DISSERTAÇÃO DE MESTRADO

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM

CIÊNCIA DA COMPUTAÇÃO

"Uma Abordagem para Migração Automática de Código no Contexto do Desenvolvimento Orientado a Modelos"

ALUNO: Marcos Antonio Possatto

ORIENTADOR: Prof. Dr. Daniel Lucrédio

São Carlos
2014

CAIXA POSTAL 676
FONE/FAX: (16) 3351-8233
13565-905 - SÃO CARLOS - SP
BRASIL

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UMA ABORDAGEM PARA MIGRAÇÃO AUTOMÁTICA DE CÓDIGO NO CONTEXTO DO DESENVOLVIMENTO ORIENTADO A MODELOS

MARCOS ANTONIO POSSATTO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software
Orientador: Prof. Dr. Daniel Lucrédio

São Carlos - SP
2014

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

P856am

Possatto, Marcos Antonio.
Uma abordagem para migração automática de código no contexto do desenvolvimento orientado a modelos / Marcos Antonio Possatto. -- São Carlos : UFSCar, 2014.
119 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2013.

1. Engenharia de software. 2. Engenharia ida-e-volta. 3. Geração de código. 4. Implementação. I. Título.

CDD: 005.1 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Uma Abordagem para Migração Automática de Código no Contexto do Desenvolvimento Orientado a Modelos"

Marcos Antonio Possatto

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação

Membros da Banca:

Daniel Lucrédio

Prof. Dr. Daniel Lucrédio
(Orientador - DC/UFSCar)

Prof. Dr. Antonio Francisco do Prado
(DC/UFSCar)

Prof. Dr. Marcelo de Almeida Maia
(UFU)

São Carlos
Outubro/2013

AGRADECIMENTO

Agradeço,

ao meu orientador, professor Daniel, por acreditar em mim e ter dado a oportunidade de trabalhar neste projeto, pela competência científica e acompanhamento do trabalho;

à agência FAPESP, que apoiou parcialmente o desenvolvimento deste trabalho;

ao professor Prado, pela participação que teve em todas as etapas deste trabalho, inclusive marcada por incentivos e conselhos importantes;

ao professor Marcelo Maia, especialista em assuntos relevantes à este trabalho, que nos presenteou com sua participação na banca examinadora;

ao professor Valter, pelas sugestões e correções apontadas no exame de qualificação, que foram importantes para a realização deste trabalho;

aos que contribuíram com o experimento: os alunos pela participação, o colega Evelton, por disponibilizar os laboratórios e os equipamentos necessários e o professor Jorge Oishi, pela orientação no tratamento estatístico dos dados;

aos demais professores, colegas de mestrado e amigos do DC pelo convívio e aprendizado;

aos companheiros de trabalho da ProAd, que supriram as minhas ausências e me apoiaram nesta jornada;

a meus pais Geraldo e Cilia, pela valorização e incentivo;

à minha esposa Carla e meus filhos Beto e Giovana, pelo carinho, compreensão e encorajamento, durante todo este período;

a Deus por me abençoar em todos momentos desta vida.

RESUMO

Os geradores de código desempenham um papel fundamental no desenvolvimento de software orientado a modelos. São responsáveis pela transformação dos artefatos de alto nível de abstração (modelo) em elementos de implementação (código). Os tipos mais comuns de geradores são os baseados em *template*. São compostos fundamentalmente por elementos de expansão de código, que recebem uma entrada e a convertem em código, conforme a programação inserida nesses *templates*. O código de uma implementação já testado e validado pode servir de referência para a criação de *templates*, por meio de um processo conhecido como migração de código. Com a dinâmica da evolução do software e a necessidade de efetuar mudanças no gerador de código ocorre a perda de sincronismo entre os *templates* e esse código de referência, sendo necessário um esforço adicional para mantê-los sincronizados. O desafio de manter esses artefatos sincronizados constituiu o objetivo desta dissertação de mestrado, que proporcionou ganhos de produtividade, por meio de uma automação, ainda que parcial, desse processo. Nesse sentido, foi desenvolvido um mecanismo para propagar automaticamente as alterações introduzidas no código de referência para os *templates*, que reduziu o tempo e facilitou a manutenção de geradores de código que sofrem com o problema da perda de sincronismo nesses artefatos. O protótipo para a migração automática de código desenvolvido nesta dissertação foi submetido a um estudo empírico, atingindo um bom desempenho com a sua utilização na maioria das tarefas de migração de código avaliadas, o que indica que a automação pode ajudar a resolver o problema e reduzir o esforço de manutenção no desenvolvimento de software orientado a modelos.

Palavras-chave: Desenvolvimento Orientado a Modelos, Engenharia Ida-e-Volta, Geração de Código Baseada em Templates, Implementação de Referência, Migração de Código

ABSTRACT

Code generators play a key role in model-driven software development. They are responsible for transforming high-level assets (models) into implementation assets (code). Most generators are based on templates, which are pieces of text instrumented with code expansion elements. They receive an input and produce an output according to the template's programming. To build such template-based generators, the code of an existing implementation, already tested and validated, can be used as a reference, in a process known as code migration. With software evolution and the need for changes in the code generator, the templates start to differ from this reference implementation. In order to restablish the synchronization, additional effort is required. Tackling the challenge of keeping these assets synchronized (reference implementation and templates) is this dissertation's subject. The goal is to provide some automation to the code migration process, even if partial, in order to increase productivity in the maintenance of code generators. A mechanism was developed to make it possible to automatically reproduce changes that are performed in the reference implementation into one or more code generation templates. This mechanism was evaluated through an empirical study, yielding good performance in a controlled environment. This indicates that automation can help to reduce the effort in the maintenance of code generators in a model-driven development context.

Keywords: Model-Driven Development, round-trip engineering, template-based code generation, reference implementation, code migration

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AST - *Abstract Syntax Tree* / Árvore de Sintaxe Abstrata

DOM - Document Object Model / Modelo de Objetos de Documentos

DSL - *Domain-Specific Language* / Linguagem Específica de Domínio

DSM - *Domain Specific Model* / Modelo Específico de Domínio

EER - *Extended Entity-Relationship* / Entidade-Relacionamento Estendido

EMF - *Eclipse Modeling Framework* / Framework de Modelagem do Eclipse

GMF - Graphical Modeling Framework / Framework de Modelagem Gráfica

HTML - Hypertext Markup Language / Linguagem de Marcação de Hipertexto

JET - *Java Emitter Templates* / Modelos Emissores de Java

JSP - *Java Server Pages* / Páginas Servidoras Java

MDA - *Model-Driven Architecture* / Arquitetura Orientada a Modelos

MDCB - *Model-Driven and Component-Based* / Orientada a Modelos e Baseado em Componentes

MDD - *Model-Driven Development* / Desenvolvimento Orientado a Modelos

MDE - *Model-Driven Engineering* / Engenharia Orientada a Modelos

MVC - *Model-View-Controller* / Modelo-Visão-Controlador

PIM - *Platform Independent Model* / Modelo Independente de Plataforma

PSM - *Platform Specific Model* / Modelo Específico de Plataforma

RTE - *Round-Trip Engineering* / Engenharia Ida-e-Volta

SAX - *Simple API for XML* / API Simples para XML

UML - *Unified Modeling Language* / Linguagem de Modelagem Unificada

UnQL - *Unstructured Query Language* / Linguagem de Consulta não Estruturada

XML - *eXtensible Markup Language* / Linguagem de Marcação Extensível

XSLT - *eXtensible Stylesheet Language for Transformation* / Linguagem de Estilos Extensível para Transformações

XVCL - *XML-based Variant Configuration Language* / Linguagem Configuração Variante Baseada em XML)

SUMÁRIO

Capítulo 1

INTRODUÇÃO

O Desenvolvimento Orientado a Modelos, ou MDE (*Model-Driven Engineering*) é a combinação de programação generativa, linguagens específicas de domínio e transformações de software, que já vem sendo exploradas desde a década de 1980 (NEIGHBORS, 1980; LUCRÉDIO et al., 2006). Seu objetivo é reduzir a distância semântica entre o problema e implementação/solução, através de modelos de alto nível que protegem os desenvolvedores das complexidades da plataforma de implementação (FRANCE; RUMPE, 2007). Em MDE, modelos são usados para se expressar conceitos do domínio de forma mais efetiva, enquanto transformações geram automaticamente os artefatos que refletem a solução expressa nos modelos (SCHMIDT, 2006).

Neste contexto, geradores de código desempenham um papel essencial, pois são eles os responsáveis por traduzir as especificações em alto nível (modelos) em trechos de código executável. Podem agregar também uma série de benefícios à atividade de desenvolvimento de software: Geradores produzem sempre o mesmo código, de forma previsível, controlada e com menos erros. Por exemplo, um gerador pode automaticamente aplicar padrões de código de forma sistemática, com menor taxa de erros (CLEAVELAND, 1988). Geradores podem realizar tarefas extremamente difíceis para um ser humano. Por exemplo, a implementação manual de uma máquina de estados pode se tornar extremamente complexa, especialmente quando se faz necessário verificar milhares de transições que acabam ficando implícitas no código. Otimizações diversas, como remoção de redundâncias, também podem ser realizadas automaticamente pelo gerador. Além disso, geradores

realizam tarefas repetitivas mais rapidamente. Em outro exemplo, pode-se citar a criação manual de métodos acessores (*set* e *get*) em todas as classes de um sistema de grande porte, ou mesmo realizar refatorações (FOWLER et al., 1999) simples, como extração de métodos ou renomeação de classes, de forma manual. Tais tarefas podem se mostrar extremamente demoradas se feitas manualmente.

Porém, o elemento humano não pode ser descartado completamente. O desenvolvimento de software envolve um alto grau de criatividade e, portanto o uso de geradores precisa ser planejado de forma a potencializar as habilidades naturais do ser humano, ao mesmo tempo em que tarefas repetitivas e tediosas podem ser retiradas de sua responsabilidade. Quando adotada corretamente, a geração de código pode levar a ganhos importantes em termos de produtividade e qualidade (TOLVANEN; KELLY, 2001).

O casamento ideal entre especificação (trabalho intelectual, humano) e automação (trabalho braçal, mecânico) na engenharia de software é o ponto crítico no projeto de um gerador, sendo alvo das principais pesquisas na área (MERNIK; HEERING; SLOANE, 2005; CLEAVELAND, 1988; PAUL, 2005; HESSELLUND; CZARNECKI; WASOWSKI, 2007; JOUAULT; BÉZIVIN; KURTEV, 2006; HAMZA, 2005; KORHONEN, 2002; WILE, 2004). Entre as dificuldades para se projetar um gerador, destacam-se a necessidade de conhecimentos diversos para se projetar a linguagem de especificação, tais como conhecimentos de projetista, engenheiro, autor de manuais e treinamento, especialista do domínio, especialista em linguagens, entre outros; a dificuldade em se reconhecer os limites do domínio; a integração entre múltiplas linguagens de especificação; e a reutilização, evolução e manutenção dos artefatos para geração de código. Essa última é foco deste trabalho.

Pesquisadores dessa área (MERNIK; HEERING; SLOANE, 2005; FEILKAS, 2006) citam diversas abordagens para implementar um gerador, desde o uso de diretrizes de pré-processamento, a construção de interpretadores dedicados, até o uso de *templates*, padrões ou a chamada abordagem visitante (*visitor approach*), que consiste em percorrer o modelo segundo alguma ordem específica, executando transformações para cada elemento visitado (FEILKAS, 2006). Tecnologias XML (*eXtensible Markup Language*), tais como XSLT (*eXtensible Stylesheet Language for Transformation*), e abordagens baseadas em DOM (*Document Object Model*) e SAX

(*Simple API for XML*) também são opções (CLEAVELAND, 2001). Dentre estas, o uso de *templates* é a abordagem preferida para geradores mais robustos (CLEAVELAND, 2001; FEILKAS, 2006).

Um *template* é um arquivo de texto qualquer instrumentado principalmente com construções de seleção e expansão de código (CZARNECKI; EISENECKER, 2000). Estas construções são responsáveis por realizar consultas em uma entrada, que pode ser um programa ou especificação textual, um conjunto de janelas e diálogos, no estilo *wizard*, ou mesmo diagramas (CLEAVELAND, 1988). A informação resultante destas consultas é então utilizada como parâmetro para produzir código personalizado, em qualquer linguagem textual.

Porém, construir *templates* a partir do zero não é uma tarefa fácil, pois um *template* mistura código gerado com código para geração, o que acaba dificultando o processo. Neste sentido, Cleaveland (1988) destaca a importância da existência de uma aplicação-exemplo como base para construção de geradores. Ao invés de se projetar um gerador a partir do zero, esse autor sugere analisar código já existente em busca de padrões que se repetem, com o objetivo de identificar possibilidades de emprego de um gerador.

Em outro trabalho (BIERHOFF; LIONGOSARI; SWAMINATHAN, 2006), pesquisadores relatam o sucesso de uma abordagem que se baseia em uma aplicação típica como ponto de partida para o desenvolvimento de geradores. A partir da aplicação típica, trechos de código são movidos para um gerador, identificando quais desses trechos se repetem e como eles variam entre si. Com base nessa informação, projeta-se uma linguagem capaz de especificar esta variabilidade, e um gerador que utiliza estas informações como parâmetro para a geração de código. Com a evolução do processo, novas alternativas de variação são identificadas e o gerador evolui até ser capaz de atender a uma maior variedade de soluções. Abordagens similares são relatadas na literatura (VISSER, 2007) e (WIMMER et al., 2007).

Na literatura encontramos o termo “Implementação de Referência” para se referir a uma aplicação-exemplo (MUSZYNSKI, 2005). Neste trabalho, adotatemos essa denominação para identificar o código base que é inserido no gerador de código, inclusive evidenciando a implementação de referência como fonte central de informações para o processo de desenvolvimento de geradores.

As vantagens do desenvolvimento de geradores guiado pela implementação de referência são (MUSZYNSKI, 2005):

- Permite a geração e reutilização de maiores quantidades de código, pois se reaproveita qualquer trecho de código pré-existente;
- A inclusão incremental das anotações facilita a criação de geradores não triviais. Produz geradores com maior qualidade, pois a implementação de referência é pré-testada, depurada e validada da maneira tradicional;
- Permite utilizar o ambiente de preferência do desenvolvedor para a construção da implementação de referência;
- Possibilita que o desenvolvedor utilize funcionalidades que normalmente são pertinentes à atividade de desenvolvimento de software, como *auto-completion*[1], refatorações, além de outras que são fortemente associadas com código-fonte e não funcionam no nível do gerador; e
- Agiliza o processo de desenvolvimento e evolução do gerador. Isto porque mudanças estão fortemente associadas com o processo de depuração durante a execução, que não funciona no nível do gerador. A agilidade no desenvolvimento de geradores é também um requisito importante (CLEAVELAND, 1988).

1.1 Motivação e Caracterização do Problema

Os sistemas de software necessitam evoluir, e o desenvolvimento baseado em mecanismos MDE não é uma exceção (DEURSEN, VISSER, WARMER, 2007). A evolução do software engloba todas as atividades relacionadas ao seu ciclo de vida, desde a concepção inicial até a fase de manutenção. É necessário contar com processos que minimizem os custos de manutenção e evolução durante o ciclo de vida do desenvolvimento do software.

[1] Recurso de um ambiente de programação que sugere possíveis entradas com base no contexto do cursor e em alguns caracteres digitados pelo programador

A evolução dos sistemas de software constitui um problema fundamental. Se a geração de código da aplicação não for completa os desenvolvedores terão que fazê-la de forma manual. A evolução simultânea do código, dos modelos e dos *templates* introduzem a necessidade de realizar alterações, acarretando a perda de sincronismo entre esses elementos. Devido às consequências da dinâmica de evolução do software, os geradores de código, para continuarem a produzir aplicações consistentes, devem incorporar as mudanças, restabelecendo novamente a sincronização existente entre modelo, *templates* e código gerado.

O gerador de código baseado em *templates* pode ser construído a partir de uma implementação de referência e, essa técnica, como já discutido, apresenta diversas vantagens para a produção de aplicações correspondentes, através da variação de entradas para a geração de código. Porém, também não estão isentos dos problemas ocasionados por mudanças nesses artefatos, que são necessárias após a primeira sincronização dos *templates* com a implementação de referência.

Para melhor caracterizar o problema, a Figura 1.1 ilustra o ciclo básico do desenvolvimento orientado a modelos no contexto deste projeto, isto é, com a utilização de uma implementação de referência e *templates* para geração de código.

O processo está centrado nas figuras do engenheiro de domínio e do desenvolvedor de aplicações. O engenheiro de domínio é responsável por, em um primeiro momento, produzir uma infraestrutura reutilizável e configurável, dentro de um determinado domínio de aplicações, para que em um segundo momento o desenvolvedor de aplicações possa construir aplicações daquele domínio mais facilmente (CZARNECKI; EISENECKER, 2000). Esses dois momentos são representados pelo lado esquerdo e direito da Figura 1.1, separados pela linha tracejada.

No contexto deste projeto, o engenheiro de domínio é responsável pela construção e manutenção da implementação de referência, e da migração de seu código para um ou mais *templates* de geração de código. O desenvolvedor de aplicações utiliza os *templates* em conjunto com modelos do domínio - especificações da aplicação sendo construída - para gerar o código final.

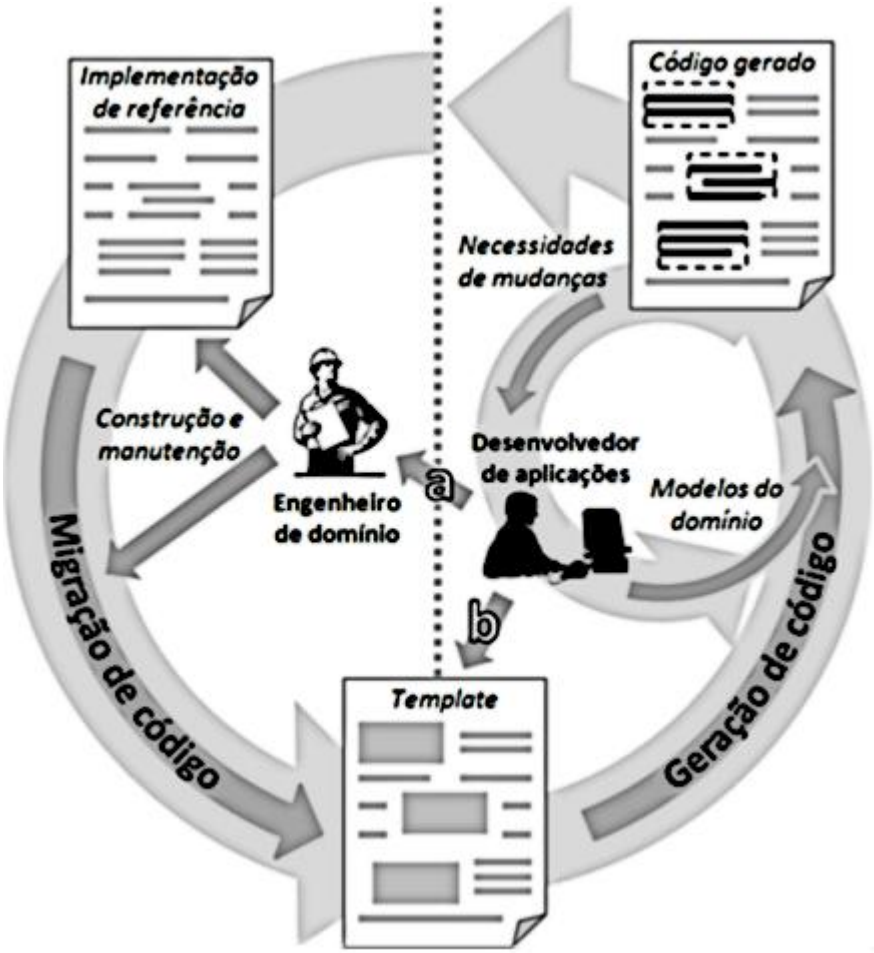

Figura 1.1 - Contexto deste projeto: ciclo do desenvolvimento orientado a modelos

Em uma primeira iteração desse ciclo, os artefatos refletem um conjunto inicial de requisitos do domínio. Isto significa que o desenvolvedor é capaz de produzir e manter as aplicações apenas modificando a entrada dos geradores, isto é, os modelos do domínio. Este miniciclo é representado no lado direito da Figura 1.1. Porém, uma determinada necessidade de mudança em uma ou mais aplicações pode requerer mudanças nos *templates* de geração de código. A implementação de tais mudanças pode seguir dois caminhos, representados pelas letras “a” e “b” na Figura 1.1, ambos com problemas:

No caminho representado pela letra “a”, antes de realizar alterações nos *templates*, as mesmas são realizadas, testadas e validadas na implementação de referência. Em seguida, uma nova migração de código é necessária para que os *templates* reflitam a nova implementação. Este é o caminho ideal, conforme discutido na seção anterior. Porém, a migração de código consome cerca de 20-25%

do tempo de desenvolvimento da implementação de referência (VOELTER; BETTIN, 2004). Além disso, causa a duplicação de código entre a implementação de referência e os *templates*, o que requer cautela para manutenção da consistência entre estes dois artefatos (MUSZYNSKI, 2005).

Esses problemas, principalmente o custo e esforço extras, acabam às vezes inibindo a execução do ciclo completo. A alternativa é realizar as alterações necessárias diretamente nos *templates* (caminho 'b'). Porém, conforme discutido anteriormente existe uma série de dificuldades em se trabalhar no nível dos *templates* de geração. Por se tratar de um artefato que se situa em um meta-nível acima, o processo de alteração, testes e depuração é dificultado, pois envolve uma etapa extra de compilação. Além disso, *templates* são artefatos complexos, pois misturam o código que implementa a lógica das aplicações do domínio com código específico para geração. Com isto, muitas vezes é difícil localizar os trechos onde as mudanças devem ser efetuadas. Finalmente, as ferramentas atuais para edição de *templates* não oferecem as mesmas facilidades que os editores de código tradicionais, tais como ajudas para construção, inspeção e depuração de código.

Estes problemas impõem uma barreira à criação e manutenção de uma infraestrutura baseada em modelos para desenvolvimento de software. É verdade que com o tempo, à medida que o domínio amadurece e se estabiliza menos alterações são necessárias nos artefatos de geração de código. Porém, até esse amadurecimento, podem ocorrer problemas como o não cumprimento de prazos devido aos esforços extras necessários ou mesmo a falta de credibilidade na tecnologia. Caso o retorno do investimento não seja compensador, o que irá provavelmente acontecer é o abandono do esforço em direção à adoção do MDE como paradigma de desenvolvimento.

1.2 Objetivo

Para refletir as mudanças necessárias no código da aplicação os *templates* do gerador precisam ser alterados. Dois caminhos são possíveis para efetuar essa correção (Figura 1.1), sendo implementá-la diretamente nos *templates* (caminho 'b')

ou na implementação de referência e migrá-la para os *templates* (caminho ‘a’). O caminho ‘b’ é difícil de ser executado, principalmente pela complexidade da programação embutida nos *templates*, além de acarretar a descontinuidade do mecanismo introduzido pela implementação de referência na construção inicial do gerador, que passa a não ter mais sincronia com o gerador. O caminho ‘a’ é o ideal, pois é mais fácil de ser executado e mantém atualizados os *templates* com a implementação de referência, porém são necessários custos adicionais e esforços específicos para continuar mantendo esse sincronismo.

Dessa forma, o objetivo deste trabalho é a busca de mecanismos que permitam redução de custos a partir de uma automação parcial do processo de migração de código, visando obter o sincronismo entre código e *templates*. A Figura 1.2 ilustra esse objetivo.

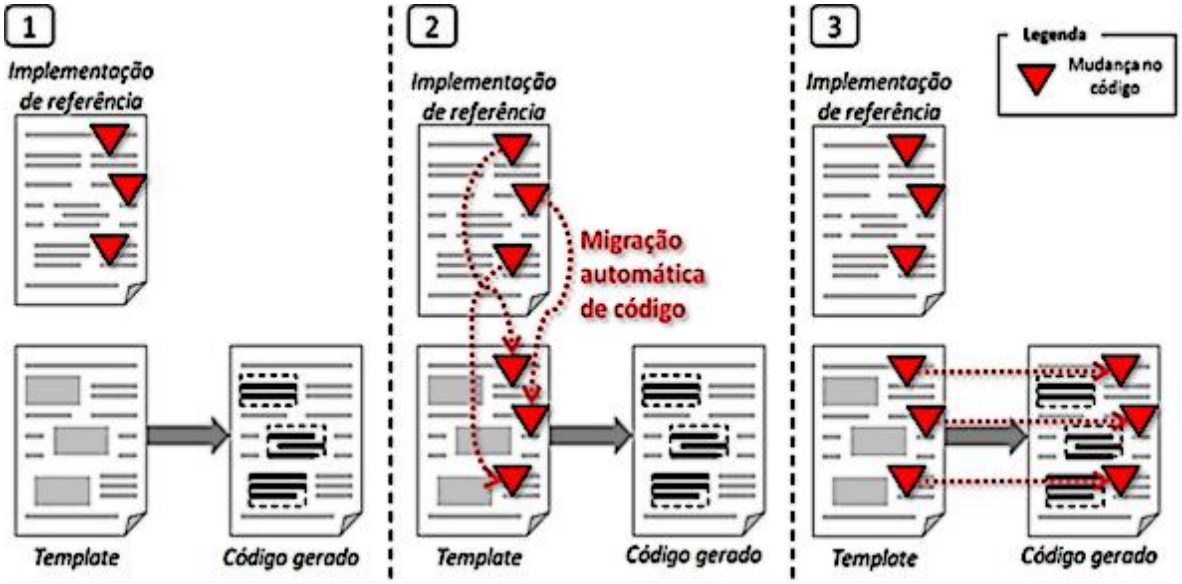

Figura 1.2 - Objetivo do projeto: Migração automática de código

Uma vez que o código da implementação de referência tenha sido migrado para um *template*, eventuais mudanças e alterações no código são realizadas diretamente na implementação de referência (1). Estas mudanças são migradas automaticamente para o *template* correspondente (2); A partir deste momento, toda geração de código passa a incorporar as mudanças (3). Com isso, é possível reduzir o esforço de migração.

Dessa forma, espera-se contribuir para a resolução dos problemas citados na seção anterior. A contribuição esperada é a redução do custo e esforços extras necessários para a migração de código. Ainda que a automação seja apenas parcial, ou seja, mesmo que o mecanismo de migração automática opere em determinadas mudanças no código de referência e os *templates* do gerador permaneçam editáveis, para suportar extensões manuais que não serão automatizadas neste projeto, qualquer redução pode levar a benefícios consideráveis para o desenvolvedor. Em particular, espera-se que esses custos sejam reduzidos a ponto de não inibir a realização do ciclo completo de manutenção passando pela implementação de referência antes da atualização dos *templates* (caminho “a” da Figura 1.1, considerado como ideal).

1.3 Definição do Escopo

Esta dissertação tem como objetivo específico o desenvolvimento de um mecanismo para propagar automaticamente determinadas alterações que são introduzidas na implementação de referência para os *templates* do gerador de código, visando manter esses dois artefatos sincronizados e, com isso, facilitar e reduzir o tempo que é despendido com a manutenção em geradores.

O paradigma de programação adotado para a realização deste projeto é o desenvolvimento baseado em modelos, associado especificamente à estratégia de migração do código de implementações de referência atrelada à dinâmica de produção de geradores de código baseados em *templates*.

Entretanto, no contexto deste projeto também estão correlacionados seguimentos e extensões que, embora implícitos à abordagem utilizada, por conta do limite de tempo estabelecido para projetos em nível de mestrado, ficaram fora do escopo deste trabalho, sendo alguns relacionados a seguir:

- A análise da aplicação da abordagem de migração de código em linhas de produto de software:

  Geradores de código possibilitam a automação das partes variáveis de um determinado produto, sendo amplamente empregados na

construção de múltiplos produtos de uma família de software. Assim, as alterações que são realizadas no código de referência em determinado produto, podem estender-se para os demais da linha de produto, conforme a programação introduzida no gerador. Essa e outras relações pertinentes às linhas de produtos de software e associadas à abordagem empregada não foram estudadas neste trabalho.

- A construção inicial de geradores de código a partir do zero:

  Adotou-se como pressuposto a existência de um gerador em funcionamento para executar atividades de manutenção no código de referência, no mínimo na fase inicial de desenvolvimento, inclusive com uma estrutura que permita extrair as referências e refletir os mapeamentos a partir de posições pertinentes aos artefatos já estabelecidos. Contudo, o desenvolvimento inicial do gerador de código não foi estudado neste trabalho, o que deixa explícito que somente serão praticáveis manutenções de continuidade no gerador, ou seja, alterações de incremento e/ou correção no código de referência, não sendo possíveis alterações radicais que alterem a concepção inicial do gerador.

- A atualização dos modelos de entrada a partir de modificações introduzidas no código da aplicação:

  O código gerado resultante do processamento do gerador de *templates* carrega as informações obtidas dos modelos de entrada, inclusive produzindo as correspondentes variações que são inseridas nesses modelos. Porém, essa automação não foi desenvolvida neste projeto, somente foi estabelecido um controle para a ocorrência desse tipo de alteração, visando preservar os *templates* de geração para não serem danificados.

Outros detalhes específicos sobre o escopo da pesquisa, em termos de técnicas e mecanismos que foram empregados, inclusive os limites e as restrições da automação obtida durante o desenvolvimento efetivo do projeto, encontram-se no Capítulo 3.

1.4 Organização do Trabalho

Neste primeiro capítulo apresentou-se o contexto da abordagem de migração de código, a motivação, a caracterização do problema e os objetivos da pesquisa. O restante da dissertação foi estruturado da seguinte forma:

No Capítulo 2 discutem-se as referências importantes para a elaboração e compreensão do projeto proposto, incluindo geradores de código baseados em *templates*, implementação de referência, técnica de migração de código, tipos de mapeamento e pontos de sincronismos.

No Capítulo 3 é detalhado o desenvolvimento do protótipo para migração automática de código, inclusive os esforços iniciais para resolver o problema e os mecanismos efetivamente utilizados em sua elaboração.

No Capítulo 4 é apresentado o experimento que avaliou a utilidade do protótipo no processo de migração de código.

No Capítulo 5 são apresentados alguns trabalhos relacionados.

Por fim, no capítulo 6 estão algumas conclusões, incluindo contribuições e limitações deste trabalho, além das possibilidades de trabalhos futuros.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os principais conceitos que servem de embasamento para este trabalho. Considerando que o processo proposto tem como objetivo automatizar a migração de código, são apresentados os principais conceitos relacionados a esse processo, incluindo a técnica de geração de código baseada em *templates* (Seção 2.2), implementação de referência (Seção 2.3), os tipos de mapeamento e a sincronização dos artefatos (Seção 2.4).

2.1 Técnica de Geração de Código Baseada em *Templates*

O gerador de código constitui um componente importante do MDE, que por sua vez, tem por objetivo escrever um sistema de software usando modelos abstratos e refinar esses modelos em código. A técnica baseada em *templates* é uma maneira prática de gerar conteúdo textual a partir de modelos, o que a torna a mais difundida na transformação de modelo em texto (ANGYAL; LENGYEL; CHARAF, 2008).

A Figura 2.1 apresenta a técnica de geração de código baseado em *templates*. O Modelo contém dados organizados em uma estrutura específica com informações sobre os elementos do sistema; os *Templates* representam a sintaxe do alvo da linguagem e contém as referências dos elementos do modelo; e o Gerador é o aplicativo responsável pela realização da transformação dos artefatos. O gerador interpreta o modelo do projeto aplicando um conjunto de *templates* para produzir o

código de saída, obtido através da substituição das referências internas dos *templates* pelos dados provenientes do modelo, esse é o Alvo, resultado do processo de transformação, e que normalmente é composto de um ou mais arquivos textuais.

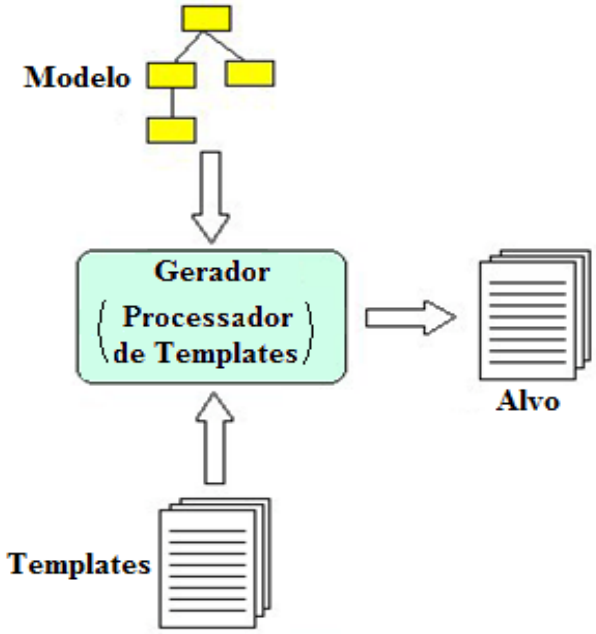

Figura 2.1 - Esquema do gerador de código baseado em *templates*

Para ilustrar esse processo, considere-se o seguinte cenário: uma empresa deseja agilizar a produção de formulários HTML (*Hypertext Markup Language*) para incluir em sua aplicação Web. Existem centenas de formulários distintos, e os campos de cada formulário mudam constantemente, de forma que a manutenção desse conjunto de páginas é normalmente trabalhosa.

Uma solução que utiliza geração de código baseada em *templates* contempla os quatro elementos citados anteriormente: o Modelo, que segue um formato de entrada específico para o problema, permite que um desenvolvedor especifique o conteúdo de cada formulário, tais como os campos, tipo de dados, tamanho, etc. Dentre os possíveis formatos, o XML apresenta vantagens pela facilidade no processamento das informações. Bibliotecas com suporte a linguagens como XPATH (W3C, 1999) facilitam o trabalho de consulta às informações.

O *Template*, neste caso, é uma página HTML, anotada com código que consulta o modelo. O terceiro elemento é o Gerador, um processador de *templates*,

responsável por instanciar o *template* com base na entrada. E o Alvo é uma página HTML gerada conforme especifica o modelo. A Figura 2.2 ilustra esse processo, onde cada trecho do *template* é processado para consultar o arquivo de entrada e produzir o código correspondente.

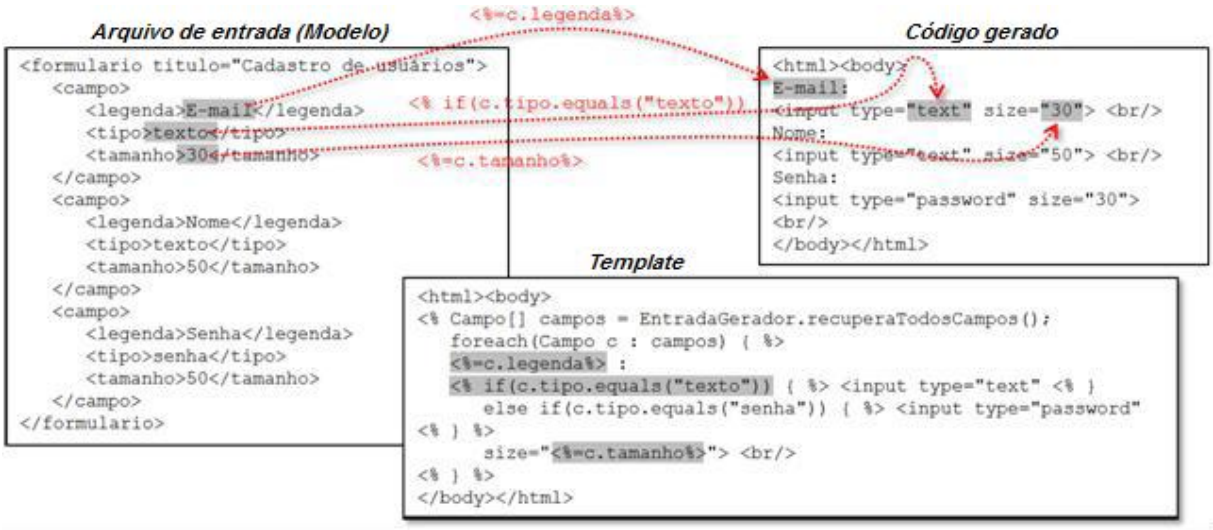

Figura 2.2 - Geração de código baseada em templates

Entre os benefícios dessa técnica, destacam-se a facilidade de adoção por parte de desenvolvedores, uma vez que esta abordagem permite gerar código em qualquer linguagem ou formato (CZARNECKI; EISENECKER, 2000). Além disso, enquanto outras abordagens produzem código-fonte que normalmente não seria escrito à mão (CLEAVELAND, 1988), o uso de *templates* propicia a geração de código mais legível, o que é crítico em muitos casos (CZARNECKI; EISENECKER, 2000). Isto porque o *template* é parecido com a saída desejada, acrescido de anotações e instruções que realizam a consulta na entrada e produzem a saída desejada (CLEAVELAND, 2001).

2.2 Implementação de Referência

Os geradores de código são artefatos extremamente complexos e o seu desenvolvimento representa um grande desafio para os programadores. Essa

dificuldade é evidenciada principalmente durante a concepção inicial dos geradores, pois devem ser dotados de programação capaz de absorver a variabilidade das entradas (modelos) e produzirem o código da aplicação correspondente com todas as funcionalidades requeridas. Nesse contexto, a utilização de uma implementação de referência como ponto de partida para o desenvolvimento dos geradores pode facilitar o processo, fornecendo exemplos concretos das variabilidades existentes ao nível de implementação.

Para melhor visualizar esse conceito, considere-se o mesmo exemplo citado anteriormente, referente à geração de formulários HTML. A Figura 2.3 ilustra o processo.

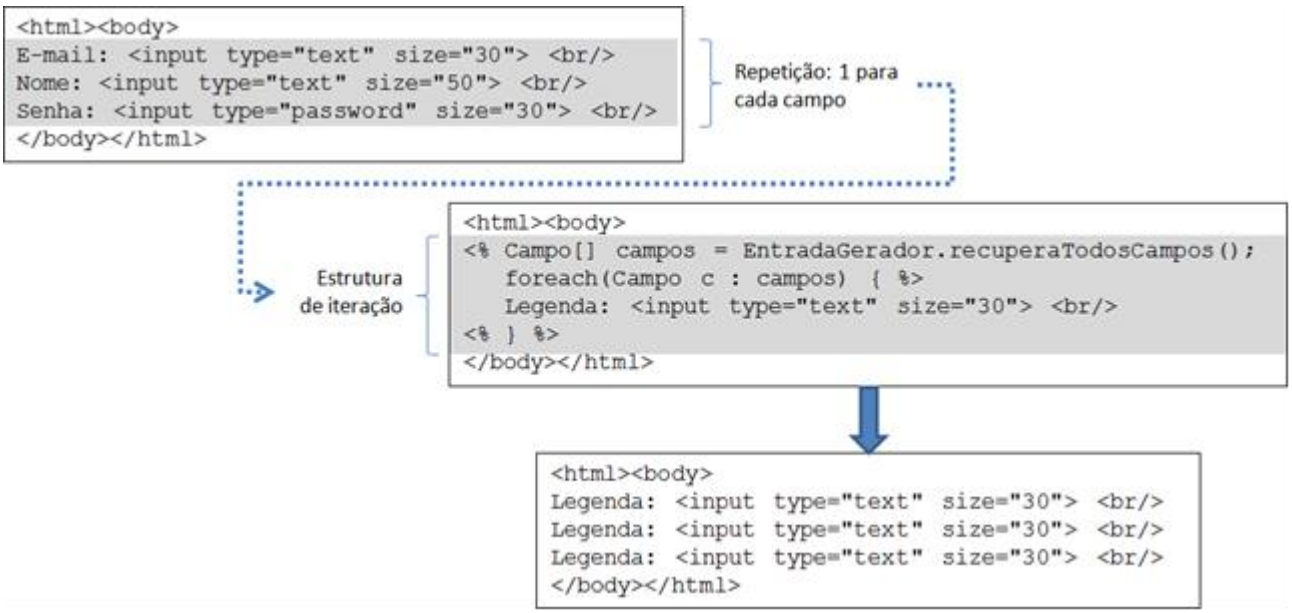

Figura 2.3 - Exemplo de utilização da implementação de referência

Inicialmente, parte-se de um exemplo do código a ser gerado, no caso, uma página HTML com um formulário. Analisando-se este código, identificam-se trechos repetidos, com potencial de automação. Em seguida, modifica-se o exemplo, introduzindo-se anotações e instruções que implementam a consulta ao arquivo de entrada e geração do código desejado. No exemplo da Figura 2.3, foi incluída uma estrutura de repetição simples, que apenas repete um mesmo trecho de código genérico (uma área para entrada de texto) para cada campo do formulário. Como resultado, na saída é produzido um formulário com três campos idênticos.

Dando continuidade ao processo (Figura 2.4), modifica-se o *template* de forma que seja feita a substituição da legenda de cada campo. Como resultado, produz-se na saída um formulário com três campos idênticos, porém com as legendas corretamente identificadas.

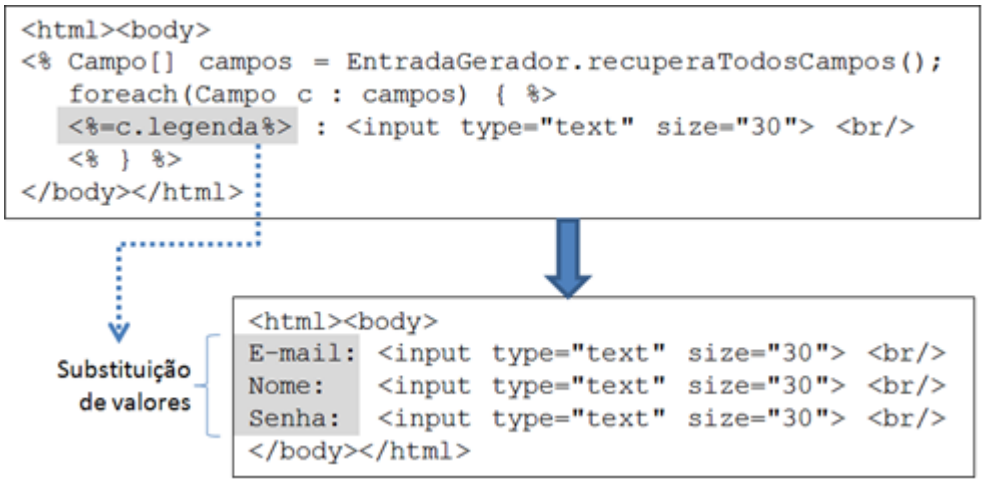

Figura 2.4 - Exemplo de utilização da implementação de referência – continuação

O seguimento deste exemplo, não ilustrado aqui, consiste na consulta ao tipo e tamanho de cada campo, e a inclusão de estruturas de controle para produzir o código correspondente. Após o término desta migração inicial, tem-se um gerador completo, capaz de gerar código idêntico à implementação de referência. Caso sejam necessárias mudanças, as mesmas são realizadas e testadas no exemplo, para somente depois serem migradas para o *template*.

Dessa forma, a dinâmica para a construção de geradores de código a partir de uma implementação de referência consiste na criação de um *template* genérico que, interpretando o código de referência, possibilita a geração de uma aplicação correspondente. Através do *template* genérico, aplicações similares podem ser criadas automaticamente, bastando introduzir elementos correspondentes como entrada no gerador de *templates*. Essa dinâmica de geração automática limita-se ao escopo do *template*, ou seja, ao poder de identificação e produção introduzido no *template*.

Como foi visto, a utilização da implementação de referência como fonte central de informações para a construção de geradores de código é uma abordagem bastante atrativa, que além de facilitar o processo de desenvolvimento possibilita a

criação de geradores consistentes, pois são fundamentados em aplicações já estabelecidas no mundo real. Porém, deve-se considerar que esse processo introduz mais um artefato no gerador, que precisa estar consistente com os demais para continuar existindo os benefícios trazidos por essa abordagem.

2.3 Migração de Código

Com a finalidade caracterizar o termo migração de código e também para garantir que não haja ambiguidades de interpretação é importante destacar a sua adoção no escopo desse trabalho. A adoção do termo migração partiu-se da utilização por Muszynski (2005) para definir a inserção do código de uma implementação de referência dentro do gerador de código, embora no trabalho anterior de Voelter e Bettin (2004) esse processo é denominado de ”extração de infraestrutura” ou “*templatization*.

O processo de migração de código consiste em utilizar uma implementação de referência como ponto de partida para o desenvolvimento de geradores, o que traz uma série de facilidades, conforme destacado no Capitulo 1, porém, o envolvimento desse novo componente agrega mais um artefato no processo de construção do gerador, que se não for garantida a sua sincronia com os outros, perderá sua funcionalidade.

Como observado por Voelter e Bettin (2004) a implementação de referência deve ser uma aplicação típica do domínio, sendo importante extrair uma infraestrutura a partir de uma aplicação que tem uma arquitetura de alta qualidade, uma vez que será a base para uma família de sistemas de software. No caso do desenvolvimento de aplicações no respectivo domínio com a abordagem dirigida a modelos, pode-se extrair a infraestrutura de aplicações existentes, se for o desenvolvimento inicial de uma categoria de novos softwares é necessário à elaboração específica de uma implementação de referência para a extração.

No processo de migração de código é necessário que seja mantida a implementação de referência consistente com o código gerado, ou seja, sempre que desejado será possível gerar uma nova implementação de referência, utilizando os

geradores construídos (LUCRÉDIO, 2009). Haverá problemas quando ocorrer modificações na implementação de referência, ocasionadas por correções, novas funcionalidades, identificações de oportunidades para suportar maiores variabilidades e outras.

A automação da migração de código não é trivial (MUSZYNSKI, 2005). Ainda não existe uma solução automática para migração de código nas principais ferramentas do mercado, como JET, *Velocity* e *Microsoft Text Templating*, entre outras. Não foi encontrada proposta na literatura que apresente uma solução parcial ou total para estes problemas, mesmo após a revisão sistemática de revistas, como a *Software and Systems Modeling*, conferências e workshops dedicados ao tema, como *ACM/IEEE MoDELS conference*, *Workshop on Domain-Specific Modeling*, *International Conference on Software Language Engineering*, entre outros.

A inserção do código de uma implementação de referência dentro do gerador (migração de código) resulta em duplicação de código e é difícil de automatizar, pois é necessário inserir instruções do código de geração dentro do código migrado (MUSZYNSKI, 2005). Baseado na experiência de autores e outros praticantes, a "*templatization*" (migração de código) consome de 20-25% do tempo gasto para desenvolver a implementação de referência (VOELTER, BETTIN, 2004).

A tarefa de sincronização dos artefatos envolvidos no processo migração de código está relacionada à complexidade de interação entre os elementos, principalmente considerando as diferentes maneiras em que eles se relacionam para produzir código. A seguir, são estudados os tipos de relacionamentos entre esses artefatos.

2.3.1 Tipos de Mapeamento

Por ser um assunto pouco explorado, não existe uma definição precisa e formal do problema da migração de código. Em um estudo inicial realizado por Lucrédio e Fortes (2010), com o objetivo de descrever as possíveis formas com os quais um ou mais *templates* podem ser combinados para gerar código, foram relacionadas sete diferentes tipos de mapeamento entre *templates* e uma implementação de referência, detalhados a seguir:

Tipo 1 - cópia simples: Este é o tipo mais simples de mapeamento. Um trecho de código em um *template* é copiado para a implementação de referência. Uma relação um-para-um entre o template e a implementação de referência é estabelecida, e o modelo de entrada não precisa ser consultado. A Figura 2.5 mostra esse tipo de mapeamento:

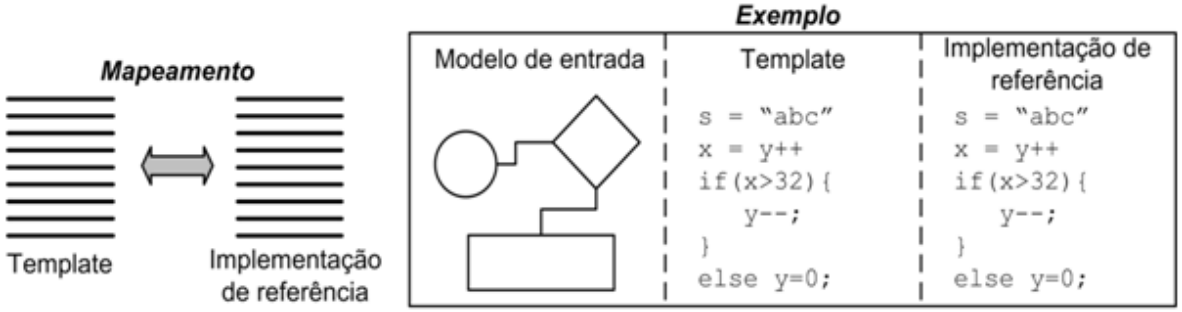

Figura 2.5 - Mapeamento tipo 1

Tipo 2 - substituição simples: Uma consulta ao modelo de entrada é inserida no *template*, sendo substituída pelo valor obtido do modelo quando o código é gerado. É possível rastrear o valor substituído até o modelo de entrada. A substituição simples é uma relação de um-para-um entre o *template* e a implementação de referência. A Figura 2.6 mostra esse tipo de mapeamento:

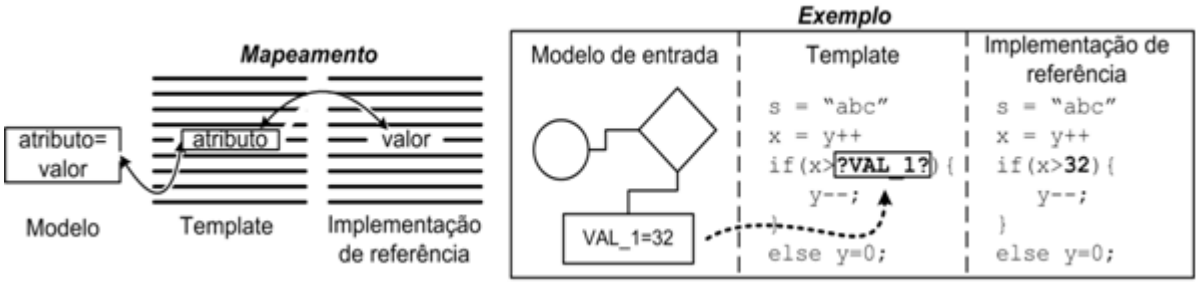

Figura 2.6 - Mapeamento tipo 2

Tipo 3 - substituição indireta: Similar ao tipo 2, mas aqui não é possível rastrear a origem do valor substituído, já que o mesmo não é explícito no modelo, sendo calculado no próprio *template*. A relação entre o *template* e a implementação de referência é de um-para-um. A Figura 2.7 mostra esse tipo de mapeamento:

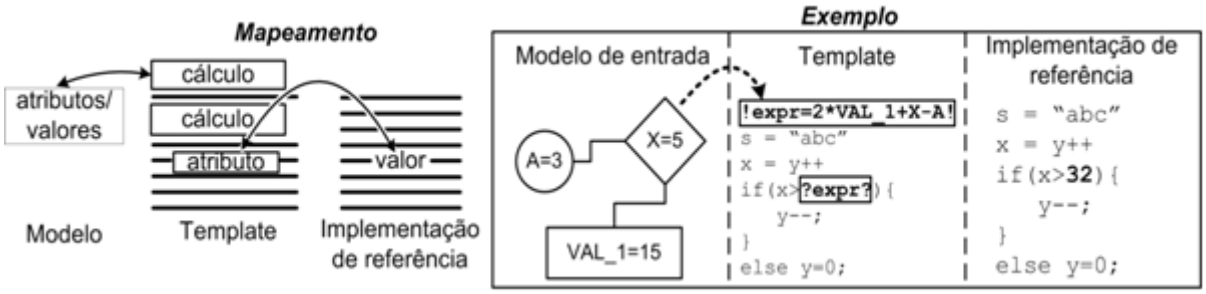

Figura 2.7 - Mapeamento tipo 3

Tipo 4 - repetição: Um pedaço de código de um *template* é repetido, zero ou mais vezes, na implementação de referência, de acordo com algum critério, que pode ter sido obtido do modelo de entrada ou não. A relação entre o *template* e a implementação de referência é de um-para-muitos. A Figura 2.8 mostra esse tipo de mapeamento:

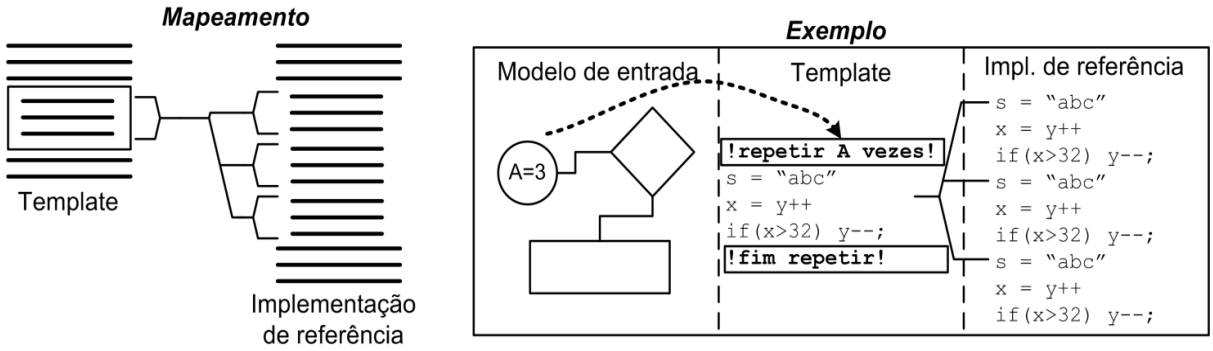

Figura 2.8 - Mapeamento tipo 4

Tipo 5 - condicional: Um trecho da implementação de referência pode ser mapeado para diferentes trechos de um *template*, com base em alguma condição estabelecida no *template*. A condição pode consultar o modelo de entrada ou não. A relação entre o *template* e a implementação de referência é de muitos-para-um. A Figura 2.9 mostra esse tipo de mapeamento:

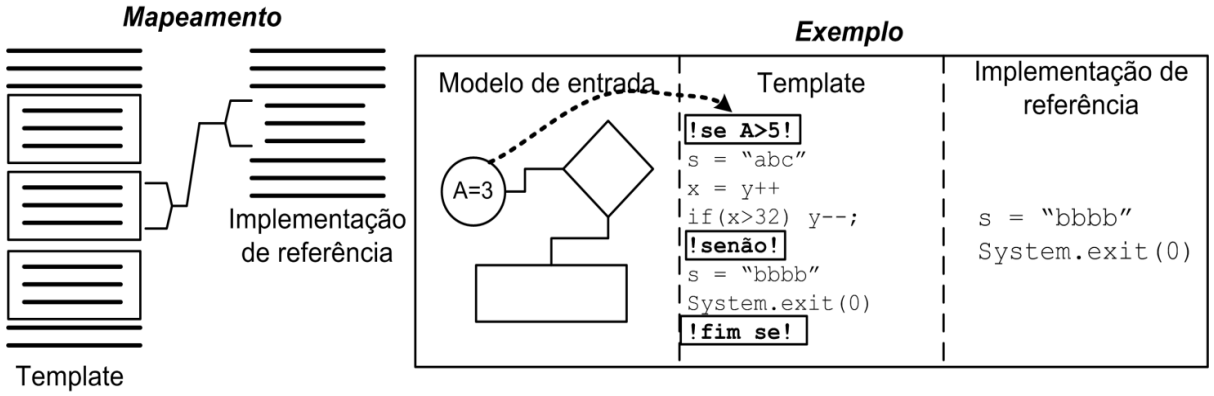

Figura 2.9 - Mapeamento tipo 5

Tipo 6 - inclusão: Um *template* inclui outro *template* para gerar um determinado trecho de código. A relação entre os *templates* e a implementação de referência é de muitos-para-um. É importante destacar que é sempre possível identificar, para cada linha da implementação de referência, o *template* responsável por sua geração.

Tipo 7 - novo arquivo: Um *template* solicita a criação de um novo arquivo. Após a requisição, o código gerado pelo *template* é redirecionado para o arquivo recém-criado. A relação entre o *template* e os arquivos da implementação de referência é de um-para-muitos. Como no tipo 6, é sempre possível identificar, para cada arquivo diferente da implementação de referência, o *template* responsável por sua geração. A Figura 2.10 mostra os tipos de mapeamento 6 e 7:

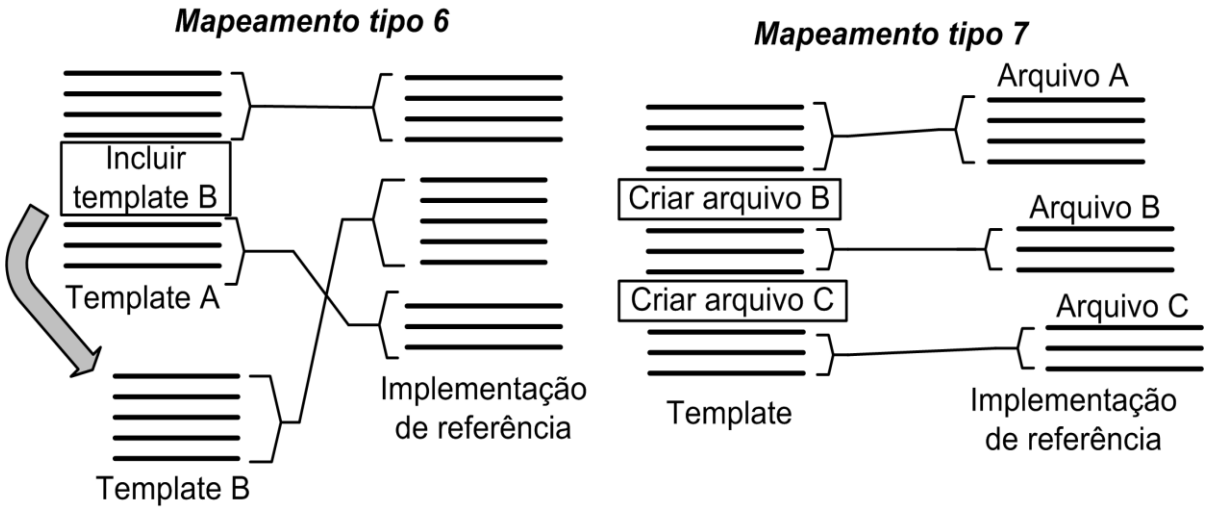

Figura 2.10 - Mapeamentos tipo 6 e 7

Muitas situações podem ser descritas combinando-se esses sete tipos de mapeamento. Com o desenvolvimento do protótipo de migração de código e principalmente no tratamento desses mapeamentos com técnicas para estabelecer o vínculo entre *template* e a implementação de referência, essa lista foi analisada com relação à forma de associação utilizada pelo protótipo, cujos detalhes estão descritos no próximo capítulo.

Para efeito de desenvolvimento deste projeto foram consideradas as especificidades dos mapeamentos apresentados para manter o sincronismo entre os artefatos. Esse sincronismo ocorre em momentos específicos durante o desenvolvimento e a evolução do gerador, conforme será abordado a seguir.

2.3.2 Pontos de Sincronização

A sincronização dos artefatos ocorre em determinado momento do ciclo de desenvolvimento do MDE. A Figura 2.11 apresenta o detalhamento das atividades relacionadas com a geração de código baseada em *templates* a partir de uma implementação de referência, no contexto do MDE, incluindo o momento da sincronização.

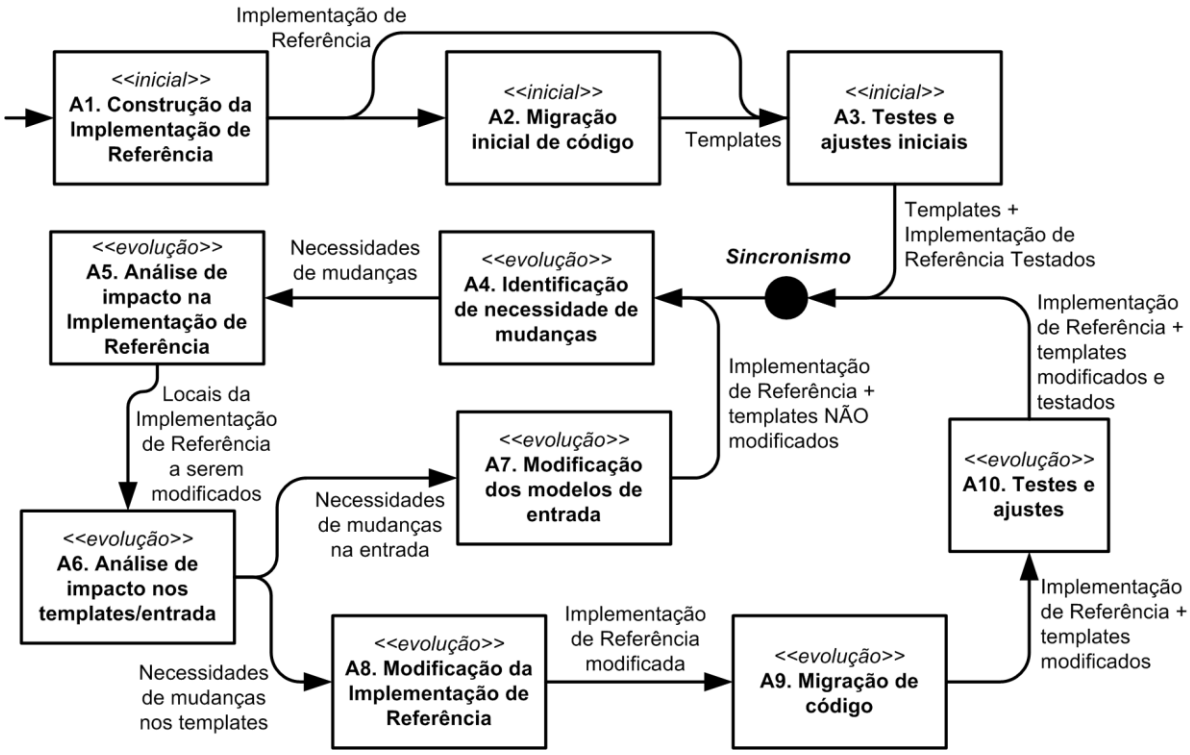

Figura 2.11 - Detalhamento das atividades do ciclo de desenvolvimento orientado a modelos utilizando uma implementação de referência

A Figura 2.11 contém a marcação <<inicial>> que indica as atividades relativas ao desenvolvimento inicial e <<evolução>> que indica as atividades relacionadas à evolução e manutenção de software. Esta figura é uma visão mais detalhada das atividades que compõem o ciclo MDE descrito anteriormente na Figura 1.1. Para melhor ilustrar essa relação: as atividades A1 a A3 correspondem ao lado esquerdo da Figura 1.1. As atividades A4 a A10 correspondem ao lado direito da Figura 1.1. As atividades A4 a A7 correspondem ao miniciclo representado no lado direito da Figura 1.1. As atividades A8 a A10 correspondem ao caminho “a” da Figura 1.1. O caminho “b” não é representado aqui, pois deve ser evitado, mas se fosse, seria uma seta entre a atividade A6 e o ponto de sincronismo, evitando as atividades A8 a A10.

A identificação das atividades do ciclo de MDE, partindo da dinâmica do sincronismo para estabelecer o melhor momento de conciliar os artefatos é importante para este projeto. Esta identificação contribui para determinar o caminho que oferece maior facilidade para obtenção do sincronismo entre os artefatos, mediante a identificação do elemento que oferece menor complexidade e o momento mais favorável para realizar as modificações.

Seguem abaixo os detalhes das dez atividades que foram identificadas como relevantes no contexto do ciclo de MDE conciliadas com o ponto de sincronismo (LUCRÉDIO; FORTES, 2010):

A1. Construção da implementação de referência: Esta é uma atividade normal de desenvolvimento de software, que produz um exemplo completo do código a ser gerado;

A2. Migração inicial de código: Consiste em anotar o código de exemplo com scripts e etiquetas, transformando-o em *templates* para geração de código;

A3. Testes e ajustes iniciais: Esta atividade visa avaliar os resultados da migração inicial, realizando ajustes conforme necessário;

A4. Identificação de necessidade de mudanças: Atividade que inicia um novo ciclo de evolução/manutenção, a partir de necessidades que surgem de uma ou mais aplicações;

A5. Análise de impacto na Implementação de Referência: Consiste em identificar, na implementação de referência, que partes do código precisam ser modificadas;

A6. Análise de impacto na entrada/*templates*: Algumas mudanças requerem apenas mudanças nos modelos de entrada e a regeração de código. Outras exigem a modificação de *templates*. Esta atividade cuida da identificação correta dessas necessidades, inclusive verifica a possibilidade dos modelos e/ou dos *templates* absorverem as mudanças pretendidas;

A7. Modificação dos modelos de entrada: Consiste em modificar os modelos e regerar o código. Este é o uso "canônico" do MDE. As atividades A6 e A7 representam o pequeno ciclo apresentado no lado direito da Figura 1.1;

A8. Modificação da implementação de referência: Esta atividade é realizada quando são necessárias mudanças nos *templates*. Conforme discutido anteriormente, as mesmas devem ser efetuadas primeiro na implementação de referência, onde podem ser testadas e validadas de maneira convencional;

A9. Migração de código: Nesta atividade, mudanças feitas na implementação de referência são propagadas para os *templates*. O objetivo é o mesmo da atividade A2, mas neste momento os *templates* já existem, e precisam somente ser atualizados; e

A10. Testes e ajustes: Busca avaliar os resultados da migração de código, realizando ajustes conforme necessário. As atividades A8, A9 e A10 representam o ciclo maior da Figura 1.1.

O círculo preto na Figura 2.11 indica o momento em que a implementação de referência e os *templates* devem estar sincronizados. Nota-se que isso sempre ocorre após a migração de código, nas atividades A2 (migração no estado inicial) e A9 (migração após as modificações), porém, neste projeto apenas a migração de código correspondente à atividade A9 será automatizada. A atividade de migração inicial de código (A2), conforme discutido anteriormente, exige muita habilidade e conhecimento humano sobre o domínio, sendo de caráter bastante criativo. Já na atividade A9, o conhecimento humano sobre o domínio já está codificado e embutido

nos *templates*. Dessa forma, é mais simples identificar e localizar mudanças de forma a propagá-las automaticamente.

É importante ressaltar que essas atividades identificadas podem não cobrir todas as possibilidades, ou seja, podem existir casos onde algumas não são necessárias e casos onde atividades não incluídas aqui sejam necessárias. Trata-se de uma abstração utilizada como base para o entendimento deste projeto, indicando o caminho e os elementos que oferecem maior facilidade para obter o sincronismo.

Com relação aos dois momentos do ciclo em que ocorre a migração do código, é extraída a Figura 2.12 do ciclo MDE, com o objetivo de ilustrar esse raciocínio. Durante o processo do MDE a implementação de referência entra e sai de um estado de sincronismo a cada iteração do ciclo, conforme descrito na figura (LUCRÉDIO; FORTES, 2010):

Figura 2.12 - Possíveis estados da implementação de referência durante o ciclo de desenvolvimento

Sincronizada: Neste estado, a implementação de referência está sincronizada com os *templates* de geração de código. Este estado sempre decorre da migração de código bem sucedida. Porém, apenas a migração não inicial será automatizada neste projeto; e

Dessincronizada: Existe um ou mais trechos de código onde a sincronização é perdida por causa de modificações na implementação de referência.

Retornar o sincronismo entre os artefatos do gerador de forma automática é o desafio deste projeto. Para isso, foram indicados no contexto da abordagem MDE os momentos para proceder ao sincronismo, que serviram de base para introduzir mecanismos para atuarem no processo de migração e geração de código, visando atingir esse resultado.

2.4 Considerações Finais

Neste capítulo foram discutidos os principais conceitos do processo de migração de código, sendo destacado que o método de desenvolvimento de geradores de código a partir da migração do código baseada na implementação de referência facilita esse processo, mas também introduz um novo elemento no gerador, que precisa estar consistente com os *templates* do gerador, para que os mecanismos que integram essa abordagem continuem a produzir os efeitos esperados.

Estudos iniciais detectaram sete tipos de mapeamentos existentes entre a implementação de referência e os *templates* do gerador, e cada um deles apresenta especificidades com relação ao sincronismo que deve haver após alterações serem introduzidas no gerador. Também foram apresentados os momentos específicos do ciclo MDE em que os componentes do gerador devem retornar o sincronismo.

Esses conceitos e estudos serviram de base para a busca de mecanismos visando solucionar o problema da perda de sincronismo que é agravada pela dificuldade existente em restabelecê-la no gerador. No próximo capítulo é apresentado o desenvolvimento do projeto de migração automática de código.

Capítulo 3

PROTÓTIPO DE MIGRAÇÃO AUTOMÁTICA DE CÓDIGO

Este capítulo apresenta o desenvolvimento do projeto de migração automática de código, que consiste na elaboração de um protótipo para sincronizar automaticamente os *templates* do gerador de código com determinadas alterações efetuadas na implementação de referência. O capítulo inicia com a apresentação da proposta que foi identificada inicialmente para resolver o problema e segue para as etapas do efetivo desenvolvimento do protótipo, incluindo o detalhamento e a análise das técnicas que foram utilizadas em sua composição.

3.1 Desafios do Projeto

A técnica de migração de código, conforme argumentado anteriormente, representa um mecanismo importante com potencial para agregar à abordagem MDE maiores facilidades e agilidade no processo de desenvolvimento de geradores. Porém, complicadores e limitações dificultam a garantia de continuidade do processo de migração, principalmente com relação à manutenção da sincronia da implementação de referência com os *templates* do gerador. Neste contexto, o projeto proposto consiste no desenvolvimento de um mecanismo que automatize parcialmente a migração de código com o objetivo de contribuir com essa

abordagem, que conforme Voelter e Bettin (2004) o processo de migração consome de 20 a 25 % do tempo de desenvolvimento de uma implementação de referência.

Para iniciar o desenvolvimento do mecanismo de migração automática é necessário encontrar uma forma de estabelecer o mapeamento entre o código gerado e os *templates* de geração de código. Esse mapeamento possibilita um caminho inverso ao da geração de código. Normalmente, mudanças realizadas nos *templates* são automaticamente propagadas para o código gerado, conforme o processo natural. O desafio é realizar o caminho inverso, ou seja, dada uma mudança no código gerado, encontrar o local, em um ou mais *templates*, onde estas mudanças devem ser propagadas. Dessa forma, pode-se utilizar o código gerado como uma implementação de referência e realizar manutenção e evolução diretamente nesse código, obtendo assim vantagens dessa abordagem.

Entretanto, estabelecer o vínculo entre diferentes tipos de mapeamento que ocorrem entre os *templates* e essa implementação de referência é apenas parte dos desafios da migração automática do código. É necessária a criação de mecanismos que executem a sincronização de forma dinâmica para realizar a migração de código com base na informação contida no mapeamento.

Em resumo, os objetivos do protótipo de migração automática de código são:

- Estabelecer fisicamente um mapeamento entre o código gerado (que fará o papel de implementação de referência) e *templates* para geração de código;
- Criar um mecanismo automatizado que, com base no mapeamento estabelecido e em determinadas mudanças realizadas na implementação de referência, consiga identificar os locais exatos dos *templates* a serem atualizados; e
- Efetuar as atualizações necessárias para sincronizar ambos os artefatos.

Nesse sentido, se faz necessário a montagem de um ambiente de desenvolvimento compatível com a abordagem MDE, com auxílio de ferramentas de geração de código que possibilite a aplicação dos mecanismos identificados com potencial de realizar a migração automática de código.

3.2 Gerador JET (*Java Emitter Templates*)

A ferramenta JET integra o *Eclipse Modeling Project*, consiste de um recurso para a geração de código baseado em *templates* (POPMA, 2004), sendo normalmente utilizado na implementação de geradores de código. JET permite gerar código a partir de modelos de entrada, além da possibilidade de utilizá-lo como gerador de código para uma DSL, pois o *template* JET pode ser acoplado a modelos da EMF (*Eclipse Modeling Framework)* ou arquivos XML. A saída do gerador JET não se limita a um formato específico, ou seja, qualquer tipo de código pode ser gerado por um *template* JET apropriado, como SQL, XML, Java, XML, HTML, JSP, texto e outros.

A sintaxe empregada pelo JET na criação dos *templates* que especificam o código a ser gerado é similar à do JSP (*Java Server Pages*). Os *templates* elaborados com o JET são convertidos em classes de implementação Java, em um processo denominado tradução. A classe Java é utilizada para criar a saída final, realizando a geração do código resultante, em um processo denominado geração.

A ferramenta JET auxiliando no desenvolvimento de geradores vem melhorar e acelerar o processo MDE, pois, embora seja possível transformar manualmente os modelos em código são diversas as vantagens obtidas com a automatização no processo geração de código a partir de modelos. Ainda com relação a essa dinâmica de desenvolvimento de software, o JET possibilita ao desenvolvedor incluir o mecanismo de variabilidade no gerador de código, ou seja, produzir modificações nos modelos para criar os artefatos automaticamente, e, assim substituir as implementações obtidas desses modelos.

Portanto, com o auxílio da ferramenta JET pode-se programar os *templates* para absorver a variabilidade dos modelos e isso permite criar um ambiente compatível com a abordagem MDE para a geração de código. Dessa forma, obtém-se as condições necessárias para aplicação e desenvolvimento de mecanismos, com a finalidade de atualizar automaticamente o *template* e retornar o sincronismo em seus componentes, conforme objetivo deste projeto.

3.3 Proposta Inicial: Anotações de Código e Comparação de Arquivos

A intenção inicial neste trabalho para estabelecer o mapeamento físico entre o código gerado e o *template* foi implementar uma abordagem baseada na técnica de anotações de código para demarcar os trechos de código gerado. Em seguida, uma comparação textual foi utilizada para detectar mudanças. Porém, o resultado desses esforços não foi satisfatório, como discutido a seguir.

3.3.1 Geração Automática de Anotações de Código

O objetivo da utilização dessa técnica consiste em introduzir anotações no código de referência para identificar a origem desse código nos *templates*, de forma que ao inserir modificações no código gerado seja possível rastreá-las de volta ao *template* original. Com a introdução de anotações no código o mapeamento é então estabelecido fisicamente e, após ocorrer a migração, o novo código gerado substitui a implementação de referência. Esse novo código contém marcações especiais que demarcam os limites de cada tipo de mapeamento. A Figura 3.1 ilustra esta técnica, para os mapeamentos de tipo 1 e 4 apresentados na seção 2.4.

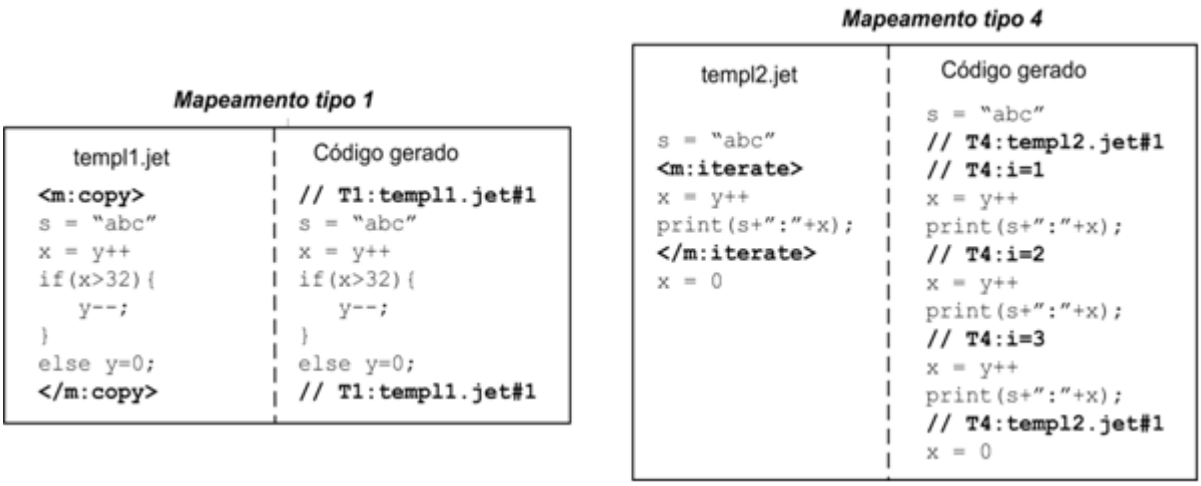

Figura 3.1 - Anotações no código para rastreamento utilizando JET (LUCRÉDIO; FORTES, 2010)

Para demonstrar essa técnica foi utilizado um recurso da ferramenta JET que permite adicionar *tags* (etiquetas) personalizadas para cada opção do código gerado.

No exemplo da Figura 3.1 foram criadas *tags* personalizadas para dois tipos de mapeamento, identificadas pelo prefixo “m:”. As novas *tags* reproduzem o comportamento correspondente às *tags* originais, mas também geram anotações (neste caso, comentários Java), que marcam localizações relevantes no código, conforme cada tipo de mapeamento. Nesse exemplo, foi introduzida a *tag* <*m:copy*> para copiar uma parte do código e também inserir automaticamente um comentário no início e no fim (mapeamento tipo 1). Também foi inserida a *tag* <*m:iterate*>, que realiza uma repetição, ao mesmo tempo que marca o começo e o fim do código gerado, além do início de cada iteração (mapeamento tipo 4).

A Figura 3.2 mostra a implementação dessa técnica na criação de uma página HTML. O arquivo de *template* (*html.jet*) contém as *tags* responsáveis pela geração das anotações, envolvendo as referências ao modelo e também ao tipo de mapeamento. O arquivo gerado (*teste.html*) contém o código gerado, com as anotações de comentários para cada mapeamento especificado no *template*. Os trechos destacados na figura indicam uma das correspondências produzida nesses arquivos, onde a *tag* <*mbct:scopy*> introduzida no arquivo de *template* produziu a cópia do texto para o arquivo gerado e incluiu a *tag* <*!--T1:template.html.jet#1233*> que demarcou o início e o fim desse mapeamento. Nesse exemplo, além das *tags* implementadas no exemplo anterior também são demonstradas a *tag* <*mbctGet*> responsável pela consulta ao modelo (mapeamento tipo 2) e a *tag* <*mbctMyIfTag*> que implementa o mapeamento condicional (mapeamento tipo 5).

A vantagem dessa técnica é a simplicidade na dinâmica de implementação, bastando inserir anotações nos trechos em que se deseja realizar manutenção entre as instruções do *template*. Porém, é evidente que a técnica resulta em poluição do código e, além disso, caso alguma marcação desapareça acidentalmente do código de referência, corre-se o risco de se perder toda a associação. Há maneiras de resolver o problema protegendo essas *tag*s de modificações, porém, para simplificar o início do desenvolvimento do protótipo assumiu-se que esses problemas seriam considerados *a posteriori*.

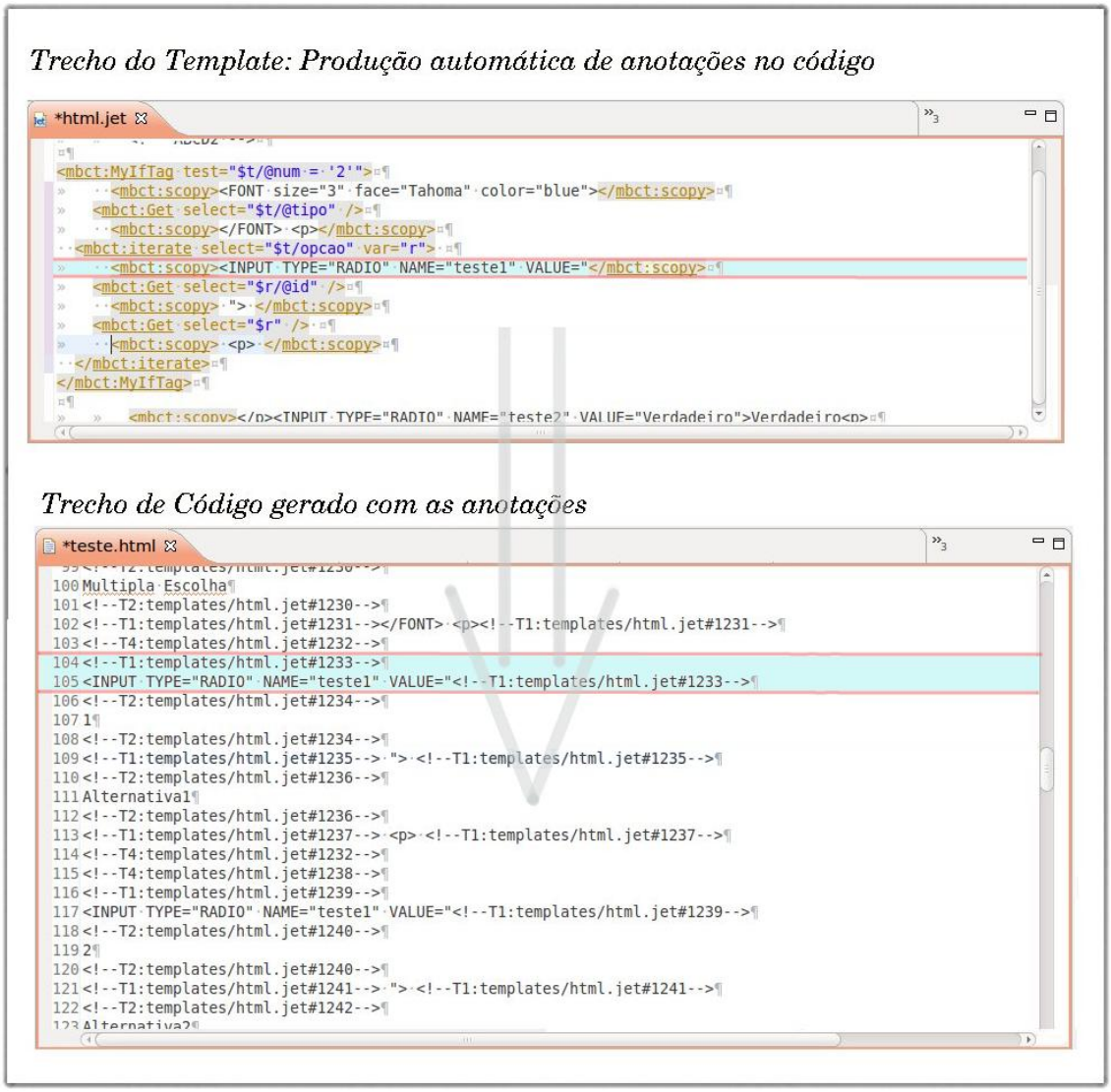

Figura 3.2 - Detalhes da implementação da técnica automática de anotação de código

3.3.2 Esquema de Sincronização Automática Baseada na Técnica de Anotação de Código

Estabelecer o mapeamento físico é o passo inicial para a migração automática de código. O próximo passo é identificar as diferentes mudanças possíveis que podem ocorrer na implementação de referência, e a partir disso desenvolver um mecanismo que replique automaticamente cada tipo de alteração feita na implementação e retorná-la para os *templates*, usando as informações de referência para isso. A Figura 3.3 ilustra o processo de sincronização automática projetado.

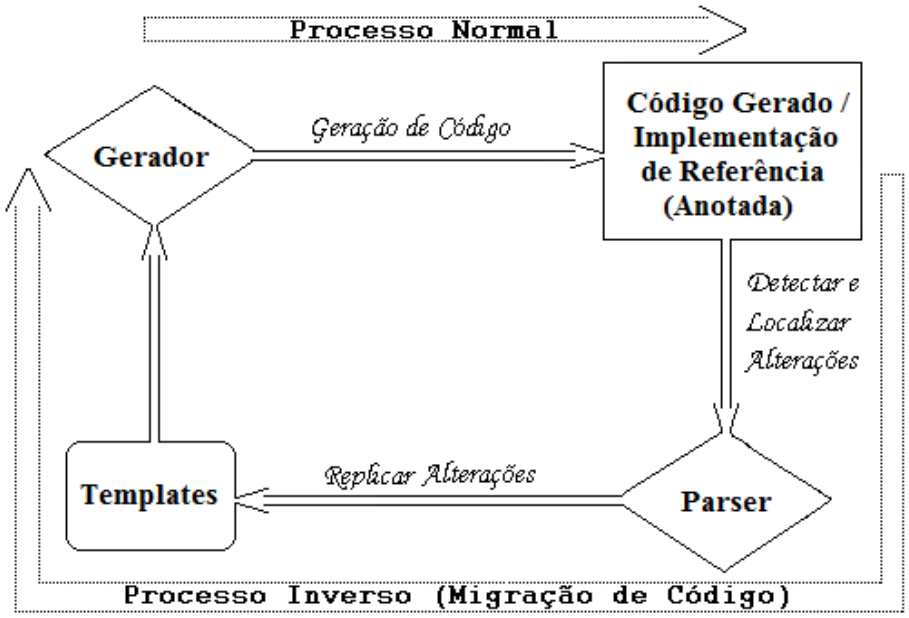

Figura 3.3 - Esquema de migração automática incluindo o mecanismo de sincronização baseado na técnica de anotação de código

O gerador inicia o processo por meio da geração de código, que corresponde ao caminho “normal”. O código, uma vez gerado, torna-se a nova implementação de referência. Esse código está devidamente anotado, conforme descrito na subseção anterior, de modo que eventuais alterações podem ser detectadas por algum tipo de analisador ou *parser*. O *parser* também precisa localizar tais alterações, isto é, determinar em qual(is) *template*(s) as mesmas devem ser replicadas. Uma vez localizadas, as alterações são então replicadas nos *templates* correspondentes, de onde podem ser utilizadas novamente pelo gerador para produzir uma nova implementação de referência, agora sincronizada.

Assim, o esforço foi centrado na busca de um mecanismo que encontrasse as alterações realizadas no código gerado com as respectivas anotações, para posteriormente associá-las ao *template* correspondente. Como tentativa para essa finalidade utilizou-se a biblioteca “*Java Open Source java-diff-utils*”, distribuída sob licença Apache 2.0, que implementa o algorítimo “*Myer's diff*”, desenvolvido por Naumenko (2011) para a comparação de arquivos.

As alterações ocorridas nos arquivos decorrentes de operações de *insert*, *delete* e *replace* são detectadas via execução do *java-diff-utils*, que separa as diferenças obtidas na comparação entre dois arquivos. Os arquivos são comparados

com base nessas três operações e são convencionalmente denominados de revisados e originais. Dessa forma, se fez necessário efetuar a duplicação do código gerado para utilizar esse mecanismo de captura de diferenças entre arquivos, ou seja, produzir uma cópia de cada arquivo do código da aplicação durante a sua geração e, assim, manter uma versão preservada da aplicação original para realizar a comparação, de forma a extrair as alterações introduzidas nos arquivos do código da aplicação.

A Figura 3.4 mostra as informações das alterações ocorridas durante a comparação de dois arquivos, que foram detectadas e retornadas pelo *java-diff-utils*. Durante a comparação dos arquivos, além do retorno do texto que foi modificado, também é extraída a linha do arquivo em que ocorre a diferença. No exemplo, essas informações são exibidas no terminal do *Eclipse,* sendo que o campo '*position*' indica a posição da linha no arquivo e o campo '*lines*' retorna os textos correspondentes do arquivo original e do arquivo alterado, separados por colchetes.

Figura 3.4 – Alterações detectadas via comparação de arquivos

O comparador informa a posição da linha do arquivo, juntamente com as demarcações representadas pelas *tags* de início e fim de cada mapeamento, que foram produzidas pelo gerador JET no código gerado. Essas referências indicam o tipo de correção necessária, sua abrangência e o trecho do código correspondente que deverá ser replicado para atualizar o *template*.

3.3.3 Limitações das Técnicas de Anotação de Código e Comparação de Arquivos

Como já previsto, as anotações introduzidas pelo gerador JET poluem o código gerado e dificultam o seu entendimento, visto que essas anotações estão associadas aos detalhes internos do processo de geração do *template* e, portanto,

destoam do uso convencional dos comentários que são inseridos nos arquivos da implementação.

A aplicação geralmente contém inúmeros formatos de arquivos (*.java, *.jsp, *.html e outros) e o gerador JET precisa gerar a *tag* de comentário conforme o padrão pré-definido para cada tipo de arquivo. Por outro lado, também é necessário introduzir um mecanismo que durante a edição do código proteja essas anotações, caso alguma seja acidentalmente excluída o mapeamento do código com o *template* será perdido.

O mecanismo *java-diff-utils* utilizado para encontrar as alterações no código por meio da comparação de arquivos retorna a linha do arquivo em que ocorre a diferença, porém, existem linhas no código gerado compostos por mais de um mapeamento. Portanto, seria necessário adaptar esse mecanismo para encontrar alterações em trechos da linha, ou seja, separar as alterações contidas em intervalos correspondentes a cada mapeamento contido nessa linha. Em contraste, existem mapeamentos que englobam mais de uma linha e a alteração refletida somente na linha capturada poderá danificar a funcionalidade do *template*.

Mesmo se introduzir um mecanismo para encontrar diferenças em trechos dentro de uma mesma linha ainda haveria outras limitações para serem tratadas. Uma limitação típica verifica-se no exemplo clássico de geração de uma classe Java, com os métodos acessores "*get* e *set*". O nome do método, como por exemplo "*setMarca* e *getMarca*" são obtidos com pela concatenação do texto "Marca" obtido do modelo de entrada e o "*get* ou *set*" originários do template, por meio de um mecanismo de repetição (*iterate*) para cada atributo da entidade contido no modelo. Assim, observa-se a existência de mais de um mapeamento gerado na mesma linha abrangendo consulta ao modelo de entrada (mapeamento tipo 2) que combina informações do *template* e os envolve em uma repetição (mapeamento tipo 3), e produzem, conforme exemplo, o nome do método acessor da classe Java. Essa composição impede seu mapeamento através da técnica de anotação de código, pois danificaríamos o código com a inserção de uma *tag* de comentário entre as cadeias "*get*" e "Marca" ou "*set*" e "Marca".

Essas limitações fizeram com que se buscasse alternativas com a finalidade de evitar a marcação direta no código para reconhecer os artefatos envolvidos no processo de geração e de resgatar as alterações específicas de cada mapeamento

efetuadas no código da aplicação. Porém, cabe o reconhecimento de que essas técnicas contribuíram na fase inicial de desenvolvimento do protótipo, à medida que apontaram a forma em que os elementos contidos nos artefatos do gerador precisam ser associados e capturados para automatizar o processo de migração de código.

3.4 Segunda Proposta: Arquivo Externo de Mapeamento e Registro de Modificações

Em face das limitações observadas com a utilização da técnica de anotações de código para demarcar o próprio código da aplicação e também as dificuldades apresentadas para adaptar o comparador de arquivos nesse processo, optou-se por desenvolver um novo protótipo, fazendo uso de duas técnicas diferentes:

- Arquivo externo: ao invés de anotar o código da aplicação adotou-se geração de um arquivo externo, contendo os mapeamentos e a associação dos artefatos do gerador; e
- Registro das alterações: como a comparação mostrou-se limitada para o propósito desta pesquisa, buscou-se um mecanismo de captura de eventos de modificações durante a edição de arquivos.

Essas duas técnicas são detalhadas a seguir.

3.4.1 Arquivo de Mapeamento

Introduziu-se no gerador JET um mecanismo que, ao invés de produzir anotações de mapeamento no código gerado, criasse fisicamente um novo arquivo em paralelo, contendo a relação das posições de cada mapeamento, obtido da associação dos arquivos de *templates* com as referências dos modelos de entrada e com o código gerado correspondente. Um trecho desse arquivo de associação é mostrado na Figura 3.5.

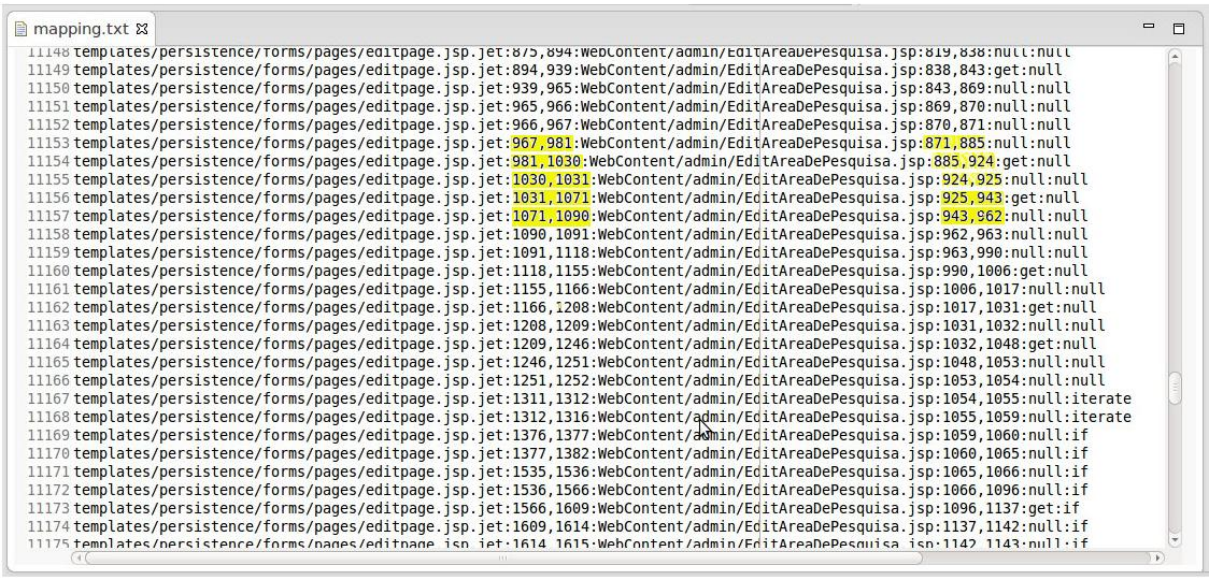

Figura 3.5 - Detalhes do arquivo de mapeamento

Os detalhes dos campos do arquivo de associação criado pelo gerador JET podem ser visualizados na Figura 3.5, onde cada linha representa um mapeamento. Os campos são delimitados com o caractere ":" e seguem a sequência: caminho e nome do arquivo de *template* com as posições de início e fim do arquivo de *template*; caminho e nome do arquivo gerado da implementação com as respectivas posições de início e fim do arquivo gerado; informação da origem do código produzido e do tipo de mapeamento. As posições correspondentes aos trechos destacados nos outros artefatos estão identificadas nas próximas figuras (Figura 3.6

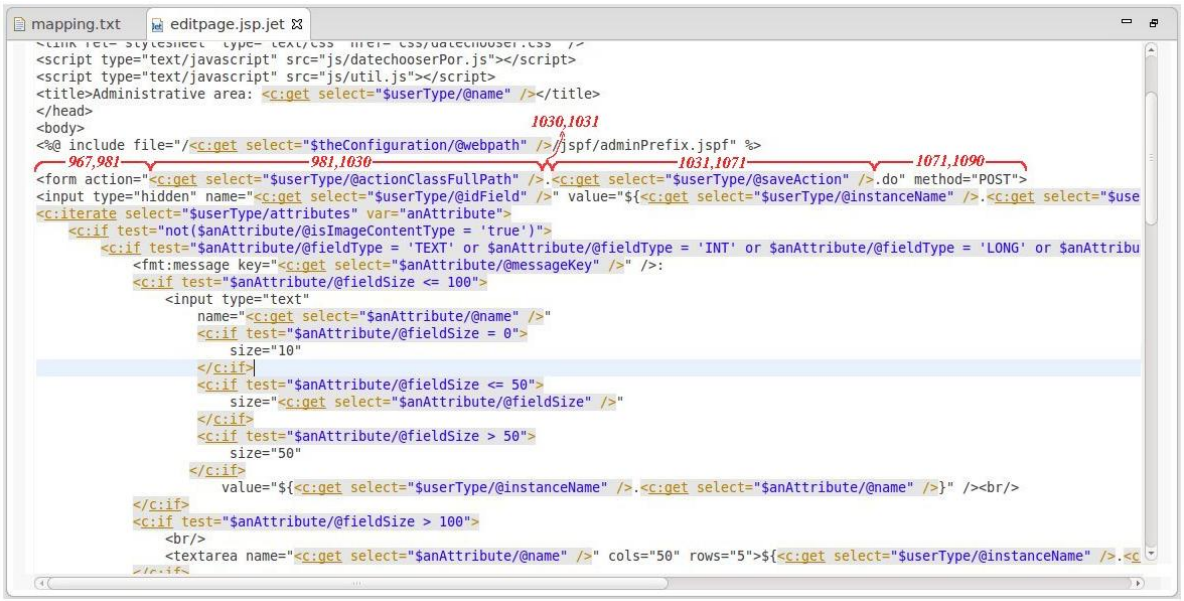

Figura 3.6 - Código do *template* com as posições do mapeamento

e Figura 3.7).

Para ilustrar o relacionamento produzido entre os artefatos do gerador, por meio da associação estabelecida no arquivo de mapeamento, apresenta-se a Figura 3.6, que mostra o arquivo de *template* com as posições correspondentes do mapeamento destacadas e a Figura 3.7, representando o arquivo gerado com os destaques nas posições do respectivo código que foi produzido.

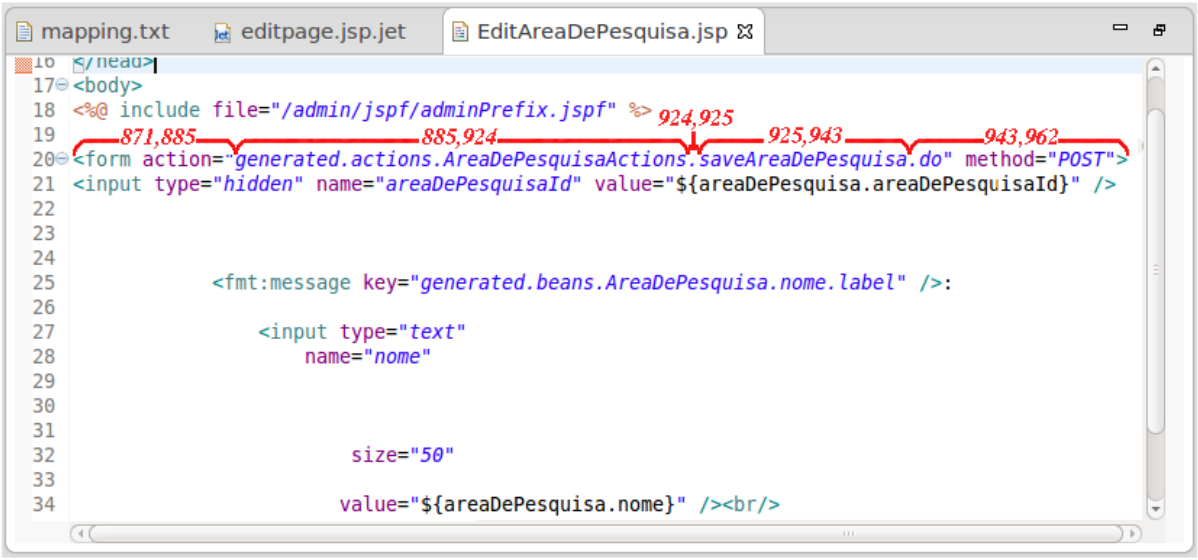

Figura 3.7 - Código gerado com as posições do mapeamento

Esse arquivo mapeando os artefatos que integram o gerador é criado dinamicamente durante o processo de geração de código, ou seja, toda vez que é executado o gerador JET o arquivo de associação é refeito e os mapeamentos correspondentes são atualizados. Para implementar foram feitas modificações no código original do JET, realizadas por outro membro do grupo de pesquisa em parceria com o autor desta dissertação.

Cabe destacar que este trabalho está focado basicamente em soluções para estabelecer a sincronia do *template* com o código gerado e não na atualização do modelo a partir do código. A variabilidade dos modelos de entrada já integra o ciclo MDE em seu caminho de ida, ou seja, o código da aplicação é regerado dinamicamente toda vez que se altera o modelo, até o limite de absorção determinado pela programação embutida no *template*. Porém, o código gerado carrega as informações obtidas do modelo e essas também devem ser

transportadas para o arquivo de mapeamento, possibilitando que à medida que for detectado algum trecho de código e que tenha origem no modelo, seja possível coibir qualquer tentativa de mudança no modelo a partir do código.

3.4.2 Registro de Alterações

A existência de um arquivo de mapeamento criado fisicamente durante o processo de geração, contendo as associações dos artefatos, possibilita localizar as posições das alterações ocorridas no código e associá-las com as posições correspondentes no *template*, mas antes é preciso de um mecanismo que identifique ou capture as posições das modificações ocorridas na implementação de referência para proceder a busca nesse arquivo de mapeamento.

As tentativas de comparar arquivos para encontrar suas diferenças utilizando a biblioteca *java-diff-utils* foram interrompidas devido às dificuldades apresentadas, então, partiu-se para a utilização de um mecanismo que possibilitasse a captura de eventos ocorridos durante a edição de arquivos no editor do *Eclipse*. Para essa finalidade utilizou-se a ferramenta *Fluorite* que registra eventos no *Eclipse*, disponibilizada por Yoon e Myers (2011).

O *plug-in* do *Fluorite* após ser instalado no *Eclipse* captura todos os eventos durante a utilização do editor de código. As informações dos eventos são registradas de forma detalhada em um arquivo de *log* no formato XML. Nesse arquivo o *Fluorite* grava todas as informações das operações de alteração de dados nos arquivos editados, além dos parâmetros específicos de cada comando executado.

A Figura 3.8 mostra um trecho de um arquivo gerado pelo *Fluorite*, onde foram destacadas as informações de uma operação *delete*: o evento *DocumentChange* demarca essas informações e relaciona os parâmetros correspondentes, entre eles estão destacados: *_type* que representa o tipo de operação*, offset* informa a posição da alteração no arquivo e a *tag text que* demarca o texto que foi excluído*.*

Utilizando os dados coletados e registrados no arquivo de *log* é possível recuperar todas as alterações ocorridas no editor de código, como a digitação de um novo texto, os comandos copiar e colar, desfazer e refazer, além de todos os dados

excluídos ou substituídos durante a edição de arquivos. Assim, para processar as informações contidas nesse arquivo foi necessário um "*Parser* XML" específico, inclusive com as funcionalidades de recuperar somente os campos referentes às alterações de dados e de buscar o último arquivo gerado, pois o *Fluorite* grava um novo arquivo a cada vez que o *Eclipse* é iniciado.

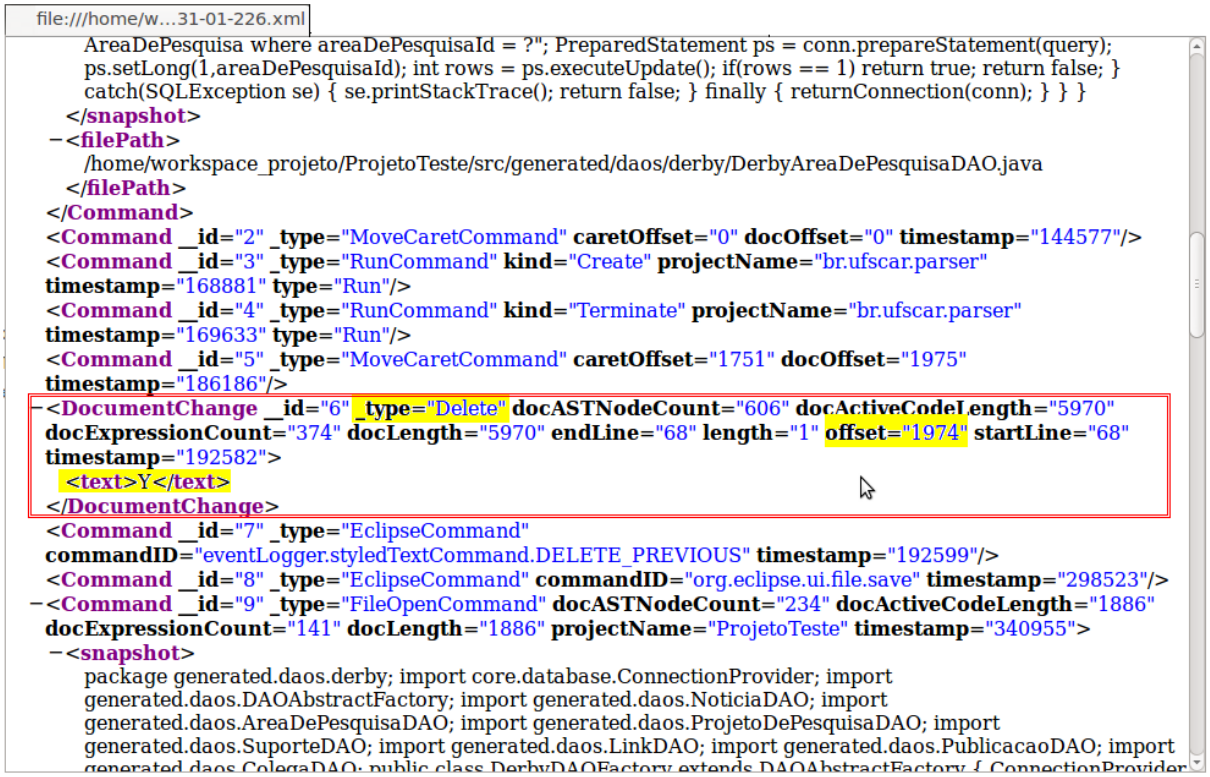

Figura 3.8 - Detalhes do arquivo de *log* gerado pelo *Fluorite*

Com os eventos de alteração de dados registrados e com o arquivo de mapeamento partiu-se para o desenvolvimento do *Parser*, com a finalidade de associar os eventos aos mapeamentos relacionados e replicar as alterações realizadas no código gerado para o *template* correspondente.

3.4.3 Replicação das Alterações nos *Templates*

Contando com as informações da associação dos artefatos do gerador, agora asseguradas por meio do arquivo de mapeamento, criado durante o processo de transformação dos modelos e *templates* em código e também com o arquivo de *log* gerado pelo *Fluorite*, que mantém o registro de todos os eventos capturados durante

a edição dos arquivos no *Eclipse*, tornou-se possível o desenvolvimento de um *Parser* para processar esses arquivos, de forma a resgatar do arquivo de *log* todas as informações referente às alterações realizadas nos arquivos da aplicação e associá-las no arquivo de mapeamento, para reproduzi-las nas posições corretas dos respectivos arquivos de *template*.

A Figura 3.9 mostra como ficou o esquema do projeto de migração automática de código com os mecanismos de mapeamento e de captura de eventos. O *Parser* percorre o arquivo de *log* buscando os eventos relacionados com alterações em arquivos. Ao encontrar uma alteração ele extrai as informações necessárias desse evento para associá-las no arquivo de mapeamento e então encontra a posição correspondente do arquivo para realizar a alteração. Para cada alteração processada ocorre mudança na correspondência contida no arquivo de mapeamento, o que necessita ser corrigido a cada execução, para continuar mantendo o mapeamento consistente para as próximas alterações.

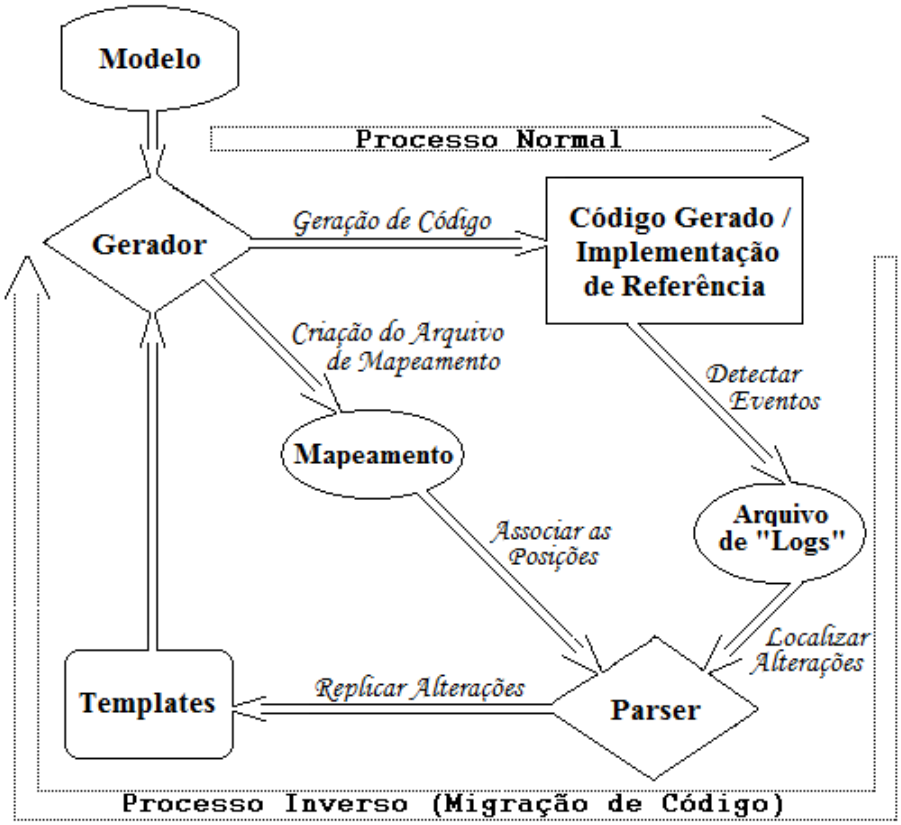

Figura 3.9 - Esquema de migração automática de código com base em arquivo de mapeamento e registro de alterações

As informações contidas nos arquivos da aplicação que são originadas dos modelos de entrada não podem sofrer alterações, pois conforme discutido anteriormente, a sincronização com o modelo está fora do escopo desta pesquisa. Então foi incluída no *Parser* uma funcionalidade para abortar a operação à medida que detectar qualquer alteração em trecho de código referente ao modelo, processando somente a atualização no arquivo de mapeamento para mantê-lo consistente com as demais alterações.

Para alterações ocorridas em trechos da aplicação originados de mecanismo de repetição do *template* (*iterate*) foi introduzida uma caixa de diálogo (Figura 3.10) para o programador decidir quanto à propagação, pois caso seja confirmado, todos os trechos de código gerados pela iteração serão alterados ou em caso negativo, somente o arquivo de mapeamento é atualizado para continuar consistente nas próximas alterações. Essa caixa de diálogo também oferece algumas informações da operação para auxiliar a decisão do programador: no campo '*id*' é informado o número de identificação do evento constante do arquivo de *log,* o campo '*off*' informa a posição do primeiro caractere alterado no arquivo, o campo '*ins*' indica o texto que será incluído e o campo '*del*' o texto que será excluído e na última linha são mostrados os nomes e os caminhos do arquivo da aplicação que ocorreu a alteração e do arquivo do template que poderá ser atualizado.

Figura 3.10 - Caixa de diálogo *Iterate*

No caso de alterações do código via operações de *delete* e *replace* que atingirem mais de um mapeamento na sequência o *Parser* também apresenta uma caixa de diálogo com o rótulo *Overtake* (Figura 3.11), contendo todas as informações coletadas dos arquivos, o que possibilita a decisão do programador por meio de uma análise específica, pois a replicação composta de mapeamentos diferentes poderá danificar o *template*. Em caso de confirmação os mapeamentos envolvidos na alteração serão convertidos em um único mapeamento ou em caso negativo,

somente o arquivo de mapeamento é atualizado. Os campos da caixa de diálogo são similares aos do *Iterate.*

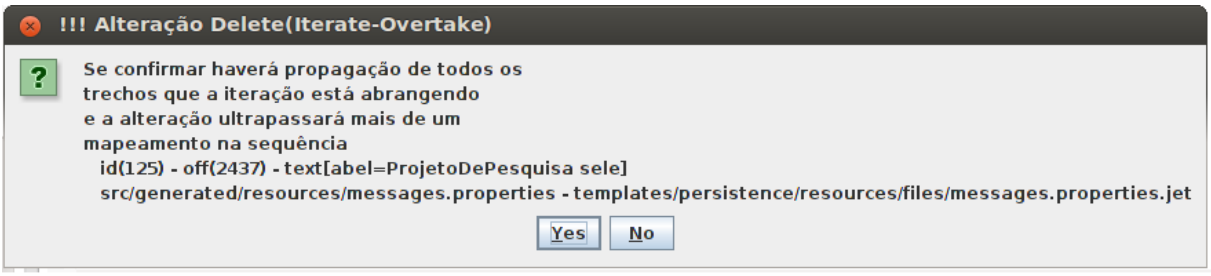

Figura 3.11 - Caixa de diálogo *Overtake*

As informações da execução do *Parser* são mostradas no terminal do *Eclipse*, inclusive as tarefas abortadas (Figura 3.12). Dessa forma, o programador tem como visualizar e acompanhar todas as alterações que foram processadas antes de proceder a geração de código. Na primeira linha é informado o nome e o caminho do arquivo de *log* e nas próximas linhas são exibidas as ações executadas pelo *Parser*, sendo que no início de cada linha é indicado o resultado do processamento e o tipo da ação seguido das informações específicas da operação processada, cujos campos de informações são os mesmos das caixas de diálogo.

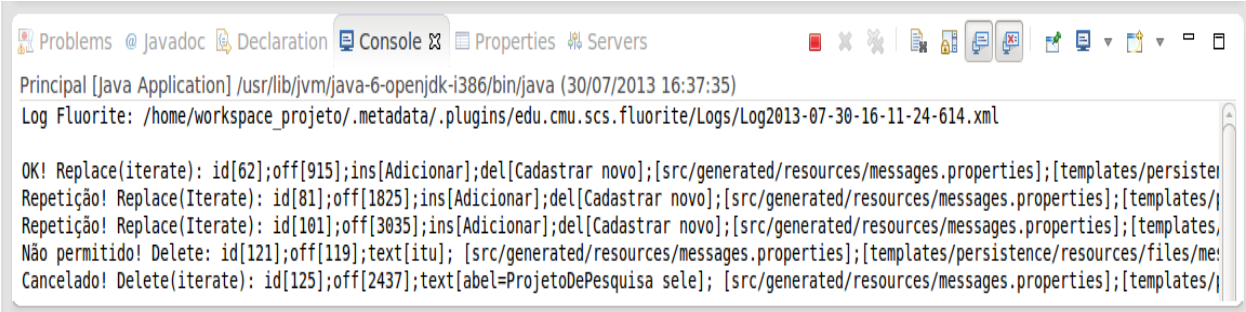

Figura 3.12 - Informações da execução do *Parser* via terminal

Com relação à manipulação dos arquivos de *log* pelo *Parser*, considerando que um novo arquivo é gerado pelo *Fluorite* toda vez que o *Eclipse* é iniciado, foi necessário o *Parser* capturar o último arquivo da pasta que está sendo manipulado e também percorrer esse arquivo para determinar o ponto inicial do processamento, a

cada vez que é executado. Com isso foi evitado o processamento repetido da alteração, que poderia danificar os arquivos do gerador.

3.4.4 Análise do *Parser* conforme o Tipo de Mapeamento

As alterações efetuadas no código da aplicação que o *Parser* replicará para o *template* devem acompanhar o tipo de mapeamento estabelecido na geração, para produzir o código modificado. Os tipos de mapeamento estão relacionados no Capitulo 2, extraídos do estudo realizado por Lucredio e Fortes (2008) e uma análise da forma de associação utilizada pelo *Parser* estão descritas a seguir:

a) Cópia simples: Consiste na cópia de um trecho do *template* para o código (relação um para um) e não há consulta ao modelo de entrada. O *Parser* detecta a alteração no arquivo de *log*, associa com o arquivo de mapeamento e replica a alteração na posição correspondente do *template*. A sincronização nesse tipo de mapeamento foi resolvida integralmente pelo *Parser*.

b) Substituição Simples: Refere-se a uma consulta realizada pelo *template* ao modelo de entrada, cujo valor obtido no modelo é substituído quando o código é gerado. Apesar de estar identificado no arquivo de mapeamento o trecho correspondente ao *template*, a sincronização do código gerado com as informações oriundas do modelo não está no escopo desse projeto. Assim, se o *Parser* detectar mudança envolvendo o modelo de entrada, ele a desconsiderará e nada aplicará no *template*, porém, corrigirá as posições desses artefatos no arquivo de mapeamento para que o arquivo de *log* não fique dissociado e o *Parser* continue atuando nas próximas alterações detectadas.

c) Substituição Indireta: Refere-se a dados provenientes do modelo que são calculados no *template*. Da mesma forma que o mapeamento anterior, as informações constantes no código gerado que são oriundas dos modelos serão tratadas pelo *Parser* somente para manterem a sincronia dos artefatos, porém, não serão replicadas para os modelos,

pois a atualização dos modelos de entrada a partir do código não está sendo investigada neste projeto.

d) Repetição: Um trecho do código é repetido, podendo haver consulta ao modelo de entrada ou não. O *Parser* contempla as modificações em trechos de código oriundos de iterações, exceto na parte que instancia a informação do modelo. Foi introduzida uma caixa de diálogo para que o programador decida sobre a atualização do *template*. Caso a opção seja de não replicar a iteração, então o processo será abortado, mas as posições do arquivo de mapeamento serão atualizadas para manter a correspondência para o próximo evento. A alternativa do programador em determinar se vai haver a correção no caso da iteração se deve ao tipo de associação que é de um para muitos, onde um único trecho do *template* se propaga para diversos locais do código e, como se captura a alteração efetuada no código da aplicação, a sua replicação para o *template* refletirá em toda a abrangência da iteração. Caso ocorram alterações em mais de um trecho de código que tenha origem no mesmo intervalo de iteração do *template*, somente a primeira sequência das posições da alteração será refletida no *template* e as seguintes serão abortadas.

e) Condicional: Por meio de uma condição estabelecida no *template* um trecho de código é mapeado para diferentes trechos de um *template*. No caso da condição estabelecida ser verdadeira, o arquivo de mapeamento apresentará o resultado dessa produção, trazendo as posições do código gerado associado com o *template*. Assim, se o dado for originário do modelo de entrada a operação será abortada, senão será atualizada a informação do *template* no trecho de código resultante da condição estabelecida.

f) Inclusão: Um *template* inclui outro *template* para gerar um determinado trecho de código. O arquivo de mapeamento identifica o trecho de cada *template* que é combinado para a geração de código. Assim o *Parser* realizará a sincronização de cada template com o código modificado, exceto os trechos de código resultante da consulta ao modelo de entrada.

g) Novo Arquivo: Um *template* pode ser responsável pela criação de vários arquivos. O *Parser* trata esse mapeamento de forma semelhante ao da repetição, com a diferença que essa iteração associa um trecho do *template* a vários arquivos que são gerados. Portanto, a propagação da modificação existente em um desses arquivos para o *template* refletirá em todos os outros arquivos associados. Caso o programador corrija todos os arquivos no mesmo ponto de geração, o *Parser* desprezará os mapeamentos seguintes que se referem às mesmas posições do *template*.

Com a finalidade de ilustrar a aplicação do mecanismo de migração automática de código é exibida a Figura 3.13 para o mapeamento do tipo repetição. A partir da geração de código um arquivo de mapeamento é criado em paralelo com o código gerado / implementação de referência. Neste exemplo, é exibido o mapeamento do tipo repetição, onde um trecho do *template* é repetido para vários locais do código gerado / implementação de referência e no arquivo de mapeamento são inscritas todas as posições correspondentes dessa associação. As alterações introduzidas no código gerado são registradas no arquivo de *log* do *Fluorite* na forma de eventos. Quando o *Parser* é executado ocorre a busca das alterações registradas no arquivo de *log*, que também associa as informações encontradas com o arquivo de mapeamento e as replica para o *template* na posição correspondente.

De forma geral, se o programador não confirmar a execução ou se forem detectadas modificações no código da aplicação relacionada com as informações obtidas do modelo de entrada, o Parser descartará, mas refará as posições do mapeamento que foram modificadas para não perder a sincronia com o template. Inclusive, em operações que envolvem mecanismo de repetição, o Parser corrigirá a somente a primeira alteração envolvida, desabilitando os próximos mapeamentos relacionados para que não ocorra duplicação no template e, caso for confirmada alguma modificação que abrange mais de um mapeamento na sequência o Parser os transformará em um único.

Com relação a esses procedimentos específicos que foram adotados neste projeto durante o desenvolvimento do *Parser* é importante ressaltar que foram definidos em discussões internas com o grupo de pesquisa, cujo fator determinante nessa avaliação foi viabilizar o funcionamento do protótipo de migração automática

de código. A investigação sobre a existência de alternativas melhores para lidar com essas situações, em um processo mais dinâmico e interativo, é trabalho futuro.

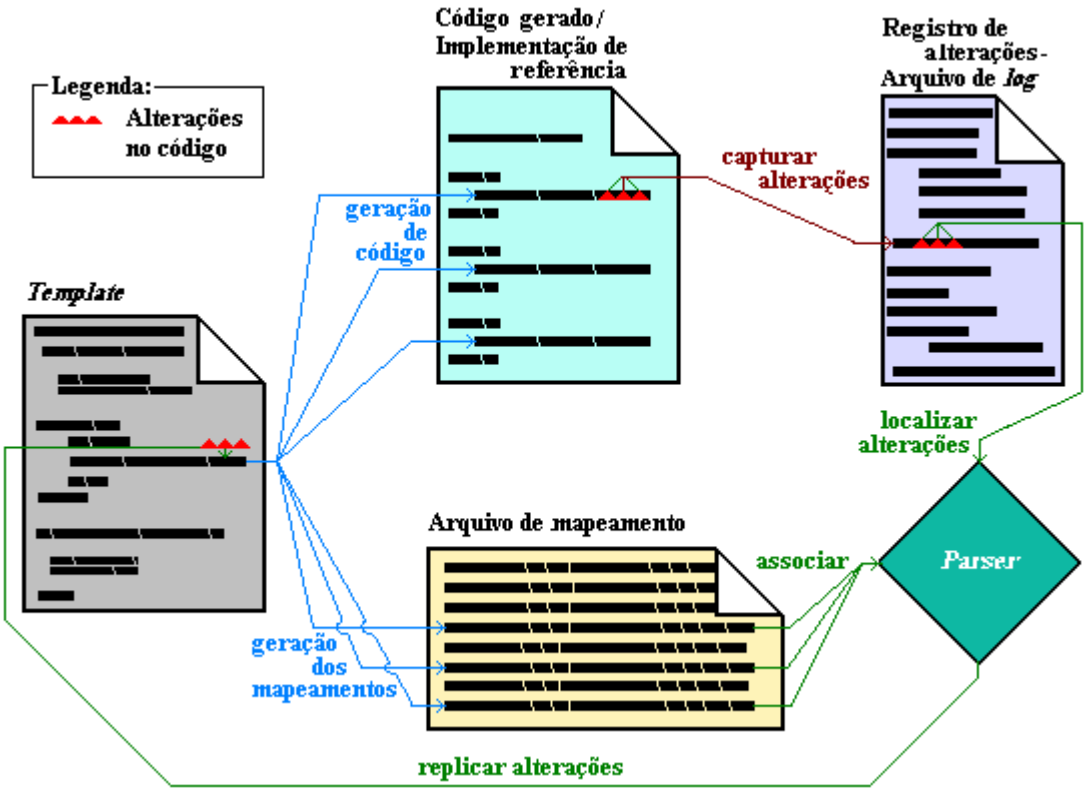

Figura 3.13 - Mecanismo de migração automática de código para mapeamento do tipo repetição

3.5 Considerações Finais

Neste capítulo foi apresentado o detalhamento do processo de desenvolvimento do protótipo de migração automática de código, que embora esteja na fase inicial, representa um passo importante na exploração de mecanismos que facilitem o desenvolvimento e a manutenção de geradores de código.

Foi demonstrado também a tentativa de implementar as técnicas mencionadas na proposta inicial, que apesar de não atingir o êxito esperado esses esforços despendidos contribuíram para o entendimento do problema e para a

indicação de suas limitações, inclusive podem servir de alerta aos demais pesquisadores da área.

O protótipo foi submetido a um estudo empírico com a finalidade de identificar se as expectativas esperadas com a abordagem de migração automática de código foram atingidas. A avaliação do protótipo é apresentada no próximo capítulo.

Capítulo 4

AVALIAÇÃO DA ABORDAGEM

Este capítulo apresenta o experimento de migração de código que foi realizado com o objetivo de avaliar o efeito da abordagem proposta no tempo gasto para a migração de código, do ponto de vista do desenvolvedor de software no contexto de manutenção/evolução em um projeto de desenvolvimento de software orientado a modelos.

A avaliação do estudo empírico foi estabelecida por meio da comparação das diferenças entre o uso do protótipo de migração automática de código e o processo manual, quando utilizados para efetuar correções nos *templates* do gerador, de forma a sincronizá-los com as alterações efetuadas código gerado (implementação de referência), em termos de esforço. O planejamento do estudo foi realizado considerando as diretrizes propostas por Wohlin et al. (2000).

4.1 Definição do Estudo Empírico

- **Objetivo**

O objetivo do estudo foi comparar o esforço em sincronizar os arquivos de *templates* do gerador com o código gerado, após serem introduzidas modificações no código da implementação, da forma manual e com a utilização do protótipo.

- **Sujeito do Estudo**

Um gerador de código completo baseado em *templates*, contendo os *templates* geradores elaborados com o mecanismo JET, que produzem uma aplicação *web* completa, a qual faz o papel da implementação de referência. Três modelos de entrada são utilizados pelo gerador: um modelo de dados construído por meio do GMF (*Graphical Modeling Framework*), um modelo de navegação construído com a linguagem *Xtext* (EFFTINGE; VOELTER, 2006) e um modelo de *features* simples, baseado em EMF (*Eclipse Modeling Framework*). Esse gerador desenvolvido por Lucrédio (2009), com foco no paradigma MDE, possibilitou que a avaliação das técnicas de migração de código ocorresse no contexto dessa abordagem.

As informações básicas da composição desse gerador, inclusive os arquivos dos modelos de entrada e da aplicação *web* gerada foram incluídas no Apêndice A e constam no documento “Elementos do Projeto” (Figuras A.2 a A.4).

Enfoque Quantitativo

O enfoque quantitativo envolve a identificação de esforço com base nos tempos gastos por cada participante para concluir as tarefas de sincronização do código da implementação com os *templates*.

- **Perspectiva**

Os experimentos foram conduzidos sob a perspectiva de desenvolvedores de geradores de código que necessitem implementar mudanças nos *templates* para reestabelecer a sincronização no processo de migração de código.

- **Objeto de Estudo**

O objeto de estudo deste capítulo é o esforço para concluir um processo de migração de código após alterações serem efetuadas no código gerado.

4.2 Planejamento do Estudo

O experimento foi planejado considerando a seguinte questão: “A técnica implementada com o protótipo diminui o tempo de conclusão das tarefas de migração de código quando comparado com a migração manual, realizada de forma *ad hoc*?”. Essa questão é então analisada no término do capítulo para avaliar se ela foi respondida com sucesso.

Foi solicitado aos participantes executar tarefas de manutenção no gerador de código visando retornar o sincronismo dos *templates* do gerador após serem introduzidas modificações no código da aplicação, utilizando ambas as técnicas. Foram selecionadas 05 (cinco) tarefas de migração de código, cujos seus detalhes e os procedimentos para a execução estão relacionados nos formulários das tarefas constantes do Apêndice A (Figuras A.5 a A.14). A tarefa piloto foi adotada para o treinamento dos participantes e as demais foram submetidas à avaliação, sendo:

Tarefa Piloto - Manual (Figura A.5), Protótipo (Figura A.6);
Tarefa 1: Manual (Figura A.7), Protótipo (Figura A.8);
Tarefa 2: Manual (Figura A.9), Protótipo (Figura A.10);
Tarefa 3: Manual (Figura A.11), Protótipo (Figura A.12);
Tarefa 4: Manual (Figura A.13), Protótipo (Figura A.14);

As tarefas compreenderam atividades básicas de manutenção em aplicações *web*, englobando correções, inserções e modificações no seu código. Cada tarefa foi elaborada de forma específica, considerando locais diferentes no código da aplicação, que foram selecionados para os participantes realizarem a alteração e medir o tempo gasto com a técnica utilizada. Entretanto, o pressuposto fundamental deste experimento foi avaliar a técnica no contexto geral das tarefas e não analisar a técnica em relação às especificidades das tarefas e, embora fosse necessário estabelecer tarefas diferentes para não ocorrer interferência nos resultados com o aprendizado obtido pelo participante, cada uma dessas tarefas foi executada com a técnica manual e com o protótipo, de forma que suas diferenças não interferissem na análise global. Porém, nada impede que este experimento busque indícios da interferência do tipo de tarefa na comparação das técnicas, que poderá servir para

planejar um futuro refinamento desta pesquisa, todavia focada na avaliação específica das tarefas de migração de código.

Os seguintes dados pertinentes a este estudo foram coletados e analisados:

- Técnica de migração de código empregada;
- Tarefa executada;
- Hora do início de execução de cada tarefa; e
- Hora do término de execução de cada tarefa.

O item "Técnica de migração de código empregada" é utilizado para definir se o participante do estudo utilizou o protótipo de migração automática ou um processo manual *ad hoc* durante a correção e sincronização do código com o *template*. "Tarefa executada" corresponde à atividade estabelecida a ser realizada pelo participante. Os itens "Hora do início" e "Hora do término" de execução de cada tarefa são usados para registrar o momento de início e fim de cada execução, cujos valores foram considerados para calcular o tempo gasto em cada processo.

A hora do sistema constante em cada microcomputador foi utilizada para o participante informar o início e o término da operação no formulário da tarefa, que foi monitorado durante a execução das tarefas e confrontado com o arquivo de *log* gerado pelo *Fluorite* durante a execução do *Eclipse*, para conferir maior garantia de precisão dos tempos informados.

Os arquivos de *log* das máquinas utilizadas pelos participantes foram coletados após o término das atividades, possibilitando verificar se o tempo informado durante a operação estava dentro do intervalo de tempo (*timestamp*) de determinados eventos capturados pelo *Fluorite*, pois é sempre criado um arquivo de *log* a cada vez que o *Eclipse* é iniciado.

O registro de tempo considerado importante para a avaliação desse experimento foi estabelecido em cada formulário de execução das tarefas (Apêndice A: Figuras A.5 a A.14) e compreendeu somente o intervalo de tempo em que o participante dedicou-se efetivamente a corrigir o *template.* Esse período inclui procedimentos individuais que não são registrados pelo *Fluorite*, tais como: o tempo dedicado ao estudo da mudança nos códigos da aplicação e no gerador, a realização de buscas, as tentativas de correção até a conclusão do processo. Por outro lado, muitos eventos que embora registrados no arquivo de *log* não foram

necessários, entre eles o tempo que o participante utilizou para iniciar o ambiente de trabalho no *Eclipse*, fazer a geração inicial da aplicação, iniciar a aplicação *web* no servidor, verificar se tudo está funcionando e realizar a leitura do problema. Portanto, foi necessário deixar o participante à vontade para fazer a correção solicitada sem nenhuma intervenção, definindo-se que o início da contagem fosse a partir do momento em que o participante se dedicasse efetivamente a realizar a tarefa e finalizava quando obtinha êxito na sua conclusão.

Ao final do experimento foi entregue aos participantes um formulário com questões específicas sobre a utilização do protótipo, que tiveram o objetivo de obter a opinião individual com relação ao desempenho do protótipo nas tarefas de migração de código e possibilitar aos participantes relatar a existência de dificuldades para executar as tarefas e a ocorrência de erros, inclusive realizar comentários e sugerir melhorias. O formulário de informações dos participantes sobre o experimento consta no Apêndice A (Figura A.15).

4.2.1 Seleção de Contexto

O experimento ocorreu em ambiente universitário, sendo realizado com alunos de Ciência da Computação que são aqui chamados de participantes, no Laboratório de Ensino do Departamento de Computação da Universidade Federal de São Carlos. Oito alunos participaram do experimento, sendo seis alunos de pós-graduação e dois alunos de graduação.

Durante o experimento, os participantes utilizaram um gerador de código baseado em templates elaborado com o mecanismo JET (LUCREDIO, 2009). Esse gerador permitiu a montagem de um ambiente de desenvolvimento compatível com o paradigma MDE, assegurando que a avaliação das técnicas de migração de código ocorresse no contexto da abordagem proposta.

4.2.2 Formulação de Hipóteses

Na Tabela 4.1 estão listadas as hipóteses para o experimento de migração de código. As hipóteses foram criadas para comparar a produtividade, com base no

tempo utilizado, do protótipo de migração automática de código com a forma manual *ad hoc*, durante a sincronização dos artefatos.

$H0_m$	Não há diferença entre usar o protótipo de migração automática de código e utilizar um processo manual *ad hoc* para concluir um processo de migração com sucesso em termos de produtividade (tempo). Portanto, as técnicas são equivalentes. $Tm_m - Tp_m \approx 0$
Hp_m	Há uma diferença positiva entre usar o protótipo de migração automática de código e utilizar um processo manual *ad hoc* para concluir um processo de migração com sucesso em termos de produtividade (tempo). Portanto, a técnica manual leva mais tempo do que o protótipo de migração automática de código. $Tm_m - Tp_m > 0$
Hn_m	Há uma diferença negativa entre usar o protótipo de migração automática de código e utilizar um processo manual *ad hoc* para concluir um processo de migração com sucesso em termos de produtividade (tempo). Portanto, a utilização da técnica manual leva menos tempo do que a protótipo de migração automática de código. $Tm_m - Tp_m < 0$

Tabela 4.1 - Hipóteses para o Estudo de Migração de Código

Na Tabela 4.1 são listadas duas variáveis: Tm_m e Tp_m, onde Tm_m representa o tempo total necessário para migrar o código utilizando a técnica manual e Tp_m representa o tempo total necessário para migrar o código utilizando o protótipo.

Na Tabela 4.1 são listadas três hipóteses: $H0_m$, Hp_m e Hn_m. A hipótese $H0_m$, também chamada de hipótese nula, é verdadeira quando ambas as técnicas são equivalentes; portanto, o tempo necessário para completar o processo utilizando a técnica manual menos o tempo necessário para completar o processo utilizando o protótipo da técnica de migração automática é aproximadamente zero.

A hipótese Hp_m é verdadeira quando a técnica manual leva mais tempo do que o protótipo; portanto, o tempo necessário para completar o processo utilizando a

técnica manual menos o tempo necessário para completar o processo utilizando o protótipo da técnica de migração automática é positivo.

A hipótese Hn_m é verdadeira quando a técnica manual leva menos tempo do que a técnica do protótipo; portanto, o tempo necessário para completar o processo utilizando a técnica manual menos o tempo necessário para completar o processo utilizando o protótipo da técnica de migração automática é negativo.

Considerando os intervalos de um valor real das hipóteses, só há uma hipótese verdadeira para um experimento de sucesso. A hipótese nula possui precedência com relação às outras, visto que isso depende da margem de erro estipulada no teste estatístico.

4.2.3 Seleção de Variáveis

As variáveis são divididas em duas categorias: dependentes e independentes.

As variáveis dependentes são aquelas que são analisadas neste experimento. Dessa forma, é analisado o tempo necessário para completar a tarefa de migração de código para sincronização do *template* com o código.

As variáveis independentes são controladas e manipuladas pelos pesquisadores, neste caso, são “Técnica de migração de código empregada” e “Tarefa executada”.

4.2.4 Critério de Seleção dos Participantes

Os participantes selecionados eram alunos dos cursos de graduação e de pós-graduação em Ciência da Computação, todos envolvidos em projetos de Engenharia de Software, com um mínimo de experiência em abordagens dirigidas a modelos. Dois alunos de graduação e seis de pós-graduação. Nenhum participante do experimento foi omitido neste trabalho.

Esses participantes foram selecionados por possuírem experiência prévia com o desenvolvimento de software orientado a modelos e com a manutenção de aplicações *web*. Durante o experimento, os participantes tiveram que efetuar

determinadas correções nos *templates* para sincronizá-los com as alterações efetuadas no código da aplicação *web*. As alterações foram estabelecidas conforme os procedimentos contidos nas tarefas apresentadas.

4.2.5 Projeto do Estudo

O experimento foi organizado de forma que todos os tratamentos tivessem o número de participantes igual e o projeto do experimento ficasse balanceado.

Cada tarefa foi realizada simultaneamente nas duas formas distintas: com a técnica manual e com uso do protótipo.

Os participantes foram divididos em dois grupos. Cada um dos grupos foi composto por um estudante de graduação e três estudantes de pós-graduação. Cada grupo foi balanceado considerando também os dados obtidos no Formulário de Caracterização, que consta no Apêndice (Figura A.1), de forma que cada grupo tivesse uma distribuição homogênea, considerando seus conhecimentos e experiências com a utilização de aplicativos usados no MDE e em desenvolvimento e manutenções de aplicações *Web*. Na Tabela 4.2 são listadas as fases executadas no estudo empírico.

Tarefas	Grupo 1	Grupo 2
Tarefa Piloto (Treinamento)	Manual	Manual
	Protótipo	Protótipo
Tarefa 1	Manual	Protótipo
Tarefa 2	Protótipo	Manual
Tarefa 3	Manual	Protótipo
Tarefa 4	Protótipo	Manual

Tabela 4.2 - Projeto do Estudo Migração de Código

Cada tarefa representada na Tabela 4.2 está associada ao grupo de participantes e a técnica empregada para migrar o código. Por exemplo, durante a execução da Tarefa 1, os participantes do primeiro grupo utilizaram a técnica manual e ao mesmo instante os participantes do segundo grupo utilizaram o protótipo de migração de código para executar essa mesma tarefa. O objetivo dessa inversão foi minimizar o efeito do aprendizado de uma tarefa e/ou técnica utilizada nos resultados.

4.2.6 Instrumentação

Para que os participantes aprendessem a utilizar o gerador de código e tivessem um esquema para acompanhar o treinamento foi fornecido o documento "Elementos do projeto" (Apêndice A), contendo os detalhes dos principais elementos que constituem esse gerador e as funções dos comandos de execução do *Parser,* que seriam utilizados para realizar as tarefas de migração de código.

As instruções para a execução das tarefas foram fornecidas na forma de formulários. Cada formulário continha os procedimentos para a realização de uma tarefa e incluíam além dos detalhes da instrução, o nome e o caminho do arquivo do código da aplicação, cuja alteração deveria ser propagada para o gerador. Para cada técnica o formulário fornecido foi relativo à técnica de migração empregada, contendo o espaço para o participante indicar o tempo de início e término da operação. Os formulários foram importantes para especificar o tipo de modificação a ser efetuada no *template* responsável pela geração e, também, para que os participantes obedecessem a sequência de operações definidas. Os formulários e outros documentos utilizados durante os experimentos estão disponíveis no Apêndice A.

Assim, cada tarefa foi realizada de forma manual por um grupo e com o uso do protótipo pelo outro, sendo invertido na próxima tarefa. A complexidade do processo de migração varia conforme a tarefa fornecida, cujos detalhes estão contidos nos formulários de execução das tarefas e juntados no Apêndice A. Portanto, concluir o processo de migração para cada tarefa exigia dos participantes

a modificação do *template* para reproduzir o código alterado, conforme especificado na tarefa.

4.3 Operação

4.3.1 Preparação

Inicialmente, foi entregue a todos os participantes o documento “Elementos do projeto” para consulta e acompanhamento, em seguida foi realizada no laboratório que ocorreu o experimento a apresentação do gerador de código baseado em *templates*, da função da implementação de referência no processo de migração de código e a importância de propagar suas alterações para o gerador, da dinâmica da combinação de modelos de entrada para alternar a geração do código, do protótipo desenvolvido que implementa a técnica automática de migração de código e, finalmente, ensinado como realizar manutenção no gerador e propagar as alterações do código para o *template*, com ambas as técnicas.

Os participantes responderam o formulário de caracterização para formação dos grupos de forma balanceada, com questões em geral relacionadas com conhecimentos da abordagem, utilização de aplicativos usados no MDE e em desenvolvimento e manutenções de aplicações *Web*.

Em seguida, os alunos foram treinados como realizar alteração no *template* com as duas técnicas de migração de código, mesmo aqueles eles já possuíam experiência na forma manual. Executaram a tarefa piloto simulando o experimento de forma manual e com o uso do protótipo, treinaram a anotação do tempo e foram tirando suas duvidas. Os participantes foram divididos em grupos considerando a caracterização.

No piloto, o local de correção no *template* e no código gerado foi diferente daqueles que seriam realizados no experimento real, porém, equivalentes e os participantes eram autorizados a fazer perguntas sobre o que não haviam compreendido durante o treinamento. Isso afetaria a validade dos valores, portanto, os dados dessa atividade foram utilizados somente para a finalidade de treinamento.

4.3.2 Execução

Durante o experimento real, os participantes realizaram tarefas de modificação em alguns locais da aplicação e do *template*, com técnicas diferentes para cada grupo. As técnicas foram invertidas nos grupos para executarem a próxima tarefa, ou seja, o grupo que executou a primeira tarefa com a técnica manual, na segunda executou com o protótipo, até atingiram o total de quatro tarefas. Portanto, cada grupo executou quatro tarefas diferentes, sendo duas com a técnica manual e duas com o protótipo.

Os participantes informaram o tempo de início e término com base no relógio do microcomputador utilizado. Durante a execução da tarefa não foi permitida ao participante nenhuma ajuda exterior. O início das tarefas registrado pelos participantes variou conforme o tempo que gastaram para inicializar o ambiente de trabalho e realizar a leitura das instruções da tarefa antes de iniciar a execução, também, para evitar que o participante ficasse ocioso, à medida que ia terminando uma tarefa era passado à próxima tarefa com a técnica alternada.

Assim que um participante concluía a última tarefa ele recebia o formulário de Informações sobre o experimento, contendo cinco perguntas relacionadas ao seu contato com o protótipo, cuja finalidade foi obter um comentário específico de cada participante após a utilização do protótipo, conforme descrito na seção 4.3.

4.3.3 Validação de Dados

Os formulários das tarefas que foram preenchidos pelos participantes, com os campos início e término da execução da tarefa, foram confrontados após o final do experimento com o intervalo de tempo registrado nos arquivos de *log* do *Fluorite*, gerados na máquina utilizada por cada participante durante a utilização do *Eclipse*.

Para obter a confirmação de que os participantes executaram as tarefas integralmente e a migração foi bem sucedida, foram analisados ao final experimento os códigos dos arquivos do gerador e da aplicação *web* em todas as máquinas utilizadas, verificando se continham as modificações solicitadas e se os resultados estavam corretos.

4.3.4 Coleta de Dados

Os tempos registrados durante os processos de migração de código com ambas as técnicas durante a execução do experimento estão listados na Tabela 4.3.

Grupo	Participante	Tarefa	Técnica	Início hh:mm	Término hh:mm	Tempo hh:mm
G2	P8	T3	P	16:30	16:32	00:02
G1	P1	T2	P	16:29	16:32	00:03
G1	P3	T4	P	17:10	17:13	00:03
G2	P5	T1	P	16:06	16:10	00:04
G2	P6	T1	P	16:09	16:13	00:04
G2	P5	T2	M	16:22	16:26	00:04
G2	P5	T3	P	16:29	16:33	00:04
G2	P6	T3	P	16:30	16:34	00:04
G1	P1	T4	P	16:46	16:50	00:04
G2	P8	T2	M	16:22	16:27	00:05
G2	P7	T3	P	16:40	16:45	00:05
G1	P4	T4	P	16:48	16:53	00:05
G2	P6	T4	M	16:39	16:45	00:06
G2	P8	T1	P	16:07	16:14	00:07
G1	P4	T2	P	16:26	16:33	00:07
G2	P6	T2	M	16:21	16:28	00:07
G1	P2	T4	P	16:56	17:04	00:08
G1	P2	T2	P	16:21	16:31	00:10
G1	P3	T2	P	16:30	16:40	00:10
G1	P1	T3	M	16:34	16:44	00:10
G2	P5	T4	M	16:45	16:56	00:11
G1	P2	T1	M	16:05	16:17	00:12
G2	P7	T1	P	16:08	16:20	00:12
G1	P1	T1	M	16:09	16:22	00:13
G2	P7	T2	M	16:23	16:37	00:14
G1	P4	T3	M	16:34	16:48	00:14
G1	P3	T1	M	16:10	16:26	00:16
G2	P7	T4	M	16:47	17:03	00:16
G1	P4	T1	M	16:08	16:25	00:17
G1	P3	T3	M	16:50	17:07	00:17
G1	P2	T3	M	16:34	16:55	00:21
G2	P8	T4	M	16:42	17:03	00:21

Tabela 4.3 - Tempos de execução das tarefas

A Tabela 4.3 contém os dados coletados do experimento, incluindo os tempos que cada um dos oito participantes, divididos em dois grupos, gastou para realizar cada uma das quatro tarefas e a técnica de migração de código utilizada, que pode ser “M” para Manual e “P” para Protótipo. O Participante “P2” do grupo “G1” e o “P7” do grupo “P2” são alunos de graduação e os demais alunos de pós-graduação.

Ao final das tarefas, foi recolhido dos participantes o formulário de informações sobre o experimento com seus comentários sobre a utilização do protótipo no processo de migração de código.

4.4 Análise de Dados e Interpretação

Os dados do experimento, constantes na Tabela 4.3, estão ordenados pelo tempo utilizado para completar a tarefa. Ao analisar a tabela, nota-se que as linhas que representam o uso do protótipo, identificadas pela letra “P”, são encontrados nos primeiros cinco lugares. As linhas da técnica manual, as quais são identificadas pela letra “M”, são os últimos nove resultados.

Para permitir uma melhor visualização dos resultados, também é mostrada uma versão gráfica da Tabela 4.3 em forma de quatro gráficos de barras, cada um representando a execução de uma tarefa de migração de código. Assim, a Figura 4.1 representa o gráfico da Tarefa 1; a Figura 4.2 representa o gráfico da Tarefa 2; a Figura 4.3 representa o gráfico da Tarefa 3 e a Figura 4.4 representa o gráfico da Tarefa 4.

Uma informação importante extraída da Tabela 4.3 é que comparando a soma dos tempos gastos de cada participante para executar as tarefas de forma manual e com o uso do protótipo, nenhum participante foi capaz de realizar a migração de código de forma mais rápida ao utilizar a técnica manual.

Na Tabela 4.4 estão listadas as médias de tempo e as suas proporções. A primeira coluna lista qual é a técnica considerada no cálculo da média, que pode ser “Manual” ou “Protótipo”. A segunda coluna lista qual é a tarefa considerada no cálculo da média dos tempos gastos entre os participantes do grupo. Na coluna “Média” há os resultados do cálculo considerando a tarefa, o grupo e a técnica

executada em cada linha. Na coluna “Soma das Médias” é mostrado o valor das médias para cada técnica somando as tarefas. Na coluna “Porcentagem” é mostrada a relação entre o valor das somas parciais com o total.

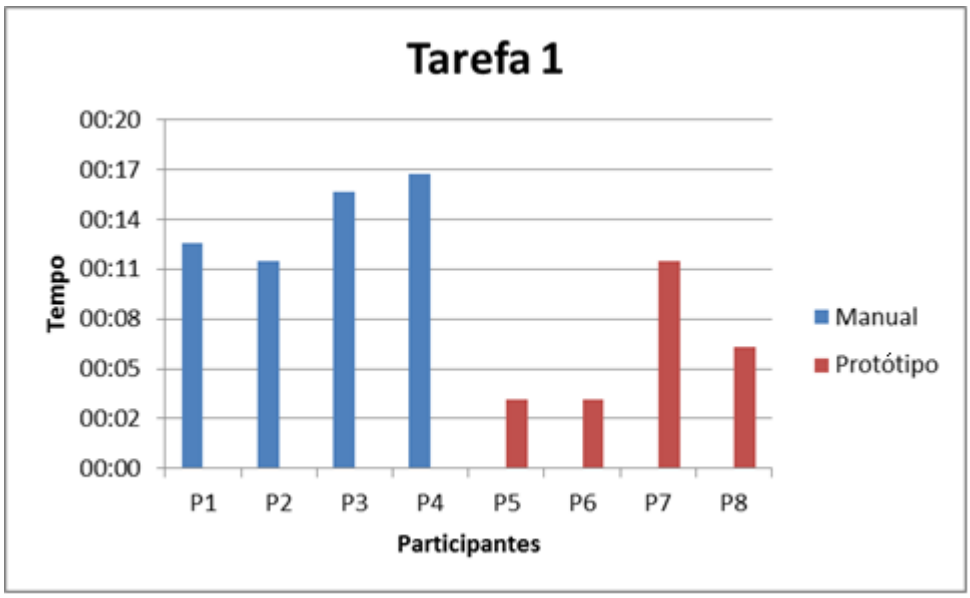

Figura 4.1 - Gráfico de barras para a Tarefa 1 de migração de código

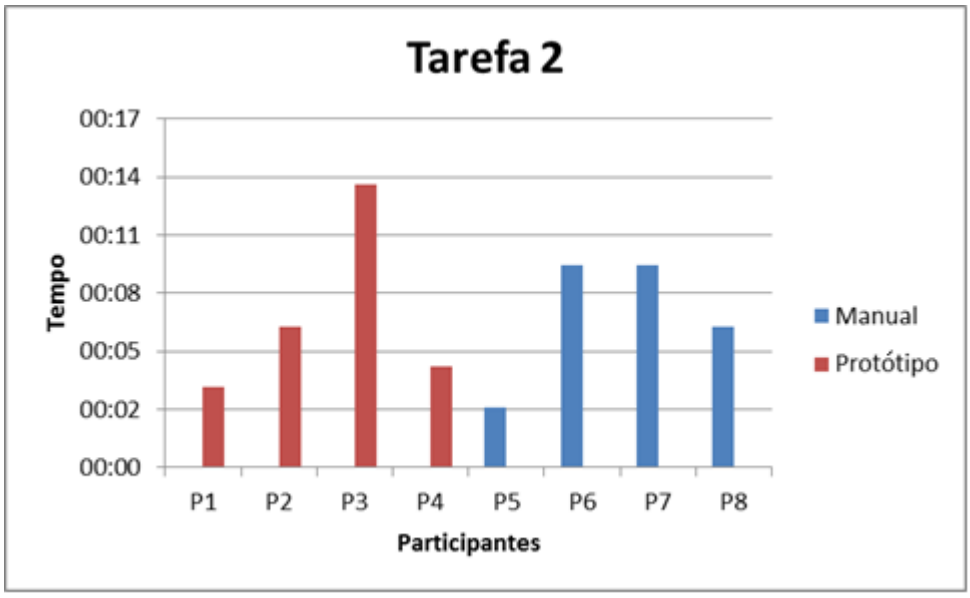

Figura 4.2 - Gráfico de barras para a Tarefa 2 de migração de código

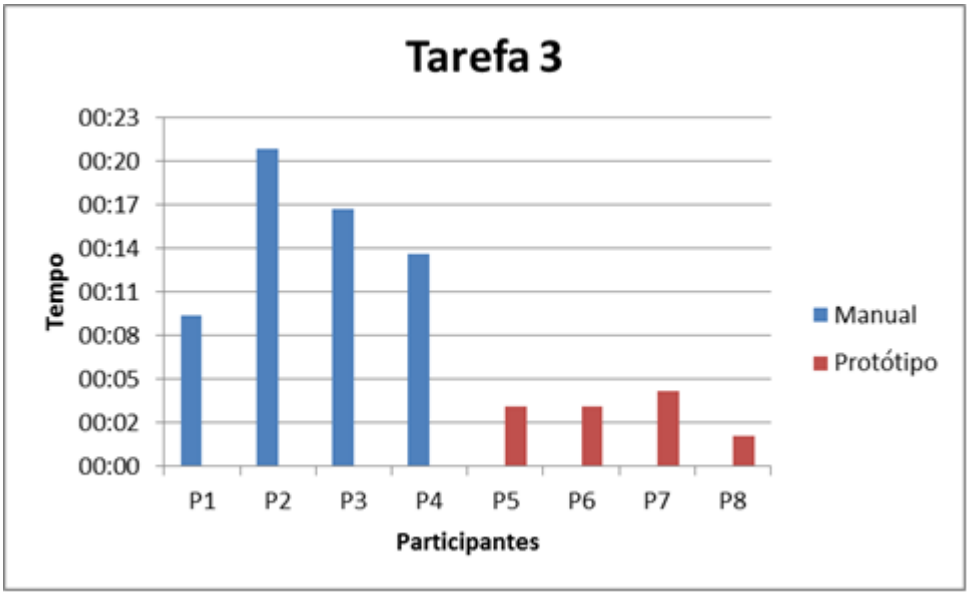

Figura 4.3 - Gráfico de barras para a Tarefa 3 de migração de código

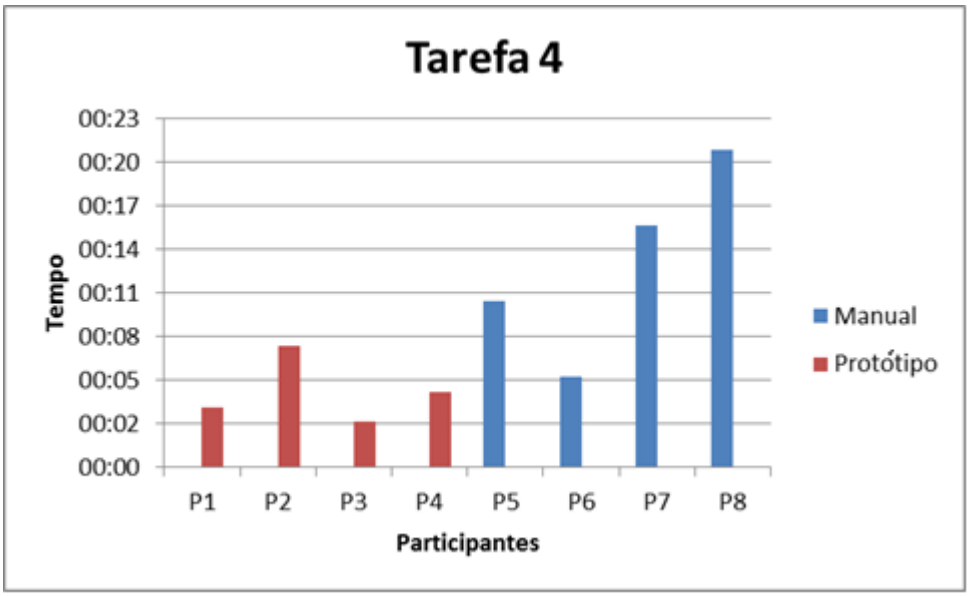

Figura 4.4 - Gráfico de barras para a Tarefa 4 de migração de código

Analisando os gráficos das tarefas, observa-se maior rapidez com o uso do protótipo. Somente no gráfico referente à execução da Tarefa 2 foi constatada uma paridade entre os tempos gastos com o uso de ambas as técnicas. Essa equivalência confirma-se na Tabela 4.4 de médias, onde o grupo que a realizou de forma manual obteve a média de tempo similar à do grupo que realizou com o

protótipo. Em contraste, a Tarefa 3 foi aproximadamente cinco vezes mais rápida com o uso do protótipo.

Técnica	Tarefas	Grupo	Média	Soma das Médias	Porcen-tagem
Manual	T1	G1	00:14	00:51	68,92%
	T2	G2	00:07		
	T3	G1	00:15		
	T4	G2	00:13		
Protótipo	T1	G2	00:06	00:23	31,08%
	T2	G1	00:07		
	T3	G2	00:03		
	T4	G1	00:05		
			Total:	01:14	100,00%

Tabela 4.4 - Médias de tempo de migração de código

Portanto, considerando o tempo médio dos participantes de ambos os grupos para completar as quatro tarefas de forma manual dividido pelo tempo médio utilizado para completar as quatro tarefas com o protótipo, a técnica manual levou aproximadamente 121,74% mais tempo para ser concluída do que com o uso do protótipo, ou seja, um pouco mais que o dobro.

Com relação às informações coletadas dos participantes ao final do experimento, verificou-se que todos admitiram que a utilização do protótipo facilitou a tarefa de migração de código, onde alguns destacaram a facilidade de não precisar procurar trechos de alteração no *template*. Quanto às dificuldades informadas, nenhuma foi dirigida ao protótipo, em geral, foram mencionados problemas de atualização (*Refresh)* no *Eclipse*. Os comentários sobre a utilidade do retorno das informações (caixas de diálogo e terminal) foram positivos e, para a melhoria do protótipo houve a sugestão de permitir a personalização das informações no console.

4.5 Teste de Hipóteses

Foram aplicados testes estatísticos no conjunto de dados experimento. Inicialmente, utilizou-se o teste de normalidade de *Shapiro-Wilk* para verificar se a distribuição de probabilidade associada aos dados apresenta uma distribuição normal. Em seguida, foi utilizada a técnica *ANOVA* (*Analise Of Variance*) que tem o objetivo de comparar médias oriundas de grupos diferentes. Essa comparação é feita a partir da análise da dispersão presente no conjunto de dados: testar se a variabilidade dentro dos grupos é maior que a existente entre os grupos e indicar se os fatores exercem influência em alguma variável dependente. Por fim, o teste de múltiplas amplitudes *Post Hoc de Duncan* foi aplicado para contrastar a magnitude das diferenças detectadas, por meio da comparação de todos pares de médias no conjunto de dados.

A Tabela 4.5 apresenta os resultados do teste de normalidade de *Shapiro-Wilk*, com os dados distribuídos em intervalos de 5 minutos. O resultado obtido de P-valor=0,927635 indica que os dados provêm de uma distribuição normal, considerando o nível de significância estatística estabelecido em 5% ($p < 0,05$).

TESTES DE NORMALIDADE	
DADOS DO PROCESSO	
Estatística: Shapiro-Wilk	**0,978716**
P-valor	**0,927635**

Tabela 4.5 - Teste *Shapiro-Wilk*

Para certificar-se que não houve dispersão entre o grupo de participantes utilizou-se a técnica *ANOVA ONEWAY.* A Tabela 4.5 apresenta os resultados desse teste estatístico, onde o resultado obtido de p=0,162114 permite considerar que as variâncias dos participantes subjacentes aos dois grupos em análise são iguais, referente ao nível de significância em 5%. Portanto, foi obtida a seguinte conclusão: A *ANOVA ONEWAY* não detectou diferenças significativas entre os grupos de

participantes. Por isso, a análise pode continuar a ser realizada, considerando-se todos os participantes como grupo único.

ANOVA - Univariate Tests of Significance for Tempo Sigma-restricted parameterization Effective hypothesis decomposition					
	SS	Degr. of Freedom	MS	F	p
Intercept	0,00132	1	0,00132	92,97114	0
Grupo	0,000029	1	0,000029	2,05433	0,162114
Error	0,000426	30	0,000014		

Tabela 4.6 - Teste *ANOVA ONEWAY*

Em sequência, foi realizado o teste estatístico com a técnica *ANOVA TWO WAY* para verificar a dispersão dos fatores Técnica e Tarefa, representada na Tabela 4.6, obtendo a seguinte conclusão: *A ANOVA TWO WAY* detectou diferenças altamente significativas a 5% entre Técnicas (p=0,000033), mas não entre as Tarefas (p=0,455973). Entretanto, foi detectada uma interação significativa entre Técnica e Tarefas (p=0,038151) que precisa ser identificada. Para isso foi aplicada a Técnica *Post Hoc de Duncan*, para identificar o contraste das interações dos fatores Técnica e Tarefa no conjunto de dados do experimento.

ANOVA - Univariate Tests of Significance for Tempo Sigma-restricted parameterization Effective hypothesis decomposition					
Effect	SS	Degr. Of Freedom	MS	F	p
Intercept	0,00132	1	0,00132	181,2745	0
Técnica	0,000189	1	0,000189	25,9531	0,000033
Tarefa	0,00002	3	0,000007	0,8993	0,455973
Técnica*Tarefa	0,000072	3	0,000024	3,2828	0,038151
Error	0,000175	24	0,000007		

Tabela 4.7 - Teste *ANOVA TWO WAY*

Os resultados obtidos com o teste *Post Hoc de Duncan* estão contidos na Tabela 4.7 e apresentam como conclusão a existência de dois grupos de Técnicas-Tarefas distintos, sendo o primeiro grupo: C-T1, C-T3 e C-T4 que envolve todas as técnicas do tipo “M” (Manual), com exceção da T2 (Tarefa 2) que ficou junto do segundo grupo. O segundo grupo, que são diferentes do primeiro grupo, envolvem todas as técnicas “P” (Protótipo) e mais a técnica “C” na T2. Logo, a única interação foi a Técnica C na Tarefa 2 que apresentou resultados fora do seu respectivo grupo.

Considerando que os testes não encontraram diferenças significativas entre os grupos de participantes e nem entre as tarefas, e de acordo com o valor de p= 0,000033 constante na Tabela 4.6, que indicou uma diferença altamente significativa entre as técnicas, estatisticamente pode-se rejeitar a Hipótese $H0_m$, em que as técnicas deveriam ser equivalentes. Como a técnica com o uso do protótipo foi proporcionalmente superior a manual, a hipótese positiva é favorecida, ou seja, Hp_m, em que o uso da técnica manual leva mais tempo do que com o protótipo de migração automática de código.

Duncan test; variable Tempo
Approximate Probabilities for Post Hoc Tests
Error: Between MS = ,00001, df = 24,000

Técnicas-Tarefas	**C-T1**	**C-T3**	**C-T4**	**P-T1**	**C-T2**	**P-T2**	**P-T3**	**P-T4**
Médias	0,01007	0,01076	0,00938	0,00469	0,00521	0,00521	0,0026	0,00347

Técnicas-Tarefas	**C-T2**	**P-T1**	**P-T2**	**P-T3**	**P-T4**	**C-T1**	**C-T3**	**C-T4**
C-T1	0,0228	0,0168	0,0266	0,0017	0,0045	1,0000	0,7193	0,7193
C-T3	0,0122	0,0083	0,0138	0,0008	0,0021	0,7193	1,0000	0,4997
C-T4	0,0392	0,0321	0,0485	0,0037	0,0092	0,7193	0,4997	1,0000
C-T2	1,0000	0,7998	1,0000	0,2337	0,4151	0,0228	0,0122	0,0392
P-T1	0,7998	1,0000	0,7874	0,3133	0,5304	0,0168	0,0083	0,0321
P-T2	1,0000	0,7874	1,0000	0,2240	0,3997	0,0266	0,0138	0,0485
P-T3	0,2337	0,3133	0,224	1,0000	0,6534	0,0017	0,0008	0,0037
P-T4	0,4151	0,5304	0,3997	0,6534	1,0000	0,0045	0,0021	0,0092

Tabela 4.8 - Teste Post Hoc de Duncan

Com relação à análise da interação significativa detectada entre Técnicas e Tarefas, constatou-se estatisticamente que o esforço para executar a Tarefa 2 de

forma manual foi compatível ao grupo de tarefas executadas com o uso do protótipo. Dessa forma, foi identificada a existência de tarefas de migração de código que se realizadas com a técnica manual serão compatíveis com o uso do protótipo e, neste caso, o protótipo não oferecerá vantagens com relação ao consumo de tempo para realizar essa tarefa.

Cabe a observação que este experimento foi planejado com o objetivo principal de avaliar as técnicas no contexto geral das tarefas e não analisar a técnica em relação às especificidades das tarefas. Entretanto, a seleção de tarefas diferentes permitiu concluir estatisticamente que o protótipo apesar de mostrar-se eficiente na avaliação global das tarefas, não obteve vantagem em uma delas. Dessa forma, para que essa análise continue é necessário produzir um experimento específico para verificar a representatividade de cada tarefa no processo migração de código.

Finalmente, considerando que o experimento foi realizado sob condições controladas é importante prevenir que as conclusões a respeito dos resultados obtidos se restringem a um pequeno grupo de desenvolvedores de software em ambiente universitário, no qual o experimento foi conduzido e às características das tarefas escolhidas. Por questões de validade e para generalizar seus efeitos para um contexto mais amplo, é necessário que novos estudos com um número maior de equipes e diversidade de tarefas sejam realizados. Inclusive, a replicação deste experimento em ambiente industrial torna-se importante já que outros fatores ausentes no ambiente acadêmico poderão ser controlados e estudados. No entanto, tal cenário só será praticável com a evolução do protótipo para aumentar a usabilidade, conforme previsto para trabalhos futuros.

4.6 Ameaças à Validade

A seguir foram relacionados os itens que podem afetar os valores e a conclusão do experimento apresentados neste capítulo.

- **Validade interna**

Nível de experiência dos participantes: os variados níveis de experiência e conhecimento dos participantes que podem afetar os dados coletados. Para amenizar essa ameaça, os participantes foram divididos em dois grupos equilibrados, considerando se o aluno cursava graduação ou pós-graduação e o seu nível de experiência. Além disso, os alunos selecionados tinham como pré-requisito alguma experiência com a abordagem a ser avaliada, pois todos estavam envolvidos com projetos e pesquisas na área de Engenharia de Software. Durante o treinamento, os participantes foram treinados em como proceder à tarefa de migração de código de forma manual e utilizando o protótipo, o que, considerando seus conhecimentos, pode fazer com que os participantes tenham mais experiência com a técnica manual do que com o protótipo.

Produtividade em avaliação: existe a possibilidade de que isso poderia influenciar os resultados do experimento, porque os alunos muitas vezes tendem a pensar que estão sendo avaliados pelo resultado dos experimentos. Com o objetivo de amenizar esse impacto, foi explicado que ninguém estava sendo avaliado e suas participações seriam consideradas anônimas.

Instalações utilizadas durante o estudo: computadores e instalações diferentes poderiam afetar os tempos registrados. No entanto, os dois grupos utilizaram a mesma configuração, sistema operacional, gerador de código, aplicativos e em números iguais. Os participantes não foram autorizados a mudar de máquina durante a mesma atividade, o que significa que um participante não pode realizar uma tarefa de forma manual usando um computador diferente que foi usado para migrar o código da aplicação com o protótipo.

- **Validade por Construção**

Expectativa nas Hipóteses: os participantes já conheciam os pesquisadores e sabia que o protótipo código foi desenvolvido com o intuito de agilizar o processo de migração de código, o que reflete uma hipótese de um experimento. Esses itens podem afetar os dados do experimento e torná-lo menos imparcial. Para evitar

imparcialidade, foi requerido que os participantes mantivessem um ritmo constante durante todas as tarefas, independente da técnica utilizada.

- **Validade Externa**

Interação entre configuração e tratamento: existem possibilidades das tarefas promovidas não serem suficientemente precisas e abrangentes para representar todo o processo de migração de código para aplicações reais; do gerador de código e as tecnologias de desenvolvimento de software empregados não serem apropriados e/ou terem representatividade limitada e dos grupos de participantes do estudo não serem significativos para a população de desenvolvedores de software. As tarefas empregadas, embora todas de manutenção em trechos do código da aplicação tinham complexidades diferentes e foram elaboradas considerando manutenções básicas em aplicações *web* com base no mundo real, sendo avaliadas no geral e individualmente com relação às técnicas. Foi utilizado um gerador de código que produz uma aplicação *web* com todas as funcionalidades pertinentes e ferramentas atuais de desenvolvimento que normalmente são empregadas em ambientes industriais, tais como o *IDE Eclipse* e a linguagem de programação Java. Com relação à representatividade dos participantes somente foram selecionados alunos que tinham experiências com o desenvolvimento de aplicativos para *web*.

- **Validade de Conclusão**

Confiabilidade das métricas: este item é relacionado com a precisão das métricas empregadas durante as atividades. Para amenizar essa ameaça, apenas foi considerado o tempo informado no formulário de execução das tarefas, com base na hora do sistema. Os tempos capturados pelo *Fluorite* registrados no arquivo de *log* serviram somente para confirmar se o tempo informado estava nesse intervalo de execução de comandos, sendo todos os tempos validados após suas análises. O formulário de informações do experimento preenchido pelo participante ao final do experimento ateve-se somente à intenção de capturar suas impressões sobre o protótipo, cujos comentários de que o protótipo facilitou o processo de migração de código foi unânime.

Baixo poder estatístico: O número de participantes pode ser pequeno para gerar dados estatísticos confiáveis. Para amenizar essa ameaça, os dados foram analisados com a aplicação de quatro testes: *Shapiro-Wilk, ANOVA ONEWAY*, *ANOVA TWO WAY* e *Post Hoc de Duncan*.

4.7 Considerações Finais

O experimento de migração de código foi apresentado neste capítulo, que terminou com bons resultados da abordagem proposta, juntamente com comentários positivos dos participantes relacionados à utilização do protótipo de migração automática de código. Entretanto, evidenciou a existência de tarefas de correção no gerador em que não há benefícios em usar o protótipo para o esforço de migração de código.

A partir dos resultados que foram obtidos no experimento tornou-se possível validar o protótipo desenvolvido para a migração automática de código, mostrando numericamente seu desempenho através da comparação das técnicas nesse processo. Portanto, foi identificado que realmente o protótipo cumpriu seus objetivos, pois em termos numéricos, esperava-se reduzir o tempo gasto na migração de código.

Devido às limitações apresentadas sobre o experimento é necessário prevenir com relação à validade de suas conclusões, principalmente porque a sua amostra restringiu-se um pequeno grupo de participantes em ambiente universitário. Por questões de validade e para generalizar seus efeitos para um contexto mais amplo, é necessário que novos estudos, envolvendo também outros ambientes e com um número maior de equipes e de tarefas sejam realizados.

Capítulo 5

TRABALHOS RELACIONADOS

Apesar de não encontrar na literatura nenhum estudo específico sobre o problema da migração de código de uma implementação de referência para um gerador de código baseado em *templates*, neste capítulo é apresentada uma linha de pesquisa que se aproxima desse problema conhecida como engenharia ida-e-volta (*round-trip engineering* ou RTE), com a finalidade de associar os mecanismos responsáveis em prover o sincronismo dos artefatos, pois são problemas similares, ainda que em diferentes níveis de abstração (RTE trabalha principalmente em nível de modelo, enquanto neste projeto os artefatos são basicamente arquivos textuais de implementação).

5.1 Abordagens RTE

Nas abordagens RTE são estudadas maneiras para se manter o sincronismo entre artefatos de origem e destino de uma transformação de software (HETTEL; LAWLEY; RAYMOND, 2008; ANTKIEWICZ; CZARNECKI, 2006). Por exemplo, considere-se uma ferramenta que transforma um diagrama Entidade-Relacionamento em scripts de definição de dados. Através dessa técnica, é possível modificar diretamente os *scripts* e automaticamente propagar as mudanças de volta para o diagrama de origem. Da mesma forma é possível sincronizar um modelo OO e um modelo ER (PAESSCHEN; MEUTER; D'HONDT, 2005).

Considerando que as abordagens RTE podem contribuir para solucionar o problema, são apresentadas a seguir algumas dessas abordagens que propõem a sincronização dos artefatos na MDE. O objetivo é identificar as técnicas e os mecanismos utilizados para resolver ou mesmo aliviar o problema da perda de sincronismo entre os artefatos responsáveis pela geração de código, bem como, na medida do possível, encontrar suas vantagens e limitações para identificar as possibilidades de serem adaptadas neste trabalho.

Na literatura relacionada, foi possível identificar duas abordagens para o problema: mapeamento bidirecional, para casos onde ocorre a transformação entre modelos fortemente relacionados e mapeamentos mais complexos, para os casos onde as transformações envolvem artefatos mais distintos.

5.1.1 Abordagens de Mapeamento Bidirecional

Uma proposta de suporte de RTE para UML (*Unified Modelling Language*) e Java é a ferramenta *Fujaba* (NICKEL; NIERE; ZÜNDORF, 2000). Seus autores, partindo de uma limitação na utilização da UML, justificam que embora a UML permita operações na estrutura gráfica do objeto em alto nível de abstração e impulsione o programador com referências claras em nível de código, a geração de código a partir de diagramas de classe cria apenas métodos vazios. Dessa forma, propõem a ferramenta *Fujaba* para gerar código através de diagramas de estado, porque seus elementos especificam as reações do objeto ativo em eventos, mudanças de estado e transições. Estendem a capacidade da ferramenta para também gerar código a partir de diagramas de colaboração. Outra contribuição da ferramenta *Fujaba* é a atribuição de uma semântica padrão para os elementos gráficos do diagrama de colaboração, abrangendo as operações de navegação, de identificação dos participantes em colaborações e de mudança estrutural. Com o suporte à RTE a ferramenta permite analisar as partes do código da aplicação, recuperar o diagrama da UML correspondente, modificar o diagrama e gerar um novo código para o restante do aplicativo.

Ferramentas como *Fujaba* oferecem um suporte a RTE, pois permitem que o usuário manipule as visões e a implementação do modelo, sendo que todas as

ações são convertidas diretamente para implementação do modelo e depois retornadas para ambas as visões. Porém, não há um suporte para restrições (*constraints*) ou para operações de manipulação no diagrama de classe com relação às regras de modelagem (PAESSCHEN, MEUTER E D'HONDT (2005).

A dificuldade percebida com a RTE, e que muitas vezes é negligenciada é o fato de que as transformações não são totais nem injetivas, ou seja, há conceitos no modelo origem que não tem correspondência no modelo alvo e vice-versa. Também pode haver vários modelos de origem sendo mapeados para o mesmo modelo alvo. Alterações na origem devem atualizar de forma exata o alvo, sem qualquer outra interferência, para que ocorra uma conversão viável na propagação da mudança para o elemento alvo (HETTEL; LAWLEY; RAYMOND, 2008).

É notável a contribuição inicial que o desenvolvimento da ferramenta *Fujaba* proporcionou às pesquisas relacionadas à RTE, pois a mesma é frequentemente citada na literatura. Merece destaque nessa abordagem a adoção de uma semântica padrão, que permitiu à ferramenta reconhecer os elementos do diagrama e, com isso, produzir o retorno correspondente das modificações ocorridas. Da mesma forma, a definição de uma semântica padrão para os elementos da implementação de referência pode facilitar o desenvolvimento de *templates* e a localização das modificações nesses artefatos.

A ferramenta *SelfSync* (PAESSCHEN; MEUTER; D'HONDT, 2005) é um protótipo baseado em ambiente RTE, que utiliza a linguagem *Self* e baseia-se na combinação de protótipos orientados a objetos e um editor gráfico para diagramas EER (*Extended Entity-Relationship*). As entidades da visão da modelagem de dados e os objetos correspondentes da execução são semelhantes, o que resulta em um processo de sincronização contínua, altamente dinâmica entre as duas visões. É criado um link bidirecional entre objetos da análise de domínio com objetos de implementação, pela adição de uma visão extra na arquitetura MVC (*Model-View-Controller*). A visão existente de um delineador Self, ou seja, a representação gráfica padrão de objetos de implementação é estendida com uma visão da EER, de tal forma que ambas as visões representem o objeto. Ambas as visões estão conectadas e sincronizadas em nível de atributos e operações.

A abordagem relacionada à ferramenta *SelfSync* fundamenta-se na observação de que um elemento do modelo fonte e um do destino são apenas vistas

diferentes da mesma coisa, sendo uma instância de um terceiro modelo comum. Isso faz com que a sincronização seja trivial, onde independentemente da origem ou do destino, a manipulação seja imediatamente refletida no outro modelo, pois a fonte e o elemento de destino são a mesma entidade do modelo comum. No entanto, isso pressupõe que a fonte e o modelo de destino sejam isomorfos, ou seja, que cada elemento de origem tem exatamente um elemento de destino correspondente. Dessa forma, não são permitidos dois elementos de origem diferentes mapeados para o mesmo elemento alvo (HETTEL; LAWLEY; RAYMOND, 2008).

O propósito da transformação bidirecional em manter a consistência entre dois modelos desempenha um papel importante e apresenta grande potencial para o desenvolvimento de aplicações, incluindo técnicas de sincronização do modelo, engenharia de ida-e-volta, evolução de software, múltiplas vistas dos artefatos no processo de desenvolvimento de software e engenharia reversa. No entanto, a semântica bidirecional, que não é muito clara com relação ao método bidirecional específico para o domínio, e a falta de um framework para o desenvolvimento sistemático são problemas conhecidos e que impedem o uso (HIDAKA et al., 2009).

A proposta de Hidaka et al. (2009) é a de um framework para transformações bidirecionais baseado em gráfico e na linguagem UnQL (*Unstructured Query Language*). Possibilita o desenvolvimento de várias transformações no modelo pela combinação de um número fixo de transformações primitivas bidirecionais. A transformação no modelo é descrito em UnQL+ (extensão em gráfico pela consulta à linguagem UnQL) e ocorre na forma funcional e composta. A conversão ocorre em gráfico algébrico, que consiste num grupo de construtores gráficos com três composições para manipulação, sendo sequencial, condicional e recursivo estrutural. Assim, esse gráfico algébrico pode ter semânticas direcionais claras e evoluir de forma bidirecional.

Como visto, as abordagens de bidirecionais visam manter sincronizadas as diferentes visões dos artefatos que são constituídos de forma idêntica. No caso desta proposta, os *templates* e o código migrado não são artefatos que possuem a mesma construção, visto que foram identificados sete tipos de mapeamento entre eles. Assim, devido à natureza desses artefatos, a complexidade de sincronização aumenta, pois as diversas formas de mapeamento precisam ser tratadas individualmente para que a alterações no código da implementação de referência

possam se refletir corretamente nos *templates*. No processo de geração de código a sincronização já ocorre naturalmente de forma unidirecional no sentido dos *templates* para o código, porém, a sincronização no caminho inverso é a que representa o desafio, ou seja, a propagação das alterações realizadas no código para os *templates* com a finalidade de manter o sincronismo e possibilitar sua evolução.

5.1.2 Abordagens para Sincronização Compostas

Com a evolução simultânea do código-fonte e dos modelos de software ocorre a perda de sincronismo entre eles. Para preservar o código que é introduzido manualmente enquanto o modelo está evoluindo, Angyal; Lengyel e Charaf (2008) apresentam uma abordagem visando a personalização do código gerado, onde o *gap* de abstração é preenchido pelo resultado das transformações do modelo com uma mesclagem incremental. Essa abordagem fundamenta-se na obtenção de níveis de sincronização que envolve todas as mudanças do processo de ida-e-volta, através de uma proposta de sincronização incremental do código-fonte e um modelo específico de domínio (*Domain Specific Model* ou DSM) e com a utilização de transformações gráficas e mesclagem baseada em árvore em três vias para conciliar os dois artefatos. Assim, em baixo nível de abstração o modelo AST (*Abstract Syntax Tree*) representa o código que é construído a partir de elementos AST da linguagem alvo.

A abordagem de Angyal, Lengyel e Charaf (2008) é demonstrada através da taxonomia de modelos em MDA (*Model Driven Architecture*), onde um DSM corresponde ao *Platform-Independent Model* (PIM) e o modelo AST corresponde ao *Platform-Specific Model* (PSM). Há duas tarefas diferentes de ida-e-volta entre PIM-PSM e PSM-Código, onde o objetivo é conciliar o PIM com o código, sendo o PSM apenas um artefato intermediário que não pode ser editado diretamente e permanece inalterado até a sincronização. Assim, o espaço de sincronização é composto desses dois domínios sobrepostos e a união deles é o PSM. Tanto as tarefas de ida como as de volta são incrementais e não destroem os artefatos, possibilitando reconstruí-las novamente a partir do zero.

O processo de ida e volta entre o PIM e o código-fonte é dividido em três fases. Em primeiro lugar o PIM e o PSM serão conciliados, o modelo PSM ainda não foi modificado nesse ponto e as alterações no PIM devem ser propagadas para o PSM. Um modelo de rastreamento devidamente projetado pode facilitar a detecção de mudanças no PIM e essa transformação pode ser unidirecional. A próxima fase reconcilia os estados do PSM (modelo AST) e o código fonte por uma mesclagem incremental bidirecional baseada nas diferenças da árvore de três vias, que também considera o código fonte anterior sincronizado como estado inicial. A terceira fase da tarefa de ida-e-volta é o inverso da primeira transformação do modelo unidirecional. A responsabilidade dessa transformação é atualizar o PIM de forma incremental, se necessário, usando um modelo de rastreamento - *tracing* (ANGYAL; LENGYEL; CHARAF, 2008).

O elemento principal dessa abordagem é a técnica de mesclagem do código com o modelo AST. Ocorre um mapeamento de “um-para-um” entre o modelo AST e o código fonte compilado, tornando fácil a comparação e a sincronização entre eles. Após cada sincronização o arquivo fonte é armazenado com o último estado sincronizado. Essa correspondência facilita o rastreamento entre os nós encontrados e pode ser considerada a "identidade dos dados" entre o modelo AST e o código fonte e, assim, o script editado pode ser derivado das mudanças e dos nós não correspondidos. A Figura 5.1 apresenta os detalhes do processo de sincronização. As árvores A0, A1 e M1 representam os artefatos iniciais, as ASTs A0, A1 e A2, os arquivos interpretados pelo código e M1, M2 representam o modelo. Na primeira fase, antes da diferenciação, as árvores são convertidas em estruturas de memória temporária (T0 e T1), projetadas para seguir os algoritmos. As diferenças a serem analisadas são derivadas da correspondência da árvore e da edição do script gerado (ANGYAL; LENGYEL; CHARAF, 2008).

Dentre as vantagens dessa técnica de sincronização do modelo com o código destacam-se a garantia da correção sintática e a inexistência de separação do código que é gerado daquele inserido manualmente nos artefatos gerados. Grande parte da tarefa de ida-e-volta entre modelo-código pode ser resolvida pela abordagem de mesclagem no modelo-código AST geral. A técnica torna as transformações modelo-modelo simples, possibilitando realizar a validação em alto nível e verificar a consistência em modelos específicos de domínio, derivados de

códigos fontes alterados. Por outro lado, são identificadas algumas limitações relacionadas com as mudanças significativas na estrutura de código que podem causar correspondências inadequadas, como scripts desnecessários e longos, devido à compilação e subsequente geração do código fonte. Os comentários e as informações de formatação (espaços brancos) podem ser perdidos, assim, são necessários nós AST dedicados para os comentários. Como os conflitos são tratados em baixo nível de abstração é difícil encontrar as verdadeiras intenções do desenvolvedor e pequenas mudanças no código podem causar diferenças significativas na estrutura da AST, o que complica a manutenção das regras de transformação (ANGYAL; LENGYEL; CHARAF, 2008).

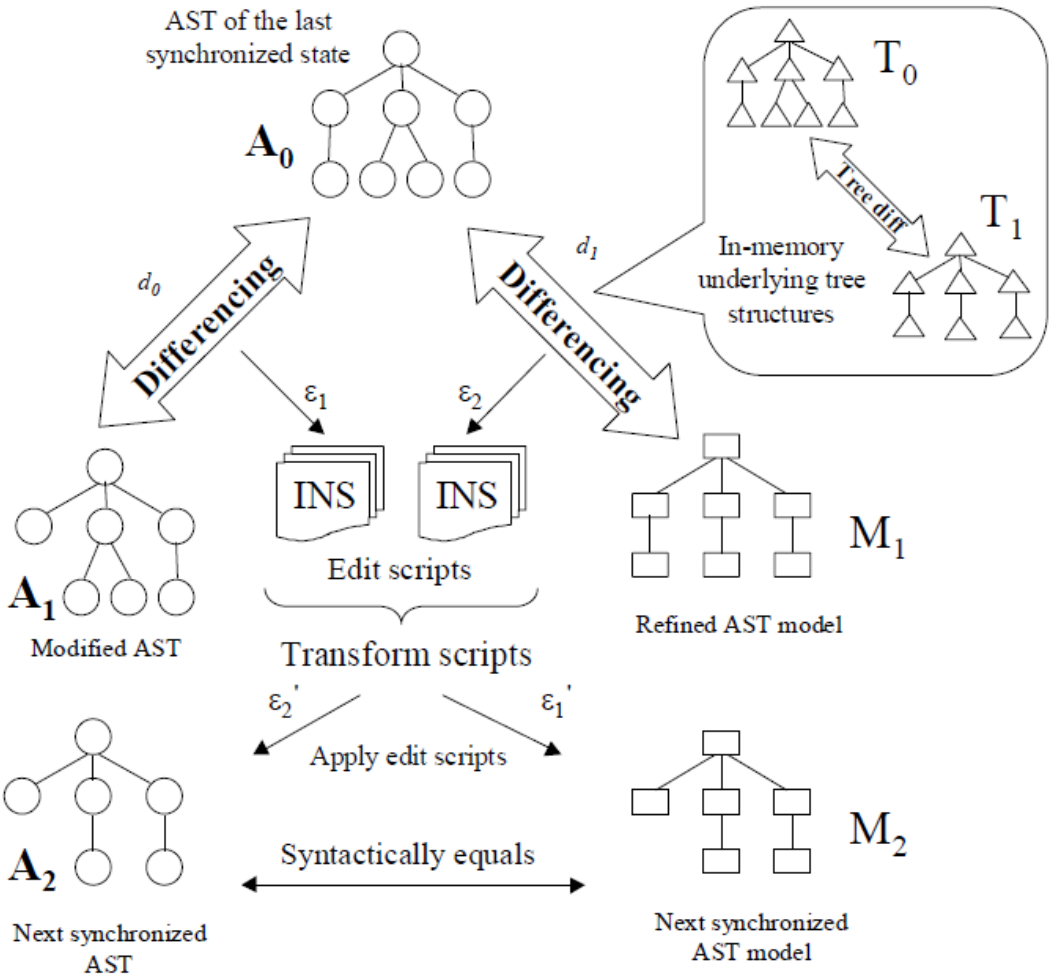

Figura 5.1 - Detalhes da sincronização em três vias (extraído de Angyal; Lengyel; Charaf (2008))

Essa abordagem utiliza o modelo AST combinado com a técnica de mesclagem incremental do código, além do método *tracing* para tratar as diferenças identificadas na correspondência do modelo com o código, que é bastante atrativa, mas apresenta um alto grau de complexidade para implementá-la no processo de migração de código, principalmente por pressupor que os artefatos têm uma relação simples de “um-para-um”, o que não ocorre entre os *templates* e a implementação de referência. Porém, considerando a avaliação dessa técnica, onde foi identificado que a ocorrência de mudanças significativas no código pode desconectá-lo ou causar correspondências inadequadas, abstraem-se resultados úteis para conduzir a uma análise de limite nesta proposta, ou seja, definir a abrangência das modificações suportadas na implementação de referência que poderão ser absorvidas pelo *template*.

Nessa mesma perspectiva, Jarzabek e Trung (2011) enfatizam que o código gerado a partir de modelos é muitas vezes modificado e essas mudanças não podem ser facilmente propagadas e retornadas para os modelos e, por isso, desenvolvedores que utilizam modelos e geradores na fase inicial do desenvolvimento de software, muitas vezes os abandonam mais tarde, durante o processo de evolução do software e pela dinâmica do reuso. Dentre os motivos da necessidade de modificação do código gerado a partir da DSL (*Domain-Specific Language*) estão a baixa estabilidade que envolve os domínios, consumidores que descobrem novos requisitos não previstos na DSL, aspectos específicos do uso de interfaces que os usuários requerem e muitos detalhes da lógica de negócio que a DSL não captura. Dessa forma, para usar geradores no processo de reuso e evolução de software é necessário modificar repetidamente as especificações DSL de uma aplicação gerando o código novamente e modificar o código gerado manualmente mantendo as especificações DSL e o código em sincronia.

Jarzabek e Trung (2011), analisando a aplicabilidade de técnicas RTE em geradores que produzem o código da aplicação através especificações abstratas escritas em linguagem específica de domínio (DSL), concluem que propagar as modificações da regeração do código é um processo tedioso e sujeito a erros: Há tentativas de RTE com técnicas de engenharia reversa para atualizar as especificações DSL a partir do código modificado, e também para injetar o código gerado a partir de especificações modificadas da DSL no código da aplicação, sem

substituir o restante da aplicação. Boas soluções de RTE só existem para DSLs que requerem traduções simples (como a geração de modelos a partir de diagramas de classe). Estas soluções não funcionam quando o *gap* de abstração entre DSL e código é mais amplo, quando o impacto das mudanças do modelo no código torna-se difícil de rastrear. Outro problema destacado na RTE é que a mesma não ajuda a compreender modificações manuais realizadas na aplicação ao longo do tempo.

A proposta de Jarzabek e Trung (2011) são geradores flexíveis que permitem mesclar códigos gerados a partir da DSL com quaisquer extensões manuais necessárias nas aplicações. A estratégia é utilizar o gerador convencional para interpretar a DSL e variar a técnica para marcar pontos de extensão que permitem combinar modificações manuais diretamente no processo de geração. As modificações manuais são armazenadas em formato executável de modo que a combinação possa ser refeita automaticamente, ou seja, em qualquer momento o código é gerado novamente a partir de especificações DSL modificadas, eliminando a necessidade de modificar o código gerado. Assim, a regeração de código a partir de especificações da DSL modificada e das modificações manuais pode ser repetida independentemente, mas ambos em sincronia.

A validação inicial da ferramenta de Jarzabek e Trung (2011) foi realizada com a utilização da *Microsoft DSL Toolkit*, para definir DSLs e geradores convencionais associados à técnica de variação XVCL (*XML-based Variant Configuration Language*) para facilitar modificações manuais. Para os autores o conceito de um gerador de arquitetura aberta com alguns detalhes internos revelados aos desenvolvedores não é novo. Em nível técnico, parece um gerador flexível com arquitetura aberta, composto pela simples integração de duas tecnologias existentes, ou seja, de um gerador convencional e uma técnica de variabilidade de XVCL. Mesmo assim, um gerador flexível supera os problemas de geração que continuam sem solução. No entanto, o gerador flexível é baseado na técnica abordada, que é nova em tecnologia de geração de código, ou seja, na separação cuidadosa de tarefas que não são afetadas por modificações manuais e podem ser realizadas por um gerador convencional e das tarefas interessadas com as modificações manuais que exigem a técnica de variabilidade. Cada arquitetura do gerador é nova e está relacionada com a complexidade e praticidade da abordagem

proposta, uma vez que revela aos desenvolvedores o mínimo necessário sobre o funcionamento do gerador interno.

Ciccozzi, Cicchetti e Sjodin (2011) consideram que as limitações de recursos em sistemas embarcados tornam as medidas das propriedades extrafuncionais mais críticas em nível de execução de código e propõem uma solução para implementação de forma automatizada, com suporte a RTE no contexto da MDE, focada na preservação das propriedades extrafuncionais em todo o processo, ou seja, desde o nível da modelagem até o código executável.

A utilização da RTE nessa abordagem visa apoiar o processo de desenvolvimento do ciclo MDCB (*Model-Driven and Component-Based*) no sentido de garantir a preservação extrafuncional no código gerado, ou seja, requisitos que não podem ser previstos estaticamente na modelagem, podendo avaliar os valores medidos em relação ao comportamento esperado que foi modelado a nível de projeto. A atividade crucial para tal abordagem é a "*back-annotation*" do código de execução e o monitoramento (*tracing*) dos resultados em nível de modelagem para uma avaliação extrafuncional completa (CICCOZZI; CICCHETTI; SJODIN, 2011).

O código de implementação é gerado a partir dos modelos de origem. Através do monitoramento dessa execução, os valores registrados são automaticamente anotados em nível de modelagem. Nessa abordagem existem três focos fundamentais, o armazenamento de *links* de *tracing* entre os modelos e o código gerado; a recuperação de informações úteis a partir dos resultados de execução do código monitorado e o retorno dos valores registrados para os modelos. Informações de *tracing* identificados por links durante a tarefa de geração de código e os resultados do monitoramento são armazenadas no modelo através da "*back-annotation*", consistindo num rastreamento que permite navegar através de todas as informações armazenadas para realizar a "*back-annotation*" no modelo com os resultados do monitoramento (CICCOZZI; CICCHETTI; SJODIN, 2011).

Da mesma forma que o mecanismo de "*back-annotation*" possibilitou a recuperação de informações extrafuncionais obtidas com o *tracing* durante a execução do código, poderá, no caso deste projeto, ser utilizado para inserir anotações no código da implementação de referência para identificar os locais onde ocorrerão as alterações do código. A partir de anotações introduzidas no código, que obviamente deverão ser exclusivas para cada tipo de mapeamento existente entre o

template e o código de referência, será possível identificar o local para se propagar as alterações e obter o sincronismo.

Diferentemente da seção 5.3.1, as abordagens da seção 5.3.2 lidam com soluções mais próximas do cenário deste trabalho. O sincronismo entre *templates* e a implementação de referência exige técnicas mais complexas do que um simples mapeamento bidirecional.

5.2 Considerações Finais

As abordagens RTE apresentadas representam uma pequena amostra do que existe na literatura com esse enfoque, portanto, elas não esgotam esse tópico, mas sinalizam sua relação com o problema da perda de sincronismo no processo de geração de código baseado em *templates* a partir da implementação de referência. Nesse sentido, podemos citar a abordagem de Ciccozzi, Cicchetti e Sjodin (2011), relacionada à técnica de anotação de código, que apoiou o desenvolvimento inicial do protótipo de migração automática de código.

É importante que se registre, com relação à pesquisa realizada sobre abordagens da RTE, que não foi encontrado na literatura nenhum estudo específico sobre o problema da migração de código. Mas, sem dúvida esse estudo foi válido à medida que contribuiu para a exploração do problema, inclusive os esforços para associar os mecanismos utilizados para resolver o problema da perda de sincronismo foram importantes para planejar a sincronia da implementação de referência com o *template.* Porém, é importante a continuidade dessa pesquisa, inclusive que seja mais intensificada para seguir com a avaliação da aplicabilidade do conjunto de técnicas RTE neste problema.

Capítulo 6

CONCLUSÃO

Considerando que a automação do processo de migração de código para o desenvolvimento e manutenção de geradores é algo pouco explorado, como já relatado não foi encontrado na literatura nenhum trabalho específico. Acredita-se que, com o desenvolvimento deste protótipo foi dado um passo inicial para diminuir essa distância, onde melhorias necessitam ser realizadas para corrigir as limitações, que devem ser mais bem compreendidas com o aprofundamento e a evolução dessa abordagem.

6.1 Contribuições

Acredita-se que este protótipo desenvolvido para a migração automática de código veio a contribuir para o desenvolvimento de software orientado a modelos, possibilitando a redução de custos no processo de geração de código, à medida que introduziu mecanismos para produzir a automatização parcial do processo de migração de código, que permitiram o retorno à sincronização do código gerado (implementação de referência) com os *templates* do gerador.

A solução obtida com o protótipo, conforme resultados obtidos no experimento para a maioria das tarefas de migração de código, embora parcialmente automatizada, permitiu atingir os objetivos deste trabalho, que foram de reduzir a sobrecarga de 20 a 25% ocasionada pela migração manual de código (VOELTER;

BETTIN, 2004; MUSZYNSKI, 2005). Portanto, considerando os ganhos de produtividade obtidos com o protótipo de migração automática de código, essa sobrecarga foi diminuída pela metade, ou seja, em torno de 10 a 12,5% do tempo de desenvolvimento da implementação de referência e, com isso, o processo de construção e evolução de geradores de código no contexto do MDE foi facilitado de forma considerável.

Devido a este tema ser ainda pouco explorado na literatura acredita-se que esses esforços despendidos para a busca de soluções, os resultados obtidos com a avaliação do protótipo e com cada mecanismo aplicado no desenvolvimento deste trabalho, podem interessar à comunidade de maneira a fortalecer o conhecimento nessa área.

As tentativas realizadas durante o desenvolvimento inicial do protótipo com a implementação das técnicas de anotações de código e comparação de arquivos, apesar de não proporcionarem o resultado esperado, também se configuram contribuições deste trabalho, pois colaboraram para uma maior exploração do problema e indicaram suas limitações em automatizar o processo de migração de código.

A redução da distância entre a tecnologia oferecida pela MDE e a sua adoção representa um desafio, pois a MDE ainda tem como principais barreiras a falta de conhecimento e a complexidade de sua utilização, devido à introdução de artefatos notoriamente mais complexos, tais como os geradores de código. Reduzindo-se esta complexidade, tem-se um maior potencial de aceitação e disseminação por parte da comunidade de desenvolvedores.

6.2 Limitações

Para situar o protótipo no contexto do processo de migração de código é importante ressaltar que a sua atuação está limitada a pré-existência de um gerador em funcionamento, no mínimo em sua fase inicial de desenvolvimento. Portanto, a automatização proporcionada pelo protótipo é parcial e o seu princípio de

funcionamento está atrelado a uma estrutura de geração de código pré-existente, inclusive que possibilite refletir os mapeamentos a partir de posições pertinentes aos artefatos já estabelecidos.

A atualização automática dos modelos de entrada do gerador a partir do código da aplicação, apesar de estar fora do escopo deste trabalho merece ser verificada com mais atenção, pois desempenha um papel fundamental nesta abordagem, principalmente porque dentro do reconhecimento dos *templates* é possível alternar a produção de aplicações, por meio da variação dos modelos de entrada. Entretanto, as alterações geralmente são realizadas no código da aplicação que mistura trechos de cada artefato. Neste projeto, produziu-se o arquivo de mapeamento com a demarcação do trecho de código do *template* que referencia o modelo, porém, só foi utilizado para controlar e restringir esse tipo de modificação.

A adaptação do *Fluorite* neste projeto para o processamento automático da migração de código precisa ser revista e, possivelmente, algumas deficiências já sinalizam para a necessidade do desenvolvimento de um mecanismo específico de captura de alterações. Inclusive, é importante relatar uma das limitações que influenciou o desempenho do protótipo com relação à existência de um intervalo de tempo entre a alteração realizada na edição do arquivo e a captura pelo *Fluorite*, pois o arquivo de *log* não registra o evento no momento em que está ocorrendo a alteração, ou seja, a atualização do arquivo de *log* sempre ocorre com um evento em atraso. Dessa forma, se uma alteração estiver dividida em mais de um evento no arquivo de *log* e o *Parser* replicar essa alteração incompleta, o *template* será danificado.

Com relação às conclusões obtidas com o estudo experimental apresentado no Capítulo 4, apesar de ter apontado ganhos de produtividade com a adoção do protótipo para determinadas tarefas de migração de código, não podemos deixar de considerar as evidencias de limitações relacionadas ao experimento, pois em conformidade com os benefícios já esperados da automação, deve-se considerar que o teste empírico reteve-se a uma pequena amostra e limitou-se ao escopo de desenvolvedores no ambiente universitário. Certamente, a introdução desse mecanismo no mundo real de desenvolvimento de geradores trará novos desafios.

6.3 Trabalhos Futuros

A exploração de técnicas para melhoria do processo de migração de código está no início e muitos desafios ainda estão por vir. Esta abordagem representa um passo inicial na automação desse processo e ainda muitas melhorias são necessárias para atingir completamente a funcionalidade esperada, que poderá ser obtida com novos estudos, no sentido de potencializar o processo de desenvolvimento e manutenção de geradores de código construídos a partir de uma implementação de referência.

Neste projeto foi desenvolvido um mecanismo responsável pelo mapeamento da associação dos artefatos do gerador, inclusive contendo as referências das informações obtidas dos modelos de entrada pelos *templates* e produzidas no código da aplicação. Esse mapeamento pode servir de base para novos estudos, no sentido de agregar à abordagem mecanismos que também introduzam no gerador a funcionalidade de atualizar esses modelos a partir de modificações capturadas no código da aplicação, visando garantir o sincronismo do gerador no sentido contrário ao da geração para modificações relacionadas diretamente com os modelos de entrada.

Para detectar as alterações no código da aplicação utilizou-se o *Fluorite*, que foi fundamental para o desenvolvimento do protótipo de migração automática de código, porém, o desenvolvimento de um mecanismo específico para refletir as alterações diretamente no gerador poderá trazer maiores benefícios e corrigir as deficiências de adaptação. Dessa forma, um mecanismo específico de captura de alterações sintonizado com o gerador traria maiores benefícios à abordagem, visto que estaria fundamentado nesse processo, fazendo com que os componentes externos que foram adaptados passassem a compor os dispositivos de geração e sincronização.

O protótipo não permite o desenvolvimento completo do gerador a partir de uma aplicação de referência, pois atua na sincronização dos artefatos em um gerador pré-existente. Automatizar a construção inicial do gerador com base na implementação de referência é um desafio ainda maior, pois essa tarefa depende

fortemente da criatividade do programador, mas poderá ser possível a introdução de mecanismos que facilitem a identificação dos componentes da aplicação dentro da dinâmica de geração, atuando como auxiliador no desenvolvimento inicial do gerador. Nesse sentido, pode também ser explorada a possibilidade de estender a automação parcial obtida pelo protótipo na manutenção do gerador para colaborar com o desenvolvimento inicial dos *templates,* ou seja, a partir de um *template* mínimo estabelecido introduzir automaticamente modificações incrementais até que o mesmo esteja completo.

O experimento identificou que o protótipo não trouxe benefícios na realização de uma das tarefas de migração de código, o que precisa ser mais bem analisado. Para estabelecer uma avaliação mais precisa da interação existente entre a técnica de migração automática de código e as tarefas de correção dos *templates* é necessário a reprodução do experimento com maior número e diversidade de tarefas, incluindo a atribuição de coeficientes de complexidade e dificuldade às mesmas, de forma a mapear os resultados da produtividade do protótipo com as especificidades dos grupos de tarefas.

As abordagens RTE apresentadas com o intuito de detectar soluções para o problema da perda de sincronismo dos artefatos do gerador de código, apesar de não ser encontrado nenhum estudo específico, foram aproveitadas para identificar os mecanismos e planejar a sincronia da implementação de referência com o *template*. Porém, é importante a continuidade dessa pesquisa, inclusive que seja mais intensificada para avaliar a aplicabilidade do conjunto de técnicas RTE neste problema.

Ainda na mesma linha, experimentos em ambiente industrial certamente serão necessários. No entanto, para isso é preciso que o protótipo evolua, principalmente em termos de usabilidade, para tornar praticável o seu uso nesse tipo de ambiente.

REFERÊNCIAS

ANGYAL, L.; LENGYEL, L.; CHARAF, H. A Synchronizing Technique for Syntactic Model-Code Round-Trip Engineering. Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, Budapest, 2008. p. 463-472.

ANTKIEWICZ, M.; CZARNECKI, K. Framework-specific modeling languages with round-trip engineering. In: NIERSTRASZ, O. et al. (Ed.). Model Driven Engineering Languages and Systems (MoDELS 2006). [S.l.]: Springer Berlin / Heidelberg, 2006. (Lecture Notes in Computer Science, v. 4199/2006). p. 692–706.

BIERHOFF, K.; LIONGOSARI, E. S.; SWAMINATHAN, K. S. Incremental development of a domain-specific language that supports multiple application styles. In: GRAY, J.; TOLVANEN, J. P.; SPRINKLE, J. (Ed.). 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06), Portland, Oregon, USA. [S.l.]: Jyväskylä University Printing House, Jyväskylä , Finland, 2006. p. 67–78, ISBN: 951-39-2631-1, ISSN: 1239-291X.

CICCOZZI, F.; CICCHETTI, A.; SJODIN, M. Towards a Round-Trip Support for Model-Driven Engineering of Embedded Systems; Conference on Software Engineering and Advanced Applications (SEAA), 37th EUROMICRO, 2011.
CLEAVELAND, J. C. Building application generators. IEEE Software, v. 7, n. 1, 1988, p. 25-33.

CLEAVELAND, J. C. Separating concerns of modeling from artifact generation using XML. In: Proceedings of the 1st OOPSLA Workshop on Domain-Specific Visual Languages -Tampa Bay, USA. [S.l.]: Jyväskylä University Printing House, Jyväskylä , Finland, 2001. p. 83–86, ISBN: 951-39-1056-3, ISSN: 0359-8470.

CZARNECKI, K.; EISENECKER, U. W. Generative Programming: Methods, Tools, and Applications. [S.l.]: Addison-Wesley, 2000.

DEURSEN, A.; VISSER E.; WARMER, J. Model-Driven Software Evolution: A Research Agenda. In: D. Tamzalit, editor, CSMR Workshop on Model-Driven Software Evolution, Amsterdam, The Netherlands, 2007. p. 41-49.

EFFTINGE, S., VOELTER, M. oAW xText: A framework for textual DSLs., Workshop: Modeling Symposium, 2006.

FEILKAS, M. How to represent models, languages and transformations? In: GRAY, J.; TOLVANEN, J.-P.; SPRINKLE, J. (Ed.). 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06), Portland, Oregon USA. [S.l.]: Jyväskylä University Printing House, Jyväskylä , Finland, 2006. p. 169-176, ISBN: 951-39-2631-1, ISSN: 1239-291X.

FOWLER, M. et al. Refactoring: Improving the Design of Existing Code. [S.l.]: Addison Wesley, 1999.

FRANCE, R.; RUMPE, B. Model-driven development of complex software: A research roadmap. In: 29th International Conference on Software Engineering 2007 - Future of Software Engineering. Minneapolis, MN, USA: IEEE Computer Society, 2007. p. 37–54.

HAMZA, H. S. On the impact of domain dynamics on product-line development. In: The 5th OOPSLA Workshop on Domain-Specific Modeling, San Diego, USA. [S.l.: s.n.], 2005.

HESSELLUND, A.; CZARNECKI, K.; WASOWSKI, A. Guided development with multiple domain-specific languages. In: ENGELS, G. et al. (Ed.). Model Driven Engineering Languages and Systems (MoDELS 2007). [S.l.]: Springer, 2007. (Lecture Notes in Computer Science, v. 4735), p. 46–60. ISBN 978-3-540-75208-0.

HETTEL, T.; LAWLEY, M. J.; RAYMOND, K. Model synchronisation: Definitions for round-trip engineering. In: VALLECILLO, A.; GRAY, J.; PIERANTONIO, A. (Ed.). Theory and Practice of Model Transformations (ICMT 2008). [S.l.]: Springer Berlin / Heidelberg, 2008. (Lecture Notes in Computer Science, v. 5063/2008), p. 31–45.

HIDAKA, S.; HU, Z.; KATO, H., NAKANO, K. A compositional approach to bidirectional model transformation. In ICSE Companion, 2009, p. 235–238,

JARZABEK, S; TRUNG, H. D. Flexible Generators for Software Reuse and Evolution. Int. Conf. On Software Eng., ICSE'2011, New Ideas and Emerging Results Track, Honolulu, USA, 2011

JET (Java Emitting Template) code generator web site, disponível em: <http://wiki.eclipse.org/M2T-JET-FAQ>

JOUAULT, F.; BÉZIVIN, J.; KURTEV, I. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: JARZABEK, S.; SCHMIDT, D. C.; VELDHUIZEN, T. L. (Ed.). Fifth International Conference on Generative Programming and Component Engineering (GPCE'06). [S.l.]: ACM, 2006. p. 249–254. ISBN 1-59593-237-2.

KORHONEN, K. A case study on reusability of a dsl in a dynamic domain. In: Proceedings of the 2nd Workshop on Domain-Specific Visual Languages, 17th ACM Conference on Object Oriented Programming, Systems. Languages and Applications (OOPSLA 2002). [S.l.: s.n.], 2002.

LUCRÉDIO, D. Uma abordagem orientada a modelos para reutilização de software. Tese de Doutorado. Universidade de São Paulo. Instituto de Ciências Matemáticas e de Computação, São Carlos, SP, 2009.

LUCRÉDIO, D. et al. The Draco approach revisited: Model-driven software reuse. In: VI WDBC -Workshop de Desenvolvimento Baseado em Componentes. Recife, PE, Brasil: [s.n.], 2006. p. 72–79.

LUCRÉDIO, D.; FORTES, R. P. de M. Mapping code generation templates do a reference implementation - towards automatic code migration. In: I Brazilian Workshop on Model-Driven Development, 2010, Salvador, BA, Brasil. [S.l.: s.n.], 2010. p. 85–92.

LUCRÉDIO, D.; JACKSON, E. K.; SCHULTE, W. Playing with fire: Harnessing the hottest technologies for ultra-large-scale systems. In: 15th Monterey Workshop - Foundations of Computer Software, Future Trends and Techniques for Development, September 24-26, 2008, Budapest University of Technology and Economics. [S.l.: s.n.], 2008.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. ACM Computing Surveys, v. 37, n. 4, dez. 2005, p. 316–344, ISSN 0360-0300.

MUSZYNSKI, M. Implementing a domain-specific modeling environment for a family of thick-client gui components. In: The 5th OOPSLA Workshop on Domain-Specific Modeling, San Diego USA. [S.l.: s.n.], 2005. p. 5–14.

NAUMENKO, D. java-diff-utils-The DiffUtils library for computing diffs, applying patches, generationg side-by-side view in Java, 2011. Disponível em: <http://code.google.com/p/java-diff-utils/>. Acesso em: 15 ago 2012

NEIGHBORS, J. M. Software Construction Using Components. Tese (Ph.D. Thesis) - University of California at Irvine, 1980.

NICKEL, U.; NIERE, J.; ZÜNDORF, A. The Fujaba Environment; Proc. 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland: ACM Press, 2000. p. 742-745.

PAESSCHEN, E. V.; MEUTER, W. D.; D'HONDT, M. Selfsync: A dynamic round-trip engineering environment. In: BRIAND, L.; WILLIAMS, C. (Ed.). Model Driven Engineering Languages and Systems (MoDELS 2005). [S.l.]: Springer Berlin / Heidelberg, 2005. (Lecture Notes in Computer Science, v. 3713/2005). p. 633–647.

PAUL, R. Designing and implementing a domain-specific language. Linux J., Specialized Systems Consultants, Inc., Seattle, WA, USA, v. 2005, n. 135, p. 7, 2005. ISSN 1075-3583.

POPMA, R. JET Tutorial Part 1 (Introduction to JET). May 2004. Eclipse Corner Article. Disponível em: <http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html>. Acesso em: 14 jun 2009.

SCHMIDT, D. C. Guest editor's introduction: Model-driven engineering. IEEE Computer, v. 39, n. 2, 2006. p. 25-31.

TOLVANEN, J.-P.; KELLY, S. Modelling languages for product families: A method engineering approach. In: Proceedings of the 1st OOPSLA Workshop on Domain-Specific Visual Languages -Tampa Bay, USA. [S.l.]: Jyväskylä University Printing

House, Jyväskylä , Finland, 2001. p. 135–140. ISBN: 951-39-1056-3, ISSN: 0359-8470.

VISSER, E. WebDSL: A case study in domain-specific language engineering. In: LÄMMEL, R.; VISSER, J.; SARAIVA, J. (Ed.). Generative and Transformational Techniques in Software Engineering (GTTSE 2007). [S.l.]: Springer, 2007. (Lecture Notes in Computer Science, v. 5235), p. 291–373. ISBN 978-3-540-88642-6.

W3C. XML Path Language (XPath) Version 1.0 -W3C Recommendation 16 November 1999. [S.l.], 1999.

VOELTER, M., BETTIN, J., Patterns for Model-Driven Software Development, Version 1.4, May 10, 2004, www.voelter.de.

WILE, D. Lessons learned from real DSL experiments. Sci. Comput. Program., Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 51, n. 3, 2004, p. 265–290, ISSN 0167-6423.

WIMMER, M. et al. Towards model transformation generation by-example. In: 40th Hawaii International Conference on System Sciences (HICSS'07). Hawaii: [s.n.], 2007. p. 285–294.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. Experimentation in Software Engineering: an introduction. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN 0-7923-8682-5.

YOON, Y.; MYERS, B. A. Capturing and analyzing low-level events from the code editor. In: PLATEAU (2011).

Apêndice A

DOCUMENTOS DO EXPERIMENTO

Durante os experimentos descritos no Capítulo 4, foram utilizados diversos documentos e formulários que são fornecidos dentro deste Capítulo de Apêndice.

Dessa forma, na Figura A.1 é mostrado o Formulário de Caracterização;

Da figura A.2 a A.4 são mostrados os Elementos do Projeto;

Os formulários para execução das tarefas de migração de código estão apresentados:

Tarefa Piloto - Manual (Figura A.5), Protótipo (Figura A.6);

Tarefa 1: Manual (Figura A.7), Protótipo (Figura A.8);

Tarefa 2: Manual (Figura A.9), Protótipo (Figura A.10);

Tarefa 3: Manual (Figura A.11), Protótipo (Figura A.12);

Tarefa 4: Manual (Figura A.13), Protótipo (Figura A.14);

Por fim, a Figura A.15 mostra o formulário de Informações dos participantes sobre o experimento.

Nome: _____

Aluno de:
( ) Graduação
( ) Mestrado
( ) Doutorado

Formulário de Caracterização

Responda todas as questões abaixo com a maior sinceridade possível.

1. *Importante:* Esses dados são confidenciais e não serão divulgados de forma alguma.

1. Qual a sua experiência com a utilização da IDE Eclipse em geral?

   1.1. ( ) nunca utilizei
   1.2. ( ) utilizei uma vez
   1.3. ( ) poucas vezes
   1.4. ( ) uso com frequência

2. Qual é a sua experiência com a utilização de ferramentas de modelagem (UML, Ecore. EMF, GMF)?

   2.1. ( ) nunca utilizei
   2.2. ( ) utilizei uma vez
   2.3. ( ) poucas vezes
   2.4. ( ) uso com frequência

3. Qual é a sua experiência com o desenvolvimento e/ou manutenção de aplicações Web?
   3.1. ( ) nunca executei
   3.2. ( ) executei uma vez
   3.3. ( ) executei poucas vezes
   3.4. ( ) executo com frequência

4. Qual é sua experiência com o desenvolvimento e/ou manutenção de geradores de código baseados em *template*?

   4.1. ( ) nunca executei
   4.2. ( ) executei uma vez
   4.3. ( ) executei poucas vezes
   4.4. ( ) executo com frequência

Figura A.6.1 - Formulário de Caracterização

Elementos do projeto

***Experimento**:* Avaliação do protótipo para Migração Automática de Código

Modelos de Entrada: "ProjetoTesteGerador"

"src/entradaGerador"

Modelo1 - Diagrama de dados e relacionamentos (persistência/classes)

"paginaPessoal.webdomainadministrationsubdomain_diagram"

Modelo 2 - Modelo de navegação (páginas e o conteúdo)

"paginaPessoal.webnav"

Modelo 3 – Features (conteúdo de usuário/moderação)

"My.webdomainfeatures"

Implementação de referência: "ProjetoTeste"

- É alvo da geração dos modelos. A ligação é definida no Modelo1, através do atributo "Project".
- Servidor tomcat / Banco de dados local (apache derby)
- Parte gerada:
    - Código-fonte (Java) para persistência e controle (Modelo1: atributos "Src folder" e "Package");
    - Scripts de geração de esquema de banco de dados (Modelo1, pasta "scripts");
    - Páginas administrativas (Modelo1: inserção, remoção, edição e listagem dos dados: pasta web e subpasta: atributos "Web folder" e "Path");
    - Páginas de navegação (informações e conteúdo acessados pelo usuário: Modelo2 e atributo "Web folder": Modelo 2: subpasta "gen");
    - O conteúdo de usuário e moderação (Features: afeta páginas administrativas e de navegação: Modelo3).

Página 1 de 3

Figura A.6.2 - Elementos do Projeto - Pagina 1

- Pastas geradas:
  - ProjetoTeste/src/generated.* (ref. Modelo 1 + Modelo 3)
  - ProjetoTeste/scripts/create.sql (ref. Modelo 1 + Modelo 3)
  - ProjetoTeste/WebContent/admin/* (ref. Modelo 1 + Modelo 3)
  - ProjetoTeste/WebContent/gen/* (ref. Modelo 2 + Modelo 3)

- Pastas fixas:
  - ProjetoTeste/src/core.*
  - ProjetoTeste/scripts/* (exceto create.sql)
  - ProjetoTeste/WebContent/css
  - ProjetoTeste/WebContent/js
  - ProjetoTeste/WebContent/META-INF
  - ProjetoTeste/WebContent/popups
  - ProjetoTeste/WebContent/WEB-INF
  - ProjetoTeste/WebContent/admin.jsp
  - ProjetoTeste/WebContent/index.html

Gerador JET (Java Emmiter Templates) - EMF/Eclipse

"generatorForWebDomain"

- "Run Configurations...": Permite especificar o modelo de entrada.
- "main.jet": Inicia o gerador jet e carrega os demais modelos
- Run "JET Transformation": Executa o gerador
- Os detalhes da geração são exibidos no terminal:
  - Primeira geração (arquivos não existem): exibem no terminal todos os arquivos gerados .
  - Próximas: exibe somente os arquivos modificados. Se não houver nenhuma alteração é exibido "Successful Execution".
- É recomendado efetuar "Refresh" ou "Clean" no projeto do gerador ("generatorForWebDomain") antes de executar "JET Transformation"

mapping.txt:

- Arquivo de mapeamento: contém as posições dos arquivos de templates associados aos modelos de entrada e a código gerado correspondente. É refeito automaticamente toda vez que é executado o gerador Jet.

logging: Eventos do Fluorite.

- Captura os eventos do editor de código do IDE Eclipse (alteração de dados), sendo gravados em arquivo de log formato xml:

"workspace/metadata/plugins/edu.cmu.scs.fluorite/Logs/log*.xml"

Página 2 de 3

Figura A.6.3 - Elementos do Projeto - Página 2

Preparar:

- Seta o início do processo capturando esse instante do arquivo de log, através da variável "timestamp".
- *Importante:* Executá-lo sempre após "Run JET Transformation" e antes de iniciar as alterações no editor do eclipse.

Principal:

- É responsável pela replicação automática das mudanças efetuadas na implementação de referência para os templates:
- *Importante:* Deve ser executado após as modificações efetuadas no código serem salvas no editor do eclipse e, também, para que seja garantida a gravação do último evento no arquivo de log do Fluorite é necessário fechar e abrir o arquivo modificado.
- Captura o evento (logChange.xml) e busca sua associação (mapping.txt). Processa essas informações e executa a atualização do template.
- A execução do Principal também reinicia o timestamp do arquivo de log (logchange.xml) para que a alteração processada não seja capturada novamente

- **Caixas de diálogo:** O usuário pode decidir pela execução ou cancelamento da execução, sendo exibida:
  - Quando ocorrem alterações associadas a mapeamento de repetição (iterate), cuja geração propagará para outros locais associados.
  - Quando a alteração envolve mais de um mapeamento na sequência (overtake).

- **Informações da execução exibidas no terminal:**
  - Todos os eventos de alteração de dados capturados pelo Fluorite que forem processados pelo parser são exibidos no terminal, inclusive as alterações abortadas pelo parser e/ou canceladas pelo usuário.
  - O parser abortará e exibirá informação no terminal sobre qualquer modificação no código da aplicação oriunda dos modelos de entrada, para garantir a integridade do template.

Figura A.6.4 - Elementos do Projeto - Página 3

Roteiro do Experimento

Tarefa (Piloto) - Sincronização Convencional (Manual)

1. Inicialização do ambiente de trabalho:

1.1. Start IDE Eclipse;
1.2. Run JET Transformation;
1.3. Run index.html do projeto "ProjetoTeste" + Servidor Tomcat;
1.4. Verificar se a aplicação web está funcionando corretamente.

2. Leitura das tarefas a serem realizadas conforme instrução abaixo:

- Corrigir o erro contido na instrução sql (Select) da classe de persistência do BD Derby para visualizar todas as listas que não estão aparecendo.

  *Arquivos da aplicação:*ProjetoTeste/src/generated/daos/derby/Derby*.java

3. : Registrar abaixo o início das tarefas (copiar do sistema)

Início: _____ : _____(hh:mm)

4. Execução das tarefas:

4.1.1. Localizar a posição no arquivo de código da aplicação Web para introduzir a modificação;
4.1.2. Encontrar o template correspondente que gerará a modificação;
4.1.3. Aplicar e salvar a alteração no template;
4.1.4. Refresh projeto "generateForWeBDomain";
4.1.5. Run JET Transformation;
4.1.6. Run index.html;
4.1.7. Verificar se a modificação foi realizada corretamente

5. Registrar abaixo o término das tarefas (copiar do sistema)

Término: _____ : _____(hh:mm)

Figura A.6.5 - Tarefa Piloto - Manual

Roteiro do Experimento

Tarefa (Piloto)- Sincronização Automática (Protótipo)

1. Inicialização do ambiente de trabalho:

1.1. Start IDE Eclipse;
1.2. Run JET Transformation;
1.3. Run index.html do projeto "ProjetoTeste" + Servidor Tomcat;
1.4. Verificar se a aplicação web está funcionando corretamente.

2. Leitura das tarefas a serem realizadas conforme instrução abaixo:

3. Corrigir o erro contido na instrução sql (Select) da classe de persistência do BD Derby para visualizar todas as listas que não estão aparecendo.

3.1.1. *Arquivos da aplicação:*ProjetoTeste/src/generated/daos/derby/Derby*.java

4. : Registrar abaixo o início das tarefas (copiar do sistema)

Início: _____ : _____(hh:mm)

5. Execução das tarefas:

5.1.1. Run “Preparar”;
5.1.2. Localizar a posição no arquivo de código da aplicação Web e introduzir a modificação
5.1.3. Salvar, Fechar e Abrir novamente o arquivo que foi modificado;
5.1.4. Run “Principal”;
5.1.5. Refresh projeto "generateForWeBDomain";
5.1.6. Run “JET Transformation”;
5.1.7. Run “index.html”;
5.1.8. Verificar se a modificação foi realizada corretamente

6. Registrar abaixo o término das tarefas (copiar do sistema)

Término: _____ : _____(hh:mm)

Figura A.6.6 - Tarefa Piloto - Protótipo

Roteiro do Experimento

Tarefa 1 - Sincronização Convencional (Manual)

1. Inicialização do ambiente de trabalho:

1.1. Start IDE Eclipse;
1.2. Run JET Transformation;
1.3. Run index.html do projeto "ProjetoTeste" + Servidor Tomcat;
1.4. Verificar se a aplicação web está funcionando corretamente.

2. Leitura das tarefas a serem realizadas conforme instrução abaixo:

- Modificar o formulário de moderação na Área do Usuário, tamanhos campos Nome, Email e Assunto para size=29 e na área Administrativa substituir o texto contido no título para "Moderação de Comentários" e em "userComentTitle>" para o texto "Comentários de Usuários"

Arquivos da aplicação:
ProjetoTeste/WebContent/gen/uploadUserCommentForm.jsp
ProjetoTeste/WebContent/admin/moderation.jsp

3. : Registrar abaixo o início das tarefas (copiar do sistema)

Início: _____ : _____(hh:mm)

4. Execução das tarefas:

4.1.1. Localizar a posição no arquivo de código da aplicação Web para introduzir a modificação;
4.1.2. Encontrar o template correspondente que gerará a modificação;
4.1.3. Aplicar e salvar a alteração no template;
4.1.4. Refresh projeto "generateForWeBDomain";
4.1.5. Run JET Transformation;
4.1.6. Run index.html;
4.1.7. Verificar se a modificação foi realizada corretamente

5. Registrar abaixo o término das tarefas (copiar do sistema)

Término: _____ : _____(hh:mm)

Figura A.6.7 - Tarefa 1 - Manual

Roteiro do Experimento

Tarefa 1 - Sincronização Automática (Protótipo)

1. Inicialização do ambiente de trabalho:

1.1. Start IDE Eclipse;
1.2. Run JET Transformation;
1.3. Run index.html do projeto "ProjetoTeste" + Servidor Tomcat;
1.4. Verificar se a aplicação web está funcionando corretamente.

2. Leitura das tarefas a serem realizadas conforme instrução abaixo:

- Modificar o formulário de moderação na Área do Usuário, tamanhos campos Nome, Email e Assunto para size=29 e na área Administrativa substituir o texto contido no título para "Moderação de Comentários" e em "userComentTitle>" para o texto "Comentários de Usuários"

Arquivos da aplicação:
ProjetoTeste/WebContent/gen/uploadUserCommentForm.jsp
ProjetoTeste/WebContent/admin/moderation.jsp

3. : Registrar abaixo o início das tarefas (copiar do sistema)

Início: _____ : _____(hh:mm)

4. Execução das tarefas:

4.1.1. Run "Preparar";
4.1.2. Localizar a posição no arquivo de código da aplicação Web e introduzir a modificação
4.1.3. Salvar, Fechar e Abrir novamente o arquivo que foi modificado;
4.1.4. Run "Principal";
4.1.5. Refresh projeto "generateForWeBDomain";
4.1.6. Run "JET Transformation";
4.1.7. Run "index.html";
4.1.8. Verificar se a modificação foi realizada corretamente

5. Registrar abaixo o término das tarefas (copiar do sistema)

Término: _____ : _____(hh:mm)

Figura A.6.8 - Tarefa 1 - Protótipo

Roteiro do Experimento

Tarefa 2 - Sincronização Convencional (Manual)

1. Inicialização do ambiente de trabalho:

1.1. Start IDE Eclipse;
1.2. Run JET Transformation;
1.3. Run index.html do projeto "ProjetoTeste" + Servidor Tomcat;
1.4. Verificar se a aplicação web está funcionando corretamente.

2. Leitura das tarefas a serem realizadas conforme instrução abaixo:

- Modificar o conteúdo de todos os links “Cadastrar novo “ dos formulários de cadastro da área administrativa (Noticias, Área de Pesquisa, Projeto de Pesquisa, Suporte, Link, Publicação e Contato), contido no item “Action Mensages", substituindo para o texto "Adicionar.....".

Arquivos da aplicação:
ProjetoTeste/src/generated/resources/messages.properties

3. : Registrar abaixo o início das tarefas (copiar do sistema)

Início: _____ : _____(hh:mm)

4. Execução das tarefas:

4.1.1. Localizar a posição no arquivo de código da aplicação Web para introduzir a modificação;
4.1.2. Encontrar o template correspondente que gerará a modificação;
4.1.3. Aplicar e salvar a alteração no template;
4.1.4. Refresh projeto "generateForWeBDomain";
4.1.5. Run JET Transformation;
4.1.6. Run index.html;
4.1.7. Verificar se a modificação foi realizada corretamente

5. Registrar abaixo o término das tarefas (copiar do sistema)

Término: _____ : _____(hh:mm)

Figura A.6.9 - Tarefa 2 - Manual

Roteiro do Experimento

Tarefa 2 - Sincronização Automática (Protótipo)

1. Inicialização do ambiente de trabalho:

1.1. Start IDE Eclipse;
1.2. Run JET Transformation;
1.3. Run index.html do projeto "ProjetoTeste" + Servidor Tomcat;
1.4. Verificar se a aplicação web está funcionando corretamente.

2. Leitura das tarefas a serem realizadas conforme instrução abaixo:

- Modificar o conteúdo de todos os links “Cadastrar novo “ dos formulários de cadastro da área administrativa (Noticias, Área de Pesquisa, Projeto de Pesquisa, Suporte, Link, Publicação e Contato), contido no item “Action Mensages", substituindo para o texto "Adicionar.....".

  Arquivo da aplicação:
  ProjetoTeste/src/generated/resources/messages.properties

3. : Registrar abaixo o início das tarefas (copiar do sistema)

Início: _____ : _____(hh:mm)

4. Execução das tarefas:

4.1.1. Run “Preparar”;
4.1.2. Localizar a posição no arquivo de código da aplicação Web e introduzir a modificação
4.1.3. Salvar, Fechar e Abrir novamente o arquivo que foi modificado;
4.1.4. Run “Principal”;
4.1.5. Refresh projeto "generateForWeBDomain";
4.1.6. Run “JET Transformation”;
4.1.7. Run “index.html”;
4.1.8. Verificar se a modificação foi realizada corretamente

5. Registrar abaixo o término das tarefas (copiar do sistema)

Término: _____ : _____(hh:mm)

Figura A.6.10 - Tarefa 2 - Protótipo

Roteiro do Experimento

Tarefa 3 - Sincronização Convencional (Manual)

1. Inicialização do ambiente de trabalho:

1.1. Start IDE Eclipse;
1.2. Run JET Transformation;
1.3. Run index.html do projeto "ProjetoTeste" + Servidor Tomcat;
1.4. Verificar se a aplicação web está funcionando corretamente.

2. Leitura das tarefas a serem realizadas conforme instrução abaixo:

- Inserir uma imagem (logotipo da ufscar) "<img alt="" src="ufscar.jpeg">" no topo dos formulários de listas/edição (Area Administrativa e do Usuário), entre </head> e <body>.

Arquivos da aplicação:
Area Administrativa: WebContent/admin/*, exceto moderation.jsp
Usuário: WebContent/gen/*, exceto UploadUserCommentForm.jsp

3. : Registrar abaixo o início das tarefas (copiar do sistema)

Início: _____ : _____(hh:mm)

4. Execução das tarefas:

4.1.1. Localizar a posição no arquivo de código da aplicação Web para introduzir a modificação;
4.1.2. Encontrar o template correspondente que gerará a modificação;
4.1.3. Aplicar e salvar a alteração no template;
4.1.4. Refresh projeto "generateForWeBDomain";
4.1.5. Run JET Transformation;
4.1.6. Run index.html;
4.1.7. Verificar se a modificação foi realizada corretamente

5. Registrar abaixo o término das tarefas (copiar do sistema)

Término: _____ : _____(hh:mm)

Figura A.6.11 - Tarefa 3 - Manual

Roteiro do Experimento

Tarefa 3 - Sincronização Automática (Protótipo)

1. Inicialização do ambiente de trabalho:

1.1. Start IDE Eclipse;
1.2. Run JET Transformation;
1.3. Run index.html do projeto "ProjetoTeste" + Servidor Tomcat;
1.4. Verificar se a aplicação web está funcionando corretamente.

2. Leitura das tarefas a serem realizadas conforme instrução abaixo:

- Inserir uma imagem (logotipo da ufscar) "<img alt="" src="ufscar.jpeg">" no topo dos formulários de listas/edição (Area Administrativa e do Usuário), entre </head> e <body>.

  Arquivos da aplicação:
  Area Administrativa: WebContent/admin/*, exceto moderation.jsp
  Usuário: WebContent/gen/*, exceto UploadUserCommentForm.jsp

3. : Registrar abaixo o início das tarefas (copiar do sistema)

Início: _____ : _____(hh:mm)

4. Execução das tarefas:

4.1.1. Run “Preparar”;
4.1.2. Localizar a posição no arquivo de código da aplicação Web e introduzir a modificação
4.1.3. Salvar, Fechar e Abrir novamente o arquivo que foi modificado;
4.1.4. Run “Principal”;
4.1.5. Refresh projeto "generateForWeBDomain";
4.1.6. Run “JET Transformation”;
4.1.7. Run “index.html”;
4.1.8. Verificar se a modificação foi realizada corretamente

5. Registrar abaixo o término das tarefas (copiar do sistema)

Término: _____ : _____(hh:mm)

Figura A.6.12 - Tarefa 3 - Protótipo

Roteiro do Experimento

Tarefa 4 - Sincronização Convencional (Manual)

1. Inicialização do ambiente de trabalho:

1.1. Start IDE Eclipse;
1.2. Run JET Transformation;
1.3. Run index.html do projeto "ProjetoTeste" + Servidor Tomcat;
1.4. Verificar se a aplicação web está funcionando corretamente.

2. Leitura das tarefas a serem realizadas conforme instrução abaixo:

- Corrigir o tamanho do campo "Area de Pesquisa" do formulário de Edição ou de Novo Cadastro, contido em Publicação/Área Administrativa. Alterar para size = 30

Arquivos da aplicação:
ProjetoTeste/WebContent/admin/EditProjetoDePesquisa.jsp
ProjetoTeste/WebContent/EditPublicacao.jsp

3. : Registrar abaixo o início das tarefas (copiar do sistema)

Início: _____ : _____(hh:mm)

4. Execução das tarefas:

4.1.1. Localizar a posição no arquivo de código da aplicação Web para introduzir a modificação;
4.1.2. Encontrar o template correspondente que gerará a modificação;
4.1.3. Aplicar e salvar a alteração no template;
4.1.4. Refresh projeto "generateForWeBDomain";
4.1.5. Run JET Transformation;
4.1.6. Run index.html;
4.1.7. Verificar se a modificação foi realizada corretamente

5. Registrar abaixo o término das tarefas (copiar do sistema)

Término: _____ : _____(hh:mm)

Figura A.6.13 - Tarefa 4 - Manual

Roteiro do Experimento

Tarefa 4 - Sincronização Automática (Protótipo)

1. Inicialização do ambiente de trabalho:

1.1. Start IDE Eclipse;
1.2. Run JET Transformation;
1.3. Run index.html do projeto "ProjetoTeste" + Servidor Tomcat;
1.4. Verificar se a aplicação web está funcionando corretamente.

2. Leitura das tarefas a serem realizadas conforme instrução abaixo:

- Corrigir o tamanho do campo "Area de Pesquisa" do formulário de Edição ou de Novo Cadastro, contido em Publicação/Área Administrativa. Alterar para size = 30

Arquivos da aplicação:
ProjetoTeste/WebContent/admin/EditProjetoDePesquisa.jsp
ProjetoTeste/WebContent//EditPublicacao.jsp

3. : Registrar abaixo o início das tarefas (copiar do sistema)

Início: _____ : _____(hh:mm)

4. Execução das tarefas:

4.1.1. Run “Preparar”;
4.1.2. Localizar a posição no arquivo de código da aplicação Web e introduzir a modificação
4.1.3. Salvar, Fechar e Abrir novamente o arquivo que foi modificado;
4.1.4. Run “Principal”;
4.1.5. Refresh projeto "generateForWeBDomain";
4.1.6. Run “JET Transformation”;
4.1.7. Run “index.html”;
4.1.8. Verificar se a modificação foi realizada corretamente

5. Registrar abaixo o término das tarefas (copiar do sistema)

Término: _____ : _____(hh:mm)

Figura A.6.14 - Tarefa 4 - Protótipo

Informações dos participantes sobre o experimento:

Experimento: Avaliação do protótipo para migração automática de código

1. A utilização do protótipo facilitou/ajudou na tarefa de sincronização da implementação de referência (código da aplicação) com o template de geração?

2. Quais foram as dificuldades e/ou problemas encontrados durante a utilização do Protótipo?

3. Informar se ocorreram erros e se foram detectados durante a utilização do Protótipo?

4. As informações exibidas no terminal e na caixa de diálogo foram úteis?

5. Informe caso tenha sugestões para a melhoria desse protótipo?

Figura A.6.15 - Informações dos participantes sobre o experimento