `BasicTask`, de forma que seja possível adicionar imagens e estilos CSS (*Cascade Style Sheet*) ao *link* criado, ou modificar o método HTTP utilizado para efetuar a requisição ao método do *controller*.

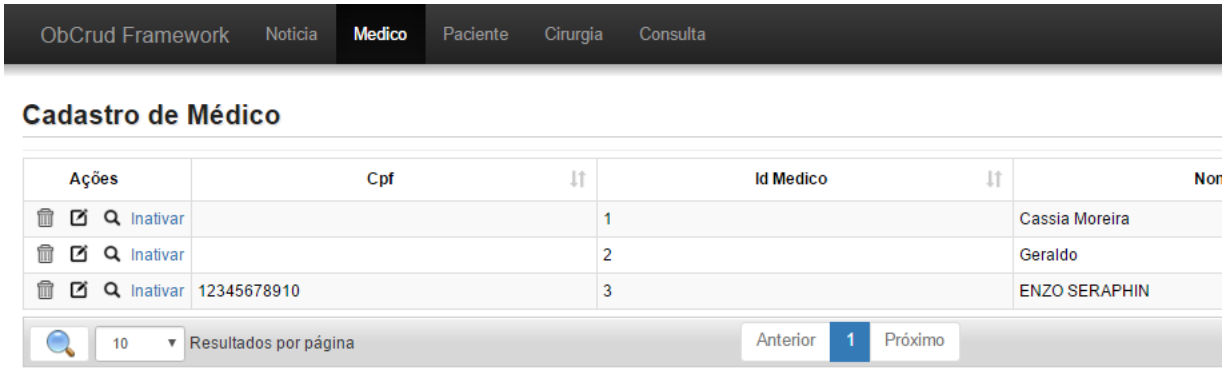

Figura 3.13: UIs CRUD com novas tarefas

3.2.6.2 Adicionando Interceptadores

Os interceptadores possibilitam que ações sejam realizadas antes ou após a execução de uma ação CRUD. Os interceptadores são as precondições e pós-condições previstas no Modelo de Tarefas. No *ObCrud* as precondições e pós-condições são disponibilizadas por meio dos tipos de anotações: `@AfterDeleteObCrud`, `@AfterInsertObCrud`, `@AfterUpdateObCrud`, `@BeforeDeleteObCrud`, `@BeforeInsertObCrud` e `@BeforeUpdateObCrud`. Esses tipos são aplicados aos métodos do controlador que devem executados antes ou depois da execução de ação CRUD.

O Código 3.22 apresenta um exemplo de uso de uma precondição. Na linha 9 foi aplicada a anotação `@BeforeInsertCrud` no método `toUpperCase`, dessa forma, antes de executar a inserção de um novo médico, o método `toUpperCase` deve ser executado. O método `toUpperCase` recebe automaticamente por parâmetro o objeto que foi preenchido na UI de cadastro, esse objeto (médico) pode ser manipulado livremente pelo desenvolvedor, no exemplo o atributo nome foi convertido para letras maiúsculas. Após executar todas as precondições e/ou modificações no objeto, ele é retornado pelo método prosseguindo com a execução da tarefa de inserção. O objeto retornado pelo método será persistido na base de dados.

```
1  @Controller
2  public class MedicoController {
3
4      @Crud(forClass = Medico.class)
5      public void index(){
6
7      }
8
9      @BeforeInsertObCrud
10     public Medico toUpperCase(Medico medico){
11         medico.setNome(medico.getNome().toUpperCase());
12         return medico;
13     }
14 }
```

Código 3.22: *Controller* com Interceptador

A Figura 3.14 (a) apresenta a UI de cadastro de médico com o nome do médico escrito em letras minúsculas, após salvar o objeto, a precondição é executada e o nome do médico é armazenado em letras maiúsculas, o resultado é mostrado na Figura 3.14 (b).

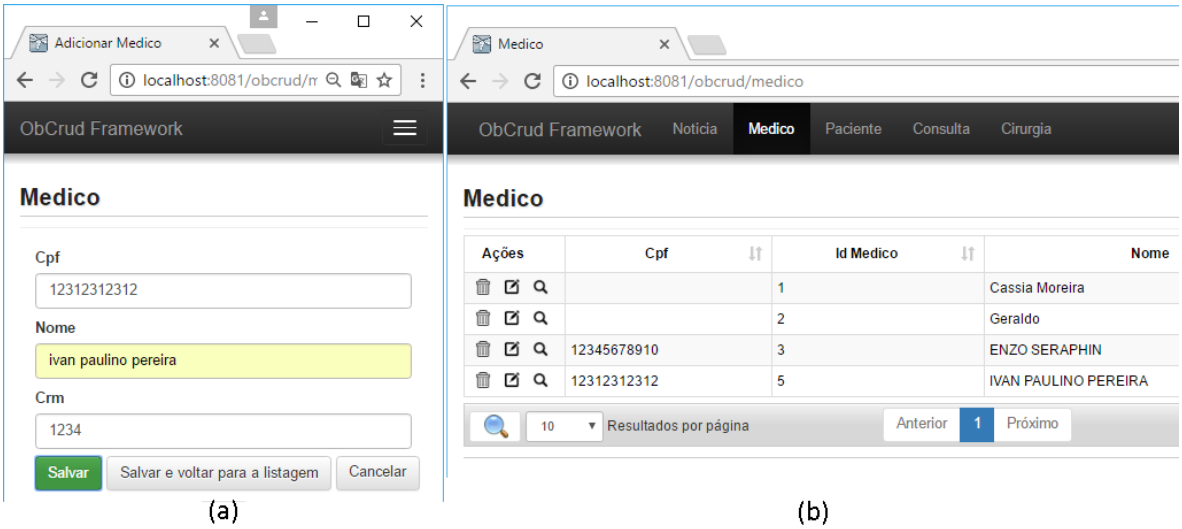

Figura 3.14: UIs CRUD antes e após a execução de uma precondição

3.3 Considerações finais

Este capítulo apresentou a arquitetura lógica e as interfaces que compõem a modelagem do *framework ObCrud*. Foi demostrado o processo de desenvolvimento de UIs CRUD utilizando o *framework*, além disso, foram apresentadas diversas situações de geração de UI utilizando os Modelos de Domínio, Tarefa e Apresentação.

Os modelos utilizados no *framework ObCrud* torna-o uma ferramenta superior as atuais ferramentas de *scaffolding*, sem que sua configuração e utilização seja complexa, como as ferramentas MBUIDEs. Ele utiliza os diversos padrões e especificações já adotados pela comunidade de desenvolvimento para gerar as UIs de forma que o desenvolvedor não necessite aprender novas linguagens ou técnicas de modelagem.

As UIs geradas pelo *framework* são adaptáveis ao contexto de uso e possuem boa usabilidade e uma aparência agradável ao usuário.

Capítulo 4

Experimentos e Resultados

Este capítulo apresenta os experimentos realizados para avaliar o *framework ObCrud*. Os experimentos aqui apresentados têm como objetivos avaliar a produtividade do *framework ObCrud*, na geração de UI CRUDs, a usabilidade das UIs CRUDs geradas e avaliar a usabilidade do *framework*, com relação as funcionalidades disponíveis e a facilidade de uso e de aprendizagem, por parte dos participantes. Todos os experimentos foram realizados nos *frameworks ObCrud* e *Grails* permitindo realizar comparações entre as duas ferramentas. Os testes foram executados de forma a fornecer aos usuários interessados nos *frameworks* subsídios para utilizá-los e para replicar os experimentos aqui realizados.

A Seção 4.1 apresenta o perfil dos participantes envolvidos nos experimentos e a descrição do ambiente onde os experimentos foram realizados. A Seção 4.2 descreve os procedimentos envolvidos na realização dos experimentos. A Seção 4.3 apresenta a análise dos dados e os resultados obtidos com os experimentos. Por fim, a Seção 4.4 apresenta as considerações finais sobre o capítulo.

4.1 Perfil dos participantes e do ambiente de testes

Participaram dos experimentos 8 (oito) discentes da Universidade Federal de Itajubá (UNIFEI), sendo 2 (dois) discentes da graduação e 6 da pós-graduação. Dos participantes apenas 2 (dois) ainda não atuaram no mercado de trabalho, na área de tecnologia da informação. Os participantes possuíam conhecimentos intermediário em Linux, Orientação a Objetos e na Linguagem Java, nas demais tecnologias e *softwares* utilizados nos experimentos, eles não possuíam conhecimento. O Apêndice G apresenta o questionário do perfil dos participantes.

Os experimentos foram realizados no Laboratório de Segurança e Engenharia de Redes (LASER) da UNIFEI. Todos os computadores desse laboratório possuem a seguinte configuração de *hardware*: processador Intel Xeon X 3450 – 2,66GHZ - 8MB de cache – 4 núcleos e 8 threads, 8GB de memória RAM DDR3 1333MHZ Dual Channel com ECC; disco rígido de 466GB SATA2; placa de vídeo NVIDIA QUADRO 600; Sistema Operacional Microsoft Windows 8.1 Professional.

Com o objetivo de manter a mesma configuração de *hardware* e *software* dos computadores utilizados nos experimentos, a fim de eliminar ou mitigar possíveis erros nos dados amostrados foi gerada uma máquina virtual, utilizando o software *Virtual Box*. A máquina virtual foi construída utilizando 6GB de memória RAM, 4 núcleos do processador e 128MB de memória de vídeo, além disso, foram instalados os seguintes *softwares* utilizados na realização dos experimentos.

- Sistema Operacional Ubuntu 16.04
- Java Standard Edition Development Kit 1.8.0 111
- Groovy 2.4
- IDE Eclipse Neon version 4.6.1
- WildFly 8.1.0
- Maven 3.3.9
- Framework Grails 2.5.1
- VRaptor 4.1.4
- MySQL 5.7.16.

4.2 Metodologia

Para avaliar as caraterísticas de produtividade e usabilidade do *ObCrud* foram realizados dois experimentos comparativos entre si, o primeiro experimento foi realizado com o *framework Grails* e o segundo com o *framework ObCrud*. Os experimentos foram estruturados em três fases, sendo elas: a fase de treinamento, a fase da execução e a fase de avaliação. As fases de treinamento e de execução eram compostas por 2 (duas) atividades e a fase de avaliação era composta por 3 (três) atividades.

As atividades da fase de treinamento e execução tinham objetivos semelhantes, mas características diferentes. Na fase de treinamento foi exposto aos participantes os objetivos dos testes, bem como foi apresentado e demonstrado o uso das tecnologias utilizadas nos experimentos. Na fase de execução, os participantes aplicaram os conhecimentos obtidos, na fase anterior, para construir e avaliar as UIs CRUDs, nesta fase os tempos gastos pelos participantes foram registrados.

A fase de avaliação foi realizada após a conclusão das fases de treinamento e de execução dos dois experimentos. Seu objetivo foi avaliar a usabilidade das UIs CRUDs produzidas

pelos *frameworks*, como também, avaliar a usabilidade dos próprios *frameworks* utilizados nos experimentos.

A Figura 4.1 apresenta a estrutura dos experimentos realizados. Os símbolos T1 e T2 representam respectivamente os tempos que cada participante utilizou para gerar uma UI CRUD e para testar as funcionalidades CRUD da UI. A seções seguintes descrevem em detalhes as fases e as atividades realizadas nos experimentos.

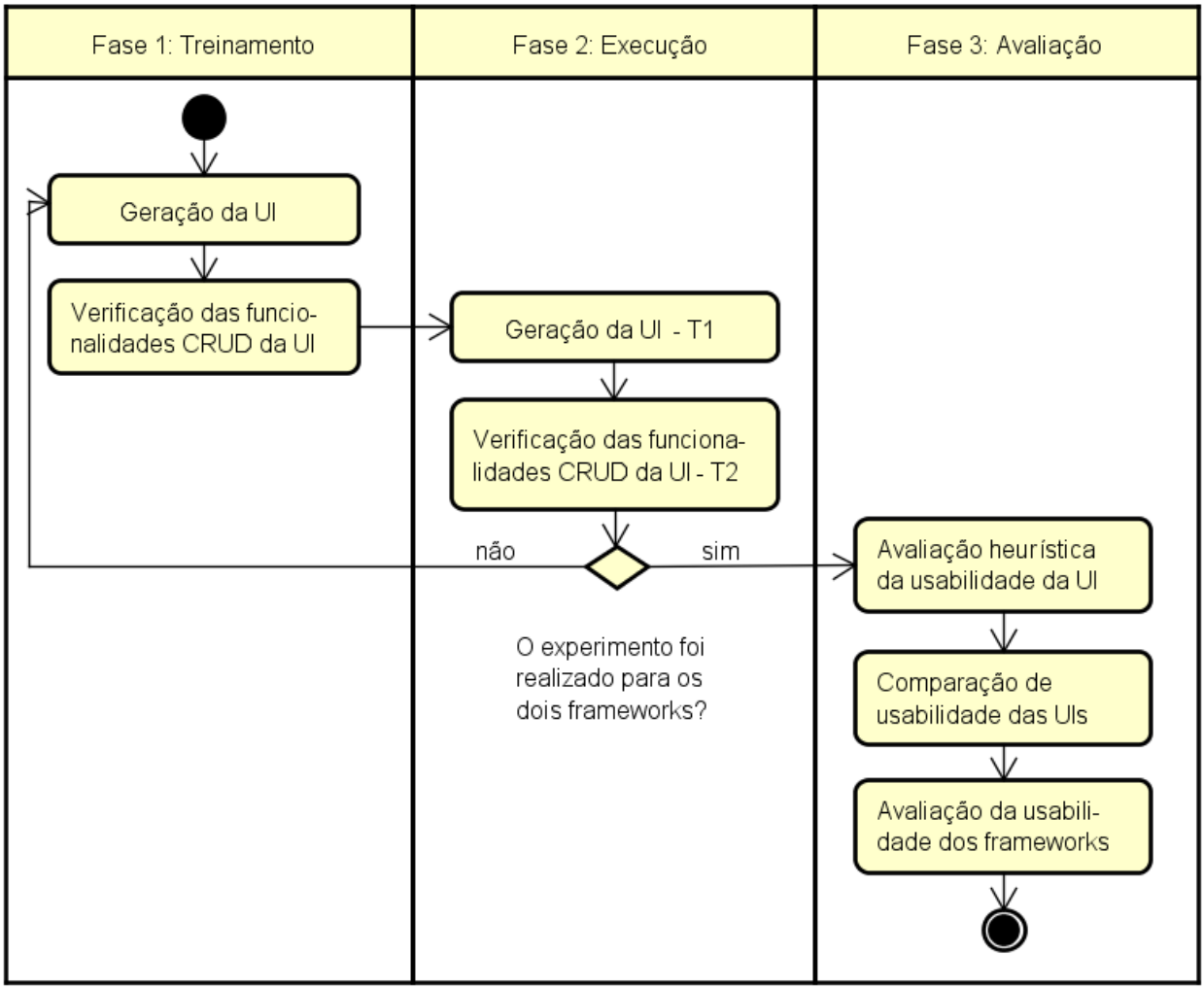

Figura 4.1: Fases e atividades do experimento

4.2.1 Fase de Treinamento

A fase de treinamento tem por objetivo apresentar aos participantes as tecnologias e os *softwares* utilizados nos experimentos. Nesta fase, o avaliador realiza os passos de um roteiro experimental para construir e avaliar as UIs CRUDs, enquanto isso, os participantes, ao mesmo tempo que observam, também constroem e avaliam as UIs CRUDs, acompanhando o avaliador e os passos do roteiro. O roteiro experimental está disponível no Apêndice A.

O avaliador, durante a fase de treinamento, fica à disposição dos participantes para esclarecer eventuais dúvidas e para auxiliar na identificação e correção dos eventuais erros cometidos pelos participantes.

Na primeira atividade desta fase, os participantes, juntamente com o avaliador, construíram as UIs CRUDs para as classes do modelo de domínio, apresentado Figura 4.2. A Figura 4.2 apresenta o modelo de domínio de uma clínica médica. A clínica realiza diversas consultas, cada consulta é realizada em uma data e horário, por um médico e com um paciente.

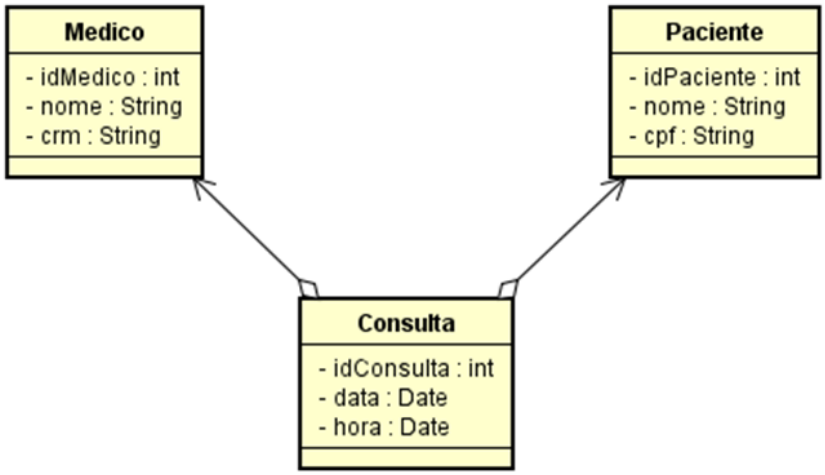

Figura 4.2: Modelo de Domínio utilizado na 1ª atividade da fase de treinamento

Na segunda atividade o avaliador demonstrou, aos participantes, como utilizar as UIs CRUDs geradas, realizando a persistência dos dados do diagrama de objetos da Figura 4.3. Apesar das UIs geradas serem intuitivas e não necessitarem de explanações sobre seu uso e funcionamento, esta atividade foi realizada com o objetivo de verificar se as funcionalidades geradas estavam de acordo com o esperado. Além disso, ela possibilitou que os participantes analisem as UIs, para posteriormente avaliar sua usabilidade.

A Figura 4.3 apresenta que a consulta de código 1 realizada no dia 14/12/2016 as 09h00 foi realizada pelo médico `m1`: José da Silva e teve como paciente `p1`: a Maria Aparecida.

A fase de treinamento é importante para obtenção de dados mais fidedignos nos experimentos. Isso porque, a princípio, os participantes não possuem conhecimento acerca das tecnologias e dos *frameworks* utilizados nos experimentos. Apesar da fase de treinamento não ser o suficiente para que os participantes se tornem fluentes nas tecnologias utilizadas, ela fornece aos participantes o conhecimento necessário para que eles possam executar os experimentos, de forma que o tempo utilizado pelos participantes seja próximo ao tempo utilizado por desenvolvedores experientes.

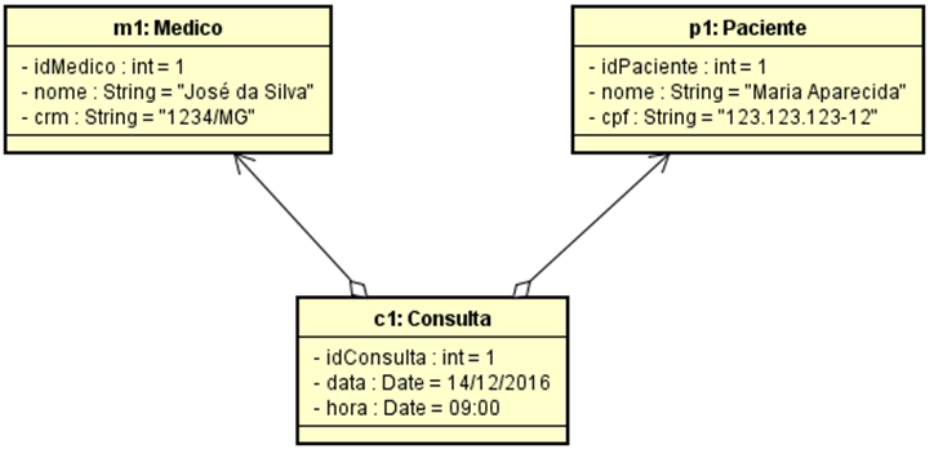

Figura 4.3: Diagrama de objetos utilizado na 2ª atividade da fase de treinamento

4.2.2 Fase de Execução

A fase de execução tem por objetivo medir os tempos utilizados pelos participantes para construir e avaliar as UIs CRUD. As atividades da fase de execução são semelhantes às da fase de treinamento, entretanto, agora os tempos que cada participante utilizou para realizar as atividades são registrados. O modelo de domínio e o diagrama de objetos, utilizados nesta fase, também são diferentes da fase de treinamento.

As atividades das fases de treinamento e execução devem ser semelhantes para que os participantes possam aplicar os conhecimentos adquiridos na primeira fase, com isso, os tempos registrados pelos participantes tendem a ser mais precisos.

A fase de execução também se difere da fase de treinamento, pelo fato do avaliador não realizar qualquer intervenção nas atividades realizadas pelos participantes. Dessa forma, o avaliador não fornece qualquer tipo de auxílio e não responde eventuais dúvidas dos participantes. Nesta fase, os participantes podem consultar o roteiro experimental e/ou outras fontes de informação.

Na primeira atividade, da fase de execução, os participantes construíram as UIs CRUD para as classes do modelo de domínio apresentadas na Figura 4.4. Ao iniciar o desenvolvimento, os participantes iniciaram um cronômetro para registrar o tempo decorrido.

A Figura 4.4 apresenta o modelo de domínio de um clube de futebol. Uma equipe (Time) pode realizar diversos contratos com diversos treinadores, cada contrato possui uma data de início e de término e é celebrado entre uma equipe e um treinador.

Os participantes, ao concluírem as atividades, paralisavam o cronômetro e registravam o tempo T1. Após, iniciava a segunda atividade que consistia na avaliação das funcionalidades CRUD das UI geradas. Para testar as funcionalidades, os participantes persistiram e manipularam os objetos do diagrama apresentado na Figura 4.5. Ao iniciar a execução

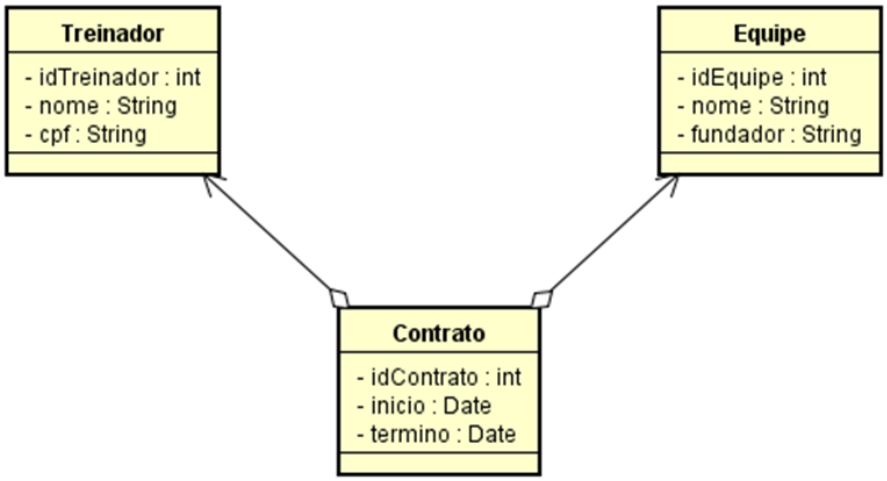

Figura 4.4: Modelo de Domínio utilizado na 1ª atividade da fase de execução

da segunda atividade o cronômetro foi reiniciado para contar o novo tempo, ao finalizar a atividade, o cronômetro foi paralisado e o tempo (T2) foi registrado.

A Figura 4.5 apresenta que o objeto contrato `c1` foi celebrado no dia 11/10/2016 com encerramento previsto para o dia 31/12/2017, este contrato foi celebrado pela equipe `e1`, Sport Club Corinthians, e pelo treinador `t1`, Oswaldo de Oliveira.

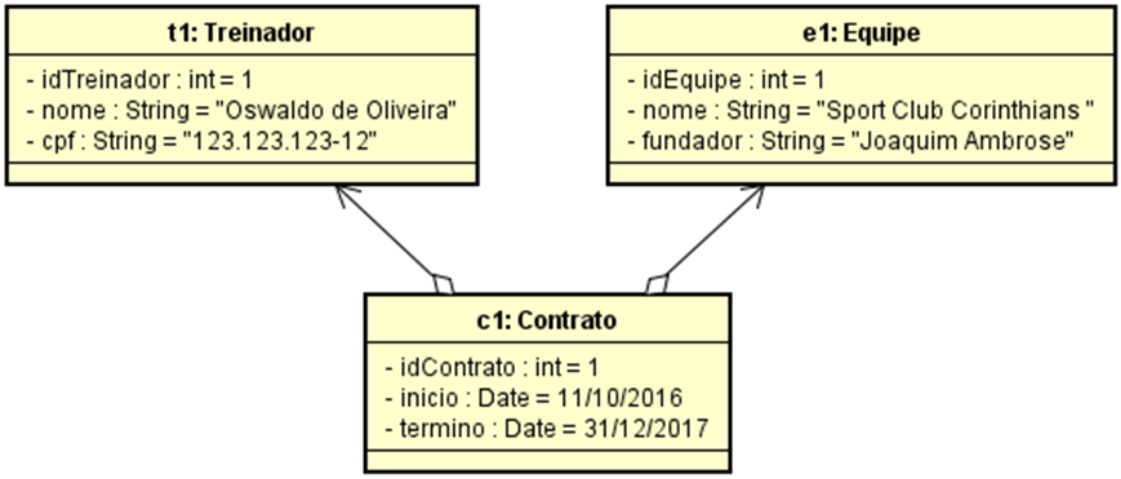

Figura 4.5: Diagrama de objetos utilizado na 2ª atividade da fase de execução

4.2.3 Fase de Avaliação

A fase de avaliação tem o objetivo de avaliar e comparar a usabilidade das UI CRUDs geradas pelos *frameworks ObCrud* e *Grails* e avaliar a usabilidade dos próprios *frameworks*. Esta etapa ocorreu após os participantes terem concluído as fases de treinamento e de execução, nos dois *frameworks*. Isto possibilitou que a avaliação fosse realizada com maior precisão, visto que, os participantes já conheciam os *frameworks* e as UIs criadas por eles.

A fase de avaliação foi composta de 3 atividades avaliativas. A primeira foi a avaliação da usabilidade das interfaces produzidas pelos *frameworks*, por meio das heurísticas de Nielsen e Mack (1994). Na segunda atividade foi realizada a comparação entre as interfaces produzidas por cada *framework*. Por fim, a última atividade realizou a avaliação de usabilidade dos próprios *frameworks*. As seções seguintes descrevem cada uma das atividades realizadas.

4.2.3.1 Avaliação heurística da UI

Nesta atividade foi realizada a avaliação de usabilidade das UIs produzidas pelos *frameworks Grails* e *ObCrud*, por meio das heurísticas de Nielsen e Mack (1994). A avaliação heurística consiste em avaliar uma interface de acordo com uma lista pré-definida de regras, ou de acordo com a experiência dos avaliadores, em busca de descobrir potenciais problemas nas interfaces, de forma econômica, rápida e fácil (NIELSEN; MOLICH, 1990).

Nielsen (1990) produziu uma lista com 9 (nove) heurísticas, que posteriormente foram refinadas, com base na análise fatorial de 249 problemas de usabilidade (NIELSEN, 1994), desse refinamento surgiu as 10 heurísticas de Nielsen e Mack (1994). As heurísticas de Nielsen são amplamente utilizadas na literatura para a identificação de problemas de usabilidade.

Antes de iniciar a avaliação das UIs, o avaliador apresentou as heurísticas de Nielsen aos participantes, orientando-os sobre como proceder a avaliação de usabilidade. Os participantes receberam um formulário, onde registraram todos os problemas de usabilidade encontrados nas UI de cada *framework*. Após, foi realizada a categorização dos problemas encontrados, utilizando as heurísticas e os graus de severidade, todos propostos por Nielsen (1995), Nielsen e Mack (1994). A ficha de avaliação heurística está disponível no Apêndice E.

O grau de severidade é utilizado para classificar os problemas de usabilidade e estimar o esforço necessário para solucionar tais problemas, é também utilizado para avaliar o impacto desses problemas no mercado, de forma a evitar uma possível queda de popularidade do produto (NIELSEN, 1995).

Nesta avaliação, o grau de severidade é utilizado apenas para fins de análise dos dados. A escala de classificação proposta por Nielsen (1995) vai de 0 a 4, sendo que 0 não é considerado um problema de usabilidade e 4 é uma catástrofe de usabilidade, que impede o lançamento do produto para o mercado.

4.2.3.2 Comparação da usabilidade entre as UIs

Esta atividade teve por objetivo realizar uma comparação entre as duas UIs geradas. Os participantes responderam um questionário com 3 questões, cujas respostas eram dadas em escala de Likert, variando de -5 a 5+, sendo -5 a pior avaliação e +5 a melhor avaliação. As seguintes questões foram aplicadas.

a) A interface produzida pelo *framework ObCrud*, em termos de usabilidade, é melhor ou pior, que a interface produzida pelo *framework Grails*?

b) A interface produzida pelo *framework ObCrud* é melhor ou pior esteticamente que a interface produzida pelo *framework Grails*?

c) A interface produzida pelo *framework ObCrud*, de forma geral, é melhor ou pior que a produzida pelo *framework Grails*?

4.2.3.3 Avaliação da usabilidade dos frameworks

Esta atividade buscou avaliar a usabilidade dos *frameworks ObCrud* e *Grails*, essa avaliação é importante, pois analisa se o sistema é de fácil utilização e aprendizado. É importante que o *framework* produza boas UI e que elas possuam boa usabilidade, como também é importante que o *framework* seja amigável ao desenvolvedor, de fácil utilização e aprendizagem. Um *framework* com baixa usabilidade está fadado ao fracasso. Para avaliar a usabilidade dos *frameworks* optou-se por utilizar o questionário *System Usability Scale* (SUS) criado por Brooke (1996). O questionário SUS, aplicado neste trabalho, está disponível no Apêndice F.

O questionário SUS é formado por 10 questões em escala de *Likert*, variando entre 1 e 5, onde 1 é "Discordo Totalmente" e 5 é "Concordo Totalmente". As questões do questionário SUS são classificadas em dois grupos: as de número par e as de número ímpar. As questões de número par, são questões negativas e evidenciam problemas no sistema, assim quanto maior o valor na escala, pior é a avaliação do usuário. As questões ímpares são questões positivas, que indicam qualidade do sistema, assim quanto maior o valor na escala, melhor é a avaliação do usuário.

4.3 Análise e resultados

Esta seção apresenta a análise dos dados e os resultados obtidos nos experimentos realizados, ela está organizada da seguinte forma. A Subseção 4.3.1 apresenta a análise e os resultados referentes aos experimentos de produtividade de geração de UIs. A Subseção 4.3.2 apresenta a análise e os resultados da avaliação heurística das UIs geradas. A

Subseção 4.3.3 apresenta a análise e os resultados da comparação entre as UIs geradas por cada um dos *frameworks* e a Subseção 4.3.4 apresenta a análise e os resultados da avaliação de usabilidade dos *frameworks ObCrud* e *Grails*.

4.3.1 Avaliação da produtividade

Com objetivo de avaliar, em termos estatístico, se a produtividade do *framework ObCrud* é similar ao do popular *framework Grails*, os experimentos realizados, neste trabalho, coletaram os tempos que os participantes gastaram para construir as UIs CRUD *T1* e o tempo gasto para utilizá-las, verificando suas funcionalidades CRUD *T2*.

Assumindo que as amostras foram coletadas em uma população não distribuída normalmente, e dadas as características dos experimentos, realizados dentro do grupo (*within group*), com amostras pareadas, o método adequado e utilizado para testar as hipóteses é o *Wilcoxon Signed Rank* (WILCOXON, 1945).

O método de *Wilcoxon* é um método não paramétrico geralmente utilizado para comparação de duas amostras pareadas, ou para comparação de uma única amostra. Nas amostras pareadas, a diferença entre cada par de dados é calculada, sendo que os valores resultantes podem aumentar, diminuir ou manter-se iguais. Nas amostras únicas, também é realizado a diferença entre os valores, no entanto, deve ser fornecido um parâmetro que será utilizado para formar a hipótese nula. Nos dois casos os valores do resultado da subtração são ordenados por seus valores absolutos. Em seguida, esses valores são substituídos pelos valores das posições que ocupam, caso exista valores repetidos é atribuída a média dos postos dos valores repetidos. Uma vez criado o *rank* das posições, é atribuído o sinal do valor ao posto correspondente, então são somados os postos negativos e os postos positivos. Por fim, é calculada a probabilidade de aceitar ou não a hipótese nula. As hipóteses são estabelecidas, conforme apresentado na Equação 4.1.

$$\begin{cases} H_0: & \Delta = 0 \\ H_1: & \Delta \neq 0 \end{cases} \quad \begin{cases} H_0: & \Delta = 0 \\ H_1: & \Delta > 0 \end{cases} \quad \begin{cases} H_0: & \Delta = 0 \\ H_1: & \Delta < 0 \end{cases} \tag{4.1}$$

As hipóteses podem ser definidas para uma ou duas caldas. Quando se avalia com base em uma calda, considera-se na hipótese nula H_0 que a diferença da mediana é nula e nas hipóteses alternativas H_1 que a diferença da mediana é menor ou maior que 0. Com duas caldas, na hipótese nula H_0, a diferença da mediana é 0 e na hipótese alternativa H_1 a diferença da mediana é diferente de 0.

O *p-value* é a probabilidade obtida com a aplicação do método de *Wilcoxon*. Quanto menor for o valor de *p-value* mais indícios existem para rejeitar a hipótese nula H_0, e quanto

maior for o valor de *p-value* mais indícios existem para aceitar a hipótese nula H_0. O valor de *p-value* varia entre 0 e 1, ou seja, de 0 a 100%. Considera-se neste trabalho um nível de significância de 95% de confiança, dessa forma, se o *p-value* obtido for menor que 0,05 a hipótese nula H_0 é rejeitada e a hipótese alternativa H_1 é aceita.

Para a avaliação da produtividade dos *frameworks ObCrud* e *Grails* na geração de UIs CRUDs foram definidas as seguintes hipóteses.

- **H_0**: Não existe diferença, estatisticamente significante, entre os *frameworks Grails* e *ObCrud* em termos de produtividade na geração automática de UIs CRUDs.
- **H_1**: Existe diferença, estatisticamente significante, entre os *frameworks Grails* e *ObCrud* em termos de produtividade na geração automática de UIs CRUDs.

Aplicado o método de *Wilcoxon* sobre os tempos *T1* coletados nos experimentos de produtividade na geração de UIs CRUDs, com os dois *frameworks*, obteve-se o *p-value = 0,9453*. Uma vez que, o *p-value* calculado foi superior que 0,05, a hipótese nula H_0 foi aceita. Desta forma, pode-se concluir que, estatisticamente, a produtividade dos dois *frameworks* é equivalente.

A Figura 4.6 apresenta o gráfico dos tempos *T1* gastos para gerar as UIs CRUDs. Em relação a mediana, o tempo utilizado pelos participantes para gerar as UI CRUDs utilizando o *Grails* foi levemente inferior ao tempo gasto no *ObCrud*. Além disso, os tempos obtidos no *ObCrud* são mais dispersos do que no *Grails*. Observa-se também a existência de 2 *outlier* no *Grails*, um na parte inferior e outro na superior. Apesar da existência de *outliers* nas amostras, eles não inviabilizam o teste das hipóteses, ao contrário esses dados devem ser considerados, uma vez que, o teste de *Wilcoxon* não leva em consideração a média, mas sim a posição dos valores em relação a mediana para efetuar os cálculos estatísticos.

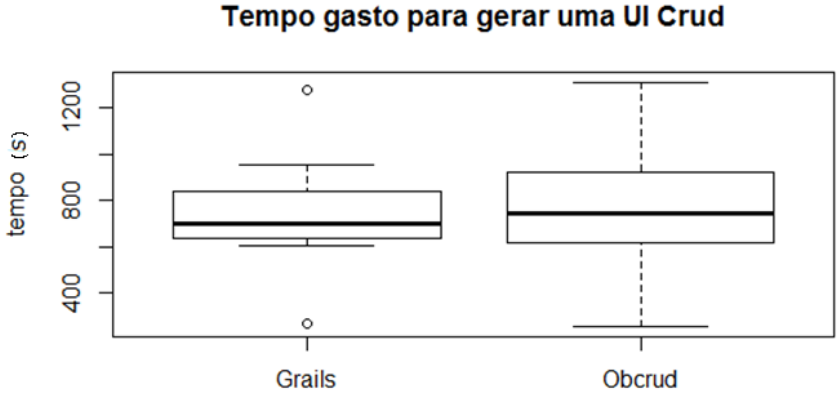

Figura 4.6: Tempos *T1* gastos para gerar as UIs CRUDs

Com relação a produtividade dos participantes no uso das funcionalidades CRUD das UI geradas pelos *frameworks ObCrud* e *Grails* foram definidas as seguintes hipóteses.

- **H_0**: Não existe diferença, estatisticamente significante, entre as UIs geradas pelos *frameworks Grails* e *ObCrud*, no que tange a produtividade dos usuários na sua utilização.
- **H_1**: Existe diferença, estatisticamente significante, entre as UIs geradas pelos *frameworks Grails* e *ObCrud*, no que tange a produtividade dos usuários na sua utilização.

Aplicado o método de *Wilcoxon* sobre os tempos *T2* coletados nos experimentos de produtividade na utilização das funcionalidades CRUDs das UIs geradas pelos dois *frameworks*, obtivemos o *p-value = 0,1953*. Uma vez que o *p-value* é superior que 0,05, a hipótese nula H_0 foi aceita. Desta forma, pode-se concluir que, estatisticamente, a produtividade dos usuários, ao utilizar as UIs geradas pelos dois *frameworks* são equivalentes.

A Figura 4.7 apresenta o gráfico relativo ao tempo *T2* gasto pelos participantes para utilizar e verificar as funcionalidades CRUD das UIs. Neste ponto, analisando a mediana, verificamos que os participantes precisaram de menos tempo para utilizar as UIs geradas pelo *ObCrud*. Observa-se ainda que a distribuição dos dados está próxima a mediana, o que evidenciou um *outlier*.

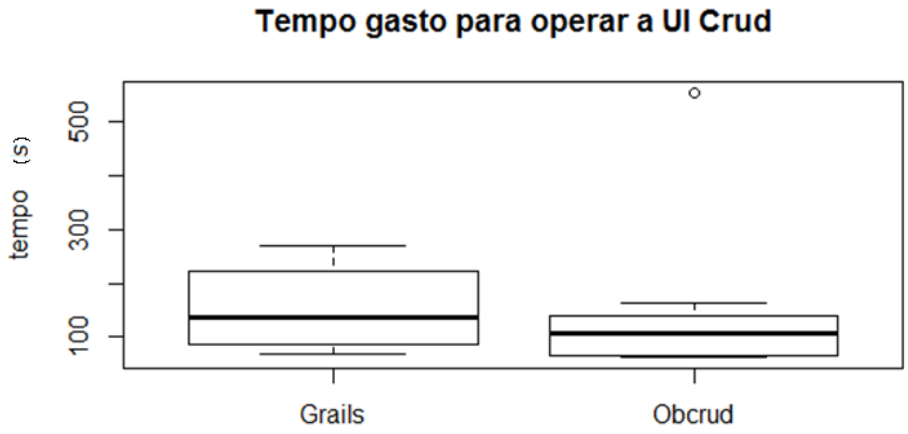

Figura 4.7: Tempos *T2* gastos para gerar as UIs CRUDs

4.3.2 Avaliação heurística da UI

As avaliações realizadas, pelos participantes, nas UI CRUDs, com base nas heurísticas de Nielsen e Mack (1994), apresentaram que as UIs geradas praticamente não apresentam problemas de usabilidade. Os participantes não encontraram nenhum problema de usabilidade nas UIs geradas pelo *framework Grails*. Nas UIs geradas pelo *ObCrud* um participante relatou um problema de usabilidade, conforme mostra a Tabela 4.1.

Tabela 4.1: Problemas de usabilidade identificados no *ObCrud*

Problema relatado	Heurística	Grau de severidade
Os botões de edição e exclusão poderiam além dos ícones possuir um texto referente a sua função.	8	2

O problema relatado pelo participante foi categorizado como heurística 8 que trata da estética e do *design* minimalista, quanto a severidade, o problema foi classificado com grau 2, que indica um problema de usabilidade menor.

O problema encontrado não chega a ser considerado um problema de usabilidade, uma vez que diversos sistemas, utilizam em suas barras de ferramentas, apenas ícones para identificar as funcionalidades. O *ObCrud* apesar de utilizar apenas ícones, para as ações de atualização e remoção de registros, apresentava informações do ícone, ao usuário, quando o cursor do mouse era posicionado sobre o elemento. Mesmo considerando que a UI do *ObCrud* apresentou um possível problema de usabilidade, pode-se considerar que, por ser um único problema relatado, e pelo baixo grau de severidade atribuído, as duas interfaces possuem boa usabilidade e atendem as heurísticas de Nielsen e Mack (1994).

4.3.3 Comparação da usabilidade da UI

Para realizar a comparação de usabilidade das UIs, geradas pelos *frameworks ObCrud* e *Grails*, utilizamos o método *Wilcoxon Sum Rank* (para amostra única), descrito Subseção 4.3.1. O nível de significância utilizado também foi de 95% de confiança. Para cada uma das 3 (três) questões realizadas foram definidas as seguintes hipóteses.

Questão 01: A interface produzida pelo *framework ObCrud*, em termos de usabilidade, é melhor ou pior, que a interface produzida pelo *framework Grails*?

- $\mathbf{H_0}$: Não existe diferença, estatisticamente significante, entre as usabilidades das UIs CRUDS geradas pelos os *frameworks Grails* e *ObCrud*.
- $\mathbf{H_1}$: Estatisticamente, a usabilidade da UI gerada pelo *framework ObCrud* é superior a usabilidade da UI gerada pelo *framework Grails*.

Questão 02: A interface produzida pelo *framework ObCrud* é melhor ou pior, esteticamente, que a interface produzida pelo *framework Grails*?

- $\mathbf{H_0}$: Não existe diferença, estatisticamente significante, entre a estética das UIs geradas pelos os *frameworks Grails* e *ObCrud*.

- **H_1**: Estatisticamente, a estética da UI gerada pelo *framework ObCrud* é superior a estética da UI geradas pelo *framework Grails*.

Questão 03: A interface produzida pelo *framework ObCrud*, de uma forma geral, é melhor ou pior que a produzida pelo *framework Grails*?

- **H_0**: De forma geral, não existe diferença estatisticamente significante, entre as UIs geradas pelos os *frameworks Grails* e *ObCrud*.
- **H_1**: Estatisticamente, a UI gerada pelo *framework ObCrud* é superior a UI gerada pelo *framework Grails*.

A Tabela 4.2 apresenta o resultado dos testes estatísticos de *Wilcoxon* para as questões analisadas. Para todas as questões, o valor de *p-value* foi inferior a 0,05. Dessa forma, rejeitou-se todas as hipóteses nulas H_0 e concluiu-se que, estatisticamente, a UI gerada pelo *ObCrud* é superior a UI gerada pelo *Grails* em termos de estética e de usabilidade.

Tabela 4.2: Resultado do teste estatístico para comparação das UIs geradas pelos *frameworks ObCrud* e *Grails*

Questão	p-value	Hipótese nula - H_0	Hipótese alternativa – H_1
Q1	0,01439	Rejeitada	Aceita
Q2	0,0103	Rejeitada	Aceita
Q3	0,006492	Rejeitada	Aceita

4.3.4 Avaliação de usabilidade dos frameworks

A avaliação de usabilidade dos *frameworks* foi realizada por meio do questionário SUS. Para calcular a usabilidade de um sistema utilizando o questionário, os seguintes procedimentos devem ser realizados (BROOKE, 1996):

- Para as questões ímpares subtraia 1 da resposta do participante;
- Para as questões pares subtraia de 5 a resposta do participante;
- Após as correções, as questões estarão em uma escala de 0 a 4, sendo 4 a resposta mais positiva;
- Some o valor de todas as respostas normalizadas, o valor total estará entre 0 e 40;
- Multiplique o valor total por 2,5. O resultado final indica a pontuação do questionário SUS, essa pontuação varia de 0 a 100.

Realizado os procedimentos para análise do questionário SUS, obteve-se os resultados apresentados na Tabela 4.3. A Tabela 4.3 apresenta as pontuações do questionário SUS obtidas pelos *softwares ObCrud* e *Grails*. O *framework ObCrud* obteve pontuação média de 86,3 pontos com desvio padrão de 10 pontos, enquanto que o *framework Grails* obteve pontuação média 78,1 com desvio padrão de 9,6.

Tabela 4.3: Pontução SUS dos *frameworks ObCrud* e *Grails*

	Pontuação SUS	
Participante	ObCrud	Grails
1	80,0	82,5
2	75,0	75,0
3	90,0	67,5
4	72,5	70,5
5	97,5	87,5
6	100	95,0
7	85,0	77,5
8	90,0	70,0
Média	86,3	78,1
Desvio Padrão	10,0	9,6

Apesar da pontuação do questionário SUS variar de 0 a 100 pontos, ela não deve ser considerada como uma porcentagem. Para interpretação dos resultados obtidos no questionário SUS, a literatura emprega o estudo realizado por Sauro (2011), que utilizou o questionário SUS para avaliar 500 sistemas diferentes, com participação 5000 usuários na avaliação. O estudo apresentou que a média de pontuação dos sistemas avaliados foi de 68 pontos. Assim, um sistema com usabilidade média deve possuir 68 pontos, ao invés de 50 pontos no caso da porcentagem. Sistemas com pontuação superior a 68 pontos estão acima da média dos sistemas avaliados por Sauro, por conseguinte conclui-se que possuem bons níveis de usabilidade.

Os dois *frameworks* avaliados apresentaram bons níveis de usabilidade, acima da média dos 68 pontos. O *framework ObCrud* teve uma pontuação superior ao do *framework Grails*. Assim, de acordo com os dados obtidos, é possível concluir que a usabilidade do *ObCrud* é superior ao do *Grails*.

4.4 Considerações finais

Nielsen (2012) apresenta que com apenas 5 avaliadores em um estudo de usabilidade é possível encontrar praticamente o mesmo número de problemas de usabilidade que se encontraria com um número maior de avaliadores. Os experimentos realizados neste trabalho contaram coma participação de apenas 8 participantes, devido a dificuldade de aplicação do experimento, por se tratar de um experimento extenso e demorado, bem como pela indisponibilidade de um número maior de participantes. Apesar do número reduzido, os estudos de Nielsen demostram a eficiência da avaliação com um número limitado de participantes.

O uso do método não paramétrico de *Wilcoxon* é aplicável à pequenas amostras, assim os resultados produzidos tendem a ser confiáveis, mesmo contando com um número reduzido de participantes. Todos os participantes eram da área de tecnologia da informação e a maioria já havia atuado no mercado de trabalho, fazendo esta amostra aceitável para a realização dos experimentos.

Os resultados obtidos demostraram que o *ObCrud* possui produtividade equivalente ao *Grails*, esse resultado é considerado positivo, visto que o *Grails* é um *framework* já estabelecido no mercado e utilizado por grandes empresas. No que tange a usabilidade das interfaces, os experimentos mostraram que elas possuem bons níveis de usabilidade e que a usabilidade das UIs geradas pelo *ObCrud* é superior as UIs geradas pelo *Grails*. Por fim, a avaliação de usabilidade dos *frameworks*, mostrou que os dois possuem boa usabilidade e que o *ObCrud* possui usabilidade superior ao do *Grails*.

Capítulo 5

Conclusão

O desenvolvimento de Interfaces do Usuário (UIs) é uma atividade complexa que colabora para baixa produtividade de sistemas (SILVA, 2010). Ao longo dos últimos anos, diversas pesquisas e ferramentas foram produzidas, para lidar com essa complexidade e aumentar a produtividade do desenvolvimento das UIs. Grande parte dessas pesquisas e ferramentas foram realizadas utilizando a abordagem *Model-Based User Interface Development* - MBUID. A abordagem MBUID utiliza diversos tipos de modelos que permitem descrever as diversas nuances das UIs. Os principais modelos utilizados na abordagem MBUID são os modelos de domínio, de tarefas e o de apresentação.

A técnica de *scaffolding*, criada pelo *framework Rails*, também possibilita a criação de UIs, especificamente para operações CRUD, tem ganhado a aceitação dos desenvolvedores. Essa técnica vem sendo empregada em diversos *frameworks* MVC, seu uso geralmente é muito simples e por ser integrada à *frameworks* MVC não é necessário realizar configurações extras para poder utilizá-la. Apesar de aumentar a produtividade, as UIs produzidas pelos *scaffoldings*, geralmente são básicas demais e necessitam que os desenvolvedores realizem adaptações. A técnica de *scaffolding* pode ser vista como uma versão incompleta de uma ferramenta MBUIDE.

Os *scaffoldings* podem ser divididos em duas categorias, os dinâmicos e os estáticos. O *scaffolding* estático gera código-fonte que pode ser modificado pelo programador, tornando possível a modificação de comportamento e a customização das UIs. O problema do *scaffolding* estático é que caso ocorra uma alteração no modelo de domínio, atividade muito comum principalmente nas fases iniciais de desenvolvimento, uma nova execução da ferramenta de *scaffolding* deve ser realizada. A nova execução exclui toda a customização de aparência e comportamento da UI realizada a priori. O *scaffolding* dinâmico, por sua vez, não gera código-fonte, o que limita as possibilidades de customização de comportamento e de aparência das UIs, no entanto, todas modificações realizadas nos modelos são refletidas automaticamente nas UIs em tempo de execução.

O *framework ObCrud*, proposto neste trabalho, teve por objetivo preencher as lacunas entre as ferramentas MBUIDE e os *scaffoldings* estáticos e dinâmicos. As ferramentas MBUIDEs possibilitam a construção de UIs mais complexas e ricas, por utilizar diferentes tipos de modelos. Entretanto, suas ferramentas geralmente são difíceis de utilizar, e como buscam

desenvolver UIs que atendam diversos contextos e dispositivos diferentes, a qualidade das UIs geradas é prejudicada. Os *scaffoldings* são ferramentas de simples utilização, mas que utilizam apenas o modelo de domínio para construir as UIs, com isso as UIs geradas são básicas demais. Os *scaffoldings* estáticos geram maior possibilidade de customização, ao passo que, os *scaffoldings* dinâmicos atualizam automaticamente as UIs quando os modelos são alterados.

O *ObCrud*, assim como os *MBUIDEs*, utiliza os modelos de domínio, de tarefas e de apresentação, possibilitando que UIs melhores e mais complexas sejam criadas e que o desenvolvedor tenha maior possibilidade de customizar o comportamento e a aparência das UIs, como ocorre com o *scaffolding* estático. Por outro lado, o *ObCrud*, produz as UIs CRUD em tempo de execução, como o *scaffolding* dinâmico, de forma que qualquer alteração nos modelos reflita automaticamente nas UI geradas, sem que isso implique em prejuízos nas customizações realizadas anteriormente, como ocorre com os *scaffoldings* estáticos. Além disso, o *ObCrud* é uma ferramenta de simples utilização, possuindo desta forma, as melhores características de cada ferramenta.

Os resultados dos experimentos mostraram que as UIs, geradas pelo *framework ObCrud*, possuem boa usabilidade e são superiores às UIs geradas pelo *Grails*. Os níveis de produtividade nos dois *frameworks* são equivalentes, vale destacar que o estudo realizado por Magno (2015) mostrou que os *scaffolding* são cerca de 91% mais produtivos do que a codificação manual. Portanto, o *ObCrud* possibilita que os desenvolvedores construam UIs de boa qualidade e usabilidade no menor tempo possível

Por fim, os níveis de usabilidade do *framework ObCrud* foram considerados acima da média dos sistemas avaliados por Sauro (2011), bem como foi considerado superior ao do *framework Grails*. Essa característica é importante, visto que a ferramenta se mostrou de fácil utilização e aprendizagem, podendo ser aceita em larga escala pelo mercado.

As tecnologias empregadas no desenvolvimento do *ObCrud* são, em sua maioria, especificações da plataforma Java, ou seja, são consideradas padrões de mercado e aceitas por grande parte dos desenvolvedores, dentre elas temos a Linguagem Java, o *Context Dependency Injection* – CDI, o *Java Persistence API*, o *Bean Validation API*, o *Java Reflection API*, o *VRaptor*, o *RESTful*, *Html 5*, *CSS3* e o *framework Bootstrap*.

5.1 Contribuições

As principais contribuições deste trabalho são:

- Implementação do *framework ObCrud*, que utiliza a abordagem MDD, semelhante aos *scaffoldings*, para gerar automaticamente as UIs CRUD, utilizando os modelos de

domínio, de tarefas e de apresentação, que na maior parte das vezes são subutilizados ou possuem limitações nos *scaffoldings*.

- A opção deste trabalho em utilizar outros modelos para geração de UIs CRUD permitiu maior flexibilidade de customização da aparência e do comportamento das UIs. Esses conceitos podem ser implementados nos *scaffoldings* atuais, possibilitando a evolução dessas ferramentas, e da forma como desenvolvedores constroem as UIs.
- A ferramenta desenvolvida adicionou novos recursos ao *framework VRaptor*, que não possuía o *scaffolding* em sua última versão.

5.2 Dificuldades Encontradas

Muitas dificuldades foram encontradas durante a realização deste trabalho, as dificuldades mais significativas foram:

A princípio o *ObCrud* tinha como objetivo ser a ferramenta de *scaffolding* para o *framework ObInject* (CARVALHO et al., 2013). O *ObInject* é um *framework* para persistência de dados, desenvolvido sob a orientação do mesmo orientador deste trabalho. O *framework* é mais eficiente nas operações de inserção e recuperação de dados do que outros *frameworks* ORM, como, por exemplo, o *Hibernate*. A integração da ferramenta de *scaffolding* com o *ObInject* permitiria maior adesão da comunidade de desenvolvimento, ao *framework* de persistência, uma vez que, simplificaria ainda mais sua utilização para persistência de dados. No entanto, o *ObInject* ainda não possuía implementada todas as operações CRUD, uma vez que faltava ser implementadas as operações de exclusão e atualização. Dessa forma, não foi possível utilizar o *ObInject* como *framework* de persistência para o *ObCrud*.

Antes de iniciar o desenvolvimento do *ObCrud* foi realizado alguns estudos sobre os principais *frameworks Web* MVC. Esses *frameworks* são essenciais para aplicação da técnica de *scaffolding*, pois realizam a separação das responsabilidades em camadas, além de possuir diversos recursos que podem ser reutilizados para construção das UIs CRUD. Muitos dos *frameworks* já possuíam o recurso de *scaffolding*, o que a princípio não foi um problema, uma vez que o presente trabalho pretendia apresentar novos recursos que poderiam ser incorporados as técnicas atuais de *scaffolding*, após a realização dos estudos foi escolhido o *framework ActiveWeb JavaLite*.

O desenvolvimento do *ObCrud* e sua integração com o *framework* MVC exigia um profundo conhecimento, sobre como o último foi construído. Esse conhecimento demandava certo tempo para ser adquirido e não pode ser realizado com todas ferramentas disponíveis, ao escolher o *ActiveWeb* e iniciar o desenvolvimento, esse conhecimento foi sendo construído, conforme a implementação avançava.

Ocorre que devido algumas restrições apresentadas pelo *framework ActiveWeb* não foi possível desenvolver o *ObCrud* para atuar neste *framework*, isso foi percebido após o desenvolvimento do *ObCrud* estar adiantado, isso provocou um certo atraso no desenvolvimento, pois diversas funcionalidades tiveram que ser descartadas e outras recodificadas. Após descartar o *ActiveWeb*, outros *frameworks* MVC foram estudados, como por exemplo o *Play* e o *Grails*, mas em algum ponto do desenvolvimento eles também apresentaram problemas que impediram o avanço do desenvolvimento. Por fim, o *VRaptor* foi avaliado, e com este foi possível implementar o *ObCrud*.

Todos os *frameworks* MVC considerados para a implementação do *ObCrud* são excelentes ferramentas e consideradas padrões de mercado. O *ActiveWeb*, *Play* e o *Grails* foram considerados antes que o *VRaptor*, para o desenvolvimento do *ObCrud*, pois eles realizam o *"automatic build"*, ou seja, após qualquer alteração de código a construção *(build)* do sistema é realizada rápida e automaticamente. No *VRaptor* a construção do sistema é realizada juntamente com a implantação *(deploy)*, o processo de implantação demorava cerca de 2 minutos, na máquina utilizada para o desenvolvimento, um *ultrabook* com processador *core* i3 com 4GB RAM. Essa latência dificultava o desenvolvimento, pois a cada pequena alteração do código era preciso esperar o tempo de implantação, na maior parte das vezes o tempo de espera era maior que o tempo que se passava desenvolvendo. A substituição do computador utilizado para o desenvolvimento resolveu esse problema, foi utilizado, para o restante do desenvolvimento, um *notebook* com processador *core* i7 com 8GB RAM.

Além disso, os *frameworks Grails* e *Play* utilizam comandos de terminal para criar um projeto web, enquanto que no *VRaptor*, o projeto geralmente é construído utilizando o gerenciador de dependências *Maven*. A utilização do *Maven* não é complicada, mas a criação do projeto exige certo conhecimento, além de diversas configurações em arquivos XMLs.

Um dos requisitos não funcionais ao desenvolver o *ObCrud* era facilitar ao máximo o uso da ferramenta pelo desenvolvedor. A configuração do projeto web, incluindo as dependências do *VRaptor* e do *ObCrud*, violavam esse requisito não funcional. Após alguns estudos, optou-se por criar um projeto com todas as dependências e configurações necessárias para uso do *ObCrud*. Esse projeto foi utilizado como *template* (modelo) para criar um *Archetype* do *Maven*. Um *Archetype* é um *template* de projeto que possibilita a criação do ambiente de desenvolvimento de forma mais rápida e prática. Caso essa solução não fosse encontrada, outras alternativas poderiam ser tomadas o que atrasaria ainda mais o desenvolvimento do *framework ObCrud*.

5.3 Trabalhos Futuros

A ferramenta implementada, neste trabalho, possui os recursos necessários para gerar UIs CRUD com qualidade e usabilidade, entretanto, sempre é possível adicionar novas funcionalidades e realizar correções de forma a torna-lá mais madura e estável. Futuros trabalhos podem direcionar o foco para:

- *Internacionalização*: O *ObCrud* ainda não possui um mecanismo para realizar a internacionalização dos idiomas utilizados nas UIs. Apesar de simples, o desenvolvimento dessa funcionalidade ainda não foi implementado.
- *Listener para os Widgets*: Uma funcionalidade importante nas UIs é a possibilidade de ouvir eventos dos *Widgets* e realizar ações. Por exemplo, quando um campo de texto receber o foco, ou quando um item é selecionado em uma caixa de seleção, ou ainda quando o ponteiro do *mouse* passa por um componente. Esses eventos podem desencadear a execução de métodos que executam funcionalidades ou alteram o estado de um *Widget*.
- *Repositório de Templates*: Pode ser criado um repositório de *templates* para que os usuários e desenvolvedores possam selecionar temas que melhor atendam às necessidades dos projetos. Apesar de não ser complexa a criação ou modificação dos *templates* do *ObCrud*, alguns desenvolvedores visando a produtividade, podem optar por utilizar *templates* prontos ao invés de fazer customizações.
- *Ferramenta de edição de templates*: Ainda que não seja complicado a edição de *templates*, a edição manual pode causar erros. Dessa forma, poderia ser criada uma ferramenta para desenvolver os *templates*, igual à do gerador de *templates* utilizada pelo gerenciador de conteúdos *Joomla*.
- *Criação de novas Anotações*: Novos estudos podem avaliar a necessidade da criação de novas anotações visando ampliar a interatividade e a possibilidade de customização das UIs.
- *Ferramenta de autoria de modelos*: O *ObCrud* consiste numa ferramenta de transformação de modelos de código-fonte em UIs. Poderia ser construída ou adaptada uma ferramenta de autoria e de transformação de modelos gráficos (UML) em modelos utilizados pelos *ObCrud*. Dessa forma, o processo de criação de UIs seria ainda mais simples e produtivo.
- *Avaliação de desempenho*: Por gerar as UIs CRUD em tempo de execução, igual aos *scaffoldings* dinâmicos, o *framework ObCrud* exige do servidor de aplicação mais recursos de *hardware*, como memória e processador. Estudos poderiam ser realizados

para analisar se o consumo extra de recursos pode comprometer as funcionalidades dos sistemas gerados. Além disso, melhorias no código da ferramenta poderiam ser realizadas para aproveitar melhor os recursos disponíveis.

Referências Bibliográficas

ARNOLD, K.; GOSLING, J. **A linguagem de programação Java**. [S.l.]: Bookman Editora, 2007.

ATKINSON, C.; KUHNE, T. Model-driven development: a metamodeling foundation. **IEEE software**, IEEE, v. 20, n. 5, p. 36–41, 2003.

ATTARDI, G.; CISTERNINO, A. Reflection support by means of template metaprogramming. In: **Proceedings of the Third International Conference on Generative and Component-Based Software Engineering**. London, UK, UK: Springer-Verlag, 2001. (GCSE '01), p. 118–127. ISBN 3-540-42546-2.

BARTLETT, J. **The art of metaprogramming**. [S.l.], 2005. Disponível em: <`https://www.ibm.com/developerworks/library/l-metaprog1/`>. Acesso em: 20 de março de 2017.

BATORY, D. Product-line architectures. In: **Smalltalk and Java Conference**. [S.l.: s.n.], 1998.

BERNARD, E.; PETERSON, S. Jsr 303: Bean validation. **Bean Validation Expert Group, March**, 2009.

BEZERRA, E. **Princípios de Análise e Projeto de Sistema com UML**. [S.l.]: Elsevier Brasil, 2015.

BÉZIVIN, J. On the unification power of models. **Software and systems modeling**, Springer, v. 4, n. 2, p. 171–188, 2005.

BRASIL. **Qualidade e Produtividade no Setor de Software Brasileiro**. [S.l.]: MINISTÉRIO DA CIÊNCIA E TECNOLOGIA/SECRETARIA DE POLÍTICA DE INFORMÁTICA (MCT/SEPIN). Brasília, 1999.

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

CALVARY, G.; COUTAZ, J.; BOUILLON, L.; FLORINS, M.; LIMBOURG, Q.; MARUCCI, L.; PATERNÒ, F.; SANTORO, C.; SOUCHON, N.; THEVENIN, D. et al. The cameleon reference framework, deliverable 1.1. **CAMELEON project**, 2002.

CARVALHO, L. O.; SERAPHIM, T. F. P.; JR., C. T.; SERAPHIM, E. Obinject: a noodmg persistence and indexing framework for object injection. **JIDM**, v. 4, n. 3, p. 220–235, 2013.

CAVALCANTI, L. **VRaptor: Desenvolvimento ágil para web com Java**. [S.l.]: Editora Casa do Código, 2014.

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

CORDEIRO, G. **CDI: Integre as dependências e contextos do seu código Java**. [S.l.]: Editora Casa do Código, 2014.

CORDY, J. R.; SHUKLA, M. Practical metaprogramming. In: IBM PRESS. **Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research-Volume 1**. [S.l.], 1992. p. 215–224.

CZARNECKI, K.; EISENECKER, U. W. Intentional programming. **Generative Programming. Methods, Tools, and Applications**, v. 2000, p. 41, 2000.

DAMAŠEVIČIUS, R.; ŠTUIKYS, V. Taxonomy of the fundamental concepts of metaprogramming. **Information Technology and Control**, v. 37, n. 2, 2015.

DAVID, D. A. **Considering non-functional requirements in model-driven engineering**. Dissertação (Mestrado) — Universidad Politécnica de Catalunya, 2009.

DEMICHIEL, L. **JSR 317: JavaTM persistence API, version 2.0**. [S.l.], 2009.

DEY, A. K. Understanding and using context. **Personal and ubiquitous computing**, Springer-Verlag, v. 5, n. 1, p. 4–7, 2001.

EVANS, E. **Domain-driven design: tackling complexity in the heart of software**. [S.l.]: Addison-Wesley Professional, 2004.

FERREIRA, S. B. L.; RODRIGUES, R. N. **e-Usabilidade**. [S.l.]: Grupo Gen-LTC, 2008.

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

FOWLER, M. **UML Essencial: um breve guia para linguagem padrão**. [S.l.]: Bookman Editora, 2014.

____. **Language Workbenches and Model Driven Architecture**. [S.l.], 2017. Disponível em: <http://martinfowler.com/articles/mdaLanguageWorkbench.html>. Acesso em: 15 de Janeiro de 2017.

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

HANSSON, D. H. **Ruby on Rails Framework**. [S.l.], 2003. Disponível em: <http://rubyonrails.org/>. Acesso em: 15 de Setembro de 2016.

JENDROCK ERIC, e. a. **The Java EE 6 Tutorial**. [S.l.], 2013. Disponível em: <`http://docs.oracle.com/javaee/6/tutorial/doc/index.html`>. Acesso em: 20 de Maio de 2017.

JSR-299 Expert Group. **JSR-299: Contexts and Dependency Injection for the Java EE platform**. [S.l.], dez. 2009.

KING, G. Jsr-299: Contexts and dependency injection for the java ee platform.[sl], 2009. **JSR-299 Expert Group**, 2009.

LARMAN, C. **Utilizando UML e padrões**. [S.l.]: Bookman Editora, 2002.

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

LUMERTZ, P. R.; RIBEIRO, L.; DUARTE, L. M. User interfaces metamodel based on graphs. **Journal of Visual Languages & Computing**, Elsevier, v. 32, p. 1–34, 2016.

MAGNO, D. G. **Aplicação da Técnica de Scaffolding para a Criação de Sistemas CRUD.** Dissertação (Mestrado) — Universidade Federal de Itajubá, 2015.

MALENFANT, J.; JACQUES, M.; DEMERS, F. N. A tutorial on behavioral reflection and its implementation. In: **Proceedings of the Reflection**. [S.l.: s.n.], 1996. v. 96, p. 1–20.

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

MEIXNER, G.; PATERNO, F.; VANDERDONCKT, J. **Past, present, and future of model-based user interface development. i-com 10 (3): 2-11, 2011**. 2011.

MELLOR, S. J.; BALCER, M.; BY-JACOBOSON, I. F. **Executable UML: A foundation for model-driven architectures**. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2002.

MELLOR, S. J.; CLARK, T.; FUTAGAMI, T. Model-driven development: guest editors' introduction. **IEEE software**, IEEE Computer Society, v. 20, n. 5, p. 14–18, 2003.

MILLER, J.; MUKERJI, J. Mda guide version 1.0. 1, object management group. **Inc., June**, 2003.

MOLINA, A. I.; GIRALDO, W. J.; GALLARDO, J.; REDONDO, M. A.; ORTEGA, M.; GARCÍA, G. Ciat-gui: A mde-compliant environment for developing graphical user interfaces of information systems. **Advances in Engineering Software**, Elsevier, v. 52, p. 10–29, 2012.

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

MYERS, B. Challenges of hci design and implementation. **interactions**, ACM, v. 1, n. 1, p. 73–83, 1994.

MYERS, B.; HUDSON, S. E.; PAUSCH, R. Past, present, and future of user interface software tools. **ACM Transactions on Computer-Human Interaction (TOCHI)**, ACM, v. 7, n. 1, p. 3–28, 2000.

MYERS, B. A. User interface software tools. **ACM Transactions on Computer-Human Interaction (TOCHI)**, ACM, v. 2, n. 1, p. 64–103, 1995.

MYERS, B. A.; ROSSON, M. B. Survey on user interface programming. In: ACM. **Proceedings of the SIGCHI conference on Human factors in computing systems**. [S.l.], 1992. p. 195–202.

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

NIELSEN, J. Improving a human-computer dialogue: What designers know about traditional interface design. **Communications of the ACM**, v. 33, 1990.

ACM. **Enhancing the explanatory power of usability heuristics**. 152–158 p.

NIELSEN, J. **Severity Ratings for Usability Problems: Article by Jakob Nielsen**. [S.l.], 1995. Disponível em: <https://www.nngroup.com/articles/how-to-rate-the-severity-of-usability-problems/>. Acesso em: 15 de Dezembro de 2016.

____. **How Many Test Users in a Usability Study?** 2012. Disponível em: <http://www.nngroup.com/articles/how-many-test-users/>. Acesso em: 15 de Dezembro de 2016.

NIELSEN, J.; MACK, R. L. Usability inspection methods john wiley & sons. **New York**, 1994.

NIELSEN, J.; MOLICH, R. Heuristic evaluation of user interfaces. In: ACM. **Proceedings of the SIGCHI conference on Human factors in computing systems**. [S.l.], 1990. p. 249–256.

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

PATERNÒ, F. Concurtasktrees: an engineered notation for task models. **The handbook of task analysis for human-computer interaction**, Lawrence Erlbaum Associates, 2004, Mahwah, New Jersey, p. 483–503, 2004.

PATERNO, F. **Model-based design and evaluation of interactive applications**. [S.l.]: Springer Science & Business Media, 2012.

PATERNO, F.; SANTORO, C.; SPANO, L. D. Maria: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. **ACM Transactions on Computer-Human Interaction (TOCHI)**, ACM, v. 16, n. 4, p. 19, 2009.

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

PUERTA, A.; EISENSTEIN, J. Towards a general computational framework for model-based interface development systems. **Knowledge-Based Systems**, Elsevier, v. 12, n. 8, p. 433–442, 1999.

PUERTA, A. R. The mecano project: Comprehensive and integrated support for model-based interface development. In: CITESEER. **CADUI**. [S.l.], 1996. v. 96, p. 19–36.

RIDEAU, F. Metaprogramming and free availability of sources. In: **Proc. of Autour du Libre Conference, Bretagne**. [S.l.: s.n.], 1999.

ROCHER, G.; BROWN, J. S. **The definitive guide to grails**. [S.l.]: Apress, 2009.

ROCHER, G.; LEDBROOK, P.; PALMER, M.; BROWN, J.; DALEY, L.; BECKWITH, B.; HOTARI, L. **Grails Scaffolding**. [S.l.], 2014. Disponível em: <`http://grails.org/doc/latest/guide/scaffolding.html`>. Acesso em: 13 de Agosto de 2016.

SANCHEZ, J. **Crudo Plugin**. [S.l.], 2008. Disponível em: <`http://crudo.sourceforge.net/`>. Acesso em: 22 de Maio de 2016.

SAURO, J. **Measuring Usability With The System Usability Scale (SUS)**. [S.l.], 2011. Disponível em: <`http://www.measuringu.com/sus.php`>. Acesso em: 05 de Dezembro de 2016.

SCHLUNGBAUM, E. **Model-based user interface software tools-current state of declarative models**. [S.l.], 1996.

SELIC, B. Personal reflections on automation, programming culture, and model-based software engineering. **Automated Software Engineering**, Springer, v. 15, n. 3, p. 379–391, 2008.

SHARAN, K. Model-view-controller pattern. In: **Learn JavaFX 8**. [S.l.]: Springer, 2015. p. 419–434.

SHEARD, T. Accomplishments and research challenges in meta-programming. In: SPRINGER. **International Workshop on Semantics, Applications, and Implementation of Program Generation**. [S.l.], 2001. p. 2–44.

SHNEIDERMAN, B.; PLAISANT, C. Designing the user interface: Strategies for effective human-computer interaction. **ACM SIGBIO Newsletter**, ACM, v. 9, n. 1, p. 6, 1987.

SIEGEL, J. M. Model driven architecture (mda)–mda guide rev. 2.0. **Object Management Group, Tech. Rep. ORMSC/14-06-0**, 2014.

SILVA, A. R. da. Model-driven engineering: A survey supported by the unified conceptual model. **Computer Languages, Systems & Structures**, Elsevier, v. 43, p. 139–155, 2015.

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

SILVA, P. P. D. User interface declarative models and development environments: A survey. In: SPRINGER. **International Workshop on Design, Specification, and Verification of Interactive Systems**. [S.l.], 2000. p. 207–226.

SILVA, P. P. D.; PATON, N. W. User interface modeling in umli. **IEEE software**, IEEE, v. 20, n. 4, p. 62–69, 2003.

SILVA, W. C. d. **Gerência de Interfaces para Sistemas de Informação: uma abordagem baseada em modelos**. Dissertação (Mestrado) — Universidade Federal de Goiás, 2010.

SILVEIRA, P. **Introdução à arquitetura e design de software: uma visão sobre a plataforma Java**. [S.l.]: Editora Campus, 2012.

ŠTUIKYS, V.; DAMAŠEVIČIUS, R. **Meta-programming and model-driven meta-program development: principles, processes and techniques**. [S.l.]: Springer Science & Business Media, 2012.

SZEKELY, P. Retrospective and challenges for model-based interface development. In: **Design, Specification and Verification of Interactive Systems' 96**. [S.l.]: Springer, 1996. p. 1–27.

THIMBLEBY, H. **User Interface Design**. New York, NY, USA: ACM, 1990. ISBN 0-201-41618-2.

TYPESAFE. **Play Framework Documentation**. [S.l.], 2014. Disponível em: <https://www.playframework.com/>. Acesso em: 22 de Maio de 2017.

VANDERDONCKT, J. Automatic generation of a user interface for highly interactive business-oriented applications. In: ACM. **Conference Companion on Human Factors in Computing Systems**. [S.l.], 1994. p. 123–124.

VANDERDONCKT, J. M.; BODART, F. Encapsulating knowledge for intelligent automatic interaction objects selection. In: ACM. **Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems**. [S.l.], 1993. p. 424–429.

VÖLTER, M.; STAHL, T.; BETTIN, J.; HAASE, A.; HELSEN, S. **Model-driven software development: technology, engineering, management**. [S.l.]: John Wiley & Sons, 2013.

VRAPTOR, F. **VRaptor**. [S.l.], 2017. Disponível em: <http://www.vraptor.org/pt/>. Acesso em: 12 de fevereiro de 2017.

WALKER, J. Through the looking glass. In: LAUREL, B.; MOUNTFORD, S. J. (Ed.). **The art of human-computer interface design**. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1990.

WARMER, J. B.; KLEPPE, A. G. **The object constraint language: getting your models ready for MDA**. [S.l.]: Addison-Wesley Professional, 2003.

WEISSAMANN, H. L. **Groovy e Grails sem Pivotal: e daí?** [S.l.], 2015. Disponível em: <http://www.itexto.net/devkico/?p=2116>. Acesso em: 12 de fevereiro de 2017.

WILCOXON, F. Individual comparisons by ranking methods. **Biometrics bulletin**, JSTOR, v. 1, n. 6, p. 80–83, 1945.

Apêndice A

Roteiro Experimental

Introdução

Este roteiro deve ser seguido para a obtenção de dados qualitativos e quantitativos que indiquem o nível de usabilidade e de produtividade do *framework ObCrud*. O Roteiro está dividido em 2 experimentos a serem realizados com os *frameworks ObCrud* e *Grails*. Cada experimento é composto por três fases distintas: de treinamento; de execução e de avaliação.

Na fase de treinamento você deverá observar e realizar juntamente com o avaliador as tarefas deste roteiro, basicamente você deverá seguir o passo a passo deste roteiro, nesta fase o avaliador poderá ser consultado para sanar qualquer dúvida ou problema técnico, já na fase de execução você tentará executar tarefas parecidas, mas sem qualquer auxílio do avaliador.

Ao final das fases de treinamento e execução dos dois experimentos a fase de avaliação deverá ser realizada. Nesta fase, o participante irá avaliar a usabilidade das UI CRUDs geradas e dos *frameworks* utilizados nos experimentos.

As duas primeiras fases dos experimentos são compostas por duas atividades. A primeira atividade é a geração da UI CRUD e a segunda atividade é a avaliação das funcionalidades CRUD das UIs geradas. A fase de avaliação é composta por três atividades. A primeira atividade é a avaliação de usabilidade das UIs geradas utilizando as heurísticas de Nilsen. A segunda atividade é a comparação das UIs geradas pelos 2 *frameworks*. A última atividade é a avaliação de usabilidade, dos *frameworks* utilizados nos experimentos, por meio do questionário *System Usability Scale* (SUS). O resumo deste roteiro é descrito a seguir:

```
Experimento 1 - Framework Grails
      i - Fase de Treinamento
            a) Geração das UIs CRUD.
            b) Verificação das funcionalidades CRUD.
     ii - Fase de Execução
            a) Geração das UIs CRUD - T1.
```

```
            b) Verificação das funcionalidades CRUD - T2.
    iii - Fase de Avaliação
            a) Avaliação heurística das UIs .
            b) Comparação de usabilidade das UIs.
            c) Aplicação do questionário de usabilidade (SUS).

Experimento 2 - Framework ObCrud
      i - Fase de Treinamento
            a) Geração das UIs CRUD.
            b) Verificação das Funcionalidades CRUD.
     ii - Fase de Execução
            a) Geração das UIs CRUD - T1.
            b) Verificação das funcionalidades CRUD - T2.
    iii - Fase de Avaliação
            a) Avaliação heurística das UIs.
            b) Comparação de usabilidade das UIs.
            c) Aplicação do questionário de usabilidade (SUS).
```

Experimento 1 - framework Grails

Nesta parte será utilizado o *framework Grails* para a realização dos procedimentos. O avaliador demonstrará o passo a passo de como gerar as UIs CRUD a partir de classes de modelo. O funcionamento do sistema gerado também será demonstrado pelo avaliador. Os participantes deverão observar os passos, do avaliador, atentamente, ao mesmo tempo que também os realiza.

Fase de Treinamento [Avaliador e participante]

- 1ª Tarefa - Geração das UIs CRUD

  1. Abra o eclipse IDE.
  2. Acesse o menu: FILE -> New -> Other -> Grails Project.
  3. Digite o nome do Projeto: **exp01_treinamento**.
  4. Clique com botão direito do mouse no projeto e selecione: Propriedades -> Groovy Compiler, após altere o Groovy compiler level for **exp01_treinamento** para 2.4.
  5. Crie a classe Médico em **domain/exp01_treinamento/Medico.groovy**.

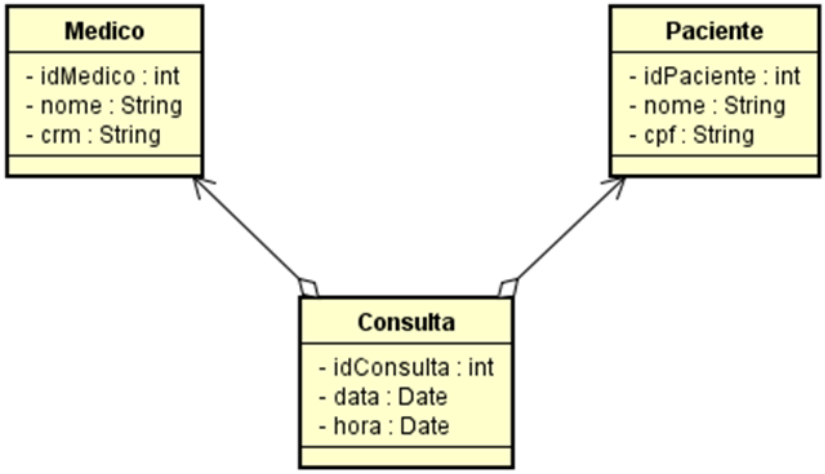

Figura A.1: Experimento 1 - Treinamento - Diagrama de classes 1

```
1  package exp01_treinamento
2
3  class Medico{
4
5      String nome
6      String crm
7      String telefone
8
9      static constraints = {
10         telefone nullable: true
11         crm nullable: true
12         nome size: 10..20
13     }
14
15     String toString(){
16         return nome;
17     }
18 }
```

6. Crie a classe Paciente em **domain/exp01_treinamento/Paciente.groovy**.

```
1  package exp01_treinamento
2
3  class Paciente{
4
5      String nome
6      String cpf
7      String telefone
8
9      static constraints = {
10         telefone nullable: true
11         cpf nullable: true
12     }
13
14     String toString(){
15         return nome
16     }
17 }
```

7. Crie a classe Consulta em **domain/exp01_treinamento/Consulta.groovy**.

```
1  package exp01_treinamento
2
3  class Consulta{
4
5      Date data
6      Date hora
7
8      static belongsTo = {
9          medico Medico,
10         paciente Paciente
11     }
12 }
```

8. Crie a classe MedicoController em **controllers/exp01_treinamento/Medico-Controller.groovy**

```
1  package exp01_treinamento
2
3  class MedicoController{
4      static scaffold = Medico
5  }
```

9. Crie a classe PacienteController em **controllers/exp01_treinamento/Paciente-Controller.groovy**

```
1 package exp01_treinamento
2
3 class PacienteController{
4     static scaffold = Paciente
5 }
```

10. Crie a classe ConsultaController em **controllers/exp01_treinamento/ConsultaController.groovy**

```
1 package exp01_treinamento
2
3 class ConsultaController{
4     static scaffold = Consulta
5 }
```

11. Clique em Run -> Run As -> Grails Command (Run App)
12. Abra o navegador Google Chrome e digite na barra de endereços: `http://localhost:8080/exp01_treinamento`

- 2ª Tarefa - Verificação das funcionalidades CRUD das UIs geradas

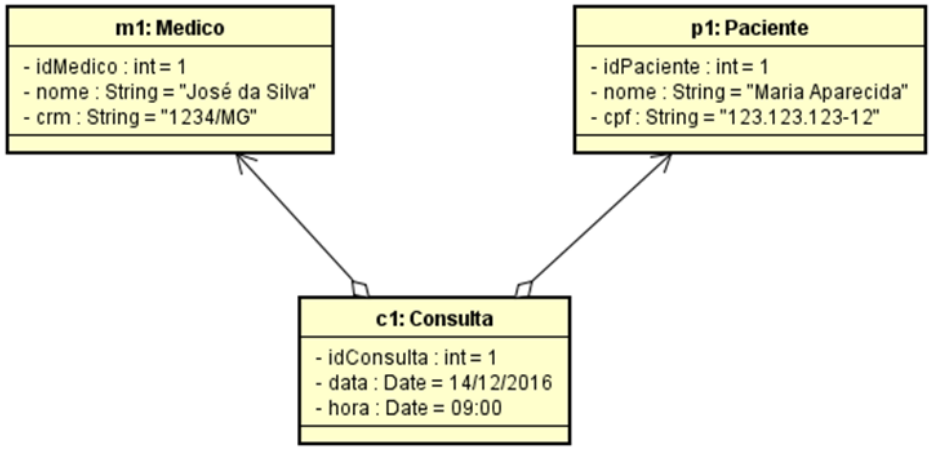

Figura A.2: Experimento 1 - Treinamento - Diagrama de objetos 1

13. Crie a instância **m1** e **p1**. Esta operação testa a criação de objetos (Create).
14. Crie a instância **c1** associando a ela o Médico **m1** e o Paciente **p1**.
15. Visualize o objeto **c1**. Esta operação testa a leitura de objetos (Read). Repare que os objetos **m1** e **p1** estão contidos em **c1**. Isso garante o funcionamento correto do relacionamento entre as classes Consulta, Médico e Paciente.
16. Altere a data da consulta para **15/12/2016**. Esta operação testa a alteração de objetos (Update).
17. Exclua o objeto **c1**. Esta operação testa a exclusão de objetos (Delete).

Fase de Execução [Estudante]

- 1ª Tarefa - Geração das UIs CRUD

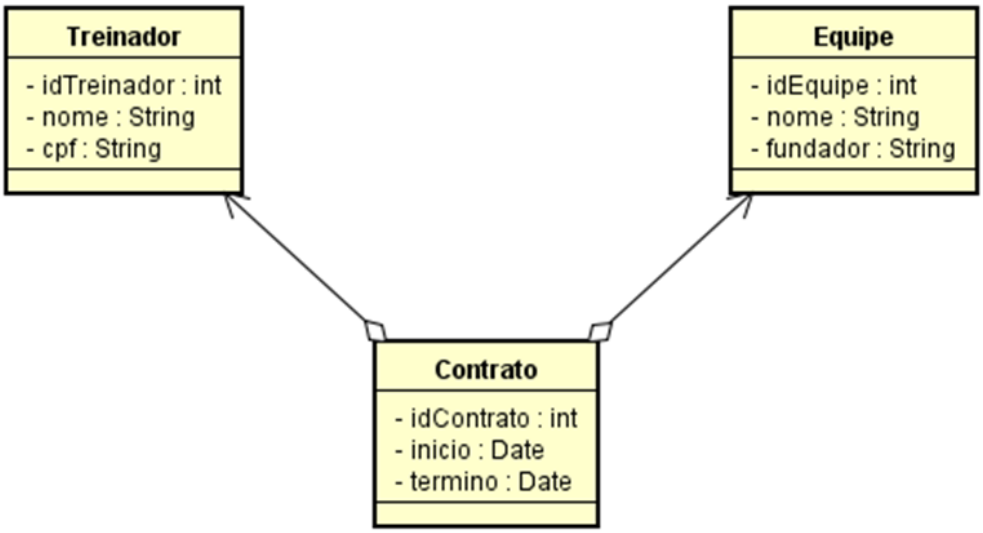

Figura A.3: Diagrama de classes da fase de execução

1. Inicie o cronometro (http://cronometronline.com.br/).
2. Repita os passos 1 a 8 executados na fase de treinamento, mas desta vez utilize o diagrama de classes da Figura 4.4. Siga os passos com calma e atenção. O avaliador não poderá ser consultado nesta fase do experimento.
3. Ao finalizar o desenvolvimento pare o cronometro.
4. Anote o tempo utilizado no campo T1 do formulário duração de atividades.

- 2ª Tarefa - Verificação das funcionalidades CRUD das UIS geradas

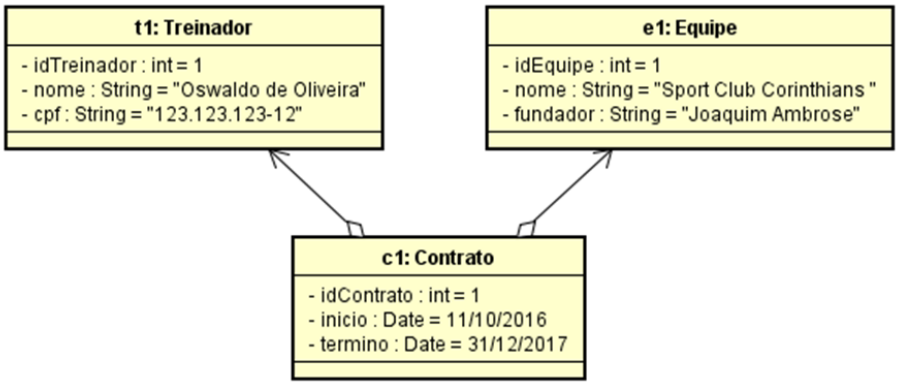

Figura A.4: Diagrama de objetos da fase de execução

1. Inicie o cronometro (http://cronometronline.com.br/)

2. Crie a instância **t1** e **e1**. Esta operação testa a criação de objetos (Create).
3. Crie a instância **C1** associando a ela o Treinador **t1** e Equipe **e1**.
4. Visualize o objeto **c1**. Esta operação testa a leitura de objetos (Read). Repare que os objetos **t1** e **e1** estão contidos em **c1**. Isso garante o funcionamento correto do relacionamento entre as classes Contrato, Equipe e Treinador.
5. Altere a data de termino do contrato para 01/12/2017. Esta operação testa a alteração de objetos (Update).
6. Exclua o objeto **c1**. Esta operação testa a exclusão de objetos (Delete).
7. Ao finalizar o desenvolvimento pare o cronometro
8. Anote o tempo utilizado no campo **T2** do formulário duração de atividades.

Experimento 2 - framework ObCrud

Nesta parte será utilizado o *framework ObCrud* para a realização dos procedimentos. O avaliador demonstrará os passos de como gerar as UIs CRUD a partir de classes do modelo domínio da Figura A.5. O funcionamento do sistema gerado também será demonstrado pelo avaliador. Os participantes deverão observar os passos atentamente, ao mesmo tempo que também os realizam. Ao final da demonstração os participantes deverão realizar o mesmo procedimento cronometrando o tempo. Todos os tempos obtidos serão salvos no formulário duração de atividades.

Fase de Treinamento [Avaliador e participante]

- 1ª Tarefa - Geração das UIs CRUD

1. Abra o eclipse IDE
2. Acesse o menu: FILE -> New -> Other -> Maven Project. Clique em Next e novamente em Next
3. Em Catalog: Default Local, selecione o archetype-obcrud, clique em Next
4. Em Artefact Id, digite o nome do projeto: **exp02_treinamento**
5. Crie a classe Médico em **model/Medico.java**.

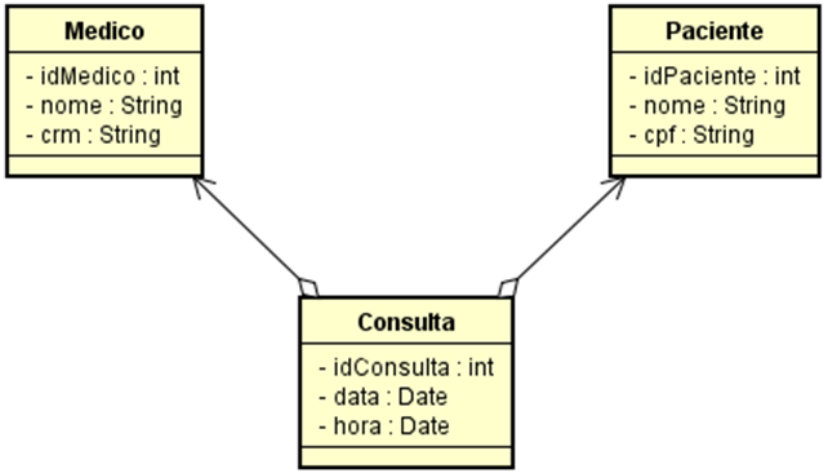

Figura A.5: Diagrama de classes da fase de treinamento

```
1  @Entity
2  public class Medico{
3
4      @Id
5      @GeneratedValue
6      private Integer idMedico;
7      @NotNull
8      @Size(min=10, max=20)
9      private String nome;
10     private String crm;
11     private String telefone;
12
13     public String toString(){
14         return nome;
15     }
16     //gerar getter e setter
17 }
```

6. Crie a classe Paciente em **model/Paciente.java**.

```
@Entity
public class Paciente{

    @Id
    @GeneratedValue
    private Integer idPaciente;
    @NotNull
    private String nome;
    private String cpf;
    private String telefone;

    public String toString(){
        return nome;
    }
    //gerar getter e setter
}
```

7. Crie a classe Consulta em **model/Consulta.java**.

```
@Entity
public class Consulta{

    @Id
    @GeneratedValue
    private Integer idConsulta;
    @Temporal(TemporalType.DATE)
    private String data;
    @Temporal(TemporalType.TIME)
    private String hora;
    @ManyToOne
    private Medico medico;
    @ManyToOne
    private Paciente paciente;

    //gerar getter e setter
}
```

8. Crie a classe MedicoController em **controller/MedicoController.java**

```
@controller
public class MedicoController{

    @Crud(forClass = Medico.class)
    public void index(){
    }
}
```

9. Crie a classe PacienteController em **controller/PacienteController.java**

```
@controller
public class PacienteController{

    @Crud(forClass = Paciente.class)
    public void index(){
    }
}
```

10. Crie a classe ConsultaController em **controller/ConsultaController.java**

```
@controller
public class ConsultaController{

    @Crud(forClass = Consulta.class)
    public void index(){
    }
}
```

11. Edite a unidade de persistência localizada no diretório: src/main/resources/META-INF/persistence.xml e adicione as classes de modelo, conforme Figura A.6

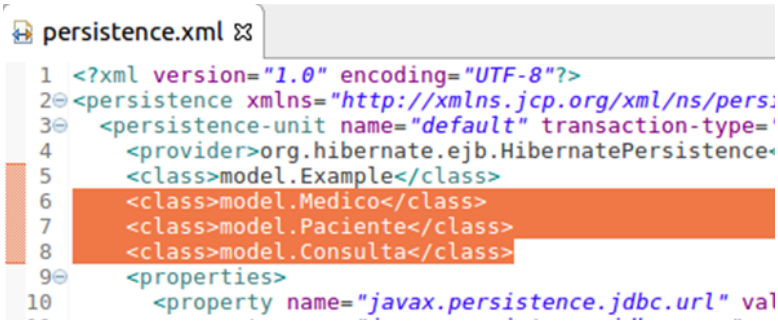

Figura A.6: Diagrama de classes da fase de treinamento

12. Clique em Run -> Wildfly 8.x -> Next -> adicione o exp02_treinamento em `configured` e clique em `Finish`
13. Abra o navegador Google Chrome e digite na barra de endereços: `http://localhost:8080/exp02_treinamento/medico`

- 2ª Tarefa - Verificação das funcionalidades CRUD das UIs geradas

14. Crie a instância **m1** e **p1**. Esta operação testa a criação de objetos (Create).
15. Crie a instância **c1** associando a ela o Médico **m1** e o Paciente **p1**.
16. Visualize o objeto **c1**. Esta operação testa a leitura de objetos (Read). Repare que os objetos **m1** e **p1** estão contidos em **c1**. Isso garante o funcionamento correto do relacionamento entre as classes Consulta, Médico e Paciente.

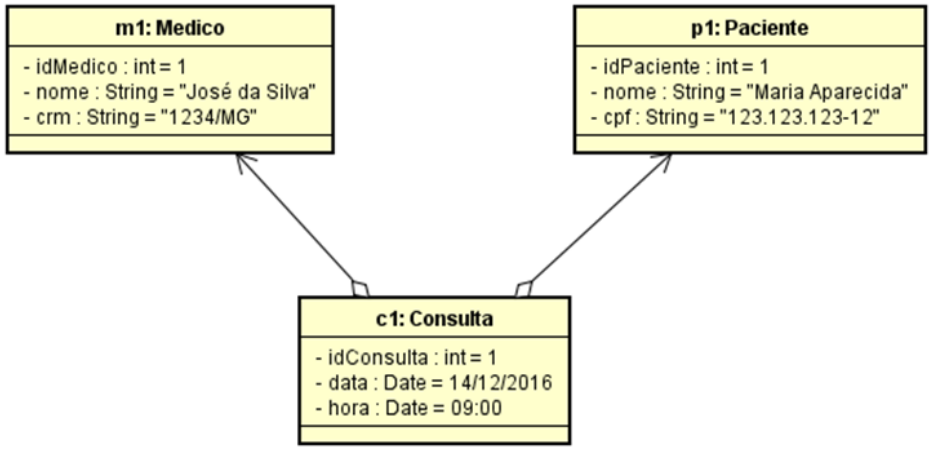

Figura A.7: Diagrama de objetos da fase de treinamento

17. Altere a data da consulta para **15/12/2016**. Esta operação testa a alteração de objetos (Update).
18. Exclua o objeto **c1**. Esta operação testa a exclusão de objetos (Delete).

Fase de Execução [Estudante]

- 1ª Tarefa - Geração das UIs CRUD

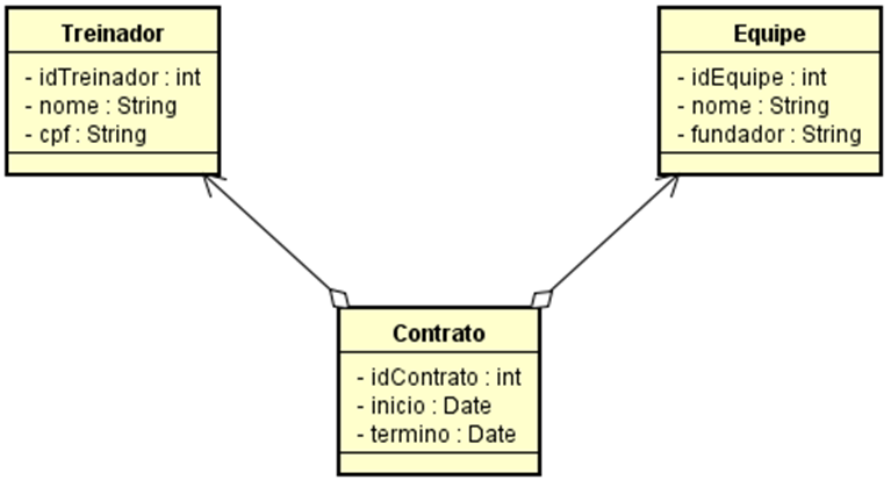

Figura A.8: Diagrama de classes da fase de execução

1. Inicie o cronometro (http://cronometronline.com.br/).
2. Repita os passos 1 a 9 executados na fase de treinamento, mas desta vez utilize o diagrama de classes da Figura A.8. Siga os passos com calma e atenção. O avaliador não poderá ser consultado nesta fase do experimento.
3. Ao finalizar o desenvolvimento pare o cronometro.

4. Anote o tempo utilizado no campo T2 do formulário duração de atividades.

- 2ª Tarefa - Verificação das funcionalidades CRUD das UIS geradas

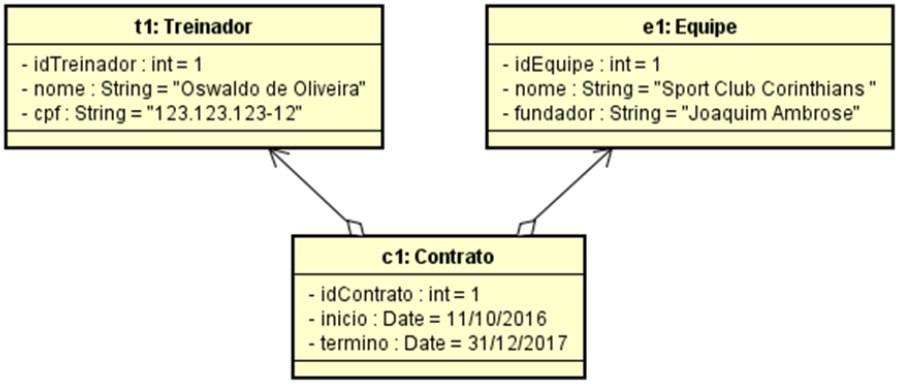

Figura A.9: Diagrama de objetos da fase de execução

1. Inicie o cronometro (http://cronometronline.com.br/)
2. Crie a instância **t1** e **e1**. Esta operação testa a criação de objetos (Create).
3. Crie a instância **C1** associando a ela o Treinador **t1** e Equipe **e1**.
4. Visualize o objeto **c1**. Esta operação testa a leitura de objetos (Read). Repare que os objetos **t1** e **e1** estão contidos em **c1**. Isso garante o funcionamento correto do relacionamento entre as classes Contrato, Equipe e Treinador.
5. Altere a data de termino do contrato para 01/12/2017. Esta operação testa a alteração de objetos (Update).
6. Exclua o objeto **c1**. Esta operação testa a exclusão de objetos (Delete).
7. Ao finalizar o desenvolvimento pare o cronometro
8. Anote o tempo utilizado no campo **T2** do formulário duração de atividades.

Fase de Avaliação dos Experimentos 1 e 2

Após criar as UIs CRUDs utilizando os dois *frameworks ObCrud* e *Grails* faça a avaliação de usabilidade deles e das UIs CRUDs geradas.

Fase de Treinamento [Avaliador e participante]

- Avaliação heurística das UIs .

Fase de Treinamento [Avaliador e participante]

- Comparação de usabilidade das UIs.

Fase de Treinamento [Avaliador e participante]

- Aplicação do questionário de usabilidade (SUS).

Apêndice B

Dados Experimentais

Este Apêndice apresenta todos os dados obtidos nos experimentos realizados neste trabalho. A Tabela B.1 exibe o tempo gasto por cada um dos 8 participantes na execução dos experimentos referentes aos *frameworks ObCrud* e *Grails*. Foram calculados a média e a mediana de cada tempo. As tabelas Tabela B.3 e Tabela B.4 apresentam as respostas do questionário e as pontuações SUS relativas aos *frameworks ObCrud* e *Grails* respectivamente. Foram calculados a média e o desvio padrão das pontuações obtidas.

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

	Experimento 1: Grails		Experimento 2: ObCrud	
Participante	T1	T2	T1	T2
1	954	95	985	63
2	603	224	1309	102
3	702	77	694	69
4	1274	268	860	553
5	701	223	791	163
6	674	150	686	119
7	270	122	255	110
8	717	69	548	65
Média	737	154	766	156
Mediana	701,5	136	742,5	106

Tabela B.2: Respostas dos participantes (escala -5 a +5) comparação de usabilidade das UIs

	Respostas dos participantes							
Questões	1	2	3	4	5	6	7	8
1	4	-2	5	5	3	3	2	2
2	3	3	5	5	5	0	3	1
3	4	2	5	5	2	2	3	2

Tabela B.3: Respostas dos participantes (escala de 1 a 5) e a pontuação SUS do sistema *ObCrud*

	Experimento 1 - ObCrud										
	Respostas (Questões 1 a 10)										
Participante	1	2	3	4	5	6	7	8	9	10	Pontuação SUS
1	4	1	4	1	4	1	4	1	4	4	80
2	4	2	3	2	5	2	4	1	4	3	75
3	5	1	5	3	5	1	4	1	4	1	90
4	3	4	4	2	4	1	4	1	4	2	72,5
5	5	1	5	1	5	1	4	1	5	1	97,5
6	5	1	5	1	5	1	5	1	5	1	100
7	4	2	4	2	4	1	4	1	5	1	85
8	4	1	5	2	4	1	4	1	5	1	90
										Média	86,3
										Desvio Padrão	10

Tabela B.4: Respostas dos participantes (escala de 1 a 5) e a pontuação SUS do *framework Grails*

	Experimento 2 - Grails										
	Respostas (Questões 1 a 10)										
Participante	1	2	3	4	5	6	7	8	9	10	Pontuação SUS
1	4	1	5	1	3	1	5	1	4	4	82,5
2	4	1	4	2	4	2	3	1	4	3	75
3	4	3	3	3	5	1	4	1	3	4	67,5
4	3	3	5	2	3	3	5	1	3	2	70
5	5	2	4	1	5	1	5	1	5	4	87,5
6	4	1	4	1	5	1	5	1	5	1	95
7	3	2	4	2	3	1	4	1	5	2	77,5
8	3	2	4	2	3	1	4	2	3	2	70
										Média	78,1
										Desvio Padrão	9,6

Apêndice C

Formulário de Registro de Tempo

Este formulário é utilizado para armazenar os diversos tempos produzidos pelos participantes durante a execução do primeiro experimento.

Identificação

Experimento 01: Geração das UIs CRUD utilizando o Framework Grails

Procedimento	Tempo produzido
T1 - Geração das UIs CRUD	
T2 - Verificação das funcionalidades CRUD das UIs geradas	

Experimento 02: Geração das UIs CRUD utilizando o Framework ObCrud

Procedimento	Tempo produzido
T1 - Geração das UIs CRUD	
T2 - Verificação das funcionalidades CRUD das UIs geradas	

Apêndice D

Comparação das UIs CRUD

Avalie as UIs CRUDs produzidas pelos *frameworks Obcrud* e *Grails* assinalando uma alternativa na escala de -5 a 5+ para cada uma das questões abaixo.

Identificação

A interface produzida pelo framework ObCrud, em termos de usabilidade, é melhor ou pior, em relação a interface produzida pelo Framework Grails?

Pior	-5	-4	-3	-2	-1	0	1	2	3	4	5	Melhor

A interface produzida pelo framework ObCrud é melhor ou pior esteticamente, em relação a interface produzida pelo Framework Grails?

Pior	-5	-4	-3	-2	-1	0	1	2	3	4	5	Melhor

A interface produzida pelo framework ObCrud, de uma forma geral, é melhor ou pior, em relação a produzida pelo Framework Grails?

Pior	-5	-4	-3	-2	-1	0	1	2	3	4	5	Melhor

Apêndice E

Avaliação heurística de usabilidade das UIs CRUD

Avalie a usabilidade das UIs CRUD produzidas pelos *frameworks ObCrud* e *Grails* utilizando as heurísticas de Nielsen. Siga as orientações para realizar a avaliação.

- Na coluna Descrição dos problemas descreva os problemas encontrados nas interfaces gráficas geradas pelos *frameworks ObCrud* e *Grails*.
- Na coluna *framework* indique em qual *framework* o problema ocorreu
- Na coluna Heurística relacione o problema encontrado com 1 das 10 heurísticas de Nielsen.
- Na coluna grau de severidade classifique o problema encontrado quanto a seu grau de severidade.

Descrição dos problemas	Framework	Heurística	Grau de Severidade

Observação: A relação das heurísticas e do grau de severidade, se encontram no verso da ficha de avaliação, no entanto, faça a classificação, após identificar todos os possíveis problemas.

Tabela E.1: Heurísticas de Nielsen

Num	Heurística	Descrição
1	Visibilidade do status do sistema	O sistema deve manter o usuário informado sobre o que está sendo acontecendo, as informações devem ser apresentadas, ao usuário, por meio de feedback adequados, ou seja, com informações relevantes, clara, e apresentadas com antecedência, de forma que, os usuários, possam tomar as devidas providencias.
2	Correspondência entre sistema e o mundo real	O sistema deve se comunicar com o usuário utilizando termos que sejam familiares ao usuário, ao invés de utilizar termos técnicos e códigos de erros voltados para o sistema.
3	Controle e liberdade do usuário	É comum os usuários acessarem funcionalidades do sistema por engano, desta forma, o sistema deve disponibilizar recursos para que o usuário cancele a operação a qualquer momento e que ele ainda possa desfazer ou refazer uma ação.
4	Consistência e padrões	O sistema deve manter a consistência e o padrão das interfaces e dos diálogos, assim as funcionalidades devem ser identificadas sempre da mesma maneira, de forma a facilitar a aprendizagem e a compreensão do sistema pelo usuário.
5	Prevenção de erros	Mensagens de erros são importantes, pois permite que o usuário possa solucionar problemas, no entanto, melhor ainda é evitar que erros ocorram. Assim o sistema deve apresentar mensagens de confirmação, antes que o usuário realize ações definitivas de forma a prevenir os erros.
6	Reconhecimento ao invés de lembrança	O sistema deve deixar a mostra as principais funcionalidades que o usuário pode executar em cada momento. Dessa forma, o usuário é poupado de memorizar onde se encontra as ações e como elas devem ser executadas para alcançar o resultado esperado.
7	Flexibilidade e eficiência de uso	O sistema deve ser projetado para atender usuários leigos e usuários experientes. Usuários leigos necessitam de mais informações e orientações para executar tarefas, ao passo que usuários experientes não. O sistema deve possuir atalhos para que usuários experientes realizem atividades com maior produtividade, ao mesmo, tempo o sistema deve ser encadeado para que usuários leigos possam utiliza-lo com facilidade.
8	Estética e design minimalista	A visibilidade do status do sistema é importante, mas ela não deve ser confundida com a apresentação de informações irrelevantes ou raramente necessárias. O excesso de diálogo e de informações desnecessárias atrapalham o uso do sistema, além de retirar a atenção do usuário das informações realmente importantes.
9	Ajuda para o usuário identificar, diagnosticar e corrigir erros	Assim como sistema deve ter correspondência com o mundo real, as mensagens de erro devem ser expressas de forma simples e com sugestões de solução, para que o usuário possa identificar, diagnosticar e corrigir os erros.
10	Ajudas (Helps) e documentação	É importante que o sistema seja bem projetado de forma que possa ser utilizado sem a necessidade de documentação, no entanto, caso seja necessário ela deve estar disponível. A documentação deve ser clara, precisa e focada na atividade que o usuário pretende realizar.

Tabela E.2: Grau de severidade

Num	Descrição
0	Eu não concordo que isso seja um problema de usabilidade.
1	Problema cosmético: deve ser corrigido apenas em caso de haver tempo extra no projeto para tal;
2	Problema de usabilidade menor: a correção desse problema é uma tarefa de baixa prioridade
3	Problema de Usabilidade maior: é importante que esse problema seja corrigido e tratado como grande prioridade;
4	Catástrofe de usabilidade: é obrigatório corrigir o problema antes que o produto seja liberado.

Apêndice F

Questionário SUS

1. Acho que gostaria de usar este sistema com frequência.

Grails						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

ObCrud						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

2. Achei o sistema desnecessariamente complexo.

Grails						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

ObCrud						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

3. Achei que o sistema era fácil de usar.

Grails						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

ObCrud						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

4. Eu acho que iria precisar do apoio técnico para ser capaz de usar este sistema.

Grails						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

ObCrud						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

5. Eu achei que as várias funções deste sistema foram bem integradas.

Grails						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

ObCrud						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

6. Eu achei que havia muita inconsistência neste sistema.

Grails						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

ObCrud						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

7.Acho que a maioria das pessoas iriam aprender facilmente utilizar este sistema.

Grails						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

ObCrud						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

8. Eu achei o sitema muito complicado de usar.

Grails						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

ObCrud						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

9. Eu me senti muito confiante usando o sistema.

Grails						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

ObCrud						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

10. Eu precisarei aprender várias coisas antes de continuar a utilizar o sistema.

Grails						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

ObCrud						
Discordo totalmente	1	2	3	4	5	Concordo totalmente

Apêndice G

Questionário do perfil dos participantes

Identificação

Qual nível você está cursando?
( ) Graduação
( ) Pós-graduação

Você trabalha ou já trabalhou na área de tecnologia da informação?
( ) Sim
( ) Não

Qual é o seu conhecimento sobre o Sistema Operacional Ubuntu?
( ) Nenhum
( ) Básico
( ) Intermediário
( ) Avançado

Qual é o seu conhecimento sobre a IDE Eclipse?
( ) Nenhum
( ) Básico
( ) Intermediário
( ) Avançado

Qual é o seu conhecimento sobre Orientação a Objetos?
( ) Nenhum
( ) Básico
( ) Intermediário
( ) Avançado

Qual é o seu conhecimento sobre a linguagem de programação Java?
( ) Nenhum
( ) Básico
( ) Intermediário
( ) Avançado

Qual é o seu conhecimento sobre a linguagem de programação Groovy?
( ) Nenhum
( ) Básico
( ) Intermediário
( ) Avançado

Qual é o seu conhecimento sobre o framework Grails?
( ) Nenhum
( ) Básico
( ) Intermediário
( ) Avançado

Qual é o seu conhecimento sobre o framework VRaptor?
( ) Nenhum
( ) Básico
( ) Intermediário
( ) Avançado

Anotações