

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL  
INSTITUTO DE INFORMÁTICA  
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MARCELO MRACK

**Geração Automática e Assistida de  
Interfaces de Usuário**

Dissertação apresentada como requisito parcial  
para a obtenção do grau de Mestre em Ciência  
da Computação

Prof. Dr. Álvaro Freitas Moreira  
Orientador

Prof. Dr. Marcelo Soares Pimenta  
Co-orientador

Porto Alegre, março de 2009.

## CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Mrack, Marcelo

Geração Automática e Assistida de Interfaces de Usuário / Marcelo Mrack – Porto Alegre: Programa de Pós-Graduação em Computação, 2008.

78 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2008. Orientador: Álvaro Freitas Moreira; Co-orientador: Marcelo Soares Pimenta.

1.IHC. 2.IU. 3.MBUIDE. 4.MERLIN. 5.Geração Baseada em Modelos. 6.Interface de Usuário. 7.Java. I. Moreira, Álvaro Freitas. II. Pimenta, Marcelo Soares. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. José Carlos Ferraz Hennemann

Vice-reitor: Prof. Pedro Cezar Dutra Fonseca

Pró-Reitora Adjunta de Pós-Graduação: Profa. Valquiria Linck Bassani

Diretor do Instituto de Informática: Prof. Flávio Rech Wagner

Coordenador do PPGC: Prof<sup>a</sup> Luciana Porcher Nedel

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

## AGRADECIMENTOS

À Deus, por permitir.

Aos meus amados, mas nem sempre abraçados pais, Sérgio e Claudete.

À 3layer, por saber que ela vai renderizar todos os projetos que colocarmos em seu *classloader*. E dela, pelo início, ao Diego, Júlio e Hilton.

Ao Rodrigo, por ter dado aquela “mão”, ao enviar os trabalhos de aula enquanto eu estava no hospital.

À Medicina, e em especial, ao médico Marcelo Capra do Hospital Conceição de Porto Alegre; ao Hospital das Clínicas de Curitiba, com atenção redobrada à Dra. Vaneuza e equipe; ao excelente grupo de TMO do Hospital Amaral Carvalho em Jaú – SP, e com carinho para a “Bel”, Renata, Paula e aos macanudos e excelentes doutores (adjetivo merecido) Marco, Mair, Vergílio e trupe.

Também, e com mérito, à Novartis, pelo medicamento Glivec e aos pesquisadores que o desenvolveram. Sem ele, eu nunca teria tido *tempo* para este trabalho.

Em parágrafo à parte, e com gratidão infinita, agradeço à Fernanda pelo que foi, pelo que é, e pelo que sempre será. E ainda hoje sem palavras, pelo que fez.

E ao final, em algum lugar e para toda a minha existência, agradeço a minha vida ao meu doador de medula óssea, pessoa que possibilitou não só o meu renascimento, mas também a vinda de meu primogênito, Joaquim.

## **DEDICATÓRIA**

Ao meu incansável pai Sérgio, e não só esses três anos de pesquisa, mas sim todas as gratificantes noites em claro já trabalhadas no Merlin.

# SUMÁRIO

<b>LISTA DE ABREVIATURAS E SIGLAS</b> .....	<b>7</b>
<b>LISTA DE FIGURAS</b> .....	<b>9</b>
<b>LISTA DE TABELAS</b> .....	<b>10</b>
<b>RESUMO</b> .....	<b>11</b>
<b>ABSTRACT</b> .....	<b>12</b>
<b>1 INTRODUÇÃO</b> .....	<b>13</b>
<b>1.1 Contribuição</b> .....	<b>16</b>
<b>1.2 Organização do texto</b> .....	<b>16</b>
<b>2 GERAÇÃO DE IU BASEADA EM MODELOS</b> .....	<b>18</b>
<b>2.1 Histórico</b> .....	<b>18</b>
<b>2.2 As MBUIDE</b> .....	<b>18</b>
2.2.1 Os Tipos de Modelos.....	19
2.2.2 Tipos de Ferramenta.....	25
2.2.3 Arquitetura para Geração da IU.....	26
2.2.4 Processo de Desenvolvimento.....	27
<b>2.3 Projetos e Características</b> .....	<b>29</b>
<b>2.4 Discussão</b> .....	<b>29</b>
<b>3 O PROJETO MERLIN</b> .....	<b>30</b>
<b>3.1 METAGEN, o início</b> .....	<b>30</b>
<b>3.2 MERLIN, a evolução</b> .....	<b>31</b>
<b>3.3 Características da solução</b> .....	<b>32</b>
3.3.1 Interfaces CRUD em Evidência.....	32
3.3.2 Modelos Utilizados.....	33
3.3.3 Reuso de Padrões de Mercado.....	35
3.3.4 Geração em Tempo de Execução.....	36
3.3.5 Modelos Auto-Contidos.....	37
3.3.6 Interligação dos Elementos.....	37
3.3.7 Edição Textual Assistida dos Modelos.....	39
3.3.8 Tipos de Interface CRUD Suportadas.....	41
3.3.9 Layout da IU.....	42
3.3.10 Realimentação.....	44
3.3.11 Outras Funcionalidades.....	46
<b>3.4 Arquitetura</b> .....	<b>48</b>
<b>3.5 Processo de Desenvolvimento</b> .....	<b>49</b>

<b>4 ESTUDO DE CASO.....</b>	<b>51</b>
4.1 O Cenário.....	51
4.2 Processo de Desenvolvimento .....	51
4.3 Primeira Interação de Desenvolvimento.....	52
4.4 Análise do Código-Fonte .....	53
4.5 Resultado da Geração da IU na Primeira Interação de Desenvolvimento.....	54
4.6 Segunda Interação de Desenvolvimento .....	55
4.7 Resultado da Geração da IU na Segunda Interação de Desenvolvimento.....	57
4.8 Considerações sobre o Estudo de Caso .....	60
<b>5 CONCLUSÃO .....</b>	<b>62</b>
<b>REFERÊNCIAS.....</b>	<b>65</b>
<b>APÊNDICE A – TABELA COMPARATIVA DAS MBUIDE ANALISADAS .....</b>	<b>70</b>
<b>APÊNDICE B – ANOTAÇÕES DE CONFIGURAÇÃO DOS MODELOS DEFINIDAS PELA FERRAMENTA MERLIN.....</b>	<b>75</b>

## LISTA DE ABREVIATURAS E SIGLAS

AIO	Abstract Interface Object
BD	Banco de Dados
CIO	Concret Interface Object
CPE	Configuração por Exceção
CRUD	Create, Retrieve, Update and Delete
CSS	Cascading Style Sheets
DBC	Design by Contract
EL	Expression Language
ETA	Edição Textual Assistida
FISL	Fórum Internacional de Software Livre
GBM	Geração Baseada em Modelos
GUI	Graphic User Interface
GWT	Google Web Toolkit
IDE	Integrated Development Environment
IHC	Interface Humano-Computador
IU	Interface do Usuário
JDNC	Java Desktop Native Controls
JSF	Java Server Faces
JSR	Java Specification Request
MB	Model-Based
MBD	Model-Based Development
MBUIDE	Model-Based User Interface Development Environment
MVC	Model-View Controller
OCL	Object Constraint Language
OMG	Object Management Group
PAC	Presentation-Abstract-Control
RAD	Rapid Application Development

SBES	Simpósio Brasileiro de Engenharia de Software
SGBD	Sistema de Gerenciamento de Banco de Dados
SWT	Standard Widget Toolkit
WYSIWYG	What You See Is What You Get
XHTML	Extensible Hypertext Mark-up Language
XUL	XML-based User Interface Language



## LISTA DE FIGURAS

Figura 2.1 : Um típico Modelo de Domínio (mostrado como um Diagrama de Classes em notação UML) e uma possível interface CRUD produzida por uma MBUIDE rudimentar.....	20
Figura 2.2: Modelo de Tarefas da ferramenta TRIDENT (SOUCHON, LIMBOURG e VANDERDONCK, 2002). .....	22
Figura 2.3 : Trechos de conteúdo dos Modelos de Apresentação nas ferramentas TRIDENT (BODART, LEHEUREUX e VANDERDONCKT, 1994) e MERLIN (MRACK, 2007). .....	24
Figura 2.4: Arquitetura geral das MBUIDE. ....	26
Figura 3.1: Uma interface CRUD simples gerada pelo software METAGEN (MOREIRA e MRACK, 2003). .....	31
Figura 3.2: Custo da Interface de Usuário e das Telas de Cadastro (TC) em um sistema de banco de dados qualquer (MRACK, 2007). ....	33
Figura 3.3: Transformação do grafo de objetos (Modelo de Domínio) na respectiva IU. ....	36
Figura 3.4: Interligação de eventos e regras de negócio para controles de IU utilizando Agentes na forma de anotações sobre o Modelo de Domínio. ....	38
Figura 3.5: Uso do assistente textual para edição do Modelo de Domínio do MERLIN. ....	40
Figura 3.6: Suporte para a correção de erros durante a edição dos modelos da ferramenta MERLIN. ....	41
Figura 3.7: Redefinindo a ordem de controles e a posição de textos descritivos na IU. ....	43
Figura 3.8: Um exemplo de <i>layout</i> de IU gerado a partir de um <i>template</i> manual. ....	43
Figura 3.9: Mensagens de auxílio com dicas de preenchimento de tela geradas automaticamente. ....	47
Figura 3.10: Utilização de similaridades e sinônimos para geração dos controles na IU. ....	48
Figura 3.11: Processo de desenvolvimento de interfaces CRUD utilizando a ferramenta MERLIN. ....	49
Figura 4.1: Listagem do código-fonte da primeira interação de desenvolvimento do Estudo de Caso sobre o software MERLIN. ....	52
Figura 4.2: Resultado da geração da IU na primeira interação de desenvolvimento do Estudo de Caso sobre o software MERLIN. ....	54
Figura 4.3: Listagem do código-fonte da segunda interação de desenvolvimento do Estudo de Caso. ....	57
Figura 4.4: Resultado da geração da IU na segunda interação de desenvolvimento do Estudo de Caso sobre o software MERLIN. ....	58

## LISTA DE TABELAS

Tabela 3.1: Padrões utilizados pela ferramenta MERLIN.....	35
Tabela 3.2: Necessidades e propostas correntes para o mecanismo de configuração realimentada do MERLIN. ....	45
Tabela A.1: Tabela comparativa entre as características das MBUIDE.....	72
Tabela B.1: Conjunto de anotações de configuração e propriedades definidas pela ferramenta MERLIN.....	75

## RESUMO

A geração automatizada de Interfaces de Usuário (IU) é objeto de estudo há muitos anos. Desde a década de 80, dezenas de projetos foram desenvolvidos e várias soluções apresentadas ao mercado.

Entretanto, mesmo com os avanços obtidos neste cenário, inúmeras dificuldades ainda continuam presentes no dia-a-dia dos desenvolvedores de sistemas. A necessidade de processos, sintaxes e linguagens proprietárias associado ao elevado esforço de configuração e à baixa reutilização de tecnologias são os principais problemas nessa área.

Objetivando sanar essas dificuldades, este trabalho propõe uma solução diferenciada para o problema, a qual reutiliza e integra-se à maioria das tecnologias comumente presentes nos ambientes de desenvolvimento e que conta com um exclusivo sistema de configuração, capaz de minimizar o trabalho de geração das IU.

Sendo um típico gerador baseado em modelos, o software proposto é chamado MERLIN, e objetiva automatizar completamente a geração de interfaces CRUD, as quais estão presentes em até 30% dos sistemas que operam sobre banco de dados.

Para alcançar este objetivo, o software aposta no uso intensivo de heurísticas e na formação de uma estrutura auto-contida e realimentada de configurações, a qual reside unicamente nas classes compiladas da aplicação. Completando a sua arquitetura, um processo de geração em tempo de execução inibe a produção de qualquer linha de código-fonte, o que evita a necessidade de refatoração ao longo da evolução dos sistemas.

Com esses elementos em evidência e focando inicialmente a plataforma Java, sinaliza-se uma solução diferenciada, apta para ser utilizada em ambientes profissionais de desenvolvimento.

**Palavras-Chave:** IHC, IU, MBUIDE, MERLIN, Geração Baseada em Modelos, Interface de Usuário, Java.

## **Automatic and Aided Generation of User Interfaces**

### **ABSTRACT**

The automated generation of User Interfaces (UI) has been the object of study for many years. Since the 1980s, dozens of projects have been developed and various solutions presented to the market.

However, even with the advances obtained under this scenario, innumerable difficulties still continue to present themselves in the daily routine of system developers. The demands of proprietary methodologies, syntaxes and languages, the high level of effort needed for configuration and low reuse of technologies are the main problems in the area.

With the object of rectifying these difficulties, this work proposes a solution specific to the problem, which reuses and combines the majority of the technologies already existing in development environments and relies on an exclusive configuration system, capable to minimize the work of generating the UI.

Being a typical model based generator, the software under consideration is called MERLIN, and has as its objective to completely automate the generation of CRUD interfaces, which are present in up to 30% of the systems that interact with data bases.

To achieve this objective, the software relies on the intensive use of heuristics and the creation of a self contained configuration feedback structure, which exists solely in the compiled classes of the application. Completing this architecture, a process of execution time generation eliminates the need for any source code, which significantly reduces the costs of refactoring the code throughout the evolution of the systems.

With these elements in evidence and focusing initially on the Java platform, indicates that this distinguished solution is ready for use in professional development environments.

**Keywords:** HCI, UI, MBUIDE, MERLIN, Model-Based Generation, User Interface, Java.

# 1 INTRODUÇÃO

Nos últimos anos, o advento das Interfaces Gráficas de Usuário (GUI) modificou decisivamente a forma com que as Interfaces de Usuário (IU) são concebidas.

Contrastando com os antigos sistemas baseados em IU de modo caractere, as modernas interfaces gráficas apresentam semântica padronizada, elaborados mecanismos para o tratamento de eventos e uma diversidade de elementos de interação (GALITZ, 2002). A associação dessas características a um modelo de interação baseado em eventos disparados pelo usuário acabam elevando a complexidade para a construção desse tipo de software.

Com o intuito de reduzir essa complexidade ou, pelo menos, torná-la gerenciável, existem no mercado as ferramentas de Desenvolvimento Rápido de Aplicações (RAD). Essas ferramentas propõem realizar esta tarefa através do uso de editores visuais, os quais utilizam o conceito WYSIWYG (*What You See Is What You Get*), ou seja, o que você vê (ou, o que você desenha no editor) é o que você tem quando o sistema estiver em execução. Devido à sua relativa facilidade de uso, essas ferramentas são bastante populares e aceitas universalmente como opções de primeira linha para a construção de qualquer aplicativo que possua uma GUI (SCHLUNGBAUM, 1997).

Ao se considerar as IU para sistemas de banco de dados, a realidade não é diferente. Ou seja, devido à inerente complexidade do modelo de dados da aplicação, as suas interfaces do usuário acabam tornando-se complicadas por conseqüência. Ainda, considerando os sistemas desenvolvidos sob o paradigma da Orientação a Objetos (OO), essa complexidade torna-se cada vez mais presente.

Por outro lado, a existência de um modelo de dados fortemente relacionado à interface do usuário também pode se traduzir em ganhos para a sua construção (SHIROTA e IIZAWA, 1997). Isso ocorre porque muitas informações da IU podem ser inferidas a partir do próprio modelo de dados do sistema. Relações entre o tipo de um dado e o controle de tela associado, ou a obrigatoriedade de um campo e as regras de validação atreladas, são exemplos típicos e que são explorados com relativo sucesso em trabalhos de geração automatizada de IU (GOHIL, 1999), (CLIFTON, 2005a), (ALOIA, 2003), (MOREIRA e MRACK, 2003).

É partindo dessa mesma proximidade entre o modelo de dados da aplicação e as suas respectivas interfaces de usuário, que freqüentemente as ferramentas RAD disponibilizam módulos auxiliares para criação dessas IU. São os chamados Assistentes de Criação, ou simplesmente *wizards*. Através de uma série de telas, esses assistentes ajudam o programador a coletar informações da estrutura de um modelo de dados (como uma tabela em um SGBD) para produzir como resultado o código-fonte de uma IU capaz de exibir e editar os dados contidos nesse modelo (GALITZ, 2002).

De um ponto de vista amplo, é possível afirmar que o produto de tais geradores é uma interface de usuário simples, com um *layout* de tela rústico e pequena preocupação quanto ao acabamento dos controles de tela produzidos. De forma geral, feita uma primeira geração e com os devidos ajustes manuais realizados por um desenvolvedor, as interfaces produzidas por esses mecanismos tendem a entrar em produção em tempos menores se comparadas à programação tradicional (MYERS, 1995).

Todavia, se por um lado, o primeiro ciclo de desenvolvimento apresenta ganhos em relação à produtividade, com o passar do tempo, quando o sistema cresce e regras adicionais precisam ser acrescentadas ou modificadas, esta facilidade inicial acaba acarretando diversos problemas. Alterações na distribuição e posicionamento de controles das telas, adequação do código produzido pelo gerador aos padrões definidos pela equipe de desenvolvimento e modificações de estilo, aparência, usabilidade ou comportamento são freqüentemente ignoradas por esse tipo de software. Atividades para correção manual desses problemas acabam consumindo grande parte do tempo e esforço dos programadores.

A característica ausente nesses tipos de assistentes é conhecida como *roundtrip*, a qual pode ser entendida como a capacidade do gerador suportar e interagir, ao longo de todos os ciclos de desenvolvimento, com as alterações de código realizadas manualmente pelos programadores (MOREIRA e MRACK, 2003).

Objetivando contornar essas dificuldades, existe um outro conjunto de soluções automatizadas, que englobam o chamado *Model-Based Development* (MBD) (MYERS, 1995). Ferramentas norteadas por esse conceito também utilizam modelos para geração da interface do usuário, porém elas ultrapassam as limitadas capacidades dos assistentes de geração. Isso ocorre, em grande parte, porque o nível de detalhamento e a variedade de modelos suportados por tais ferramentas vão muito além daqueles utilizadas pelos assistentes de geração tipicamente presentes nas ferramentas RAD.

Enquanto a maioria dos *wizards* comentados anteriormente vale-se de um modelo de dados pré-existente, como o esquema de um banco de dados, os geradores baseados em modelos suportam muitos outros, capazes de armazenar não somente informações estruturais, mas também comportamentais e de aparência. Modelos de Domínio (que contém informações sobre os objetos da aplicação e seus relacionamentos), de Tarefas (os quais descrevem os eventos, condições e atividades disponíveis na IU), de Plataforma (que detalham o ambiente em que o sistema está sendo executado) e de Usuário (capazes de especificar preferências de uso com base no perfil do usuário) são apenas alguns dos tipos de modelos encontrados nessa gama de ferramentas (PUERTA e EISENSTEIN, 1999). É justamente essa riqueza de informações que possibilita as maiores e melhores capacidades de geração desse tipo de software.

Geradores baseados em modelos não são uma novidade na área de geração de IU. Na literatura da área, identifica-se nada menos do que 15 projetos ao longo das duas últimas décadas de pesquisas (SCHULUNGBAUN, 1996). Entretanto, e de forma geral, o fato é que os geradores baseados em modelos ainda não alcançaram os objetivos previamente estabelecidos, os quais tangem principalmente à facilidade de uso, rapidez de desenvolvimento e integração ao ambiente de programação (PUERTA e EISENSTEIN, 1999).

Dentre as muitas dificuldades encontradas no caminho, o não seguimento das tendências de mercado, a utilização de linguagens e modelos proprietários, o foco essencialmente acadêmico, a busca pela generalidade das soluções e a falta de continuidade nesses projetos implicam na colocação desse tipo de software em segundo plano dentro dos ambientes profissionais de desenvolvimento (PUERTA et al., 1996) e (MYERS, 1995). Corroborando com essas afirmativas, está o panorama moldado pelas pesquisas realizadas para esse trabalho, as quais indicam que essas ferramentas alcançaram seu ápice na década de 90 e hoje passam quase despercebidas no cenário comercial.

O histórico desta dissertação parte de um projeto iniciado em 2001, onde muitas expectativas foram idealizadas para a construção de um tipo específico de gerador baseado em modelos, o qual tinha como premissa a geração de IU para visualização e edição de dados provenientes de Sistemas Gerenciadores de Banco de Dados (SGBD) relacionais. Com base em tecnologias de mercado e utilizando como fonte de informações o esquema de um banco de dados enriquecido com algumas meta-informações, o software METAGEN (MOREIRA e MRACK, 2003) era um interpretador que produzia, em tempo de execução, as IU de diversos sistemas baseados em banco de dados relacionais

Esta era uma abordagem completamente nova para o cenário, pois as IU das aplicações não mais possuíam um código-fonte associado e, portanto, não sofreriam os efeitos tipicamente nocivos para um gerador, como a alteração manual do código-fonte gerado (CLIFTON, 2005a). Entretanto, embora alguns protótipos de sucesso tenham sido construídos, as verbas e os esforços para a continuidade do trabalho acabaram sendo redirecionados pela gerência sênior da instituição onde o trabalho era desenvolvido.

Tendo confiança na proposta idealizada e sendo fortemente influenciado pelas premissas defendidas pela comunidade de Software Livre, este projeto continuou sendo desenvolvido através de um esforço individual. Nesse ínterim a implementação da ferramenta fora trazida para a plataforma Java e novas possibilidades surgiram, como o suporte OO e o alcance dos ambientes *Web* e *Desktop*. Nesse momento, em 2004, o projeto de nome MERLIN, ganhava forma (MERLIN, 2005).

Passados alguns anos daquela proposta inicial, e com um respaldo positivo nos eventos onde o trabalho fora apresentado (MOREIRA, MRACK e PIMENTA, 2006) (MRACK, 2007), diversos conceitos foram sendo paulatinamente introduzidos na ferramenta, como o uso intensivo de heurísticas, empirismo, realimentação e um mecanismo auto-contido para armazenamento das configurações.

Entretanto, mesmo com os avanços obtidos, muitos desafios continuaram presentes. Em um trabalho de revisão bibliográfica (MRACK, 2005), observou-se que não apenas era preciso um maior detalhamento das soluções a serem implementadas, mas também a obtenção de subsídios para a escolha das melhores tecnologias a serem empregadas e mesmo o enquadramento desse projeto em relação às outras soluções existentes.

Nesse sentido, esta Dissertação de Mestrado representa uma oportunidade única para este projeto, permitindo descrever seus elementos em detalhes e comparando suas funcionalidades frente a outras soluções da área.

## 1.1 Contribuição

O objetivo geral desta Dissertação de Mestrado é descrever a proposta de geração automatizada de interfaces de usuário oferecida pelo software MERLIN, o qual se enquadra como uma ferramenta de geração baseada em modelos.

Em relação às contribuições deste trabalho frente às soluções existentes, citam-se dois aspectos: (i) o processo de desenvolvimento e tecnologias de suporte utilizadas pela ferramenta, e (ii) as características do software em si.

Quanto ao processo e tecnologias utilizadas, a proposta é totalmente baseada em padrões de mercado. Com ênfase na plataforma Java e tecnologias associadas, a premissa é uma curva de aprendizagem reduzida através do reuso de soluções existentes e com o máximo de neutralidade frente às metodologias de desenvolvimento já em uso nas empresas.

Com relação às características da ferramenta MERLIN, e antecipando o que será detalhado no decorrer do trabalho, os dois principais diferenciais do software proposto são: (i) o uso de um mecanismo realimentado de configurações, o qual objetiva a minimização dos esforços de configuração ao longo do tempo e (ii) a utilização de uma estrutura auto-contida para o armazenamento de seus modelos, a qual evita a necessidade de arquivos externos de configuração, reduzindo significativamente as dependências do sistema.

## 1.2 Organização do texto

Esta Dissertação de Mestrado está organizado em 5 capítulos.

O **Capítulo 1** é esta Introdução.

O **Capítulo 2** apresenta um levantamento bibliográfico sobre as soluções existentes na área da geração de IU baseada em modelos. Um breve histórico do cenário é apresentado, e os termos e conceitos da tecnologia são descritos. Na parte final, uma tabela comparativa é apresentada, objetivando enquadrar a proposta desse trabalho frente às soluções existentes.

O **Capítulo 3** é o ponto central do trabalho. É nele que a proposta oferecida pelo projeto MERLIN é apresentada. O capítulo inicia com uma retrospectiva do projeto e os seus principais objetivos. A partir de então, o texto segue com seções específicas que descrevem cada uma das características da ferramenta e sua arquitetura de funcionamento. O capítulo se encerra com uma visão geral do software e a sua colocação dentro de um processo de desenvolvimento de sistemas.

O **Capítulo 4**, objetivando validar os conceitos defendidos pela proposta, apresenta um Estudo de Caso baseado na versão atual da ferramenta. Para cumprir esse objetivo, é descrita passo-a-passo a construção de uma interface de usuário, a qual inclui não somente a apresentação de elementos de dados e *layout* de controles, mas também a vinculação de regras de negócio e comportamentos e aparência específicos aos



elementos de IU gerados.

A **Conclusão** encerra o trabalho fazendo uma recapitulação geral dos temas abordados durante a pesquisa. Nesse ponto são evidenciadas as contribuições do projeto e expostos os limites da solução existente. O texto encerra com uma visão geral sobre o estado atual da pesquisa e os direcionamentos previstos para trabalhos futuros.

## 2 GERAÇÃO DE IU BASEADA EM MODELOS

*Este capítulo apresenta uma visão geral sobre as ferramentas de geração de IU baseadas em modelos, introduzindo os principais conceitos e termos dessa tecnologia e fazendo uma análise sobre os projetos mais relevantes da área. Ao final do capítulo, um enquadramento entre esses projetos é realizado com o intuito de salientar as características mais relevantes.*

### 2.1 Histórico

A geração de IU baseada em modelos tem suas origens em um tipo de software conhecido como *User Interface Management System* (UIMS), aparecido no início da década de 80 (SCHLUNGBAUM, 1997). Os UIMS foram a primeira alternativa em relação à construção manual de interfaces de usuário, possibilitando aos programadores escrever aplicações em uma linguagem de mais alto nível, ao invés de trabalhar diretamente com um *toolkit* gráfico. Construídas as especificações, o gerador poderia, então, transformá-las no programa final ou, de forma alternativa, interpretá-las em tempo de execução, gerando assim a respectiva IU (SZEKELY, 1996).

A maioria dos primeiros UIMS era focada na especificação do *diálogo* existente entre o usuário e a IU da aplicação. Através de diagramas de transição de estados, gramáticas especiais ou representações baseadas em eventos de IU, essas ferramentas possibilitavam a especificação das respostas do sistema em relação aos eventos de entrada produzidos pelo usuário. Já no tocante à aparência da IU, geralmente esse detalhamento era feito em linguagens próprias, as quais eram relacionadas ao restante da aplicação através da chamada de métodos especializados. Como produtos dessas primeiras UIMS estavam a geração de menus de sistema, a vinculação de eventos, as chamadas de funções pré-determinadas e a produção de caixas de diálogo simples para a exibição de mensagens de usuário (SZEKELY, 1996).

### 2.2 As MBUIDE

Entre os anos de 1980 e 1990 aproximadamente, as linguagens de especificação tornaram-se mais sofisticadas, suportando representações mais ricas e detalhadas,

permitindo a construção de interfaces de usuário mais complexas e robustas. Nessa época, diversos conceitos desenvolvidos de forma paralela em vários projetos acabaram sedimentando-se, e uma nova gama de ferramentas aparecera (SCHLUNGBAUM, 1996).

Enquadrando-se na categoria das ferramentas *Model-Based*, as chamadas *Model-Based User Interface Development Environment* (MBUIDE) são a evolução das UIMS e, conforme alguns críticos argumentam (SCHLUNGBAUM e THOMAS, 1996), representam a gama de ferramentas mais promissora em relação à geração automatizada de interfaces de usuário.

As MBUIDE utilizam um conceito essencial para alcançar seus objetivos: valer-se de modelos declarativos de alto nível como elementos-chave em um processo de desenvolvimento interativo e incremental. De forma geral, a idéia é que tais modelos possam ser capazes de descrever todos os aspectos relevantes de um sistema, cabendo ao mecanismo gerador, a tarefa de transformá-los no aplicativo final (PUERTA e EISENSTEIN, 1999).

Embora norteada por uma premissa simples, o funcionamento e o uso de uma MBUIDE são complexos, e, nesse sentido, três elementos ganham relevância: o conjunto de modelos suportados, o processo de desenvolvimento amparado pela ferramenta e a arquitetura de geração da IU (GRIFFITHS et al., 1997).

Se entender as características de cada um desses componentes é uma necessidade para qualquer desenvolvedor que deseja utilizar uma solução desse porte, a exploração detalhada de cada um desses itens é tarefa de suma importância quando se deseja construir um software desse tipo.

Sendo o objetivo dessa dissertação apresentar uma proposta diferenciada na área das ferramentas MBUIDE, as seções a seguir apresentam maiores detalhes sobre cada um dos três elementos citados acima.

### **2.2.1 Os Tipos de Modelos**

Como elementos centrais nas MBUIDE estão seus modelos de armazenamento e configuração, os quais variam em conteúdo e objetivos. Análises diversas (PINHEIRO, 2001) (SZEKELY, 1996) (SCHLUNGBAUM, 1996) afirmam que as MBUIDE podem ser classificadas cronológica e arquiteturalmente conforme o conjunto de modelos suportados.

De fato, é a partir dos modelos que as ferramentas definem suas formas de trabalho e, conseqüentemente, o processo de desenvolvimento a ser aplicado quando da sua utilização (BODART, LEHEUREUX e VANDERDONCKT, 1994). Partindo dessa afirmativa, uma análise sobre esses artefatos é de grande importância para esse trabalho. Assim, as seções 2.2.1.1 a 2.2.1.9 descrevem, em alto nível, os nove tipos de modelos consensualmente aceitos na área das ferramentas de geração baseada em modelos.

### 2.2.1.1 Modelo de Dados

O Modelo de Dados é a base para geração de interfaces de CRUD<sup>1</sup>, pois tem um mapeamento quase direto entre um elemento de dado (um atributo em uma tabela, por exemplo) e um controle de tela (uma caixa de texto, por exemplo) e são freqüentemente utilizados pelas MBUIDE de primeira geração (SZEKELY, 1996).

Sob uma ótica simples, esse modelo pode ser interpretado como um subconjunto do Modelo de Domínio (descrito a seguir), e pode ser comparado a um típico modelo Entidade-Relacionamento, contendo nele todas as informações sobre os dados que podem ser manipulados pelo sistema.

Obrigatoriedade, valores-padrão, intervalos mínimos e máximos, tipos de dado e outras informações tipicamente presentes nesse modelo são utilizadas pela grande maioria das MBUIDE para produzir, com a ajuda de algumas heurísticas e algoritmos simplistas, as interfaces responsáveis pela visualização e edição da estrutura de dados relacionada (PUERTA et al., 1996).

Uma MBUIDE de primeira geração, utilizando um simples Modelo de Dados, produziria uma IU como a mostrada na Figura 2.1:

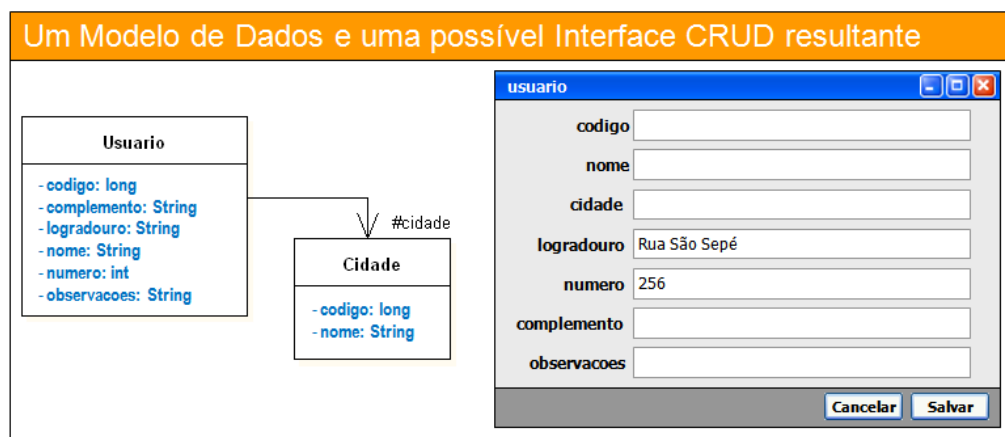


Figura 2.1 : Um típico Modelo de Domínio (mostrado como um Diagrama de Classes em notação UML) e uma possível interface CRUD produzida por uma MBUIDE rudimentar.

Observa-se na Figura 2.1 que os nomes dos controles (à esquerda na IU) foram produzidos a partir do próprio nome do atributo da classe, e que os controles de edição (à direita na IU) foram gerados em função do tipo de dado do atributo.

Como regra geral, as MBUIDE mapeiam uma tabela ou classe para uma ou várias interfaces CRUD, sendo que cada atributo é mapeado para um controle de tela (PINHEIRO, 2001) (MOREIRA e MRACK, 2003).

<sup>1</sup> Do inglês *Create, Retrieve, Update and Delete*, indicam IU responsáveis pela criação, visualização, edição e exclusão de dados tipicamente relacionados à SGBD.

### 2.2.1.2 *Modelo de Domínio*

O Modelo de Domínio é o elemento central das MBUIDE, servindo como base para geração da IU. A partir dele são extraídas várias informações que, através de heurísticas e algoritmos diversos, podem ser derivadas nos elementos que constituem os formulários da aplicação.

Este modelo excede o Modelo de Dados por conter também as informações comportamentais do sistema. De certa forma, ele pode ser interpretado como um Diagrama de Classes da UML, suportando diversos tipos de relacionamentos entre as estruturas (como herança, agregação ou composição), bem como tipos de dados definidos pelo usuário e também conceitos de mais alto nível, como classes abstratas, finais, interfaces e enumerações (OMG, 2007a).

Porém, o que mais distingue esse modelo do Modelo de Dados é a presença de operações, ou métodos de classes. Dessa forma, um Modelo de Domínio pode representar não somente a estrutura de uma IU, mas também o seu comportamento. Projetos como o MECANO (PUERTA et al., 1996) e METAGEN (MOREIRA e MRACK, 2003) são apenas alguns exemplos de soluções que centralizam o desenvolvimento do sistema neste modelo.

Conforme análises (PINHEIRO, 2001) (SZEKELY, 1996), os Modelos de Domínio vieram para englobar os Modelos de Dados e são vistos como elementos essenciais nas MBUIDE mais recentes.

### 2.2.1.3 *Modelo de Tarefa*

Este modelo pode ser interpretado como uma grande estrutura no formato de árvore composta de tarefas e sub-tarefas, onde cada elemento representa uma atividade que o usuário pode executar no sistema. Como não existe um padrão para descrever esses modelos, cada ferramenta que o adota, o faz com base em um estilo próprio (SCHULUNGBAUM, 1996).

Entre as informações contidas nesse modelo, estão a ordem de execução das tarefas, pré e pós-condições, questões sobre paralelismo, dependências, obrigatoriedade e outras (GRAY et al., 1998). No tocante à vinculação dessas tarefas aos elementos de tela, algumas soluções fazem-no diretamente nesse modelo e outras optam por fazer isso junto ao Modelo de Diálogo (descrito adiante).

Entre os tipos de tarefas existentes, citam-se as abstratas (geralmente representadas por frases estruturadas como “Efetuar Saque; R\$ 200,00”); as concretas (comumente atreladas à chamadas de métodos, como “efetuarSaque({200,REAL})” e as formais (construídas frequentemente com bases em formalismos ou gramáticas proprietárias de difícil compreensão) (BARON e GIRARD, 2002).

Uma das MBUIDE que mais coloca ênfase sobre esse modelo é a TRIDENT (BODART, LEHEUREUX e VANDERDONCKT, 1994), tomando-o como ponto de referência não só para a definição dos outros modelos do sistema, mas também para a modelagem e condução de todo o processo de desenvolvimento da aplicação.

A Figura 2.2 mostra a representação gráfica de um modelo de tarefas derivado do projeto TRIDENT, no qual as tarefas são descritas através de um modelo formal (SOUCHON, LIMBOURG e VANDERDONCK, 2002):

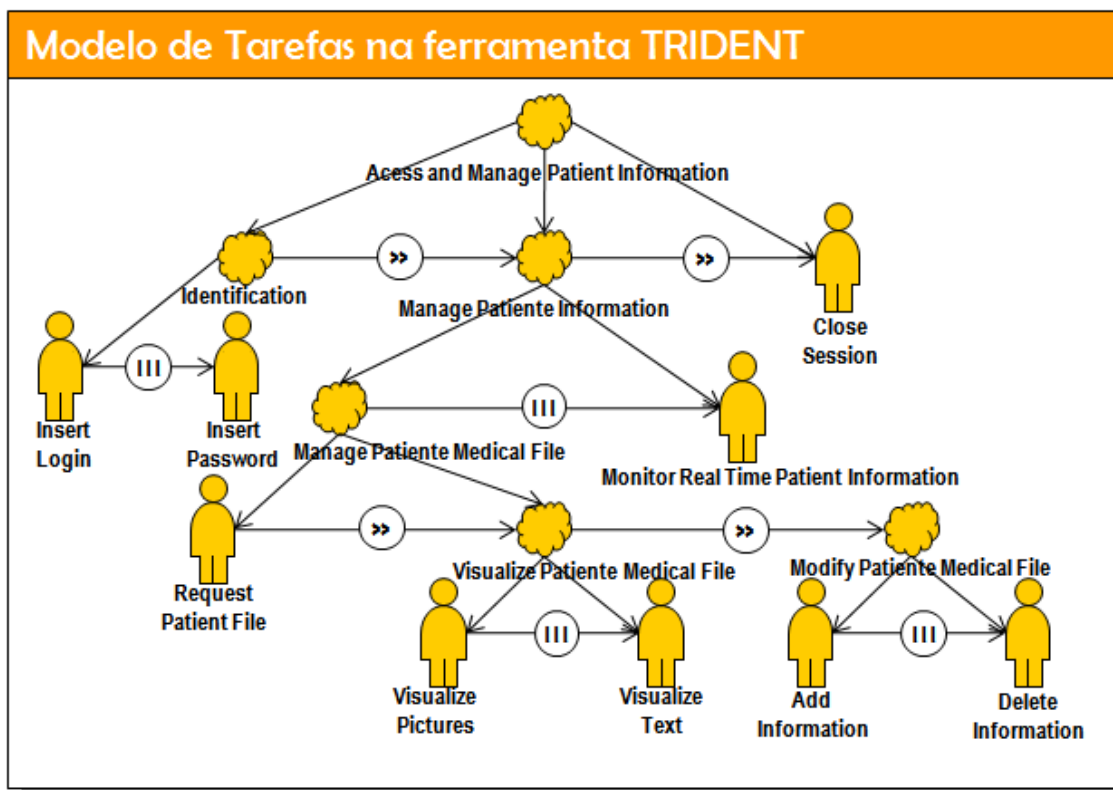


Figura 2.2: Modelo de Tarefas da ferramenta TRIDENT (SOUCHON, LIMBOURG e VANDERDONCK, 2002).

#### 2.2.1.4 Modelo de Usuário

Este modelo não tem uma descrição ou comportamento exatos. De fato, muitas ferramentas simplesmente o ignoram, e a abordagem utilizada por elas para satisfazer esse componente é dispersar suas características nos outros modelos.

Na prática, o objetivo desse componente é armazenar características de um usuário ou perfil específico da aplicação em relação às configurações gerais da ferramenta. Em outras palavras, é nesse modelo que devem ficar armazenados comportamentos e especializações do sistema, como teclas de atalho personalizadas, valores-padrão para controles de tela baseados em perfil do usuário, informações referentes à controle de acesso e outras (SCHLUNGBAUM, 1997).

A maioria das ferramentas existentes não explicita em detalhes o funcionamento desse elemento, mas, de modo geral, são capazes de satisfazê-lo em determinado nível e à sua maneira (PINHEIRO, 2001).

### 2.2.1.5 Modelo de Diálogo

Também conhecido como Modelo de Conversação (SCHULUNGBAUM, 1996), define como os objetos da IU devem interagir com o usuário. Ele representa as ações que o usuário pode invocar através dos elementos da camada de apresentação, a ordem em que podem ser invocadas e as respostas que o sistema deve produzir através desses mesmos elementos.

De forma geral, ele está muito ligado ao Modelo de Tarefas e ao Modelo de Apresentação (descrito a seguir), sendo que, muitas vezes, confunde-se com esses. Essa afirmativa pode ser tanto apreciada em (PINHEIRO, 2001), que simplesmente une-os em seu ensaio comparativo de ferramentas da área, como em (SCHULUNGBAUM, 1996), que descreve o aparecimento desse modelo como sendo uma evolução arquitetural dos modernos MBUIDE.

### 2.2.1.6 Modelo de Apresentação

Após o Modelo de Domínio, o Modelo de Apresentação pode ser considerado o segundo modelo mais importante em um MBUIDE. É nele que ficam armazenadas as informações referentes à aparência da tela (PUERTA e EISENSTEIN, 1999). Características dos elementos gráficos como *layout* de controles, posicionamento, tamanho, regras para redimensionamento, esquema de cores, ordem de tabulação e outras relacionadas à aparência da IU são centralizadas nesse componente.

Devido à sua acentuada dependência em relação ao *toolkit* gráfico utilizado pelo sistema, os MBUIDE encontram nesse item um separador de águas em relação à sua arquitetura interna. Isso ocorre porque elas precisam considerar duas abordagens diferentes para estruturar esse elemento:

- **Abstract Interface Objects (AIO):** Ferramentas que utilizam informações abstratas para descrever o Modelo de Apresentação simplesmente ignoram o *toolkit* gráfico para configuração desse componente. Assim, é eleito um conjunto de configurações que podem estar presentes nos *toolkits* gráficos suportados pelo gerador de forma que, em um segundo instante, na geração da IU, as informações abstratas são mapeadas pelo gerador para o *toolkit* gráfico escolhido. Na eventualidade de uma informação existente no AIO não estar presente no *toolkit* gráfico escolhido, uma perda de qualidade no resultado é esperada (BODART, LEHEUREUX e VANDERDONCKT, 1994).
- **Concrete Interface Objects (CIO):** Soluções que usam essa abordagem definem o modelo de apresentação utilizando as primitivas do próprio *toolkit* gráfico a ser utilizado na geração da IU.

É interessante notar que alguns trabalhos defendem que é possível operar ambos os modelos ao mesmo tempo, de forma que, em um processo evolutivo, AIO são paulatinamente transformados em CIO. A justificativa para isso é que somente através dos CIO é possível especificar os detalhes necessários para a geração de uma IU mais elaborada. O MERLIN, descrito no Capítulo 3 deste trabalho, faz parte desse conjunto

de soluções.

Na Figura 2.3 pode ser vista a comparação entre fragmentos do Modelos de Apresentação nas ferramentas TRIDENT e MERLIN. Ambos trechos representam o mesmo objeto de tela, no caso, um controle para exibição e edição do código identificador de um cliente.

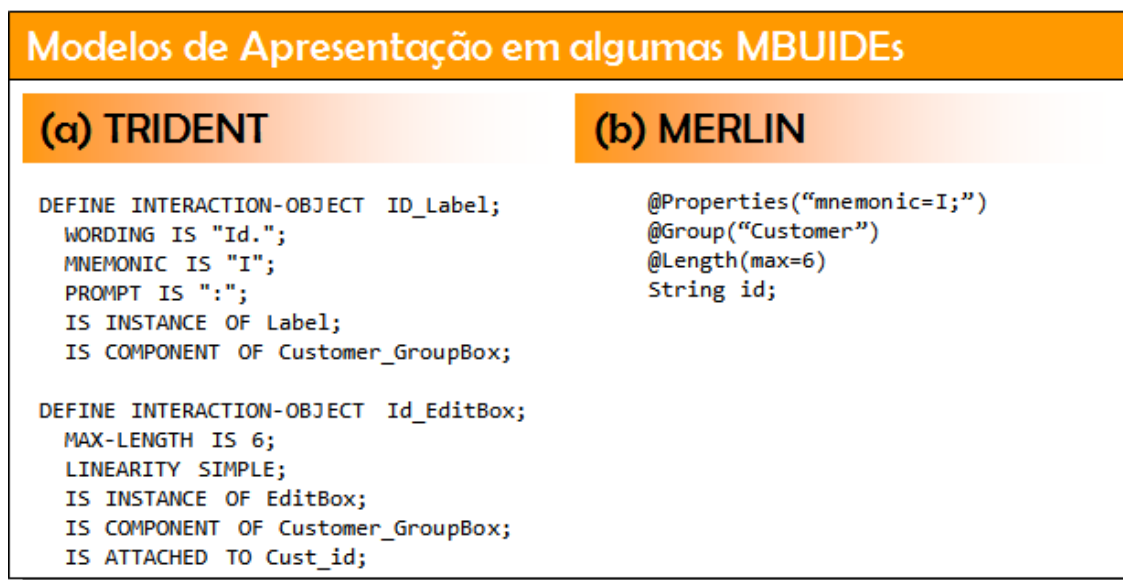


Figura 2.3 : Trechos de conteúdo dos Modelos de Apresentação nas ferramentas TRIDENT (BODART, LEHEUREUX e VANDERDONCKT, 1994) e MERLIN (MRACK, 2007).

De importante interesse, porém, pode ser a abordagem oferecida pela ferramenta MOBI-D (PUERTA e EISENSTEIN, 1999), a qual explicita um mecanismo configurável e relativamente avançado para mapeamento entre AIO e CIO. Tal ênfase é tão grande, que os autores sugerem para isso um novo tipo de modelo, chamado Modelo de Projeto. Entretanto, uma vez que somente este trabalho referencia tal componente, este não é considerado um elemento ao mesmo nível dos outros modelos existentes.

#### 2.2.1.7 Modelo de Aplicação

Este modelo pode ser interpretado como o superconjunto do Modelo de Domínio, pois contém informações que vão além daquelas existentes naquele componente. De forma geral, um Modelo de Aplicação descreve as características mais gerais do sistema, e não simplesmente aquelas referentes ao problema de IU a ser tratado.

São exemplos de informações contidas nesse modelo as classes auxiliares de suporte à aplicação, dados de configuração do sistema e parâmetros como conexões à banco de dados, serviços remotos e outros.

Quanto à formação, detalhamento e fronteiras desse componente, os trabalhos não são explícitos, sendo que muitos simplesmente ignoram-no (PINHEIRO, 2001).



### 2.2.1.8 Modelo de Plataforma

Da mesma forma que o Modelo de Aplicação, não existe um consenso sobre esse modelo. Algumas ferramentas simplesmente descartam-no (PUERTA et al., 1996) e outras fundem-no com o Modelo de Aplicação (LUYTEN, 2004) sem maiores preciosismos.

O objetivo desse modelo é descrever os detalhes necessários da aplicação perante o ambiente em que ela está sendo executada. Informações como o *toolkit* gráfico e o banco de dados a ser utilizado pelo sistema são descritos nesse componente. Embora ele seja de suma importância para a completude de uma MBUIDE, a existência explícita desse componente não é justificável, uma vez que cada ferramenta pode escolher a melhor forma de implementá-lo sem demasiados prejuízos para o cenário de desenvolvimento.

### 2.2.1.9 Modelo de Contexto

De forma geral, esse artefato não é considerado um modelo no sentido exato da palavra, pois, com exceção à (LUYTEN, 2004), nenhum dos trabalhos estudados faz menção explícita a esse tipo de componente.

Um Modelo de Contexto descreve os aspectos de uma aplicação que não podem ser conhecidos durante sua construção, pois dependem da interação do usuário para sua formação. Entre as informações que podem residir nesse modelo estão dados comportamentais do usuário, como preferências na execução de comandos (se por teclas de atalho ou chamadas de menus, por exemplo), ordem de preenchimento de campos na tela, utilização ou não de valores-padrão existentes nos controles da IU e outros.

Em suma, esse é um modelo que somente pode ser obtido ao longo do uso do sistema, não sendo possível sua concepção durante a fase de projeto da aplicação. Em certos aspectos de seu conteúdo, por conter informações sobre preferências de uso, pode ser confundido com o Modelo de Usuário e é por este motivo que algumas ferramentas acabam fundindo os dois (PINHEIRO, 2001).

## 2.2.2 Tipos de Ferramenta

Embora o cenário mundial contenha quase duas dezenas de ferramentas MBUIDE, cada qual variando consideravelmente em relação às funcionalidades oferecidas e sua estruturação interna, o fato é que muitas características são comuns, permitindo enquadrá-las em certo grau. Desse enquadramento, dois tipos de ferramenta surgem (PUERTA et al., 1996):

- **Ferramentas de Assistência:** Softwares desse tipo normalmente incluem editores visuais com avançadas capacidades de visualização e edição dos modelos suportados pela ferramenta. A filosofia desses assistentes é *facilitar* as tarefas de manipulação dos seus modelos através de interfaces com alto nível de abstração. O TRIDENT é típico exemplo de software assistente.
- **Ferramentas de Automação:** Geradores desse tipo possuem algoritmos

complexos, capazes de fazer transformações de grande monta sobre os modelos. Como resultado dessas transformações, está uma IU funcional e, possivelmente, completa. A filosofia desse tipo de MBUIDE é *minimizar* o trabalho necessário para configurar os modelos. Entretanto, para que isso seja possível, essas ferramentas exigem um conhecimento de mais baixo nível em relação às estruturas e linguagens envolvidas na configuração dos modelos. O MERLIN e o MECANO são bons exemplos de software de automação.

### 2.2.3 Arquitetura para Geração da IU

No tocante à arquitetura de geração da IU, as MBUIDE podem ser agrupadas em três grandes grupos (PINHEIRO, 2001). Isso é exemplificado na Figura 2.4.

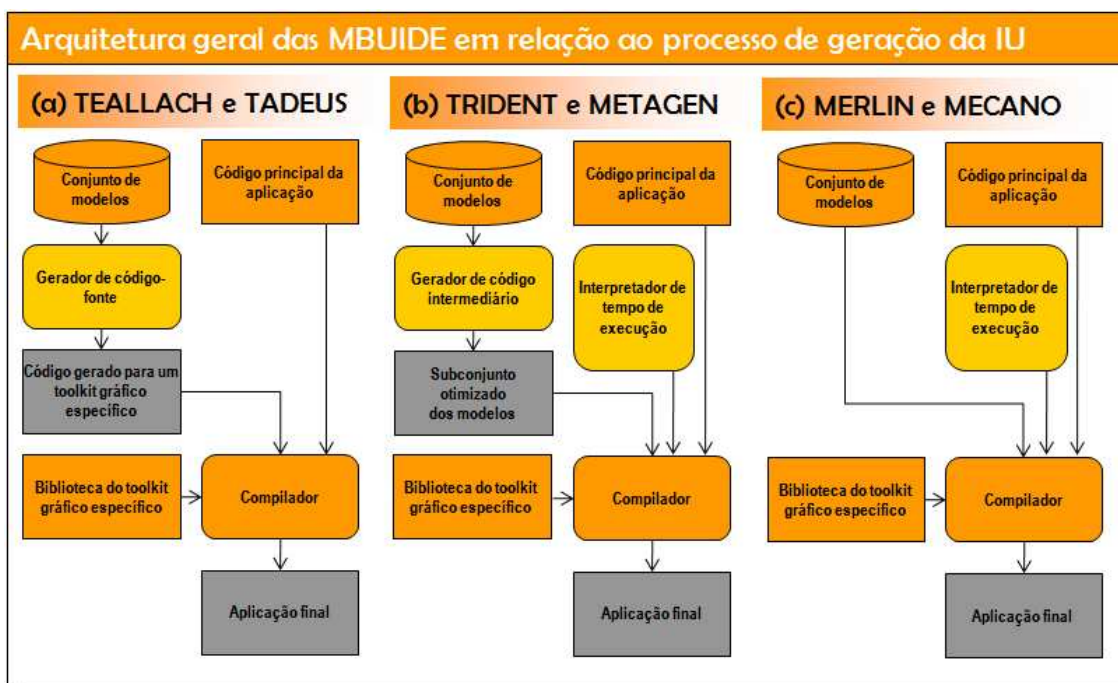


Figura 2.4: Arquitetura geral das MBUIDE.

A Figura 2.4 mostra a arquitetura geral das MBUIDE em relação ao processo de geração das interfaces de usuário e, conforme pode ser visualizado, independente da forma de geração, o Compilador, o Conjunto de Modelos, o Código Principal da Aplicação e a Biblioteca do Toolkit Gráfico Específico estão sempre presentes. O que varia nas arquiteturas é a forma com que as IU são produzidas.

Ferramentas como a TEALLACH e a TADEUS (Figura 2.4a) possuem um módulo Gerador de Código-Fonte, o qual obtém informações dos modelos da aplicação e produz como saída o código-fonte das IU com base em uma biblioteca gráfica específica. O Compilador encarrega-se de ligar todos os elementos necessários e produz como resultado a Aplicação Final. Nessa abordagem, a Aplicação Final é totalmente independente dos modelos da MBUIDE. Entretanto, ferramentas desse tipo têm pouca

flexibilidade à mudanças, uma vez que quaisquer alterações nos modelos implica em um recompilação total da Aplicação Final.

Ferramentas como a TRIDENT e a METAGEN (Figura 2.4b) possuem um módulo Gerador de Código Intermediário, que extrai do Conjunto de Modelos somente as informações necessárias para a uma determinada aplicação. O objetivo desse código intermediário é permitir uma certa independência e flexibilidade em relação ao formato utilizado pelos modelos originais. Entretanto, para que isso seja possível, é necessário a existência de um módulo Interpretador de Tempo de Execução, o qual deve ser compilado junto à Aplicação Final. Softwares que utilizam essa abordagem possuem uma flexibilidade maior do que aqueles do primeiro grupo, uma vez que geralmente é permitido alterações no conteúdo do código intermediário sem a necessidade de uma recompilação. Entretanto, devido a redundância de informações entre o modelos originais e o formato intermediário, complexidades adicionais estão presentes quanto à gerência de sincronia entre esses elementos.

O terceiro grupo (Figura 2.4c), que é representado pelas ferramentas MERLIN e MECANO abdica totalmente da geração de código-fonte. Nelas, o Interpretador de Tempo de Execução é responsável por traduzir informações diretamente dos modelos originais, produzindo como resultado as IU da Aplicação Final com base no *toolkit* gráfico escolhido. Softwares desse grupo possuem a mais alta flexibilidade, permitindo a alteração total da Aplicação Final mesmo com esta em execução. Entretanto, devido ser um processo interpretado, elas dependem completamente da presença dos modelos durante a execução do sistema e tendem a ter um desempenho mais baixo em alguns casos.

#### 2.2.4 Processo de Desenvolvimento

O processo de desenvolvimento nos MBUIDE normalmente é iniciado através da construção dos modelos do sistema, com ênfase nos mais abstratos, como os Modelos de Tarefa e de Domínio (PUERTA e EISENSTEIN, 1999). Independente da abordagem, que pode ser amparada por editores gráficos, textuais ou por assistentes de código (PINHEIRO, 2001), informações de cada um desses modelos são gradualmente refinadas até se obter um nível básico necessário. Ferramentas como TRIDENT (BODART, LEHEUREUX e VANDERDONCKT, 1993 e 1994) e TEALLACH (GRAY et al., 1998) (COOPER et al., 1997) balizam seu processo pela modelagem de tarefas, ao passo que ferramentas como MERLIN (MOREIRA, MRACK e PIMENTA, 2005) e MECANO (PUERTA et al., 1996) são norteadas pelo Modelo de Domínio.

Independente da abordagem utilizada, atingido o patamar mínimo de detalhamento, os mecanismos de geração dos MBUIDE podem efetuar a geração das versões preliminares das IU. Esse nível vai depender de cada ferramenta, e a abstração dessas informações vai estar relacionada à forma utilizada pelos elementos dos modelos, tais como o uso de AIO ou CIO. É nesse ponto que as ferramentas começam a se dividir, encaixando-se em um dos grupos citados na seção 2.2.3.

Após a primeira geração, as ferramentas podem entrar em vários ciclos evolutivos, nos quais ocorrem sucessivos refinamentos sobre os modelos. É justamente nesse ponto

em que começam os problemas relacionados ao gerenciamento do processo de desenvolvimento. Isso acontece porque, dependendo das capacidades do MBUIDE, alterações manuais no código-fonte produzido pela ferramenta acabam sendo perdidas quando ocorre um novo processo de geração.

Ferramentas com forte ênfase na geração de código-fonte devem preocupar-se em produzir estruturas de código padronizadas e aptas a serem alteradas manualmente pelos programadores. Por outro lado, ferramentas com foco na geração em tempo de execução, devem preocupar-se em deixar pontos de extensão para ligações com código programado manualmente, sob pena de tornarem-se limitadas em relação à adição e integração de recursos.

Embora a etapa de refinamento seja um *continuum* (PINHEIRO, 2001), quando alcançada a estabilidade de desenvolvimento, a tendência é que as atividades sobre a MBUIDE sejam gradativamente minimizadas. Conquanto que nesse momento, o foco do programador esteja direcionado aos ajustes finais da aplicação, outra característica ganha evidência nas MBUIDE: a capacidade de reuso.

Uma vez que as MBUIDE são compostas por diversos modelos, cada qual com informações úteis para determinado aspecto de um sistema, a possibilidade de reutilizar suas informações através múltiplos desenvolvimentos eleva consideravelmente os benefícios desse tipo de software.

De forma geral, algumas ferramentas MBUIDE permitem que informações de seus modelos sejam reutilizadas ao longo do tempo, algumas, inclusive, entre sistemas diferentes. Observa-se, porém, que para esse tipo de funcionalidade, um trabalho extra de projeto deve ser realizado, como a criação de *templates* de código propositalmente genéricos (como na ferramenta HUMANOID), a generalização de regras de aparência e comportamento (como nas ferramentas ADEPT, TRIDENT e TADEUS), as quais podem frequentemente entrar em conflito ou na correta estruturação de modelos compartilhados (no caso da ferramenta MECANO).

Com base em alguns trabalhos (PINHEIRO, 2001) (PUERTA e EISENSTEIN, 1999) (PUERTA et al., 1996) (SCHLUNGBAUN, 1997) se observa que MBUIDE baseadas em elementos concretos (CIO) e com maior apelo textual na formação dos modelos são mais flexíveis e orientadas para desenvolvedores experientes. Entretanto, elas tendem a ter uma curva de aprendizado maior

Por outro lado, as ferramentas focadas em modelos abstratos (AIO) e dirigidas por assistentes visuais são propícias para equipes de menor experiência, trazendo, contudo, menores recursos quanto a ajustes finos e integrações de mais baixo nível.

Ainda, e de forma quase geral, MBUIDE com maiores capacidades de automação têm seu foco de atuação limitado a certos tipos de IU (como as CRUD), visto que muitas decisões de geração são baseadas em heurísticas derivadas do tipo de interfaces às quais elas se propõem atuar.

### 2.3 Projetos e Características

Nessa Dissertação de Mestrado, 12 características consideradas relevantes para o projeto MERLIN foram comparadas frente às 15 ferramentas estudadas.

Objetivando evidenciar essas características, uma estrutura tabular está disponível no APÊNDICE A – TABELA COMPARATIVA DAS MBUIDE ANALISADAS, neste mesmo documento e pode ser consultada para maiores detalhes caso haja necessidade no aprofundamento dessas informações.

### 2.4 Discussão

Considerando os trabalhos analisados e as soluções oferecidas, verifica-se que um grande avanço ocorreu desde os primeiros UIMS. As soluções de geração baseada em modelos cresceram, e suas capacidades de geração e abrangência aumentaram significativamente. Através de seus variados modelos, as mais importantes facetas de um sistema podem ser documentadas, estruturadas e validadas em níveis mais abstratos, o que é positivo para os desenvolvedores em geral. Utilizando algoritmos internos configuráveis em conjunto com diversas regras de transformação, as capacidades de geração dos MBUIDE se estendem muito além da geração de simples IU, alcançando, muitas vezes, grandes porções de um sistema completo.

Porém, mesmo com esse progresso, observa-se que as soluções MBUIDE parecem ter alcançado um ápice em meados da década de 1990, e que atualmente as equipes de desenvolvimento acabam optando por ferramentas mais simplistas, como editores WYSIWYG, utilitários de geração baseados em *templates* ou mesmo produtos caseiros (PUERTA e EISENSTEIN, 1999) (MOREIRA e MRACK, 2003).

Causas para este distanciamento do cenário profissional são bem comentadas em vários trabalhos (PUERTA et al., 1999) (GRAY et al., 1998) (GRIFFITHS et al., 1997) (PINHEIRO, 2001) e tangem, principalmente, a fatores como a falta de reuso de padrões de mercado, a excessiva quantidade de configurações sobre muitos e diferentes modelos, a exigência de processos de desenvolvimento exclusivos e limitações frente a um processo evolucionário de desenvolvimento. De fato, como exposto por alguns autores (SCHLUNGBAUM, 1997) (MYERS, 1994), os MBUIDE existentes parecem servir mais como protótipos para testes de novas tecnologias, ou como experimentos de nível ainda acadêmico.

Buscando a minimização dos esforços para a construção e configuração dos modelos, o reuso efetivo de padrões de mercado e a transparência de uso frente aos processos de desenvolvimento existentes, esta Dissertação de Mestrado apresenta a proposta oferecida pelo software MERLIN, o qual é descrito a seguir.

## 3 O PROJETO MERLIN

*Este capítulo apresenta o projeto MERLIN. Uma breve descrição de seu histórico é efetuada e logo após os pontos-chave de sua motivação são elencados. O capítulo segue mostrando as características mais salientes na ferramenta e também a sua arquitetura de funcionamento. Ao final do capítulo é mostrado como funciona o processo de desenvolvimento de IU utilizando o MERLIN.*

### 3.1 METAGEN, o início

No ano de 2001, alguns integrantes do Setor de Informática da Universidade de Santa Cruz do Sul (UNISC) estavam empolgados com as possibilidades que surgiam a partir de um projeto interno denominado METAGEN (MOREIRA e MRACK, 2003). Utilizando uma abordagem bastante simplista, o pequeno protótipo desenvolvido em Visual Basic – que posteriormente fora portado para Delphi/Kylix – era um gerador baseado em modelos capaz de automatizar completamente a construção de IU para sistemas de banco de dados de plataforma cliente-servidor.

A partir de um esquema de banco de dados relacional qualquer enriquecido com informações textuais especificamente projetadas para a ferramenta, algoritmos especializados renderizavam em tempo de execução as IU do sistema. Essa abordagem, embora rudimentar, era extremamente rápida e eficaz, possibilitando a produção de interfaces funcionais tão logo um esquema de banco de dados estivesse disponível. Conforme testes realizados na época, aproximadamente 70% das IU de cadastro de um sistema podiam ser integralmente automatizadas com o uso da solução. Em relação ao tempo de desenvolvimento, os ganhos oscilavam em 40%, uma boa margem para um projeto de segundo plano dentro de um setor lateral em uma Universidade privada (MRACK e MOREIRA, 2003).

A título de exemplificação, a Figura 3.1 mostra uma típica interface CRUD gerada pelo Software METAGEN no ano de 2002.

**Cadastro de cliente**

Código: 3

Nome: João da Silva Pereira

Doc. Identificação: 85461815541 SSP RS I Identidade

Data Nasc.: 8\_/12/1958

Categoria: 3 Industria

UF Naturalidade: RS Rio Grande do Sul

Naturalidade: 607 PORTO ALEGRE

Sexo: M Masculino

**Endereço Residencial**

Logradouro Res.: Av. Medeiros 214 apto 21

UF Res.: RS Rio Grande do Sul

Localidade Res.: 692 SANTA CRUZ DO SUL

Cep Res.: 96810-250

Novo, Alterar, Cancelar, Excluir, Ajuda, Sair

Figura 3.1: Uma interface CRUD simples gerada pelo software METAGEN (MOREIRA e MRACK, 2003).

Na Figura 3.1, observa-se uma estruturação essencialmente tabular dos controles na IU. Os controles de dados são posicionados ao centro em uma área de deslize (*scroll*) horizontal e vertical. À direita, uma barra vertical de botões, os quais são ligados a regras de negócio pré-definidas. Todas as IU geradas pelo software METAGEN tinham aparência semelhante e não podiam ser flexibilizadas muito além disso.

O software METAGEN, embora tenha apresentado um bom conjunto de resultados, acabou sendo descontinuado. Isso ocorreu principalmente devido a falta de afinidade das atividades do projeto em relação às demandas primárias da equipe e também pelos direcionamentos impostos pela gerência sênior da instituição onde o projeto fora desenvolvido.

Porém, certos que as idéias surgidas nesse ínterim eram válidas, parte da equipe decidiu continuar o desenvolvimento do projeto através de uma iniciativa independente, agora conhecida pelo codinome MERLIN (MOREIRA, MRACK e PIMENTA, 2006).

### 3.2 MERLIN, a evolução

O MERLIN<sup>2</sup>, é uma MBUIDE com ênfase na geração de interfaces CRUD, também conhecidas no jargão dos programadores como *telas de cadastro*.

Essa prerrogativa segue com as bases do projeto METAGEN, as quais incluem (i) a geração das telas em tempo de execução; (ii) a configuração norteada pela edição

<sup>2</sup> Nome dado em alusão ao lendário mágico da Idade Média, cuja mitologia atribui a criação do círculo de pedra Stonehenge, na Inglaterra.

textual assistida dos modelos; (iii) uma base construída em padrões e linguagens de mercado; (iv) o reuso transparente e gerenciado de configurações e (v) a modelagem centrada no Modelo de Domínio da aplicação.

Tal posicionamento é baseado em um histórico profissional ao longo de vários anos de construção de software para banco de dados, o qual evidenciou quatro pontos-chave:

- a grande maioria dos programadores não está preparada para utilizar ferramentas complexas ou baseadas em linguagens abstratas, como as propostas pelas MBUIDE existentes;
- as empresas não estão dispostas a pagar os custos para a adoção de uma ferramenta complexa em seu ambiente e tão pouco interessadas em mudar seu processo de desenvolvimento em virtude dessa prática;
- qualquer tipo de geração de código-fonte, por mais parametrizada que seja, acaba produzindo um legado de difícil manutenção e com custos de refatoração cada vez maiores conforme a evolução do sistema;
- independente do tipo de sistema desenvolvido, o Modelo de Domínio está sempre presente e, devido sua íntima relação com as interfaces CRUD do sistema, ele pode ser considerado o elemento principal em qualquer processo de desenvolvimento, seja este automatizado ou não.

Este trabalho e, em específico esse capítulo, visa detalhar os princípios do projeto MERLIN e mostrar como ele pode ser utilizado para produzir interfaces de usuário funcionais e robustas, as quais podem ser utilizadas em sistemas de nível profissional.

As próximas seções descrevem as principais características da solução e mostram a sua arquitetura geral enquadrada dentro de um processo de desenvolvimento genérico.

### **3.3 Características da solução**

Nesta seção, as principais características da ferramenta MERLIN são detalhadas. Cada subseção aborda uma funcionalidade, justificando sua forma de implementação e, sempre que possível, demonstrando-a através de exemplos didáticos simples.

#### **3.3.1 Interfaces CRUD em Evidência**

Durante o desenvolvimento de aplicativos para banco de dados, é saliente a presença das interfaces de edição e visualização de informações que são provenientes do SGBD, as quais são simplesmente conhecidas como interfaces CRUD.

Entretanto, embora sejam simplistas, essas interfaces tendem a consumir valiosos esforços durante a construção em um sistema. De fato, uma pesquisa de 1992 sobre 69 sistemas diferentes (MYERS apud PIZANO, SHIROTA e IIZAWA, 1993) mostrou que aproximadamente 50% do tempo dedicado à programação de uma aplicação são dedicados à interface do usuário. Embora esta pesquisa não detalhes os números em relação às interfaces CRUD, uma estimativa mais singela, realizada sobre uma base de



15 sistemas de uso corporativo (MRACK, 2007), mostrou que, em média, 30% do código de uma aplicação para banco de dados é dedicado às telas de cadastro. Traduzindo isso em um gráfico, temos algo como mostrado pela Figura 3.2:

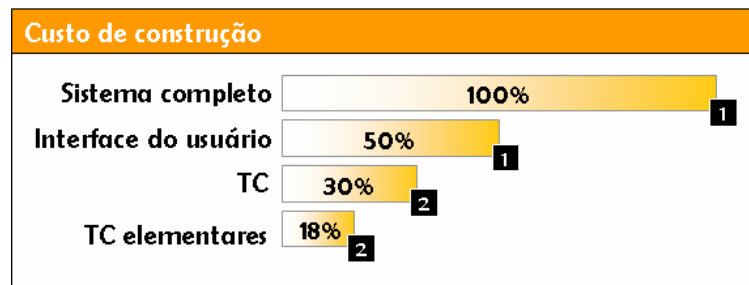


Figura 3.2: Custo da Interface de Usuário e das Telas de Cadastro (TC) em um sistema de banco de dados qualquer (MRACK, 2007).

Observa-se que 18% do sistema são constituídos de interfaces CRUD elementares. Nelas, todo o conteúdo e comportamento é ditado pelos objetos de dados relacionados à IU, não existindo nenhuma funcionalidade extra, seja em relação a um *layout* diferenciado ou à presença de regras de negócio vinculadas aos elementos de tela.

Fundamentada nesses valores, a proposta da ferramenta MERLIN é automatizar completamente a geração das IU de cadastro elementares (até 18% do sistema) sem nenhum custo de configuração. Para abranger a totalidade das telas de cadastro (até 30% da aplicação), informações de configuração devem ser acrescentados e eventualmente, novos algoritmos auxiliares construídos e anexados ao gerador. O alcance dos 50% do sistema (ou seja, todas as IU), embora não descartado, não faz parte do escopo inicial da ferramenta.

### 3.3.2 Modelos Utilizados

As pesquisas realizadas para formação desse trabalho demonstram que o Modelo de Domínio pode ser considerado o denominador comum para uma MBUIDE. Baseado nesse fato, e considerando a forte relação deste com as interfaces CRUD relacionadas (GOHIL, 1999) (MOREIRA e MRACK, 2003), esta proposta elege o Modelo de Domínio como o ponto de partida para a geração das IU.

Entretanto, conforme destacam vários trabalhos (BODART, LEHEUREUX e VANDERDONCKT, 1994) (PUERTA e EISENSTEIN, 1999) (SCHLUNGBAUM, 1997), o Modelo de Domínio não é capaz de expressar todas as características de uma interface de usuário. Assim, para obter uma completude de geração, outros modelos também são suportados. Abaixo, um maior detalhamento de como esses modelos são utilizados na solução:

- **Modelo de Domínio:** É o modelo principal e essencial da ferramenta e pode ser visto como um Modelo de Classes da UML. Na prática, é implementado por uma classe ou interface Java normal, que não precisa estender ou implementar nenhuma outra classe ou interface padrão. É um modelo Orientado a Objetos por natureza e, seguindo as premissas defendidas pelas

modernas escolas de modelagem (AVRAM e MARINESCU, 2004) (EVANS, 2003), deveria conter somente declarações de atributos, sejam eles tipos primitivos ou relacionamentos de qualquer cardinalidade para as outras classes do sistema. O objetivo desse modelo é prover as informações necessárias para a estruturação básica da IU.

- **Modelo de Apresentação:** Implementado através do uso de *Java Annotations*, ou simplesmente Anotações (JCP, 2004a), é fortemente relacionado ao Modelo de Domínio. Objetivando refinar as características de aparência da IU, ele pode ser definido em termos de AIO ou CIO. Se definido em termos de AIO, é dos algoritmos de renderização atrelados ao gerador a incumbência de reproduzir o comportamento e aparência da IU; se definido em termos de CIO, suas informações são interpretadas diretamente pelo toolkit gráfico utilizado no momento da renderização da IU. Diferente de outras MBUIDE, o esquema de mapeamento entre AIO e CIO é bidirecional, permitindo o reuso de informações mesmo entre toolkits gráficos diferentes.
- **Modelo de Tarefas:** Também descrito através de Anotações, esse modelo é relacionado ao Modelo de Domínio e têm suas as tarefas descritas em termos de AIO ou CIO. Na prática, esse modelo é bastante simples, sendo que as tarefas nele definidas nada mais são do que métodos de classe, os quais podem estar definidos em qualquer outra parte do sistema, ou mesmo fora dele. Sobre as tarefas desse modelo podem existir pré-condições, as quais podem ser definidas através de expressões Object Constraint Language (OCL) (KLEPPE e WARMER, 1999), Expression Language (EL) (JCP, 2006c), ou código nativo computável a partir do contexto de execução da IU, utilizando para isso linguagens de *script* como a BeanShell (JCP, 2005b) e a Groovy (JCP, 2004d). Observa-se que, embora menos poderosa que a OCL, a EL é a opção preferível para a criação das pré e pós-condições, uma vez que ela já faz parte da linguagem Java e, portanto, está mais próxima dos programadores em geral.
- **Modelo de Diálogo:** É formado por Anotações também, e objetiva fazer a ligação entre o Modelo de Apresentação e o Modelo de Tarefas. Na prática, nesse modelo se definem as relações entre os elementos que constituem a IU, os eventos que podem ocorrer e as regras de negócio a serem executadas. Seguindo a premissa do modelo Evento-Condição-Ação (ECA) (OLIVEIRA apud KLINGER e KROTH, 2001), uma implementação própria do conceito de Agentes (A4J, 2007) definido pela linguagem Eiffel (MEYER, 2000) permite grande flexibilidade na formação desse modelo, capacitando que a ligação entre as suas partes ocorra sem dependências sintáticas.
- **Modelo de Usuário:** Nessa proposta, o Modelo de Usuário nada mais é do que uma derivação do Modelo de Domínio. Em outras palavras, todas as vezes que uma IU individualizada é necessária (SCHLUNGBAUM, 1997), um novo Modelo de Domínio é criado a partir do Modelo de Domínio existente. Ao se ajustar esse novo modelo, uma IU individualizada é obtida.

Para suportar essa funcionalidade, 4 abordagens podem ser utilizadas:

- Criação de uma nova classe, herdando da classe-base do Modelo de Domínio. Essa abordagem pode ser utilizada quando a classe-base não for uma classe final (ou classe-folha).
- Criação de uma interface e sua consequente implementação pela classe-base. O inconveniente nesse caso é a necessidade de alteração da classe-base, o que pode não ser viável em alguns casos (uma classe-base que não tenha disponível o seu código-fonte ou devido a problemas de licenciamento, por exemplo).
- Criação de uma nova classe-base para a IU e a aplicação de padrões estruturais como o Adapter e Delegate (GAMMA et al., 1995) (MARINESCU, 2002). Essa abordagem é a que permite menor acoplabilidade entre as partes e a recomendada pela ferramenta;
- Utilização de anotações (JCP, 2004a) em conjunto com o suporte de tipos genéricos da linguagem Java (JCP, 2004b) para interligação da classe-base com a respectiva classe derivada.

### 3.3.3 Reuso de Padrões de Mercado

Salvo a ferramenta AME, que é baseada no já ultrapassado padrão OMT (RUMBAUGH et al., 1990), todas as outras MBUIDE utilizam formatos proprietários para a definição de seus modelos. Embora algumas soluções busquem uma aproximação de padrões como a CORBA IDL, o fato é que nenhuma delas consegue descrever a totalidade de seus artefatos a partir de especificações padronizadas pelo mercado.

Tendo como diferencial o uso da linguagem Java, o MERLIN obtém vantagens importantes, as quais facilitam de sobremaneira o assentamento em padrões já em uso por essa plataforma. A Tabela 3.1 descreve os principais padrões utilizados pela ferramenta:

Tabela 3.1: Padrões utilizados pela ferramenta MERLIN.

Padrão	Aplicação
Java (GOSLING, et al., 2005)	Linguagem de programação, tanto para o sistema final quanto para extensão das funcionalidades do <i>framework</i> .
JSR 14 (JCP, 2004a) e JSR 175 (JCP, 2004b)	Configuração e interligação de todos os modelos da ferramenta.
JSR 227 (JCP, 2003), JSR 273 (JCP, 2005a) e JSR 295 (JCP, 2006a), JSR 299 (JCP, 2008c)	Interligação dos modelos de Domínio e Apresentação.
Java Swing	<i>Toolkit</i> gráfico para construção dos elementos de tela que compõem o Modelo de Apresentação e Diálogo. Observa-se que quaisquer outros <i>toolkits</i> gráficos baseados em Java são suportados a partir da implementação de algoritmos adicionais. Previsões incluem o suporte para SWT (The Eclipse Foundation, 2007a), GWT (Google Inc., 2007), JSF (JCP, 2006d) e Thinlet (Thinlet, 2007).
JSR 303 (JCP, 2006b)	Validação de informações contidas no Modelo de Domínio e Apresentação.
JSR 245 (JCP, 2006c), OCL (OMG, 2007b), JSR 274 (JCP, 2005b) e JSR 241 (JCP, 2004d)	Definição das expressões de pré-condições sobre o Modelo de Tarefas.

### 3.3.4 Geração em Tempo de Execução

Seguindo as premissas defendidas pelo projeto METAGEN, o MERLIN também é uma MBUIDE que produz as IU da aplicação em tempo de execução, através de um processo de interpretação das informações contidas em seus modelos.

Em suma, o processo utilizado pela ferramenta é receber uma instância da classe-base da IU e inspecioná-la através do uso de reflexão (SUN MICROSYSTEMS, INC., 2007), uma técnica em Java que permite a introspecção de objetos em memória.

A partir do objeto recebido como parâmetro, são extraídas e analisadas informações tanto sobre a estrutura da classe de origem, quanto das anotações de configuração atreladas. De posse dessas informações, os algoritmos de geração produzem a IU, podendo, também, se encarregar do preenchimento dos controles da tela a partir dos valores existentes no objeto passado. Tal processo é exemplificado na Figura 3.3:

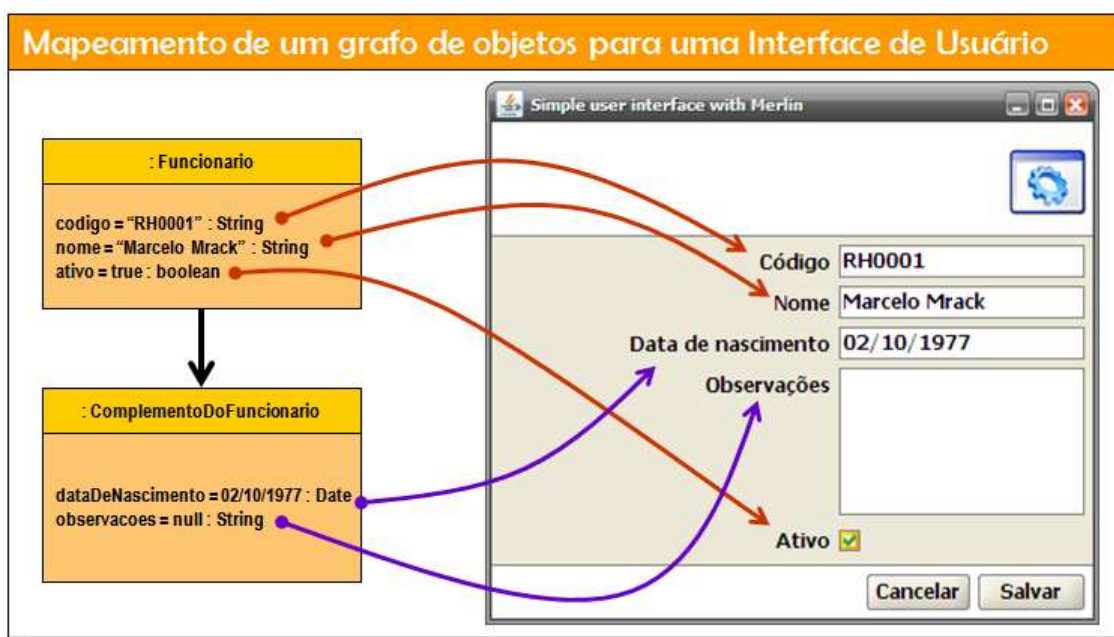


Figura 3.3: Transformação do grafo de objetos (Modelo de Domínio) na respectiva IU.

Na Figura 3.3, uma instância da classe `Funcionario` é passada como parâmetro para o software MERLIN. Utilizando o recurso da introspecção, essa instância é varrida e as informações sobre sua estrutura (atributos, relacionamentos, tipos de dados, métodos, anotações atreladas e outras) são coletadas.

Heurísticas, implementadas na forma de blocos de código plugáveis à ferramenta, como “dados do tipo `String` são mapeados para campos de texto de tamanho médio” ou “dados do tipo `String` e de tamanho longo devem ser mapeados para campos de texto expansíveis” são aplicadas pelo software, produzindo a respectiva IU.

Quando o objeto base para a IU é um grafo, este é percorrido na íntegra ou conforme a necessidade, o que é definido pelo conjunto de anotações atreladas ao objeto.

Montada a IU, a ferramenta também pode efetuar o preenchimento dos campos na tela, conforme regras de mapeamento de interface integradas ao software ou definidas pelo programador.

Com relação aos tempos consumidos nesse processo, característica que pode ser crucial para o sucesso desse tipo de ferramenta, os testes na versão atual do MERLIN indicam um desempenho na ordem de centésimos de segundo para cada IU. Também, uma vez que Java é uma linguagem com suporte *multithreading*, o uso dessa funcionalidade em conjunto com técnicas de *caching* permite níveis de desempenho compatíveis à IU construídas manualmente.

### 3.3.5 Modelos Auto-Contidos

Considerando a arquitetura geral das ferramentas MBUIDE (PINHEIRO, 2001), se observa que para ser viável um processo de geração em tempo de execução, informações provenientes dos modelos devem ser acessíveis pelo mecanismo de geração durante o uso do sistema final. Para suportar isso, ou as MBUIDE exigem a presença dos modelos originais junto ao sistema final, ou incluem um passo adicional para a criação de um formato intermediário de maior desempenho (ou de menor tamanho), tal como demonstrado na Figura 2.4b.

Uma interessante característica das *Java Annotations* (JCP, 2004a) é a capacidade delas serem compiladas junto às classes da aplicação. Essa particularidade promove uma significativa vantagem para o MERLIN, que é a formação de um modelo auto-contido onde todas as informações necessárias para o mecanismo de geração estão disponíveis sempre e em um formato nativo da linguagem. Em outras palavras, os modelos *fazem parte* da aplicação final.

### 3.3.6 Interligação dos Elementos

De forma geral, as MBUIDE existentes não possuem um tratamento robusto em relação aos eventos produzidos na interface do usuário (SCHLUNGBAUM, 1997). Em suma, o suporte à essa funcionalidade se limita à atribuição de funções simplistas aos objetos da IU, tais como botões (BODART, LEHEUREUX e VANDERDONCKT, 1994) ou à ativação de comportamentos pré-definidos para determinados tipos de controles de tela (GRAY et al., 1998).

Considerando que as modernas IU são aderentes aos padrões *Model-View-Controller* (MVC) (REENSKAUG, 1979) ou *Presentation-Abstract-Control* (PAC) (COUTAZ, 1987) e que toda a geração das IU é baseada no Modelo de Domínio, a ferramenta MERLIN propõe utilizar o Modelo de Domínio como ponto central para a ligação dos controles de tela produzidos, seus tratadores de eventos, regras de negócio e os dados relacionados.

Para cumprir esse objetivo, o conceito de *Agentes* definido na linguagem Eiffel

(MEYER, 1991) foi portado para a linguagem Java através de uma implementação especificamente desenvolvida para o MERLIN. É o subprojeto Agent4Java (A4J, 2007). Para exemplificar esse conceito de interligação de elementos através de Agentes, considere a Figura 3.4:

Interligação de eventos e regras de negócio para controles de IU	
(a) Classe do Modelo de Domínio	(b) Classe com regras de negócio
<pre> 1 public class Funcionario { 2   @Agent( 3     event = { "lostFocus" }, 4     action = { "fillEmail" } 5   ) 6   String nome; 7   String email; 8 } </pre>	<pre> 1 public class RegrasDeNegocio { 2   @In Context ctx; 3 4   void fillEmail() { 5     ctx.eval( 6     "\$txtEmail.text = \$txtNome.text.trim + 7     '@3layer.com.br'"); 8   } 9 } </pre>

Figura 3.4: Interligação de eventos e regras de negócio para controles de IU utilizando Agentes na forma de anotações sobre o Modelo de Domínio.

A interpretação da Figura 3.4 deve ser feita em três etapas:

1. Primeiro, na Figura 3.4a, o Modelo de Domínio é definido por uma classe `Funcionario` composta por dois atributos do tipo `String`. Esses atributos quando renderizados na IU, serão traduzidos pelo mecanismo de geração como campos de texto de tamanho fixo.
2. Segundo, sobre o atributo `nome`, um Agente é declarado através de uma anotação `@Agent` (linha 2). Este agente especifica que, quando o atributo `nome` for renderizado na IU, no seu evento `lostFocus` associado, a ação (ou regra de negócio) `fillEmail` deve ser invocada.
3. Já na Figura 3.4b, uma outra classe `RegrasDeNegocio` é declarada. Essa classe pode residir tanto dentro da aplicação em desenvolvimento, quanto em um meio externo, tal como um *webservice* (W3C, 2002). Nessa classe está a definição do método de negócio `fillEmail`, o qual encarrega-se de, dado um nome de funcionário, preencher o seu email com a preocupação de remover os espaços em branco do nome original (método `trim()`, na linha 6). Observa-se que esse trecho de código utiliza dois recursos importantes. O primeiro é o uso da injeção de recursos sobre o atributo `ctx`, o qual representa o contexto da execução de chamada do evento `lostFocus`. A injeção de recursos efetuada pela ferramenta permite que o atributo `ctx` tenha acesso a todos os valores existentes na IU do funcionário. Essa injeção de recursos é realizada com o uso da anotação `@In`, que é reusada do

*framework* Web Beans (JCP, 2008c). Em segundo lugar, o código de `fillEmail` usa os recursos da linguagem de *script* Groovy, a qual é integrada ao Java na sua versão 6. Como a expressão a ser avaliada é executada sobre o contexto `ctx`, a execução do método `fillEmail` acontece como se ele estivesse definido dentro da própria classe `Funcionario`.

A vantagem nessa abordagem é que em nenhum momento ocorre uma dependência sintática entre os elementos. A classe `Funcionario` não tem “conhecimento” da classe `RegrasDeNegocio` e vice-versa. Também, se observa que a ligação entre os elementos é neutra em relação ao pacote gráfico, ou seja, o método tratador de eventos `lostFocus` não precisa estender classes ou implementar interfaces do *toolkit* gráfico. Da mesma forma, o método de negócio `fillEmail` não se preocupa com o recebimento de parâmetros, uma vez que, graças à injeção de recursos sobre o atributo de contexto `ctx`, ele é executado como se estivesse dentro da própria classe `Funcionario` e, por isso, tem acesso a todas as informações contidas nessa classe.

Dessa flexibilidade, a consequência mais saliente é que a aplicação final pode ser colocada em operação mesmo que as regras de negócio envolvidas não estejam implementadas ainda. Isso é uma importante característica em projetos que utilizam ciclos evolutivos de desenvolvimento e também na disponibilização de versões preliminares (protótipos) do sistema.

### 3.3.7 Edição Textual Assistida dos Modelos

Tradicionalmente, em ambientes de desenvolvimento, três condições são verificadas:

- as atividades de criação de interfaces CRUD são delegadas a programadores iniciantes, os quais não tem maiores conhecimentos além do uso básico da sua IDE de trabalho e da linguagem de programação do sistema;
- os desenvolvedores experientes são mais produtivos e sentem-se mais à vontade utilizando bons editores textuais do que trabalhando com editores gráficos;
- a tendência das modernas IDE é incrementar seus editores textuais, tornando-os pró-ativos e suportando a Edição Textual Assistida (ETA) através de modelos de código configuráveis baseados em contexto de uso.

Com esses critérios em evidência, a ferramenta proposta defende que toda a edição de seus modelos seja realizada através da premissa ETA. A Figura 3.5 mostra essa abordagem na IDE Eclipse (The Eclipse Foundation, 2007b).

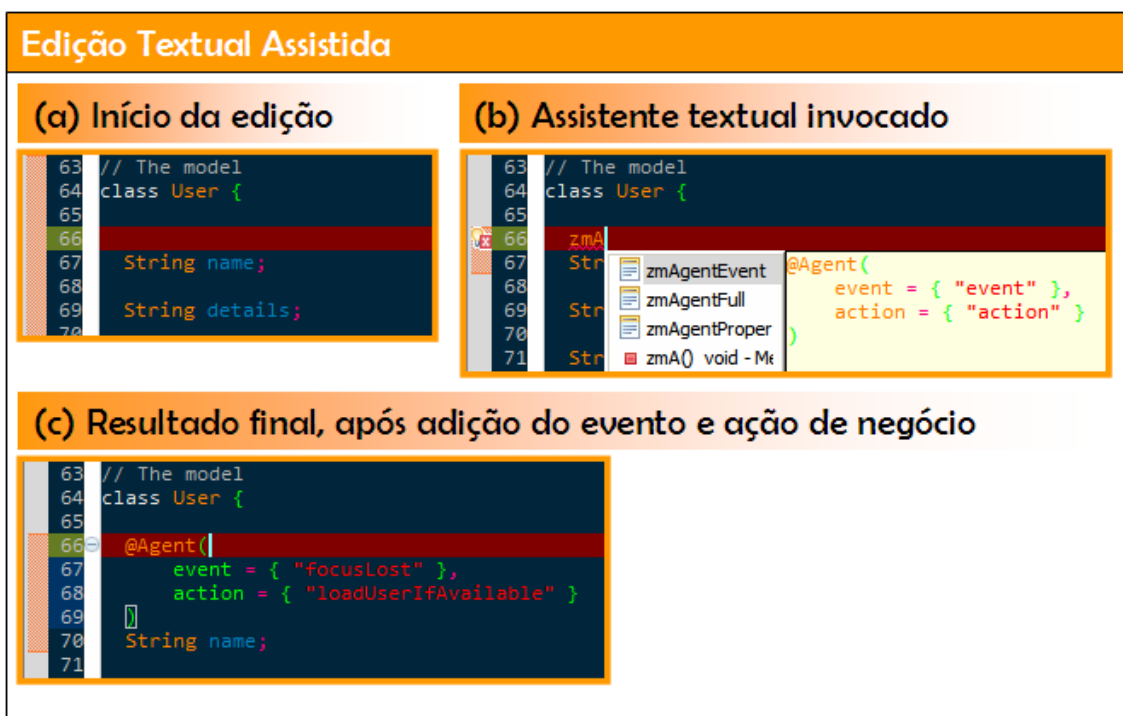


Figura 3.5: Uso do assistente textual para edição do Modelo de Domínio do MERLIN.

Na Figura 3.5 percebe-se três momentos. Em 3.5a, na linha 66, o programador está com o cursor posicionado sobre a propriedade `User.name` com a intenção de incluir um comportamento de IU através de um agente. Para tanto, em 3.5b ele utiliza o *template* de código configurável `zmAgentEvent`, o qual vai incluir a declaração do agente necessário. Observa-se que a própria IDE encarrega-se de mostrar uma lista de opções para o programador escolher. Em 3.5c está como resultado a declaração do Agente inserida pelo Assistente Textual, onde o programador teve como trabalho simplesmente definir os nomes do evento e da ação de negócio a serem executadas.

Também se observa que, quando uma configuração inválida é criada (como um evento `focusLost` ou uma ação `loadUserIfAvailable`), o Assistente Textual captura esse erro e sugere ações de correção, como mostra a Figura 3.6:



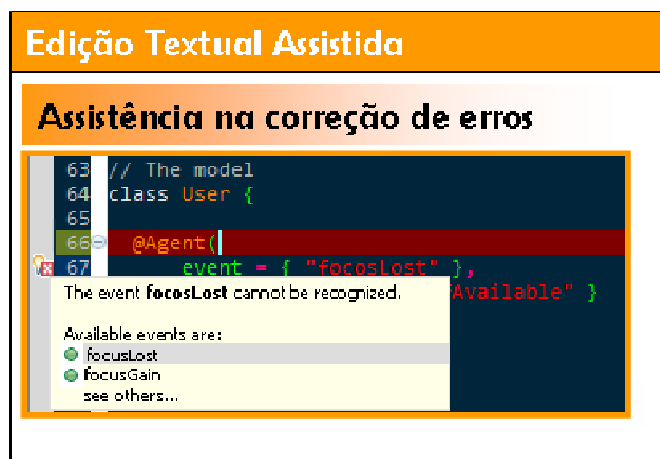


Figura 3.6: Suporte para a correção de erros durante a edição dos modelos da ferramenta MERLIN.

### 3.3.8 Tipos de Interface CRUD Suportadas

As Interfaces de Usuário para visualização e edição de dados, tipicamente presentes em aplicações baseadas em SGBD, podem ser divididas três grandes grupos (MRACK, 2005):

- **Interfaces CRUD *um-para-um*:** São as IU mais simples que existem. Nelas, o seu conteúdo é ditado por somente uma instância de um objeto de dados, sendo que, ou esse objeto contém somente propriedades de tipos primitivos (como String, booleanos, inteiros, etc.), ou associações para outros objetos onde a cardinalidade máxima igual a 1. A IU da Figura 3.3 é um exemplo típico desse caso.
- **Interfaces CRUD *um-para-muitos*:** Também conhecidas como IU Mestre-Detalhe, são interfaces de usuário compostas por um grafo de objetos onde o objeto raiz é o elemento-base da IU. A partir desse objeto principal (dito Mestre), diversas associações para outros objetos (ditos Detalhe) podem existir. Essas associações têm na origem (no mestre) uma cardinalidade igual a 1 e no destino (nos detalhes) cardinalidade máxima maior do que 1. Um exemplo clássico para esse tipo de IU é uma tela para emissão de notas fiscais, onde o objeto mestre é a nota fiscal e os objetos de detalhes são os itens que a constituem.
- **Interfaces CRUD *muitos-para-muitos*:** São as IU mais complexas que existem. Nessas interfaces, o objeto mestre também está relacionado a diversos objetos de detalhes. Porém, cada objeto de detalhe também pode estar associado a diversos objetos-mestre. Nesse sentido, o conceito de mestre e detalhe fica distorcido, uma vez que dependendo da perspectiva do usuário, ora um objeto pode ser o mestre, ora ele pode ser o detalhe. Um exemplo para esse tipo de IU é um cadastro de cursos, onde um aluno pode cursar diversas disciplinas, e cada disciplina pode ser cursada por diversos

alunos.

Sob a ótica da geração automatizada, as características dessas IU são influenciadas pelo conteúdo e capacidades oferecidas pelos Modelos de Dados, de Domínio, de Apresentação e de Diálogo. Considerando as MBUIDE analisadas e, embora não apresentando maiores detalhes, a ferramenta DRIVE (GRIFFITHS, 1997) é a que mais se aproxima das funcionalidades desejadas para suportar esses tipos de IU.

Neste trabalho, a versão inicial da ferramenta MERLIN se propõe a gerar automaticamente, e completamente, interfaces CRUD do tipo *um-para-um*, as quais fazem parte dos 18% apresentados na Figura 3.2. O suporte à geração de interfaces CRUD dos tipos *um-para-muitos* e *muitos-para-muitos* também são objetivos do trabalho, porém, devido à complexidade e nuances existentes (MRACK, 2005), os esforços para a sua implementação devem acontecer nas versões futuras do software.

### 3.3.9 Layout da IU

A capacidade de formatação do arranjo dos componentes que constituem a IU é uma das características mais salientes em uma ferramenta MBUIDE. Ao contrário das ferramentas WYSIWYG, onde a composição, o posicionamento e o tamanho dos elementos da IU são definidos através de operações manuais de arrastar e soltar, as MBUIDE geralmente utilizam algoritmos configuráveis para a realização dessa tarefa (SCHLUNGBAUM, 1997). Regras clássicas de usabilidade, como a “redução da perda de espaços na tela”, ou a “minimização dos movimentos oculares do usuário” [BODART, LEHEUREUX e VANDERDONCKT, 1994] formam as bases desses algoritmos.

Na ferramenta proposta, tais regras são aplicadas e a geração do *layout* pode ser realizada de três maneiras diferentes:

- **Uso de *Java Annotations*:** Os algoritmos de geração integrados à ferramenta podem ter seu comportamento modificado através do uso de anotações sobre os elementos do Modelo de Domínio. Tais anotações podem ser definidas tanto em termos de AIO como CIO.
- **Criação de novos algoritmos:** Quando os algoritmos disponíveis na ferramenta não forem suficientes para o cenário existente, novos algoritmos podem ser desenvolvidos e anexados ao gerador.
- **Uso de *templates*:** Na eventualidade de geração de uma IU complexa, situação em que as características de posicionamento e tamanho dos componentes de tela ficam fora dos limites alcançáveis pelas *Java Annotations* ou pelos algoritmos, é possível a criação manual de um *template*, o qual é utilizado como molde de substituição no momento da geração. Esse *template* é construído em termos de CIO e, por isso, é dependente do *toolkit* gráfico utilizado. Por exemplo, para *toolkits* gráficos como o Swing e SWT, os *templates* são classes Java simples; para *toolkits* como o Thinlet e XUL, os *templates* são arquivos XML e para *toolkits* como o GWT e o JSF, os *templates* são arquivos XHTML/CSS.

As Figuras 3.7 e 3.8 mostram, respectivamente, o uso de anotações e de *templates* para a redefinição de *layout* de tela:

**Redefinindo a ordem de controles e a posição de textos descritivos na IU**

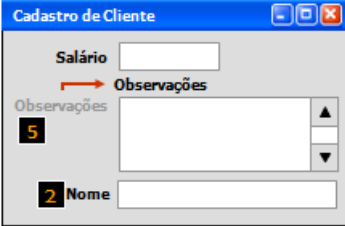
A classe de dados	A tela gerada
<pre> 1 class Cliente { 2   @Order(after="observacoes") 3   String nome; 4   float salario; 5   @Caption(pos=Caption.TOP_LEFT) 6   String observacoes; 7 } </pre>	

Figura 3.7: Redefinindo a ordem de controles e a posição de textos descritivos na IU.

Na Figura 3.7, a anotação `@Caption(pos=Caption.TOP_LEFT)` indica que o texto descritivo do campo `observacoes` deve ser posicionado acima e à esquerda deste controle quando renderizado na IU. A anotação `@Order(after="observacoes")` sobre o campo `nome`, indica que este deve ser posicionado depois das observações do cliente. Percebe-se que isso é uma informação abstrata, pois o conceito “depois” vai depender, na verdade, do tipo de fluxo de preenchimento adotado pelo do algoritmo de *layout* utilizado.

**Um exemplo de layout manual**

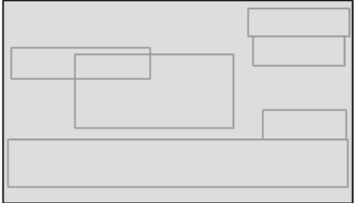
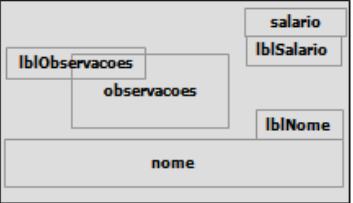
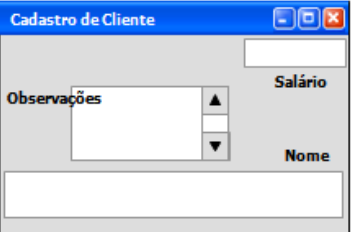
(a) O template inicial	(b) O template configurado	(c) A IU gerada
<p>AlgumaClasse extends JPanel</p> 	<p>AlgumaClasse extends JPanel</p> 	
<p><b>(d) A chamada ao gerador, informando o template de IU a ser utilizado</b></p> <pre> merlin.createIhc(Cliente.class, AlgumaClasse.class); </pre>		

Figura 3.8: Um exemplo de *layout* de IU gerado a partir de um *template* manual.

A Figura 3.8 mostra o processo esquematizado de criação e uso de uma classe de *template* para uma IU baseada no *toolkit* gráfico Swing.

Em um primeiro momento (3.8a) é criada uma classe que serve como *template*, tarefa que pode ser feita manualmente ou com o uso de um editor WYSIWYG qualquer. É nesse momento que se define a posição desejada dos controles na IU. O segundo passo (3.8b), é definir os nomes dos controles de acordo com o padrão de geração da ferramenta, característica que pode ser ajustada conforme a necessidade de cada equipe de desenvolvimento. O último passo é simples (3.8d), e consiste em utilizar a versão

sobrecarregada do método básico de geração `createIhc()`, informando, além da classe-base para a IU, o *template* a ser aplicado na renderização da tela. O resultado aparece na parte (3.8c) da figura.

De forma geral, anotações são utilizadas para efetuar ajustes simples de *layout*, como a alteração da ordem ou o alinhamento de controles na tela; a implementação de algoritmos é uma alternativa interessante quando diversas IU de mesmo formato devem ser produzidas e os algoritmos disponíveis não são adequados o suficiente; o uso de *templates* é a solução mais adequada para IU de características muito diferenciadas.

### 3.3.10 Realimentação

A utilização de informações históricas como forma de tornar um sistema pró-ativo é uma característica benévola para qualquer mecanismo de geração. Embora alguns trabalhos corroboram com esse ponto de vista [BODART, LEHEUREUX e VANDERDONCKT, 1994], as pesquisas realizadas evidenciam que o uso de realimentação nas ferramentas MBUIDE ainda é um cenário inexplorado. Com exceção à proposta oferecida no software MECANO (PUERTA et al., 1996), o qual promove o reuso através de uma base compartilhada para os seus modelos, nenhum dos outros trabalhos analisados aborda esse assunto de forma direta.

Amparado na experiência em cenários de desenvolvimento de sistemas, a qual evidencia que a aplicação de configurações semelhantes é um processo recorrente ao longo de diversos desenvolvimentos, essa proposta defende a prática de reuso baseado em histórico para a configuração dos seus modelos.

Entretanto, embora esse tópico seja de grande interesse, as pesquisas realizadas mostram que a implementação deste conceito é uma tarefa complexa. Nesse sentido, as considerações sobre esse item são idéias preliminares baseadas em alguns protótipos desenvolvidos e que, portanto, precisam ser validadas em trabalhos futuros.

Com o objetivo de apresentar percepções detectadas até o momento, a Tabela 3.2 é apresentada. Nela, cada situação de interesse é descrita e as propostas de soluções são comentadas.

Tabela 3.2: Necessidades e propostas correntes para o mecanismo de configuração realimentada do MERLIN.

Necessidade	Descrição	Propostas correntes
Modelos Auto-Contidos	Na ferramenta proposta, não existe nada além dos arquivos compilados do sistema para formação dos modelos. Isso implica que, para ser possível o reuso de informações entre sistemas distintos, os arquivos compilados que formam os modelos de um sistema S1 devem estar disponíveis para o novo sistema S2 e assim sucessivamente.	Para sistemas que são executados dentro de servidores de aplicação, como o JBossAS (JBoss, 2007b), é possível a utilização do mecanismo de carregamento de classes compartilhado ( <i>bootstrap classloader</i> ), como forma de localizar e reutilizar as classes de todos os sistemas disponíveis no ambiente de execução. Para sistemas executados de forma independente ( <i>standalone</i> ), é possível ou (i) a utilização de bibliotecas Java compartilhadas (JAR FILE SPECIFICATION, 2007) ou (ii) se valer de técnicas de engenharia de <i>bytecodes</i> (JAVASSIST, 2007a) para compilar as classes do sistema de forma que as informações de histórico necessárias sejam implicitamente inseridas nessas classes.
Informações necessárias	Se referindo ao reuso, é importante evidenciar quais informações podem ser empregada na sua implementação. Pela experiência, se percebe que algumas informações comportamentais existentes no Modelo de Tarefas (como as baseadas em regras de negócio) são suscetíveis ao contexto de execução e, por esse motivo, têm um menor grau de reusabilidade. Por outro lado, informações para a validação de campos, ou para definição de <i>layout</i> de tela são mais propícias ao reuso. Elencar quais informações podem ser reutilizadas é de importância vital para o sucesso no reuso de configurações.	De forma geral, as informações históricas são formadas pelas anotações atreladas ao Modelo de Domínio. Assim, a eleição de quais anotações devem ser utilizadas pelo mecanismo de histórico é o fator-chave para o sucesso da solução. Nesse momento, não existe uma definição clara sobre quais anotações, ou mesmo quais as propriedades de anotações devem ser consideradas para o mecanismo de reuso. Tão pouco se sabe como devem ser as estruturas internas dos algoritmos responsáveis por essas tarefas.
Transparência	Um dos requisitos desde o início do projeto é que todo o mecanismo de histórico de configurações seja transparente para o desenvolvedor. Em outras palavras, espera-se simplesmente que o mecanismo de geração compute os valores históricos e gere um resultado; se o programador não estiver satisfeito com o produto, então ele, utilizando a premissa da Configuração Por Exceção (CPE), ajusta o resultado via anotações sobre o Modelo de Domínio, fato que, ciclicamente, realimenta o sistema de geração.	A implementação desse requisito não é de todo difícil. Porém, é importante verificar o desempenho da computação de uma base histórica muito grande quando não utilizada a técnica de engenharia de <i>bytecodes</i> .
Gerenciamento	Utilizar informações de configuração oriundas de outros desenvolvimentos pode incorrer em uma situação interessante, que é a suscetibilidade à alterações ao longo da cadeia de dependências. Por exemplo, suponha-se que um sistema S3 tem configurações herdadas de um sistema S2, o qual, por sua vez, tem informações reusadas de um	Dependendo da abordagem utilizada, o problema de gerenciamento do histórico pode ser minimizado. Se utilizada a técnica de engenharia de <i>bytecodes</i> , uma versão do histórico pode ser compilada junto ao sistema. Nesse caso, alterações no sistemas formadores do histórico compilado não seriam percebidas pelo sistema

Necessidade	Descrição	Propostas correntes
	<p>sistema S1. Qual seria o impacto em S2 e S3 caso as configurações de S1 fossem alteradas? Uma política para o gerenciamento do histórico é outro fator de importância para o mecanismo de geração.</p>	<p>atual. Entretanto, também as evoluções (e correções) não seriam propagadas pela cadeia sem a recompilação do sistema dependente. Por outro lado, computando o histórico sob demanda, essas evoluções seriam instantâneas e, nesse caso, alguma técnica de Configuração Por Exceção poderia ser utilizada para indicar quais informações não devem ser afetadas pela mudança no histórico.</p>
<p>Pesos e Medidas</p>	<p>Um sistema de histórico é interessante quando a pró-atividade existente tem boas taxas de acerto. Assim, é importante que o mecanismo de geração consiga diferenciar e escolher quais informações devem ser reutilizadas e quais não devem. Não entrando em conceitos mais complexos, como técnicas de aprendizado de máquina ou Inteligência Artificial, este trabalho prevê uma simples política de pesos e medidas para resolver esse problema.</p>	<p>De forma simples, os algoritmos de geração totalizam as ocorrências das anotações e suas propriedades no histórico do sistema. Desses totais, médias são obtidas e os valores resultantes inferidos como a melhor escolha de geração. Novamente utilizando o conceito de Configuração Por Exceção, caso do desenvolvedor não esteja satisfeito com o resultado obtido, ele insere a anotação de configuração necessária para o ajuste sobre o respectivo elemento do modelo. Essa configuração entra para o sistema de histórico e, então, é reutilizada para uma nova geração. Ainda não é claro qual a melhor técnica para o cálculo dessas médias; no momento, advoga-se que a Média Ponderada (onde as configurações mais recentes e mais próximas no grafo de objetos tenham um maior peso) seja a melhor escolha.</p>

### 3.3.11 Outras Funcionalidades

Além das características citadas, a proposta desse trabalho agrega outras funcionalidades menos salientes:

#### 3.3.11.1 Automatização de mensagens de auxílio

Diversos tipos de mensagens podem ser apresentadas para o usuário durante a utilização do sistema. Devido as suas características, muitas delas podem ser geradas automaticamente, utilizando premissas bastante simples. Um pequeno exemplo disso é mostrado na Figura 3.9:

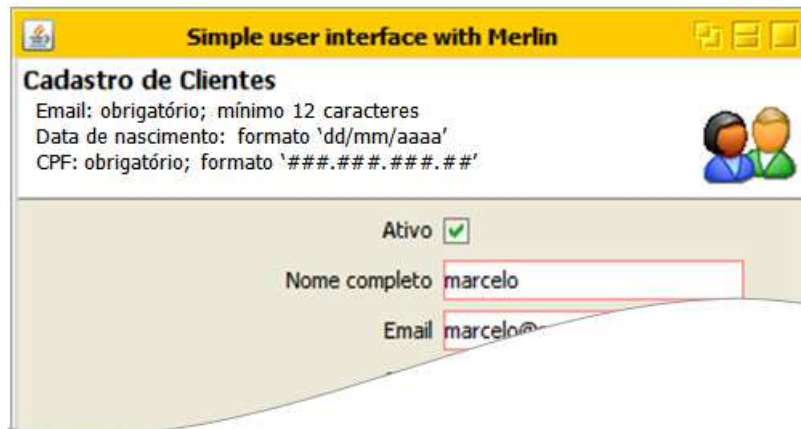


Figura 3.9: Mensagens de auxílio com dicas de preenchimento de tela geradas automaticamente.

Na Figura 3.9 está um fragmento de uma IU gerada automaticamente a partir de um Modelo de Domínio decorado com anotações provenientes do framework Hibernate Validator, base do padrão JSR 303 (JCP, 2006b). Utilizando uma máquina de *template* em conjunto com os algoritmos de renderização de textos, mensagens informativas, de auxílio e reportagem de erros podem ser produzidas integralmente. A criação de novos *templates* de mensagens é uma tarefa manual e feita conforme os moldes da Máquina de Template escolhida, que são, por padrão, o Velocity (VELOCITY, 2007a) e o FreeMaker (FREEMAKER, 2007a). Nessas mensagens, o suporte à internacionalização é possível e segue os padrões da plataforma Java.

### 3.3.11.2 Renderização de textos

Estendendo suporte ao processo de renderização de textos, mecanismos externos podem ser utilizados. Exemplos simplistas são o uso de corretores ortográficos, sejam eles nativos ou disponibilizados na Internet. O Capítulo 4 deste trabalho mostra isso em maiores detalhes.

### 3.3.11.3 Sinônimos e Similaridades

Uma característica não percebida nas MBUIDE analisadas é a capacidade de inferir resultados com base em aproximações. Tendo como heurística que nuances podem ocorrer durante a modelagem das classes do sistema (pequenos erros de grafia; falta de padronização no uso do singular ou plural; não-uniformidade na escolha das palavras, etc.) essa particularidade é colocada em evidência pela ferramenta.

Um exemplo dessa técnica é mostrado na Figura 3.10:

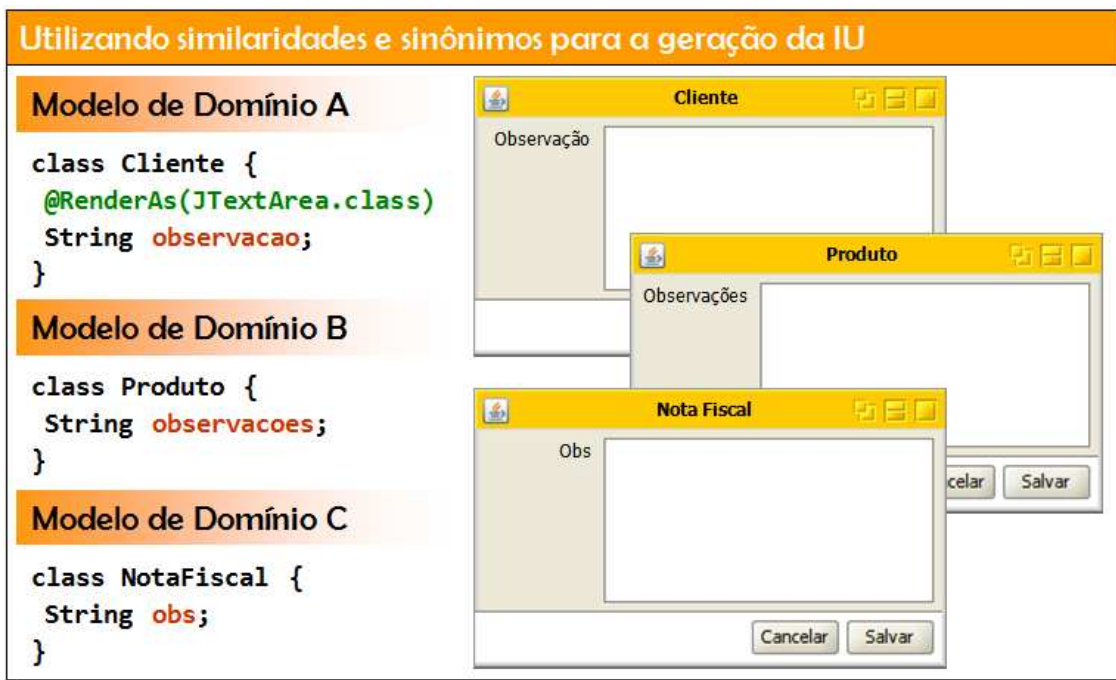


Figura 3.10: Utilização de similaridades e sinônimos para geração dos controles na IU.

Na Figura 3.10 são apresentados três modelos de domínios e respectivas as interfaces CRUD geradas. É intuito dessa ilustração evidenciar que no primeiro modelo (3.10a) o atributo `observacao`, é configurado para ser renderizado como um controle de texto de área. No segundo modelo (3.10b), o atributo `observacoes` (perceba o uso do plural), que não possui nenhuma informação de configuração, também é renderizado como um controle de texto de área. No terceiro modelo (3.10c) o atributo `obs`, que também não possui nenhuma configuração adicional, é, igualmente, renderizado como uma caixa de texto de área.

Para alcançar esse objetivo, sobre o campo `observacoes` (3.10b) foram aplicadas técnicas de similaridade de Strings disponíveis na biblioteca SimMetrics (SIMMETRICS, 2007) e para o campo `obs` (3.10c) foi realizada uma busca sobre um dicionário de sinônimos local no próprio computador do usuário. Ambos comportamentos são implícitos pelos algoritmos de renderização da ferramenta e podem ser modificados, se necessário.

### 3.4 Arquitetura

A arquitetura da ferramenta MERLIN pode ser descrita a partir de seus componentes, os quais dividem-se em quatro grupos.

- **Modelos:** Figurando como elemento essencial para o gerador, o Modelo de Domínio é o ponto de partida do gerador. Ele é formado pelo conjunto de classes e interfaces que representam a camada de persistência da aplicação. Sobre esse elemento primordial orbitam os Modelos de Apresentação, Tarefas, Diálogo e de Usuário, os quais são construídos e interligados através do uso das *Java Annotations*.
- **Anotações:** São elementos de configuração opcionais, que utilizando o



padrão *Java Annotations* (JCP, 2004b) balizam o detalhamento dos modelos da aplicação. As anotações podem ser tanto as definidas pela própria ferramenta (descritas no APÊNDICE), quanto reutilizadas de padrões de mercado (descritas na Tabela 3.1). Sendo as anotações compiladas junto às classes do sistema, forma-se um modelo auto-contido, que descarta dependências externas, seja para banco de dados ou arquivos de qualquer formato.

- **Algoritmos de renderização:** Os algoritmos de renderização nada mais são que fragmentos de código Java substituíveis que se encarregam de traduzir as anotações existentes nos modelos nos controles que constituem a interface do usuário. A ferramenta possui um conjunto básico de algoritmos para todos os modelos suportados e também admite a criação de novos. Exemplos típicos são a definição de novas funcionalidades para redefinição de *layout* de tela e para a geração de mensagens de auxílio.
- **Mecanismo de geração:** Utilizando recursos de introspecção da linguagem Java, a estrutura do Modelo de Domínio é varrida e as informações necessárias são extraídas. Nomes de classes, interfaces, pacotes, atributos, relacionamentos e métodos são utilizados para, em conjunto com as anotações, produzir a interface CRUD relacionada. Todo esse comportamento é balizado pelos algoritmos de renderização vinculados. Em suma, o mecanismo de geração tem como entrada uma classe Java compilada com as anotações e produz como resultado uma IU capaz de visualizar e manipular os valores contidos em quaisquer instâncias dessa classe.

### 3.5 Processo de Desenvolvimento

No tocante ao processo de desenvolvimento utilizado pela ferramenta MERLIN, a Figura 3.11 revela os detalhes de interesse:

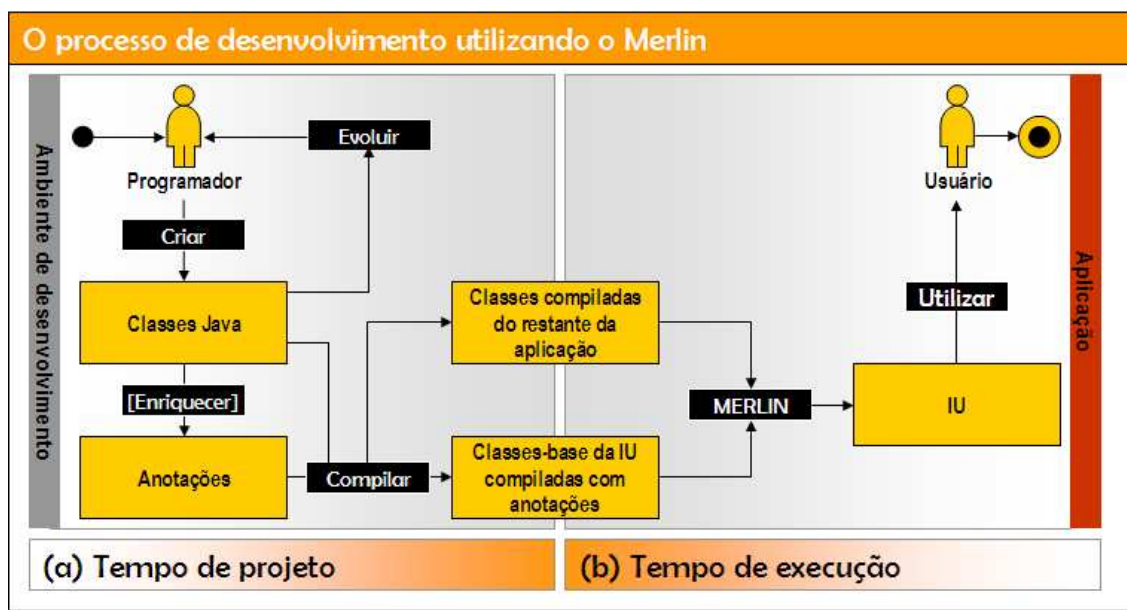


Figura 3.11: Processo de desenvolvimento de interfaces CRUD utilizando a ferramenta MERLIN.

Dividida em duas partes (*a* e *b*), a Figura 3.11 mostra o processo de desenvolvimento utilizando a ferramenta MERLIN.

Durante o Tempo de Projeto (3.11a), o programador cria o Modelo de Domínio do sistema, que nada mais é do que um conjunto de classes e interfaces Java. Quando inseridas as anotações sobre esse modelo, trabalho que pode ser realizado com o suporte textual da IDE de sua preferência, os outros modelos da ferramenta vão ganhando forma. Observa-se que esse é um processo interativo, que pode ocorrer inúmeras vezes durante o ciclo de vida do sistema. A cada interação, as classes compiladas do sistema são produzidas. Essas classes tanto podem ser as classes relativas às IU, quantas aquelas relativas ao restante da aplicação.

No Tempo de Execução (3.11b), o mecanismo de geração do MERLIN interpreta as informações contidas nas classes compiladas. Durante esse processo, os algoritmos de renderização traduzem a estrutura das classes envolvidas e as suas anotações atreladas nas IU CRUD correspondentes. É nesse ponto que a ligação das IU com o restante do sistema acontece. Ao final desse processo de geração, que dura alguns milissegundos, o usuário pode então utilizar as IU produzidas.

Observa-se que esse processo não exige nenhuma mudança de paradigma em relação aos processos convencionais de desenvolvimento. Em outras palavras, o único passo adicional corresponde aos trabalhos de adição das anotações nas classes Java, tarefa que não representa nenhum grande esforço, uma vez que o uso de anotações para a configuração de vários outros aspectos de um sistema é realidade para o desenvolvedor de aplicações Java.

## 4 ESTUDO DE CASO

*Este Capítulo aplica os conceitos apresentados nos capítulos anteriores, mostrando como criar uma interface CRUD de baixa complexidade utilizando a versão atual do software MERLIN.*

Tendo intuito didático, o exemplo abordado neste Estudo de Caso é simples. Esse posicionamento visa facilitar a compreensão das principais características da ferramenta sem poluir demasiadamente o texto com elementos técnicos. Caso fosse abordado um exemplo mais complexo, do leitor seria exigido o conhecimento de estruturas de programação Java de mais baixo nível, tais como *clousures*, Tipos Genéricos, conceitos de Enterprise Java Beans (EJB), injeção e ejeção de variáveis e Programação Orientada a Aspectos; conhecimentos estes aplicáveis somente a desenvolvedores de nível profissional.

Porém, caso o leitor tenha esse interesse por tais experimentos, situações mais elaboradas podem ser encontradas na forma de vídeos demonstrativos no endereço virtual <http://merlin.dev.java.net>, que é o espaço colaborativo do projeto MERLIN, local onde qualquer pessoa pode solicitar a participação.

### 4.1 O Cenário

Para demonstrar as capacidades da ferramenta MERLIN, o Estudo de Caso aborda a criação de uma Interface CRUD do tipo *um-para-um* (vide seção 3.3.8) sobre um objeto que representa um Cliente em um sistema de banco de dados qualquer.

Nessa entidade Cliente, algumas propriedades de tipo primitivo são declaradas e restrições de preenchimento de campos, *layout* e aparência de tela são aplicadas. Com o intuito de evidenciar a facilidade de ligação de regras de negócio na interface produzida, métodos de negócio declarados em outra classe do sistema são associados aos controles da interface CRUD gerada, tarefa que é realizada através do uso de Agentes, como descrito na seção 3.3.6.

### 4.2 Processo de Desenvolvimento

O processo de desenvolvimento utilizado não é baseado em nenhuma metodologia específica, porém segue as premissas da modelagem Orientada a Objetos e faz uso da linguagem Java para apresentação do Modelo de Domínio. No tocante ao pacote gráfico utilizado, a biblioteca Swing, integrada à distribuição Java, é o padrão utilizado.

Seguindo as bases de um desenvolvimento evolutivo e incremental, o Estudo de Caso é dividido em duas etapas. Em um primeiro momento, é criada a classe `Cliente`, a qual representa o Modelo de Domínio da aplicação. Feito isso, uma geração inicial é realizada, utilizando para isso uma outra classe chamada `TesteMinimoDoMerlin`, a qual representa o programa principal. Com uma versão preliminar da interface CRUD produzida, os comentários sobre essa geração são colocados.

Após essa primeira etapa, uma nova interação de desenvolvimento acontece. Nesse momento, a classe `Cliente` é refatorada para ter parte de seu conteúdo migrado para uma ancestral chamada `Pessoa` e o seu atributo `sexo` transformado em um tipo multivalorado (uma enumeração). Um outra classe, chamada `AlgumasRegras` é criada e nela um método de negócio é declarado. Esse método de negócio, que visa preencher o email do cliente a partir do seu nome, é associado à interface gerada através do uso de uma anotação `@Agent`. Outras anotações são colocadas na classe `Pessoa` e `Cliente` visando melhorias quanto à aparência e posicionamento dos elementos na tela. Com essas alterações, uma nova geração é executada e os comentários pertinentes apresentados.

### 4.3 Primeira Interação de Desenvolvimento

Na primeira interação de desenvolvimento são criadas as versões iniciais das classes referentes ao cliente e ao programa principal. O código-fonte de ambas as classes pode ser visualizado na Figura 4.1:

```

1 //Arquivo Cliente.java v.1
2 public class Cliente {
3     String nome;
4     String sexo;
5     Date dataDeNascimento;
6     String descricao;
7     String observacoes;
8     boolean ativo;
9     boolean vip;
10    String email;
11
12 //Arquivo TesteMinimoDoMerlin.java
13 public class TesteMinimoDoMerlin {
14
15     private static Logger log = LoggerFactory.getLogger(TesteMinimoDoMerlin.class);
16     private static IMerlin merlin = Factory.getMerlin(Implementation.SWING);
17
18     public static void main(String[] args) {
19         long startup = System.currentTimeMillis();
20         JPanel crud = (JPanel) merlin.createIhc(new Cliente(), true);
21         log.info("Geracao finalizada em: " + (System.currentTimeMillis() - startup) + "ms.");
22         JFrame frame = new JFrame("Simple user interface with Merlin");
23         frame.add(crud, BorderLayout.CENTER);
24         showIU(frame);
25     }
26
27     static void showIU(JFrame frame) {
28         JPanel barraDeBotoes = new JPanel(new FlowLayout(FlowLayout.RIGHT));
29         barraDeBotoes.setBackground(Color.white);
30         barraDeBotoes.setBorder(BorderFactory.createMatteBorder(1,0,0,0,Color.gray));
31         barraDeBotoes.add(new JButton("Cancelar"));
32         barraDeBotoes.add(new JButton(" Salvar  "));
33         frame.add(barraDeBotoes, BorderLayout.SOUTH);
34         frame.setSize(400, 370);
35         FrameUtil.showFrame(frame, null, FrameUtil.CANTO_INFERIOR_DIREITO);
36     }
37 }
38

```

Figura 4.1: Listagem do código-fonte da primeira interação de desenvolvimento do

Estudo de Caso sobre o software MERLIN.

#### 4.4 Análise do Código-Fonte

No código-fonte da Figura 4.1, na linha 2, está a definição da classe `Cliente`, com oito atributos simples. Observa-se que essa classe não é obrigada a estender ou implementar nenhuma outra classe ou interface e, da mesma forma, não contém nenhuma informação de configuração na forma de anotações. Esta classe `Cliente` representa o Modelo de Domínio do sistema e é utilizada como base de geração da interface CRUD.

Na linha 13 está a declaração da classe `TesteMinimoDoMerlin`, a qual representa o programa principal. Na linha 15 é declarado um objeto para fazer a contagem do tempo de geração consumido pela ferramenta. Na linha 17 é declarada uma instância do gerador MERLIN. Observa-se que isso é feito através de uma chamada a um método que utiliza o padrão *Factory* (GAMMA et al., 1995) indicando o *toolkit* gráfico a ser utilizado, que no caso é o Swing.

Na linha 21 é feito o início da contabilização do tempo de geração da interface CRUD para a classe `Cliente` e, na linha 22, uma chamada ao gerador MERLIN é efetuada. O método `createIhc()` do gerador recebe como parâmetro uma instância de uma classe `Cliente` e um parâmetro booleano, no caso `true`.

A partir da instância recebida, o gerador utiliza técnicas de introspecção para analisar e interpretar as informações sobre a estrutura e configurações existentes nesse objeto. Através dos algoritmos internos de renderização do gerador, essas informações são convertidas nos elementos que compõem a interface CRUD, tal como descrito no Capítulo 3. O parâmetro booleano é um indicador, informando que os controles de IU devem ser preenchidos com os valores das propriedades existentes na instância de `Cliente` especificada.

Como retorno do método `createIhc()`, está um painel gráfico contendo todos os controles de IU necessários para visualizar e editar as informações do objeto `Cliente` passado como parâmetro. É justamente esse painel o produto do gerador MERLIN.

Na mesma linha 22 se percebe que uma operação de conversão para o tipo `JPanel` (um tipo de controle de IU do Swing) é executada no retorno do método `createIhc()`. Isso é necessário uma vez que a ferramenta MERLIN é neutra em relação ao *toolkit* gráfico em uso e, portanto, não tem como retornar um tipo válido para cada biblioteca suportada<sup>3</sup>. A linha 23 computa e exibe tempo decorrido durante o processo de geração.

Nos testes realizados em um computador Pentium IV de 3.2Mhz e 2Gb de memória RAM executando o Sistema Operacional WindowsXP com a Java versão 1.6, a média aritmética de execução oscilou entre 290ms e 330ms para 10 passagens dos testes.

As linhas 25 à 27 são burocráticas, apenas adicionando o painel gerado ao formulário principal da aplicação.

Da linha 20 à 33 está o método auxiliar `showIU()`, que se encarrega de definir o

---

<sup>3</sup> A utilização dos tipos genéricos da Java (JCP, 2004a) é uma alternativa capaz de evitar esse tipo de conversão. Entretanto, isso depende de mais estudos e não está implementado na versão atual da ferramenta.

tamanho e posicionamento do formulário principal, bem como adicionar a barra de botões “Cancelar” e “Salvar”, que, salienta-se, não são geradas pela ferramenta MERLIN.

#### 4.5 Resultado da Geração da IU na Primeira Interação de Desenvolvimento

Com o código-fonte completo para o primeiro teste e, após a compilação das classes, a execução do sistema final produz um resultado conforme mostrado na Figura 4.2:

Figura 4.2: Resultado da geração da IU na primeira interação de desenvolvimento do Estudo de Caso sobre o software MERLIN.

Na Figura 4.2 está o resultado da geração referente à listagem de código mostrada na Figura 4.1. As numerações indicam as características mais salientes da interface e são assim descritas:

1. Painel de mensagens de texto, utilizado para exibir mensagens informativas, de auxílio e erros. À direita está a imagem associada ao formulário, com o intuito de criar uma associação cognitiva para o usuário (*affordance*). A exibição do painel de mensagens, bem como sua posição e conteúdo podem ser alterados com o uso de anotações. O painel de mensagens apresentado é o padrão da ferramenta e segue as premissas de exibição de mensagens da plataforma Eclipse (The Eclipse Foundation, 2007b).
2. Texto descritivo associado ao controle. Esse texto é gerado automaticamente pela ferramenta a partir do nome da propriedade na classe do Modelo de Domínio. Percebe-se que o texto teve o primeiro caractere colocado em

maiusculo e o nome separado em palavras conforme as regras *CamelCase* (CAMELCASE, 2007) aplicadas por inferência.

3. Nesse elemento percebe-se duas características. A primeira é a correção ortográfica aplicada sobre o texto; a segunda é o formato do controle, que é um texto de tamanho variável horizontalmente. Isso está em concordância com os algoritmos de renderização, os quais inferem, baseados em heurísticas, que um campo `descricao` deve ser maior que um campo `nome`, por exemplo.
4. Os campos `ativo` e `vip` são renderizados como caixas de marcação, uma vez que seus domínios de valores são booleanos. Renderizadores diferentes poderiam produzir controles como caixas de seleção de dois valores, ou grupos de botões mutuamente exclusivos para esses atributos. Anotações podem ser utilizadas para alterar esse comportamento.
5. Este controle é renderizado como uma caixa de texto de tamanho variável na horizontal e na vertical. Isso ocorre porque os algoritmos de renderização do gerador inferem que um campo `observacao` é factível de conter uma grande quantidade de texto e, por isso, deve ser maior que um campo `nome` ou `descricao`.
6. Observa-se que o posicionamento dos controles segue um formato tabular, onde os descritivos estão posicionados à esquerda e os controles de visualização e edição de informações, à direita. Esse é o algoritmo básico de posicionamento provido pela ferramenta. Outros algoritmos podem ser anexados ao Modelo de Domínio de forma a ter uma distribuição diferenciada dos elementos. Observa-se ainda que o ordenamento dos controles não segue a ordem de declaração de seus respectivos atributos nas classes relacionadas. Utilizando heurísticas como “priorizar atributos mais importantes” (como `nome` e `sexo`) e “rebaixar atributos menos importantes” (como `email`) ou “de maior tempo de interação com o usuário” (como `observacoes`), os algoritmos de posicionamento redefinem os controles na IU objetivando uma melhor aparência e facilidade de uso da interface gerada. Tais comportamentos também podem ser alterados com o uso de anotações.
7. Esta barra inferior de botões não é gerada pela ferramenta. Ela faz parte da aplicação final e é utilizada para realizar o salvamento dos dados atrelados ao formulário ou, para cancelar a edição atual. Uma das premissas da ferramenta MERLIN é ser neutra quanto à origem e destino dos dados do formulário. Em outras palavras, apenas se espera que um grafo de objetos (Modelo de Domínio) esteja disponível em memória para, a partir de um de seus objetos (no caso uma instância de `Cliente`) ser possível uma geração de IU. Quanto às formas de recuperação e salvamento do objeto em questão, a responsabilidade é da aplicação final, e não do gerador.

## 4.6 Segunda Interação de Desenvolvimento

Com o objetivo de comprovar a flexibilidade da ferramenta, quatro ações são executadas:

1. Primeiro, a classe `Cliente` é fragmentada em três outras classes, uma situação de refatoração muito comum em ambientes norteados pelo desenvolvimento

evolutivo.

- 2.Segundo, anotações de configuração (ênfatisadas por negrito e iniciadas pelo caractere arroba – @) são inseridas sobre as classes com o intuito de aprimorar a interface gerada.
- 3.A terceira alteração diz respeito à adição de duas regras de negócio simples. Uma delas exige que o email do cliente seja computado a partir do seu nome e a outra que, caso o cliente tenha um salário superior à R\$1500,00, ele seja considerado especial, tendo seu atributo `vip` marcado como verdadeiro. Ambas regras são definidas como métodos na classe auxiliar `AlgumasRegras`. Para vincular essas regra de negócio à interface CRUD do Cliente, anotações `@Agent` são utilizadas.
- 4.A quarta modificação implica na inicialização dos atributos nas classes que representam o Modelo de Domínio. Isso é feito com o intuito de demonstrar as capacidades da ferramenta em relação ao preenchimento dos controles na IU.

O resultado dessa evolução sobre a aplicação pode ser visualizada na Figura 4.3:

```

1 //Arquivo Sexo.java
2 public enum Sexo {
3     NAO_DECLARADO, MASCULINO, FEMININO
4 }
5
6 //Arquivo Pessoa.java
7 public class Pessoa {
8
9     @Agent(event = { "focusLost" }, action = { "proposeEmail" })
10    @NotNull
11    String nome = "marcelo";
12
13    public Sexo sexo = Sexo.MASCULINO;
14
15    @Agent(property = { "foreground=Color.blue" }, binder = DateToTextComponent.class)
16    Date dataDeNascimento = new Date(1977-1900,9,02);
17
18 }
19
20 //Arquivo Cliente.java v.2
21 @Caption("Cadastro de Clientes")
22 public class Cliente extends Pessoa {
23
24     @Agent(
25         event = { "focusLost" },
26         action = { "isVip" }
27     )
28     @Caption("Salário (R$)")
29     int salario;
30
31     @Order(after = "observacoes")
32     String descricao;
33
34     @RenderAs(JCheckBox.class)
35     @Dependence("descricao")
36     @Agent(property = { "selected=true" })
37     String observacoes;
38
39     @Dependence(":all")
40     @Order(Order.FIRST)
41     @Agent(property = { "selected=true;" })
42     boolean ativo;
43
44     @RenderAs(value = JTextField.class, binder = BooleanToTextComponentBinder.class)
45     @Order(LAST)
46     boolean vip = true;

```



```

47
48     @Order(after = "nome")
49     @NotNull
50     @Length(min=12)
51     String email;
52
53 }
54
55 //Arquivo AlgumasRegras.java
56 public class AlgumasRegras {
57
58     @In
59     Context ctx;
60
61     public void isVip() {
62         ctx.eval("$vip.text = $salario.text > 1500 ? 'SIM' : 'NÃO');
63     }
64
65     public void proposeEmail() {
66         ctx.eval("$email.text = $nome.text.trim + '@merlin.com");
67     }
68
69 }

```

Figura 4.3: Listagem do código-fonte da segunda interação de desenvolvimento do Estudo de Caso.

## 4.7 Resultado da Geração da IU na Segunda Interação de Desenvolvimento

Uma vez alteradas as classes referentes ao Modelo de Domínio, nenhuma outra modificação é necessária em qualquer outra parte do sistema. Assim, basta compilar e executar novamente<sup>4</sup> o programa principal e o resultado da geração é conforme o apresentado na Figura 4.4:

---

<sup>4</sup> É possível ativar o recarregamento automático de classes da ferramenta MERLIN através de uma chamada ao método polimórfico `createIhc(Object, boolean, boolean)`, sendo o terceiro parâmetro um valor `TRUE`. Nesse caso, a ferramenta monitora alterações nas classes compiladas do sistema, recarregando-as automaticamente e permitindo que modificações nas interfaces CRUD sejam efetuadas mesmo com o sistema em execução. Entretanto, embora essa abordagem seja um recurso interessante, ela pode causar instabilidades para o usuário da aplicação se for mal utilizada. Em suma, esse recurso torna-se bastante interessante em ambientes distribuídos e em prototipadores.

The image shows a web form titled "Simple user interface with Merlin" with the subtitle "Cadastro de Clientes". The form has several fields and controls, each marked with a number from 1 to 10. Field 1 is the title. Field 2 is a checkbox labeled "Ativo" which is checked. Field 3 is a text input for "Nome completo" containing "marcelo". Field 4 is a text input for "Email" containing "marcelo@merlin.com". Field 5 is a dropdown menu for "Sexo" set to "Masculino". Field 6 is a date picker for "Data de nascimento" set to "02 de Outubro de 1977". Field 7 is a text input for "Salário (R\$)" containing "940". Field 8 is a checkbox for "Observações" which is unchecked. Field 9 is a text area for "Descrição". Field 10 is a dropdown for "Vip" set to "NÃO". At the bottom are "Cancelar" and "Salvar" buttons.

Figura 4.4: Resultado da geração da IU na segunda interação de desenvolvimento do Estudo de Caso sobre o software MERLIN.

Dez pontos são destacados na Figura 4.4:

1. O painel de mensagens do sistema contém algumas informações. Em fonte maior, está a descrição da interface CRUD. Essa descrição é originada na anotação `@Caption` anexada à classe `Cliente`. Logo abaixo, em fonte menor, está uma lista de mensagens informativas de auxílio ao usuário. As mensagens informativas são produzidas automaticamente pelo gerador em função das anotações de validação (`@NotNull` e `@Length`) sobre as propriedades `email` e `nome`. Essas anotações não fazem parte da ferramenta, mas são reusadas do padrão JSR 303 (JCP, 2006b). Salienta-se que isso é uma característica muito importante, uma vez que, independente de utilizar ou não uma ferramenta de geração automatizada para a IU, essas anotações de validação estão comumente presentes no Modelo de Domínio, visto que a tendência das aplicações Java é convergir para esse padrão de validação. Observa-se, também, que a imagem à direita presente no painel de mensagens é diferente da primeira geração. Isso ocorre porque, utilizando uma heurística simples, a presença de arquivos de imagem de mesmo nome no mesmo diretório da classe do Modelo de Domínio, faz com que tais arquivos sejam utilizados como *affordances* de cognitividade na tela. Esse comportamento é balizado pelos algoritmos de renderização e pode ser modificado via anotações.
2. O controle de IU `ativo`, aparentemente, não possui nenhum comportamento especial. Entretanto, ao ser desmarcado, todos os controles do formulário (exceto ele) são desabilitados. Isso acontece devido à presença da anotação `@Dependence(" :all")` sobre ele. Da mesma forma, ele é o primeiro controle do formulário porque está marcado com a anotação `@Order(FIRST)`. Ainda, ele (e todos os outros controles da IU) estão preenchidos com os valores das respectivas propriedades do objeto `Cliente` especificado como parâmetro.

Observa-se que a anotação `@Order` é abstrata (um AIO), uma vez que a noção de primeiro (FIRST) ou último (LAST) é relativa e dependente do algoritmo de layout atrelado à IU.

3. O controle de caixa de texto com o nome do cliente está com uma borda em vermelho devido o renderizador associado ter detectado a presença da anotação `@NotNull` sobre esse campo. Essa borda é outro *affordance* visual que auxilia o usuário na identificação de campos de preenchimento obrigatório na tela. O mesmo vale para o campo de email.
4. O campo `email` está preenchido com um valor. Isso ocorre porque sobre o campo `nome` existe uma anotação que declara um agente. Esse agente monitora o controle de IU `nome` e, quando este perder o foco de edição na tela, a regra de negócio `proposeEmail()` é executada. Essa regra de negócio está definida dentro da classe `AlgumasRegras`, e utiliza código de *scripting* Groovy (JCP, 2004d) para sua execução. Observa-se que a técnica de injeção de recursos é utilizada, permitindo transparência na passagem de parâmetros entre a IU e o método de negócio. Como consequência dessa abordagem, também se obtém independência de pacote gráfico, uma vez que o código do método de negócio não faz referência explícita a nenhum *toolkit* gráfico específico.
5. O campo `sexo` é de interesse. Na primeira versão do Modelo de Domínio ele era do tipo `String` e, agora, ele é um tipo enumerado, que apresenta três valores distintos conforme o modelo. Assim, durante a renderização da IU, o gerador infere que o melhor controle de IU para apresentar seus valores é uma caixa de seleção de valor único. O valor inicial é definido conforme o existente no objeto do parâmetro. Variações sobre isso podem ser obtidas com o uso de anotações.
6. O campo `dataDeNascimento` também sofreu as regras de quebra no padrão CamelCase e tem outros dois diferenciais. Primeiro, o texto está exibido na cor azul devido a existência da anotação `@Agent` que define uma propriedade de tela. Segundo, o formato de data exibido está num padrão personalizado. Isso ocorre devido a presença de uma classe de *binder* específico, que converte e apresenta valores conforme a necessidade do projeto. Isso também foi especificado pela anotação `@Agent` sobre essa propriedade.
7. Um texto descritivo personalizado é usado para o campo `salario` e, sobre ele, uma regra de negócio também é aplicada através de um agente. Quando esse controle de IU `salario` perde o foco de edição, o valor contido nele é convertido para um número inteiro e, se maior que 1500, incorre na colocação do valor “SIM” no controle de IU `vip`. Se o valor for menor que esse limite, o controle de IU `vip` recebe o valor “NÃO”.
8. Interessante no campo `observacoes` é que ele está renderizado como uma caixa de marcação – antes, conforme mostra a Figura 4.2, ele era uma caixa de texto. Isso ocorre porque ele está configurado com a anotação `@RenderAs(JCheckBox.class)`. Essa anotação informa que os algoritmos de renderização devem deixar de usar as heurísticas de mapeamento e balizar-se pelo conteúdo da anotação. Nesse caso, a classe de renderização é um CIO, que vincula diretamente uma classe do pacote Swing. Nada impede de ser utilizado um AIO, como `@RenderAs(CHECKBOX)`, abordagem que abre possibilidade de mapeamentos genéricos para diferentes *toolkits* gráficos.

9. O campo `descricao` está desabilitado. Isso ocorre porque no campo de `observacoes` existe uma anotação `@Dependence`. Essa anotação implica que, ao desmarcar o controle de observações, o campo de descrição deve ser desabilitado por consequência.
10. O campo booleano `vip` também está sendo renderizado com um controle de IU diferente do padrão esperado, que seria uma caixa de marcação. Isso ocorre porque uma anotação `@RenderAs` está sobre ele. Observa-se, também, que a palavra “NÃO” é exibida no controle. Isso acontece porque existe uma classe de *binder* atrelada que se encarrega de mapear os valores booleanos `TRUE` e `FALSE` para as palavras “SIM” e “NÃO”, respectivamente. Essa classe de *binder* é padrão da ferramenta, mas também poderia ser criada por um programador conforme as necessidades de negócio.

Em relação aos tempos de execução do sistema nesse segundo ciclo de testes, a média aritmética entre 10 passagens de geração oscilou entre 290 e 313ms. Em outras palavras, não é percebida nenhuma diferença de desempenho em relação ao Modelo de Domínio sem anotações. Obviamente, esse exemplo é muito pequeno, e não pode ser generalizado sem maiores cuidados.

## 4.8 Considerações sobre o Estudo de Caso

O exemplo apresentado nesse Estudo de Caso é simples devido seu caráter didático. Ele não engloba estruturas mestre-detalle nem dependências entre sistemas distintos. Porém, apresenta características importantes, como a distribuição de elementos na tela (*layout*), a aplicação de regras de usabilidade, o suporte automático a validadores de conteúdo e máscaras, a geração de mensagens de usuário e a ligação de regras de negócio sem dependências sintáticas de compilação.

Com o reuso efetivo de padrões de mercado, a solução incita uma redução na curva de aprendizagem do desenvolvedor. Não obstante, devido à premissa da Edição Textual Assistida (ETA) para edição dos modelos, os ambientes de desenvolvimento com recursos de complemento de código (*code completion*) facilitam consideravelmente o preenchimento das anotações.

Entretanto, é evidente que o código-fonte do modelo de dados tende a aumentar consideravelmente com essas meta-informações (anotações) associadas. Para reduzir esse efeito podem ser utilizados recursos como “dobras de código” (*code folding*) existente nas modernas IDE. Outra abordagem, ainda em estudo, é a migração dessas anotações para outras partes do código, como classes relacionadas. Para tanto, podem ser aplicados *patterns* específicos, como o *Decorator* (GAMMA et al., 1995) ou adicionadas novas funcionalidades à própria IDE (*plugins*), de forma a possibilitar a ligação transparente e gerenciada entre as classes do Modelo de Domínio e as respectivas anotações.

Salienta-se ainda que o mecanismo de realimentação pode reduzir drasticamente a quantidade de informações necessárias à configuração dos modelos. Entretanto, isso ainda não é possível na versão atual da ferramenta.

Quanto ao código-fonte da aplicação principal, a chamada ao método `createIhc()`

encapsula qualquer complexidade para o programador, recebendo como parâmetro um objeto de dados e retornando o conteúdo de uma interface CRUD completa.

De forma geral, este Estudo de Caso demonstra que é possível utilizar o software MERLIN para produzir interfaces CRUD de forma automatizada, sem que isso implique em aprender conceitos novos, externos à plataforma de desenvolvimento, no caso a Java.

## 5 CONCLUSÃO

Nessa dissertação foi apresentado o software MERLIN, que é uma nova proposta na área das ferramentas MBUIDE. O MERLIN é um típico gerador baseado em modelos capaz de assistir e automatizar a geração de Interfaces CRUD comumente presentes em sistemas de banco de dados.

Como base para este trabalho, uma análise de 15 outras MBUIDE foi efetuada, permitindo o detalhamento de diversas de suas características e a identificação dos principais modelos de armazenamento utilizados por estas ferramentas.

Deste panorama traçado, se conclui que as MBUIDE apresentam uma solução elegante e com bom nível de abstração para o desenvolvimento de interfaces de usuário. Entretanto, devido a freqüente utilização de linguagens proprietárias e sintaxes de difícil compreensão, elas acabam se tornando inviáveis para a maioria das equipes profissionais de desenvolvimento de sistemas.

Visando sanar essas dificuldades, o MERLIN objetiva trazer os benefícios da geração automatizada de IU ao mesmo tempo em que reutiliza tecnologias e conceitos comumente presentes em ambientes profissionais, mais evidentemente na plataforma Java, foco da implementação atual.

Em relação aos diferenciais oferecidos pelo MERLIN, cita-se em ordem de relevância:

1. A utilização da Java como linguagem de programação de formação dos modelos, característica que evita o uso de formalismos, gramáticas e sintaxes proprietárias para a definição de modelos;
2. A centralização das configurações sobre o Modelo de Domínio, permitindo que todos os outros modelos da ferramenta sejam construídos e configurados em um único ponto, utilizando para isso o padrão *Java Annotations*;
3. A forte ênfase no reuso de configurações, sugerindo ainda um mecanismo de configuração baseado em histórico;
4. A formação de um modelo auto-contido de configuração que, constituído pelas próprias classes compiladas do sistema, não causa dependências externas para o aplicativo final;
5. A produção das IU em tempo de execução e sem a geração de código-fonte, o que inibe os efeitos nocivos da evolução de um sistema;
6. Norteadada pela Configuração Por Exceção (CPE), a premissa é que os

algoritmos internos sejam pró-ativos, fazendo com que somente as exceções de configuração sejam definidas, característica que reduz significativamente a quantidade de configurações necessárias para a geração das IU;

7. Adotando o paradigma da Orientação a Objetos, a ferramenta está em concordância com as modernas escolas de modelagem e em sintonia com *frameworks* e ferramentas de mercado;
8. Suportando configurações em níveis abstratos e concretos, diferentes *toolkits* gráficos podem ser alcançados;
9. Oriundos da linguagem Eiffel, os Agentes permitem que a ligação dos modelos aconteça sem dependências sintáticas – um importante diferencial para sistemas distribuídos;
10. Balizada por algoritmos de renderização substituíveis, o gerador pode se adequar confortavelmente à diversos cenários de desenvolvimento;
11. Com três abordagens distintas para a geração de *layout*, não existem restrições em relação ao formato das IU produzidas;
12. Utilizando o conceito da Edição Textual Assistida (ETA), a ferramenta se integra às IDE existentes sem maiores dificuldades;
13. Amparada pela configuração por similaridade e proximidade, a capacidade de inferir comportamentos é aumentada;
14. Acoplado-se a recursos nativos do ambiente, como corretores ortográficos e dicionários de sinônimos, a geração de mensagens de textos na IU é facilitada.

Porém, mesmo com os avanços obtidos durante a pesquisa, muitos desafios continuam presentes. Os mais salientes são:

1. Suportar interfaces CRUD dos tipos *um-para-muitos* e *muitos-para-muitos*: previstas e inicialmente exploradas no Trabalho Individual (MRACK, 2005), esses tipos de IU não têm suporte na implementação atual da ferramenta. Necessitando estudos mais detalhados, que devem incorrer em evoluções nos Modelos de Diálogo e de Apresentação, a geração dessas IU deve fazer parte de uma próxima versão do software.
2. Finalizar o suporte às expressões Object Constraint Language (OCL) e Expression Language (EL): Com o Modelo de Tarefas já preparado para dar o suporte necessário, a implementação desses recursos depende da escolha e integração dos *frameworks* de apoio, os quais são responsáveis por efetuar o *parse* e a execução das expressões vinculadas.
3. Maximizar o conjunto de algoritmos de *layout* disponíveis na ferramenta: No momento, apenas um algoritmo de posicionamento de controles na IU está disponível, efetuando a disposição dos controles no formato tabular. Incrementar as configurações possíveis, bem como disponibilizar novos algoritmos de posicionamento é um ponto importante, uma vez que é objetivo da ferramenta automatizar o máximo possível a apresentação e a distribuição dos elementos nas IU.

4. Estabilizar a implementação do *binding* e do suporte à validação de controles na IU: Embora a ferramenta seja baseada nos padrões de mercado emergentes, muitos deles apresentam conflito entre si e a tendência é que uma estabilidade aconteça somente a partir da finalização dos padrões JSR 295 e JSR 303, prevista para o final do ano 2008. Aguardar a definição desse cenário e ajustar a implementação existente é um trabalho importante a ser desenvolvido.
5. Implementar renderizadores para outros *toolkits* gráficos: A implementação corrente é baseada no *toolkit* gráfico Swing, padrão Java para ambientes *desktop*. Duas versões para web também devem ser disponibilizadas em um segundo momento através da implementação de renderizadores nos pacotes gráficos Java Server Faces (JSF) e Google Web Toolkit (GWT). Porém, muitos outros bibliotecas gráficas existem. Implementar versões para pacotes como o Standard Widget Toolkit (SWT), Thinlet e o XML-based User Interface Language (XUL) são alternativas válidas quando o mercado comercial é colocado em perspectiva.
6. Desenvolver *plugins* e ferramentas de suporte para as IDE: A premissa da Edição Textual Assistida (ETA) é um interessante recurso para edição assistida dos modelos. Entretanto, para dar um suporte completo ao programador, é importante a construção de módulos de auxílio (*plugins*) para as IDE, os quais podem habilitar funções como o *parsing* das expressões OCL e EL e a visualização dos resultados da geração em tempo de projeto e também a validação do conteúdo e das ligações entre os modelos.
7. Explorar em profundidade o mecanismo de configuração baseado em histórico: Quando iniciadas as pesquisas, muitas expectativas foram colocadas sobre esse item. Porém, no decorrer do trabalho, dificuldades começaram a surgir, principalmente devido aos modelos da ferramenta serem auto-contidos nas próprias classes compiladas do sistema. Assim, técnicas para manutenção, gerenciamento e propagação de informações ao longo da cadeia de configurações precisam ser estudadas. Vislumbrando isso como um dos elementos mais complexos da ferramenta, a realização de um estudo profundo, em nível de Doutorado pode ser uma alternativa interessante.



## REFERÊNCIAS

- AGENT FOR JAVA (A4J). 3Layer Tecnologia. Disponível em <https://agent4java.dev.java.net/>. Acesso em: dez. 2007.
- ALOIA, N.; CONCORDIA, C.; PARATORE, M. T.. **Automatic GUI Generation for Web Based Information Systems**. 2003. Disponível em <http://dienst.isti.cnr.it/Dienst/UI/2.0/Describe/ercim.cnr.isti/2003-TR-50?tiposearch=cnr&langver=>. Acesso em: out. 2005.
- AMBLER, S. W.. **User Interface Design: Tips and Techniques**. Ambler, Scott W.. 2004. Disponível em <http://www.ambysoft.com/userInterfaceDesign.pdf>. Acesso em: jan. 2006.
- AVRAM, A.; MARINESCU, F.. **Domain-Driven Design Quickly**. 1.ed. EUA: C4Media Inc, 2006.
- BACKHAUS, D.. **From Business Process to Online Solutions: Using XML Schemas as the Central Component for Rapid Application Development**. XML Europe Internationales Congress Centrum (ICC), Berlin. 2001.
- BARON, M.; GIRARD, P.. **SUIDT: A Task Model Based GUI-Builder**. Citeseer, Proceedings... Task Models and Diagrams for User Interface Design (TAMODIA'02), Bucareste, [s.n.], p. 64-71, jul. 2002.
- BODART, F.; LEHEUREUX A.; VANDERDONCKT, J.. **A Model-Based Approach to Presentation: A Continuum from Task Analysis to Prototype**. Institut d'Informatique, Bélgica, [s.n.]. 1994.
- BODART, F.; VANDERDONCKT, J. M.. **Encapsulating Knowledge For Intelligent Automatic Interaction Objects Selection**. Proceedings... Conference on Human Aspects in Computing Systems (InterCHI'93), Amsterdam, [s.n.], p. 424-429, abr. 1993.
- CAMELCASE. Wikipédia, the free encyclopedia. Disponível em <http://en.wikipedia.org/wiki/CamelCase>. Acesso em:dez. 2007.
- CLIFTON, M.. **A Dynamically Generated XML Data Editor**. 2005. Disponível em <http://www.codeproject.com/dotnet/xmldataeditor.asp>. Acesso em: nov. 2005a.
- CLIFTON, M.. **Web & Window Form Unification: Synchronous And Asynchronous Event Handling For Controls Created At Runtime**. 2005. Disponível em <http://www.codeproject.com/cs/miscctrl/dynformpartii.asp>. Acesso em: nov. 2005b.
- COOPER, R.; MCKIRDY, J.; GRIFFITHS, T.; BARCLAY, P. J.; PATON, N.; GRAY P. D.; KENNEDY, J.; GLOBE C. A.. **Conceptual Modelling for Database User Interfaces**. University of Manchester, Reino Unido, [s.n.], 1997.
- COUTAZ, J.. **PAC, an Object Oriented Model for Dialog Design**. Human-Computer Interaction (INTERACT'87), Alemanha, Elsevier Science Publishers p. 431-436, set.

1987.

ECLIPSE IDE. The Eclipse Foundation. Disponível em <http://www.eclipse.org/eclipse/>. Acesso: dez. 2007b.

EVANS, E.. **Domain-Driven Design – Tackling Complexity in the Heart of Software**. Final Manuscript. Califórnia, EUA, , 2003.

FREEMAKER: Java Template Engine Library. SourceForge.net. Disponível em <http://freemarker.sourceforge.net/>. Acesso: dez. 2007a.

GALITZ, O. W.. **The Essential Guide to User Interface Design-An Introduction to GUI Design Principles and Techniques**. 2.ed. New York: John Wiley and Sons, 2002.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J.. **Design Patterns: Elements of Reusable Object-Oriented Software**. 1.ed. [S.l.]: Addison Wesley, 1994.

GOHIL, B.. **Automatic GUI Generation**. 1999. 58f. Dissertação (Bacharelado em Engenharia de Software) - Bournemouth University, Reino Unido.

GOOGLE WEB TOOLKIT (GWT). Google, Inc. Disponível em <http://code.google.com/webtoolkit/>. Acesso: dez. 2007.

GOSLING, J.; JOY, B.; STEELE, G.; BRACHA, G.. **The Java™ Language Specification**. 2.ed. [S.l.]: Prentice Hall, 2005.

GRAY, P.; COOPER, R.; KENNEDY, J.; BARCLAY, P.; GRIFFITHS, T.. **A Lightweight Presentation Model for Database User Interfaces**. [S.l.:s.n.]. 1998.

GRIFFITHS, T.; MCKIRDY, J.; FORRESTER, G.; PATON, N.; KENNEDY, J.; BARCLAY, P.; COOPER, R.; GOBLE, C.; GRAY, P.. **Exploiting model-based techniques for user interfaces to database**. [S.l.:s.n.]. 1997.

JAR FILE SPECIFICATION. Sun Microsystems, Inc. Disponível em <http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html>. Acesso em dez: 2007.

JAVA COMMUNITY PROCESS (JCP). **JSR 14 - Add Generic Types To The Java™ Programming Language (Java Generics)**. [S.l.], 2004a.

JAVA COMMUNITY PROCESS (JCP). **JSR 175 - A Metadata Facility for the Java™ Programming Language (Java Annotations)**. [S.l.], 2004b.

JAVA COMMUNITY PROCESS (JCP). **JSR 220 - Enterprise JavaBeans™ 3.0**. [S.l.], 2004c.

JAVA COMMUNITY PROCESS (JCP). **JSR 227 - A Standard Data Binding and Data Access Facility for J2EE™**. [S.l.], 2003.

JAVA COMMUNITY PROCESS (JCP). **JSR 241 – The Groovy Programming Language**. [S.l.], 2004d.

JAVA COMMUNITY PROCESS (JCP). **JSR 245 – JavaServer™ Pages 2.1**. [S.l.], 2006c.

JAVA COMMUNITY PROCESS (JCP). **JSR 252 – JavaServer Faces 1.2**. [S.l.], 2006d.

JAVA COMMUNITY PROCESS (JCP). **JSR 273 - Design-Time API for JavaBeans™ JBDT**. [S.l.], 2005a.

JAVA COMMUNITY PROCESS (JCP). **JSR 274 – The BeanShell Scripting Language**. [S.l.], 2005b.

- JAVA COMMUNITY PROCESS (JCP). **JSR 295 – Beans Binding**. [S.l.], 2006a.
- JAVA COMMUNITY PROCESS (JCP). **JSR 303 – Bean Validation**. [S.l.], 2006b.
- JAVA COMMUNITY PROCESS (JCP). **JSR 317 – Java Persistence API 2.0**. [S.l.], 2008a.
- JAVA COMMUNITY PROCESS (JCP). **JSR 318 – Enterprise Java Beans 3.1**. [S.l.], 2008b.
- JAVA COMMUNITY PROCESS (JCP). **JSR 299 – Web Beans**. [S.l.], 2008c.
- JAVASSIST. JBoss, a division of Red Hat. Disponível em <http://labs.jboss.com/javassist/>. Acesso em: nov. 2007a.
- JBOSSAS. JBoss, a division of Red Hat. Disponível em <http://labs.jboss.com/jbossas/>. Acesso em dez. 2007b.
- JOHNSON, P.; WILSON, S.; MARKOPOULOS, P.; PYCOCK, J.. **ADEPT - Advanced Design Environment for Prototyping with Task Models**. Department of Computer Science, Queen Mary & Westfield College, University of London, [s.n.]. 1993.
- KLEPPE, A.; WARMER, J.. **The Object Constraint Language: Precise Modeling With UML**. 1.ed. [S.l.]: Addison-Wesley, 1999.
- KLINGER, D. A.; KROTH, E.. **Um Software Assistente para Especificação de Regras de Negócio**. [S.l.:s.n.]. 2001.
- LUYTEN, K.. **Dynamic User Interface Generation for Mobile and Embedded Systems with a Model-Based User Interface Development**. 2004. 252 f. Tese ( Doutorado em Informática ) – Limburg Universiteit (extensão), Bélgica.
- MARINESCU, F.. **EJB Design Patters – Avanced Patterns, Processes and Idioms**. 1.ed. EUA: John Wiley and Sons, 2002.
- MEYER, B.. **Eiffel : The Language**. 1.ed. [S. l.]: Prentice Hall, 1991.
- MEYER, B.. **Object Oriented Software Construction**. 2.ed. [S. l.]: Prentice Hall, 2000.
- MRACK, M.. **Interfaces de Usuário em Tempo de Execução**. 2005. 114 f. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.
- MRACK, M.. **Merlin : Um Novo Horizonte na Criação das Telas de Cadastro**. 8º Fórum Internacional de Software Livre (FISL). Porto Alegre, 2007.
- MOREIRA, A. F.; MRACK, M.; PIMENTA, M. S.; **Merlin: Interfaces CRUD Em Tempo de Execução**. XX Simpósio Brasileiro de Engenharia de Software (SBES'06) – Sessão de Ferramentas. Florianópolis, 2006.
- MOREIRA, D.; MRACK, M.. **Sistemas Dinâmicos Baseados em Metamodelos**. II Workshop de Computação e Gestão da Informação (WCOMPI), Lajeado, 2003.
- MYERS, A. B.. **User Interface Software Tools**. ACM Transactions on Computer-Human Interaction (TOCHI'95), New York, v. 3, n.1, p. 64-103, Mar. 1995.
- OBJECT MANAGEMENT GROUP (OMG). **Introduction to OMG Unified Modeling Language (UML)**. Disponível em: [http://www.omg.org/gettingstarted/what\\_is\\_uml.htm](http://www.omg.org/gettingstarted/what_is_uml.htm). Acesso em: nov. 2007a.

OBJECT MANAGEMENT GROUP (OMG). Object Constraint Language Specification, version 2.0. Disponível em: <http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf>. Acesso em: dez. 2007b.

PINHEIRO, P.. **User Interface Declarative Models and Development Environment: A Survey**. Lecture Notes in Computer Science : Springer-Verlag, Germany, [s.n.], p. 207-226. 2001.

PIZANO, A.; SHIROTA, Y.; IIZAWA, A.. **Automatic Generation of Graphical User Interfaces for Interactive Database Applications**. Conference on Information and Knowledge Management : ACM Press, Washington, D.C., p. 344-355, 1993.

PUERTA, A.; ANGEL, R.; ERIKSSON, H.; GENNARY, J. H.; MUSEN, M.. **Beyond Data Models for Automated User Interface Generation**. Stanford University – Califórnia, 1996.

PUERTA, A.; EISENSTEIN, J.; **Towards a General Computational Framework for Model-Based Interface Development Systems**. Stanford University, Califórnia. 1999.

REENSKAUG, T.. **THING-MODEL-VIEW-EDITOR an Example from a planning system**. [S. l.], [s.n.], 1979. Disponível em <http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf>. Acesso em: dez. 2007.

REFLECTION API, The Java Reflection API. Sun Microsystems, Inc. Disponível em <http://java.sun.com/j2se/1.5.0/docs/guide/reflection/index.html>. Acesso: dez. 2007.

RUMBAUGH, J. R.; BLAHA, M. R.; LORENSEN, W.; EDDY, F.. **Object-Oriented Modeling and Design**. 1.ed. EUA: Prentice Hall. 1990.

SCHLUNGBAUM, E. **Individual User Interfaces and Model-Based User Interfaces Software Tools**. [S.l.:s.n.]. 1997.

SCHLUNGBAUM, E.; THOMAS, E.. **Automatic User Interface Generation from Declarative Models**. Proceedings... Computer-Aided Design of User Interfaces (CADUI'96). Namur (Bélgica), [s.n.]. 1996.

SCHLUNGBAUM, E.. **Model-based User Interface Software Tools Current State of Declarative Models**. [S.l.:s.n.]. 1996.

SHIROTA, Y.; IIZAWA, A.. **Automatic GUI generation from database schema information**. Systems and Computers in Japan, Tokyo, v. 28, n.5 , p. 1-10, dez. 1997.

SIMMETRICS: Open Source Similarity Measure Library. University of Sheffield, Department of Computer Science. Reino Unido. Disponível em <http://www.dcs.shef.ac.uk/~sam/simmetrics.html>. Acesso: dez. 2007a.

SOUCHON, N.; LIMBOURG, Q.; VANDERDONCK, J.. **Task Modelling in Multiple Contexts of Use**. Citeseer, [S.l.:s.n.]. 2002.

SWT, The Standard Widget Toolkit (SWT). The Eclipse Foundation. Disponível em <http://www.eclipse.org/swt/>. Acesso: dez. 2007a.

SZEKELY, P.. **Retrospective and Challenges for Model-Based Development**. Califórnia, [s.n.]. 1996.

THINLET. Thinlet.com. Disponível em <http://www.thinlet.com/>. Acesso: dez. 2007.

VANDERDONCK, J.. **Automatic Generation of a User Interface For Highly Interactive Business-Oriented Applications**. Facultés Universitaires Notre-Dame, Bélgica, [s.n.]. 1994.

VELOCITY, The Apache Velocity Project. The Apache Software Foundation. Disponível em <http://velocity.apache.org/>. Acesso: dez. 2007a.

WORLD WIDE WEB CONSORTION (W3C). **WebServices Activity**. [S.l.], 2002.

XML-BASED USER INTERFACE LANGUAGE (XUL). Mozilla.org. Disponível em <http://www.mozilla.org/projects/xul/>. Acesso: dez. 2007.

## APÊNDICE A – TABELA COMPARATIVA DAS MBUIDE ANALISADAS

Este apêndice estrutura as informações pesquisadas neste trabalho de forma a permitir uma análise mais direta entre as MBUIDE estudadas.

Para o entendimento da tabela apresentada, as seguintes informações são relevantes:

- **Linha 2. Modelos suportados:** Os caracteres em maiúsculo nas palavras [dAdos; Domínio; Tarefa; Usuário; dIálogo; aPresentação; apLicação; plataForma; cOntexto;] denotam o tipo de modelo suportado pela ferramenta, sendo que o primeira letra mostrada na célula é o modelo principal da ferramenta e, quando não separados por vírgula, entende-se que estes não têm uma separação clara no software.
- **Linha 3. Especificação da IU:** O termo Abstracta, indica que os elementos da IU são especificados em uma linguagem de mais alto nível e, possivelmente, proprietária do MBUIDE; Concreta, indica que os elementos da IU são especificados diretamente através das primitivas do pacote gráfico a ser utilizado na aplicação. Se ambos estiverem presentes, indica que o MBUIDE suporta as duas formas indiscriminadamente.
- **Linha 4. Especificação das Tarefas:** O termo Abstata, indica que o Modelo de Tarefas utiliza estruturas de alto nível para ligar os elementos da aplicação; Concreta, indica que a MBUIDE faz as ligações do Modelo de Tarefas utilizando recursos da própria linguagem de programação e do pacote gráfico.
- **Linha 12. Suporte à edição dos modelos:** O termo **Textual**, indica que o desenvolvedor precisa alterar manualmente, e sem nenhuma assistência de correção, o código-fonte dos modelos (tal como editar um arquivo de texto); **Editores gráficos**, indica o uso do conceito WYSIWYG para manipulação dos modelos; **Formulários**, sugere o uso de caixas de diálogo para edição das propriedades do modelo; **Textual Assistida**, indica que a edição do modelo é feita com o suporte de assistentes de código baseados em contexto, capazes de sugerir e corrigir o trabalho do desenvolvedor (vide Figura 3.3 para mais detalhes).
- **Linha 10. Layout da IU:** O termo **Guidelines**, indica que o posicionamento dos elementos na IU é feito por padrões, mas ignora-se se estes podem ser alterados ou não; **Guidelines fixos**, indica que o posicionamento dos elementos na IU é fixo e baseado em determinados padrões imutáveis;

**Guidelines configuráveis**, indica que o posicionamento dos elementos na IU é fixo e baseado em determinados padrões, mas que esses podem ser alterados conforme a necessidade; **Templates configuráveis**, indica que o posicionamento dos elementos na IU é feito através de estrutura pré-definidas de tamanho, posição e aparência e que estes podem ser modificados conforme a necessidade; **Algoritmos**, indica que o posicionamento dos elementos na IU é realizado por algoritmos, os quais podem ser dirigidos por *guidelines* fixos, configuráveis ou por templates; **Histórico**, indica que o posicionamento dos elementos na IU é computado por algoritmos que se baseiam em informações históricas do contexto de desenvolvimento; **Manual**, indica que a ferramenta também suporta a definição explícita e manual de layout.

- **Linha 11. Processo de geração da IU (conforme Figura 2.4):** O termo **Runtime**, indica que a ferramenta utiliza um módulo Interpretador de Tempo de Execução; **Código**, indica que a ferramenta efetua a geração de código-fonte em tempo de projeto; **Código e Runtime**, indica que a ferramenta suporta as duas abordagens de geração da IU
- Ainda, células que mostram um travessão (–) sinalizam informações que não puderam ser identificadas no decorrer da pesquisa.

Tabela A.1: Tabela comparativa entre as características das MBUIDE.

	1. Local e ano	2. Modelos suportados	3. Especificação de IU	4. Especificação de Tarefas
<b>MERLIN</b>	Brasil, 2005	D, T, PUI	AC	C
<b>METAGEN</b>	Brasil, 2002	DT	A	C
<b>MECANO</b>	USA	D	C	A, C
<b>JANUS</b>	Alemanha	D	-	X
<b>GENIUS</b>	Alemanha	D, I	-	-
<b>UIDE</b>	USA	DTP	A	A
<b>TADEUS</b>	Alemanha, 1995	T, D, U, I	AC	A, C
<b>HUMANOID</b>	USA, 1995	L, P, I	AC	A
<b>MASTERMIND</b>	USA, 1996	T, D, P	C	A, C
<b>TRIDENT</b>	Bélgica, 1993	TP, D	A	A, C
<b>ADEPT</b>	Reino Unido, 1992	T, U	A	A
<b>AME</b>	Alemanha, 1996	D	A, C	C
<b>FUSE</b>	Alemanha, 1996	T, D, U, I	A	A
<b>TEALACH</b>	Reino Unido, 1997	D, P, T	AC	A
<b>ITS</b>	USA	D, P	-	-
<b>DRIVE</b>	1995	D, P, U	A	A



	5. Paradigma de desenvolvimento	6. Abrangência	7. Padrões utilizados na especificação dos modelos	8. Linguagem-fim
<b>MERLIN</b>	OO	Interfaces CRUD	Java Annotations	Java
<b>METAGEN</b>	Relacional	Interfaces CRUD	Arquivos texto	Visual Basic e Delphi/Kylix
<b>MECANO</b>	-	-	Ext. sobre C++	-
<b>JANUS</b>	OO	Interfaces CRUD	Ext. sobre CORBA IDL e ODMG ODL	C++
<b>GENIUS</b>	Relacional	Interfaces CRUD	Ext. sobre o modelo relacional	-
<b>UIDE</b>	OO	-	-	-
<b>TADEUS</b>	OO	-	OMT, TKS, Redes de Petri e ext. sobre HTA	-
<b>HUMANOID</b>	Relacional	Genérica	Proprietária	-
<b>MASTERMIND</b>	OO	Genérica	Ext. sobre CORBA IDL	C++
<b>TRIDENT</b>	Relacional	Interfaces CRUD	DSL, ACG e ext. sobre ER	-
<b>ADEPT</b>	Relacional	Interfaces CRUD	TKS, LOTOS e CSP	Smalltalk
<b>AME</b>	OO	Genérica	OMT	C++
<b>FUSE</b>	OO	Genérica	Álgebra, HTA, DAG, DFD e HTI	C++
<b>TEALLACH</b>	OO	Genérica	Árvore de objetos com estados	Java
<b>ITS</b>	-	-	ITS Style Rules	-
<b>DRIVE</b>	Relacional	Interfaces CRUD	NOODL	-

	9. Toolkit gráfico	10. Layout da IU	11. Processo de geração da IU	12. Suporte à edição dos modelos
<b>MERLIN</b>	Neutro	Guidelines configuráveis, Algoritmos, Manual e Histórico	Runtime	Textual assistida
<b>METAGEN</b>	MFC	Guidelines fixos	Runtime	Textual
<b>MECANO</b>	–	–	Runtime	Formulários
<b>JANUS</b>	ISA e Open Interface	Guidelines	Código	Editores gráficos
<b>GENIUS</b>	ISA	Guidelines	Código	Editores gráficos
<b>UIDE</b>	–	–	Runtime	–
<b>TADEUS</b>	ISA	Guidelines configuráveis	Código	Editores gráficos
<b>HUMANOID</b>	–	Templates configuráveis	Runtime	Editores gráficos
<b>MASTERMIND</b>	Garnt e Amulet	Guidelines configuráveis	Código e Runtime	Editores gráficos
<b>TRIDENT</b>	–	Guidelines configuráveis	Código	Editores gráficos
<b>ADEPT</b>	Open Look	Guidelines configuráveis	Código	Editores gráficos
<b>AME</b>	ISA e Open Interface	Guidelines configuráveis	Código e Runtime	Editores gráficos
<b>FUSE</b>	Athena e OSF/Motif	Templates configuráveis	Runtime	Editores gráficos
<b>TEALLACH</b>	Swing	Guidelines fixos	Código	Editores gráficos
<b>ITS</b>	–	Guidelines configuráveis	Runtime	Textual
<b>DRIVE</b>	–	Guidelines fixos	Runtime	Textual

## APÊNDICE B – ANOTAÇÕES DE CONFIGURAÇÃO DOS MODELOS DEFINIDAS PELA FERRAMENTA MERLIN

As anotações de configuração, construídas com base no padrão *Java Annotations* (2004b) são responsáveis por definir e interligar os modelos da ferramenta. Além de reutilizar padrões oriundos dos *frameworks* de apoio, a versão corrente da ferramenta define o seguinte conjunto de anotações:

Tabela B.1: Conjunto de anotações de configuração e propriedades definidas pela ferramenta MERLIN.

Anotação	Propriedade	Níveis	Função
@Agent			Anotação de cunho genérico, que se encarrega de especificar informações referentes aos Modelos de Apresentação, Diálogo e de Tarefas.
	action:String[]	AIO, CIO	Especifica o nome do método de negócio a se executado quando o evento associado ocorrer e a pré-condição for satisfeita. O conteúdo é o nome de um método de classe ou um mapeamento (atalho) para ele. Seu valor pode ser parcial (como “método”) ou completo (como “aplicação.pacote.classe.método”). No caso de ambigüidade em valores parciais, o conceito de “proximidade de métodos” é utilizado.
	event:String[]	AIO, CIO	Especifica o nome do evento de IU que, quando ocorrido, dispara a ação de negócio ou o <i>script</i> associado se a pré-condição for satisfeita. Pode ser tanto um valor dependente do <i>toolkit</i> gráfico corrente, ou um mapeamento abstrato para este.
	condition:String[]	CIO	Expressão OCL ou EL que define uma pré-condição para execução da ação indicada quando o evento especificado ocorrer.
	script:String[]	CIO	Fragmento de código em linguagem de <i>scripting</i> a ser executado quando o evento especificado ocorrer e a pré-condição for satisfeita. Os <i>scripting</i> suportados são dependentes do framework de <i>scripting</i> utilizado, sendo o BeanShell o padrão.
	property:String[]	AIO, CIO	Através seqüências no formato “nomeDaPropriedade=valorDaPropriedade” separadas pelo caractere ponto-e-vírgula, especificam atributos que devem ser aplicados diretamente nos controles de IU.
	binder:Class	CIO	Especifica uma classe de conversão de valores entre o objeto de dado e o controle de IU.

Anotação	Propriedade	Níveis	Função
			Embora o framework possua um conjunto essencial de conversores, esta propriedade permite especificar um valor alternativo.
@Order			Define o posicionamento do controle de IU gerado. A interpretação dessa anotação é subjetiva, uma vez que conceitos como “antes” ou “depois” dependem do fluxo de preenchimento adotado pelo algoritmo de posicionamento atrelado.
	value:int	CIO	Indica a posição ordinal, de 0 a N (onde N é o número de controles na IU) onde deve ser inserido o elemento anotado.
	after:String	CIO	Indica o nome do controle de IU imediatamente anterior ao elemento anotado.
	before:String	CIO	Indica o nome do controle de IU imediatamente posterior ao elemento anotado.
@RenderAs			Especifica o tipo de controle de IU a ser renderizado na tela quando o tipo de controle inferido pelo software não for adequado. Por exemplo, um dado booleano, que, por padrão é renderizado com uma caixa de marcação pode ser configurado com essa anotação para ser exibido com uma árvore de dois valores (verdadeiro e falso), por exemplo.
	value:Class	AIO, CIO	Define a classe do componente de IU a ser utilizado para exibição do valor contido no elemento anotado. Podem ser utilizados valores dependentes do <i>toolkit</i> gráfico em uso (como <code>JTextField.class</code> , no caso do pacote Swing) ou um valor abstrato (como <code>TEXT_FIELD.class</code> ) que é mapeado pelo software para o tipo mais adequado frente à biblioteca gráfica em uso.
	binder:Class	CIO	VIDE <code>@Agent.binder</code>
@Caption			Define mensagens de texto a serem exibidas, tanto na tela em si (título e área de mensagens), quanto para os controles de IU. Todas as mensagens exibidas têm suporte à internacionalização através dos mecanismos nativos utilizados pelo sistema.
	value:String	CIO	Mensagem de texto a ser exibida sobre o elemento anotado. Na prática, para controles de IU é o texto descritivo ( <i>label</i> ) associado; se utilizado sobre a classe, é o texto descritivo do formulário (título). Se a o valor for envolvido por aspas simples, então este é considerado uma chave que deve ser localizada no arquivo de recursos do sistema, de forma a obter a mensagem de texto relacionada (útil para internacionalização ou externalização de mensagens). Se o valor contiver o fragmento “;tip=”, então toda a parte à direita deste é considerado uma mensagem de auxílio (conhecida como <i>Tooltip</i> ou <i>Hint</i> ) a ser exibida quando o mouse é posicionado sobre o controle de IU.
@Dependence			Utilizada para criar dependências entre elementos de IU. É útil, por exemplo, quando um determinado controle somente é visível ou habilitado se outro controle for preenchido corretamente ou se alguma regra de negócio for satisfeita.
	value:String	CIO	Separados por ponto-e-vírgula, indicam os controles de IU afetados pela lógica de

Anotação	Propriedade	Níveis	Função
			dependência implementada. Por padrão as lógicas de habilitação e visibilidade estão disponíveis para controles de IU simples, como caixas de marcação e caixas de texto. Alguns atalhos podem ser utilizados nessa propriedade, como “:all”, que indica que todos os controles da IU (exceto o elemento anotado) são afetados pela lógica de dependência.
	opposite:Boolean	CIO	Indica se a lógica de dependência deve ser invertida. Por exemplo, o padrão para uma dependência entre uma caixa de marcação e um outro controle de IU é que, ao “marcar” a caixa, o controle dependente é habilitado; indicar “opposite=true”, faz com que ao “marcar” a caixa, o controle dependente é desabilitado.
	action:Class	AIO, CIO	Define a classe que implementa a lógica de dependência. Por padrão, a lógica de dependência é a habilitação/desabilitação de controles de IU. A lógica de visibilidade/invisibilidade também pode ser utilizada como parte da ferramenta. Outras lógicas podem ser implementadas e anexadas ao gerador, como, por exemplo, a habilitação/desabilitação de controles de IU baseados na execução de regras de negócio diversas.
@Alias			Aplicado sobre classes e métodos, indica um nome de atalho para ser utilizado pelo restante do sistema. Isso é útil para definir nomes mais significativos ou menores para os elementos da aplicação
	value:String	CIO	Indica o nome do atalho. Todos os atalhos são armazenados pelo mecanismo de geração na forma de listas invertidas, as quais são utilizadas pelas outras anotações do sistema para localizar os elementos. Na eventualidade de colisões ocorrerem, o conceito de proximidade entre o atalho e o item anotado é aplicado.
@Ignore			Utilizado também como um atalho para evitar a criação de um Modelo de Usuário, essa anotação indica que um determinado atributo não deve ser renderizado na IU.
@Group			Indica que os controles de IU devem ser agrupados quando mostrados na tela. Por ser utilizada sobre uma classe ou sobre suas propriedades. Na prática, elementos agrupados serão exibidos tipicamente em painéis com ou sem abas, dependendo do tipo de renderizador associado. Por padrão, o renderizador de painéis é utilizado.
	value:String	AIO, CIO	Separados por ponto-e-vírgula, indicam quais elementos fazem parte do agrupamento. Esse valor é descartado no caso da anotação ser utilizada diretamente sobre uma propriedade. Este valor pode apontar para outro agrupamento, formando subagrupamentos.
	name:String	CIO	Indica o texto que representa o nome do agrupamento, também suportando internacionalização através do uso do prefixo “id=” e a exibição de um texto de auxílio com o uso do fragmento “:tip=” (como em

Anotação	Propriedade	Níveis	Função
			@Caption.value). O nome do agrupamento é mostrado ou não dependendo do tipo de renderizador utilizado.
	in:String	AIO, CIO	Utilizado para criar subagrupamentos, aponta para o nome do agrupamento superior.
	render:Class	AIO, CIO	Indica o nome da classe de renderização do agrupamento. Por padrão, no Swing, o renderizador é um JPanel. Outras classes, como um JTabbedPane ou uma classe personalizada podem ser utilizados sem problemas.

