Luciano Carlos Perini

Um Editor Simultâneo para Facilitar a Evolução de *Templates* de Geração de Códigos

Brasil

2015, Janeiro

Luciano Carlos Perini

Um Editor Simultâneo para Facilitar a Evolução de *Templates* de Geração de Códigos

Dissertação de Mestrado.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação

Orientador: Prof. Dr. Daniel Lucrédio

Brasil

2015, Janeiro

Luciano Carlos Perini
Um Editor Simultâneo para Facilitar a Evolução de *Templates* de Geração de Códigos/ Luciano Carlos Perini. – Brasil, 2015, Janeiro-
126 p. : il. (algumas color.); 30 cm.

Orientador: Prof. Dr. Daniel Lucrédio

Dissertação (Mestrado) – Universidade Federal de São Carlos – UFSCar
Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação, 2015, Janeiro.

1. Engenharia de *Software*. 2. Engenharia de *Software* Dirigida por Modelos. I. Editor Simultâneo II. Geração de códigos baseada em *templates*. III. Edição de textos sincronizada.

CDD 005.1 (20ª)

Este trabalho é dedicado ao meu filho Benjamin.

Agradecimentos

Eu agradeço,

ao meu orientador, Prof. Dr. Daniel, pela ajuda e a oportunidade de poder desenvolver este trabalho;

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

à Prof.ª Dr.ª Rosângela, pelas sugestões e correções apontadas no exame de qualificação, importantes para a realização deste trabalho;

ao Prof. Dr. Prado, pelas sugestões e correções apontadas no exame de qualificação, que foram muito importantes para a realização deste trabalho;

ao Bruno, Possatto e colegas membros do grupo de pesquisa pelo apoio;

aos demais professores;

aos que contribuíram com o experimento, pela sua participação e também pelo apoio;

a todos que, de uma forma ou de outra, me fizeram ser quem eu sou hoje.

E por último, mas não menos importante, agradeço especialmente à minha mãe Izabel e ao meu pai Rubens.

"Eu chamo de bravo aquele que ultrapassou
seus desejos, e não aquele que venceu seus inimigos;
pois a mais dura das vitórias é a vitória sobre si mesmo".
(Aristóteles, 384a.C - 322a.C)

Resumo

Na Engenharia de *Software* Dirigida por Modelos as transformações tem um papel fundamental, assim como os modelos, na geração de artefatos concretos e dependentes de plataforma a partir de outros artefatos mais abstratos e independentes de plataforma. Com a evolução do *software* e a necessidade de mudanças nos artefatos do processo de geração de código, verifica-se a perda do sincronismo entre esses artefatos, o que demanda um esforço adicional para mantê-los sincronizados. Desse modo, com o intuito de identificar e realizar essas modificações de um modo mais dinâmico, um editor simultâneo lado a lado foi implementado buscando a diminuição do tempo despendido no processo de criação e manutenção dos artefatos de geração de código. Este trabalho, portanto, concentrou-se no desenvolvimento de um mecanismo que possibilita a edição, visualização e a propagação automática das alterações efetuadas em *templates* de geração de códigos. Com isso, a manutenção dos artefatos envolvidos será facilitada, permitindo então ganhos de produtividade no processo e melhora na evolução de *software*. O protótipo desenvolvido foi submetido a um estudo empírico para aferir sua utilidade dentro do processo. Foi obervada e constatada a redução no esforço adicional necessário para manter os artefatos sincronizados.

Palavras-chaves: Engenharia de *Software* Dirigida por Modelos. *MDD. MDE.* Geração de códigos baseada em *Templates.* Editor de textos estruturado. Sincronização. Edição de textos sincronizada.

Abstract

In Model-Driven Software Engineering, transformations play a fundamental role, as do the models, in the generation of concrete, platform-dependent assets, from platform-independent, abstract assets. With software evolution and the need to perform changes in code generation assets, there is a loss of synchronism between these assets, what requires extra effort to keep them synchronized. In this way, in an attempt to help in the identification and realization of these modifications in a dynamic way, we developed a side by side simultaneos editor. This editor seek to reduce the time spent in the process of creating and maintaining code generation templates. In this sense, this work focused on the development of a mechanism that allows the editing, visualization and automatic propagation of changes between templates and generated code, and vice-versa. As a result, the maintenance of the involved assets be facilitated, leading to productivity gains and better software evolution. The developed prototype was subjected to an empirical to assess its usefulness in the process. We observed and found a reduction in the additional effort required to keep these assets synchronized.

Key-words: Model Driven Software Engineering. MDD. MDE. Template based code generation. Structured text editors. Synchronization. Synchronized text editors.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ADT	Abstract Data Type
AMMA	Atlas Model Management Architecture
ASP	Answer Set Programming
AST	Abstract Syntax Tree
ATL	Atlas Transformation Language
DSL	Domain Specific Language
DTD	Data Type Definition
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework
EMOF	Extended Meta Object Facility
EOF	End of File
EOL	End Of Line
JET	Java Emitter Templates
JTL	Janus Transformation Language
M2C	Model to Code
M2E	Model to Execution
M2M	Model to Model
M2P	Model to Platform
M2T	Model to Text
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MDSD	Model-Driven Software Development

MDSE	Model-Driven Software Engineering
PREP	Propagate Replay Evaluate Pick
QVT	Query/View/Transformation
RTE	Round-Trip Engineering
TGG	Triple Graph Grammar
UML	Unified Modeling Language
UnCal	Unstructured CALculus
UnQL+	Unstructured Query Language +
WYSIWYG	What You See Is What You Get
XML	eXtensible Markup Language

Lista de símbolos

$\approx$	símbolo de igualdade aproximada (Matemática).

Sumário

1 Introdução

1.1 Contextualização

O processo de desenvolvimento de *software* vem passando por intensas transformações nos últimos anos. Novos paradigmas de desenvolvimento vêm surgindo no intuito de aperfeiçoar o processo. Um desses paradigmas é o *MDE* (*Model-Driven Engineering*), que introduz modelos rigorosos durante todo o processo de desenvolvimento oferecendo, dentre outras vantagens, a geração automática de código a partir desses modelos.

A geração automática de código é possível por meio de transformações de dados aplicadas aos modelos. Essas transformações, que encapsulam conhecimentos tanto do domínio da aplicação como da experiência técnica do especialista em desenvolvimento de sistemas, utilizam-se de mapeamentos de dados previamente construídos que descrevem e/ou relacionam essas estruturas com outros modelos geralmente menos abstratos (FRANCE; RUMPE, 2007). Assim sendo, partindo de modelos com estruturas abstratas e independentes de plataforma e utilizando-se de transformações de modelos com conhecimento encapsulado, podemos chegar ao código concreto da aplicação destino .

As vantagens aparecem na reutilização do Gerador de Códigos, devido à sua estrutura de artefatos de transformação que são específicos para aquele tipo de aplicação, mas que são configuráveis para outros domínios, *e.g.*, o Gerador de Códigos utilizado no estudo experimental deste trabalho pode gerar diversas aplicações, desde o controle e publicação de notícias para um sítio *Web* acadêmico até a divulgação de produtos para um sítio *Web* de comércio eletrônico.

Existem várias abordagens na construção de Geradores de Códigos, mas o uso de *templates* é a abordagem preferida para geradores mais robustos (CLEAVELAND, 1988; FEILKAS, 2006). *Templates* são artefatos instrumentados com construções de seleção e expansão de código e utilizados na saída de um Gerador de Códigos para imprimir código fonte dentro de artefatos alvo (CZARNECKI; EISENECKER, 2000).

Na Figura 1 pode-se vizualizar a representação esquemática do processo de geração de códigos por meio de um Gerador de Códigos. Nesta figura, o gerador utiliza um modelo de dados como entrada (1); faz o uso de transformações de dados (2); e dá saída a um ou mais arquivos de código gerado (3). Neste trabalho o Gerador de Códigos faz o uso de *Templates* de geração de códigos para dar forma ao código gerado segundo a plataforma destino escolhida.

Apesar da técnica de *Templates* ser vantajosa (vide SubSeção 2.1.2), a construção destes não é uma tarefa fácil, pois um *template* mistura código para geração com código

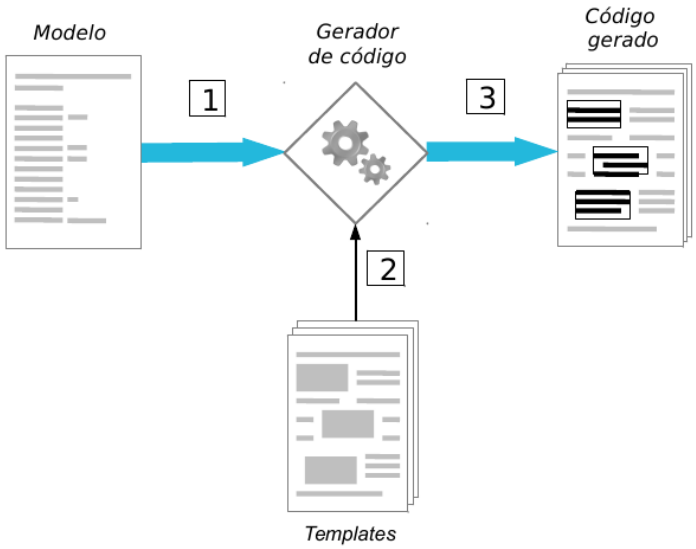

Figura 1: Representação de um Gerador de Códigos que faz o uso de *Templates*.

gerado, o que dificulta o processo de edição e depuração deste tipo de artefato.

Uma forma de mitigar esses problemas é manter, como referência, uma implementação sem artefatos da *MDE* (MUSZYNSKI, 2005). A ideia é que o desenvolvedor possa trabalhar nessa versão primeiro, realizar testes e depuração normalmente, sem se preocupar com detalhes de transformações e geração de códigos. Com isso o processo de evolução de *software* pode acontecer como no desenvolvimento convencional, não dirigido por modelos. Uma vez que essa versão, que também pode ser chamada de Implementação de Referência, esteja testada e validada, seu código é então migrado para os *templates* dentro do Gerador de Códigos, em um processo conhecido como ”Migração de Código”. Assim as mudanças passam então a constar também nos artefatos de geração de códigos.

Por outro lado, o problema dessa abordagem é e o custo da migração que fica em torno de 20 a 25% do tempo de desenvolvimento da Implementação de Referência. Além disso, causa a duplicação de código entre a Implementação de Referência e os *templates* de geração de códigos, o que requer cautela para manutenção da consistência entre esses dois artefatos (MUSZYNSKI, 2005).

Uma solução seria automatizar o processo de migração de código, o que não é uma tarefa trivial (MUSZYNSKI, 2005). Mas mesmo que essa automação seja parcial, as reduções de custo podem ser compensadoras. É nesta linha que trabalhou Possatto (2014). Ele propõe um mecanismo que faz a migração automática de mudanças realizadas em uma Implementacao de Referência para os respectivos *templates* de geração de códigos. No entanto, o pesquisador não oferece meios dinâmicos para realizar as modificações necessárias durante a evolução. Sua solução replica alterações feitas no código diretamente para os *templates* em lote de uma só vez.

Para que o processo de evolução de *software* na *MDE*, com base em migração de

código, tenha maior controle, é necessário um ambiente especializado nas tarefas de edição do código gerado, de *Templates* e da geração de códigos. Isso porque, sem um controle mais fino sobre onde as mudanças são realizadas, corre-se o risco de se realizar propagação não desejada de mudanças para os *templates*. Além disso, o desenvolvedor não consegue ver claramente os locais das mudanças sem o auxílio de ferramentas de análise de texto, uma vez que todas as modificações são propagadas de uma única vez.

1.2 Cenário

Sistemas de *software* sofrem com o impacto do tempo se não se adaptarem às necessidades e ao ambiente. Daí a necessidade da evolução do *software* por meio de reparos, adaptações e melhorias (MENS et al., 2005; NEIGHBORS, 1980). Sendo assim, evoluir é uma tarefa essencial, e estas mudanças em *software* podem vir a abranger quaisquer artefatos que constituem um sistema de *software*, tais como o código-fonte, documentação e outros artefatos de apoio ao desenvolvimento. Também faz-se necessário um cuidado com o sincronismo entre esses elementos, para garantir a consistência do *software*. Por exemplo, modificações feitas no código devem ser apropriadamente refletidas na documentação, caso contrário podem haver problemas futuros no entendimento da lógica de negócio, ou da implementação.

A *MDE* não apresenta uma exceção à regra, sendo necessário uma preocupação com a evolução de *software* durante todo o ciclo de vida desse *software* considerando todos os artefatos de *software* adicionais elaborados segundo essa abordagem (DEURSEN; VISSER; WARMER, 2007). Isso quer dizer que também devem ser considerados na evolução de *software* artefatos como Modelos, Transformações de Modelos, Geradores de Códigos e *Templates*. Na *MDE*, o sincronismo entre estes elementos torna-se ainda mais crítico, pois o processo quase sempre é apoiado por algum tipo de automação (Transformação de Modelos) envolvendo uma cadeia de artefatos específicos (Modelos).

O processo de desenvolvimento utilizado neste trabalho origina-se na figura do Engenheiro de Domínio que atua em um período inicial do ciclo de vida do sistema de *software*.

No ciclo resumido da Figura 2, o Engenheiro de Domínio tem a responsabilidade de produzir uma infraestrutura reutilizável, parametrizada e configurável dentro de um determinado escopo e em um determinado domínio de aplicações.

Essa infraestrutura inicial envolve principalmente o Gerador de Códigos em seu núcleo, que é composto des *Templates* para geração de códigos, entre tantos outros artefatos. Assim, em um segundo momento, o Desenvolvedor de Aplicações pode construir aplicações daquele domínio mais facilmente reutilizando os artefatos do domínio, e também fazendo uso da geração de código automática baseada em *Templates* (CZARNECKI;

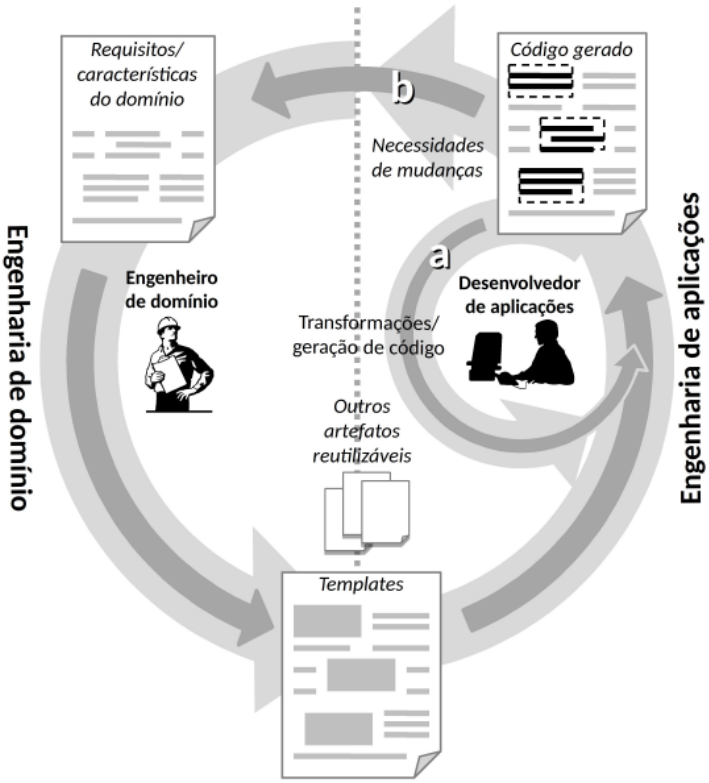

Figura 2: Ciclo resumido do desenvolvimento de *software* dirigido por modelos.

EISENECKER, 2000).

Existem duas formas possíveis de manter/evoluir o *software* no ciclo apresentado pela figura anterior.

1. O primeiro caminho, a letra ”a” na Figura 2, é o que representa o objetivo principal de todo o esforço de uma abordagem dirigida por modelos. Esse caminho é percorrido na necessidade de mudanças, e estas mudanças podem ser realizadas nos modelos e após uma nova sequência de transformações obtém-se código automaticamente gerado pelo Gerador de Códigos. Essa automação reduz bastante o esforço dispendido no processo. Como exemplo desse caminho, pode-se citar a alteração do nome de um atributo de uma classe do modelo, e após a alteração desse atributo, repete-se a geração de código e obtém-se a alteração no código da aplicação; e

2. O segundo caminho, representado pela letra ”b” na Figura 2, é percorrido quando se percebe a necessidade de mudanças nos artefatos de domínio. Por exemplo, pode haver um erro em um *template*, que é um artefato reutilizável, e para esta alteração será necessário retomar o processo de engenharia de domínio fazendo uma análise mais profunda das mudanças e o impacto destas no *software* antes de realizá-las.

Analisar e planejar mudanças em um conjunto de artefatos convencionais já é um desafio considerável que requer esforço e disciplina. Na *MDE*, a situação é agravada pela

presença dos Geradores de Códigos com seus *Templates* que são bem mais complexos de lidar, pois estes possuem informações de meta-modelagem misturando trechos de código que serão impressos no alvo com trechos que serão substituídos por outros trechos e relacionamentos com diversos modelos de dados, e isso pode tornar-se bastante complexo e até confuso. Exatamente por esse motivo, existe a possibilidade de utilizar uma implementação de referência como artefato de apoio ao desenvolvimento.

Uma Implementação de Referência é uma aplicação típica, e sua utilização dentro do processo dá-se analisando o seu código e identificando quais os trechos que se repetem e como eles variam entre si. E de posse dessas informações, aliada à expertise do Engenheiro de Domínios, implementa-se a variabilidade encontrada em uma linguagem capaz de representá-la por meio de *templates*. Por meio deste processo pode-se implementar um gerador que utiliza essas informações como parâmetro para o processo de geração (BIERHOFF; LIONGOSARI; SWAMINATHAN, 2006).

Nas duas figuras seguintes podemos ver um exemplo de utilização da Implementação de Referência na migração de um artefato de código gerado para o Gerador de Códigos na forma de um *template* de geração de códigos. Para um determinado formulário *Web*, os trechos de código com potencial de automação são identificados e substituidos por expressões na linguagem do *template* e este consegue reproduzir o código quantas vezes for necessário.

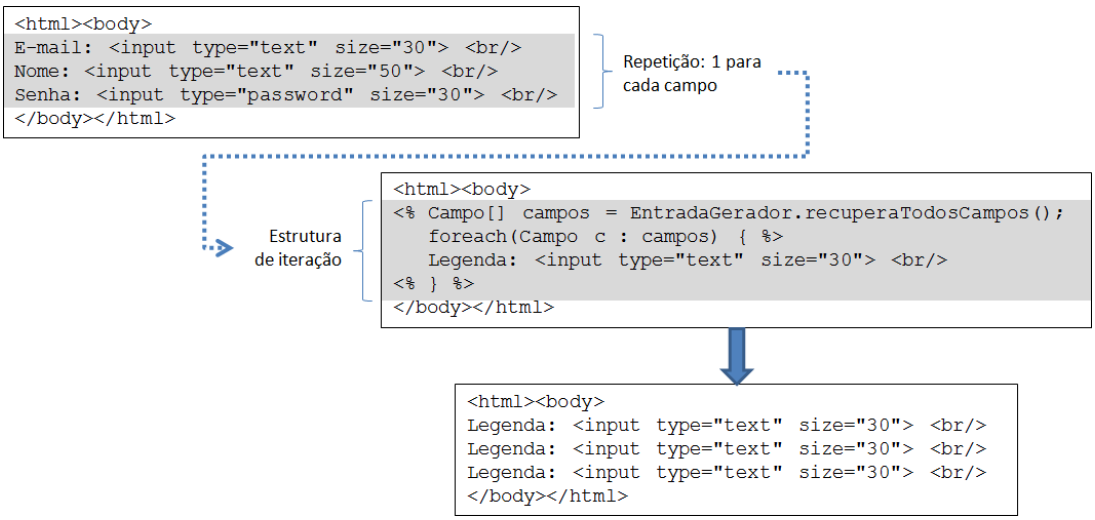

Figura 3: Primeiro exemplo de uso da Implementação de Referência na criação de *Templates*. Implementando iterações de dados.

Na Figura 3, a repetição foi detectada e reproduzida no *template* por meio de uma estrutura de iteração que acessa o modelo de dados. Desse modo, a estrutura de iteração imprime três vezes um texto e um campo *HTML* genérico no arquivo de saída.

Dando continuidade no processo, a Figura 4 nos mostra uma outra modificação no código do *template* para que esse substitua a legenda genérica de cada campo pelo seu

respectivo nome de elemento no modelo de entrada. Assim, a expressão destacada acessa, recupera e imprime o nome de cada campo no arquivo de saída.

As etapas seguintes do processo, que não são descritas aqui, modificariam o *template* incluindo expressões para a recuperação de informações sobre o tipo e tamanho de cada campo do formulário. Ao término desta migração parcial, tem-se um *template* completo.

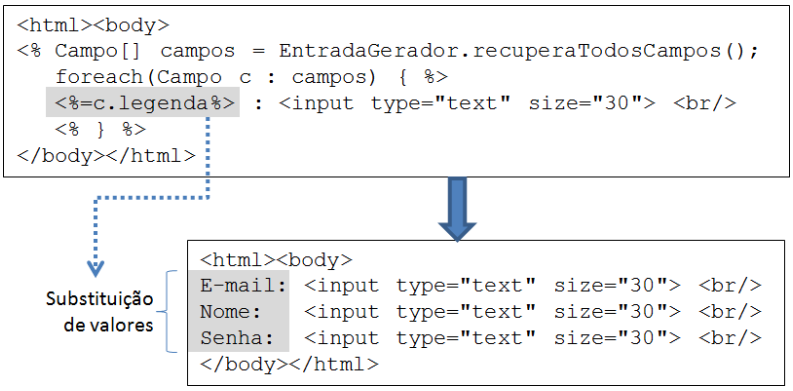

Figura 4: Segundo exemplo de uso da Implementação de Referência na criação de *Templates*. Implementando nomes de campos.

E, ao final de todo o processo de migração, tem-se um gerador completo que é capaz de gerar código idêntico à Implementação de Referência. E, no caso de mudanças na aplicação, estas são feitas e testadas na própria Implementação de Referência para depois serem migradas para o *template*.

Desenvolver geradores a partir da Implementação de Referência, a partir deste trecho referenciada como IR, traz vantagens como (MUSZYNSKI, 2005):

- Permite a geração e reutilização de maiores quantidades de código, pois reaproveita-se qualquer trecho de código preexistente;
- Produz geradores com maior qualidade, pois a IR é pré-testada, depurada e validada da maneira tradicional;
- Permite utilizar o ambiente de preferência do desenvolvedor para a construção da IR; e
- Possibilita que o desenvolvedor utilize funcionalidades que normalmente utiliza ao desenvolver *software*, como autopreenchimento, refatorações, ajuda contextual e depuração assistida que são fortemente associadas com código-fonte e que não funcionam no nível do gerador.

Considerando os três tipos de artefatos envolvidos neste processo de evolução: a IR, os *templates* e o código gerado (deste trecho em diante denominado CG), percebe-se

uma certa duplicação de dados nestes artefatos, pois ao final do processo de migração a IR é similar ao CG. Apesar disso, a IR possui as particularidades de ter sido escrita manualmente e de ser a base do desenvolvimento do gerador, e deste modo, não é aconselhável então seu mero descarte. Além do mais, a IR é o lugar onde as modificações são feitas.

Ao mesmo tempo, dentro do Gerador de Códigos podemos também observar uma duplicação de estruturas de dados no código dos *templates* com os outros dois artefatos anteriormente mencionados. O código de *templates*, apesar de todo decorado com expressões que acessam o modelo de dados, também é similar à IR e ao CG.

Entretanto, essas similaridades desaparecem quando levamos em conta a manutenibilidade destes artefatos. Pois o código de *templates* é completamente integrado ao modelo de entrada, o que torna a visualização das modificações uma tarefa complexa quando comparado à mudanças efetuadas nos outros dois tipos de artefatos.

Assim, considerando o problema da sincronização destes artefatos aliado à falta de controle e a dificuldade da visualização das alterações, buscou-se um mecanismo que possibilita modificações sincronizadas nestes artefatos e um maior controle sobre o impacto destas mudanças no código automaticamente gerado.

Deste modo, surgiu a hipótese da edição simultânea dos artefatos do tipo *Template* de geração de códigos considerando a propagação automática das modificações de um lado para o outro, o que possibilita a visualização *in loco* das modificações efetuadas do CG para os *templates*.

1.3 Objetivo

Este trabalho teve seu foco na criação de uma ferramenta que facilitasse o processo de evolução de *software* na *MDE*. Em particular, pretende-se facilitar a localização e realização de mudanças em *templates* de geração de códigos.

A abordagem foi construída reutilizando os resultados de um trabalho anterior feito por Possatto (2014). Em seu estudo, é utilizado o conceito de Implementação de Referência, e com isso, pode-se trabalhar inicialmente direto no código convencional, ou seja, sem a presença de Geradores de Códigos. Assim sendo, as mudanças são realizadas, testadas e validadas normalmente, como em um processo não dirigido por modelos.

Nesta abordagem, um editor de código simultâneo para *Templates* de geração de códigos é utilizado na manutenção e a evolução do Gerador de Códigos. O conceito de Implementação de Referência também é explorado, e o editor permite a sincronização desta com os *Templates* de geração de códigos.

Na Figura 5, temos um esquema do uso do editor. Modificações são feitas no

CG/Implementação de Referência (1); o mecanismo interno do editor reconhece as mudanças (2), e as replica para o *template* atualmente em edição (3).

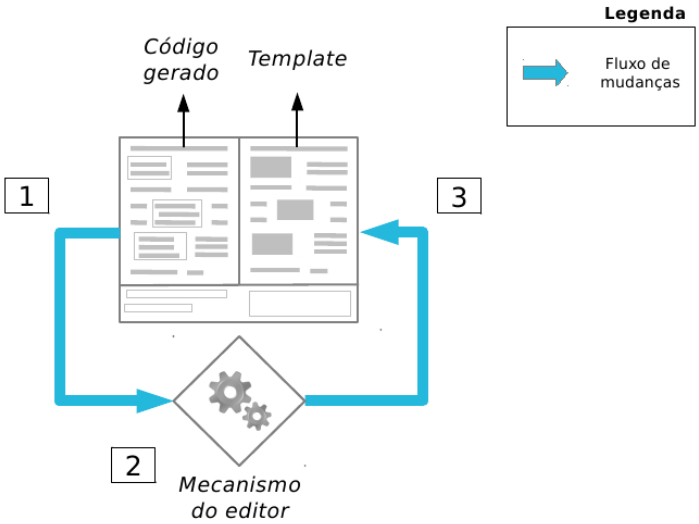

Figura 5: Esquema do protótipo do editor.

Uma vez efetuada uma alteração, esta é imediatamente propagada para o *template*. E, depois que o *template* está atualizado, novas gerações de código passam a incorporar as mudanças no código gerado. O código gerado então torna-se a Implementação de Referência. Essa não é a melhor solução, pois um código escrito à mão deveria ser tido sempre como uma referência e nunca substituído por código gerado. Mas neste caso essa substituição é necessária, pois os *scripts* automáticos de migração dependem de registros especiais que são produzidos somente para o código gerado. Na teoria, perde-se por um lado, pois a Implementação de Referência deixa de ser um exemplar feito à mão, mas ganha-se com a automação do processo. Uma análise mais aprofundada do custo-benefício desta abordagem sobre o processo de evolução foi excluída do escopo deste trabalho por não haver tempo hábil para tal.

1.4 Relevância

O protótipo faz o uso do mapeamento de códigos entre artefatos, e assim acrescenta um novo método para a sincronização de artefatos relacionados dentro do processo de geração de códigos.

A edição de *templates* e de seu código gerado, fortemente embasada pelos mapeamentos entre códigos, mostra uma edição de *templates* que vai além da simples edição dos tipos de uma linguagem de *templates* quando permite a edição simultânea entre este e o seu CG. O que traz um avanço para os editores de *templates*.

A experiência de edição simultânea com janelas flexíveis do protótipo, pretende contribuir para com os editores simultâneos facilitando o processo de edição de artefatos. Deste modo busca facilitar a visualização e o entendimento de artefatos e também de seus relacionamentos, sejam estes um-para-um ou um-para-muitos, diminuindo o tempo do processo de edição destes.

O presente trabalho propõe uma edição especializada simultânea e portanto uma evolução no sentido de oferecer maior controle ao desenvolvedor sobre o processo de evolução de *software* neste processo. Em vez de uma migração automática do código, propõe-se a edição simultânea entre os dois tipos de artefatos de geração de códigos envolvidos de modo passo a passo.

Sua natureza interativa, permite uma evolução incremental do código dando mais interatividade ao processo. Além do mais, sua *interface* lado a lado amplia o conceito de interatividade promovendo um ambiente que dá margem a experimentação e criatividade dentro do processo de evolução de *software*.

1.5 Organização

O texto está organizado em 7 capítulos distintos.

O Capítulo 1 iniciou o trabalho contextualizando-o, descrevendo o seu cenário de uso e justificando a sua relevância para o meio acadêmico. O Capítulo 2 concentra a pesquisa que fundamentou este trabalho. Na sequência, o Capítulo 3 apresenta outros trabalhos relacionados a este.

O Capítulo 4 apresenta e descreve a ferramenta computacional desenvolvida na forma de um protótipo. Dando continuidade, o Capítulo 5 descreve o estudo experimental levado a cabo para a avaliação desta abordagem de modo empírico. Na sequência, e com o foco em resultados, o Capítulo 6 traz a interpretação dos dados do estudo empírico efetuado. Finalizando, temos o Capítulo 7 que conclui, aponta as limitações e dá novos rumos de pesquisa a este.

2 Fundamentação teórica

2.1 Engenharia de *Software* Dirigida por Modelos (*Model-Driven Engineering*)

A Engenharia de *Software* Dirigida por Modelos é baseada no uso de artefatos específicos chamados Modelos e Transformações de Modelos. Os modelos representam informações e são considerados por esta abordagem como seus principais elementos (HETTEL; LAWLEY; RAYMOND, 2008). Ao mesmo tempo, as Transformações de Modelos tem semelhante importância pois é por meio de transformações de dados que o sistema de *software* é moldado em sua forma final (SENDALL; KOZACZYNSKI, 2003).

Bézivin (2004) sintetizou o princípio básico da Engenharia Dirigida por Modelos com o pensamento "tudo é um modelo" . Essa frase apesar de ter soado forte, do mesmo modo como soou forte o pensamento "tudo é um objeto" na década de 80, motivou inúmeras pesquisas na área que frutificaram em muitos métodos, técnicas, abordagens, gerando assim mais pesquisa na Engenharia de *Software*.

A *MDE* deslocou o centro de atenção do código para os modelos no desenvolvimento de sistemas de *software* (BÉZIVIN, 2004), e isso veem mudando o jeito de pensar *software*. A essência dessa abordagem é a abstração, nos termos de como pensamos no problema e especificamos a solução, e também a automação dentro do processo de desenvolvimento de *software* (SELIC, 2003). Com esses dois preceitos em mente, e na posse de métodos que buscam pelo aumento da produtividade, interoperabilidade, comunicação, escalabilidade, portabilidade, manutenibilidade e reúso - temas recorrentes dentro do processo de desenvolvimento de *software* – esses métodos ganham novos ares e a sinergia disso veem contribuindo sobremaneira para a Engenharia de *Software*, aumentando a qualidade de *software* e diminuindo o tempo de entrega de *software* e o impacto da evolução em *software*.

2.1.1 Artefatos da *MDE*

Dentro de seu processo de desenvolvimento, encontram-se produtos especializados denominados artefatos.

Um artefato pode ser descrito como um conjunto de dados que segue um determinado esquema. Podem ser classificados de diferentes maneiras, ora pelo seu conteúdo ora pelo seu tipo, *e.g.*, do tipo código fonte; do tipo programas executáveis; de arquivos texto para configuração e documentação.

Dentre os principais tipos utilizados dentro da abordagem, existem dois tipos principais que são detalhadamente apresentados a seguir:

Modelos: Existem várias definições como pode ser visto a seguir:

- "Um modelo é uma simplificação de um sistema construído com um objetivo em mente. O modelo deve ser capaz de responder perguntas no lugar do sistema real." (BéZIVIN; GERBé, 2001);
- "Os modelos oferecem abstrações de um sistema físico que permitem aos engenheiros raciocinar sobre esse sistema, ignorando detalhes irrelevantes enquanto incidindo sobre os relevantes." (BROWN, 2004);
- "Modelos de engenharia têm como objetivo reduzir o risco, ajudando-nos a compreender melhor tanto um problema complexo como suas possíveis soluções antes de proceder com as despesas e o esforço de uma implementação completa." (SELIC, 2003); e
- "Um modelo de um sistema é uma descrição ou especificação desse sistema e seu ambiente para algum determinado fim." (MILLER; MUKERJI et al., 2003).

Apesar da clareza e da qualidade de cada uma das citações, podemos perceber pontos de vista diferentes. E este trabalho, em vez de optar por uma destas, busca em Muller et al. (2012) um estudo mais aprofundado e formal sobre padrões entre modelos que define um conjunto de quatro relações essenciais de representação de uma coisa por uma outra coisa em um processo de modelagem. E com as seguintes relações, esses pesquisadores não criaram uma nova definição, mas sim ofereceram um meio de classificar modelos. A seguir os padrões de relacionamento:

1. Intenção: Quanto a intenção na representação do modelo. Pode (i) diferenciar, (ii) ser a mesma, (iii) ser subconjunto, (iv) ser superconjunto ou de (v) compartilhar a representação;
2. Natureza: O modelo é utilizado para (i) expressar uma descrição, tendo assim uma natureza analítica, ou (ii) para expressar uma especificação, tendo então uma natureza sintética;
3. Causalidade: Quando e como a representação é estabelecida ou mantida. Podendo ser (i) contínua ou (ii) discreta;
4. Transitividade: Pode ser uma composição de representações de mesma intenção.

Deste modo, um determinado modelo de classes tem uma natureza descritiva quando relacionado ao sistema em que esse se baseia, *i.e.*, podendo ser utilizado na especificação do sistema de *software*. Ao mesmo tempo, esse mesmo modelo de classes tem uma natureza sintética quando foi utilizada para a geração de um esqueleto de código Java (JOY et al., 2000) deste ou de outro sistema de *software*. Além disso, levando em consideração este mesmo modelo em um cenário de sincronização de artefatos com engenharia ida-e-volta, onde modelos de classes geram esqueletos de código *Java*, esse modelo tem ainda uma relação causal contínua com este esqueleto de código *Java*, pois o primeiro deve ser atualizado quando o segundo muda. Assim, por meio destas relações amplia-se o conceito de modelos ganhando novas formas para classificá-los e entendê-los dentro de diferentes processos em que participem.

Dentro da abordagem *MDE*, existem mais dois tipos de modelos específicos de bem mais alto nível de abstração que são chamados metamodelo e meta metamodelo. O metamodelo, também conhecido como ”MM” ou ”M2”, é um modelo de mais alto nível que define o modelo, sendo o modelo uma instância deste. O metametamodelo, também conhecido como ”MMM” ou ”M3”, especifica o esquema das possíveis construções que podem ser usadas em um metamodelo. Não existe nível mais alto de abstração que o metametamodelo, pois este já se auto descreve (KURTEV et al., 2006).

Transformações de Modelos: São transformações de dados que geram artefatos a partir de outros artefatos. Estas transformações desenvolvem, mantém e evoluem *software* a partir de uma especificação como entrada e geram uma outra especificação ou uma implementação como saída. Isso se dá por meio de mapeamentos entre entrada e saída previamente definidos, *e.g.*, uma Transformação de Modelos do tipo de Modelo para Modelo, onde um um modelo de classes pode ser transformado em um outro modelo de classes reduzido.

A seguir, uma terminologia auxiliar, descrevendo elementos que compõem uma Transformação de Modelos (BIEHL, 2010):

1. *Fonte* (também conhecido como ”entrada”, ”origem”): É a entrada do processo de transformação. Transformações utilizam-se de uma entrada de dados para alimentar o processo de transformação;

2. *Alvo* (também conhecido como ”saída”, ”destino”): É a saída do processo de transformação. Transformações geram uma saída que deve estar em conformidade semântica com o modelo de entrada do processo.

3. *Linguagem de transformação*: Executa e controla o fluxo de transformações;

4. *Roteiro de transformações*: Escrito na linguagem de transformação descrevendo as transformações; e

5. *Mecanismo de transformação*: Executa ou interpreta o roteiro de transformação. Basicamente localiza o elemento na entrada, executa uma ação associada de transformação e direciona para a saída.

A Figura 6 traz os principais artefatos da *MDE*. Modelos (1) são utilizados como entrada no processo. Transformações (2) são aplicadas nessa entrada dando origem a artefatos alvo que visam uma plataforma específica (3), ou mesmo dando origem a modelos intermediários (4) que realimentarão o processo de refinamento de modelos abstratos para modelos concretos (5). Deste modo, ao final obtém-se o sistema de *software* especificado.

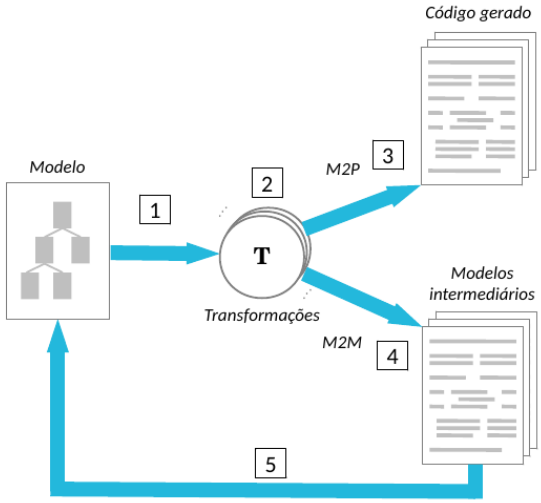

Figura 6: Principais artefatos e o processo de geração de código dentro da abordagem dirigida por modelos.

Um esquema de classificação de propriedades características de transformações e de linguagens de transformação em seus cenários de uso, são listados a seguir (BIEHL, 2010):

1. *Mudanças na abstração*: As transformações podem mudar o nível de abstração, que é medido em quantidade de detalhes, entre a fonte e o alvo. Mudanças na abstração podem ser: (i) *horizontais* mantendo assim o nível de abstração, *e.g.*, uma refatoração de código; (ii) *verticais de refinamento* adicionando detalhes ao alvo, *e.g.*, transformações de modelo para código; e (iii) *verticais de abstração* reduzindo a quantidade de detalhes no alvo.

2. *Mudanças de metamodelos*: As transformações podem ser realizadas com fonte e alvo em diferentes metamodelos. Mudanças de metamodelos podem ser: (i) *endógenas* quando a fonte e o alvo possuem o mesmo metamodelo; (ii) *exógenas* quando a fonte e o alvo estão em dois diferentes metamodelos, *e.g.*, a transformação de um modelo EMOF [1]

[1] *EMOF (Essential MOF)* é o meta metamodelo criado pela *OMG*.

para *Ecore* [2] (EMF, 2013).

3. *Espaços Técnicos suportados*: Um Espaço Técnico [3] (KURTEV; BéZIVIN; AKSIT, 2002) como um conjunto de trabalho dentro de uma tecnologia (deste ponto em diante referenciado como TS), pode ser considerado como limitante e caracterizador de transformações. As transformações então podem dar-se: (i) *dentro de um mesmo TS*; (ii) *entre diferentes TS*, *e.g.*, uma transformação que tem como fonte um modelo *EMF* e como alvo um modelo *EBNF (Extended Backus-Naur Form)*;

4. *Número de artefatos suportados*: É a quantidade de modelos envolvidos na transformação, seja tanto na fonte ou no alvo. Pode ser especificado por meio de variações de elementos do conjunto 1, n (um para muitos) formando uma tupla. Um caso especial é a transformação em que fonte e alvo são o mesmo modelo, e esta transformação é chamada de "transformação *in-place*";

5. *Tipo do alvo suportado*: Essa classificação distingue a transformação pelo tipo do alvo gerado. Ao invés de aqui utilizarmos os dados de Biehl (2010), que cita transformações de modelo para modelo e transformações de modelo para código, vamos citar Voelter (2014) que possui uma visão mais precisa quando afirma que o alvo gerado pode ser (i) *um modelo* ou um (ii) artefato de plataforma. O que leva a dois tipos de transformação classificados quanto ao tipo do alvo gerado: as transformações modelo para modelo, também conhecidas como "*M2M*", e as transformações de modelo para plataforma, também conhecidas como "*M2P*"; e

6. *Preservação de propriedades*: Diz respeito às propriedades entre fonte e alvo que serão mantidas após a transformação. Transformações podem preservar: (i) *preservação de propriedades semânticas*: semântica preservada após a aplicação da transformação, *e.g.* refatorações; e (ii) *preservação de propriedades sintáticas*: mantendo a sintaxe abstrata preservada, *e.g.*, a sintaxe abstrata da fonte é preservada no alvo, mas a sintaxe concreta não.

As Linguagens de transformação possuem as seguintes propriedades características:

1. *Paradigma*: É o paradigma da linguagem de transformação. Pode ser: (i) *imperativo*: A linguagem é definida como uma sequência de ações, *e.g.*, Velocity (2013); (ii) *declarativo*: A linguagem define o que deve ser mapeado, *e.g.*, Acceleo (2014); (iii) *híbrido*: As transformações aceitam tanto a linguagem imperativa como a declarativa, *e.g.*, ATL

[2] *Ecore (Essential Core)* é o meta metamodelo criado para o *framework EMF (Eclipse Modeling Framework)*.

[3] Espaços Técnicos referem-se às tecnologias usando um nível mais alto de abstração, *e.g.*, o Espaço Técnico *EMF* inclui aqui toda a sua *foundation*, os *frameworks*, as ferramentas, enfim todo o ecossistema *EMF*. Espaços técnicos então são contextos de trabalho com um conjunto de conceitos associados, conhecimentos, ferramentas, habilidades e possibilidades.

(2014); (iv) *baseado em grafos*: A linguagem é baseada nos fundamentos teóricos das gramáticas de grafos algébricos e são declarativas por natureza (KINDLER; WAGNER, 2007); (v) *baseado em Templates*: A linguagem contém fragmentos de texto e meta elementos que serão substituídos por dados do modelo, *e.g.*, o *JET* (POPMA, 2004a); e (vi) *manipulação direta*: A linguagem acessa o modelo por meio de uma API específica;

2. *Controle de aplicação*: É a ordem de aplicação das transformações. Pode ser: (i) *implícito*: A linguagem não permite a especificação da ordem da aplicação de regras; (ii) *explicito*: A linguagem pode especificar a ordem de execução de regras; (iii) *externo*: A linguagem especifica a ordem de execução separado das regras; e (iv) *escopo de aplicação de regra*: restringe o escopo de aplicação de regras a uma determinada parte do modelo.

3. *Organização das regras*: É o modo da linguagem de compor transformações na forma de grupos. A composição pode ser: (i) *interna*: A linguagem pode compor transformações; e (ii) *externa*: A linguagem pode compor transformações de diferentes linguagens.

4. *Rastreabilidade*: Uma linguagem pode implementar mecanismos de rastros de execução de transformações. Os rastros mapeiam elementos entre a fonte e o alvo. Rastros para sincronização podem então ser gerados, capturados e armazenados em modelos, banco de dados ou no alvo;

5. *Direcionalidade*: É a direção da transformação entre fonte e alvo. Pode ser: (i) *unidirecional* permitindo o mapeamento entre fonte e alvo, ou (ii) *multidirecional* permitindo mais de uma direção na interpretação da regra entre fonte e alvo;

6. *Modelo de Transformações Incremental*: Em alguns mecanismos de transformação, transformações atualizam o alvo após alguma modificação no fonte, e esta atualização pode ser: (i) *não-incremental*: Este tipo gera o alvo completo sobrescrevendo-o; ou (ii) *incremental/persistida*: Permite atualizações no fonte e a propagação das mudanças para o alvo atualizando-o. Podem ser (iia) *incremental do alvo*: Este tipo gera somente parte do alvo, fazendo uma reconstrução seletiva do alvo, ou (iib) *incremental do fonte*: Este tipo minimiza a quantidade de elementos do fonte que precisam ser rechecados em uma transformação incremental; e

7. *Representação da transformação*: Podem representar transformações como (i) *texto*; ou como (ii) *modelo*. Sendo modelos, podem manipular outras transformações utilizando o conceito de *Higher-Order Transformations* (*HOT*) (MULIAWAN, 2008).

Deste modo, a classificação de linguagens de transformação e a classificação dos tipos de transformações nos dão uma visão mais ampla do potencial das Transformações de Modelos, ao mesmo tempo que, permitem o compartilhamento comum de conceitos.

2.1.2 Geradores de Códigos e *Templates* de geração de códigos

Os Geradores de Códigos são mecanismos de transformação que englobam diversas Transformações de Modelos em sua organização. E, além de permitir a geração de trechos de código executável ou mesmo para configuração, podem agregar também uma série de benefícios à atividade de desenvolvimento de *software*, pois produzem sempre o mesmo código de forma previsível, controlada. Por exemplo, um gerador pode automaticamente aplicar padrões de código de forma sistemática, com menor taxa de erros (CLEAVELAND, 1988).

Geradores de Códigos automatizam tarefas para um ser humano, algumas destas extremamente difíceis, *e.g.*, simulação e outras somente trabalhosas ou repetitivas, *e.g.*, validação de código em um menor tempo e com menor taxa de erros. No entanto, o desenvolvimento de *software* envolve um alto grau de criatividade, e o uso de geradores não pode ser empregado em todas as fases do ciclo de desenvolvimento onde há a necessidade da criatividade. Mas pode e deve ser utilizado de forma a automatizar tarefas tediosas e de repetição liberando o ser humano para as habilidades naturais deste. Quando adotada corretamente, a geração de códigos pode levar a ganhos importantes em termos de produtividade e qualidade (TOLVANEN; KELLY, 2005).

Entre as dificuldades existentes para se projetar um Gerador de Códigos, destacam-se Cleaveland (1988), Hamza (2005), Hessellund, Czarnecki e Wasowski (2007), Jouault, Bézivin e Kurtev (2006), Korhonen (2002), Mernik, Heering e Sloane (2005), Paul (2005), Wile (2004):

- A necessidade de conhecimentos diversos para se projetar a linguagem de especificação, tais como conhecimentos de Projetista de *Software*, Engenheiro de *Software*, Autor de manuais e treinamento, Especialista de domínio, Especialista em linguagens, entre outros;
- A dificuldade em se reconhecer os limites do domínio;
- A integração de múltiplas linguagens de especificação, *i.e.*, ter em mente a linguagem de entrada, a linguagem de transformação e a linguagem de saída;
- A reutilização dos artefatos para geração de código; e
- A evolução do *software*.

Na ponta final destes geradores encontram-se os *Templates* que especificam e delimitam a sintaxe do código a ser gerado por meio de variáveis, técnicas de expansão de texto e comandos de controle do fluxo de execução que estão embutidas dentro do mecanismo do próprio *template*. A grosso modo, esses atuam como se fossem carimbos à

tinta comuns, mas com facilidades de formatação e configuração de texto em tempo de execução. Assim sendo, o texto de saída gerado por meio de um determinado *template* é o resultado específico de determinadas entradas nesse Gerador de Códigos que sofreram transformações de dados.

São muitas as vantagens dessa técnica, destacando-se a facilidade de adoção por parte de desenvolvedores, uma vez que essa abordagem é agnóstica quanto à linguagem gerada (CZARNECKI; EISENECKER, 2000). Pode-se também citar o fato de que, 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, à exceção de outras construções de geração de código, o *template* é bastante parecido com a saída desejada (CLEAVELAND, 2001).

Ao se optar por implementar a especificação do código alvo por meio de *Templates*, estes estarão intimamente ligados ao Gerador de Códigos da solução e também a muitos outros artefatos do processo. A Figura 7 apresenta a técnica de geração de códigos baseada em *Templates*. O modelo contém dados estruturados com informações sobre elementos do sistema de *software* (1). O *template*, que representa a sintaxe da linguagem alvo, contém as referências aos elementos do modelo (2). O Gerador de Códigos (3) interpreta o modelo do projeto e aplica um determinado conjunto de *templates* a esse, e produz o código de saída (4). O código alvo gerado geralmente é composto de um ou mais arquivos textuais e foi obtido por meio da substituição das referências internas dos *templates* pelos dados provenientes do modelo.

Ainda na mesma Figura 7, podemos visualizar algumas particularidades dos artefatos da solução:

- O Modelo, no formato *XML* [4], especifica o conteúdo de informações de cada formulário, tais como os campos, tipo de dados, tamanho, etc;
- O *Template* é um arquivo *JSP* [5] anotado com códigos de controle de fluxo e com variáveis que serão impressas em tempo de execução;
- O Gerador de Códigos (ou Processador de *Templates*, ou Mecanismo de Transformações) instancia o *template*;
- O código gerado é uma página *HTML* [6] de acordo com o modelo.

[4] *XML* é uma tecnologia que oferece uma linguagem de marcação para estruturação e intercâmbio de dados.

[5] *JSP* (*Java Server Pages*) é uma tecnologia que oferece uma linguagem para a criação de páginas *Web* dinâmicas basedas em *HMTL*, *XML*, entre outros.

[6] *HTML* (*HyperText Markup Language*) é uma tecnologia que oferece uma linguagem de marcação para a criação de páginas *Web*.

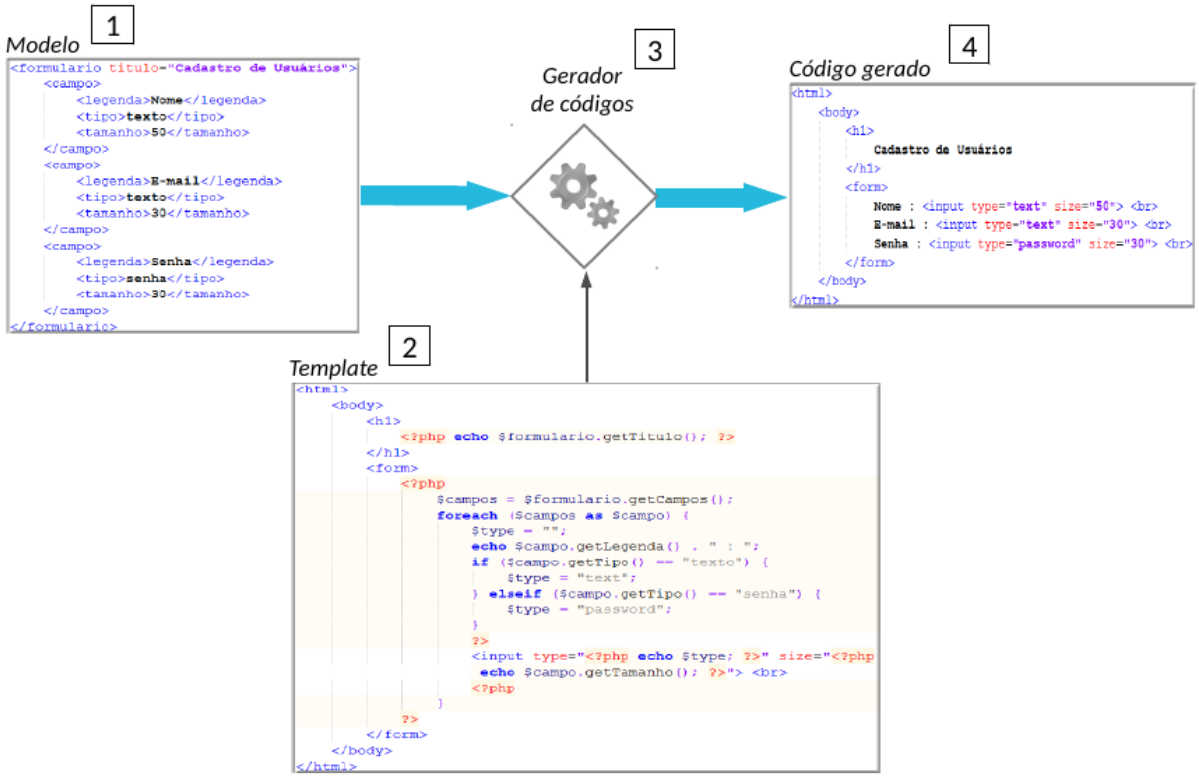

Figura 7: Exemplo do uso de *Templates* na geração de código.

2.1.3 A Implementação de Referência e o processo de Migração de Códigos

Uma certa aplicação típica, previamente verificada e validada para um determinado domínio, pode vir a ser utilizada como ponto de partida para o desenvolvimento de Geradores de Códigos. Essa abordagem mostrou-se bem sucedida, de acordo com relatos encontrado na literatura (BIERHOFF; LIONGOSARI; SWAMINATHAN, 2006).

Esta aplicação que primeiramente alimenta o processo, é denominada de Implementação de Referência, e é um código fonte que é base para o processo de geração. Este código geralmente é um código funcional livre de erros, desenvolvido fora do ambiente do gerador, um todo ou parte de uma aplicação.

O processo seria: desenvolve-se o código, valida-se e verifica-se o código, e então, este código é migrado para o gerador (MUSZYNSKI, 2005). Deste modo, nesta implementação típica, os trechos de código são analisados identificando quais destes trechos se repetem e como eles variam entre si. Com base nesta informação, implementa-se essa variabilidade em uma linguagem capaz de representá-las por meio de *templates* e por fim, utilizando estas informações como parâmetros para a geração de códigos. Com a evolução do processo, novas nuances de variação são identificadas e o *template* evolui até ser capaz de atender a uma maior variedade de soluções. Abordagens similares são relatadas por Visser (2007) e Wimmer et al. (2007).

A Figura 8 auxilia no entendimento do processo de análise da IR. A IR é varrida

buscando-se pela variabilidade das informações contidas no código. Neste processo códigos de controle de fluxo são separados das informações da aplicação, do mesmo modo que, o código genérico da implementação relacionado com a plataforma alvo deve ser reconhecido e separado (1). Informações relativas ao modelo da aplicação e informações que virão a definir os próprios modelos são identificadas e separadas (2). Do mesmo modo, informações sobre transformações de dados, *e.g.*, informações da ordem de aplicação de transformações e seus *scripts*, também são identificados e separados de informações relativas à plataforma destino (3).

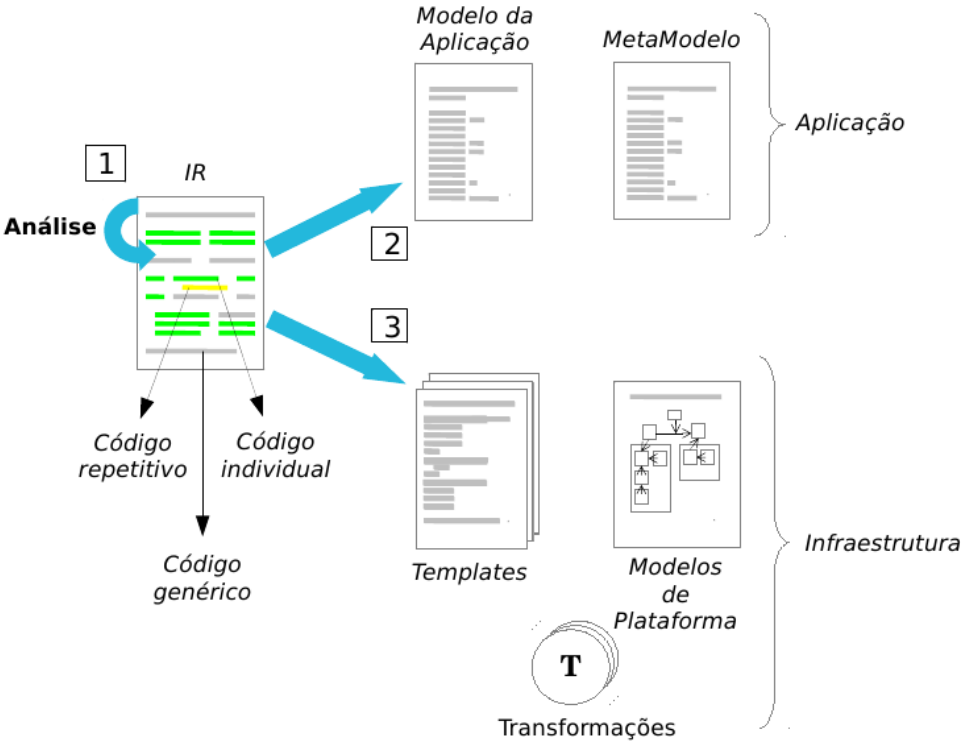

Figura 8: Análise da IR.

A IR utilizada por este trabalho é uma aplicação *Web* que denominamos "Aplicação *Web* de Referência". Este trabalho parte do princípio que esta aplicação já foi verificada, validada e já se encontra migrada para dentro dos Geradores de Código.

Neste cenário tecnológico, temos a necessidade da geração de formulários *HTML* para a Aplicação *Web* de Referência. São vários formulários que devem ser criados para as muitas entidades do sistema de *software* possuindo uma mesma característica operacional. A variabilidade nesse caso são os campos que cada formulário deverá conter. A criação manual de todos estes formulários seria algo extremante repetitivo e claramente tenderia ao erro por diversos motivos particulares a este processo.

Nesse caso a geração de código automática baseada em *templates* soluciona o problema anterior com primazia. Após a verificação e validação dos *templates*, estes produzirão um código de saída controlado que reproduz fielmente o especificado no modelo.

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

- Permite a geração e reutilização de maiores quantidades de código, pois reaproveita-se qualquer trecho de código preexistente;
- A inclusão incremental das anotações facilita a criação de geradores não-triviais;
- Produz geradores com maior qualidade, pois a IR é pré-testada, depurada e validada da maneira tradicional;
- Permite utilizar o ambiente de preferência do desenvolvedor para a construção da IR; e
- Possibilita que o desenvolvedor utilize funcionalidades que normalmente utiliza ao desenvolver *software*, como autopreenchimento, refatorações, ajuda contextual e depuração assistida que são fortemente associadas com código-fonte e que não funcionam no nível do gerador.

A utilização da IR para a construção de Geradores de Código mostra-se bastante atrativa, mas o principal problema desse processo é o custo extra da migração, que é estimado em cerca de 20-25% do tempo de desenvolvimento da IR (MUSZYNSKI, 2005). No desenvolvimento inicial, quando não existe artefato algum, esse custo extra é certamente amortizado após duas ou três gerações de novos sistemas. Mas uma vez que os geradores estejam prontos e sendo utilizados, existe um problema adicional, associado a evolução de *software*. Ao longo de seu ciclo de vida, estes geradores certamente precisarão evoluir, seja para corrigir erros ou incorporar novas funcionalidades. É nesse momento que, com a presença da IR, existirá uma duplicação de código.

Assim, após a migração do código da IR para o gerador, tem-se os *templates*, que são uma representação da IR dentro do gerador, e, após o processo de geração de códigos, tem-se também o CG a partir dos *templates*. Ou seja, um mesmo trecho de código, salvo algumas modificações, pode aparecer repetido em três artefatos diferentes.

Neste cenário, a partir de uma necessidade de modificação no código - em decorrência do processo evolutivo do *software* - teoricamente é possível alterar qualquer um destes três diferentes artefatos. Dependendo do artefato escolhido, tais mudanças gerariam impactos diferentes no processo:

- Modificação no código alvo: alterações podem ser realizadas diretamente no CG, o que de certa forma é a maneira mais fácil de realizá-las, mas também a mais desaconselhável. Deste modo, as alterações não serão propagadas para o domínio, o que significa que futuras gerações não irão incorporar as mudanças. E por exemplo, em caso de uma correção de erro, a mudança teria que ser repetida toda vez que um novo código é gerado.

Além disso, essas mudanças irão causar inconsistência entre o CG e os *Templates*, que precisa ser devidamente documentada e gerenciada. Para eliminar essa inconsistência, pode-se utilizar o CG como nova IR. O problema é que a noção de IR vem do fato de ser uma solução escrita a mão e já depurada e testada. Este caminho teria como resultado negativo a perda desta implementação de caráter manual por uma implementação de caráter automático;

- Modificação nos *templates*: Ao se modificar os *templates*, as alterações são automaticamente incorporadas ao domínio, passando a figurar em futuras gerações. Mas devido a natureza complexa do artefato, como mencionado anteriormente, realizar alterações diretamente neste código pode ser uma tarefa onerosa pois é difícil localizar as mudanças, analisar seu impacto, testá-las e validá-las. Do mesmo modo a falta de um ambiente especializado de edição e de depuração também dificulta o processo. Além disso, ao se modificar diretamente o *template*, perde-se o sincronismo com a IR, o que também pode ser problemático em futuras mudanças mais conceituais; e

- Modificação na IR: Efetuando-se as modificações necessárias na IR, tem-se a vantagem de se trabalhar em código convencional. Estas modificações são efetuadas fora do ambiente do gerador e portanto deverão ser novamente migradas para dentro deste, gerando um custo extra. Mas apesar do custo extra, não se perde a consistência entre os três artefatos, ou seja, a IR continua consistente com os *templates*, que continuam consistentes com o CG.

Como pode ser visto, as três opções anteriores apresentam desvantagens. Algumas mais que outras, como é o caso da primeira. Mas que, dependendo do cenário e da natureza das mudanças, o desenvolvedor pode optar por uma ou outra forma. Pode-se argumentar que a terceira opção é a preferida por gerar menos inconsistências, mas ainda assim gera esforço extra não havendo então uma escolha ideal para todos os casos. Mas seja qual for a opção, o problema da sincronização precisa ser tratado.

2.2 A sincronização de artefatos

Com a evolução do *software*, artefatos serão modificados e estes quando participam como alvos no processo de geração de código, não mais refletirão os resultados das transformações que os geraram. É nesse contexto que surge a necessidade da sincronização de artefatos. Pesquisadores, dentro do desenvolvimento dirigido por modelos, tem utilizado a engenharia ida-e-volta (*Round-Trip Engineering ou RTE*) para lidar com o problema da sincronização de artefatos (MUSZYNSKI, 2005; HETTEL; LAWLEY; RAYMOND, 2009).

A engenharia ida-e-volta (deste ponto em diante referenciada como *RTE*), é o ramo da Engenharia de *Software* que atenta para o processo de sincronização consistente entre artefatos, por meio de relacionamentos entre elementos de entidades, possibilitando que artefatos sejam sincronizados.

Sincronização é o processo de reforçar a consistência entre um conjunto de artefatos relacionados (ANTKIEWICZ; CZARNECKI, 2008) e é vital para a *MDE* para manter um sistema de modelos mutuamente consistente (DISKIN; XIONG; CZARNECKI, 2011).

Transformações podem ser usadas para a sincronização de artefatos, mas nem todas as transformações fazem sincronizações. Embora sincronizações possam ser vistas como um tipo de transformação que modificam algum alvo para fazê-lo consistente com alguma fonte, a diferenciação existe, pois uma sincronização (re)estabelece relacionamentos entre réplicas (parciais), *i.e.*, artefatos que se sobrepõe semanticamente e que existem em paralelo (CZARNECKI et al., 2009).

Uma adaptação da classificação de sincronizações de modelos na *RTE*, compilada em Hettel, Lawley e Raymond (2009) e complementada por (BUCAIONI, 2013) pode ser usada como terminologia auxiliar:

1. *Direção da transformação*: Pode ser (i) *progressiva* ou (ii) *reversa*. A direção reversa pode ser obtida (ii.a) *automaticamente* pela linguagem de transformação; ou (ii.b) *computada*;

2. *Automação da geração*: Pode ser (i) *automática*; (ii) *semiautomática* onde a direção reversa foi derivada da progressiva; ou (iii) *manual*;

3. *Domínio da transformação*: Pode ser (i) *total* onde há correspondência de todos os elementos entre fonte e alvo; ou (ii) *parcial* cuja fonte não possui todos os elementos para criação do alvo. Neste caso a relação entre fonte e alvo pode ser (ii.a) *injetiva*; ou (ii.b) *não-injetiva*. Um relacionamento não-injetivo pode gerar múltiplas soluções, pois o alvo possui elementos que tem correspondência e com mais de um elemento da fonte. Uma solução para o mapeamento pode ser derivada da situação ou uma interação humana deverá acontecer para a escolha entre as possíveis opções; e

4. *Rastros*: Implementados por meio de (i) *links*; ou por meio de (ii) *elementos*. Os *links* são obtidos na relação entre os elementos por meio de um mapeamento, e se elementos de rastro são criados estes podem ser persistidos em algum tipo de armazenamento (BUCAIONI, 2013).

A noção de rastreabilidade é o grau no qual um relacionamento pode ser estabelecido entre dois ou mais produtos do processo de desenvolvimento, especialmente produtos que tem um relacionamento predecessor/sucessor ou mestre/subordinado (IEEE, 1990).

Sob a ótica da rastreabilidade em transformações, os relacionamentos entre fonte e alvo definem rastros. Estes rastros possuem determinadas características como (GALVÃO; GOKNIL, 2007):

1. Podem ser (i) *explícitos* possuindo *links* diretos ou gerados por transformações ou (ii) *implícitos* tendo os seus links derivados a partir de informações de um elemento;
2. Possuem uma relação de dependência do tipo (i) *intra nível* se os seus rastros são entre elementos de um mesmo nível de abstração; ou do tipo (ii) *inter nível*, cujos rastros são entre elementos de diferentes níveis de abstração.

Abordagens de sincronização por meio de transformações precisam criar e manter os relacionamentos entre fonte e alvo (AIZENBUD-RESHEF et al., 2006). Estas transformações podem ser por:

1. *Composição* de transformações unidirecionais; ou
2. *Bidirecionais*: por meio de linguagens de transformação descrevendo ambas as direções simultaneamente.

De acordo com Bucaioni (2013), a maioria das transformações, em geral, são Não-Injetivas. Deste modo, algumas dificuldades surgem no uso de transformações unidirecionais, pois o caminho reverso não pode ser obtido automaticamente se não existirem conceitos do fonte para com o alvo e vice-versa. Um outro problema é quando vários elementos do fonte podem mapear para somente um elemento no alvo, fazendo com que o caminho reverso seja de um-para-muitos e deste modo, uma escolha deverá ser feita de alguma forma para que se encontre o elemento fonte desta transformação reversa (HETTEL; LAWLEY; RAYMOND, 2009).

Os trabalhos Schwarz, Ebert e Winter (2010), Winkler e Pilgrim (2010) levantam pontos interessantes na classificação de mecanismos de rastreabilidade como: a especificação de mapeamentos reversos pode ser de modo (i) automático ou (ii) especificado; rastros armazenados (i) gerenciáveis ou (ii) não-gerenciáveis; e no tipo de metamodelo de rastreabilidade como (i) genérico ou (ii) específico.

E finalmente, segundo Oldevik (2005), uma solução de rastreabilidade deveria fazer parte de todo *framework* dirigido por modelos. Assim é a importância de manter e controlar os relacionamentos entre artefatos dentro de uma abordagem dirigida por modelos, o que ainda apresenta-se como desafio nos dias de hoje.

2.3 Sistemas de edição

Sistemas de edição de textos são utilizados para criar e modificar arquivos de texto contendo alguma sequência de caracteres. Um sistema de edição de textos tem a função de prover o usuário com primitivas de edição (BIEBER; ISAKOWITZ, 1989) - inserção, deleção e recuperação - que operam sobre um arquivo virtual que pode ser manipulado de varias maneiras.

Basicamente editores podem ser classificados em 2 (dois) grupos:

1. Editores de texto plano: Livres quanto ao seu formato estrutural;
2. Editores estruturados: Possuem um modelo subjacente que dá estrutura ao seu conteúdo.

Editores básicos de texto plano ainda cumprem funções importantes, dependendo da necessidade e do tipo de arquivo a ser editado, *e.g.*, para o preenchimento de um simples campo de um formulário em uma página *Web* (uma edição leve dentro de um navegador *Web*), ou ainda, um arquivo de configuração de uma aplicação (uma edição de valores de parâmetros em um arquivo de texto puro). Sistemas operacionais, por convenção e conveniência, são distribuídos geralmente com alguma aplicação de edição de texto puro (GOLDBERG, 1982).

Do ponto de vista do sistema operacional, e levando em consideração que cada plataforma possui a sua forma de armazenar um arquivo, uma diferenciação geralmente é efetuada quanto ao tipo de conteúdo de arquivos (FINSETH, 1991):

1. Arquivos de texto: São implementados por um conjunto de caracteres padrão, podendo então serem lidos por diferentes sistemas. As características desse tipo são:
   - Limites de linha são implementados por um carácter especial que divide o arranjo, o *EOL* (Fim de linha); e
   - Caracteres especiais não são impressos. Estes podem ser de formatação, *e.g.*, o Tab (tabulação), ou de controle, *e.g.*, o *EOF* (Fim de arquivo).
2. Arquivos binários: Geralmente os arquivos que não são arquivos de texto são considerados arquivos binários. Sua transferência para o armazenamento deve ser integral, *i.e.*, sem qualquer tratamento. Um arquivo binário possui as seguintes particularidades:
   - Linhas podem conter qualquer carácter e podem ter qualquer tamanho; e
   - Arquivos podem não ser divididos por linhas.

Uma edição é implementada por meio da abstração de um arquivo virtual em memória, onde modificações e formatações podem ser efetuadas sem os limites impostos pelo arquivo em disco (FINSETH, 1991). Assim, um tipo de dados abstrato (*ADT – Abstract Data Type*) deve ser empregado para manter o texto em memória e também para, oferecer seu conteúdo na forma de objetos atômicos (GOLDBERG, 1982). Esses objetos atômicos podem, por brevidade, ser chamados de caracteres.

Os editores de texto plano, que podem ser simplesmente implementados por arranjos de *bytes*, possuem um esquema simplificado de caracteres agrupados em palavras, que estão agrupadas em linhas, e estas agrupadas em documentos (GOLDBERG, 1982). Então um editor oferece um conjunto especializado de serviços para manipular esse esquema, *e.g.*, recuperar e armazenar o documento, modificar o conteúdo, imprimir, buscar etc. Assim, o processo de edição acontece modificando elementos ou propriedades do conteúdo de documentos ou mesmo propriedades do próprio documento.

A seguir, uma terminologia auxiliar para editores que pode ser utilizada como referência aos conceitos utilizados em edição de textos (STALLMAN, 2000; BURKHART; NIEVERGELT, 1980):

- Janela lógica (também conhecido como "*buffer*", "arquivo virtual"): Permite que uma unidade básica de texto seja editada na memória;
- Ponteiro do cursor (também conhecido como "ponto"): Todas as operações que mudam o conteúdo de uma janela lógica ocorrem em uma posição referenciada por um ponteiro;
- Marcas: Outras posições do texto que podem ser definidas. São usadas para referências futuras, *e.g.*, a última posição do cursor, ou como limites de interação, *e.g.*, selecionando um espaço dentro do texto para a colagem de algum outro texto;
- Região: Um determinado bloco de texto entre uma marca a posição atual do cursor;
- Modos: O editor pode ter somente um modo ("*modeless*"), ou possuir outros modos além do modo de edição, *e.g.*, o editor *vi* (LAMB, 1998) que possui os modos de inserção e comando;
- *Display*: Qualquer hardware operado pelo usuário. Um *display* tem um teclado, uma tela ou talvez até algum outro dispositivo de entrada;
- Tela: Componente do *display* onde é visualizada a saída; e
- Janela: A tela lógica.

No processo de edição, dependendo do tipo de documento editado, pode ser necessário o armazenamento de informações extras, *e.g.*, formatações de linha, parágrafo,

página, margens, centralização, e ainda, objetos não textuais (mapas de *bits* ou objetos embutidos que precisam de informações extra para a sua correta visualização). O armazenamento destas informações pode ser feito de duas maneiras (FINSETH, 1991):

- *In-Band*: Visíveis para o usuário no próprio texto, *e.g.*, caracteres de formatação do *LaTeX* (LAMPORT, 1986); ou
- *Out-of-Band*: Controle e formatação são armazenados dentro de estruturas paralelas por meio de referências externas.

Decompondo um editor genérico em módulos, de acordo com as diferentes funções executadas, poderíamos ter então (FINSETH, 1991):

1. *Editor interno*: Também conhecido como "sub-editor", "*decoder*". Sua tarefa é manter a janela lógica. É composto de:
   a) *Interface de procedimentos*: Implementação das operações do editor. Operações globais de inicializar e finalizar o editor, e salvar e carregar arquivos. E ainda para o documento em edição, manter uma janela lógica, a posição de edição, as marcas e efetuar buscas, substituições etc;
   b) *Estruturas de dados internas*: Descritores são utilizados para manter as propriedades destas estruturas retendo informações como, o conteúdo do texto, lista de marcas, posição do cursor, flag de modificação (*dirty flag*), entre outros.
2. *Formatador*: Também conhecido como "impressor", "reformatador" ou "*redisplay*". Geralmente possui duas funções:
   a) *Reformatação da janela lógica para a saída*: Todas as mudanças na janela lógica devem ser refletidas prontamente para o *display*. O envio das informações pode ser total ou incremental;
   b) *Gerenciamento de janelas e telas*: Que é composto do controle do *refresh* da janela, gerenciar os diferentes estados entre o editor e a janela, e ainda, interpretar os caracteres de controle e de formatação.
3. *Interface de comandos*: É o módulo de entrada dos comandos do usuário. Este módulo implementa o conjunto de comandos disponíveis, recebe estes comandos e determina as operações a serem executadas pelo editor interno.

A necessidade de edição especializada motivou o surgimento de editores que pudessem lidar com dados estruturados. Assim, editores estruturados foram criados ainda antes

da abordagem *WYSIWYG* [7] (SHNEIDERMAN, 1981; HAMMER et al., 1981; NOVICK et al., 2002) e já ofereciam, em certa proporção, facilidades para uma edição com mais recursos.

Os editores estruturados são guiados por uma sintaxe, sendo assim, possuem um modelo subjacente que age sobre a edição dando estrutura e formato ao texto. Esta abordagem permite que um editor vá além da simples edição plana de textos e permita manipular tipos básicos de representação como a textual, a estruturada e a gráfica de maneira sistemática.

Alguns tipos de editores estruturados são, *e.g.*:

- Processador de palavras: Possui seu foco em estruturas para unidades léxicas;
- Editor *XML*: Permite configurações do modelo e de visualização;
- Projecionais: Com o crescimento na adoção de ferramentas de suporte ao trabalho com linguagens específicas de domínio, também conhecidas como "*Language Workbenches*" (FOWLER, 2005), *DSL (Domain Specific Languages)* (FOWLER, 2010) e "*Grammarware*" (WIMMER; KRAMLER, 2006), editores projecionais especializados vem sendo criados para lidar com essas linguagens.

A abordagem *DSL* mostra-se uma técnica muito útil dentro da Engenharia de *Software* e vem sendo muito estudada. Basicamente para se criar uma *DSL* é necessário uma sintaxe abstrata, uma sintaxe concreta e a definição de uma semântica para ela (FEILKAS, 2006). A partir disso, mecanismos derivam um editor para essa linguagem, e deste modo, temos uma ferramenta para programar em pouco tempo.

Este trabalho teve seu foco em um tipo de editor estruturado de códigos (também conhecido como "editor de programas" ou "editor de linguagem de programação"). Esse tipo de editor é utilizado para a escrita de códigos em alguma linguagem de programação, com uso em algum contexto de programação e que pode ou não fazer parte de um ambiente de programação maior, *e.g.*, com execução, depuração assistida etc.

A Figura 9, ilustra uma classificação de artefatos que instanciam alguma linguagem de programação, quanto aos tipos básicos de representação e o nível de detalhamento em anotações nestes artefatos. Deste modo, podemos observar as diferenças na estrutura destes artefatos e o tipo de edição especializada que um editor para este artefato precisa.

Nos editores que possuem a sintaxe da linguagem de programação como modelo interno, enquanto manipula-se o código, uma estrutura interna vai sendo criada paralelamente descrevendo a forma e a estrutura deste código. Esta estrutura interna criada é

[7] *What You See Is What You Get* baseia-se no conceito e nos princípios da manipulação direta de elementos da *interface* gráfica.

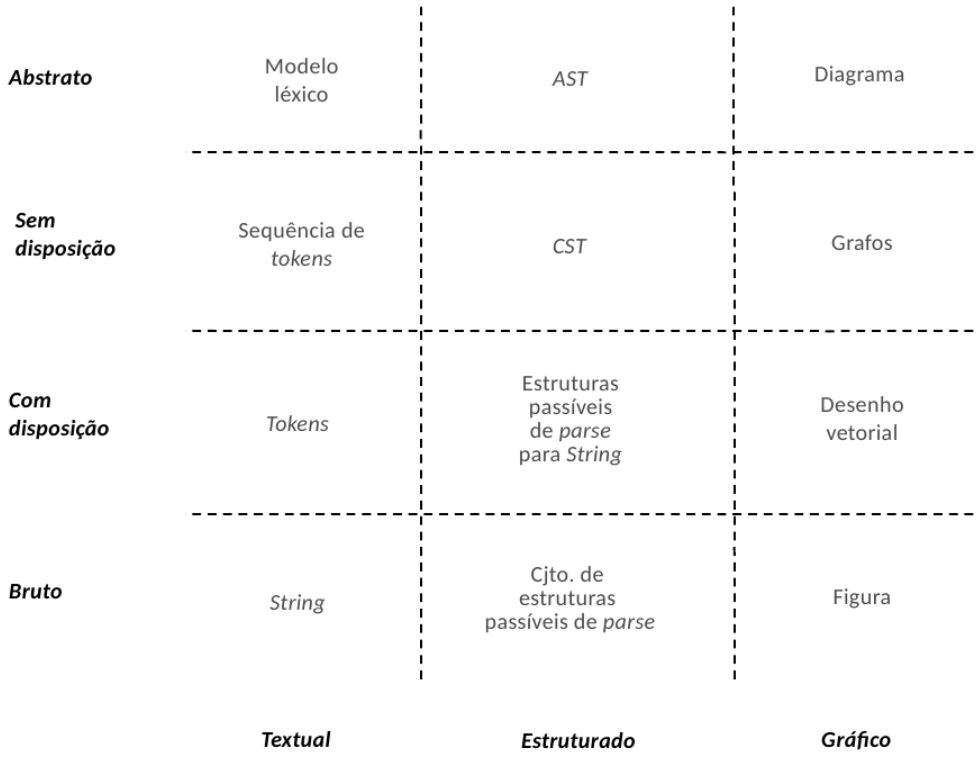

Figura 9: Artefatos classificados de acordo com o tipo de representação dos dados e o detalhamento de suas anotaçõese. Extraído e adaptado de Zaytsev e Bagge (2014).

em forma de árvore e é chamada de árvore de sintaxe abstrata (*AST - Abstract Syntax Tree*).

Assim sendo, a partir de um código sintaticamente correto e através do processo de análise sintática (por meio de um *Parser*), uma *AST* é obtida. Após execução da checagem de tipos (por meio de um *type checker*), erros de tipos na edição são apontados. O processo inverso também pode ser obtido, indo na direção da *AST* para o código (por meio de um *unparser* ou *pretty-printer*) (KOORN; BAKKER, 1993).

2.4 O mecanismo de *Templates JET (Java Emitter Templates)*

O mecanismo *JET* (POPMA, 2004a) é parte da *EMF* e seu uso típico é na implementação de Geradores de Códigos (EMF, 2013). Mais especificamente, é um mecanismo de transformação para gerar artefatos de plataforma a partir de modelos.

O *JET* utiliza a tecnologia de *Templates* para a geração de artefatos alvo que podem ter os mais variados formatos. Sua estrutura é muito semelhante à tecnologia *JSP* da linguagem *Java*. Sua linguagem de programação é composta de expressões do tipo diretivas, expressões de código e *scriptlets*.

O processo de execução do *JET* dá-se em duas etapas: A tradução e a geração.

A Figura 10 demonstra o processo de geração de códigos por meio do mecanismo

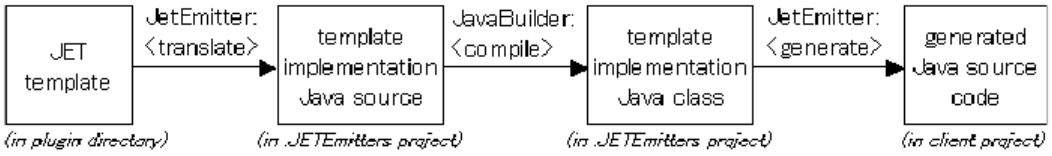

Figura 10: Processo de execução efetuado pelo mecanismo *JET*. Extraído de Popma (2004a).

JET. Arquivos com extensão ".jet" são criados e o *JETEmitter* os traduz para classes *Java* de implementação. Essas classes de implementação *Java* já compiladas serão então as entradas para o processo de geração de código pelo *JET* na criação do código fonte destino.

As vantagens do *JET* são (POPMA, 2004a; POPMA, 2004b):

- É utilizado dentro do *Java*;
- Possui uma linguagem expansível por meio da biblioteca de *tags TAGLIB*;
- É versátil na personalização dos modelos para a entrada de dados;
- É versátil na formatação da saída de dados. Vários tipos de arquivos podem ser gerados;
- É versátil na implementação de esqueletos *Java* nos *templates*; e
- Possui código aberto.

Deste modo, a programação de *templates* fica facilitada tornando o desenvolvimento de Geradores de Códigos menos complexo, mas ao mesmo tempo aumentando a flexibilidade destes na implementação de uma gama maior de soluções possíveis. Apesar de todas as facilidades deste mecanismo, o Projeto *JET* [8] vem perdendo espaço para outros mecanismos de geração de códigos, devido à falta de atividade do projeto para com novas versões.

2.5 A *IDE (Integrated Development Environment) Eclipse*

O Projeto *Eclipse* foi escolhido como plataforma de desenvolvimento deste trabalho, em primeiro lugar por sua natureza extensível, em segundo por oferecer várias facilidades já embutidas e terceiro por ser uma *IDE* bem madura. Sendo assim, o editor especializado foi implementado por meio do *Eclipse PDE - Plug-in Development Environment* (PDE, 2014), que por meio da extensão de componentes internos do Eclipse, possibilitou o desenvolvimento de um editor lado a lado.

[8] A página *Web* do Projeto *JET* pode ser encontrada no *link* `http://projects.eclipse.org/projects/modeling.m2t.jet`

O *Eclipse* oferece os conceitos de Ponto de Extensão e de Extensão. As Extensões para o *Eclipse* são criadas na forma de *plug-ins*, e estes acessam componentes internos da plataforma que também são *plug-ins*.

Um *plug-in* é uma abstração que agrupa código em uma unidade modular, extensível e compartilhável (PDE, 2014). Este componente quando estende algum outro componente, também oferece pontos de extensão para poder do mesmo modo ser estendido.

Um esquema da arquitetura da Plataforma *Eclipse* pode ser vista na Figura 11. As novas ferramentas desenvolvidas para esta plataforma podem estender componentes internos da plataforma e assim especializar estes componentes para a implementação de funcionalidades da nova ferramenta, *e.g.*, para a criação de uma nova *IDE* para uma determinada linguagem de programação nos mesmos moldes da *IDE Eclipse JDT* [9].

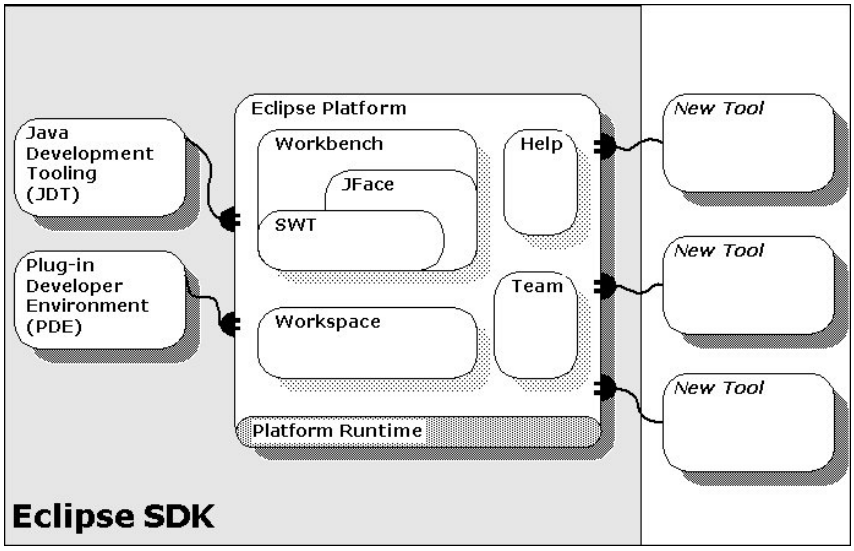

Figura 11: A arquitetura da Plataforma *Eclipse*.(PDE, 2014)

Em um *plug-in* básico da plataforma denominado "*UI*", podemos encontrar os seguintes componentes para o interfaceamento com o usuário (DESRIVIÈRES; WIEGAND, 2004):

- *Workbenches*: Fornecem a estrutura para a interação do usuário e as ferramentas da plataforma *Eclipse*;
- *Perspectives*: São seleções e arranjos de partes de uma *Workbench*, *i.e.*, Editores e Visões;
- *Editor*: São partes de uma *Workbench* especializadas em edição. É um componente específico de edição e sua função é criar e modificar recursos (arquivos); e

[9] *Eclipse JDT IDE* (*Java Development Tools*) é a *IDE* específica para a linguagem *Java* do Projeto *Eclipse*.

- *Views*: São partes de uma *Workbench* especializadas em visualização. Apresentam informações sobre recursos nos quais o usuário está trabalhando.

Assim, um *Workbench* pode conter diversos *Perspectives* e estes, vários *Editors* e *Views*. Outros elementos que poderiam ser adicionados para um *Perspective*, *e.g.*, o *Outline View*, o *Navigator View* e também o *Task View*.

O *Editor plug-in* é o principal mecanismo de edição, é um componente versátil e pode ser utilizado para editar qualquer tipo de texto. Uma implementação deste *plug-in* pode tanto ser feita visando um simples mecanismo de edição de texto simples não estruturado, ou até mesmo um editor estruturado multipaginado, *e.g.*, o editor do arquivo *Manifest* de projetos do *Eclipse JDT* é uma implementação específica do *Editor plug-in*.

A Figura 12 mostra o *Eclipse IDE* e sua área de trabalho ativa (*Workbench*) em um determinado arranjo (*Perspective*) composto por três *Views* e um *Editor* que estão ativos.

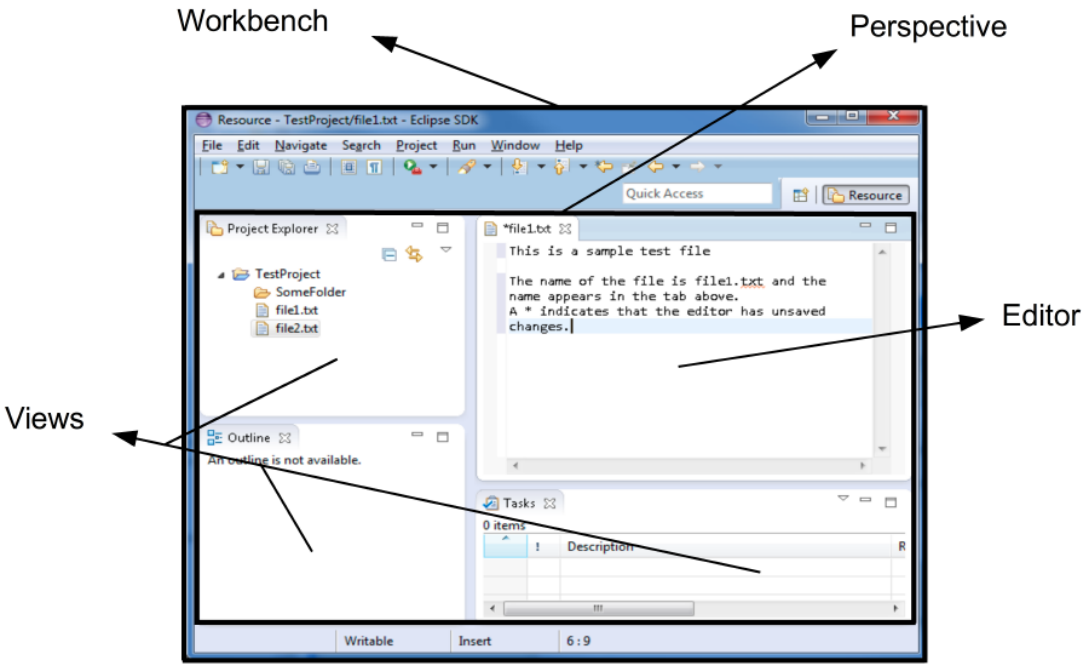

Figura 12: A *IDE Eclipse* e seus componentes básicos de *interface* com o usuário.

O Projeto *Eclipse* revelou muitas vantagens para o desenvolvimento do protótipo advindas do reuso de seus componentes internos, aliviando assim a tarefa de implementar funções básicas de um editor de textos, *e.g.*, o *dirty flag* de um arquivo em edição que sinaliza se o estado de seu conteúdo mudou.

O componente de *framework Editor* ainda apresenta várias funcionalidades de edição, além das operações padrão de escrever, apagar, cortar, copiar, colar, buscar e substituir (PDE, 2014), que estão listadas a seguir:

- Suporte para menus de contexto e menus suspensos;
- Apresentação de anotações nas guias laterais;
- Atualização automática de anotações;
- Informação adicional como números de linhas do texto;
- Destaque com sintaxe colorida;
- Assistente de conteúdo;
- Visualização da estrutura hierárquico de um texto;
- Comportamento sensível ao contexto;
- Suporte para dicas sobre o texto, também conhecidas como *tooltips*;
- Associação de teclas de atalho; e
- Configuração particular de preferências.

2.6 Considerações finais

A mudança de paradigmas trazida pela *MDE* trouxe também novos desafios, daí a necessidade de mais ferramentas, métodos, adaptar processos etc.

Os sistemas de edição são fundamentais e essencias. E são muitas as possibilidades de emprego destes. Seu uso em um ambiente sincronizado requer que sua implementação deva ser cautelosa, envolvendo conceitos de diversas áreas dentro da Engenharia de *Software*, o que eleva assim a complexidade desta tarefa. Por outro lado, e por sobre o ombro de gigantes, como o Projeto *Eclipse* e o versátil mecanismo *JET*, apenas para citar alguns dos que constam neste trabalho, podemos vislumbrar soluções de problemas - ou mesmo partes destes - de uma maneira fundamentada.

3 Trabalhos relacionados

3.1 Abordagens de sincronização

Após revisão da literatura, foram encontradas diferentes abordagens para a sincronização e manutenção da consistência de informações entre artefatos.

3.1.1 A abordagem *GRoundTram*

A abordagem *GRoundTram - Graph Round TRAnsforMation* Hidaka et al. (2011) é uma evolução de sua abordagem anterior Hidaka et al. (2010), que encontra-se referenciada no trabalho de Possatto (2014). Seu novo trabalho é um *framework* bidirecional que vem acompanhado de uma linguagem de transformações mais amigável e um ferramental especializado.

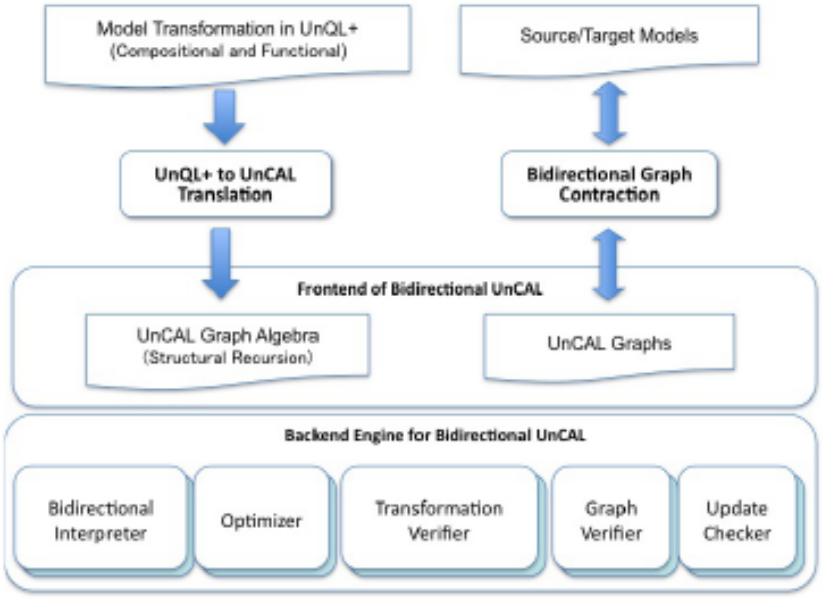

Figura 13: Visão geral da abordagem *GroundTRam*. Extraído de Hidaka et al. (2011).

Uma visão geral da abordagem encontra-se na Figura 13. O comportamento bidirecional de suas transformações é garantido pelo algoritmo de contração de grafos, que é baseado na álgebra de consultas *UnCAL* que é completamente bidirecionalizável. A *GRoundTram* foi projetada para ser composicional podendo reusar e compor transformações existentes. O metamodelo do *framework* é baseado no *KM3* [1] (JOUAULT; BÉZIVIN, 2006) e seus modelos internamente são representados por grafos.

[1] *Kernel Meta-Metamodel* é uma *DSL* para a especificação de metamodelos criada pelo *INRIA (Institut Nacional de Reserche en Informatique et Automatique (FR))*.

Suas transformações são escritas usando-se *UnQL+*[2], uma linguagem puramente funcional e similar a *SQL*. De posse da transformação, o código *UnQL+* é traduzido para a linguagem *UnCAL* [3]. Uma *IDE* específica foi criada para a *GRoundTram* contendo uma ferramenta para validar modelos e transformações, um mecanismo de otimização e um depurador para testar o comportamento bidirecional. As validações são obtidas pelo esquema e por meios de rastros de transformações.

A dinâmica do processo é descrita como: as entradas são um modelo e o seu esquema, além de uma transformação *UnQL+* e um esquema alvo. Deste modo o modelo alvo então é criado. A Figura 14 descreve esse processo.

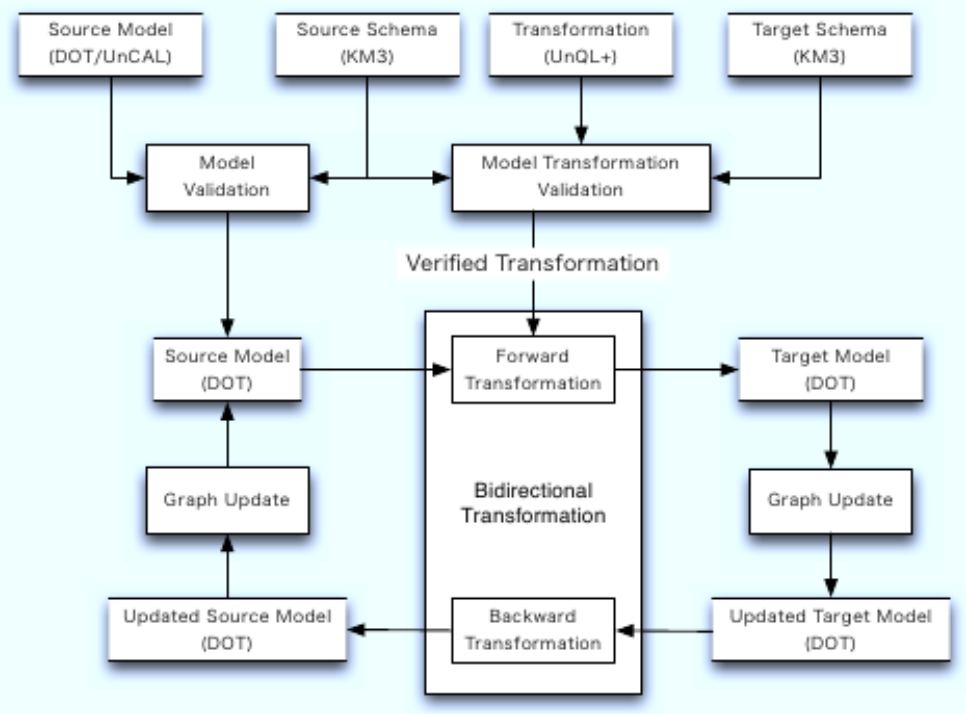

Figura 14: A dinâmica do processo da *GRoundTRam*. Extraído de Hidaka et al. (2011).

Em sua *UI*, após carregar dois modelos e uma transformação, uma transformação pode ser efetuada clicando-se no botão *"forward"*. A depuração proposta é obtida por meio de informações do rastro da transformação. Existem limitações, pois apenas transformações *M2M* são suportadas.

3.1.2 A linguagem de transformação *JTL (Janus Transformation Language)*

A linguagem de transformação *JTL (Janus Transformation Language)* (CICCHETTI; RUSCIO, 2011), é uma linguagem bidirecional especificamente projetada para

[2] *UnQL+* é o acrônimo de *Unstructured Query Language +*.

[3] *UnCAL* é o acrônimo de *Unstructured CALculus* e é um subconjunto da *UnQL+*. É de baixo nível e pode ser usada também para descrever as transformações da abordagem.

dar suporte às transformações não-injetivas de propagação de mudanças. A *JTL* é baseada na teoria *ASP* [4], que possibilita por meio de transformações, que todos os possíveis artefatos alvo consistentes com um artefato origem sejam gerados.

A linguagem foi implementada por meio de um conjunto de *plug-ins* do *Eclipse*, utilizando o *framework* de modelagem *EMF* (BUDINSKY, 2004) e o *AMMA (Atlas Model Management Architecture)* (BÉZIVIN et al., 2005). A *JTL* segue o paradigma declarativo e é uma variação da linguagem *QVT* [5]. Sua implementação é baseada nos termos da linguagem *ASP*, e as transformações escritas em *JTL* são convertidas para problemas de busca *ASP*. O mecanismo de resolução *ASP* então encontra todos os possíveis modelos alvo estáveis que são consistentes com as regras do programa fonte por meio de um processo dedutivo.

A Figura 15 mostra o esquema utilizado pela abordagem. As transformações *JTL* são convertidas em programas *ASP* utilizando um mapeamento entre estes dois metamodelos. Os programas *ASP* são enviados para o mecanismo *JTL*, integrado a um mecanismo de resolução [6], que irá gerar os modelos de saída em conformância com os seus respectivos metamodelos.

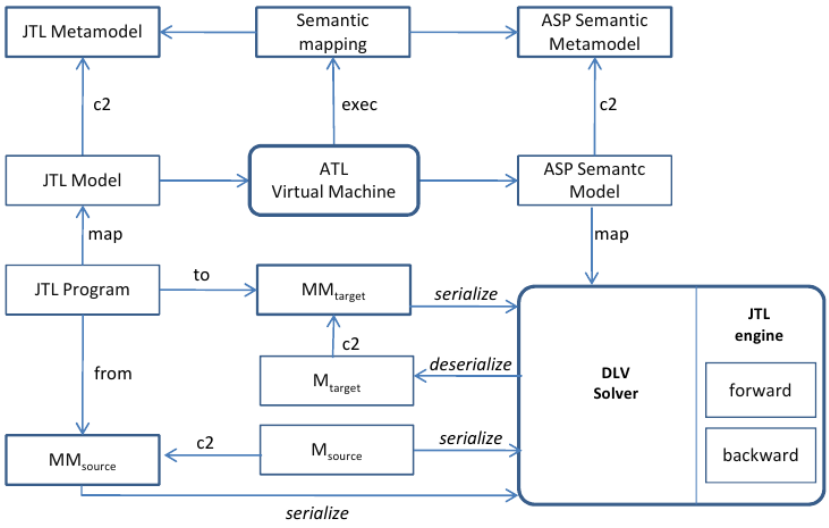

Figura 15: A dinâmica do processo da JTL. Extraído de Cicchetti e Ruscio (2011).

A *JTL* consegue lidar com relacionamentos não-injetivos por meio da exploração de rastros obtidos por suas transformações. A abordagem produz todas as possíveis soluções para a transformação reversa, porém nem sempre isto é o desejado, e restringir as não-injetivas é um modo para que esta não gere um número muito grande de soluções. Por outro lado para um processo de análise de possíveis soluções esta abordagem se torna

[4] *ASP (Answer Set Programming)* é uma forma de programação lógica orientada a problemas de busca *NP-hard*

[5] *QVT (Query/View/Transformation)* é um conjunto de linguagens de transformação do *OMG*.

[6] O *DLV Solver* é uma implementação de um Sistema de Resolução e Representação do Conhecimento.

vantajosa. É uma forma nova de abordar o problema que ainda carece de padronização e ferramentas.

3.1.3 O *Framework PREP*

O *framework* conceitual *PREP (Propagate Replay Evaluate Pick)* (SEIFERT, 2011) foi idealizado para projetos de sistemas de *RTE*. Baseia-se na composição de transformações unidirecionais de ida-e-volta do processo, e foi criada com a ideia de resolver algumas limitações que são encontradas no uso de transformações bidirecionais para sincronização de modelos, *i.e.*, situações onde a transformação inversa não foi especificada ou quando não pode ser criada automaticamente.

A abordagem *PREP* usa a *back-propagation*. Esta estratégia retorna as mudanças para a origem para depois reenviar esta propagação de volta ao alvo. O processo implementa a ideia de múltiplas opções de propagação. As opções que levam a modelos inválidos são descartadas por funções que calculam o grau de similaridade de um modelo para o outro. É um processo de sincronização por estágios checando a consistência global, a transformação escolhida e restaurada é a que produza um *score* mais alto (a que mais reflete a fonte no alvo) dentre as várias outras transformações possíveis.

Sua dinâmica é: Em uma transformação t de a para b, a mudança d gera b'. Isto dispara múltiplas propagações $(p1, p2, ..., pn)$ que irão gerar um conjunto de modelos fonte $(a'1, a'2, ..., a'n)$. Executando t em cada um destes modelos fonte, um conjunto de modelos alvo é gerado $(b"1, b"2, ..., b"n)$. E todos estes artefatos $b"$ passarão por uma função de ajuste que avaliará a possibilidade de aceitação de cada propagação e descartará as inconsistentes (vide Figura 16). Para que o processo seja bem sucedido, um controle de versões de modelos foi implementado para criar um isolamento evitando interferências entre modelos.

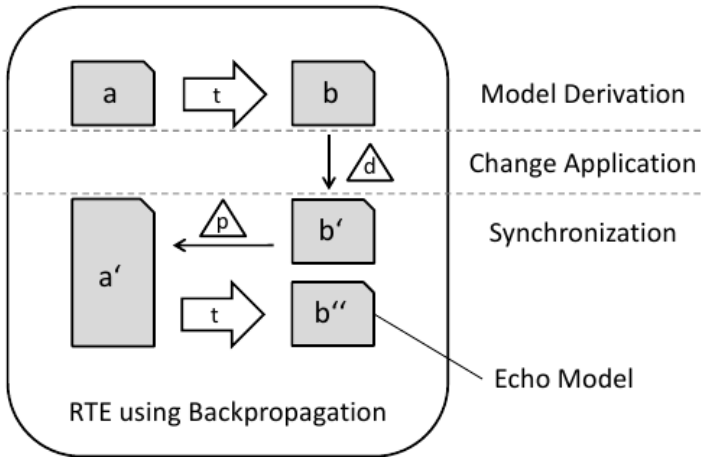

Figura 16: O processo de *back-propagation* utilizado pelo *PREP*. Extraído de Seifert (2011).

As transformações são formadas por um conjunto de quatro operadores que agem

sobre modelos: *ADDITIONS* (para a criação de novos objetos); *UPDATES* (para mudar valores de atributos); *COUPLINGS* (para conectar/desconectar referencias de modelos) e *DELETIONS* (para remoção de elementos). Deste modo uma transformação é decomposta nas suas primitivas de edição sobre elementos de um modelo. O suporte para transformações de vários elementos do fonte para o alvo é garantido pela abordagem e uma função de ajuste de conservação determina o grau em que um conjunto de modelos é similar ao estado prévio do mesmo conjunto. Seus valores são baseados no delta entre os modelos dado por um número entre 0 e 1. Valores altos indicam que *a'* e *b"* conservam muitas das propriedades dos modelos originais. Uma vantagem clara é não haver necessidade de criar a transformação reversa manualmente. A abordagem foi utilizada em um cenário que é similar à este trabalho. A Figura 17 mostra essa similaridade.

A *PREP* pode ser aplicada com a pré-condição que os artefatos envolvidos (Modelo, *Template* e CG) sejam modelos, que seus metamodelos estejam disponíveis e que existam meios de instanciar modelos concretos. O que não puder ser abstraído para modelos por meio da conversão entre sintaxe concreta para abstrata (criação de metamodelos), pode ser obtido por meio de uma ponte entre gramática livre de contexto e linguagem orientada a objetos. Uma vez que todos os artefatos envolvidos possam ser representados por modelos, o processo de geração de código pode ser passado para o nível de modelagem. O rastro da execução é obtido no processo de transformação, e para cada elemento do modelo a saída para o código gerado é conhecida, isto acarreta em informações de rastro bem detalhadas.

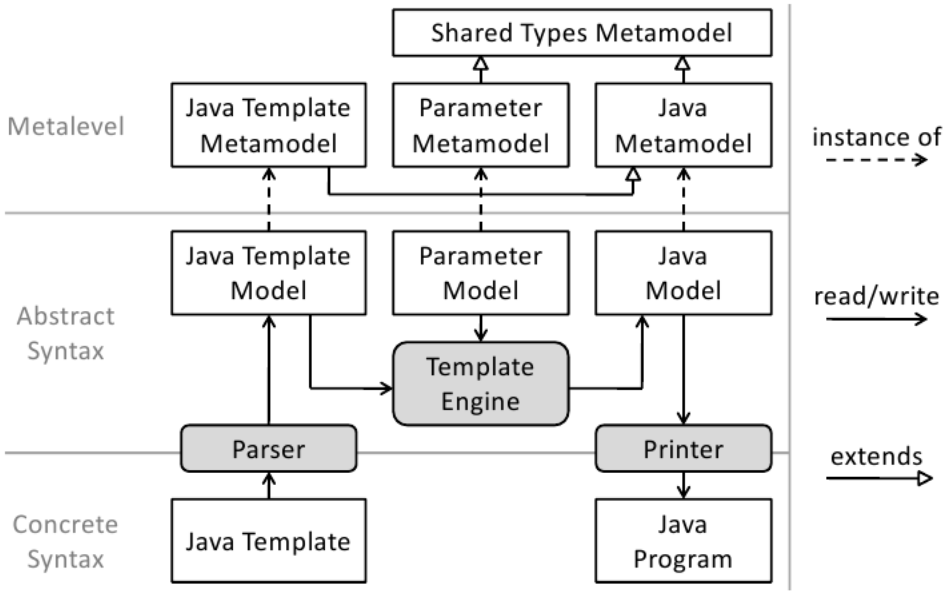

Figura 17: Aplicação do *PREP* em um ambiente de geração de códigos por meio de *Templates*. Extraído de Seifert (2011).

Existem algumas limitações de implementação, *e.g.*, a linguagem de transformação dos *templates* que foi utilizada é muito restrita em sua expressividade e problemas de

aninhamento de meta elementos que em uma certa profundidade terminam por gerar um número alto de escolhas válidas de propagação inibindo a automatização. Mas apesar das limitações, esta abordagem mostra-se interessante em vários a aspectos. O principal é que trabalha em nível de modelos sobre todos os artefatos do processo aplicando os preceitos da *MDE* integralmente na geração de código. Um outro aspecto é que em transformações que levam em conta a sintaxe abstrata, modelos são utilizados para fonte e alvo assegurando então independência quanto a representações concretas e assim distanciando e desacoplando do nível de implementação.

3.2 Abordagens de edição de *templates*

Geralmente um editor de *templates* é fornecido pelo *framework* de geração de códigos, como é o caso do mecanismo *JET* (POPMA, 2004a) e do mecanismo *Acceleo* (ACCELEO, 2014). Mas os editores limitam-se somente a edição do código fonte do *template* oferecendo um ambiente básico de edição.

3.2.1 *XMLSpy*

Os editores estruturados *XML* vem sendo utilizados já a algum tempo para a formatação, visualização e apresentação de diversos documentos. Um exemplo é o *XMLSpy* (KIM, 2002). Esses editores podem ser utilizados no contexto da edição de *templates*, mas para isso, devem diferir dos editores *XML* comuns permitindo uma edição que não seja restringida pelo *DTD (Data Type Description)*, possibilitando assim a manipulação do texto e ao mesmo tempo aproveitando-se da estrutura subjacente deste que descreve sua estrutura.

3.2.2 *XEditor*

Um editor estruturado *XML* foi proposto por Hu et al. (HU; MU; TAKEICHI, 2007) e possibilita que o usuário edite sobre uma visão do documento, e automaticamente o editor deriva um documento fonte (da transformação) para este e uma transformação que produz uma visão do documento. Este editor, é baseado em conceitos de *View-updating* [7] (BANCILHON; SPYRATOS, 1981; GREENWALD et al., 2003) e *Lens Combinators* [8] (FOSTER et al., 2005). Seu protótipo possui ainda limitações nas transformações bidirecionais e na derivação do esquema que acabam por restringir as edições e visualizações. Apesar deste editor não ser especificamente um editor de *templates*, estes podem ser editados se o esquema de um *template* for fornecido.

[7] View-updating é um conceito da área de Bancos de Dados que estuda a propagação de alterações efetuadas em *views* de volta ao banco de dados.

[8] O conceito de *Lens Combinators* estuda transformações bidirecionais em dados estruturados.

3.2.3 *Blinkit*

O *Blinkit* [9] (YU et al., 2012) é baseado no *framework GRoundTram* de Hidaka et al. (2011) que foi analisado neste trabalho na SubSeção 3.1.1. A ferramenta de sincronização de artefatos foi implementada sobre o *Eclipse* na forma de *plug-ins*, que mantém e se utiliza de rastros entre modelo e código gerado a partir de transformações bidirecionais. Nesse método, uma vez que o código de um artefato é anotado para ser uma invariante rastreável, as transformações bidirecionais criadas poderão propagar corretamente mudanças em duas direções. Por exemplo, ao renomear um atributo no alvo e anotando este com `@generatedNOT`, a transformação reversa propaga a mudança para o fonte trocando o nome do atributo. A partir deste momento a reaplicação da geração de código, fará com que o código do artefato passe a contar com um atributo com um novo nome e ainda *getters* e *setters* serão modificados de acordo. Internamente, o processo é basicamente composto de cinco passos. Os quatro primeiros passos serão similares após a saída do *framework GRoundTram*, mas executadas inversamente. O processo pode ser visto na Figura 18:

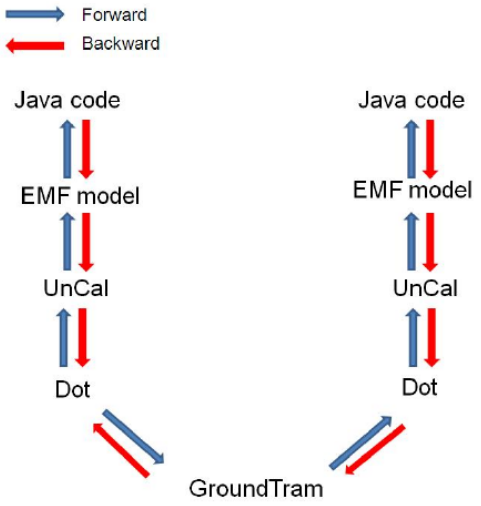

Figura 18: Uma visão geral da proposta *Blinkit*. Extraído de Yu et al. (2012).

A dinâmica do processo é composta pelos seguintes passos:

1. Gerar código a partir do modelo ("*Java code*" na figura);
2. Representar o código por um modelo *EMF* ("*EMF model*" na figura);
3. Traduzir o modelo *EMF* para *UnCAL* ("*Uncal*" na figura);
4. Transformar de *UnCAL* para o formato *DOT* [10] ("*DOT*" na figura); e

[9] O *Blinkit* está disponível para download em http://sead1.open.ac.uk/linkit/
[10] *DOT* é uma linguagem para descrição de grafos usando texto plano

5. Gerar a *UnQL+* para entrada no *GRoundTram*, usando os arquivos *DOT* e obtendo os arquivos *DOT* de saída ("*GRoundTram*" na figura).

Os passos 1 e 3 foram implementados por *grammarware*: *EMFText* e *Xtext* respectivamente. Ao final do processo uma mesclagem é feita entre o código gerado e o código do *template* gerado automaticamente para sincronizar modificações feitas diretamente no código. O *Blinkit* aproveita-se da estrutura de *RTE* do *GRoundTram* e a aplica na rastreabilidade de elementos entre artefatos podendo assim refletir as mudanças de um artefato para o outro.

3.2.4 *Considerações finais*

Pouco material relativo à edição de *Templates* foi encontrado além dos editores básicos disponibilizados por ferramentas de transformação. Portanto, podemos concluir que a edição de *Templates* especializada dentro do processo de geração de códigos é pouco explorada, sendo necessário a realização de mais pesquisas nessa área.

3.3 Abordagens de edição simultânea

Os primeiros editores que permitiam a edição em duas visões surgiram na década de 80 (CHAMBERLIN et al., 1982; SALELLES, 1983; NELSON, 1985; ASENTE, 1987). Não utilizavam os conceitos *WYSYWYG* e alguns somente permitiam a edição de um lado - o lado código e - e uma visualização do resultado da edição do outro lado. Alguns editores se destacam de uma maneira ou de outra neste cenário e são listados a seguir:

3.3.1 O editor *Grif*

O *Grif* (QUINT; VATTON, 1986) foi um editor estruturado que possuía dois tipos de visões: a "*Full text*" e a "*Table of Contents*". As duas podiam ser visualizadas simultaneamente, ambas podiam ser modificadas e eram sincronizadas. Porém, a sincronização somente era permitida para estas duas visões, e ainda, problemas de modularidade da linguagem comprometeram em muito a sua capacidade de extensão (MUNSON, 1994).

3.3.2 O editor *LILAC*

O *LILAC* (BROOKS, 1991) foi concebido como um editor *WYSIWYG* que tinha a tarefa de sincronizar a edição de documentos de código. Ofereceu uma edição limitada, pois restringia a edição de certos elementos, *e.g.*,. constantes não eram selecionáveis e nem editáveis.

3.3.3 O editor *Pan*

O *Pan* (BALLANCE; GRAHAM; VANTER, 1992) foi um sistema de edição baseado em sintaxe e concebido para uso com diferentes linguagens de programação. Combina uma janela de edição de código e uma janela de análise do programa sendo editado. Suas limitações eram grandes na sua janela de análise que possuía a visualização da formatação estruturada (disposição) e a tipográfica do código.

3.3.4 O editor *Proteus*

O Proteus (MUNSON, 1994), um sistema de apresentação adaptável, foi baseado em uma extensão do *GNU EMacs* (STALLMAN, 2000) e utilizando conceitos de editores como o *Pan* e de um editor *TEX* chamado *VORTEX* (CHEN; HARRISON, 1988). Este editor adotou uma abordagem de múltiplas representações, pois foi concebido como aplicação de integração entre diferentes representações de informações dentro do ciclo de desenvolvimento de *software*, *e.g.* a textual e a gráfica.

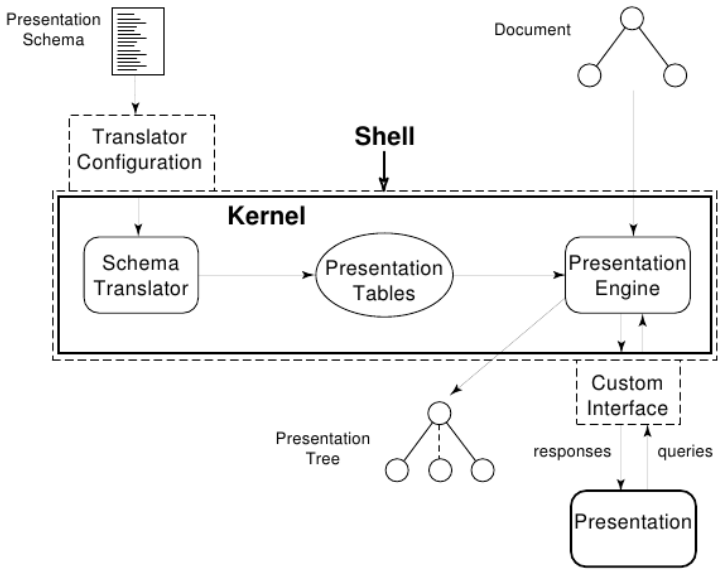

Figura 19: A arquitetura do Proteus, que permitia sua adaptabilidade para diferentes tipos de meios. Extraído de Munson (1994).

3.3.5 *Considerações finais*

A lista de editores é grande, mas muitos desses editores empregaram conceitos *ad-hoc* em sua implementação, que de uma forma ou de outra, em diferentes partes do projeto como na visualização, representação de esquemas, extensibilidade, compatibilidade, propagação de mudanças etc, acabaram por ter a sua implementação da sincronização, modularização ou camada de apresentação comprometidas, gerando um impacto negativo no que diz respeito a possíveis soluções de referência para o problema da edição simultânea.

3.4 A abordagem de migração automática de código de Possatto

O trabalho de Possatto (2014) é um ponto de partida para este trabalho. O protótipo deste trabalho faz o uso de um dos mecanismos utilizado em seu trabalho.

Seu trabalho utiliza a IR como ponto de partida para a construção e manutenção do Gerador de Códigos. Uma vez que o código da IR tenha sido migrado para um *template*, modificações eventuais serão realizadas diretamente neste mesmo código da IR. A ferramenta, focada na migração, então migrará automaticamente estas modificações para os *templates* correspondentes atualizando-os. Com isso, seu método automático de migração verificou uma redução do esforço de migração.

A Figura 20 descreve o processo de seu trabalho. Após mudanças terem sido efetuadas na IR (1), essas são migradas automaticamente de uma só vez para os *templates* correspondentes dentro do Gerador de Códigos (2), e uma nova geração de códigos faz com que as mudanças sejam incorporadas ao código gerado (3).

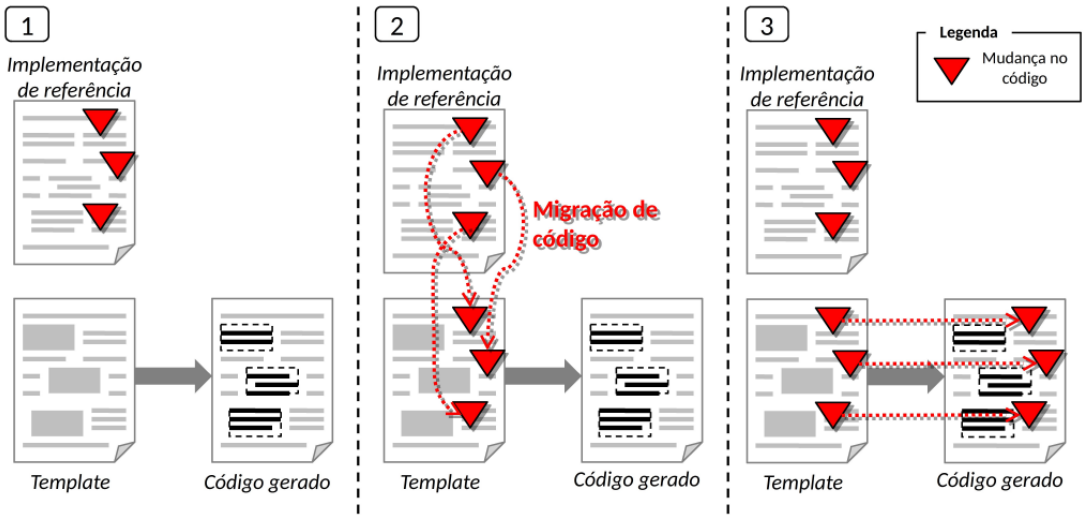

Figura 20: Esquema da migração automática de código. Extraído de Possatto (2014).

Sua ferramenta automática de migração de código faz o uso de duas técnicas para implementar a migração do código da IR para os *templates* do Gerador de Códigos. As seguintes técnicas foram utilizadas:

1. Arquivo de mapeamentos de código: O arquivo de mapeamento de código foi implementado por meio de um mecanismo *JET* Modificado. Esse mecanismo modificado possui internamente, para referência, tipos de mapeamentos que foram previamente definidos. Deste modo, o mecanismo *JET* Modificado, ao efetuar o processo de geração de *templates* reconhece esses tipos de mapeamentos dentro do código e cria fisicamente um arquivo de texto contendo a relação das posições de cada mapeamento identificado (obtidos pela associação das referências aos modelos de entrada em *templates* com o seu código correspondente);

2. Registro de captura de eventos de modificações de código: Esse registro se dá por meio de um *plug-in* para a *IDE Eclipse* que captura eventos de edição e os registra em um arquivo de *log* de alterações. Esse *plug-in*, chamado *Fluorite* (YOON; MYERS, 2011), após ser instalado e configurado, registra eventos de edição de códigos detalhadamente em um arquivo de *log* no formato *XML*. O detalhamento desse arquivo gerado é alto, e contém o texto envolvido na operação e uma descrição desta, seja essa operação de digitar, apagar, copiar, colar, desfazer ou refazer. Para tratar essas informações todas, um *Parser XML* específico foi criado para a recuperação das informações de edição e sua função é recuperar somente as informações de edição necessárias em meio a uma miríade de alterações de edição sobre um arquivo de código.

Com as informações sobre o mapeamento dos caracteres e o registro dos eventos de edição, o cruzamento dessas informações permite então a localização e reprodução das modificações nas posições exatas do arquivo de *template*. Deste modo, a replicação das alterações nos *templates* pôde ser obtida de forma automática.

A Figura 21 mostra um esquema da solução. Durante o processo normal de desenvolvimento, a geração de códigos utilizando o mecanismo *JET* Modificado registra mapeamentos encontrados (1); Na edição deste código, o *Fluorite* registra os eventos de edição ocorridos (2); No processo de migração, o *Parser* percorre o arquivo de *log* e busca por eventos relacionados com alterações em arquivos. O *Parser*, ao encontrar uma alteração em determinado artefato destino, faz uma modificação no artefato origem correspondente que está relacionado pelo arquivo de mapeamento (3). Os *templates* agora encontram-se sincronizados com o CG/IR (4).

Neste *Parser* implementado, para cada evento eleito, um determinado *script* é executado dependendo do mapeamento encontrado. Esses *scripts* foram extraídos de um estudo conduzido por Lucrédio e Fortes (2010) e descrevem o tipo da associação encontrada entre o código de um *template* e o código gerado. Os tipos de mapeamento utilizados econtram-se aqui referenciados:

1. Cópia simples;
2. Substituição simples;
3. Substituição indireta;
4. Repetição;
5. Condicional;
6. Inclusão; e
7. Novo arquivo.

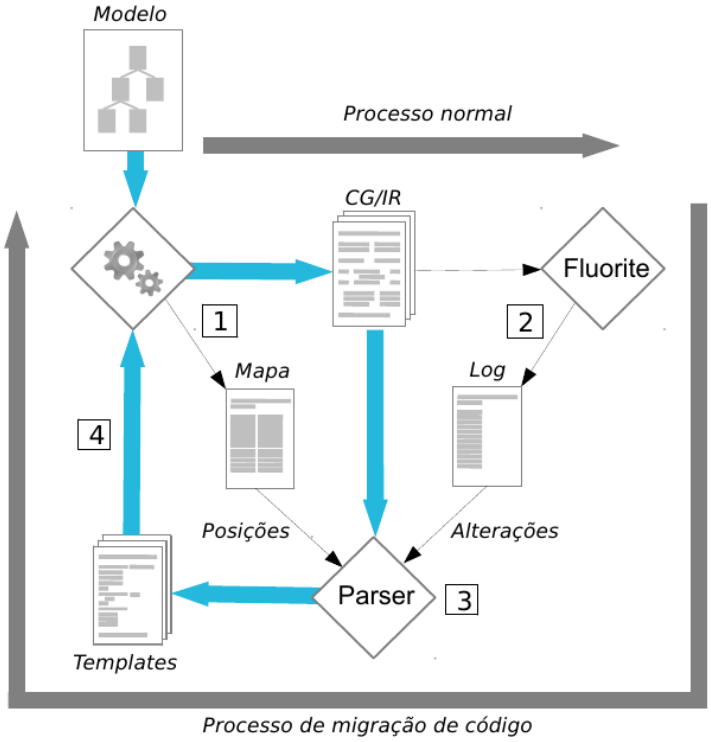

Figura 21: Esquema do trabalho de Possatto. Extraído e adaptado de Possatto (2014).

Com essa ferramenta, *templates* são gerados a partir da IR, e são transferidos em lote para dentro do Gerador de Códigos. Assim, alterações efetuadas na IR e originadas pela manutenção e evolução de *software* são propagadas para dentro do Gerador de Códigos completando assim a migração das mudanças.

3.5 Considerações finais

A edição de artefatos de geração de códigos dentro do processo dirigido por modelos, ainda é carente de mais ferramentas que suportem esse tipo especializado de edição. Temos referências de várias soluções de edição na literatura, mas a complexidade do problema é alta no caso de uma edição de artefatos tendo em vista a necessidade de sincronização destes para com suas réplicas, que precisam de uma forma ou de outra, estar em sincronia. Apesar das mais diferenciadas abordagens, a edição especializada de artefatos dentro do processo dirigido por modelos ainda mostra-se como um desafio.

4 Protótipo de edição simultânea de *Templates* de geração de códigos

4.1 Definição

A ferramenta desenvolvida neste trabalho é um editor simultâneo de *Templates* de geração de códigos. O editor foi projetado e implementado lado a lado para que mudanças feitas de um lado reflitam imediatamente para o outro lado. Deste modo, sua dinâmica permite que modificações no código gerado sejam replicadas para os *templates* de geração de código à medida que as mudanças são efetuadas.

As modificações efetuadas de *templates* para o código gerado, não são tratadas pelo editor, isso porque o próprio processo normal de geração de código, que neste caso é efetuado pelo mecanismo *JET*, já se encarrega desta etapa. Portanto apenas a propagação das mudanças efetuadas no CG é feita pelo mecanismo do editor.

4.2 Implementação

O editor foi implementado utilizando o Projeto *Eclipse*. O Eclipse é um ambiente extensível e de fácil integração. Seu conceito modular baseado em *plug-ins* e pontos de extensão permite o uso e a especificação de seus componentes internos tornando possível a implementação de várias soluções que se integram perfeitamente a *IDE* e até soluções *stand-alone* [1].

Deste modo, o editor foi implementado como um *plug-in* para a *IDE Eclipse*. Sua implementação conta com o uso de:

- Uma *View*: extensão que permite interação com o usuário;
- Tratadores de eventos de edição de texto: detectam mudanças em editores de texto ativos, e realizam alguma ação;
- Tratadores de eventos de navegação no texto: detectam a posição do cursor ou mouse em editores de textos ativos, e realizam alguma ação; e
- Formatadores de texto: aplicam estilos ou cores em editores de textos ativos, oferecendo um destaque visual.

[1] Aplicações *stand-alone* podem ser implementadas dentro do *Eclipse* na forma de clientes *RCP (Rich Client Platform)*, que é uma forma de se fazer *plug-ins* de um jeito que estes não acessem certos componentes *UI* do Projeto *Eclipse* como o *plug-in Workspace* da *IDE*

A implementação das características acima citadas e, a integração destas com a *IDE*, tornam possível a edição do CG/IR e a replicação destas mudanças para os *templates* de geração de códigos.

O mecanismo interno do editor está baseado nos mapeamentos gerados pelo mecanismo *JET* Modificado, também utilizado no trabalho de Possatto (2014). O editor espera por eventos de edição, e quando esses acontecem são reconhecidos e tratados de acordo utilizando as informações do arquivo de mapeamentos. Deste modo, o editor com as informações sobre o trecho atual de edição, e também da operação de edição sendo efetuada no arquivo CG aberto, pode produzir uma modificação na posição mapeada do arquivo de *template* aberto. A Figura 22 apresenta o ciclo das modificações e o uso do arquivo de mapeamentos no processo.

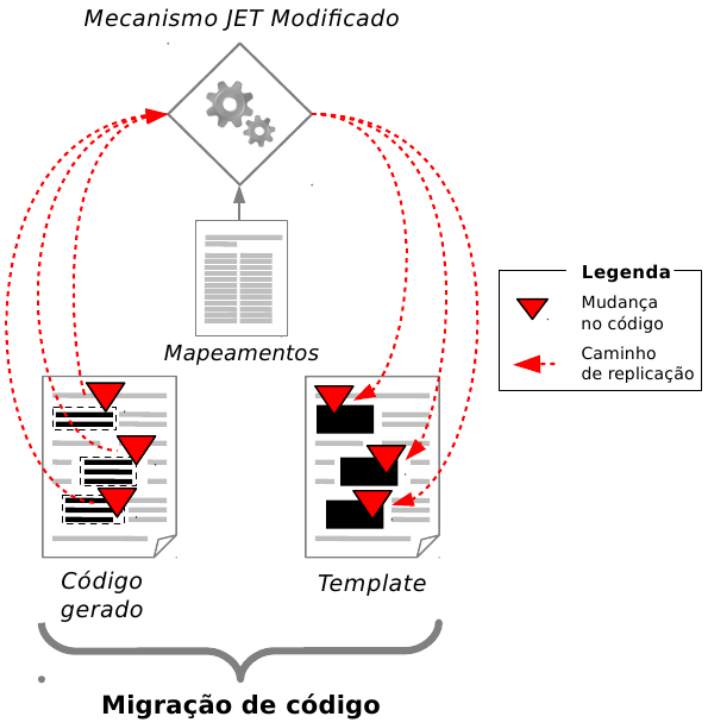

Figura 22: Esquema da edição de artefatos e o seu mecanismo associado.

O arquivo de mapeamento, que pode ser visto na Figura 23, é sempre criado no processo de geração de códigos pelo mecanismo *JET* Modificado paralelamente ao processo de geração, contendo a relação das posições de cada mapeamento, obtido da associação dos arquivos de *templates* com os correspondentes códigos gerados. No esquema deste arquivo cada linha representa um mapeamento diferente, e dentro de cada linha os dados do mapeamenteo estão dispostos sequencialmente separado por meio do carácter especial ":". Uma amostra de um arquivo de mapeamento pode ser visto na Figura 23, onde podemos ver as informações sobre o *template* (1), as informações sobre o código gerado (2) e as informações sobre o mapeamento (3). Cada mapeamento possui as seguintes informações listadas a seguir.

- Nome do arquivo de *template*;
- *Offset* inicial do arquivo de *template*;
- *Offset* final do arquivo de *template*;
- Nome do arquivo de código gerado;
- *Offset* inicial do arquivo de código gerado;
- *Offset* final do arquivo de código gerado;
- Informações da origem do código produzido: podem ser de outro *template*, de uma variável ou da execução de um *scriptlet*; e
- Tipo de mapeamento: podem ser de 7 tipos diferentes conforme listado em 3.4.

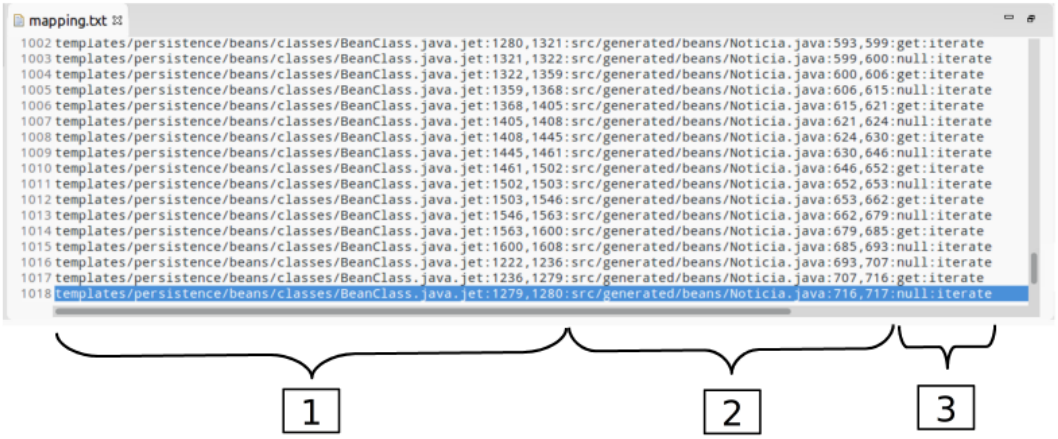

Figura 23: Conteúdo de um arquivo de mapeamentos.

Para cada nova execução do mecanismo *JET* este arquivo de mapeamento de código é refeito. Portanto as associações atualizadas são sempre escritas por sobre as anteriores.

A *interface* do editor, que pode ser vista na Figura 24, possui 3 áreas principais:

1. Lado esquerdo da edição, que edita o código gerado;
2. Lado direito da edição, que edita o *template* de geração de códigos; e
3. A visão, denominada "*Split JET Editor*", que possibilita a configuração da ferramenta e a visualização dos relacionamentos entre os arquivos de edição.

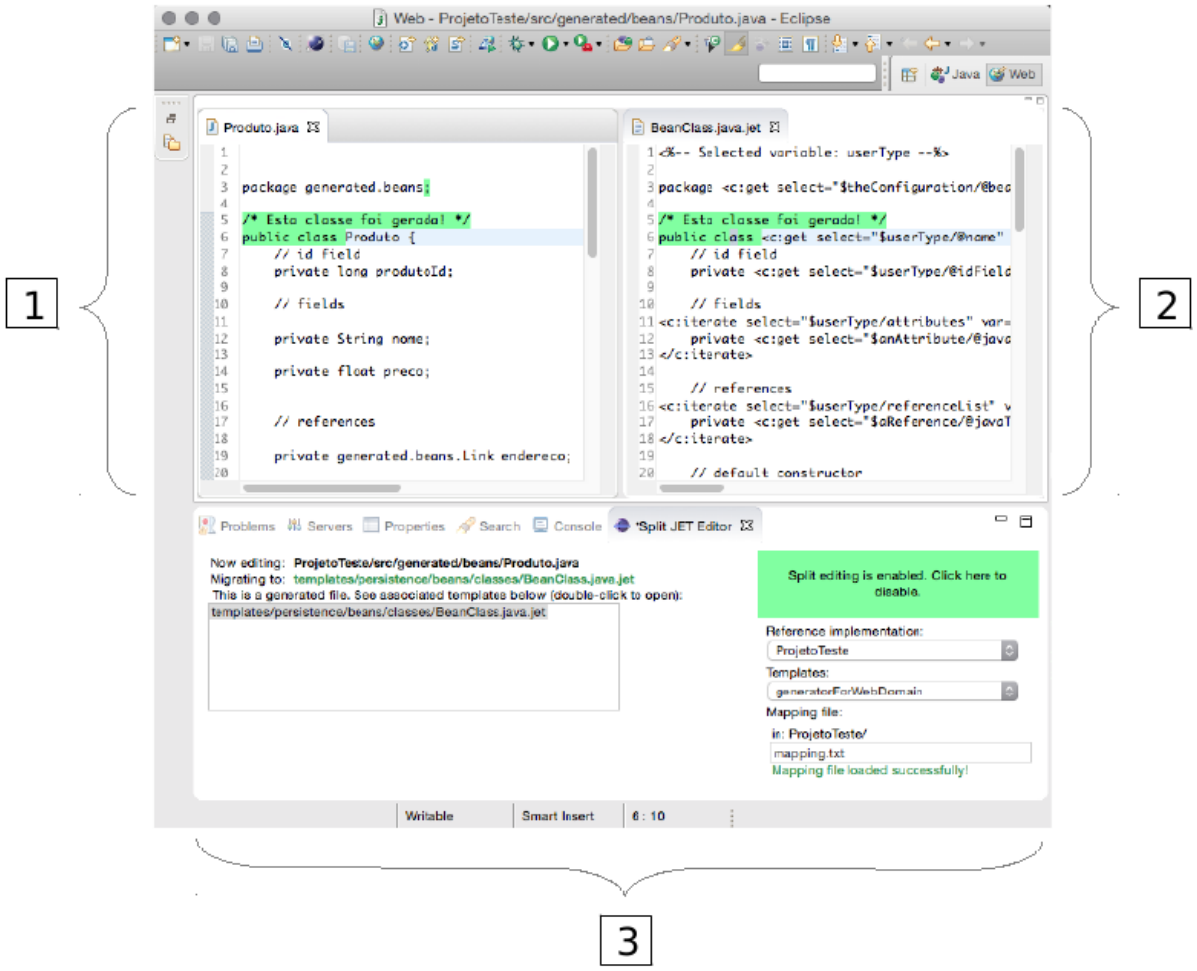

Figura 24: Tela do editor em operação.

A configuração da ferramenta pela janela "*Split JET Editor*", possui quatro itens para visualização dos relacionamentos entre os arquivos de edição. A Figura 25.a, mostra o detalhe do lado esquerdo desta visão, que possui os seguintes itens:

1. Arquivo origem da edição atual, usando o rótulo "*Now editing*" na tela;
2. Arquivo destino da edição atual, usando o rótulo "*Migrating to*" na tela;
3. Área para mensagens e avisos; e
4. Uma caixa combinada utilizada para os listar os arquivos que se relacionam com o atual sendo editado na forma de um-para-muitos, *e.g.*, editando *templates* este controle lista os arquivos de código gerado que são o resultado deste *template*.

No detalhe da Figura 25.b, localizado à direita, a mesma visão possui mais quatro itens para configuração.

1. Botão de seleção ativado/desativado: O modo de edição pode ser ativado e desativado a qualquer momento. Caso a edição esteja ligada, o botão torna-se verde, indicando

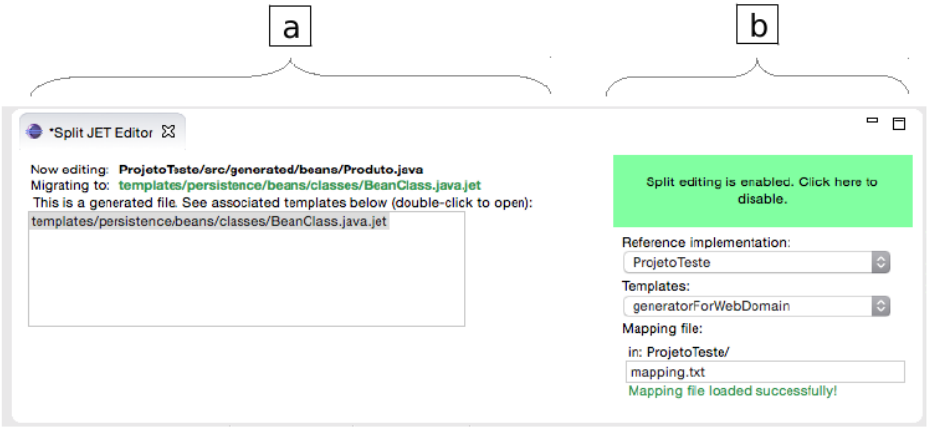

Figura 25: Detalhe da figura anterior com foco na janela "*Split JET Editor*".

que as ações de edição estão sendo tratadas e serão replicadas no arquivo destino. Caso a edição esteja desligada, o botão torna-se vermelho, e o editor opera sem replicação alguma;

2. Lista de seleção da IR, que usa o rótulo "*Reference implementation*" na tela: permite que o usuário informe qual projeto da *IDE* será a IR onde as mudanças serão efetuadas (origem da alteração);

3. Lista de seleção de *templates*, que usa o rótulo "*Templates*" na tela: e permite que o usuário informe qual projeto da *IDE* contém os *templates* para onde as mudanças serão replicadas (destino da alteração);

4. Campo de texto, que usa o rótulo "*Mapping file*" na tela: permite que o usuário informe a localização, dentro do projeto da IR, do arquivo de mapeamento.

Como a edição de trechos de arquivos é baseada no arquivo de mapeamentos, o caso de uma atualização de posições iniciais de um arquivo referenciado dentro do arquivo de mapeamento, faz com que todas as outras posições deste mesmo arquivo tenham que ser alteradas para que o mapeamento continue consistente.

O editor, baseando-se nos mapeamentos, no tratamento de eventos de edição e na manipulação de elementos internos de edição do *Eclipse*, implementa um conjunto de funcionalidades que permitem a replicação de mudanças entre artefatos de geração de códigos.

4.3 Operação

Uma vez com as informações de configuração preenchidas, o editor é capaz de operar tendo o conhecimento para onde as mudanças do código devem ser propagadas.

Sua operação então é simples e direta: Edições no arquivo do CG/IR serão propagadas automaticamente para o mesmo trecho do *template*.

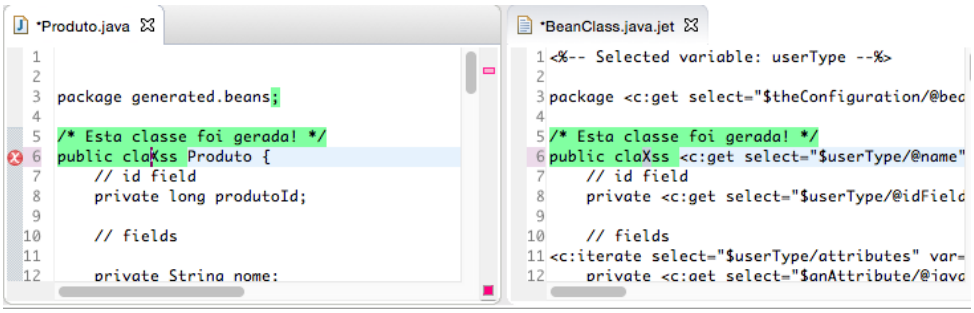

Figura 26: Resultado de uma modificação efetuada na tela esquerda (código gerado).

Na Figura 26 podemos notar a edição do arquivo "Produto.java". O cursor está posicionado no meio da palavra reservada "*class*" e o usuário digita o caracter "X". O editor reconhece a posição e destaca esse trecho de texto em verde, indicando que as mudanças estão sendo propagadas normalmente. Sendo assim, no lado direito da tela a letra é inserida na posição correspondente do arquivo de *template*.

O resultado desta modificação irá gerar uma série de situações que ajudarão a descrever a operacão do editor.

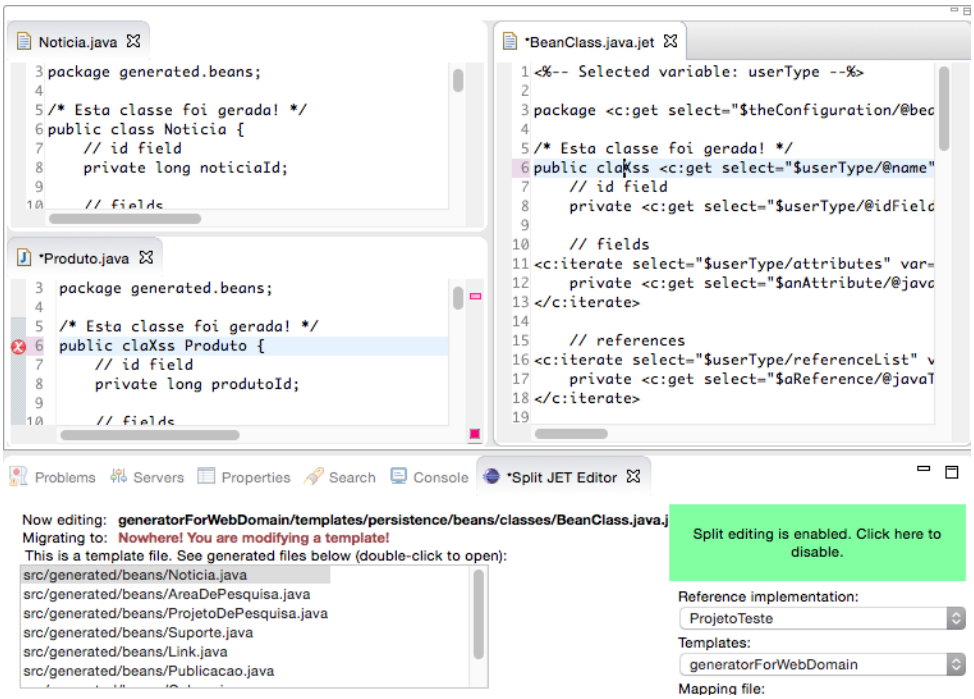

Figura 27: Inconsistência após a modificação de um *template*.

Na Figura 27 temos o editor, agora operando com três janelas ativas (os códigos gerados "Noticia.java" e "Produto.java" e o *template* "*BeanClass.java.jet*"). Isso é possível graças à flexibilidade do ambiente *Eclipse* que permite que sejam abertos vários editores

simultaneamente. A Figura 27 também mostra que o foco de edição é o arquivo de *template* "*BeanClass.java.jet*" (note o cursor sobre "*class*" e também o conteúdo do campo "*Now editing*" no lado inferior esquerdo). Neste caso, o usuário recebe um alerta de que as mudanças não estão sendo propagadas pois um *template* está sendo editado no momento (mensagem em vermelho no canto inferior esquerdo). O editor impede esta modificação bloqueando a edição. Caso deseje alterar o *template*, o usuário precisa desligar a edição simultânea, executar a geração de códigos normalmente e as mudanças serão propagadas.

Um último ponto de destaque da Figura 27 é a inconsistência entre os arquivos gerados. No exemplo da Figura 27, foi modificado o arquivo "Produto.java", e a mudança foi propagada para o *template* "*BeanClass.java.jet*". Porém, ela ainda não foi propagada para os demais arquivos gerados pelo *template* correspondente (note que o caracter "X" não aparece no arquivo "Noticia.java"). Isso acontece pois há uma relação de um-para-muitos entre *template* e arquivos gerados. Esse *template* em particular ("*BeanClass.java.jet*") gera oito arquivos diferentes, alguns dos quais são mostrados na lista no lado inferior da Figura 27.

Este comportamento é esperado, pois a mudança só será propagada para todos os arquivos quando uma nova geração de código for executada. Até lá, os arquivos não modificados ficam em estado inconsistente. Caso o usuário tente alterá-los, os trechos mapeados serão exibidos em cor amarela, alertando para o fato de que as mudanças podem gerar resultados não desejados. A Figura 28 mostra o que acontece quando o usuário tenta alterar o mesmo trecho do arquivo "Noticia.java".

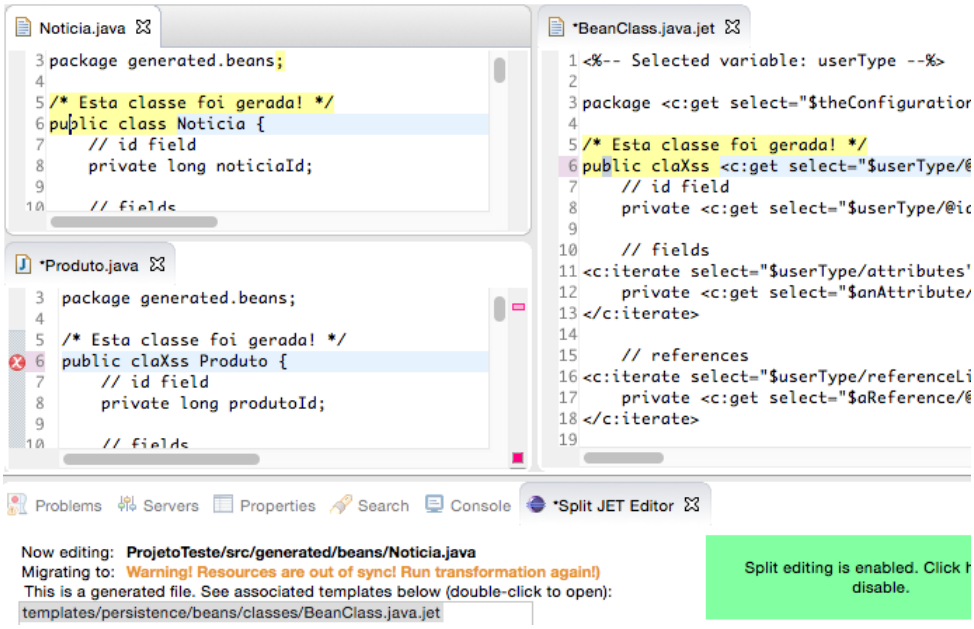

Figura 28: Edição não permitida por falta de sincronismo entre artefatos.

Na Figura 28, o foco de edição é o arquivo "Noticia.java". Além do destaque estar em amarelo, é exibida uma mensagem, no lado inferior esquerdo da tela, avisando que

os recursos não estão sincronizados, e sugerindo uma nova geração de código ("*transformation*", na figura) seja executada novamente. Mas nada impede que mudanças sejam efetuadas. Apenas é necessário cautela para não sobrepor uma mudança realizada anteriormente. No exemplo da Figura 28, se for inserida uma letra na palavra-chave "*public*", que não foi afetada pela primeira mudança, a migração ocorrerá com sucesso.

A seguinte situação merece destaque na operação do editor. Alguns elementos do *template* realizam consulta ao modelo de entrada para gerar código. Nestes casos, a mudança não deve ser feita no *template*, e sim no modelo, portanto o usuário fica impedido de realizar qualquer mudança, prevenindo assim que a lógica do *template* seja quebrada. A Figura 28 ilustra essa situação.

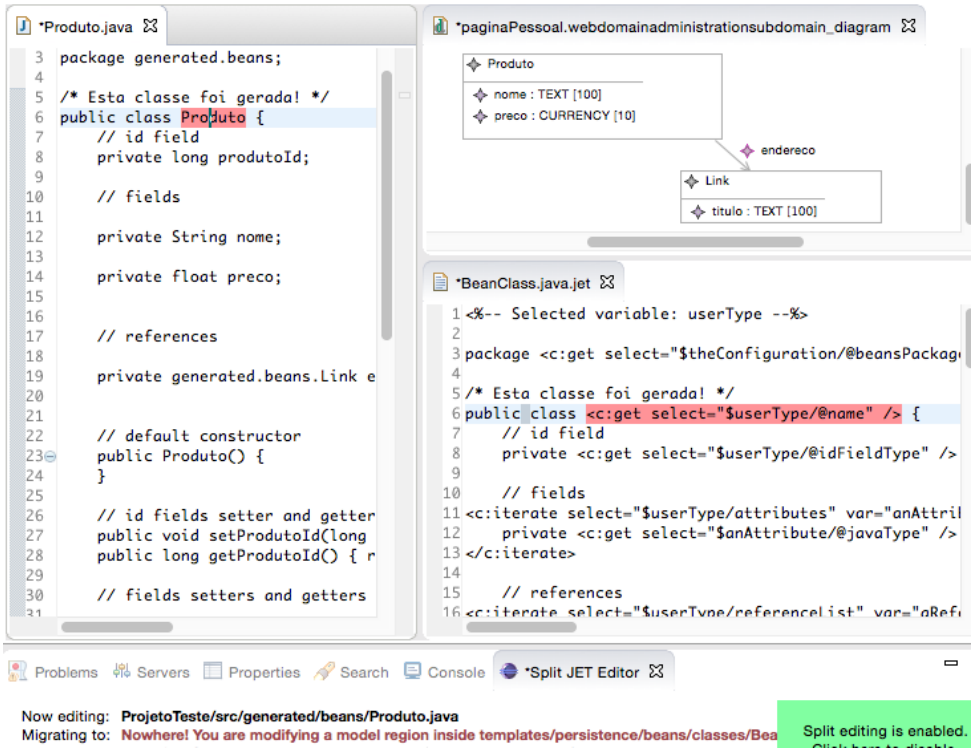

Figura 29: Tentativa de edição de trechos provenientes do modelo.

Como pode ser visto na Figura 29, o nome da entidade "Produto" de origem no modelo, é recuperado por meio da etiqueta "`<c:get select="$userType/@name" />`" do *template*. A etiqueta utiliza a tecnologia XPath [2] (W3C, 1999), que é utilizada pelo mecanismo *JET*, para traduzir "`$userType/@name`" para o texto "Produto" conforme o modelo de dados. Assim, no código gerado, esse texto é que substitui a etiqueta destacada em vermelho.

No entanto, se uma modificação deste tipo é desejada (a mudança do nome da classe para "Produtos"), o lugar correto para esta modificação é o modelo de dados. Este tipo de mudança está fora dos limites do projeto. O editor alerta o usuário com uma

[2] *XPath* é uma linguagem XML de extração de dados.

mensagem em vermelho no lado inferior esquerdo da tela, destaca o trecho que acesso o modelo em vermelho e impede a edição deste trecho. Desse modo, evita que a lógica do *template* seja corrompida.

O protótipo tem algumas particularidades em sua implementação, que merecem ser mencionadas:

- Embora a manipulação de grandes trechos de código nas operações de substituição, deleção ou copiar e colar tenha sido implementada de forma satisfatória, esta encontra-se delimitada dentro dos limites de um mapeamento;
- Mudanças realizadas dentro de estruturas de laço no *template* irão fatalmente causar inconsistência, pois se trata de um caso de mapeamento um-para-muitos (um único trecho do *template* produz múltiplos trechos de código gerado). É necessário cautela nesse tipo de mudança;
- Não há suporte automático do editor na modificação da lógica de um *template* editando etiquetas *JET*. Neste caso, o usuário precisa desligar a edição simultânea e realizar a mudança manualmente.

4.4 Considerações finais

Deste modo, implementou-se um editor simultâneo por meio de *plug-ins* que integraram perfeitamente o editor dentro da *IDE Eclipse*. O editor opera sobre uma estrutura subjacente que reconhece regiões de código de ambos os lados da relação CG/IR e *template*. Assim sendo, o editor permite que modificações no código gerado sejam propagadas para o lado *Template*, levando estas modificações pra dentro do Gerador de Códigos e assim atualizando-o.

5 Estudo empírico para avaliação da abordagem

O estudo experimental realizado para este trabalho encontra-se dividido em dois capítulos diferentes. A primeira metade do estudo trata do planejamento e da metodologia utilizada para este estudo (meta-estudo). A segunda metade, localizada no Capítulo 6, trata de apresentar e avaliar os dados coletados pelo experimento.

Deve-se destacar o fato que este estudo espelhou-se no estudo efetuado pelo trabalho de Possatto (2014). Da definição à execução do experimento, este estudo reutilizou o enfoque, a perspectiva, o planejamento, o critério de seleção dos participantes, a instrumentação (com os mesmos documentos de coleta de dados, a comparação de métodos e tarefas). O intuito desse reuso visa a comparação dos resultados deste trabalho com o trabalho anteriormente referenciado.

Esta comparação encontra-se na Seção 6.5 ao final do Capítulo 6.

5.1 Definição

Este estudo empírico tem o objetivo de analisar o esforço, medido em tempo, na edição de arquivos de código gerados por *Templates*. Nessa análise, foi comparado o tempo gasto pelo participante na conclusão de tarefas utilizando dois métodos diferentes.

Os dois métodos empregados pelos participantes na execução das tarefas diferenciavam apenas nos passos tomados pelo desenvolvedor para se chegar ao resultado final. Para os dois métodos o ambiente era o mesmo, a diferença residiu no uso ou não do protótipo para a edição/manutenção do código.

Os métodos escolhidos para a obtenção da comparação dentro processo são:

1. Método manual: Alterações são efetuadas no código gerado com o protótipo de edição *desligado*. Deste modo não havia replicação alguma para os templates. As modificações nos *templates* foram efetuadas manualmente;
2. Método que utiliza o protótipo: Alterações são efetuadas no código gerado com o protótipo de edição *ligado*. O editor replicava as mundanças no código gerado para os *templates*.

No experimento, foram projetadas quatro tarefas de migração de código (T1, T2,

T3, T4). As tarefas foram concebidas buscando exercitar as atividades típicas de manutenção em uma aplicação *Web*, envolvendo:

1. Analisar algum requisito de mudança;
2. Localizar o código gerado correspondente. Aplicar e salvar a mudança;
3. Localizar o *template* correspondente. *Aplicar* e *verificar* a mudança;
4. Executar a geração de códigos; e
5. Verificar se a modificação foi realizada corretamente.

5.2 Sujeito do estudo

O sujeito do estudo é um editor simultâneo de *templates* de geração de códigos. O editor será utilizado na edição de *templates* de geração de códigos de um Gerador de Códigos. Esse Gerador de Códigos (LUCRéDIO, 2009) produz uma aplicação *Web* que é a IR do estudo.

5.3 Enfoque

Foi dado um enfoque quantitativo ao estudo pois este baseia-se na quantidade de esforço que foi despendido pelos participantes do experimento na conclusão de determinadas tarefas de evolução sobre um código gerado.

Ao mesmo tempo, a partir de um documento denominado "Questionário de Avaliação"oferecido aos participantes, buscou-se uma avaliação qualitativa do estudo por esses.

O material de enfoque qualitativo encontra-se no Apêndice A.4 deste trabalho.

5.4 Perspectivas

O experimento foi conduzido na perspectiva da figura dos desenvolvedores de *software*, mais precisamente dos desenvolvedores de Geradores de Códigos, que precisam manter e implementar mudanças no código gerado por *Templates*.

Na abordagem utilizada, que se vale do conceito de IR, a propagação de mudanças do CG/IR para os *Templates* é fundamental, pois sincroniza seus artefatos permitindo a evolução do Gerador de Códigos, e consequentemente, para a evolução do sistema de *software*.

5.5 Objeto de estudo

O esforço necessário para a conclusão de uma tarefa de evolução sobre um *template*, a partir de alterações feitas no código gerado por este, é o objeto deste estudo empírico.

5.6 Planejamento

Esse estudo empírico, busca responder a questão: "O uso do protótipo de edição simultânea de *templates* de geração de códigos traz ganhos de produtividade no processo de evolução dos *templates*?"

Desse modo, tarefas foram solicitadas aos participantes do estudo, onde se buscou aferir o tempo gasto para a conclusão de tarefas utilizando um método manual e pontual, e também, um método que utiliza o protótipo desenvolvido neste trabalho.

Anteriormente ao experimento controlado, foi aplicada uma tarefa piloto para familiarizar os participantes com a prática das tarefas de evolução na aplicação visando treiná-los para o experimento propriamente dito.

As tarefas são de evolução de *software* em uma aplicação *Web*, mais precisamente dentro do código gerado por *templates* desta aplicação. Para o experimento controlado, foram utilizadas quatro tarefas básicas distintas de manutenção em localidades diferentes da aplicação.

Uma descrição resumida das tarefas utilizadas pelo experimento encontram-se listadas a seguir:

Tarefa Piloto: Corrigir o erro na instrução *SQL* "`Select`" da classe de persistência do banco de dados *Derby*;

Tarefa 1: Modificar um formulário da Área de Usuário alterando o tamanho dos campos "Nome", "Email" e "Assunto" para o valor "29" e alterar o título do formulário de "UserContentTitle" para o texto "Comentários de Usuários";

Tarefa 2: Modificar todos os formulários de cadastro da Área Administrativa alterando o conteúdo dos *links* "Cadastrar novo" para o texto "Adicionar.....";

Tarefa 3: Modificar os formulários da Área Administrativa e de Usuário inserindo uma imagem (o logotipo da UFSCar) no topo dos formulários entre `</head>` e `<body>`;

Tarefa 4: Modificar um formulário da Área Administrativa alterando o tamanho do campo "Área de Pesquisa" para `size=30`.

No experimento, as tarefas foram entregues para participantes previamente separados em grupos balanceados para que estes realizassem as tarefas de manutenção de

acordo com um método já previamente definido, e para que, esses participantes também aferissem o horário de início e de término de cada tarefa (vide Tabela 1).

Tabela 1: Os participantes e seus grupos.

Participante	Grupo
P1	1
P2	1
P3	1
P4	1
P5	2
P6	2
P7	2
P8	2

As tarefas foram separadas por grupos e essas foram tanto executadas pelo método manual, como pelo método que utiliza o protótipo. Os métodos empregados pelos participantes variavam de acordo com o grupo no qual o participante pertencia, e foram aplicados de forma intercalada.

Ademais, partiu-se do pressuposto de avaliar o método com o protótipo no contexto geral das tarefas e não o método com o protótipo em relação aos tipos de tarefas. Do mesmo modo, para que o uso de um método não interferisse no resultado de conclusão do tempo de outro, cada tarefa foi executada em um determinado método por um participante somente uma vez.

Basicamente foram coletados pelo experimento as seguintes informações:

1. Nome do participante;
2. Tarefa executada;
3. Método empregado;
4. Horário de início da tarefa; e
5. Horário do término da tarefa.

Após o término do experimento, o participante entregava um documento de coleta de dados denominado ”Questionário de avaliação” de enfoque qualitativo, com o intuito de obter uma opinião com relação ao uso do protótipo, sugestões de modificações, situações de instabilidade ou erros. Este documento pode ser visualizado no Apêndice A.4 deste trabalho.

5.7 Seleção de contexto

O experimento deu-se em um dos Laboratórios de Ensino do Departamento de Computação da Universidade Federal de São Carlos, e contou com a presença de alunos de mestrado e doutorado em cursos de Ciência da Computação. Ao todo, 8 (oito) alunos participaram do experimento, sendo 6 (seis) alunos de mestrado e 2 (dois) alunos de doutorado.

Os equipamentos do laboratório possuiam a mesma configuração, Sistema Operacional e *software*. Os participantes não foram autorizados a mudar de máquina durante o experimento, não havendo então possibilidade de interferência entre ambientes.

No experimento, os participantes utilizaram o seguinte ambiente:

- *Ubuntu 14.04 64 bits;*
- *JRE 1.7.0_71 64 bits;*
- *Eclipse Luna Service Release 1 v4.4.1 (Build id: 201409251800);*
- *Apache Derby 10.11.1.1;*
- *Apache Tomcat 7.0.56.*

A *Eclipse IDE* estava equipada com as seguintes tecnologias:

- *Eclipse Web Tools 3.6.1.*;*
- *Eclipse Modeling Tools 4.4.1;*
- *JET Modificado 1.1.1;*
- *Xpand SDK 2.0.0.v201406030414;*
- *Xtext Complete SDK 2.7.2.v201409160908;*
- *Graphical Modeling Framework (GMF) Tooling SDK 3.2.1.201409171321;*
- *Split JET Editor plug-in 1.0.0.201411060945.*

Os seguintes projetos estavam configurados e funcionais:

- *generatorForWebDomain*: Gerador de Códigos
- ProjetoTeste: Aplicação *Web* De Referência/IR
- ProjetoTesteGerador: Modelos de dados Gerador de Códigos

5.8 Formulação de hipóteses

Foram levantadas três hipóteses para este estudo empírico. Estas hipóteses estão baseadas no tempo médio gasto para se concluir as tarefas de evolução de *software*, de acordo com os métodos aplicados. As seguintes hipóteses da Tabela 2 foram elencadas:

Tabela 2: As hipóteses do estudo.

Hipótese	Descrição	Condição
H0	Não há diferença no uso tanto do método que utiliza o protótipo, como no uso do método manual para evoluir templates.	$mTC_M - mTC_P \approx 0$
H1	Há uma redução do esforço dispendido na evolução de templates, medido em tempo, utilizando-se do método com o protótipo.	$mTC_M > mTC_P$
H2	Há uma redução do esforço dispendido na evolução de *templates*, medido em tempo, utilizando-se do método manual	$mTC_M < mTC_P$

5.9 Seleção de variáveis

Somente uma variável dependente foi elencada, "Tempo de Conclusão de Tarefa", e seu valor é obtido subtraindo o horário de término da tarefa pelo horário de início da tarefa.

Assim, podemos ter o tempo médio de execução quando empregado o método manual, dado por: mTC_M. Do mesmo modo, o tempo médio de execução quando empregado o método que utiliza o protótipo, é dado por: mTC_P.

As seguintes variáveis independentes foram verificadas: A variável "Método Empregado" e a variável "Tarefa Concluída".

A Tabela 3 lista as variáveis de forma conveniente:

Tabela 3: As variáveis selecionadas para o estudo.

Tipo	Variável	Descrição
Dependente	mTC_M	Tempo Médio de Conclusão de Tarefa pelo Método Manual
Dependente	mTC_P	Tempo Médio de Conclusão de Tarefa pelo Método que utiliza o Protótipo
Independente	T	Tarefa Concluída
Independente	M	Método Empregado

5.10 Critério de seleção de participantes

A seleção de participantes para o estudo seguiu a técnica de Amostragem Não-Probabilística por Conveniência (WOHLIN et al., 2000), escolhendo os participantes mais próximos e mais convenientes para a realização do estudo.

Os participantes escolhidos eram alunos do programa de pós-graduação (mestrandos e doutorandos) em Ciências da Computação, todos são da área de Engenharia de *Software* com experiência em abordagens baseadas na *MDE* e *Web*.

5.11 Projeto

O projeto do estudo do experimento teve o cuidado prévio de separar os participantes de modo igualitário em dois grupos distintos e balanceados utilizando critérios como o número de participantes, o nível acadêmico e o nível de experiência em tecnologias e métodos, informações que foram obtidas pelas respostas do documento "Formulário de Caracterização".

O documento "Formulário de Caracterização" fornecido continha quatro questões (Q1, Q2, Q3, Q4) sobre nível de experiência e oferecia quatro respostas padrão para cada pergunta. As questões eram as seguintes:

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

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

3 Qual é a sua experiência com o desenvolvimento e/ou manutenção de aplicações *Web*?

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

E as quatro respostas padronizadas variavam do nível de pouca experiênia (1) até o nível experiente (4). As possíveis respostas encontram-se a seguir:

1 Nunca utilizei/executei;

2 Utilizei/executei uma vez;

3 Utilizei/executei poucas vezes; e

4 Utilizo/executo com frequência

Com a contagem dos formulários, o nível acadêmico de cada participante e as respostas das questões preenchidas e classificadas pôde-se dividir os participantes. O formulário pode ser encontrado no Apêndice A.1 deste trabalho.

Do mesmo modo, o estudo teve o cuidado de aplicar as tarefas de modo simultâneo para que fossem executadas tanto pelo método usando o protótipo como pelo método manual.

Tabela 4: Grupos e tarefas intercaladas. T1-T4: Tarefas.

	Tarefas			
Grupo	**T1**	**T2**	**T3**	**T4**
1	Manual	Protótipo	Manual	Protótipo
2	Protótipo	Manual	Protótipo	Manual

Pode se notar na Tabela 4, uma alternância na aplicação dos métodos pelos grupos do estudo. Isto se deu, como mencionado anteriormente em 5.6, para evitar que o uso de algum método interferisse no uso de outro método impactando nos resultados do tempo de conclusão de uma determinada tarefa.

O documento ”Formulário de Caracterização” pode ser encontrado no Apêndice A.1 deste trabalho.

5.12 Instrumentação

Para este experimento, foram entregues documentos para os participantes na forma de formulários, questionários e fichas. Esta entrega foi feita da seguinte forma: inicialmente e após a divisão de grupos, seguindo critérios para entregar diferentes documentos para diferentes grupos.

O primeiro documento entregue aos participantes do experimento foi o ”Formulário de Caracterização”, sendo seguido pelo ”Roteiro Piloto de Execução”, e ainda pelo *kit* [1] do experimento.

O *kit* continha o roteiro das tarefas a serem executadas com o objetivo de apoiar e guiar o experimento, e também, um formulário de avaliação para obter um retorno dos participantes.

Existiam dois tipos de *kit* destinados aos dois grupos diferentes. Estes conjuntos diferiam no conteúdo do documento ”Roteiro da Execução de Tarefas”. Os grupos então alternavam o emprego de métodos para uma determinada tarefa.

[1] O *Kit* é somente um termo encontrado para se referir ao conjunto de documentos com os métodos já intercalados para cada grupo do experimento.

Os documentos fornecidos foram:

1. Formulário de Caracterização: Documentos para coleta da ficha do participante, utilizada para a classificação do participante (Documento A.1);
2. Roteiro Piloto de Execução de Tarefas (Documento A.2);
3. Kit do experimento, composto de:

   a) Roteiro da execução de tarefas: Documentos para a coleta de dados que especifica a sequência correta das tarefas; e

   b) Questionário de Avaliação: Documento para a coleta de uma avaliação dos participantes sobre a ferramenta (Documento A.4).

A seguir, o conjunto acima denominado "Roteiro de Execução de Tarefas" (3.a) é listado detalhadamente conforme a Tabela 4:

Kit/Grupo 1 :

- Tarefa 1: Manual (Documento A.3.1);
- Tarefa 2: Protótipo (Documento A.3.4);
- Tarefa 3: Manual (Documento A.3.5); e
- Tarefa 4: Protótipo (Documento A.3.8).

Kit/Grupo 2 :

- Tarefa 1: Protótipo (Documento A.3.2);
- Tarefa 2: Manual (Documento A.3.3);
- Tarefa 3: Protótipo (Documento A.3.6); e
- Tarefa 4: Manual (Documento A.3.7).

Todo este material encontra-se no Apêndice A deste trabalho.

5.13 Preparação

Primeiramente foram entregues aos participantes o documento ”Formulário de Caracterização”. Logo após, foi realizada uma apresentação da Aplicação *Web* de Referência, do processo de evolução neste ambiente, do protótipo e o funcionamento deste dentro do ciclo de desenvolvimento. A apresentação da Aplicação *Web* de Referência visou familiarizar os participantes com o projeto de uma aplicação *Web* criada dentro da abordagem *MDE* por meio de Geradores de Códigos que utilizam *templates*. O processo de manutenção e evolução de *templates* foi descrito e exercitado tanto pelo pelo método manual, como pelo método que utiliza o protótipo.

Sequencialmente, foram distribuídos aos participantes o documento ”Roteiro do Experimento Piloto” que continha uma tarefa que foi executada pelos dois métodos, com o intuito de treinar os participantes para o experimento controlado. A tarefa piloto foi feita com o auxílio do instrutor do experimento, e serviu também para a checagem de problemas que poderiam aparecer no momento do experimento controlado.

Seguindo o andamento, foi requisitada a devolução do documento ”Formulários de Caracterização” com o intuito de dividir os participantes em dois grupos de trabalho. Os formulários foram analisados e dois grupos foram criados: O grupo 1 e o grupo 2. O próximo passo foi distribuir os *kits* com os métodos intercalados.

Esse material encontra-se no Apêndice A deste trabalho.

5.14 Execução

No decorrer do experimento controlado, os grupos efetuaram tarefas de evolução dentro de *templates* da Aplicação *Web* de Referência por meio de roteiros de execução distintos que continham métodos diferentes de execução das tarefas.

Ao início de uma determinada tarefa, os participantes preenchiam o campo ”Horário de início da tarefa”, e da mesma forma, preenchiam o campo ”Horário de término da tarefa” ao final de uma determinada tarefa.

Instruções dadas previamente, informaram sobre o preenchimento do documento ”Questionário de Avaliação” e a sua devolução ao final do experimento pelo participante.

Durante a execução do experimento controlado, não foi permitido aos participantes nenhuma ajuda exterior, mesmo sendo esta por meio da rede internet, para evitar qualquer possível auxílio externo ou interferência.

5.15 Considerações finais

Um estudo experimental foi planejado e concluído. Do seu projeto à sua realização concreta, buscou-se seguir normas consagradas por métodos científicos. Os tópicos deste capítulo aqui descrevem este estudo em seus detalhes.

6 Avaliação da abordagem

Nesse capítulo é apresentada uma avaliação do experimento que teve o objetivo de aferir o uso do protótipo dentro do processo de edição por um desenvolvedor de *software*.

A avaliação deu-se pela comparação das diferenças entre o tempo médio de conclusão de tarefas utilizando o método manual de edição e o método que utiliza o protótipo na edição.

6.1 Validação dos dados

Validações foram efetuadas sobre o documento "Roteiro de Execução do Experimento" buscando por valores inválidos nos campos de resposta. Foram checados e validados os seguintes campos:

- Nome do Participante;
- Horário de início da Tarefa; e
- Horário de término da Tarefa.

Foi procurado por campos em branco e horários inconsistentes, *e.g.*, horário de término anterior ao horário de início da tarefa

6.2 Coleta de dados

O documento "Formulário de Caracterização" possibilitou coletar os dados do nível de experiência dos participantes (Tabela 6.2).

Do mesmo modo, o documento "Roteiro de Execução do Experimento" possibilitou coletar os horários de início e término de tarefas. Os horários já se encontram convertidos de horário de início e término de tarefas para tempo de conclusão de tarefas em MM:SS (minutos e segundos) (Tabela 6.2).

Na Tabela 6.2 podemos notar que ao passo do roteiro de execução de tarefas (T1,T2,T3,T4), os métodos empregados para a realização destas tarefas alternavam com relação ao grupo de participantes. Exemplificando: O participante 1 (P1), do grupo 1, efetuava uma tarefa pelo método manual (TM1), enquanto que o participante 5 (P5), do grupo 2, executava a mesma tarefa mas usando o método que utiliza o protótipo (TP1).

Tabela 5: Nível de experiência dos participantes. P1-P8: Os participantes do estudo; Q1-Q4: Questões sobre a experiência do participante com tecnologias e métodos.

Participante	Nível	Q1	Q2	Q3	Q4
P1	2	4	4	3	4
P2	2	4	3	4	2
P3	3	4	4	3	3
P4	2	3	3	4	3
P5	2	3	3	3	2
P6	2	3	3	3	1
P7	3	4	4	3	4
P8	2	4	3	3	2

Tabela 6: Participantes e os seus respectivos tempos de execução de tarefas do experimento. P1-P8: Os participantes do estudo; T1M-T4M: Tarefas executadas pelo método manual; T1P-T4P: Tarefas executadas pelo método utilizando o protótipo; e tempo em MM:SS (minutos e segundos).

Participante	TM1	TP2	TM3	TP4
P1	15:00	07:00	10:00	06:00
P2	07:00	02:00	04:00	03:00
P3	09:00	01:00	07:00	03:00
P4	06:00	2:00	15:00	04:00
Participante	**TP1**	**TM2**	**TP3**	**TM4**
P5	06:00	07:00	13:00	16:00
P6	02:00	05:00	06:00	08:00
P7	04:00	02:00	04:00	06:00
P8	09:00	10:00	16:00	08:00

6.3 Análise de dados e interpretação

Para que o processo de análise seja facilitado, os dados coletados pelo experimento são oferecidos na forma de gráficos.

Nestes quatro primeiros gráficos (Figuras 30 a 33), pode-se notar que os tempos de conclusão das tarefas feitas com o uso do método do protótipo mostram-se majoritariamente menores do que os tempos de conclusão de tarefas feitas com o uso do método manual.

No entanto, na Figura 32 que apresenta os dados da Tarefa 3, os tempos de conclusão pelo método do protótipo são em sua maioria, maiores do que o tempo de conclusão da mesma tarefa sendo executada pelo método manual. Após verificação, obteve-se que

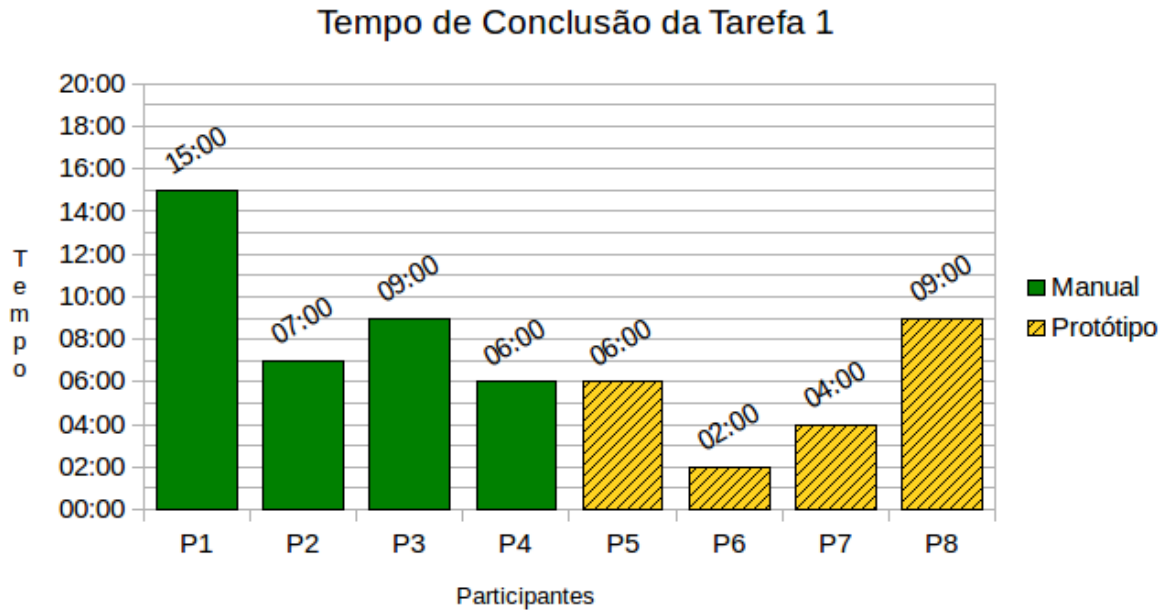

Figura 30: Tempos aferidos da Tarefa 1. Desvio padrão Tempo em MM:SS (minutos e segundos).

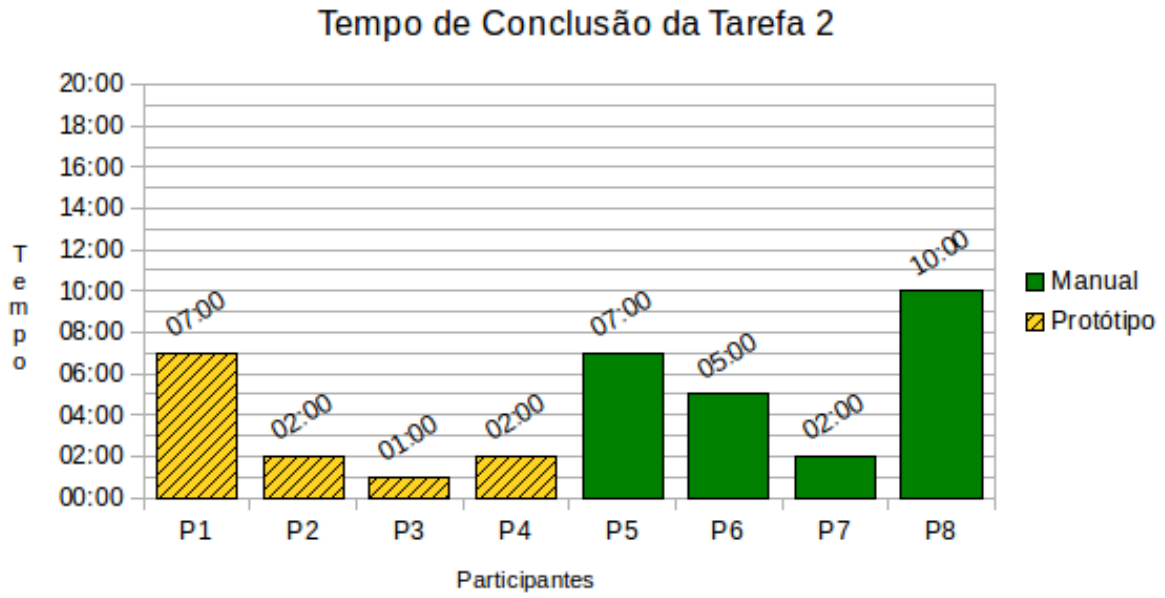

Figura 31: Tempos aferidos da Tarefa 2. Tempo em MM:SS (minutos e segundos).

estes participantes tiveram problemas nesta tarefa. A Tarefa 3 é uma tarefa de inclusão de uma imagem por meio de uma etiqueta `<img>` do *HTML* no código. O relato, no Formulário de Avaliação, indicou que um dos participantes sentiu dificuldades configurando o caminho da imagem dentro dos três projetos abertos do *Eclipse*.

Na Tabela 7 estão listadas as médias aritméticas do tempo tomado por tarefas, e a proporção de tempo que cada método tomou do tempo total do experimento.

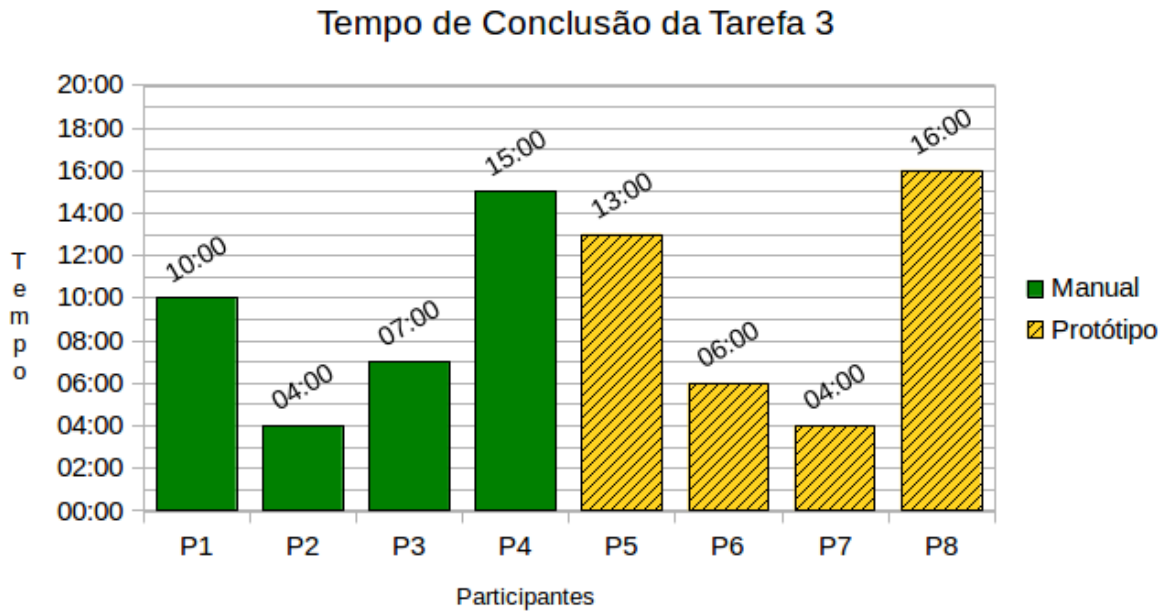

Figura 32: Tempos aferidos da Tarefa 3. Tempo em MM:SS (minutos e segundos).

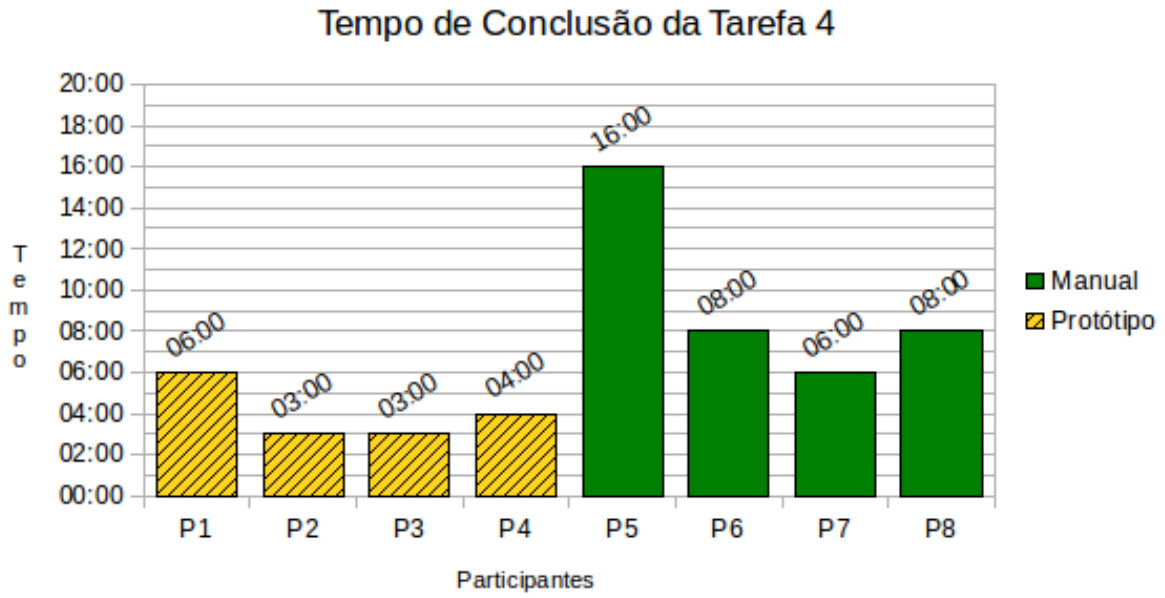

Figura 33: Tempos aferidos da Tarefa 4. Tempo em MM:SS (minutos e segundos).

Na Figura 34, na forma de um gráfico de barras, as médias aritméticas dos tempos de conclusão de tarefas da Tabela 7 encontram-se dispostas lado a lado. Nesta figura, podemos notar que a Tarefa 4 teve seu tempo reduzido, quando executada pelo método que utiliza o protótipo, para menos que metade de seu tempo de execução pelo método manual.

A Figura 35 é um gráfico que facilita a visualização de proporções, e nele podemos

Tabela 7: Comparação entre as tarefas executadas por diferentes métodos e relações entre os dados. T1-T4: Tarefas executadas; G1-G2: Grupos de participantes; e Tempo em MM:SS (minutos e segundos).

Método	Tarefa	Grupo	Tempo médio	Soma	Proporção no tempo total
Manual	T1	G1	09:15	33:45	60,54%
	T2	G2	06:00		
	T3	G1	09:00		
	T4	G2	09:30		
Protótipo	T1	G2	05:15	22:00	39,46%
	T2	G1	03:00		
	T3	G2	09:45		
	T4	G1	04:00		
			Total	55:45	100,00%

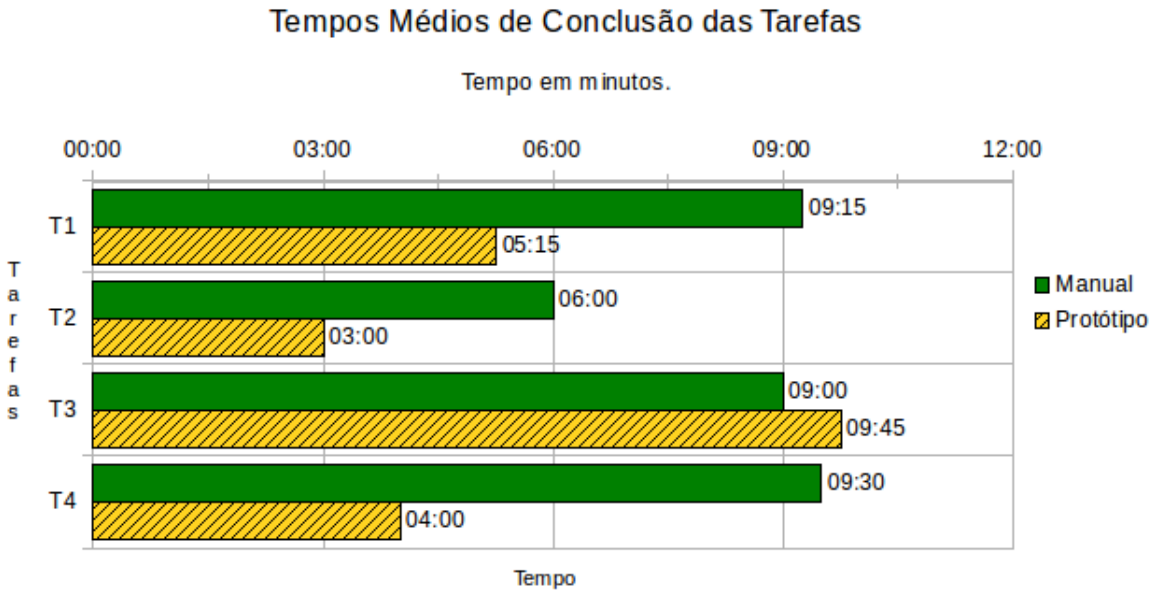

Figura 34: Médias aritméticas dos tempos de conclusão de tarefas.

visualizar a parte que cada método tomou do tempo total do experimento.

Já na Tabela 8 podemos visualizar a média total de cada método aplicado, listada pela coluna ”Média do método”, e ainda pela coluna ”Proporção do método Protótipo no Manual” que o tempo médio do protótipo corresponde a 65% (65,19) do tempo médio do método manual. E que, de acordo com a coluna ”Razão entre médias de métodos” o tempo médio do protótipo está mais de uma vez e meia (1,53) representado dentro do tempo médio do método manual.

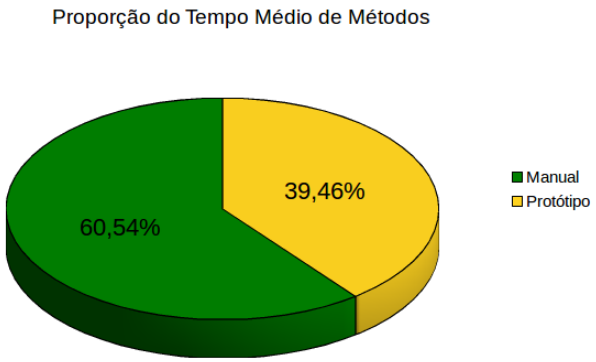

Figura 35: Tempos médios de métodos e a sua proporção no tempo total do experimento.

Tabela 8: Médias de métodos e relações entre seus valores. Tempo em MM:SS (minutos e segundos).

Método	Média do método	Proporção do método Protótipo no Manual	Razão entre médias de métodos
Manual	08:26	65,19%	1,53
Protótipo	05:30		

6.4 Teste de hipóteses

De posse dos tempos médios totais de cada método, podemos agora validar as hipóteses deste estudo.

O tempo médio aferido dos métodos apresenta uma clara diferença, e sendo assim as médias não são equivalentes. Logo, a diferença entre as duas médias não é próxima de 0 (zero), o que nos permite rejeitar a hipótese H0 (Hipótese Nula).

Levando-se em conta o ambiente simulado em laboratório, o número reduzido de participantes e principalmente as quatro tarefas propostas, este estudo aferiu que o tempo médio de conclusão do método manual foi proporcionalmente superior ao tempo médio do método que utiliza o protótipo. Nas condições do estudo, verifica-se que a Hipótese 1 é favorecida, pois leva-se menos tempo para manter e evoluir *Templates* de geração de códigos pelo método que utiliza o protótipo perante o método manual.

Portanto houve uma redução do esforço dispendido na evolução dos *Templates*, medido em tempo, pelo método que utiliza o protótipo.

6.5 Comparação entre este trabalho e o trabalho de Possatto (2014).

Conforme descrito no início do Capítulo 5, este estudo reutilizou a instrumentção do trabalho de Possatto (2014) visando uma comparação entre estas duas abordagens. Esta comparação torna-se possível devido à metodologia e a instrumentação idênticas ao trabalho anterior. Cabe destacar que os participantes não foram os mesmos para os dois experimentos.

Algumas modificações foram necessárias para as tarefas que fazem o uso do protótipo devido a diferença que existe entre os passos do protótipo de Possatto (2014) e os passos do protótipo deste trabalho. Esta modificação foi descrita anteriormente na Seção 5.1.

Desse modo, os tempos dos métodos manuais dos dois estudos foram totalizados pois são exatamente as mesmas tarefas manuais.

Na Tabela 9 que compara os dois trabalhos, e que foi elaborada nos mesmos moldes da Tabela 7, podemos observar a média totalizada do método manual dos trabalhos e também dos métodos que utilizaram os dois diferentes protótipos na coluna ”Média Total”. Na coluna ”Relação da média para com o método manual” temos o tempo médio dos dois diferentes protótipos e a relação destes tempos com a média total do método manual. Deste modo, podemos observar que o tempo tomado pelo protótipo deste trabalho conseguiu diminuir a proporção do tempo tomado pelo protótipo do trabalho de Possatto (2014) no processo.

Ainda, na Tabela 9 podemos vizualizar na coluna ”Razão entre médias de métodos” que a diferença entre as razões é pequena mudando pouco quando comparado com o método manual.

Com os dados aferidos, o protótipo deste trabalho que faz o uso da edição simultânea, conseguiu diminuir ligeiramente o tempo do processo de evolução de software quando comparado ao método do protótipo de migração automática de Possatto (2014). Deste modo, o protótipo deste trabalho mostra-se numericamente semelhante ao trabalho de Possatto (2014) com uma ligeira vantagem. A ligeira vantagem pode ser explicada pela interatividade da edição simultânea no processo. Por outro lado, o protótipo de Possatto (2014) oferece uma edição completamente agnóstica com relação aos *templates* devido a geração automática de passo único.

As duas abordagens apesar de diferentes parecem ser complementares trantando o problema de evoluir *software* de uma forma diferente. Uma investigação melhor deveria ser conduzida para poder verificar esta afirmação corretamente.

Tabela 9: Comparação entre as tarefas executadas por diferentes métodos e relações entre os dados deste trabalho com o trabalho de Possatto (2014). T1-T4: Tarefas executadas; G1-G2: Os dois grupos de participantes do trabalho de Possatto (2014); G3-G4: Os dois grupos de participantes deste trabalho; e Tempo em MM:SS (minutos e segundos).

Método	Tarefa	Grupo	Média	Média Total	Relação da média com método manual	Razão entre médias de métodos
Manual de Possatto	T1	G1	14:30	10:36		
	T2	G2	07:30			
	T3	G1	15:30			
	T4	G2	13:30			
Manual deste trabalho	T1	G3	09:15			
	T2	G4	06:00			
	T3	G3	09:00			
	T4	G4	09:30			
Protótipo de Possatto	T1	G2	06:45	05:45	54,28%	1,84
	T2	G1	07:30			
	T3	G2	03:45			
	T4	G1	05:00			
Protótipo deste trabalho	T1	G4	05:15	05:30	51,92%	1,93
	T2	G3	03:00			
	T3	G4	09:45			
	T4	G3	04:00			

6.6 Ameaças à validade

Os itens que podem afetar os valores e assim a conclusão do estudo empírico são apresentados neste capítulo.

- As tarefas foram projetadas de forma a exercitar a evolução exclusivamente nos *templates*, e não mudanças no modelo ou lógica de geração de códigos. As tarefas definidas para o estudo focaram em um cenário de melhor caso, e que é exatamente o oposto de um ambiente industrial. A criação, manutenção e evolução de uma infraestrutura de geração de códigos muitas vezes envolve tarefas mais complexas e que exigem mudanças que vão além dos *templates*;

- O número pequeno de participantes do experimento. Essa situação é justificável pois é difícil encontrar participantes com a experiência necessária para esse projeto, que inclui necessariamente a prática com Geradores de Códigos. Assim sendo, os participantes foram os alunos de pós-graduação com projetos de pesquisa na área Engenharia de *Software* que estudam *MDE*.

6.7 Considerações finais

Na Figura 34 da Seção 6.3, podemos visualizar o impacto dos dados da Tarefa 3 (T3) nos resultados. Os dados coletados mostram que 3 (três) dos 4 (quatro) participantes que efetuaram esta tarefa, tomaram um tempo maior utilizando o protótipo do que fazendo-a manualmente. Nesse caso, talvez um número maior de participantes conseguisse amortizar o resultado destes 3 participantes que sentiram dificuldades, ou mesmo, mais treinamento poderia ter sido efetuado para familiarizar os participantes com os projetos envolvidos dentro do ambiente da *IDE*. Um outro experimento poderia ser aplicado com um número maior de participantes para que esse fenômeno fosse melhor avaliado.

As informações originadas pelo documento "Questionário de Avaliação" propiciaram também um levantamento qualitativo com relação ao protótipo. Dentre os pontos destacados, os participantes relatam que o protótipo auxiliou principalmente no processo de localizar, entre os inúmeros artefatos disponíveis, onde a mudança deve ser feita. Também foram relatados benefícios advindos do uso da *interface*, onde os relacionamentos entre código gerado e *template* tornavam-se claros e a mudança intuitiva. Como um todo, avaliaram satisfatoriamente a iniciativa da edição simultânea de artefatos.

Cabe observar que, o experimento visou avaliar os métodos em seu contexto geral. No entanto, em uma das tarefas o método utilizando o protótipo mostrou dados contrários de ganho na conclusão de tarefas. O fenômeno encontra-se explicado pelo documento de avaliação coletado, mas devido à quantidade pequena de amostras resultado do pequeno número de participantes, ficam impossibilitadas análises mais profundas sobre o ocorrido.

Os dados coletados por este trabalho e pelo trabalho de Possatto (2014) estão disponibilizados integralmente no *link* `http://www.dc.ufscar.br/~daniel/codeMigrationExperiment` para consultas.

7 Conclusão

A evolução de *software* em um cenário da geração de códigos por meio de *Templates* implica em mudanças nos códigos da IR, de *templates* de geração de códigos e de seu CG. E, em vez de uma migração automática de código como abordado por Possatto (2014), este trabalho propõe uma abordagem de um editor simultâneo de *templates* e seu código gerado.

O editor simultâneo para *Templates* permite a visualização de uma modificação em um primeiro artefato e o reflexo desta no segundo artefato, provendo um grande auxílio para o refinamento da solução na evolução do *software* envolvido na geração de códigos.

Após sua experimentação, verificou-se que o uso do editor obteve uma diminuição percentual de 34,78% do tempo do processo de edição, nas condições do experimento. Assim, leva-se menos tempo no processo de manter/evoluir *Templates* de geração de códigos.

O resultado do uso do editor simultâneo, conforme dados do experimento, atingiu os objetivos deste trabalho, que é o de reduzir a sobrecarga de 20 a 25% ocasionada pela migração manual de código (MUSZYNSKI, 2005; VOELTER; BETTIN, 2004). E, considerando os ganhos de produtividade obtidos pelo editor, reduz-se o tempo de desenvolvimento da IR para valores de 13 a 16% do tempo no contexto do processo dirigido por modelos.

O ganho de produtividade estabelecido então leva a um dinamismo maior para o processo de evolução de Geradores de Códigos facilitando assim o trabalho de ir e vir entre especificação e a implementação, na depuração do código de geração, reduzindo assim o tempo que um *template* leva para ficar e manter-se funcional.

A solução de sincronização adotada por este trabalho em vez de apoiar-se nas transformações bidirecionais com tradução para outras linguagens intermediárias, como é o caso em *GRoundTram* (HIDAKA et al., 2011) e *JTL* (CICCHETTI; RUSCIO, 2011), utiliza-se de um mapeamento de estruturas comuns entre as linguagens de programação do CG e de *Templates*. A escolha deu-se pela não necessidade de reimplementar o caminho inverso (transformação inversa) que é a própria geração de códigos efetuada pelo mecanismo *JET* (geração dos *templates* para o CG).

O mecanismo interno do protótipo editor apesar de limitado no reconhecimento das etiquetas da linguagem de programação do mecanismo *JET* (assim como faz o editor *JET* que vem com a linguagem), tem toda a infra-estrutura para que isso seja implementado. Esta funcionalidade não foi implementada por falta de tempo hábil. Entretanto, a edição de arquivos de *template* de geração de códigos foi estendida para uma edição especializada

onde alvo e origem são editados e sincronizados simultâneamente.

Relacionando também o protótipo aos editores simultâneos elencados por este trabalho, o protótipo consegue trazer versatilidade para este tipo de edição lado a lado por não restringir a operação somente a uma visualização no estilo fonte e alvo (relação um-para-um). Com o protótipo editor é possível visualizar relações entre um *template* e seus vários CG (relação um-para-muitos) utilizando-se de várias janelas de uma forma muito mais intuitiva e flexível.

Ainda há muita pesquisa a ser feita no sentido de estabelecer a fundação, ferramentas, métodos e processos para a *MDE*. E com esta proposta acredita-se contribuir com uma pequena parte, mais especificamente na criação, manutenção e evolução dos artefatos envolvidos no processo de migração e manutenção de Geradores de Códigos, tornando assim os projetos mais simples e rápidos, diminuindo então as fontes de erros e facilitando a evolução de *software*.

7.1 Limitações

Encontram-se fora dos limites deste trabalho as seguintes situações:

1. Alterações de informações que são de origem do modelo por meio do editor; e
2. Um estudo sobre as melhorias no processo de evolução neste cenário tecnológico.

Na primeira situação, o usuário fica impedido pela *interface* do editor de realizar uma alteração em um dado que venha do modelo, estando este usuário editando tanto um *template* ou o código gerado. Isto se dá devido ao impacto que uma mudança no domínio de dados causaria em ambos os artefatos *Templates* e CG, pois estes dois artefatos dependem de informações do domínio, *e.g.*, ao modificar o nome de uma classe no código gerado espera-se que este nome mude no modelo de dados, mas como decorrência teríamos mudanças também no *template* gerador deste código. O impacto deste tipo de alteração não foi contemplado pelo trabalho, e poderá ser pesquisado futuramente com foco na rastreabilidade de mudanças entre os artefatos do processo. Portanto, caso o desenvolvedor durante a evolução tente editar regiões de informações provenientes do modelo, ou regiões que acessam dados do modelo, o editor impedirá estas mudanças e destacará o texto protegido em vermelho.

Para a segunda situação, existem pontos dentro deste processo onde alterações podem ser efetuadas gerando impactos distintos dentro do processo. E nesse trabalho, como o trabalho de Possatto (2014), assume-se que existe uma IR que foi usada inicialmente para a construção dos *templates*, e que esta implementação inicial foi substituída dando

lugar ao CG. Como discutido no Capítulo 1, há pelo menos três locais onde as alterações podem ser efetuadas. Uma das alterações possíveis é a de artefatos do tipo CG/IR. Uma outra alteração possível poderia ser efetuada nos artefatos do tipo *template* e, por fim, modificações poderiam ser feitas nos artefatos do tipo modelo de domínio. O processo escolhido neste trabalho considera somente a edição do CG/IR para a manutenção e evolução dos Geradores de Códigos.

7.2 Trabalhos futuros

Esta seção ocupa-se de procurar formas para expandir este trabalho. Portanto, utilizando-se das limitações deste ou apontando caminhos ainda não trilhados, busca-se oferecer meios para a ampliação desta pesquisa. A seguir, algumas formas de fazer com que a pesquisa possa ser retomada por outros pontos de vista, ao mesmo tempo que vislumbra novos desafios.

As situações não abordadas por este trabalho já se mostram um interessante ponto de partida para futuras pesquisas. As duas situações descritas anteriormente na seção de limitações do trabalho abrem para discussão os seguintes pontos:

1. Estudo da rastreabilidade do código de *Templates* e CG até o modelo de domínio; e
2. Estudo do processo de desenvolvimento buscando pela melhor forma para se realizar evolução de *software* dentro do processo de geração de códigos por meio de *Templates*.

Um estudo mais aprofundado da rastreabilidade entre *Templates* e o CG para modelos e vice-versa promoveria um ambiente de trabalho altamente sincronizado, contudo, um cuidado maior deve ser tomado quanto à redundância de informações em artefatos e, no alcance e impacto de replicações decorrentes de uma alteração em um artefato do qual vários outros artefatos dependem.

Quanto a uma investigação mais detalhada do processo, esta poderia ser conduzida com vistas no melhor caminho a se seguir dentro do ciclo do processo. Essa pesquisa poderia focar justamente na melhor forma de condução do processo, e de seus artefatos participantes, elegendo assim a melhor maneira de se fazer evolução de *software*. Um estudo deste tipo também poderia trazer uma luz sobre o questionamento: quando é melhor utilizar a abordagem deste trabalho e quando é melhor utilizar a abordagem de Possatto (2014)? As duas abordagens visam a evoculçao de *software* dentro de um Gerador de Códigos que utiliza *Templates* na *MDE*, mas focam em diferentes meios e caminhos para isto.

Uma outra forma de expandir este trabalho poderia ser na implementação do editor. O aumento da capacidade de visualização das mudanças e impacto gerado por estas

em código, poderia ampliar a interatividade do processo. Isso pode ser feito de várias formas, uma delas seria por meio de telas que ofereçam a possibilidade de visualização da interpretação/execução do código gerado. No caso da Aplicação *Web* de Referência, os códigos gerados são páginas *Web* dinâmicas e um visualizador destas seria muito interessante no processo de depuração e certamente agilizaria esta e outras etapas do processo, *e.g.*, no desenvolvimento de *interfaces* gráficas.

Referências

ACCELEO. *Acceleo v.3.4.0 - Acceleo Project*. 2014. http://www.eclipse.org/acceleo/. Acesso em mar, 2013. Citado 2 vezes nas páginas 30 e 54.

AIZENBUD-RESHEF, N. et al. Model traceability. *IBM Systems Journal*, IBM, v. 45, n. 3, p. 515–526, 2006. Citado na página 39.

ANTKIEWICZ, M.; CZARNECKI, K. Design space of heterogeneous synchronization. *Generative and Transformational Techniques in Software Engineering II*, Springer, p. 3–46, 2008. Citado na página 38.

ASENTE, P. J. Editing graphical objects using procedural representations. Stanford University, 1987. WRL Research Report 87/6, Digital Equipment Corporation Western Research Laboratory, Palo Alto, California, November 1987. Citado na página 56.

ATL. *ATL v.3.4 - ATL Project*. 2014. http://www.eclipse.org/atl/. Acesso em mar, 2014. Citado na página 31.

BALLANCE, R. A.; GRAHAM, S. L.; VANTER, M. L. V. D. The pan language-based editing system. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM, v. 1, n. 1, p. 95–127, 1992. Citado na página 57.

BANCILHON, F.; SPYRATOS, N. Update semantics of relational views. *ACM Transactions on Database Systems (TODS)*, ACM, v. 6, n. 4, p. 557–575, 1981. Citado na página 54.

BÉZIVIN, J. In search of a basic principle for model driven engineering. *Novatica Journal, Special Issue*, v. 5, n. 2, p. 21–24, 2004. Citado na página 26.

BÉZIVIN, J. et al. Modeling in the large and modeling in the small. In: *Model Driven Architecture*. [S.l.]: Springer, 2005. p. 33–46. Citado na página 51.

BIEBER, M.; ISAKOWITZ, T. A logic model for text editing. In: IEEE. *System Sciences, 1989. Vol. III: Decision Support and Knowledge Based Systems Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*. [S.l.], 1989. v. 3, p. 543–552. Citado na página 40.

BIEHL, M. Literature study on model transformations. *Royal Institute of Technology, Tech. Rep. ISRN/KTH/MMK*, 2010. Disponível em: <http://staffwww.dcs.shef.ac.uk-/people/A.Simons/remodel/papers/BiehlModelTransformations.pdf>. Citado 3 vezes nas páginas 28, 29 e 30.

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.: s.n.], 2006. p. 67–78. Citado 2 vezes nas páginas 21 e 34.

BROOKS, K. P. Lilac: A two-view document editor. *Computer*, IEEE, v. 24, n. 6, p. 7–19, 1991. Citado na página 56.

BROWN, A. W. Model driven architecture: Principles and practice. *Software and Systems Modeling*, Springer, v. 3, n. 4, p. 314–327, 2004. Citado na página 27.

BUCAIONI, A. *Bidirectionality in model transformations*. Tese (Doutorado) — Mälardalen University, 2013. Citado 2 vezes nas páginas 38 e 39.

BUDINSKY, F. *Eclipse Modeling Framework: A developer's guide*. [S.l.]: Addison-Wesley Professional, 2004. Citado na página 51.

BURKHART, H.; NIEVERGELT, J. Structure-oriented editors. Springer Berlin Heidelberg, 1980. Citado na página 41.

BéZIVIN, J.; GERBé, O. Towards a precise definition of the omg/mda framework. In: IEEE. *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*. [S.l.], 2001. p. 273–280. Citado na página 27.

CHAMBERLIN, D. D. et al. Janus: An interactive document formatter based on declarative tags. *IBM Systems Journal*, IBM, v. 21, n. 3, p. 250–271, 1982. Citado na página 56.

CHEN, P.; HARRISON, M. A. Multiple representation document development. *Computer*, IEEE, v. 21, n. 1, p. 15–31, 1988. Citado na página 57.

CICCHETTI, A.; RUSCIO, D. D. JTL: A Bidirectional and change propagating transformation language. *Software Language ...*, p. 1–20, 2011. Disponível em: <http://www.springerlink.com/index/07855H3U63462001.pdf>. Citado 4 vezes nas páginas 8, 50, 51 e 90.

CLEAVELAND, J. C. Building application generators. *IEEE Software*, v. 7, n. 1, p. 25–33, 1988. Citado 3 vezes nas páginas 17, 32 e 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.: s.n.], 2001. p. 83–86. Citado na página 33.

CZARNECKI, K.; EISENECKER, U. W. *Generative Programming: Methods, Tools, and Applications*. [S.l.]: Addison-Wesley, 2000. Citado 3 vezes nas páginas 17, 20 e 33.

CZARNECKI, K. et al. Bidirectional transformations: A cross-discipline perspective. *... Model Transformations*, 2009. Disponível em: <http://www.springerlink.com/index-/860L777222408610.pdf>. Citado na página 38.

DESRIVIÈRES, J.; WIEGAND, J. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, IBM, v. 43, n. 2, p. 371–383, 2004. Citado na página 46.

DEURSEN, A. van; VISSER, E.; WARMER, J. Model-Driven Software Evolution: A Research Agenda. *CSMR Workshop on Model-Driven Software Evolution (MoDSE 2007)*, p. 41–49, 2007. Disponível em: <http://swerl.tudelft.nl/twiki/pub/Main-/TechnicalReports/TUD-SERG-2007-006.pdf>. Citado na página 19.

DISKIN, Z.; XIONG, Y.; CZARNECKI, K. From state- to delta-based bidirectional model transformations: the asymmetric case. *The Journal of Object Technology*, v. 10, p. 6:1, 2011. ISSN 1660-1769. Disponível em: <http://www.jot.fm/contents-/issue_2011_01/article6.html>. Citado na página 38.

EMF. *EMF - Eclipse Modeling Framework*. 2013. http://www.eclipse.org/modeling/emf/. Acesso em mar, 2014. Citado 2 vezes nas páginas 30 e 44.

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.: s.n.], 2006. p. 169–176. Citado 2 vezes nas páginas 17 e 43.

FINSETH, C. A. The craft of text editing: Emacs for the modern world. Springer-Verlag, 1991. Citado 3 vezes nas páginas 40, 41 e 42.

FOSTER, J. N. et al. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2005. v. 40, n. 1, p. 233–246. Citado na página 54.

FOWLER, M. Language workbenches: The killer-app for domain specific languages?, 2005. 2005. Citado na página 43.

FOWLER, M. *Domain-specific languages*. [S.l.]: Addison-Wesley Professional, 2010. ISBN 9780321712943. Citado na página 43.

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. Citado na página 17.

GALVÃO, I.; GOKNIL, A. Survey of traceability approaches in model-driven engineering. *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, IEEE, p. 313–313, Oct 2007. Disponível em: <http:/-/ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4384003>. Citado na página 39.

GOLDBERG, R. N. Software design issues in the architecture and implementation of distributed text editors. 1982. Disponível em: <http://dl.acm.org/citation-.cfm?id=910286>. Citado 2 vezes nas páginas 40 e 41.

GREENWALD, M. B. et al. A language for bi-directional tree transformations. *Pat*, v. 333, p. 4444, 2003. Citado na página 54.

HAMMER, M. et al. The implementation of etude, an integrated and interactive document production system. *ACM SIGPLAN Notices*, ACM, v. 16, n. 6, p. 137–146, 1981. Citado na página 43.

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. Citado na página 32.

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. Citado na página 32.

HETTEL, T.; LAWLEY, M.; RAYMOND, K. Towards model round-trip engineering: An abductive approach. p. 100–115, 2009. Citado 3 vezes nas páginas 37, 38 e 39.

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. Citado na página 26.

HIDAKA, S. et al. Bidirectionalizing graph transformations. In: ACM. *ACM Sigplan Notices*. [S.l.], 2010. v. 45, n. 9, p. 205–216. Citado na página 49.

HIDAKA, S. et al. GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, Ieee, p. 480–483, nov. 2011. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6100104>. Citado 5 vezes nas páginas 8, 49, 50, 55 e 90.

HU, Z.; MU, S.-C.; TAKEICHI, M. A programmable editor for developing structured documents based on bidirectional transformations. *...on Partial evaluation and semantics-based ...*, v. 122, p. 89–122, 2007. Disponível em: <http://dl.acm.org-/citation.cfm?id=1014025>. Citado na página 54.

IEEE. Ieee standard glossary of software engineering terminology/ieee std 610.12-1990. *Office*, Institute of Electrical and Electronical Engineers, v. 1990, n. 1, p. 96, 1990. Acesso em mar, 2013. Disponível em: <http://www.amazon.com/Standard-Glossary-Engineering-Terminology-610-12-1990/dp/155937067X>. Citado na página 38.

JOUAULT, F.; BÉZIVIN, J. Km3: a dsl for metamodel specification. In: *Formal Methods for Open Object-Based Distributed Systems*. [S.l.]: Springer, 2006. p. 171–185. Citado na página 49.

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. Citado na página 32.

JOY, B. et al. {Java}(tm) language specification. Addison-Wesley, 2000. Citado na página 28.

KIM, L. *The XMLSPY Handbook*. [S.l.]: John Wiley & Sons, Inc., 2002. Citado na página 54.

KINDLER, E.; WAGNER, R. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. *University of Paderborn*, 2007. Citado na página 31.

KOORN, J.; BAKKER, H. Building an editor from existing components: an exercise in software re-use. n. 2177, p. 1–15, 1993. Disponível em: <http://citeseerx.ist.psu-.edu/viewdoc/download?doi=10.1.1.39.2230\rep=rep1\type=pdf>. Citado na página 44.

KORHONEN, K. A case study on reusability of a dsl in a dynamic domain. In: *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. Citado na página 32.

KURTEV, I.; BéZIVIN, J.; AKSIT, M. Technological spaces: An initial appraisal. p. 1–6, 2002. Disponível em: <http://eprints.eemcs.utwente.nl/10206/>. Citado na página 30.

KURTEV, I. et al. Model-based DSL frameworks. *Companion to the 21st ACM ...*, p. 602–615, 2006. Disponível em: <http://dl.acm.org/citation.cfm?id=1176632>. Citado na página 28.

LAMB, L. Learning the vi editor. O'Reilly & Associates, Inc., 1998. Citado na página 41.

LAMPORT, L. Latex: User's guide & reference manual. Addison-Wesley, 1986. Citado na página 42.

LUCRéDIO, D. *A model-driven software reuse approach (in portuguese)*. Tese (PhD thesis) — University of São Paulo, São Carlos, SP, Brazil, 2009. Citado na página 71.

LUCRéDIO, D.; FORTES, R. P. de M. Mapping code generation templates to 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. Citado na página 59.

MENS, T. et al. Challenges in software evolution. In: IEEE. *Principles of Software Evolution, Eighth International Workshop on*. [S.l.], 2005. p. 13–22. Citado na página 19.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. *ACM Computing Surveys*, v. 37, n. 4, p. 316–344, dez. 2005. ISSN 0360-0300. Citado na página 32.

MILLER, J.; MUKERJI, J. et al. Mda guide version 1.0.1. *Object Management Group*, v. 234, p. 51, 2003. Citado na página 27.

MULIAWAN, O. Extending a model transformation language using higher order transformations. *Reverse Engineering, 2008. WCRE'08. 15th ...*, p. 315–318, 2008. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4656425>. Citado na página 31.

MULLER, P.-A. et al. Modeling modeling modeling. *Software & Systems Modeling*, Springer, v. 11, n. 3, p. 347–359, 2012. Citado na página 27.

MUNSON, E. V. *Proteus: An adaptable presentation system for a software development and multimedia document environment*. Tese (Doutorado) — Citeseer, 1994. Citado 3 vezes nas páginas 8, 56 e 57.

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. Citado 6 vezes nas páginas 18, 22, 34, 36, 37 e 90.

NEIGHBORS, J. M. *Software Construction Using Components*. Tese (Ph.D. Thesis) — University of California at Irvine, 1980. Citado na página 19.

NELSON, G. Juno, a constraint-based graphics system. In: ACM. *ACM SIGGRAPH Computer Graphics*. [S.l.], 1985. v. 19, n. 3, p. 235–243. Citado na página 56.

NOVICK, D. et al. Extending direct manipulation in a text editor. p. 127–132, 2002. Citado na página 43.

OLDEVIK, J. Transformation composition modelling framework. *Lecture notes in computer science*, Springer, p. 108–114, 2005. ISSN 0302-9743. Disponível em: <http://cat.inist.fr?aModele=afficheN\cpsidt=17026534>. Citado na página 39.

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. Citado na página 32.

PDE. *Eclipse Documentation - Platform plug-in developer guide*. 2014. http://help.eclipse.org/luna/index.jsp. Acesso em mar, 2013. Citado 4 vezes nas páginas 8, 45, 46 e 47.

POPMA, R. *JET Tutorial Part 1 (Introduction to JET)*. 2004. Eclipse Corner Article. Acesso em: mar 2014. Disponível em: <http://www.eclipse.org/articles/Article-JET-/jet_tutorial1.html>. Citado 5 vezes nas páginas 8, 31, 44, 45 e 54.

POPMA, R. *Jet tutorial Part 2 (Write code that writes code)*. 2004. Eclipse Corner Article. Acesso em: mar 2014. Disponível em: <http://www.eclipse.org/articles/Article-JET/jet_tutorial2.html>. Citado na página 45.

POSSATTO, M. A. *Uma Abordagem para Migração Automática de Código no Contexto do Desenvolvimento Orientado a Modelos*. Dissertação (Mestrado) — Departamento de Computação - Universidade Federal de São Carlos, 2014. Citado 16 vezes nas páginas 8, 10, 15, 18, 23, 49, 58, 60, 62, 70, 87, 88, 89, 90, 91 e 92.

QUINT, V.; VATTON, I. Grif: An interactive system for structured document manipulation. In: *Text Processing and Document Manipulation, Proceedings of the International Conference*. [S.l.: s.n.], 1986. p. 200–312. Citado na página 56.

SALELLES, Z. Interacting with graphic objects. *Dissertation Abstracts International Part B: Science and Engineering[DISS. ABST. INT. PT. B- SCI. & ENG.],*, v. 44, n. 2, 1983. Citado na página 56.

SCHWARZ, H.; EBERT, J.; WINTER, A. Graph-based traceability: a comprehensive approach. *Software & Systems Modeling*, Springer, v. 9, n. 4, p. 473–492, 2010. Citado na página 39.

SEIFERT, M. *Designing round-trip systems by change propagation and model partitioning*. Tese (Doutorado) — PhD thesis, Technische Universität, Dresden, 2011. Disponível em: <http://www.qucosa.de/urnnbn/urn:nbn:de:bsz:14-qucosa-71098>. Citado 3 vezes nas páginas 8, 52 e 53.

SELIC, B. The pragmatics of model-driven development. *IEEE Software*, v. 20, n. 5, p. 19–25, 2003. Citado 2 vezes nas páginas 26 e 27.

SENDALL, S.; KOZACZYNSKI, W. Model transformation: The heart and soul of model-driven software development. *Software, IEEE*, 2003. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1231150>. Citado na página 26.

SHNEIDERMAN, B. Direct manipulation: A step beyond programming languages. In: ACM. *ACM SIGSOC Bulletin*. [S.l.], 1981. v. 13, n. 2-3, p. 143. Citado na página 43.

STALLMAN, R. M. *GNU Emacs manual*. [S.l.]: Free Software Foundation, 2000. Citado 2 vezes nas páginas 41 e 57.

TOLVANEN, J.-P.; KELLY, S. Defining domain-specific modeling languages to automate product derivation: Collected experiences. In: OBBINK, J. H.; POHL, K. (Ed.). *9th International Software Product Line Conference (SPLC-Europe 2005)*. [S.l.]: Springer, 2005. (Lecture Notes in Computer Science, v. 3714), p. 198–209. ISBN 3-540-28936-4. Citado na página 32.

VELOCITY. *Velocity v.1.7 - Apache Velocity Project*. 2013. http://velocity.apache.org/. Acesso em mar, 2014. Citado na página 30.

VISSER, E. WebDSL: A case study in domain-specific language engineering. In: *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. Citado na página 34.

VOELTER, M. *Generic tools, specific languages*. Tese (Doutorado) — TU Delft, Delft University of Technology, 2014. Citado na página 30.

VOELTER, M.; BETTIN, J. Patterns for model-driven software development. In: *Ninth European Conference on Pattern Languages of Programs (EuroPLoP 2004), Irsee, Germany*. [S.l.: s.n.], 2004. p. D3–1–D3–54. Citado na página 90.

W3C. *XML Path Language (XPath) Version 1.0 - W3C Recommendation 16 November 1999*. [S.l.], 1999. Citado na página 68.

WILE, D. Lessons learned from real DSL experiments. *Sci. Comput. Program.*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 51, n. 3, p. 265–290, 2004. ISSN 0167-6423. Citado na página 32.

WIMMER, M.; KRAMLER, G. Bridging grammarware and modelware. In: SPRINGER. *Satellite Events at the MoDELS 2005 Conference*. [S.l.], 2006. p. 159–168. Citado na página 43.

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. Citado na página 34.

WINKLER, S.; PILGRIM, J. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling (SoSyM)*, Springer-Verlag New York, Inc., v. 9, n. 4, p. 529–565, 2010. Citado na página 39.

WOHLIN, C. et al. *Experimentation in software engineering: an introduction*. [S.l.]: Kluwer Academic Publishers, 2000. Citado na página 76.

YOON, Y.; MYERS, B. A. Capturing and analyzing low-level events from the code editor. In: *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools - PLATEAU '11*. [s.n.], 2011. p. 25. ISBN 9781450310246. Disponível em: <http://dl.acm.org.prox.lib.ncsu.edu/citation-.cfm?id=2089155.2089163>. Citado na página 59.

YU, Y. et al. Maintaining invariant traceability through bidirectional transformations. *Proceedings of the 2012 ...*, 2012. Disponível em: <http://dl.acm.org/citation-.cfm?id=2337287>. Citado 2 vezes nas páginas 8 e 55.

ZAYTSEV, V.; BAGGE, A. H. Parsing in a broad sense. In: *Model-Driven Engineering Languages and Systems*. Springer International Publishing, 2014. p. 50–67. Disponível em: <http://www.ii.uib.no/~anya/papers/zaytsev-bagge-models14-parsing.pdf>. Citado 2 vezes nas páginas 8 e 44.

Apêndices

APÊNDICE A – Material do experimento

A.1 Formulário de Caracterização

Nome:_____

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

Formulário de Caracterização

Responda todas as questões abaixo com a maior sinceridade possível.

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

A.2 Roteiro Piloto de Execução de Tarefas

O roteiro de execução piloto encontra-se aqui disponível. Para esta tarefa única temos a versão manual e a versão protótipo que foram utilizadas para a prática do experimento.

A.2.1 Roteiro Piloto Manual

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.
   - Localização: ProjetoTeste/src/generated/daos/derby/Derby*.java

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

   Início: _____ : _____ (hh:mm)

4. Descrição detalhada da tarefa
   4.1. Localizar a posição no arquivo de código da aplicação Web para introduzir a modificação
   4.2. Encontrar o template correspondente que gerará a modificação
   4.3. Aplicar e salvar a alteração no template
   4.4. Run JET Transformation
   4.5. Run index.html
   4.6. Verificar se a modificação foi realizada corretamente

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

   Término: _____ : _____ (hh:mm)

A.2.2 Roteiro Piloto Protótipo

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:
   - Corrigir o erro contido na instrução sql (Select) da classe de persistência do BD Derby para visualizar todas as listas que não estão aparecendo.
   - Localização: ProjetoTeste/src/generated/daos/derby/Derby*.java

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

   Início: _____ : _____ (hh:mm)

4. Descrição detalhada da tarefa
   4.1. Mostrar a view Split JET Editor, e ativar a edição simultânea
   4.2. Localizar a posição no arquivo de código da aplicação Web e introduzir a modificação simultaneamente no arquivo gerado e no template
   4.3. Salvar ambos os arquivos
   4.4. Run “JET Transformation” (desligar a edição simultânea antes)
   4.5. Run “index.html”
   4.6. Verificar se a modificação foi realizada corretamente

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

   Término: _____ : _____ (hh:mm)

A.3 Roteiro da execução de tarefas

O roteiro de execução das quatro tarefas encontra-se aqui disponível. Para cada tarefa temos a versão manual e a versão protótipo que foram utilizadas no experimento.

A.3.1 Roteiro de execução da Tarefa 1 Manual

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”
   - Localização:
     - 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. Localizar a posição no arquivo de código da aplicação Web para introduzir a modificação
   4.2. Encontrar o template correspondente que gerará a modificação
   4.3. Aplicar e salvar a alteração no template
   4.4. Run JET Transformation
   4.5. Run index.html
   4.6. Verificar se a modificação foi realizada corretamente

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

   Término: _____ : _____ (hh:mm)

A.3.2 Roteiro de execução da Tarefa 1 Protótipo

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”
   - Localização:
     - ProjetoTeste/WebContent/gen/uploadUserCommentForm.jsp
     - ProjetoTeste/WebContent/admin/moderation.jsp

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

   Início: _____ : _____ (hh:mm)

4. Descrição detalhada da tarefa
   4.1. Mostrar a view Split JET Editor, e ativar a edição simultânea
   4.2. Localizar a posição no arquivo de código da aplicação Web e introduzir a modificação simultaneamente no arquivo gerado e no template
   4.3. Salvar ambos os arquivos
   4.4. Run “JET Transformation” (desligar a edição simultânea antes)
   4.5. Run “index.html”
   4.6. Verificar se a modificação foi realizada corretamente

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

   Término: _____ : _____ (hh:mm)

A.3.3 Roteiro de execução da Tarefa 2 Manual

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 Messages", substituindo para o texto "Adicionar.....".
   - Localização: 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. Localizar a posição no arquivo de código da aplicação Web para introduzir a modificação
   4.2. Encontrar o template correspondente que gerará a modificação
   4.3. Aplicar e salvar a alteração no template
   4.4. Run JET Transformation
   4.5. Run index.html
   4.6. Verificar se a modificação foi realizada corretamente

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

   Término: _____ : _____ (hh:mm)

A.3.4 Roteiro de execução da Tarefa 2 Protótipo

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 Messages", substituindo para o texto "Adicionar.....".
   - Localização: ProjetoTeste/src/generated/resources/messages.properties

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

   Início: _____ : _____ (hh:mm)

4. Descrição detalhada da tarefa
   4.1. Mostrar a view Split JET Editor, e ativar a edição simultânea
   4.2. Localizar a posição no arquivo de código da aplicação Web e introduzir a modificação simultaneamente no arquivo gerado e no template
   4.3. Salvar ambos os arquivos
   4.4. Run “JET Transformation” (desligar a edição simultânea antes)
   4.5. Run “index.html”
   4.6. Verificar se a modificação foi realizada corretamente

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

   Término: _____ : _____ (hh:mm)

A.3.5 Roteiro de execução da Tarefa 3 Manual

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>.
   - Localização:
     - 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. Localizar a posição no arquivo de código da aplicação Web para introduzir a modificação
   4.2. Encontrar o template correspondente que gerará a modificação
   4.3. Aplicar e salvar a alteração no template
   4.4. Run JET Transformation
   4.5. Run index.html
   4.6. Verificar se a modificação foi realizada corretamente

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

   Término: _____ : _____ (hh:mm)

A.3.6 Roteiro de execução da Tarefa 3 Protótipo

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>.
   - Localização:
     - 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. Descrição detalhada da tarefa
   4.1. Mostrar a view Split JET Editor, e ativar a edição simultânea
   4.2. Localizar a posição no arquivo de código da aplicação Web e introduzir a modificação simultaneamente no arquivo gerado e no template
   4.3. Salvar ambos os arquivos
   4.4. Run “JET Transformation” (desligar a edição simultânea antes)
   4.5. Run “index.html”
   4.6. Verificar se a modificação foi realizada corretamente

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

   Término: _____ : _____ (hh:mm)

A.3.7 Roteiro de execução da Tarefa 4 Manual

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
   - Localização:
     - 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. Localizar a posição no arquivo de código da aplicação Web para introduzir a modificação
   4.2. Encontrar o template correspondente que gerará a modificação
   4.3. Aplicar e salvar a alteração no template
   4.4. Run JET Transformation
   4.5. Run index.html
   4.6. Verificar se a modificação foi realizada corretamente

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

   Término: _____ : _____ (hh:mm)

A.3.8 Roteiro de execução da Tarefa 4 Protótipo

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
   - Localização:
     - ProjetoTeste/WebContent/admin/EditProjetoDePesquisa.jsp
     - ProjetoTeste/WebContent//EditPublicacao.jsp

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

   Início: _____ : _____ (hh:mm)

4. Descrição detalhada da tarefa
   4.1. Mostrar a view Split JET Editor, e ativar a edição simultânea
   4.2. Localizar a posição no arquivo de código da aplicação Web e introduzir a modificação simultaneamente no arquivo gerado e no template
   4.3. Salvar ambos os arquivos
   4.4. Run “JET Transformation” (desligar a edição simultânea antes)
   4.5. Run “index.html”
   4.6. Verificar se a modificação foi realizada corretamente

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

   Término: _____ : _____ (hh:mm)

A.4 Questionário de Avaliação

Questionário

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 *view* do editor foram úteis?

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