Geradores de Artefatos: Implementação e Instanciação de Frameworks

Luiz Paulo Alves Franca
Programare Informática
luizp@ism.com.br

Arndt von Staa
Departamento de Informática, PUC-Rio
arndt @ inf.puc-rio.br

Resumo

Existem muitas similaridades entre um gerador de artefatos e um framework. Sob uma ótica de framework, as partes fixas do artefato gerado corresponde aos frozen-spots, já as partes variáveis corresponde aos hot-spots. A atividade de configuração dos hot-spots de um framework corresponde ao fornecimento da especificação do artefato que vai ser gerado. Já a instanciação de um framework, no contexto do gerador, corresponde aos mecanismos de geração. Em função das semelhanças encontradas entre geradores e frameworks, este artigo procura detalhar o processo de construção de geradores sob o ponto de vista do desenvolvimento de frameworks.

Palavras chave: *Framework, Ferramentas para engenharia de software, Geração de artefatos, Meta-geradores, Processo de desenvolvimento de geradores, Reusabilidade*

Abstract

There are many similarities between an Artifact Generator and a Framework. The fixed part of generated artifacts is similar to a frameworks' frozen-spot, and the variable part corresponds to a frameworks' hot-spots. A framework is customized through configuration of its hot-spots, in the context of a generator, the customization is reached by means the specification of the artifact. The generation mechanism of a generator is analogous to the framework instantiation. In this article, we use these similarities to explain in detail a generator construction process.

Keywords: *Artifact generation, CASE tools, Frameworks, Generator development process, Meta-generators, Reusability*

1 – Introdução

Visando tornar a tecnologia de geração de aplicações acessível para um maior número de profissionais de desenvolvimento de software, elaboramos um novo processo de construção de geradores [9]. Neste trabalho, adotamos o termo *gerador de artefato* no lugar de gerador de aplicação. Segundo Staa [27]:

"Um artefato é qualquer item criado como parte da definição, manutenção ou utilização de um processo de software. Inclui entre outros, descrições de processo, planos, procedimentos, especificações, projeto de arquitetura, projeto detalhado, código, documentação para o usuário. Artefatos podem ou não ser entregues a um cliente ou usuário final".

Por esta definição, uma aplicação pode ser considerada um tipo de artefato que é entregue ao cliente final. Geradores de artefatos podem ser utilizados para produzir tanto a aplicação final, como a sua documentação. Podem ser utilizados também para gerar componentes de um sistema, como, por exemplo, o controle da interface com o usuário em um sistema de interface gráfica.

Um gerador de artefatos é uma ferramenta de software que produz um artefato a partir da sua especificação de alto nível (Fig 1). Um processo de desenvolvimento baseado em geradores preconiza que a manutenção do artefato gerado seja sempre e integralmente realizada na sua especificação e não nos seus arquivos de implementação (ex.: arquivos fonte de um programa) [13].

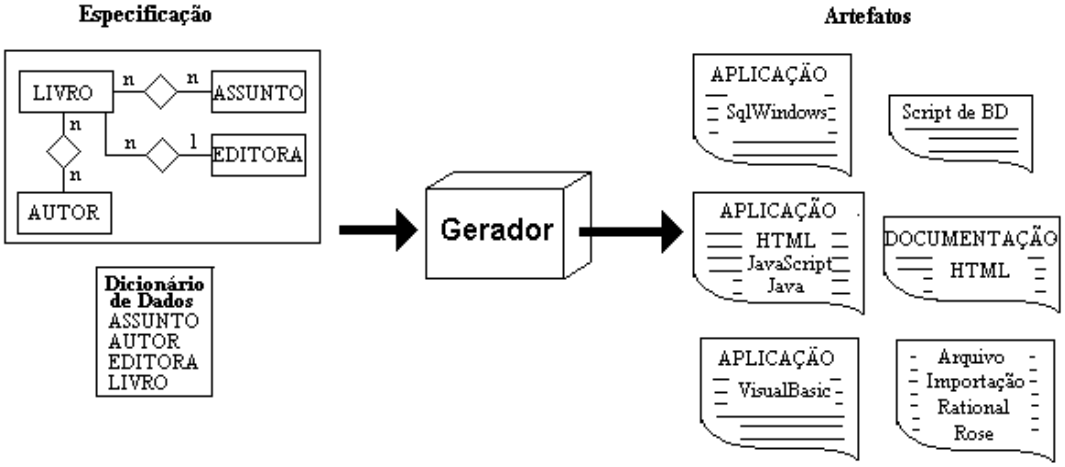

Figura 1. Visão esquemática de um gerador de artefatos

No processo de construção proposto, o gerador não é construído através de um processo de desenvolvimento tradicional e sim, utilizando um processo evolutivo, através de ciclos sucessivos de desenvolvimento aplicados sobre um exemplo do artefato a ser gerado. Cada ciclo produz uma versão executável do gerador. O artefato gerado pela última versão do gerador pode ser utilizado como exemplo para construção da próxima versão do gerador. À medida que os ciclos vão se repetindo, as versões geradas vão ficando mais completas e mais genéricas.

Dentro do contexto de reúso de software, o processo proposto é bastante similar ao processo de construção de *frameworks*. Geralmente um *framework* é resultado da experiência acumulada no desenvolvimento de aplicações no domínio do *framework* [4]. Schmid em [23, 24, 25] apresenta um processo de construção de *frameworks* baseado em ciclos de des-

envolvimentos, onde cada ciclo acrescenta novas variabilidades ou funcionalidades fixas à estrutura de classes do *framework* através de transformações de generalização. O resultado do ciclo de desenvolvimento ou é um protótipo ou uma nova versão do *framework.*

Em funções das semelhanças encontradas entre Geradores e Frameworks, este artigo procura detalhar o processo de construção de geradores sob o ponto de vista do desenvolvimento de *frameworks.* O artigo está organizado na seguinte forma: na seção 2 são apresentados os conceitos relaconados a *Frameworks*. A seção 3 são discutidas as similaridades entre Geradores e Frameworks. A seção 4 aborda a questão do mecanismo de instaciamento (geração) . O processo de construção é apresentado na seção 5. A seção 6 descreve os experimentos realizados. As conclusões são apresentadas na seção 7.

2 – Frameworks

Johnson em [15] apresenta *frameworks* e geradores como sendo técnicas que fazem tanto o reúso de *design* como de código. O termo *framework* inicialmente estava associado ao conceito de biblioteca de classes reutilizáveis. Mais recentemente, o termo passou a ser mais abrangente, significando que um *framework* é qualquer solução incompleta que pode ser completada através da instanciação e, desta forma, possibilitando a geração de mais de uma aplicação dentro do domínio-alvo do *framework* [8]. Esta definição também pode ser aplicada para um gerador de artefatos. Em função destas semelhanças procuramos buscar na literatura referente a *framework* [2, 3, 4, 12, 15, 19, 23, 24] conceitos que pudessem ser aplicados na construção de geradores de artefatos.

Em relação à estrutura de um *framework,* as suas partes variáveis são chamadas de *hot-spots* (pontos de flexibilização) [19], já as fixas, são chamadas de *frozen-spots*. Os *frameworks* em que os *hot-spots* são implementados através da especialização de classes abstratas são chamados de *whitebox frameworks*. Já os *frameworks* em que os *hot-spots* são implementados através da composição de componentes são chamados de *blackbox frameworks* (Fig. 2).

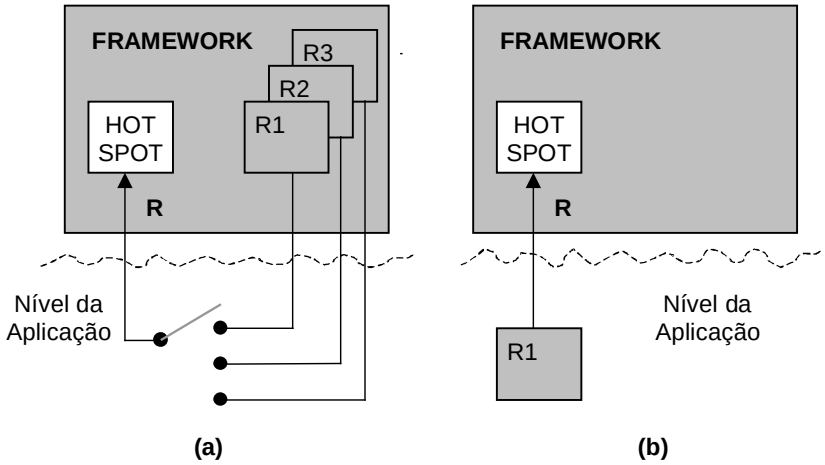

Figura 2. Mecanismo de *hot-spot*: (a) no caso do *black-box,* o usuário tem que escolher uma das classes fornecidas pelo *framework* (b) no caso do *white-box,* o usuário tem que construir uma classe que vai ser usada pelo *framework* [25]

Para utilizar os *whitebox frameworks*, o desenvolvedor precisa conhecer a estrutura do *framework*, dependendo da qualidade da documentação e da complexidade do *framework*, este aprendizado pode ser bastante longo. A tendência é que as repetidas especializações feitas em um *whitebox framework* possam indicar oportunidades de criação de parâmetros *default* para o uso em um *framework* do tipo *blackbox*. Desta forma, a evolução e a estabilização podem transformar, ao longo do tempo, um *whitebox framework* num *blackbox framework*.

Responsabilidades comuns (*common responsibilities*) e tempo de ligação são duas características importantes para a definição do tipo de implementação de um *hot-spot*. Responsabilidades comuns correspondem aos serviços que qualquer classe que implemente o *hot-spot* deverá oferecer para o restante do *framework*. Esta característica define as *commonalties* do *hot-spot*. Já o tempo de ligação, define o momento em que as características do *hot-spot* são selecionadas e associadas. Esta ligação pode ser feita pelo desenvolvedor durante a customização do *framework* ou pelo usuário durante a execução da aplicação, neste caso, a ligação pode ser feita múltiplas vezes.

3 – Gerador de Artefatos = Frameworks

Existem muitas similaridades entre um Gerador de Artefatos e um *framework*. Sob um ótica de framework, as partes fixas do artefato gerado correspondem aos *frozen-spots*, já as partes variáveis correspondem aos hot-spots. A atividade de configuração dos *hot-spots* de um *framework* corresponde ao fornecimento da especificação do artefato que vai ser gerado. Já a instanciação de um *framework*, no contexto do gerador, corresponde aos mecanismos de geração. Em relação à implementação, enquanto um *framework* utiliza mecanismos da Orientação a Objetos, um Gerador pode utilizar outros mecanismos [7]. Em função destas similaridades, o processo de construção de geradores de artefatos que foi proposto em [9] , adotou algumas abordagens da área de *frameworks.*

Considerando que a estrutura de um gerador de artefatos é uma forma de *framework*, no qual as partes variáveis do programa de descrição do artefato são os *hot-spots*, e as partes fixas são os *frozen-spots*, observamos que as heurísticas que adotamos para identificar as partes variáveis possuem semelhanças com algumas propostas de identificação de *hot-spots* (atividade de *hot-spot mining*) [20]. Em nosso trabalho, o ponto de partida para esta identificação, foi a análise das modificações realizadas durante a evolução dos protótipos através do reúso do tipo “cortar/colar/modificar”. Esta análise mostrou que grande parte do código da aplicação permanecia inalterada, ou seja, a quantidade de *frozen-spots* era bastante superior à quantidade de *hot-spots*. Em [18] é relatado um exemplo de gerador de aplicações onde as variabilidades correspondem a apenas 16% do código. Larnegan e Grasso em [16] já assinalavam o grande grau de similaridade entre as aplicações desenvolvidas para área comercial, mostrando que uma grande parte do trabalho era redundante, e apenas uma parcela correspondia à implementação de novas funcionalidades.O resultado final desta análise foi a classificação dos *hot-spots* encontrados.

Os *hot-spots* encontrados foram classificados segundo a proposta de [5, 6]. Nesta proposta, os *hot-spots* são classificados segundo o seu método de adaptação e do grau de apoio fornecido pelo *framework* para esta adaptação. Escolhemos esta classificação pois, dentre

as propostas apresentadas na literatura ela é independente do *design* ou da implementação OO.

Em relação ao método de adaptação existem cinco métodos [5, 6]:

- **Habilitar uma característica.** Consiste em habilitar uma característica que existe no *framework* mas não é parte de sua implementação *default*;
- **Desabilitar uma característica.** Consiste em desabilitar uma característica da implementação *default* do *framework*;
- **Substituir uma característica.** Consiste em desabilitar uma característica e colocar uma nova característica em seu lugar;
- **Aumentar uma característica.** Consiste em estender uma característica sem alterar o fluxo normal de controle. Esta alteração pode interceptar o fluxo de controle existente, executar alguma ação necessária e retornar o controle de volta para o *framework*;
- **Adicionar uma característica.** Envolve a adição de uma nova característica ao *framework*.

Em relação ao grau de apoio fornecido, existem três tipos [5, 6]:

- **Opção.** O desenvolvedor seleciona um componente pré-definido disponível numa biblioteca;
- **Pattern.** As adaptações estão associadas a instanciamentos pré-definidos que dependem de informações fornecidas pelo usuário da aplicação;
- **Ilimitado** (*open-ended*). As adaptações são realizadas sem apoio do *framework*. O desenvolvedor tem liberdade para alterar o *framework* adicionando características não previstas no *framework* original.

Em relação à classificação, no que diz respeito ao grau de apoio, como os geradores construídos pelo nosso processo fazem a geração a partir de informações extraídas da especificação fornecida pelo usuário, temos que a maioria dos *hot-spots* encontrados são do tipo "*pattern*". Caso o gerador apresente a possibilidade de inclusão de rotinas implementadas pelo usuário do gerador, consideremos que os pontos de inclusão das rotinas correspondem a *hot-spots* do tipo "ilimitado". A seguir, apresentaremos os tipos de *hot-spots* encontrados empiricamente em nosso trabalho.

– **Hot-Spot: Modo de Adaptação: Substituir – Grau de Apoio: Pattern**

Durante a criação de um programa a partir de um programa similar utilizando uma evolução do tipo "cortar/colar/modificar", várias partes do programa novo correspondem à substituição de um trecho de código do programa similar por um outra trecho pertinente ao contexto do novo programa. Estes pontos podem ser classificados como sendo *hot-spot* do tipo "substituir".

– **Hot-Spot: Modo de Adaptação: Habilitar – Grau de Apoio: Pattern**

Durante a criação de um programa a partir de um programa similar utilizando uma evolução do tipo "cortar/colar/modificar", observa-se que, utilizando um mesmo arquivo como

base para o reúso, certos pontos são modificados para apenas alguns dos novos arquivos criados. Geralmente, os pontos modificados correspondem a funcionalidades presentes somente em alguns dos programas criados. Estes pontos podem ser classificados como sendo *hot-spot* do tipo "habilitar".

– **Hot-Spot: Modo de Adaptação: Aumentar – Grau de Apoio: Ilimitado**

Uma rotina de escape é um fragmento de código redigido pelo usuário do gerador, vinculado a algum elemento da especificação e incorporado em pontos específicos do artefato gerado. Tal fragmento de código é incorporado no artefato gerado sem passar por qualquer tipo de transformação. Os ponto de expansão de uma rotina de escape, pode ser, por exemplo, antes de realizar a gravação de um registro, após a recuperação de um registro, na crítica de um campo de uma tela de entrada de dado, etc. Estes pontos podem ser classificados como sendo *hot-spot* do tipo "aumentar/ilimitado". A responsabilidade da qualidade da rotina de escape é do próprio desenvolvedor usuário do gerador. Em geral, o gerador não possui mecanismos para validar a rotina de escape, que é tratada pelo gerador como se fosse um texto livre.

4 – Instanciação do Framework

Em nosso trabalho o ambiente de geração utilizado, i.e., o ambiente de instanciação do *framework*, pode ser qualquer ferramenta CASE que disponibilize mecanismos de customização dos seu módulo de entrada de dados e de manipulação do repositório de dados da ferramenta (Fig. 3). A adoção de uma ferramenta CASE simplificou o processo de construção, eliminando a necessidade de codificação da infra-estrutura de geração. O componente de entrada (módulo de edição da especificação) utiliza os editores de diagramas do CASE e tem suas telas de dicionário de dados definidas através de programas codificados na linguagem nativa da ferramenta. O componente de armazenamento do gerador corresponde ao próprio repositório de dados do CASE. Já o componente de saída também é definido através de programas codificados na linguagem nativa da ferramenta.

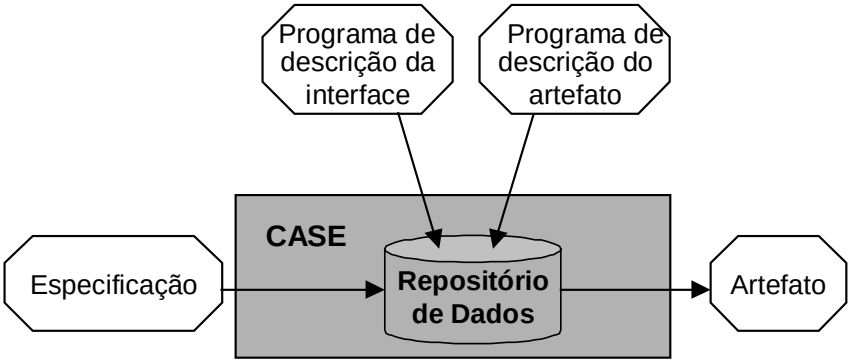

Figura 3. Ferramenta CASE como ambiente de geração

Em [14] é apresentada uma lista das características que uma ferramenta CASE deve ter para ser utilizada como ambiente de geração:

- Possuir um repositório contendo todos os dados dos modelos que estão sendo utilizados;

- Disponibilizar interfaces que permitam amplo acesso aos meta-dados do repositório;
- Possuir mecanismos para editar os meta-modelos disponíveis, adicionando ou modificando seus meta-dados;
- Permitir a criação e a execução de *scripts* que exploram e manipulam o conteúdo do repositório.

Em nosso trabalho utilizamos o meta-CASE Talisman [26] que possui as características acima descritas. Em [14, 17] são descritos trabalhos nos quais é utilizada a ferramenta Rational Rose [21] como ambiente de geração.

5 – Processo de Construção

Cada *hot-spot* identificado durante a atividade de *hot-spot mining* deve ser substituído por uma regra de transformação que manipule informações da especificação. Esta substituição é feita durante a codificação do programa de descrição do artefato (Fig. 4) através de um processo de generalização de um artefato que serve de exemplo para o gerador a ser construído. A regra de transformação é implementada através de funções codificadas na linguagem nativa do CASE. Nos geradores que construímos, a complexidade destas funções era bem baixa. Geralmente as funções retornavam valores dos atributos dos objetos do repositório de dados da ferramenta. Estes valores antes de serem retornados passavam por algum tipo de formatação (ex.: concatenação de prefixos ou sufixos).

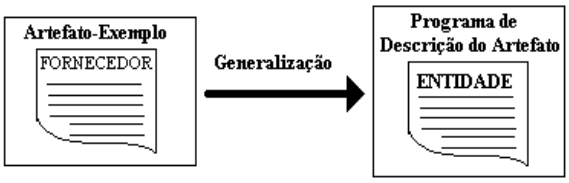

Figura 4. Visão esquemática da generalização

A tentativa de realizar manualmente a substituição dos *hot-spots* por suas regras de transformação utilizando apenas as facilidades de edição de um editor de texto, mostrou-se bastante ineficiente. Esta atividade envolvia a substituição de muitos pontos e deveria ser feita com atenção para evitar a "desformatação" das partes fixas (*frozen-spots)* não envolvidas com os *hot-spots*. Em função da natureza repetitiva destas substituições, resolvemos automatizá-la.

Ao invés de substituir o *hot-spot* identificado no código do artefato-exemplo diretamente pela sua regra de transformação correspondente, ele é substituído por um *tag* que depois será substituído automaticamente pela regra de transformação. O arquivo contendo o código fonte original acrescentado com *tags* é chamado de arquivo de *meta-descrição* do artefato (Fig 5). O arquivo de meta-descrição do artefato reduz o problema de descasamento de impedância [14, 17] entre os domínios da especificação e da implementação. Neste arquivo, os *tags* incluídos servem para marcar as dependências da especificação. Um arquivo de meta-descrição do artefato está dividido em duas partes:

- **Cabeçalho**: Contém a lista de definição dos *tags* presentes no arquivo. A lista é formada pelo par (*tag*, regra de transformação). A regra de transformação associada ao *tag* corresponde ao código da chamada da função que vai aplicar a regra de transformação.
- **Corpo**: Contém o código original marcado com os *tags*.

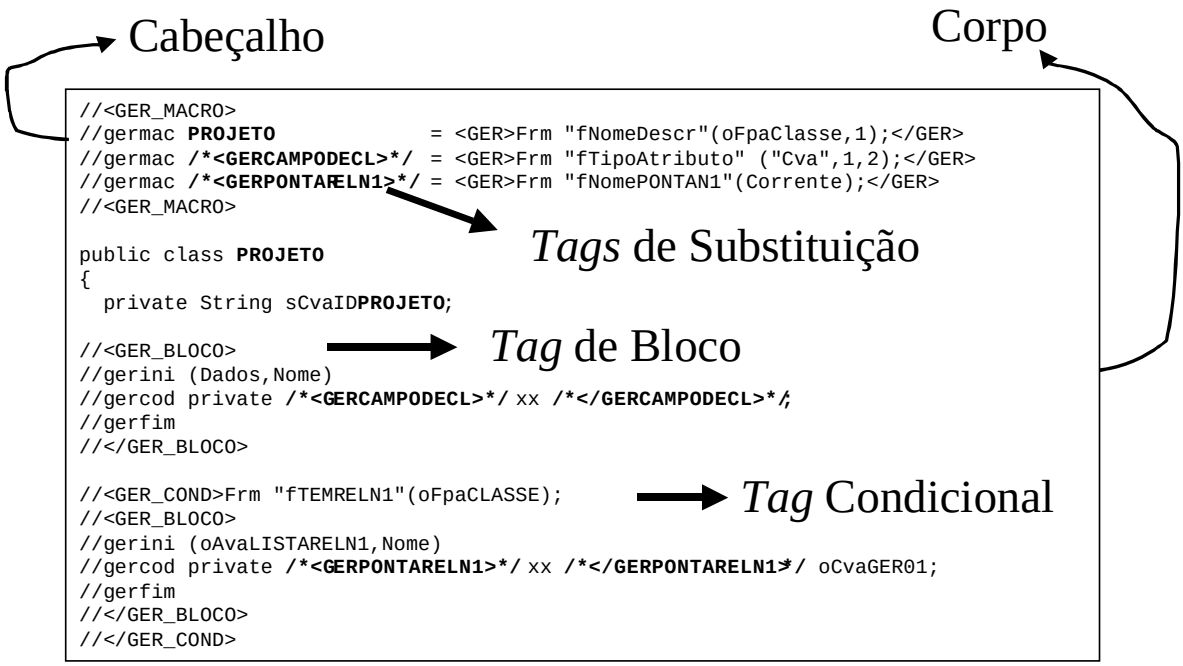

Figura 5. Exemplo de um arquivo de meta-descrição do artefato

O utilitário GERDESCR (Fig. 6) foi desenvolvido para ler o arquivo de meta-descrição do artefato e substituir cada *tag* pela regra de transformação correspondente escrita na linguagem do CASE. O resultado final deste utilitário é o programa de descrição do artefato.

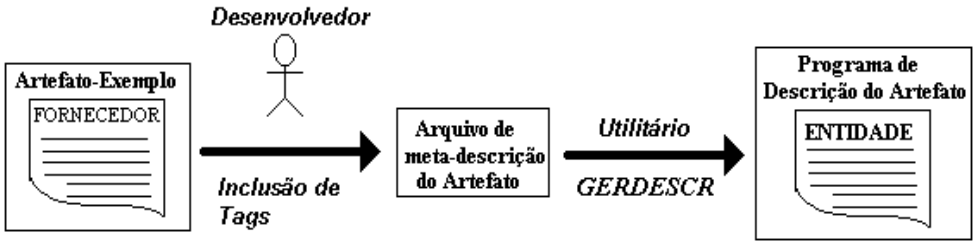

Figura 6 Geração automática do programa de descrição do artefato

Foram criado os seguintes tipos de *tag:*

- ***Tag* de Substituição:** É definido no cabeçalho do arquivo de meta-descrição do artefato. Este *tag* foi utilizado para marcar os *hot-spots* do tipo substituir / *pattern* O utilitário GERDESCR substitui toda ocorrência do *tag* dentro do corpo do arquivo pela chamada da função associada ao *tag*. Existem dois tipos de *tags* de substituição. O primeiro tipo é utilizado para implementar a substituição de todas as ocorrências de uma *string* dentro do corpo do arquivo. Neste caso, a própria *string* a ser substituída funciona como *tag*. Já o *tag* do segundo tipo precisa receber um nome e pode ser utilizado quando a substi-

tuição envolve uma ou mais *strings*, como por exemplo a lista de parâmetros de uma função.

- ***Tag* de Bloco:** Corresponde à repetição de um trecho de código para cada elemento de um conjunto (ex.: atributos de uma classe). Este *tag* foi também utilizado para marcar os *hot-spots* do tipo substituir / *pattern*. O *tag* de bloco é inserido diretamente no corpo do arquivo, não precisando estar definido no cabeçalho do arquivo. O utilitário GERDESCR vai repetir o trecho definido dentro do *tag* para cada elemento do conjunto. O trecho do bloco pode conter *tags* de substituição.

- ***Tag* Condicional:** A regra de transformação referente a um *tag* condicional corresponde a uma função que retorna Verdadeiro ou Falso. O *tag* condicional é sempre associado a um *tag* de bloco. Durante a geração de código, se a função condicional for verdadeira, a geração do código referente ao bloco seguinte ao *tag* condicional será executada. O *tag* de condicional é inserido diretamente no corpo do arquivo, não precisando estar definido no cabeçalho do arquivo. Este *tag* foi utilizado para marcar os *hot-spots* do tipo habilitar / *pattern* e do tipo aumentar / ilimitado.

A geração automática do programa de descrição do artefato, além das vantagens inerentes à automação de uma atividade, trouxe os seguintes benefícios para o processo de construção:

- ***Tratamento dos frozen-spot :*** Como foi visto na seção anterior, a maior parte do código dos protótipos corresponde a *frozen-spot*. A regra de transformação associada ao *frozen-spot* apenas reproduz o código original. Para o utilitário GERDESCR, os *frozen-spot* são todos os trechos de código que não estejam localizados entre *tags*. O utilitário GERDESCR substitui o *frozen-spot* por um comando que o re-escreve (Fig. 7).

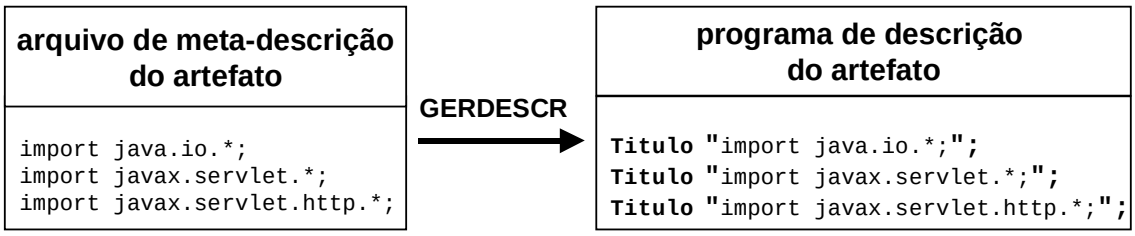

Figura 7. Exemplo da transformação de um *frozen-spot*

- ***Rastreamento das substituições :*** Quando os *hot-spots* são substituídos diretamente por sua regra de transformação através de um editor de texto, a referência entre a regra e o seu *hot-spot* de origem é perdida. No caso da utilização do arquivo de meta-descrição, o rastreamento das substituições pode ser feito através da leitura do cabeçalho do arquivo que contém a lista dos *tags* e de suas regras correspondentes (Figuras 8 e 9).

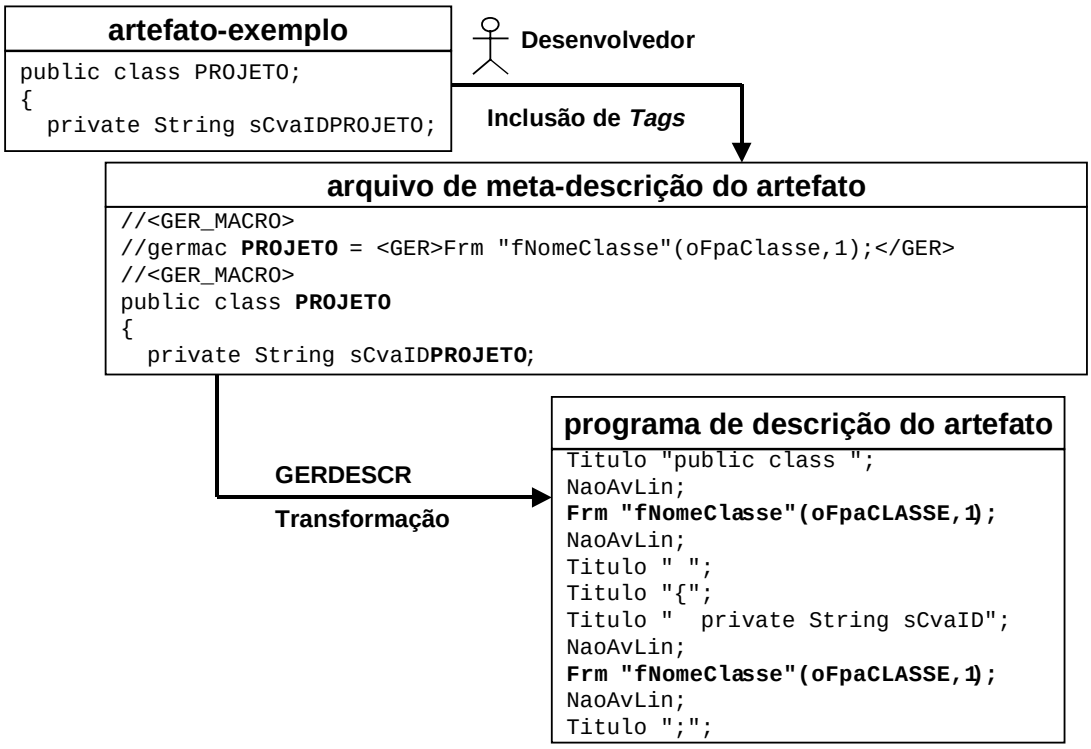

Figura 8. Exemplo da transformação de um *tag* de Substituição

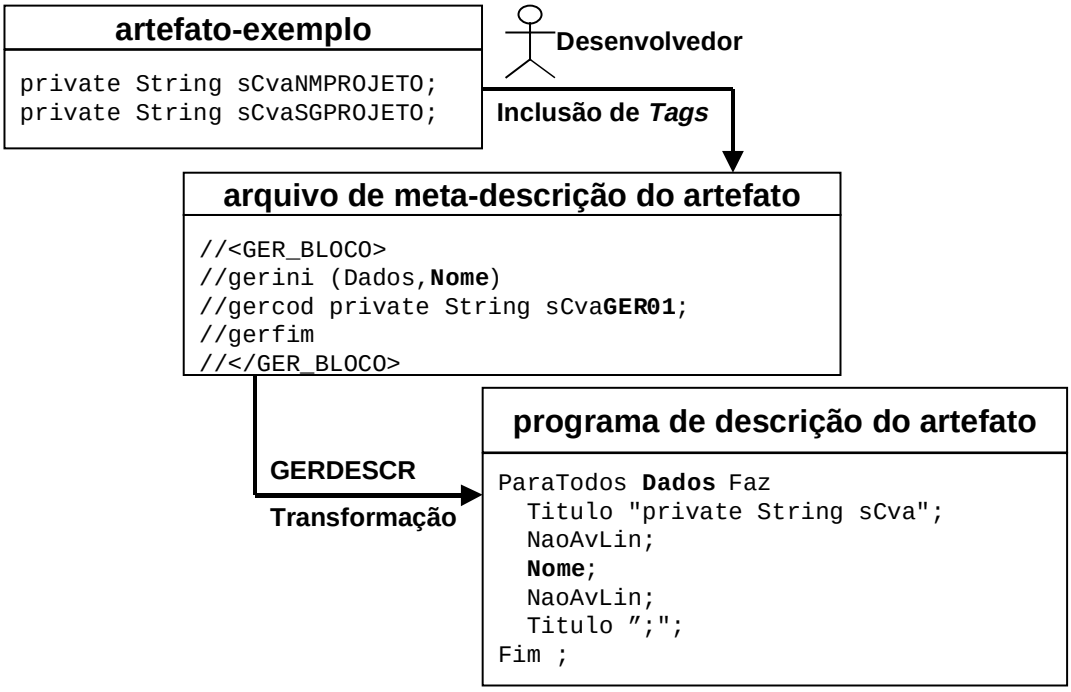

Figura 9. Exemplo da transformação de um *tag* de Bloco

6 – Experimentos

O processo proposto de construção de gerador de artefatos foi avaliado e ajustado através da realização de alguns experimentos:

Experimentos	
Objetivo	**Descrição**
Definição das etapas do processo de construção	Envolveu a construção do ProtoBD prototipador de modelo de dados [11]. A aplicação gerada executa a manutenção das tabelas e relacionamentos definidos no modelo de dados. A arquitetura da aplicação gerada era em três camadas, compostas por programas escritos em Java, JavaScript, e HTML [1]. O gerador criado era composto por treze arquivos de meta-descrição de artefato.
Verificar independência da linguagem de representação	No ProtoBD, o diagrama de Entidade-Relacionamento foi utilizado para especificar a aplicação, e o diagrama de fluxo de dados foi utilizado para definir a estrutura de menus da aplicação gerada.
Verificar independência da linguagem do artefato gerado	Foram feitos dois experimentos para estender o ProtoBD para gerar aplicações utilizando a arquitetura cliente-servidor com programas escritos em VisualBasic e em SqlWindows. Foi feito também um gerador de arquivos .MDL de importação do CASE Rose da Rational [21].
Extensão pelo usuário	Verificou a flexibilidade do gerador em permitir a inclusão de rotinas de escape. A especificação é alterada para permitir a digitação da rotina de escape. A rotina de escape é associada a um novo atributo da entidade. Durante a geração, caso a rotina de escape tiver sido fornecida, o gerador aplica a regra de transformação que expande a rotina de escape.
Verificar o processo de manutenção do gerador	Uma série de alterações estruturais foram feitas na versão original do ProtoBD. As alterações envolveram a criação de novos atributos na especificação, a alteração dos arquivos de meta-descrição e a criação de novos arquivos de meta-descrição. O processo de manutenção proposto foi capaz de realizar a evolução do gerador.
Verificar limitações na geração	Envolveu a utilização do ProtoBD para diferentes modelos de dados. Os modelos variavam tanto em número como em relação ao tipo das entidades e dos relacionamentos. Para uma aplicação de gerência de projeto de software composta por 16 entidades e 21 relacionamentos foi gerada uma aplicação de cerca de 99.000 linhas distribuídas em 650 arquivos organizados em 80 diretórios. Cabe observar que a aplicação tornou-se operacional em menos de um dia de trabalho. Os experimentos realizados neste estudo de caso serviram para mostrar que o gerador não possuía limitações quanto à diversidade e ao tamanho do modelo de dados.
Verificar independência ao tipo de artefato gerado.	Foi criado um gerador de documentação HTML de um modelo de dados. Este experimento serviu para avaliar a independência do processo de construção em relação ao tipo da aplicação gerada (manutenção, exploração, etc).

7 – Conclusão

Roberts e Johnson [22] apresentam o processo de desenvolvimento de *framework* sob a ótica da evolução de um *framework*. O modelo descreve a evolução de um *framework* a partir de uma implementação simples baseada na técnica dos três exemplos, até uma implementação sofisticada que utiliza linguagens específicas para um domínio. Analogamente em nosso trabalho a utilização de um artefato-exemplo como ponto de partida do processo de construção foi importante para delimitar o escopo do gerador a ser construído, evitando o problema da complexidade da construção de geradores extremamente genéricos. Um outro benefício da utilização do artefato-exemplo é a promoção do reúso. Uma boa parte do código do gerador corresponde a trechos inalterados (*frozen-spots*) dos arquivos fontes do artefato-exemplo.

Os *tags* de substituição, de bloco e condicional foram criados para implementar a generalização dos *hot-spots* identificados nos arquivos fontes do artefato-exemplo. A necessidade de codificação manual dos programas de descrição do artefato é eliminada através do utilitário GERDESCR que lê o arquivo de meta-descrição do artefato e substitui cada *tag* pela regra de transformação correspondente.

O arquivo proposto de meta-descrição do artefato reduz o problema de impedância [14, 17] entre os domínios de especificação e de implementação. Neste arquivo, os *tags* de transformação marcam as dependências da especificação. O restante do arquivo corresponde aos frozen-spots, que serão simplesmente replicados no artefato-gerado. Nos experimentos realizados, os três tipos de tags de transformação (substituição, bloco e condicional) criados foram suficientes para criação do arquivo de meta-descrição do artefato. O arquivo de meta-descrição também serviu para facilitar a evolução do gerador. Nos experimentos envolvendo a evolução de um gerador, o arquivo de meta-descrição possibilitou uma rápida identificação dos impactos das modificações do artefato-exemplo.

A extensão do gerador através da inclusão de rotinas de escape é feita de forma organizada. Primeiro é criado um novo campo no dicionário de dados para armazenar o conteúdo da rotina de escape. Em seguida, no arquivo de meta-descrição do artefato é incluído um *tag* condicional para marcar o ponto onde a rotina de escape será expandida.

Os experimentos realizados nos mostraram que as seguintes questões relacionadas a aplicação da abordagem de Frameworks no contexto de geradores merecem um estudo mais profundo:

- Documentação do framework. Aumentar a legibilidade do gerador e do artefato gerado, no sentido de reduzir a curva de aprendizado do novo desenvolvedor responsável pela evolução do gerador.

- Verificação da consistência do hot-spot. Muitas vezes a instanciação de um hot-spot envolve a modificação de difentes pontos do artefato a ser gerado. A identificação destes pontos por ser feita de forma manual pelo desenvolvedor, pode ser feita de forma incompleta, acarretando erros no artefato a ser gerado.

– Automação da atividade de hot-spot mining. Nos experimentos realizados, a ferramenta para o hot-spot mining era um simples editor de texto. Imaginamos que facilidades do tipo pesquisa por estruturas, lista dos hot-spots já encontrados, diferenciação dos frozen-spots e dos hot-spots, poderiam ser adicionados a ferramenta de edição.

8 – Referências Bibliográficas

[1] Barosa, R.; ”Usando Ferramentas 4GL para Gerar Código Java”; *Developers' Magazine (44);* 2000; pp 10-12

[2] Baumer, D.; Gryczan, G.; Knoll, R.; Lilienthal, C.; Riehle, D.; Züllighoven, H. “Framework Development for Large Systems”; *Communications of the ACM*; 40(10); 1997; pp 52-59

[3] Brugali, D.; Menga, G.; Aarsten, A.; “The Framework Life Span”; *Communications of the ACM*; 40(10); 1997; pp 65-68

[4] Codenie, W.; Hondt, K.; Steyaert, P.; Vercammen, A.; “From Custom Applications to Domain-Specific Frameworks”; *Communications of the ACM*; 40(10); 1997; pp 70-77

[5] Froehlich, G.; Hoover, H.; Liu, L.; Sorenson, P.; “Hooking into Object-Oriented Application Frameworks”; *Proceedings of International Conference on Software Engineering*; 1997; pp 491-501

[6] Froehlich, G.; Hoover, H.; Liu, L.; Sorenson, P.; ”Reusing Hooks”; In Mohamed Fayad and Douglas C. Schmidt (Eds.); *Building Application Frameworks: Object-Oriented Foundations of Framework Design*; John-Wiley & Sons; 1999; pp 219-236

[7] Fileto, R.; Meira, C.; Costa, C.; Masshurá, S.; “A Construção de um Gerador de Programas Aplicativos segundo Conceitos de Análise de Domínios”; *Anais do X Simpósio Brasileiro de Engenharia de Software;* Ed. SBC; 1996; pp 119-135

[8] Fontoura, M.F.; *Uma Abordagem Sistemática para o Desenvolvimento de Frameworks*; Tese de Doutorado; Depto. Informática PUC-Rio; 1999

[9] Franca, L.P.A.; *Um Processo para Construção de Geradores de Artefatos*; Tese de Doutorado; Depto. Informática PUC-Rio; 2000

[10] Franca, L.P.A.; Staa A.; Fonte, H.; “Um Modelo de Classes para uma Ambiente de Geração de Programas de Medição de Software Baseados na Web”; *Anais do XIII Simpósio Brasileiro de Engenharia de Software;* Ed. SBC; 1999; pp 225-236

[11] Franca, L.P.A.; Staa A.; Fonte, H.; “ProtoBD: Prototipador de Modelo de Dados”; *Anais do XIV Simpósio Brasileiro de Engenharia de Software;* Ed. SBC; 2000; pp 374-375

[12] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.; *Design Patterns, Elements of Reusable Object-Oriented Software*; Ed. Addison-Wesley; 1995.

[13] Harel, D.; “From Play-In Scenarios to Code: An Achievable Dream”; *IEEE Computer;* 34(1); 2001; pp 53-60.

[14] Hohenstein, U.; “An Approach for Generating Object-Oriented Interfaces for Relational Databases”; In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools*; 2000; pp 101-111.

[15] Johnson, R.; ”Frameworks = (Components + Patterns)”; *Communications of the ACM*; 40(10); 1997; pp 39-42

[16] Lanergan, R.; Grasso, C.; “Software Engineering with Reusable Designs and Code”; In Ted J. Biggerstaff and Alan Perlis (Eds.); *Software Reusability*; Addison-Wesley/ACM Press; 1989; pp 187-196

[17] Milicev, D.; “Extended Object Diagrams for Transformational Specifications in Modeling Environments”; In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools*; 2000; pp. 121-131.

[18] Masiero, P.C.; Meira, C. A.; “Development and Instantiation of a Generic Application Generator”; *J. System Software;* 23; 1993; pp 27-37.

[19] Pree, W.; Design Patterns for Object-Oriented Software Development; Prentice-Hall, NJ; 1996.

[20] Pree, W.; “Hot-Spot-Driven Development”; In Mohamed Fayad and Douglas C. Schmidt (Eds.); *Building Application Frameworks: Object-Oriented Foundations of Framework Design*; John-Wiley & Sons; 1999; pp. 379-393.

[21] Rational Co.; Rose 98 Rose Extensibility User’s Guide, 1998.

[22] Roberts, D.; Johnson, R.; “Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks”; *Proceedings of the Third conference on Pattern Languages and Programming*; Illinois;1996; http://st-www.cs.uiuc.edu/users/droberts/evolve.html

[23] Schmid, H.; “Creating Applications from Components: A Manufacturing Framework Design”; *IEEE Software*; 13(6); 1996; pp 67-75

[24] Schmid, H.; “Systematic Framework Design by Generali-zation”; *Communications of the ACM*; 40(10); 1997; pp 48-51

[25] Schmid, H.; “Framework Design by Systematic Generali-zation”; In Mohamed Fayad and Douglas C. Schmidt (Eds.); *Building Application Frameworks: Object-Oriented Foundations of Framework Design*; John-Wiley & Sons; 1999; pp. 379-393.

[26] Staa, A.; Manual de Referência: Talisman: Ambiente de Engenharia de Software Assistido por Computador; *Staa Informática*; 1993.

[27] Staa, A.; *Programação Modular*; Ed. Campus; 2000