LUIZ PAULO ALVES FRANCA

UM PROCESSO PARA CONSTRUÇÃO DE GERADORES DE ARTEFATOS

Tese de Doutorado

DEPARTAMENTO DE INFORMÁTICA

Rio de Janeiro, 17 de agosto de 2000

Luiz Paulo Alves Franca

Um processo para a construção de geradores de artefatos

Tese apresentada ao Departamento de Informática da PUC-Rio como parte dos requisitos para obtenção do título de Doutor em Informática: Ciência da Computação

Orientador: Arndt von Staa

Departamento de Informática

Pontífica Universidade Católica do Rio de Janeiro

Rio de Janeiro, 17 de agosto de 2000

Dedico este trabalho,

aos meus pais Aloisio e Therezinha

aos meus irmãos Heloisa e Eduardo,

à minha Rose, namorada, esposa, e companheira

à nossa Marcela que está quase chegando

Agradecimentos

Meu orientador, mestre e amigo, Prof. Arndt von Staa por mostrar-me a saída deste potencial labirinto que é um trabalho de doutorado.

Ao CNPQ pelo apoio financeiro.

BNDES pela experiência de 3 anos como analista de sistemas do Depto. de Informática, onde pude vivenciar além das questões técnicas, os aspectos sociológicos que envolvem a implantação de sistemas.

Colegas do BNDES, Calvano, Carlos Henrique, Fernando, João Marcelo, Sidnei, Margarida, ... , pelo incentivo no início desta jornada.

Meus amigos e sócios da Interatum Informática (atual Ingresso.com) Mauro e Jorge, pela ajuda na consolidação de nossa empresa, por suprirem as minhas ausências, enquanto estava realizando este meu outro sonho.

Colegas da Ingresso.com Anddré, Marcelo, Erick, Mássimo, Robson, Fernanda, Fábio, Deborá, Roberto, Dário, Heloisa, Patricia, ... , pelo incentivo ao longo da jornada.

Meu cliente e amigo, Pedro Barenboim, pela oportunidade de desenvolver meu primeiro sistema efetivamente útil, e por mostra-me que enquanto num romance policial, o culpado é o mordomo, numa empresa o culpado sempre é o analista de sistemas.

Prof. Firmo Freire pelas dicas no início do trabalho, e por apresentar-me a sincronicidade.

Hamilton Fonte II, futuro Engenheiro de Computação, pela ajuda na exploração do mundo da programação Internet e pelos protótipos em Java.

Paulo Caroli, cujo trabalho de mestrado proporcionou-me importantes insights.

Depto. de Informática da PUC-Rio pela superação da precariedade da época collor, e pelas excelentes condições oferecidas para realização deste trabalho.

Meus amigos de surf Márcio, Júlio, Marcelo, Armando e Krishinan pela minha ausência no pico e pelas ondas que deixei pegar durante estes 4 últimos anos.

Meu xará e amigo, Luiz Paulo, por compartilhar sua amizade.

Bahia por continuar me recebendo de braços abertos.

Resumo

Um gerador de artefatos é um software que produz um artefato a partir de sua especificação de alto nível. Um artefato é qualquer item criado como parte da definição, manutenção ou utilização de um processo de software. Geradores de artefatos permitem reduzir os custos e o tempo para a disponibilização, e aumentar a confiabilidade de artefatos dentro de um domínio de aplicação. No entanto, para que se possa disseminar mais o emprego de geradores de artefatos, devem ser reduzidos os custos e a dificuldade do seu desenvolvimento e manutenção. Esta tese apresenta um processo de baixo custo para o desenvolvimento e a manutenção de geradores de artefatos. O processo inicia com um exemplo de um artefato simples visando determinado domínio de aplicação. Dado este exemplo e levando em consideração o conjunto de todos os possíveis artefatos dentro deste domínio, são identificadas todas as partes comuns e todas as partes variáveis. São identificadas também as propriedades a serem satisfeitas pelas especificações de cada artefato-alvo. A seguir, o exemplo é modificado de modo que venha a conter marcadores de geração em cada um dos pontos variáveis. Estes marcadores estabelecem as regras de transformação a serem aplicadas ao gerar um artefato específico a partir de sua especificação. Para programar os editores das especificações e os geradores, utilizamos uma ferramenta CASE. Por intermédio de outra ferramenta, o exemplo devidamente marcado é transformado em um componente a ser combinado com a biblioteca de transformações, produzindo, assim, o gerador a ser internalizado na ferramenta CASE. O processo foi utilizado com sucesso para transformar especificações em aplicações, componentes e documentação.

Palavras Chave

Gerador de artefatos, gerador de aplicações, CASE, processo de desenvolvimento de geradores.

Abstract

An artifact generator is a software tool which produces an artifact from its high-level specification. An artifact is any item created as part of definition, maintenance or utilization of a software process. Artifact generators are a way to reduce production costs and time to market, and to increase reliability of artifacts within a given application domain. However, in order to achieve a more widespread usage of artifact generators, the difficulty and the cost of their development and maintenance must be reduced. This thesis presents a low-cost artifact generator development and maintenance process. The process departs from an example of a simple artifact within the target application domain. Given this example and considering all possible artifacts of the domain, all commonalties and variabilities are identified, as well as the properties that the specification of each specific target artifact must satisfy. The example is then modified in order to contain specific generator tags at all variable points. These tags establish the transformation rules that must be applied to the specification in order to generate the application. We have used a CASE tool, which allows the programming of the specification editors and of the generators. By means of another tool, the tagged example artifact is transformed into a component, which is combined with the transformation library, yielding the generator code to be internalized by the CASE tool. The process has been successfully used to transform specifications into applications, components and documentation.

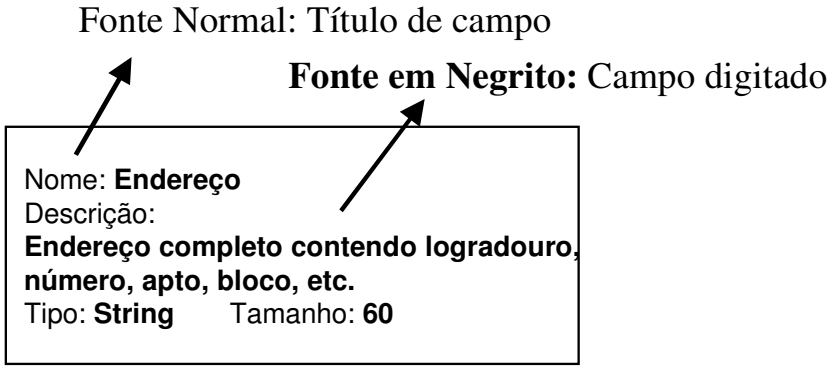
Keywords

Artifact generator, application generator, CASE, generator development process

Sumário

Lista de Figuras

Padrão de figura de formulário de entrada de dados



Capítulo 1- Introdução

Neste capítulo apresentamos os fatores que motivaram esta tese, os seus objetivos e sua organização

1.1 - Motivação

Ao longo de sua carreira, um analista de sistema observa que grande parte das atividades de desenvolvimento e de manutenção consiste no reúso de artefatos que foram desenvolvidos anteriormente. A efetividade deste tipo de reúso depende da experiência do desenvolvedor. Na maioria das vezes, o reúso é realizado através de sucessivas operações de "cortar/colar/modificar". A qualidade do artefato produzido varia conforme a quantidade dessas operações de reúso. Quanto maior a quantidade de adaptações, pior a qualidade, pois devido à sua natureza manual e repetitiva, estas operações freqüentemente levam a erros. Apesar das limitações, esta é provavelmente a prática de reutilização mais difundida.

Uma possível forma de automação deste tipo de reúso empírico é através da utilização de geradores de artefatos. Um gerador de artefatos é uma ferramenta que produz um artefato a partir de sua especificação. Os artefatos gerados podem ser: programas completos, módulos, documentação, arquivos de configuração, planos de testes, etc. A substituição do reúso do tipo "cortar/colar/modificar" pela adoção de geradores de artefatos, somente é possível, caso o processo de construção de geradores seja efetivamente dominado pela organização e seja economicamente viável.

Quando uma organização decide utilizar geradores de artefatos dentro dos seus processos de desenvolvimento, ela tem como principal objetivo a promoção do reúso de largo espectro [Pfleeger 1998]. Diferentes relatos apresentados na literatura [Biggerstaff 1998, Basset 1997] mostram resultados similares bastante expressivos em relação à adoção de geradores. Os principais benefícios esperados com a utilização de geradores são:

- **Aumento de produtividade.** Na maioria dos geradores, a maior parte de um artefato gerado corresponde a trechos fixos referentes a detalhes de

implementação. A geração automática destas partes fixas proporciona uma elevação da produtividade da equipe de desenvolvimento.

- **Redução do tempo para o mercado.** O tempo de desenvolvimento de um novo artefato fica reduzido ao tempo gasto para fornecimento de sua especificação.
- **Prototipação.** Para criar novas versões de um artefato basta ao desenvolvedor alterar a especificação fornecida ao gerador. Este baixo custo de criação de novas versões, permite ao desenvolvedor conduzir diversos experimentos para determinar o artefato mais adequado ao usuário.
- **Qualidade do artefato gerado.** Quando da utilização de um gerador pode-se pressupor que esteja correto, bem como que o artefato gerado também seja correto por construção. Caso o artefato gerado tenha problemas, estes estarão associados a problemas na especificação fornecida ao gerador.

Geralmente o escopo de um gerador comercial é bastante restrito, isto é, alterações no tipo do artefato gerado estão limitadas ao conjunto de propriedades predefinidas do gerador. Com isto, nem sempre é possível encontrar no mercado um gerador que atenda exatamente às necessidades de geração de uma organização. A análise custo X benefício em relação à técnica de desenvolvimento a ser adotada, será favorável à utilização de gerador de artefatos, caso o artefato a ser desenvolvido esteja dentro do escopo do gerador disponível pela organização. Contudo, quando o artefato-alvo não se encaixar no escopo do gerador, o custo de produzi-lo torna impeditiva a utilização do gerador, visto que o artefato após ser gerado deve sofrer alterações para atender às necessidades do usuário.

Com o objetivo de reduzir a distância entre o artefato desejado e artefato gerado, podem ser adotadas duas possíveis abordagens para construção de um gerador que atenda às necessidades da organização. A primeira seria a construção de geradores mais genéricos, possuindo um grande número de parâmetros que poderiam ser alterados em função do tipo de artefato a ser produzido. Esta abordagem pode tornar a estrutura do gerador extremamente complexa, pois o gerador embutiria inúmeras alternativas de implementação

que precisariam ser antecipadas pelo seu desenvolvedor e que, talvez nunca fossem usadas [Bassett 1997, CHSV 1997]. A segunda abordagem consiste na simplificação da construção de geradores específicos, ou seja, caso o gerador X não sirva para gerar o artefato ABC pertencente a determinado domínio de problema, a organização poderia criar um novo gerador Y específico para artefatos deste domínio e a seguir gerar o artefato ABC. O nosso trabalho descreve uma alternativa para esta segunda abordagem.

A questão do processo de construção de geradores de artefatos não é uma novidade dentro da área de Engenharia de Software. Entendemos que o maior problema das propostas apresentadas, seja o de que elas freqüentemente requerem do desenvolvedor conhecimentos nas áreas de linguagens formais e de compiladores. Como este perfil de desenvolvedor não é comumente encontrado na maioria das organizações, a técnica de construção de geradores acaba sendo pouco difundida.

1.2 – Objetivos

Em nosso trabalho, no sentido de ampliar a utilização de geradores de artefatos, procuramos simplificar o seu processo de construção bem como dotá-lo de ferramentas que disponibilizassem a infra-estrutura necessária para sua implementação. Da mesma forma que a construção de um artefato através do reúso do tipo “cortar/colar/modificar” é iniciada a partir de um artefato desenvolvido anteriormente, o processo de construção de um gerador de artefatos pode utilizar este mesmo princípio de criação. O ponto de partida do processo é um artefato-exemplo que é sistematicamente transformado num gerador de artefatos no mesmo domínio de problema alvo do artefato-exemplo.

Os principais objetivos a serem atingidos com este trabalho de tese são:

- Definir detalhadamente o processo de construção em relação, tanto às atividades a serem desempenhadas, como aos produtos envolvidos;
- Definir detalhadamente o processo de manutenção em relação, tanto às atividades a serem desempenhadas, como aos produtos envolvidos;
- Simplificar esses processos através da adoção de um processo “dirigido por exemplo”;

- Avaliar as vantagens da utilização de uma ferramenta CASE (*Computer-Aided Software Engineering*) como gerador de geradores de artefatos;
- Identificar e descrever as heurísticas empregadas na generalização de um artefato.

1.3 – Gênese do Problema

Durante uma boa parte do trabalho de doutorado, nossa área de interesse foi o campo da medição de software. Dentro desta área desenvolvemos alguns trabalhos [FS 1998, FSL 1998, FSF 1999] voltados para medição de software para pequenas organizações. Estes trabalhos visavam ao desenvolvimento da ferramenta COMPASSO, um ambiente para a geração de sistemas de medição customizável (Figura 1.1). O núcleo do COMPASSO é o meta-CASE Talisman [Staa 1993]. As facilidades disponibilizadas pelo Talisman para a customização de seus formulários de entrada e de seus arquivos de saída, são adequadas para permitir que o COMPASSO produza sistemas de medição compatíveis com a realidade de cada organização. A geração dos sistemas é feita a partir do modelo genérico de medição armazenado no repositório de dados do Talisman.

Para construir o COMPASSO, definimos as seguintes etapas de desenvolvimento:

1) ***Especificação do Modelo Genérico:*** Definição do modelo de dados para sistemas de medição de software que representasse tanto as entidades relacionadas com métricas como aquelas associadas ao processo de medição.
2) ***Desenvolvimento de Protótipos:*** Verificar alternativas de arquitetura e de implementação mais adequadas aos sistemas de medição [LD 1998] para serem utilizados por pequenas empresas de software.
3) ***Internalização no Talisman:*** Definição dos formulários de entrada de dados e dos arquivos de saída do Talisman a partir de um protótipo selecionado. Estas definições correspondem a programas escritos na linguagem nativa do Talisman. Por meio desta linguagem é possível

acessar, editar e manipular os objetos definidos pelo meta-modelo do repositório da ferramenta (seção 3.2).

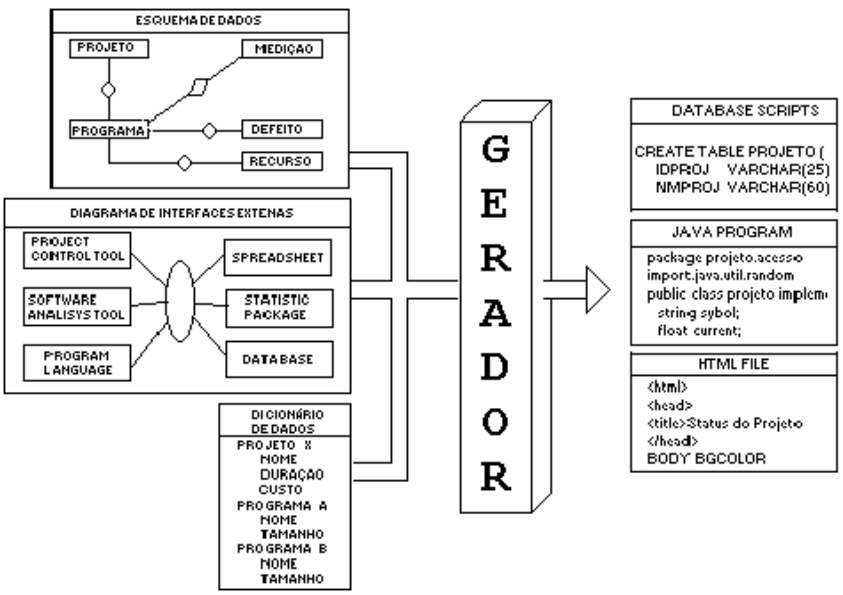


Figura 1.1 Visão Geral do COMPASSO

As 2 primeiras etapas foram concluídas e alcançaram os objetivos esperados. Em relação aos protótipos, foram feitas implementações que acessassem banco de dados por meio de diferentes linguagens de programação, como por exemplo: `VisualBasic`, `CgiLUA` [HBI 1997] + `HTML`, e `Java` + `HTML`. Como resultado da análise dos protótipos decidimos adotar a arquitetura em 3 camadas [ED 1997, Caroli 1999] (Figura 1.2) utilizando a tecnologia `Java-Servlet`. Este protótipo era composto por arquivos em `Java`, `JavaScript` e `HTML`, distribuídos em diretórios correspondentes a cada classe do modelo. Neste tipo de aplicação, boa parte da funcionalidade corresponde à execução de operações básicas de manipulação de tabelas (incluir, alterar, excluir, e consultar), a arquitetura adotada expôs as similaridades entre os arquivos que compunham a aplicação.

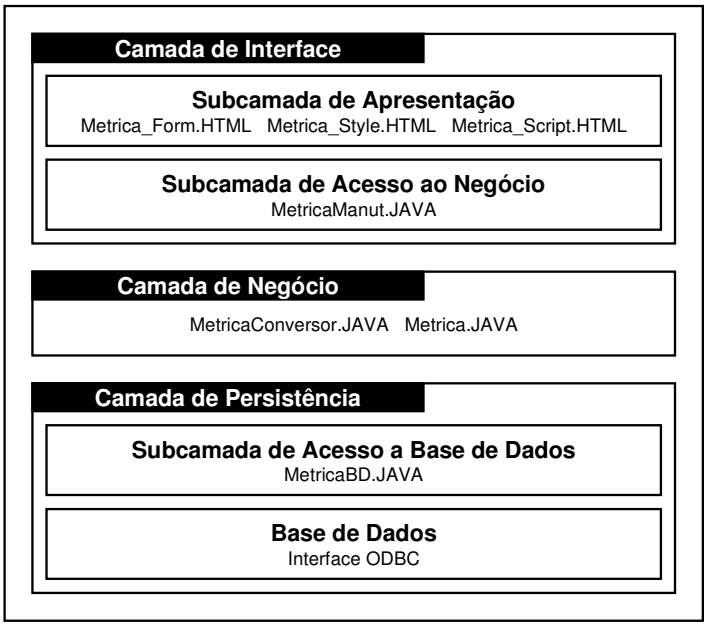


Figura 1.2 Implementação em 3 camadas da entidade Métrica do COMPASSO

Devido às similaridades presentes, utilizamos o reúso empírico do tipo "cortar/colar/modificar" durante a construção de uma versão mais completa do protótipo. A codificação de uma nova classe era baseada nos arquivos referentes a uma classe já implementada, na qual todas as referências ao nome e aos atributos eram substituídos pelos da nova classe (Figuras 1.3, 1.4). No protótipo, cada entidade do modelo estava mapeada em 4 arquivos em `JAVA` e em 3 arquivos em `HTML` (Figura 1.2-Camada de Interface). Como o modelo apresentava 15 entidades que poderiam ser criadas através deste tipo de reúso, as modificações envolveriam 105 arquivos. Como cada arquivo em média possuía 25 pontos de modificação, a versão final do protótipo seria resultado de cerca de 2500 operações de reúso do tipo "cortar/colar/modificar". Esta grande quantidade de modificações provavelmente introduziria inúmeras faltas na aplicação final [Basset 1997]. A quantidade elevada e a natureza repetitiva destas operações levou-nos a desenvolver uma solução baseada em um gerador de artefatos.

```
<form name="TELA" action="">
<div id="menu">
 <input type="button" class="botaoon" name="btMENU"
   onClick="PADListaMENU(); " value="Menu"><br>
 <input type="button" class="botaoon" name="btgravar"
  onClick="PADconf_gravacao();" value="Gravar"><br>
</div>
<div id="cabecalho"> Cadastro de METRICA </div>

<div id="formulario">
 <table>
 <tr> <td align="right"> <b>IDMETRICA</b> </td>
  <td> <input type="text" name="IDMETRICA"
     class="clIDMETRICA" readonly maxlength="8">
  </td>
 </tr>
 <tr> <td align="right"> <b>NMMETRICA</b> </td>
  <td> <input type="text" name="NMMETRICA"
     class="clNMMETRICA" maxlength="20"
     onchange= "ESPvalida(document.TELA.NMMETRICA,
        sAvaANTNMMETRICA, 'S','N',0,0,1 );">
  </td>
 </tr>
```

Figura 1.3 Exemplos de pontos modificados num trecho em HTML

```
public METRICA exibir(METRICA oFpaMETRICA) throws

java.sql.SQLException,ClassNotFoundException
{
  Class.forName(oCvaSISINFO.getDRIVERODBC());
  java.sql.Connection myConnection =

java.sql.DriverManager.getConnection(oCvaSISINFO.getODBC(),
     oCvaSISINFO.getUSUARIO(), oCvaSISINFO.getSENHA());
  java.sql.Statement exibirStatement=
myConnect.createStatement();
  java.lang.String sql;
  sql="SELECT *";
  sql=sql +" FROM METRICA";
  sql=sql +" WHERE IDMETRICA ='"
+oFpaMETRICA.getIDMETRICA()+"'";
  java.sql.ResultSet resultado = exibirStatement.
  executeQuery(sql);
  resultado.next();
  oFpaMETRICA.setIDMETRICA ( resultado.getString("IDMETRICA")
);
  oFpaMETRICA.setNMMETRICA ( resultado.getString("NMMETRICA")
);
  oFpaMETRICA.setTXTMETRICA( resultado.getString("TXTMETRICA")
);
  exibirStatement.close();
  myConnect.close();
  return oFpaMETRICA;
}
```

Deleted: ¶

Figura 1.4 Exemplos de pontos modificados num trecho em Java

Em relação à etapa de internalização no Talisman, as atividades foram divididas na programação dos formulários de entrada das especificações e nos geradores de arquivos de saída. Os formulários são utilizados para o preenchimento dos dicionários de dados referentes a um diagrama de entidade-relacionamento. Estes formulários são versões simplificadas dos programas-exemplo disponibilizados pelo Talisman. Em relação à atividade de programação dos arquivos de saída, os arquivos em `Java`, `JavaScript` e `HTML` do protótipo deveriam ser redigitados dentro do Talisman e conectados através de funções que buscassem dados do repositório da ferramenta (Figura 1.5). A quantidade de arquivos e de conexões a serem feitas tornava inviável a codificação manual destes arquivos. Em função do retrabalho que envolveria a internalização dos arquivos do protótipos, começamos a desenvolver uma solução que automatizasse esta etapa.

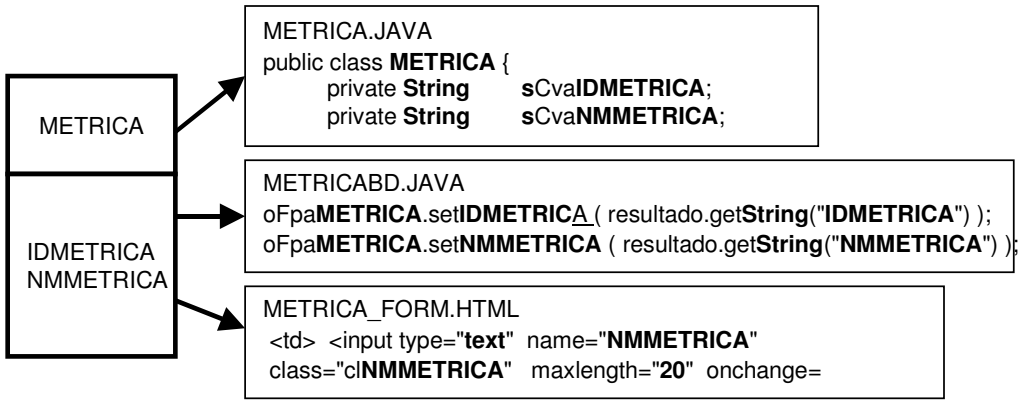


Figura 1.5 Exemplos de geração dos arquivos de saída

A partir deste momento, em função da experiência acumulada na criação de diferentes protótipos dentro do domínio de medição de software, da motivação inicial do projeto de um gerador para este domínio, bem como da necessidade de automatizar a própria construção do gerador, decidimos que o foco da trabalho de doutorado fosse deslocado para elaboração de um processo de construção de geradores de artefatos.

1.4 – Organização da Dissertação

O capítulo 2 apresenta os conceitos básicos e os trabalhos relacionados utilizados ao longo da dissertação. A arquitetura padrão de geradores e meta-geradores são apresentadas. São analisadas algumas propostas de processo de construção de geradores. Neste capítulo, os principais conceitos de *framewoks* são apresentados.

O capítulo 3 descreve os métodos, técnicas e ferramentas utilizados no processo de construção de geradores de artefatos. A adequação da utilização de um CASE como ambiente de geração é discutida. A abordagem de generalização de artefatos é apresentada.

O capítulo 4 apresenta a descrição do processo de construção de geradores de artefatos. A etapa de construção do artefato-exemplo é ponto partida da construção do gerador. Ao longo do processo, o artefato-exemplo vai sendo generalizado até ser transformado num gerador. A etapa final de controle da qualidade é necessária para certificar que o artefato gerado está em conformidade com a especificação fornecida.

O capítulo 5 é dedicado à questão do controle da qualidade na construção de um gerador. A abordagem empírica empregada é discutida, e também são apresentadas algumas referências de uma abordagem mais formal.

O capítulo 6 apresenta um resumo dos experimentos realizados. São descritos sete estudos de caso que serviram para refinar o processo proposto e avaliar os impactos na evolução de um gerador.

O capítulo 7 conclui o trabalho e apresenta sugestões para trabalhos futuros.

A tese possui 6 apêndices.

O apêndice A é um glossário dos principais termos utilizados ao longo da dissertação.

O apêndice B descreve a BNF do arquivo de meta-descrição do artefato.

O apêndice C mostra a lista das funções de transformação.

O apêndice D apresenta o padrão de documentação utilizado para descrever o gerador construído.

O apêndice E apresenta o padrão de nomeação utilizado nos programas desenvolvidos ao longo deste trabalho.

O apêndice F apresenta algumas medições que foram realizadas durante os estudos de caso.

Capítulo 2- Conceitos Básicos

Este capítulo apresenta os conceitos básicos utilizados neste trabalho. Na primeira parte apresentamos as tecnologias utilizadas para a construção de um gerador. Já na segunda parte, são descritas algumas propostas para o processo de construção de geradores. Inicialmente, é apresentada a descrição e a arquitetura de um gerador de artefatos. A seguir, é descrito como utilizar meta-geradores para automatizar a construção dos geradores e são apresentados exemplos de meta-geradores. Após, considerando que um gerador de artefatos é um tipo de framework, são apresentados os principais conceitos de frameworks. Finalmente, são descritos alguns processos de construção de geradores.

2.1 – Geradores de Artefato

Em nosso trabalho, adotamos o termo *gerador de artefato* no lugar de gerador de aplicação. Segundo Staa [Staa 2000]:

> *"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. 2.1). O artefato gerado pode ser composto por outros artefatos.

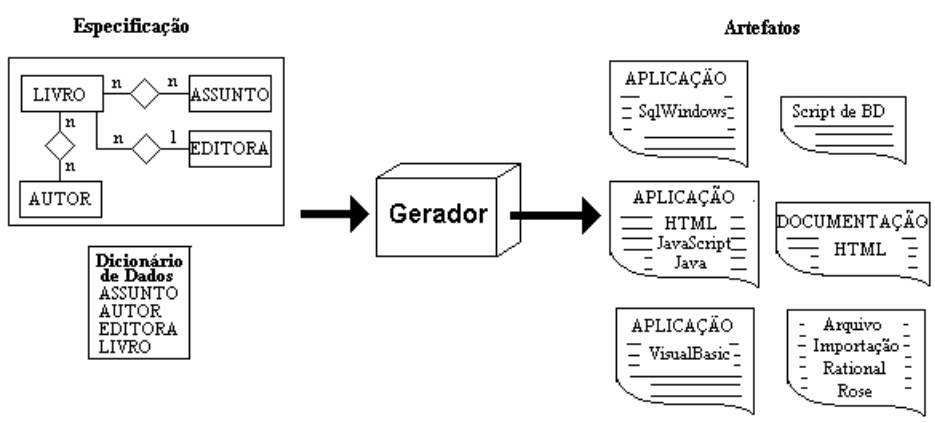


Figura 2.1. Visão esquemática de um gerador de artefatos

Geradores de artefatos não são uma idéia nova. Em [Jenkins 1985] já foi apresentado um *survey* dos geradores disponíveis comercialmente naquela época. Desde o início, geradores comerciais são caracterizados por produzir aplicativos específicos ou por automatizar tarefas dentro do processo de desenvolvimento, como por exemplo, geração de: relatórios, telas, consultas, estruturas de banco de dados, programas de conversão de arquivos, etc.

A Figura 2.2 apresenta uma visão esquemática da utilização de geradores. 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). A construção de um artefato através de um gerador envolve as seguintes etapas:

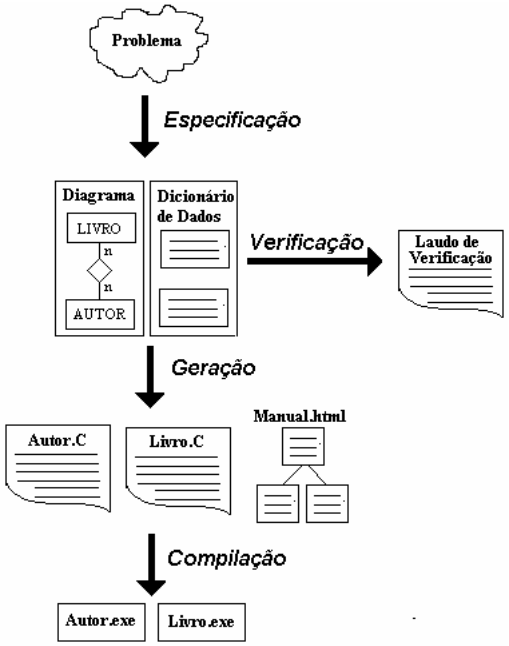


Figura 2.2 Visão esquemática da utilização do gerador de artefatos

- **Especificação do Arterfato:** A especificação do artefato deve prover todas as informações necessárias para a geração do artefato. Esta especificação pode ser fornecida através de dicionários de dados e de diagramas. Além disso, antes de disparar o processo de geração, o desenvolvedor deve verificar se a especificação está completa através do programa de verificação da especificação.

- **Geração do Artefato:** Após fornecer a especificação e, caso esta não mais contenha problemas, o desenvolvedor comanda a geração do artefato. A geração utiliza informações da especificação e da descrição do artefato que está sendo gerado. O artefato gerado pode ser composto por um ou mais arquivos, que podem estar codificados em diferentes linguagens.

- **Compilação do Artefato:** Caso o artefato gerado seja um programa, é necessário compilar os fontes gerados após a sua geração para obter uma versão executável do artefato. Dependendo das linguagens de programação utilizadas, os arquivos gerados precisam ser preparados para serem compilados. Esta preparação envolve a construção do arquivo de

projeto (*make script*) que irá guiar a compilação a partir da estrutura de diretórios em que se encontram os programas.

2.2 – Arquitetura de um Gerador de Artefatos

A Figura 2.3 sintetiza a arquitetura padrão apresentada na literatura [Balzer 1989, MM 1993, BS 1999]. Os componentes básicos da arquitetura são: o analisador de especificação e o gerador de artefato.

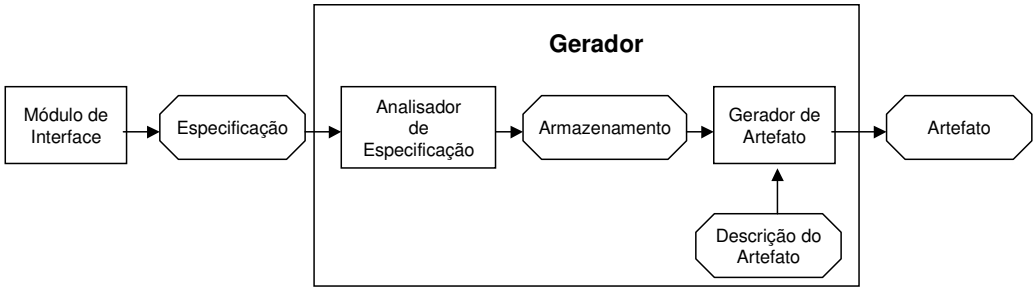


Figura 2.3. Arquitetura padrão de um gerador de artefatos

O analisador de especificação (ou *front-end*) é o módulo responsável por transformar uma especificação redigida numa linguagem "X" para uma outra redigida na linguagem "Y" e que define uma representação interna mais favorável para o processo de geração. O analisador de especificação implementa as funções de um analisador léxico e de um *parser*. O formato da especificação, geralmente é textual, hipertexto ou diagramático. Já a representação interna pode ser uma *abstract syntax tree (AST)*. Observa-se que a complexidade deste módulo está diretamente ligada às diferenças entre os formatos da especificação e da representação interna: quanto mais diferente, mais complexo será o módulo. Por outro lado, uma especificação que utilize diretamente a linguagem da representação interna pode apresentar problemas de legibilidade, pois esta linguagem pode ter pouca expressividade em relação ao domínio do problema.

Alguns geradores possuem um módulo anterior ao analisador de especificação, utilizado para capturar a especificação, denominado módulo de interface. Através deste módulo o usuário fornece a especificação,

preenchendo formulários, respondendo a uma seqüência de diálogos (*wizard*) ou desenhando diagramas.

O gerador de artefato (*back-end*) transforma a especificação armazenada na representação interna nos componentes do artefato-alvo. Esta transformação pode ser executada em 2 etapas. A primeira etapa consiste em gerar um artefato intermediário descrito através da linguagem de representação interna. A etapa complementar realiza o mapeamento entre a linguagem de representação interna e as linguagens do artefato final. Estes artefatos podem ser programas codificados em diferentes linguagens de programação.

2.2 – meta-Geradores de Artefatos

Para valer a pena o seu emprego, o custo de desenvolver e manter um gerador de artefatos deve ser superado pela economia obtida através da redução do custo de construção e manutenção dos artefatos gerados [Levy 1986]. Esta meta pode não ser alcançada através do processo tradicional de desenvolvimento de software. Para superar este problema, foram desenvolvidos processos de construção baseados em meta-geradores, nos quais o paradigma de geração é empregado na construção do próprio gerador.

Um meta-gerador (Figura 2.4) é um gerador de geradores de artefatos, classificado como um gerador de múltiplos domínios e de múltiplos artefatos [MM 1993]. Os geradores produzidos pelo meta-gerador são classificados de geradores de único domínio e de único artefato. Podemos observar na Figura 2.4 que o aspecto “meta” do meta-gerador deve-se à externalização da descrição tanto das linguagens de especificação como dos artefatos-alvo.

O processo de construção através do meta-gerador torna desnecessária a codificação dos componentes internos de um gerador de artefatos. Estes componentes são gerados a partir dos arquivos externos ao meta-gerador. Para construir um novo gerador, basta ao desenvolvedor criar os arquivos de descrição da linguagem de especificação e de descrição do artefato a ser gerado.

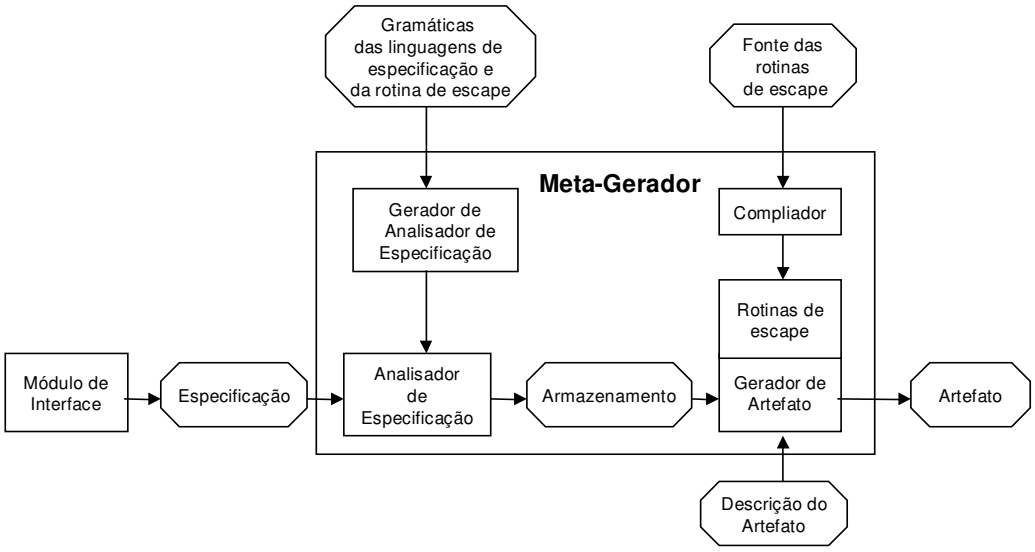


Figura 2.4. Arquitetura padrão de um meta-gerador de artefatos

O analisador da especificação é construído através do gerador de analisadores a partir da definição da gramática da linguagem de especificação. Já em relação ao gerador do artefato, a definição do artefato a ser gerado está descrita num arquivo externo ao gerador.

Para aumentar a flexibilidade do artefato gerado, alguns geradores aceitam como entrada de dados, além da especificação, também trechos de código-fonte que serão incorporados ao artefato final sem passar por qualquer tipo de transformação. Este tipo de código é denominado de *rotina de escape*. Por exemplo, no caso do gerador produzir programas para a manutenção de tabelas de dados, além do código gerado automaticamente referente à critica de dados, o usuário pode fornecer, junto com a especificação, trechos de código que serão acrescentados ao código de crítica de dados, especializando, desta forma, a crítica às necessidades específicas do artefato sendo gerado. Caso o gerador não possua mecanismos de verificação sintática da rotina de escape, caberá ao usuário evitar que esta introduza faltas[1] no artefato gerado.

Na literatura, existem diversos relatos sobre meta-geradores [Baxter 1992, Baxter 1999, BFHLPRRB 1999, CK 1988, LTV 1996, MM 1993, Neighbors

[1] *Falta* é um problema latente contido em um artefato e que pode vir a provocar uma *falha*. Faltas são inseridas por *erro* humano ou de ferramenta utilizada ao gerar o artefato [Pfleeger 1998].

1989, SKC 1993, SMcD 1996, Smith1990, Winter 1999], a maioria deles referentes à programação automática envolvendo especificações formais. No nosso caso, estamos interessados em meta-geradores que permitam a construção de geradores sem necessitar de especificações redigidas em linguagens ditas formais. Nas próximas seções, apresentaremos uma descrição dos meta-geradores que apresentam características que estão dentro do contexto do nosso trabalho.

2.2.1 – Draco: meta-Gerador baseado em Domínios

O ambiente Draco foi resultado do trabalho de doutorado de Neighbors [Neighbors 1989] no início da década de 80. Este trabalho foi importante para definição de conceitos envolvendo análise de domínio e transformações.

No ambiente Draco (Figura 2.5), o mapeamento de uma especificação até uma das suas possíveis implementações, envolve transformações inter e intra-domínios. As transformações inter-domínios, ou verticais, transformam os artefatos descritos em um domínio em descrições de outro domínio. Estas transformações também são chamadas de refinamentos. Já as transformações intra-domínios, ou horizontais, podem ser transformações de otimização ou de preparação para transformações verticais. Neste tipo de transformação não há mudança de domínio.

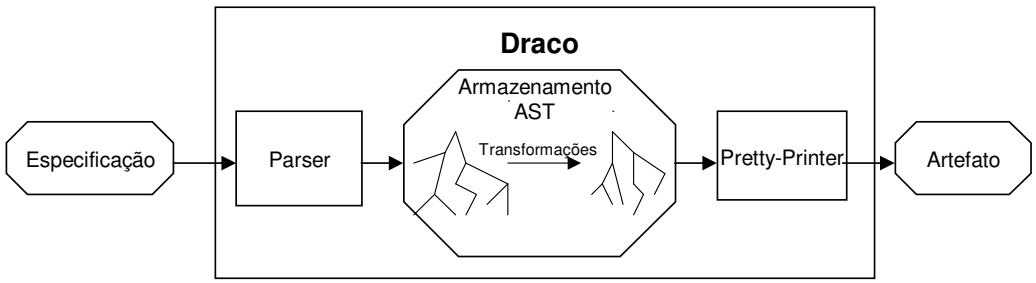


Figura 2.5. Visão Esquemática do Ambiente Draco

As transformações correspondem geralmente a um tipo de regra formada por dois padrões: o padrão de reconhecimento (LHS - *Left Hand Side*) e o padrão de substituição (RHS - *Right Hand Side*). Para que uma regra de transformação seja aplicada, o sistema procura o padrão de reconhecimento. Ao ser encontrado, ele será substituído pelo padrão de substituição. As regras

podem possuir ainda restrições semânticas, impondo condições que devem ser seguidas antes ou após sua aplicação [BPL 1996].

Internamente, os domínios são armazenados em uma estrutura do tipo *abstract sintax tree* (AST). Por conseguinte, as transformações verticais e horizontais correspondem a mapeamentos do tipo AST para AST. Através do componente *Parser*, uma especificação numa dada gramática é transformada na representação interna do tipo AST. Geralmente a especificação está no formato texto. O produto final é produzido através do componente *Pretty-Printer*. A capacidade de geração do Draco é ampliada na medida em que mais gramáticas de linguagens de especificação possam ser tratadas e que haja uma maior diversificação das saídas do componente *Pretty-Printer*.

Além de sua importância como sistema transformacional, o Draco mostrou-se um ambiente propício para realização de diferentes experimentos na área de Engenharia de Software. Desde a sua primeira versão, o Draco passou por várias evoluções destacando-se os trabalhos desenvolvidos na PUC-Rio, a partir das pesquisas do Prof. Júlio Cesar S. P. Leite. Através de um processo de re-engenharia [Prado 1992], a máquina Draco foi atualizada e disponibilizada para diferentes grupos de pesquisa, tornando-se instrumento de diversos trabalhos [FMCM 1996, FL 1997, SLP 1998]. Dentre os trabalhos mais recentes, destacamos:

Draco + Ferramentas CASE: Prado [KPCLG 1996, LPBC 1997, BPB 1999] vem realizando uma série de trabalhos onde ferramentas CASE são integradas ao ambiente Draco. Através da ferramenta CASE, procura-se superar a restrição de trabalhar-se apenas com especificações textuais, e acrescenta ainda a possibilidade de criar-se diferentes representações para a mesma especificação.

Circuitos Transformacionais: Sant'anna, baseado na sua experiência com a máquina Draco [LSF 1994], desenvolveu em sua tese doutorado [Sant'Anna 1999] uma linguagem para descrição de sistemas transformacionais baseados em componentes. O conceito de circuitos transformacionais é apresentado como elemento principal na descrição e na integração de diferentes sistemas transformacionais. No trabalho de Sant'Anna encontra-se um resumo das pesquisas mais recentes da área de sistemas transformacionais.

2.2.2 – *Stage*: Experiência AT&T em Geradores

Na AT&T, na década de 80 [Cleaveland 1988, CK 1988], foram realizadas algumas experiências bem sucedidas na utilização de geradores em sistemas de telefonia. Em função destes resultados positivos a AT&T decidiu implantar a tecnologia de geração de artefatos em seus diferentes laboratórios. A alternativa de concentrar o desenvolvimento de geradores num único grande centro de suporte e construção de ferramentas foi descartada, pois este centro não teria conhecimento dos diferentes domínios de problema presentes na empresa. A estratégia foi desenvolver o *Stage* (Figura 2.6) que é uma ferramenta de suporte à construção de geradores (*Stage*) e que pode ser utilizada pelos próprios desenvolvedores do domínio do problema.

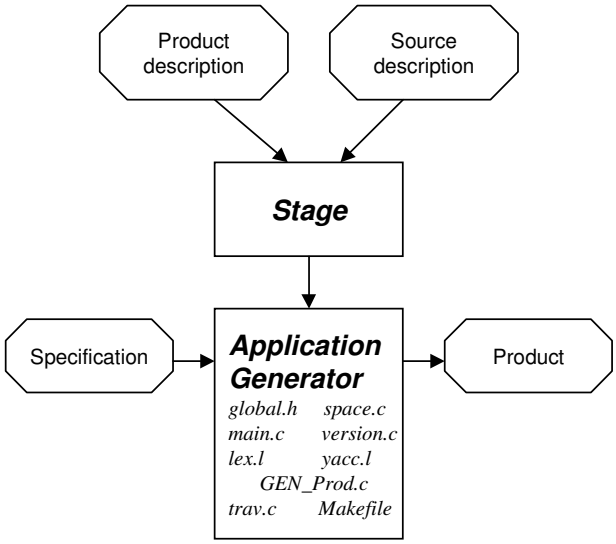


Figura 2.6. Visão Esquemática do *Stage*

Para criar o gerador, o *Stage* lê o arquivo *Source Description* e um ou mais arquivos de *Product Description.* O *Stage* cria um diretório contendo os programas do gerador codificados na linguagem C. O arquivo *Source Description* utiliza uma gramática que descreve a entrada do gerador. Já o arquivo *Product Description* contém a descrição da saída do gerador. Por meio deste arquivo, é definida a integração da saída do gerador com a *Parse-tree*. Esta integração é especificada através de marcadores que são inseridos no

arquivo com a finalidade de descrever como obter dados das *Parse-tree* e colocá-los dentro do produto.

Os geradores produzidos pelo *Stage* são compostos por: um *lexer* gerado pelo Lex, um *parser* gerado pelo YACC, definição da estrutura de dados, rotinas de construção do *parse-tree*, rotinas de caminhamento, e rotinas de geração de produtos. O Stage considera o *parser* um atividade separada, as outras atividades são feitas com relação a *parse-tree*, que também serve de repositório de informações. O Stage disponibiliza rotinas para navegar sobre a árvore. Marcadores (*tags*) são colocados em todos os *tokens*, isto possibilita relatórios de erros detalhados. Pontos de escape são disponibilizados possibilitando a inserção de trechos de código. Outras facilidades são apresentadas, como por exemplo: controle de versão, *pretty-printer*, linha de comando, etc.

Através do *Stage*, os seguintes tipos de geradores foram criados: interface de usuário, gerador baseado em máquinas de estado, simuladores de dispositivos de comunicação, ferramentas de teste, ferramentas de *design*, geradores de tradutores e compiladores. O próprio *Stage* foi gerado através dele mesmo.

Além do *Stage,* um outro fator que contribuiu para o sucesso da implantação da tecnologia de geração dentro da AT&T foi a execução de um amplo programa de treinamento (mais de 100 pesquisadores foram treinados).

O principal projetista do *Stage*, o pesquisador Cleveland, desenvolveu uma versão comercial denominada “Meta-Tool” [Sindelar 1990]. Desde o *Stage*, este pesquisador vem procurando simplificar a construção de geradores, por meio da redução da necessidade de conhecimentos na área de compiladores por parte do desenvolvedor [Cleaveland 2000].

2.2.3 – Eugene: meta-Gerador baseado em meta-Modelos

Eugene [PR 1999] é um meta-gerador cuja principal característica é a construção de geradores a partir da estrutura dos meta-dados que descrevem o gerador a ser criado. Os meta-dados são identificados através da análise da especificação e armazenados em diferentes tipos de meta-modelos. O produto final é resultado da aplicação de funções, chamadas de *translation function*

cujos resultados são trechos de fontes compostos por partes fixas do produto final e por partes variáveis obtidas a partir da extração de informações dos meta-modelos. Os meta-modelos são escritos utilizando a linguagem EXPRESS, que é uma linguagem de modelagem orientada a objetos. As *translation function* também são escritas utilizando a linguagem EXPRESS.

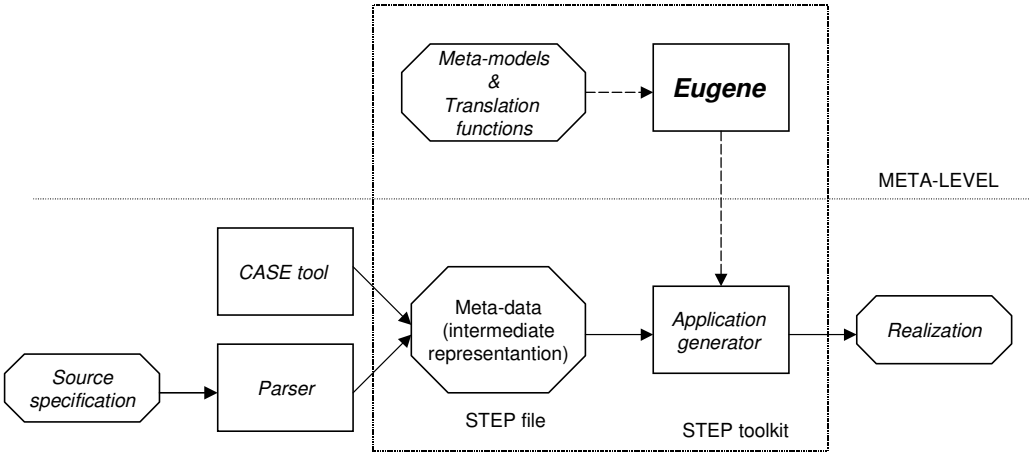


Figura 2.7. Visão Esquemática do Eugene

Outra característica do meta-gerador Eugene é sua integração com outras ferramentas de desenvolvimento. Esta integração é feita através da utilização de arquivos externos contendo meta-modelos descritos no formato STEP que segue o padrão ISO 10303 para compartilhamento de informações entre ambientes de desenvolvimento.

Dentre os resultados favoráveis da utilização do Eugene, os pesquisadores ressaltam a oportunidade de reúso de meta-modelos, e a interoperabilidade através da integração de meta-modelos.

2.2.4 – *Frame*: meta-Gerador Industrial

Em um artigo publicado em 1987 [Basset 1987], Bassett utilizou o termo *Frame* para descrever sua proposta de reúso adaptativo. Um meta-gerador baseado na tecnologia de *Frames* é simplesmente um pré-processador independente de linguagem que constrói módulos de software através da adaptação de componentes generalizados chamados de *Frames*.

Os *Frames* são organizados em hierarquias (Figura 2.8), cada *Frame* é raiz dos *frames* sub-componentes, e pode adaptá-los. A adaptação é através de comandos que combinados podem adicionar, alterar, excluir, instanciar, selecionar e iterar qualquer detalhe de um *sub-frame*. Na verdade, *frames* são *inputs read-only* para o pré-processador que começa pela raiz da árvore, lê os *frames*, interpreta os comandos, e emite o código-fonte que usualmente deveria ser escrito a mão. A hierarquia de *Frames* isola as diferente fontes de alterações em camadas separadas, tornando fácil a localização dos efeitos das alterações.

Um *Frame* é um módulo armazenado na forma de arquivo-texto. O conteúdo do *Frame* é formado por comandos de *Frame* e de texto genérico. O texto genérico pode conter variáveis cujos valores são atribuídos pelos comandos de *Frame*. A linguagem de manipulação de *Frames* é composta de 5 comandos básicos que são suficientes para permitir a transformação de uma hierarquia de *Frames* num artefato completo.

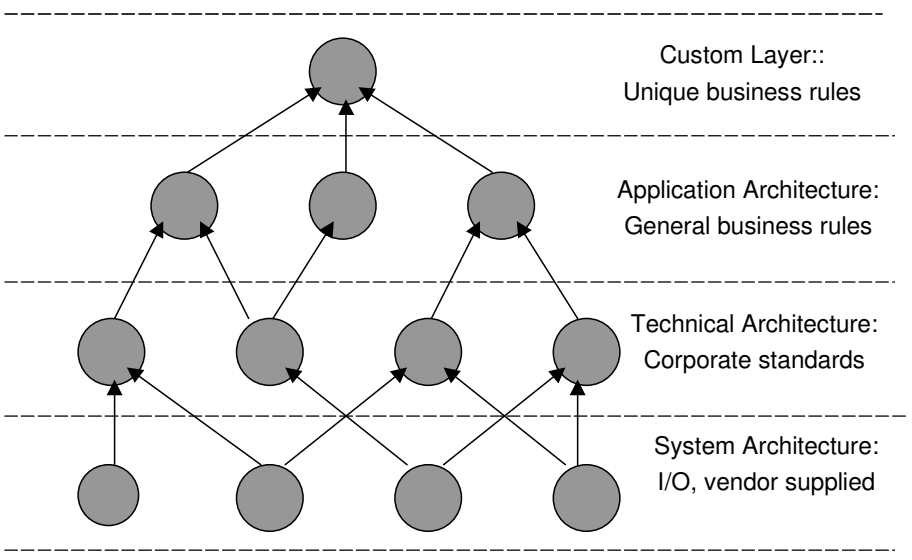


Figura 2.8. Hierarquia de *Frames* [Basset 1996]

- *System Architecture Layer*: *Frames* mais reutilizados. Ex.: funções de E/S, interface, acesso a BD. Experimentos mostraram que de 50-80% do código de uma aplicação estão nesta camada. Estes *Frames* são usados por toda a organização.

- *Technical Architecture Layer*: Funções padrões da organização. Ex.: login, mensagens de erros, padrões de tela e relatórios. Estes *Frames* são usados por toda a organização.

- *Application Architecture Layer*: Esta camada contém as regras de negócio. Estes *Frames* são reutilizados por aplicações de um mesmo domínio.

- *Custom Layer*: Frame de especificação, que determina como o módulo de software é montado e como as customizações devem ser tratadas. Esta é a única camada que o desenvolvedor de aplicação pode criar e modificar. Experimentos mostraram que estes *Frames* correspondem a 5% de todo o código da aplicação.

Em [Basset 1997] é descrito o caso de uma empresa em que, após 5 anos de adoção dos *Frames*, foram produzidas 20 milhões de linhas de código em Cobol, por uma equipe de 6 a 15 pessoas. A biblioteca de *Frames* era composta por 30 *Frames* de I/O, 38 *Frames* do domínio da aplicação, 400 *Frames* para geração de visões, telas e relatórios, e 1 *frame* de especificação por programa. Os *Frames* de especificação correspondiam a 4% do total de linhas

Numa auditoria promovida [GM 1994] para analisar o desempenho da tecnologia de *Frame* em quinze projetos realizados por nove grandes corporações, foram encontrados resultados extremamente satisfatórios. Em relação ao custo em pessoal, nos projetos analisados foi observado uma redução média de 70%. Em relação ao tempo para o mercado, nos projetos analisados foi observado uma redução média de 84%. Os resultados apresentados são similares aos publicados no estudo de Biggerstaff [Biggerstaff 1998], onde os benefícios do reúso através de técnicas de geração são comparados com o reúso por meio de grandes bibliotecas de componentes de software.

2.4.5 – GenVoca: Modelo para o domínio de construção de geradores

Desde do final da década de 80, o Prof. Don Batory da Universidade do Texas vem construindo geradores para diferentes domínios de aplicação, o resultado destas experiências foi o desenvolvimento de um modelo para o domínio de construção de Geradores, chamado de modelo GenVoca [Batory 1996, Batory 1998, BSL 1998, BCRW 1998]. Este modelo pode ser implementado usando diferentes tecnologias de construção de geradores (por Composição, meta-programação, e por transformação).

GenVoca é um modelo para construção de software baseado em componentes que são organizados em camadas de software. O modelo é composto por um conjunto finito de abstrações. Por padronizar a interface destas abstrações, os componentes podem ser combinados entre si através da exportação e da importação de suas interfaces padronizadas. A especificação da composição dos componentes é chamada de equação de tipo. Um sistema alvo é definido por um conjunto de equações de tipo (Fig. 2.9).

```
ds =  {  bintree[ ds ],      // arvore binária
         dlist[ ds ],        // lista duplamente encadeada
         odlist[ ds ],       // lista duplamente encadeada
ordenada
         heap[ mem ],        // armazenamento do tipo heap
         array[ mem ],       // armazenamento do tipo seqüencial
         ... }
mem = {  transient,          // memória transiente
         persistent,         // memória persistente
         ... }
Exemplo de equação de tipo:
   Estrutura de dados com elementos armazenados numa árvore
biná-
   ria, os nós estão ligados numa lista duplamente encadeada
orde-
   nada e estão armazenados numa heap em memória transiente.
   estr = conceptual[ bintree [ odlist[ heap[ transient ] ] ]
];
```

Figura 2.9 Equação de tipo

Batory e et.al. em [BSL 1998] registram uma séria carência de ferramentas que simplifiquem a construção de geradores. Durante a construção de

geradores baseados no modelo GenVoca, 60% do esforço é dedicado à construção da infra-estrutura para a programação independente de domínio (ex.: linguagem de especificação de componente, linguagem de composição de componente, etc.). Para solucionar este problema, foi desenvolvido o *Jakarta Tool Suite* (JTS) que é um conjunto de ferramentas independentes de domínio que podem ser utilizadas na extensão de linguagens de programação através de construtores específicos de um domínio. O JTS é um gerador do tipo GenVoca onde as linguagens e suas extensões são encapsuladas como componentes reutilizáveis. O JTS disponibiliza rotinas para navegação em estruturas do tipo AST. O JTS é escrito em Jak, que é um superset da linguagem Java que confere características de meta-programação (ex.: construtores de AST) à linguagem Java padrão (Fig. 2.10).

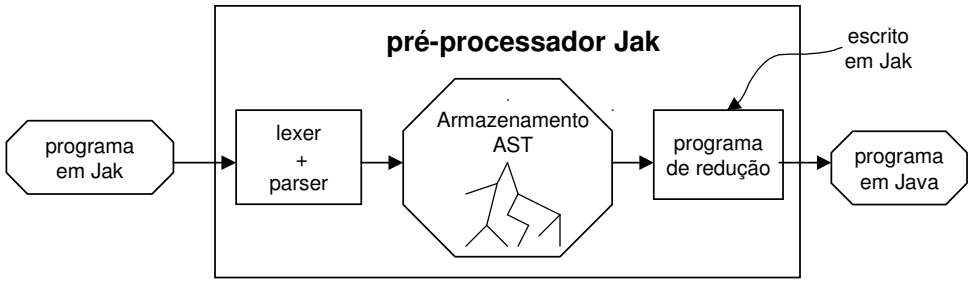


Figura 2.10 Transformação do programa em Jak

2.5 – *Frameworks*

Johnson em [Johnson 1997] 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* [Fontoura 1999]. Esta definição também pode ser aplicada para um gerador de artefatos. Em função destas semelhanças procuramos buscar na literatura referente a *framework* [Schmid 1996, Schmid 1997, BGKLRZ 1997, BMA 1997, CHSV

1997, FS 1997, Johnson 1997, GHJV 1995, Pree 1996] 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) [Pree 1996], 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. 2.11). 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*.

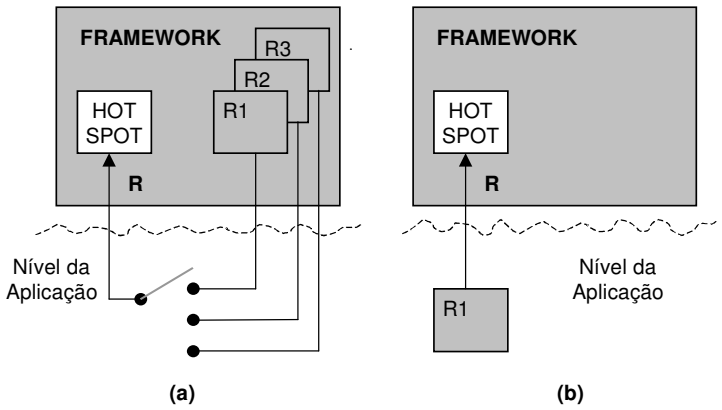


Figura 2.11 Mecanismo de *hot-spot*: (a) no caso do *black-box*, o usuário tem que escolher uma das classes fornecidas pelo *framework* (b) no caso do *white-box*, o usuário tem que construir uma classe que vai ser usada pelo *framework* [Schmid 1999]

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.

2.6 – Processo de Construção de Geradores

Nesta seção, descreveremos algumas propostas apresentadas para processo de construção de geradores. Esses processos descrevem uma forma sistemática de utilização das tecnologias descritas nas seções anteriores deste capítulo.

2.6.1 – Processo baseado em meta-Geradores

Os trabalhos sobre o meta-gerador *Stage* (seção 2.2.2), além de descreverem a arquitetura do meta-gerador, apresentam também o processo de construção utilizado [Cleaveland 1988,CK 1988, Sindelar 1990]. Este processo é composto de sete etapas:

1. Reconhecimento do domínio

Verificar se o domínio do problema apresenta características favoráveis para a aplicação de gerador de artefatos. Quanto mais padrões ou similaridades forem encontrados no domínio, mais adequada será a solução via gerador.

2. Definição do escopo do domínio

Definir as interfaces do gerador, determinando o escopo dos problemas que serão resolvidos pelo gerador.

3. Definição do modelo do gerador

O entendimento sobre um gerador é facilitado caso ele esteja baseado em algum modelo. As formas internas, a especificação, e as saídas do gerador são expressas utilizando elementos do modelo. Conjuntos, sistemas lógicos formais, máquinas de estados-finitos são alguns exemplos de modelos para um gerador.

4. Identificação das partes variantes e invariantes

A parte variante depende da especificação. Já a parte invariante corresponde a detalhes de implementação, com que o usuário do gerador não

precisa se preocupar. Um outro exemplo de parte variante é a rotina de escape.

5. Definição da entrada da especificação

Definição da forma de entrada da especificação. A especificação pode estar em formato textual, em formato diagramático, ou ser capturada através de diálogos (ex.: *wizard*).

6. Definição dos Artefatos

Definição dos artefatos a serem gerados.

7. Implementação do Gerador

Esta etapa é automatizada através do meta-gerador, que a partir da leitura dos arquivos de definição criados nas etapas anteriores produz o gerador de artefato para o domínio do problema.

2.6.2 – Processo baseado em Famílias de Aplicações

Coplien et.al. em [CHW 1998] descrevem a experiência acumulada dentro da empresa Lucent Technologies na criação de famílias de aplicações em cerca de 25 diferentes domínios utilizando o método de análise de escopo, c*ommonalty*, e variabilidade (SCV – *Scope, Commonalty and Variability analysis*). Este método permite ao desenvolvedor identificar e analisar de forma sistemática a família de aplicações que está sendo desenvolvida. Este método ajuda o desenvolvedor a:

- criar um *design* que promova o reúso e que facilite a evolução;
- antecipar a ocorrência de pontos de falha e de sucesso na evolução;
- identificar oportunidades para automatizar a criação de membros da família.

Em termos de teoria de conjuntos, o escopo seria um conjunto de objetos, a *commonalty* seriam premissas que permaneceriam verdadeiras para todos os elementos do escopo, e a variabilidade seriam premissas que seriam verdadeiras para somente alguns elementos do escopo. Segundo os autores, a análise SCV pode ser aplicada empregada em diferentes atividades de dentro da engenharia de software, desde a programação até na construção de

frameworks. Para os autores a análise SCV envolve a realização de cinco etapas:

1. Estabelecer o escopo: Identificar a coleção de objetos a ser analisada;
2. Identificar as *commonalties* e variabilidades: Identificar as partes fixas e variáveis.
3. Restringir as variabilidades: Para cada variabilidade, determinar os limites permitidos e o tempo de ligação (ex.: *runtime*, tempo de construção, etc.).
4. Explorar as *commonalties*: Buscar exaustivamente por código comum no sentido de aumentar o reúso. As *commonalties* são compartilhadas pelos membros de uma família de aplicações.
5. Acomodar as variabilidades: Para cada variabilidade, determinar a forma de implementação.

Os autores assinalam que, quando as *commonalties* são invariantes e as variabilidades são precisamente definidas, surge uma excelente oportunidade para automação do desenvolvimento.

2.6.3 – Processo baseado em *Frameworks*

Geralmente um *framework* é resultado da experiência acumulada no desenvolvimento de aplicações no domínio do *framework* [CHSV 1997]. Schmid em [Schmid 1996, 1997, 1999] apresenta um processo de construção de *frameworks* baseado em ciclos de desenvolvimentos, onde cada ciclo acrescenta novas variabilidades (*hot-spots*) ou funcionalidades fixas (*frozen-spots*) 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* (Fig. 2.12).

Uma atividade importante dentro do processo de construção do *framework* é a atividade de *hot-spot mining* [Pree 1999] cujo objetivo é a identificação dos *hot- spot*. [Schmid 1999] e [Pree 1999] mostram que uma das formas de identificar possíveis *hot-spots*, é por meio da avaliação das possibilidades de evolução que a estrutura de classes do *framework* suporta. Esta avaliação poderia ser feita através da análise das modificações realizadas em protótipos ou em sistemas legados do domínio da aplicação. Pree em [Pree 1999]

considera que os *Use cases* [SWJ 1998] possam ser fontes para identificação de *hot-spots*. Já Demeyer et.al. [DMNS 1997**]**, durante a construção de *frameworks* para sistemas abertos, utilizaram o conceito de eixos de variabilidade (ex.: interoperabilidade, distribuição e extensibilidade) para orientar a identificação dos *hot-spots*.

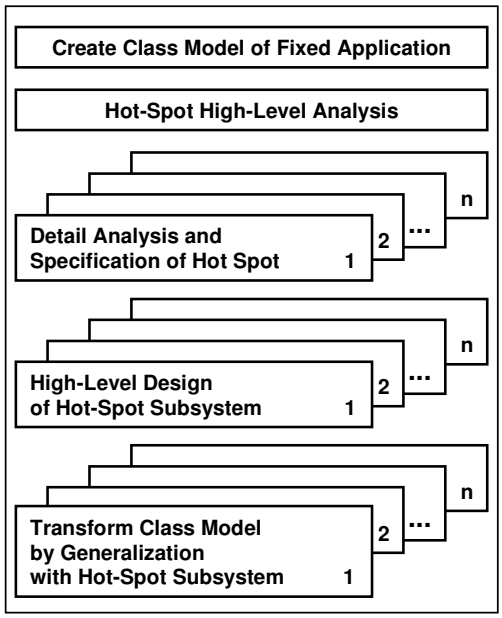


Figura 2.12 Atividades de *design* de um *framework* [Schimd 1999]

Roberts e Johnson [RJ 1996] apresentam o processo de desenvolvimento de *framework* sob a ótica da evolução de um *framework* (Fig. 2.13). 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. Como pode ser observado na figura 2.13, num instante **t** qualquer, a implementação de um *framework* pode envolver várias técnicas:

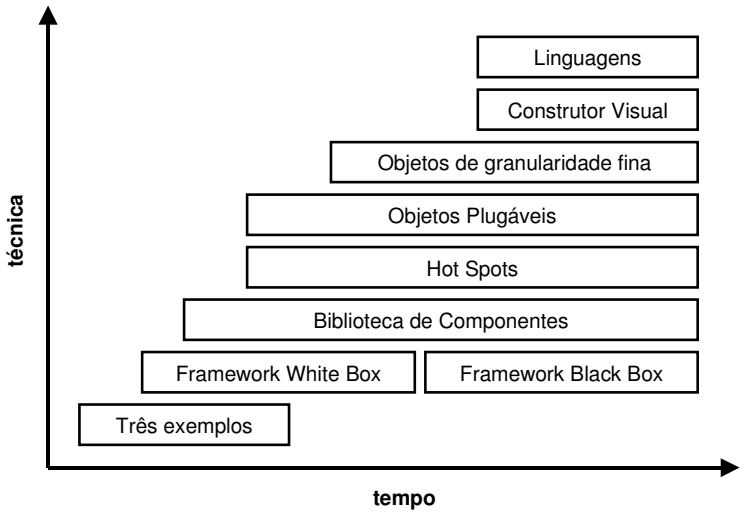


Figura 2.13 Evolução de um *framework*

- **Três exemplos**. Esta abordagem é utilizada quando a organização não tem experiência no domínio do problema. Pelo menos 3 aplicações concretas são desenvolvidas para fornecer as abstrações necessárias para a construção de um *framework*.
- ***Framework* do tipo whitebox**. Analisando as aplicações instanciadas, o desenvolvedor pode identificar partes fixas comuns, que poderão ser colocadas em classes abstratas do *framework*.
- **Biblioteca de componentes**. Com a utilização do *framework*, pode-se criar uma biblioteca de componentes a partir de objetos reutilizados em diferentes aplicações instanciadas.
- **Hot-spots.** Com a utilização do *framework*, também é possível identificar os pontos que são sempre codificados na instanciação de diferentes aplicações. Estes pontos são potenciais *hot-spots*.
- **Objetos "plugáveis".** No lugar de criar uma nova subclasse que difira ligeiramente de uma outra subclasse, o desenvolvedor mantém apenas uma subclasse no *framework*. Esta subclasse possui mecanismos de adaptação que permite a sua variação.

- **Objetos de granularidade fina**. Os objetos do *framework* são quebrados em unidades menores que facilitam a composição de novos objetos, reduzindo a duplicidade de código.
- ***Framework* do tipo *blackbox***. Com a utilização do *framework*, algumas classes abstratas são transformadas em classes concretas, que permitem a construção de novas aplicações a partir da composição dos objetos das classes existentes.
- **Construtor visual**. A instanciação do *framework* é feita através de editores de diagramas que permitam evidenciar a conexão dos objetos do *framework*.
- **Ferramentas de linguagem**. São criadas linguagens específicas que auxiliam a verificação da composição da aplicação instanciada.

Em [BMMB 1997] são identificadas 3 etapas no processo de desenvolvimento de um *framework* (Fig. 2.14):

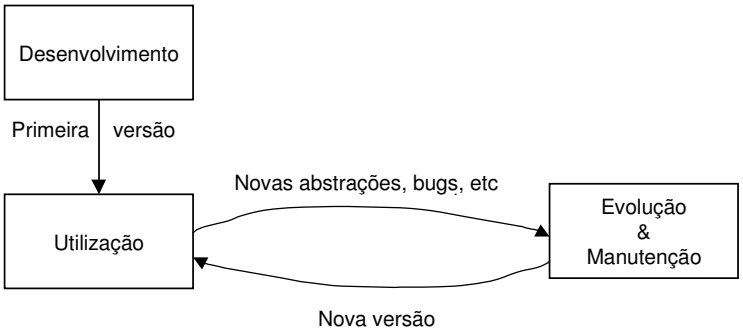


Figura 2.14 Processo de desenvolvimento de um *framework*

1. **Desenvolvimento**. Os resultados desta etapa são: modelo da análise de domínio, arquitetura do *framework*, *design* do *framework* e implementação do *framework*.

2. **Utilização**. Esta etapa é também denominada de instanciação do *framework* ou de desenvolvimento da aplicação. Nesta etapa o *framework* é usado na construção de aplicações.

3. **Evolução & Manutenção**. Novas versões do *framework* são criadas através de: correção de erros identificados nas aplicações geradas,

novas abstrações devido a mudanças no domínio do problema ou devido a generalização de elementos comuns a várias aplicações.

Em [LN 1995] é descrito detalhadamente o processo de construção de um *framework* OO (fig 2.15):

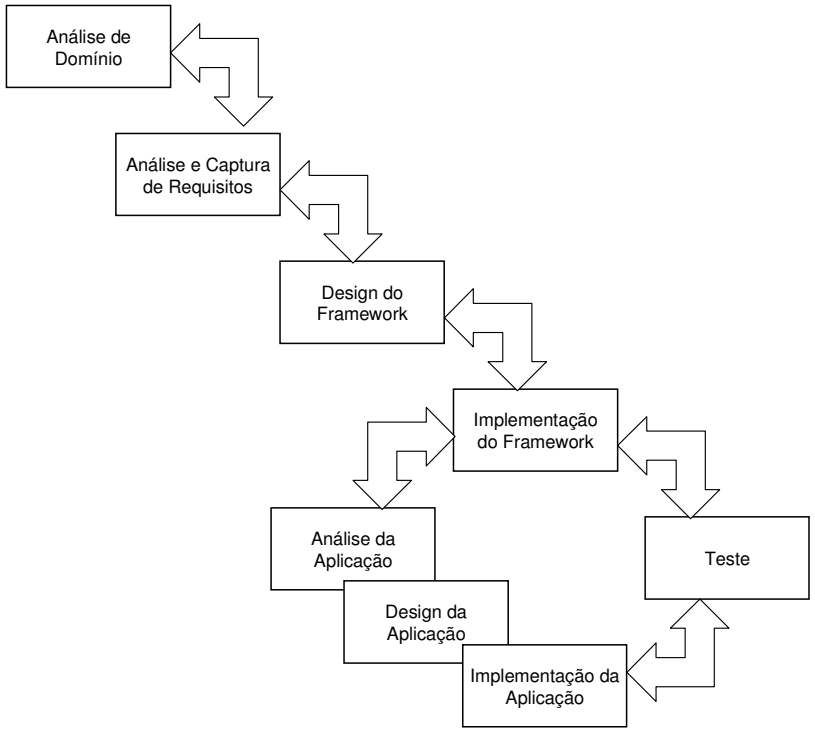


Figura 2.15 Processo de detalhado de desenvolvimento de um *framework*

Em [Fontoura 1999] é apresentado o processo de construção de *frameworks* sob um enfoque transformacional, mostrando como uma adequada linguagem de *design* pode favorecer a utilização de ferramentas ao longo do processo (Fig. 2.16).

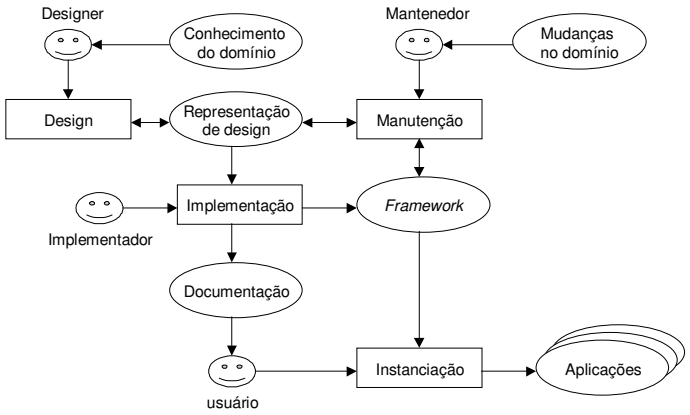


Figura 2.16 Processo de desenvolvimento de um *framework*

2.7 – Resumo

Este capítulo procurou dar uma visão geral do contexto de trabalho na área de processo de construção de geradores de artefatos.

Um gerador de artefatos é uma ferramenta que produz um artefato a partir de sua especificação de alto nível. A sua utilização pode proporcionar elevados ganhos de produtividade. A arquitetura de um gerador de artefatos é formada por componentes que realizam as transformações da especificação até o produto final. Um ponto importante da arquitetura é a estrutura de armazenamento, que apresenta características que facilitam as transformações.

O processo de construção de geradores de artefatos pode ser automatizado através da utilização de meta-geradores. O aspecto “meta” do meta-gerador deve-se a externalização da descrição tanto das linguagens de especificação como dos artefatos-alvo do gerador a ser construído. Os meta-geradores apresentados na literatura, apesar de possuírem semelhanças na sua arquitetura interna, apresentam diferenças quanto ao tipo de especificação manipulada e quanto à forma de armazenamento.

Uma boa fonte de referências para construção de geradores é a literatura sobre *frameworks*. Os conceitos e as experiências no desenvolvimento de *frameworks* podem ser aplicados dentro do contexto de geradores de artefatos.

Em relação ao processo de desenvolvimento de geradores, o processo utilizado na AT&T [CK 1988] em 1988 mostra o uso sistemático de um meta-gerador. Já os processos mais recentes de construção de *frameworks*, são caracterizados por possuírem tanto atividades relacionadas com a construção, como atividades relacionadas com o uso e instanciação do *framework*. Dentre estas atividades, destaca-se a atividade de identificação de *hot-spots* (*hot-spot mining*), que pode ser utilizada para identificar as partes fixas e variáveis de um gerador.

Capítulo 3- Métodos, Técnicas, e Ferramentas para Construção de Geradores de Artefatos

Este capítulo apresenta a infra-estrutura utilizada para a construção de geradores de artefatos. Inicialmente são apresentados os requisitos para a infra-estrutura de geração. Em seguida, é descrita a implementação do ambiente de geração através de um CASE. A seguir, é descrito o método de hot-spot mining utilizado para a identificação dos hot-spots de um artefato-exemplo. Finalmente, são apresentadas as transformações de generalização dos hot-spots.

3.1 – Requisitos para Métodos, Técnicas e Ferramentas para Construção de Geradores de Artefatos

A maioria dos geradores analisam uma especificação para fazer a geração do artefato [Cleaveland 1988, MM 1993, Neighbors 1989]. Para construir um gerador que utilize esta abordagem, o desenvolvedor precisa definir uma linguagem de especificação, um analisador para esta linguagem, e desenvolver um transformador que converta a árvore de sintaxe abstrata resultante no artefato a ser gerado. Estas atividades requerem conhecimentos de linguagens formais e de construção de compiladores que não são de domínio da maioria das organizações. Por causa disto, resolvemos adotar uma outra abordagem de geração.

Como resultado de nossa experiência no desenvolvimento da ferramenta COMPASSO (cap. 1) identificamos os seguintes requisitos para os métodos, técnicas e ferramentas a serem utilizados no processo de construção de um gerador de artefatos:

- **R1. Simplificar as transformações.** As transformações realizadas da especificação até a sua implementação devem ser simplificadas, pois transformações mais complexas dependem de conhecimentos na área de compiladores e de linguagens formais que podem não ser dominados pela equipe de desenvolvimento.

- **R2. Promover o reúso.** A nossa experiência no emprego do reúso do tipo "cortar/colar/modificar" mostrou que boa parte do código não dependia da especificação e não variava entre as diversas aplicações criadas. Este fato deve ser aproveitado durante o processo de construção do gerador.

Esta lista de requisitos pode ser estendida com os requisitos apresentados em [MM 1993], descrevendo as características desejáveis para um ambiente de geração:

- **R3. Possibilidade de definição de novas linguagens de especificação.** O ambiente de geração deve poder produzir geradores que aceitem tanto especificações na forma textual como na forma diagramática. O ambiente deve possuir mecanismos que permitam a alteração das linguagens de especificação disponíveis.
- **R4. Estrutura interna genérica.** A estrutura interna deve permitir o armazenamento de diferentes tipos de especificações.
- **R5. Ampla variedade de rotinas de acesso à estrutura interna.** Para facilitar a codificação de transformadores, o ambiente deve disponibilizar diferentes rotinas de acesso à sua estrutura interna.
- **R6. Possibilidade de definição de novos artefatos.** O ambiente de geração deve possuir mecanismos que agilizem a definição de novos artefatos.

Em nosso trabalho, a adoção de um processo de construção de software "dirigido por exemplo" permitiu que os requisitos R1 e R2 fossem alcançados. 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. Uma boa parte do código do gerador corresponde a trechos inalterados (*frozen-spots*) dos arquivos-fonte do artefato-exemplo. Já as partes variáveis do gerador são completadas através de funções que acessam a especificação armazenada no repositório do ambiente de geração. As partes variáveis e fixas do gerador são identificadas através da técnica de *hot-spot mining* (seção 3.3).

Confrontando os requisitos R3, R4, R5, e R6 com as características de uma ferramenta CASE (Computer-Aided Software Engineering), observarmos que o papel de ambiente de geração pode ser exercido por um CASE que tenha facilidades de customização. Em [Hohenstein 2000, Milicev 2000] são descritos trabalhos nos quais ferramentas CASE são utilizadas como ambiente de geração. Em [Hohenstein 2000] 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.

Por possuir as características desejáveis para um ambiente de geração e pela experiência acumulada na sua utilização, em nosso trabalho, adotamos o CASE Talisman [Staa 1993] como gerador de geradores de artefatos. Ressaltamos que qualquer CASE que possua as características acima relacionadas pode ser utilizado com ambiente de geração. Em [Hohenstein 2000, Milicev 2000] é relatada a utilização do CASE Rational Rose [Rational 1998] como ambiente de geração.

A seguir detalharemos os métodos, técnicas e ferramentas utilizados para alcançar os requisitos acima listados.

3.2 –Ambiente para construção de geradores: Talisman

O ambiente de desenvolvimento Talisman [Staa 1993] (Figura 3.1) foi escolhido para exercer o papel do meta-gerador a ser utilizado no nosso processo de construção de geradores.

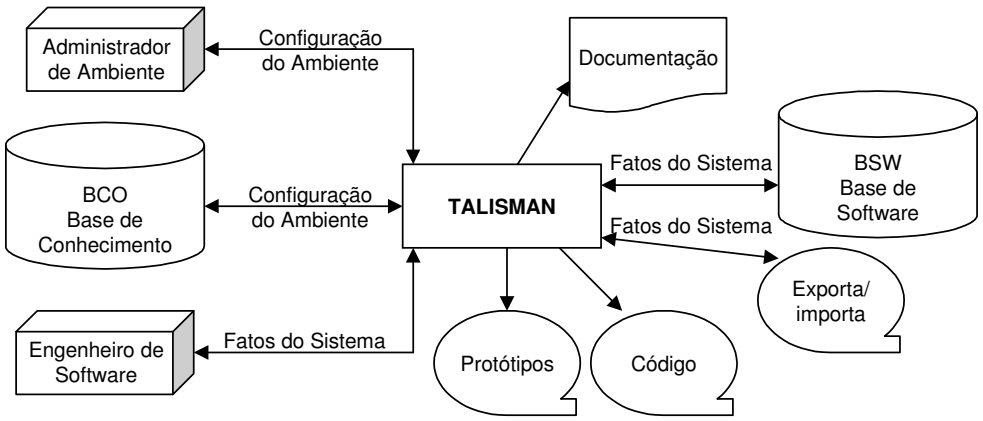


Figura 3.1 Visão geral do meta-CASE Talisman

O repositório de dados armazena as informações relativas ao sistema que está sendo desenvolvido. Estas informações podem ser: diagramas, nomes, fragmentos de texto, fragmentos de código, relacionamentos entre objetos, etc. A base de conhecimento define as propriedades do ambiente Talisman. As principais definições registradas nesta base são: relação das linguagens de representação disponíveis, descrição do meta-modelo, definições das linguagens gráficas, programas de definição das telas de especificações, programas de manipulação do repositório de dados e programas de exportação e importação do repositório de dados.

Além das características desejáveis para um ambiente de geração, o Talisman possui mecanismos bastante adequados para construção de geradores, entre elas:

- **Customização completa do repositório de dados.** Talisman permite a criação de novas classes de objetos, novos atributos de objetos e novas relações entre os objetos que compõem o seu meta-modelo. Em função disto, a estrutura do componente de armazenamento do meta-gerador pode ser facilmente evoluída.

- **Estrutura interna do repositório.** O repositório de dados do Talisman armazena objetos, diagramas e dicionários. Dicionários armazenam referências a objetos de uma mesma classe de objetos. Uma classe de objeto corresponde a um elemento do meta-modelo de uma linguagem de representação (ex.: entidade, relacionamento, processo, atributo, etc.). Um

objeto é composto de diversos atributos (Figura 3.2). Os atributos estão divididos nas seguintes categorias:

- **Nome**: É a identificação única externa do objeto.
- **Aliases**: São seqüências de caracteres usadas geralmente para nomear dados elementares. (ex.: tipo de dado, tamanho de um atributo, etc.). Cada objeto pode ter até 250 aliases.
- **Fragmentos de texto**: São linhas de texto usadas geralmente para registrar descrições (ex.: texto de documentação, descrições informais, texto de código, etc.). Cada objeto pode ter até 250 fragmentos.
- **Relações**: São vínculos entre objetos (ex.: entidade X atributo, processo X fluxo de dado, etc.). Durante a edição de diagramas ou textos, as relações são automaticamente estabelecidas ou atualizadas, liberando o usuário da necessidade de explicitamente criá-las. As ferramentas instanciadas pelo Talisman são responsáveis pela criação e manutenção da rede de relações existentes dentro do repositório. Cada relação individual tem semântica específica e os objetos podem relacionar-se entre si. Os auto-relacionamentos também são possíveis. Cada objeto pode ter até 250 relações.

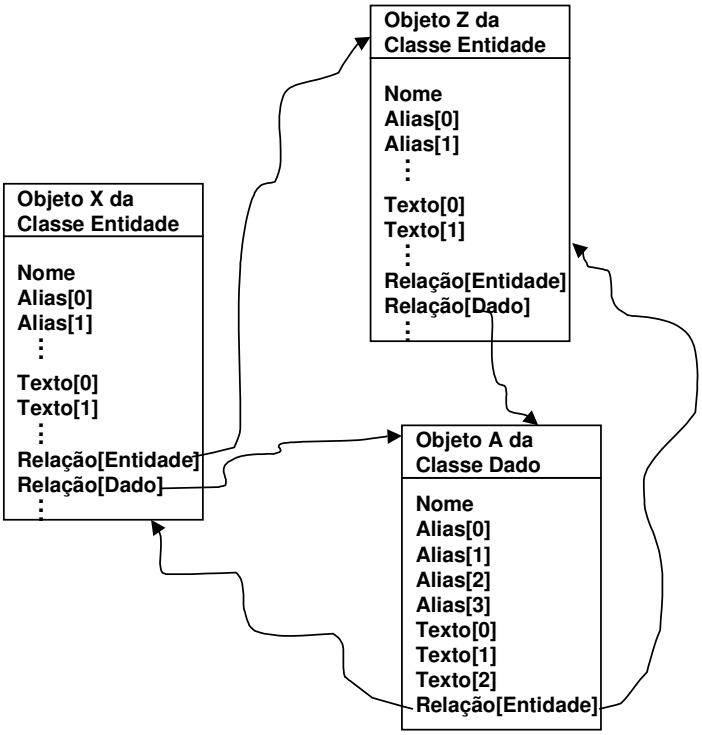


Figura 3.2 Relação entre objetos do meta-modelo do Talisman

- **Linguagem para manipulação do repositório.** Por meio de sua linguagem nativa, Talisman é capaz de criar rotinas simples para navegação e manipulação do conteúdo do repositório de dados. A linguagem apresenta comandos para manipulação de objetos do meta-modelo e comandos para manipulação de listas de objetos. Esta linguagem pode ser usada tanto para codificar um módulo de edição de especificação como para codificar as regras de transformação utilizadas na geração de um artefato a partir de sua especificação (Figura 3.3).

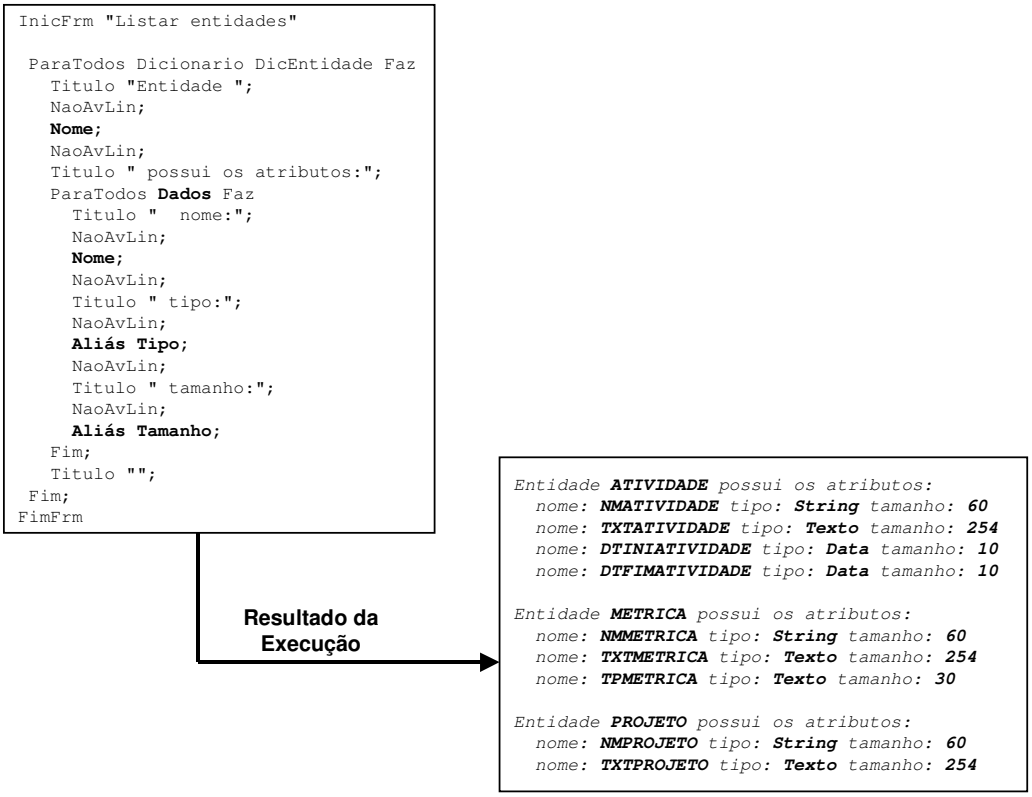


Figura 3.3 Trecho de código na linguagem nativa do Talisman

A Figura 3.4 apresenta a visão esquemática da utilização do Talisman como meta-Gerador. Os programas de edição da especificação e de descrição do artefato são arquivos externos ao meta-CASE, e ambos contêm programas escritos na linguagem Talisman. O compilador transforma estes arquivos e armazena o executável (*byte code*) do módulo de edição da especificação e do gerador de artefato numa estrutura denominada base de conhecimento. Após a compilação, estes programas tornam-se parte das ferramentas CASE instanciadas. O módulo de edição da especificação (ex.: diagrama de entidade-relacionamento mais dicionário de dados) registra a especificação da aplicação na base de software. Quando o usuário ativa o gerador de artefato, a base de software é explorada, acessada e os dados recuperados são formatados para gerar o artefato. O artefato pode ser composto de diferentes fragmentos de código, cada qual escrito em diferentes linguagens de programação.

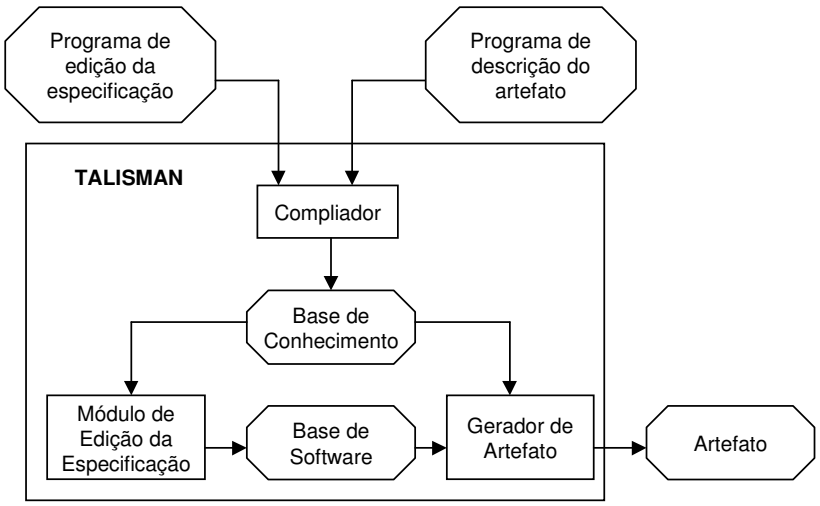


Figura 3.4 Talisman: Ambiente para construção de geradores

O programa de edição da especificação (Figura 3.5) contém as funções de definição dos formulários do dicionário de dados (Figura 3.6). Este programa contém comandos de manipulação dos objetos e relações do meta-modelo do Talisman bem como comandos de formatação dos campos da tela. Os campos exibidos pelos programas são utilizados para a entrada das informações necessárias à geração do artefato.

```
InicFrm "Editar Atributo"
  Titulo "Nome: ";
  NaoAvLin;
  Nome;
  Titulo "Descrição:";
  Texto TxtDescr;
  Titulo "Tipo:";
  NaoAvLin;
  Aliás Tipo;
  NaoAvLin;
  Titulo "  Tamanho:";
  NaoAvLin;
  Aliás Tamanho;
FimFrm
```

Figura 3.5 Trecho de um programa de edição da especificação

Nome: **Endereço**
Descrição:
Endereço completo contendo logradouro, número, apto, bloco, etc.
Tipo: **String** Tamanho: **60**

Figura 3.6 Exemplo de uma tela de entrada de dados

O programa de descrição do artefato (Figura 3.7) é responsável pela geração do artefato (Figura 3.8). O programa contém comandos de

manipulação dos objetos e relações do meta-modelo do Talisman e comandos de formatação. O artefato gerado é uma composição de partes fixas (independentes da especificação) com os dados lidos da especificação armazenada na base de software.

```
Titulo "<div id=\"titulo\">";
Titulo "   Cadastro de ";
NaoAvLin;
Frm "fTituloClasse"( oFpaCLASSE,1);
Titulo "</div>";
Titulo "<div id=\"formulario\">";
Titulo " <table>";
Titulo "  <tr>";
Titulo "   <td align=\"right\">";
Titulo "     <b>Nr.Reg</b>";
Titulo "   </td>";
Titulo "   <td>";
Titulo "     <input type=\"text\" name=\"ID";
NaoAvLin;
Frm "fNomeClasse"( oFpaCLASSE,1);
NaoAvLin;
Titulo "\" class="clID";
Frm "fNomeClasse"( oFpaCLASSE,1);
NaoAvLin;
Titulo "\"";
Titulo "         readonly maxlength=\"8\">";
Titulo "   </td>";
Titulo "  </tr>";
```

Figura 3.7 Trecho de um programa de descrição de artefato

```
<div id="titulo">
   Cadastro de Produto
</div>
<div id="formulario">
 <table>
  <tr>
   <td align="right">
     <b>Nr.Reg</b>
   </td>
   <td>
     <input type="text" name="IDPRODUTO" class="clIDPRODUTO"
         readonly maxlength="8">
   </td>
  </tr>
```

Figura 3.8 Trecho de um arquivo HTML gerado referente ao programa da Fig. 3.7

A construção de um gerador de artefato utilizando o Talisman como meta-gerador envolve a tarefa de codificação dos seus programas de edição da especificação e de descrição de artefato. Embora a construção como um todo tivesse sido simplificada, a atividade de codificação dos programas de

descrição dos artefatos ainda era demasiadamente trabalhosa. Esta atividade envolvia a recodificação dos fontes dos protótipos desenvolvidos utilizando a linguagem do Talisman. Durante esta codificação, uma boa parte do fonte original permanecia inalterada, já as partes variáveis, correspondiam a informações que deveriam ser buscadas no repositório de dados.

Na próxima seção, detalharemos a proposta empregada para identificar as partes variáveis do artefato, proporcionando a simplificação da atividade de codificação dos programas de descrição de artefato.

3.3 – *Hot-Spot* Mining

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 *hot-spot mining* [Pree 1999]. 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 [MM 1993] é relatado um exemplo de gerador de aplicações onde as variabilidades correspondem a apenas 16% do código. Larnegan e Grasso em [LG 1989] 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 [FHLS 1997]. 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:

- **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:

- **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 de escape, consideremos que os pontos de inclusão das rotinas correspondem a *hot-spots* do tipo "ilimitado".

A implementação de um hot-spot pode envolver a utilização de outros hot-spots. Por exemplo, suponha que num gerador de aplicações de manutenção de base de dados, a forma de exclusão de um registro seja um hot-spot. Um tipo de exclusão verifica se há referências ao registro a ser excluído, se

existirem, a exclusão não é feita. Numa outra forma, num primeiro passo, são excluídas todas as referências ao registro a ser excluído, e no passo seguinte o registro é excluído. Neste exemplo, dependendo da forma de exclusão escolhida, um conjunto de hot-spots mais simples é utilizado para efetivamente implementar a forma de exclusão escolhida.

A seguir, apresentaremos os tipos de *hot-spots* encontrados em nossos experimentos.

3.3.1 – Hot-Spot: Modo de Adaptação: Substituir - 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 outro trecho pertinente ao contexto do novo programa. Estes pontos podem ser classificados como sendo *hot-spot* do tipo "substituir".

Em nossos experimentos, este tipo de *hot-spot* foi identificado, por exemplo, nas seguintes situações:

- O programa `ClienteBD.java` de acesso à tabela `Cliente` foi usado como base para criação do programa `ProdutoBD.java` de acesso à tabela `Produto`. Durante a criação do programa `ProdutoBD.java`, as ocorrências da *string* `Cliente` foram substituídas pela *string* `Produto`.
- O programa `Cliente.java` de definição da classe `Cliente` foi usado como base para criação do programa `Produto.java` de definição da classe `Produto`. Durante a criação do programa `Produto.java`, o trecho da declaração dos atributos da classe `Cliente` foram substituídos pelos atributos da classe `Produto`.

Dentro do programa de descrição do artefato, um *hot-spot* do tipo "substituir" deve ser substituído por uma regra de transformação que coloque no local do *hot-spot* informações extraídas do repositório de dados (Figura 3.9).

Trecho do arquivo original	*Hot-spot* do tipo Substituir

public class Produto
{
[...]
}
	public class **NomeEntidade**
	{
	[...]
	}

public class Produto
{
private String sIDPRODUTO;
private String sNMPRODUTO;
private String txtOBS;
private String dtDTCRIACAO;
}
	public class **NomeEntidade**
	{
	AtributosEntidade
	}

Figura 3.9 Exemplos de *hot-spots* do tipo "substituir"

3.3.2 – Hot-Spot: Modo de Adaptação: Habilitar - 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".

Em nossos experimentos, este tipo de *hot-spot* foi identificado, por exemplo, na página `HTML` de manutenção de uma tabela referente a uma entidade. Caso a entidade possua relacionamentos `N:N`, a página conterá *links* para página de manutenção de cada relacionamento `N:N`. Quando este arquivo `HTML` foi utilizado como base para o reúso, o trecho do arquivo referente ao links do relacionamento `N:N` somente é expandido para as entidades que tinham relacionamentos do tipo `N:N`.

Dentro do programa de descrição do artefato, um *hot-spot* do tipo “habilitar” deve ser substituído por uma regra de transformação que tenha pré-condições para ser aplicada. Se tal regra for aplicada, sua aplicação é similar à aplicação da regra referente à um *hot-spot* do tipo “substituir”. (Figura 3.10).

Trecho do arquivo original	***Hot-spot* do tipo Habilitar**

public class Produto
{
`private String sIDPRODUTO;`
`private String sNMPRODUTO;`
[...]
 	
`private ATIVIDADE oAtivProd;`
`private PROJETO oProjProd;`
}
	public class NomeEntidade
	{
	AtributosEntidade
	
	Se existe relacionamentos N:N
	DeclaraçãoRelacionamento
	}

Figura 3.10 Exemplos de *hot-spots* do tipo “habilitar”

3.3.3 –Hot-Spot: Modo de Adaptação: Aumentar - 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”.

Dentro do programa de descrição do artefato, um *hot-spot* do tipo “aumentar” deve ser substituído por uma regra de transformação que busque no repositório de dados o trecho de código da rotina de escape. (Figura 3.11).

Trecho do arquivo original	***Hot-spot* do tipo Aumentar**

`if ( sTipo == "Numerico")`
`{`
`/** CODIGO GERADO ***/`
 	
`if isNumber( nQtdhoras.`
`value ) == "N"`
`{`
`alert(`
	`if ( sTipo == "Numerico")`
	`{`
	`/** CODIGO GERADO ***/`
	
	`if isNumber(nQtdhoras.value)=="N"`
	`{`
	`alert("Qtd horas inválida ");`
	`return false;`

```
      "Qtd horas inválida ");
   return false;
  }

  /* CODIGO DA ROTINA DE ESCAPE */

  if (nQtdhoras.value > 10 )
  {
   alert("Qtd horas acima do
      limite superior ");
   return false;
  }

  if nQtdhoras.value) < 1
  {
   alert("Qtd horas abaixo do
       limite inferior ");
   return false;
  }
}
```

```
 }

  /* CODIGO DA ROTINA DE ESCAPE */

      Rotina de Escape

}
```

Figura 3.11 Exemplos de *hot-spot* do tipo "aumentar"

3.4 – Transformações de Generalização

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 (Figura 3.12). 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).

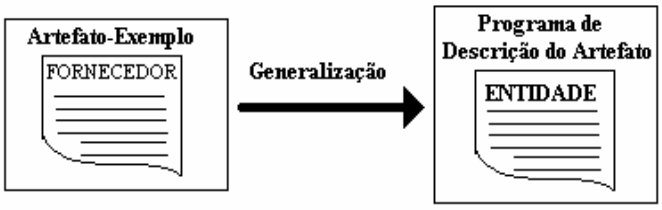


Figura 3.12 Visão esquemática da generalização

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 (Apêndice B). O arquivo de meta-descrição do artefato reduz o problema de descasamento de impedância [Hohenstein 2000, Milicev 2000] entre os domínios da especificação e da implementação. Neste arquivo, os *tags* incluídos servem para marcar as dependências da especificação. Foram criado três tipos de *tag* para tratar os tipos de *hot-spot* identificados na seção 3.3. Os *tags* criados foram chamados de: *substituição*, *bloco* e *condicional*. A próxima seção apresenta um detalhamento de cada um destes tipos.

O utilitário GERDESCR (Figura 3.13) 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.

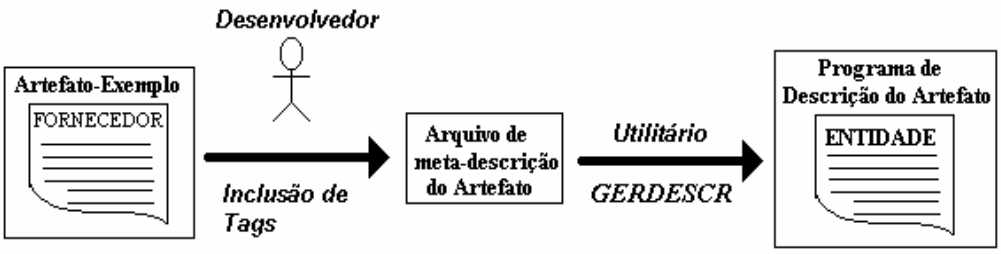


Figura 3.13 Geração automática do programa de descrição do artefato

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 (Figura 3.14).

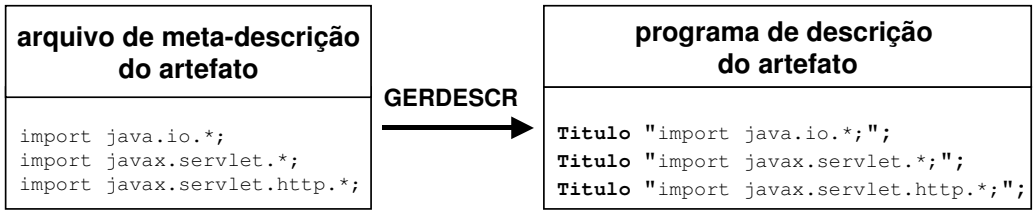


Figura 3.14 Transformação de um *frozen-spot*

- ***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.

Um arquivo de meta-descrição do artefato (Figura 3.15) está dividido em duas partes:

- **Cabeçalho**: Contém a lista de definição dos *tags* do tipo substituição 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*.

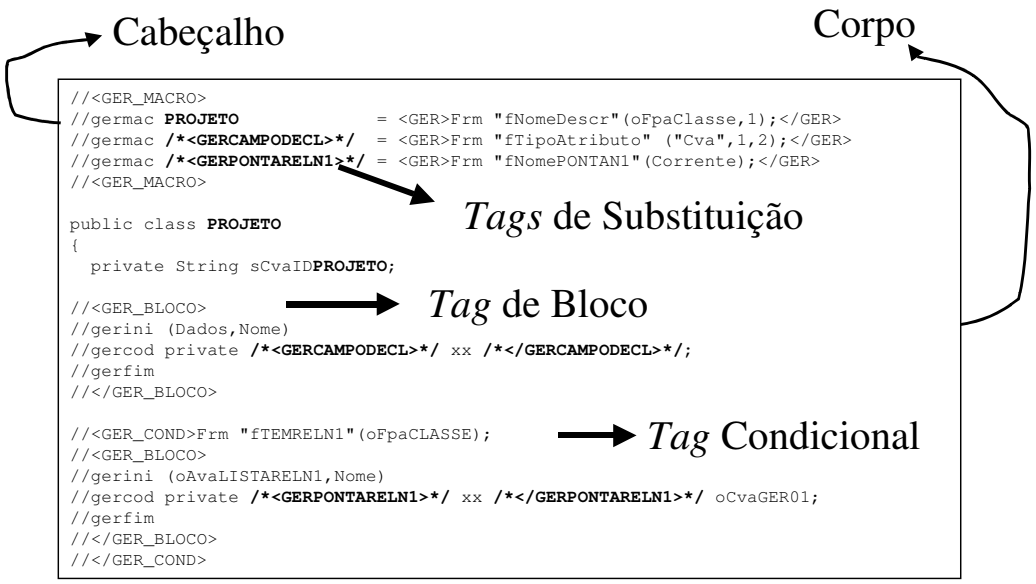


Figura 3.15 Exemplo de um arquivo de meta-descrição do artefato

3.4.1 – *Tag* de Substituição

Um *tag* de substituição (Figura 3.16 e 3.17) 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 substituição envolve uma ou mais *strings*, como por exemplo a lista de parâmetros de uma função.

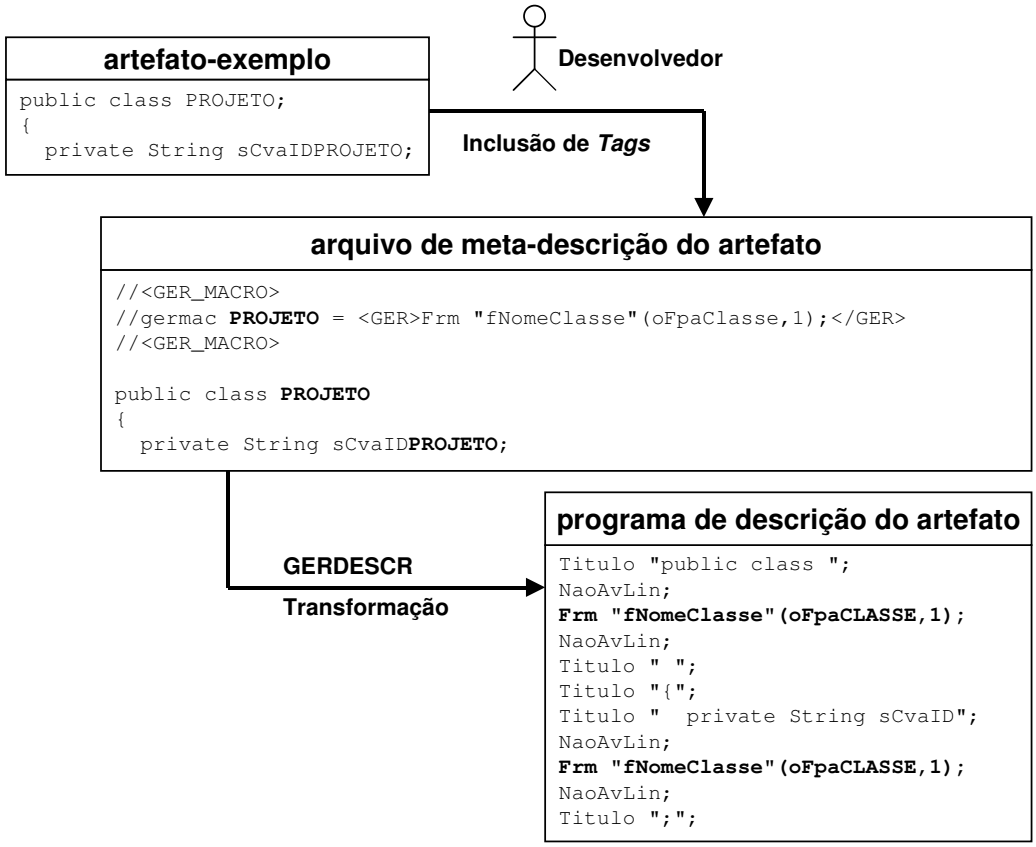


Figura 3.16 Exemplo de *tag* de Substituição

Figura 3.17 Exemplo de *tag* de Substituição

3.4.2 – *Tag* de Bloco

Um *tag* de bloco (Figura 3.18) 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.

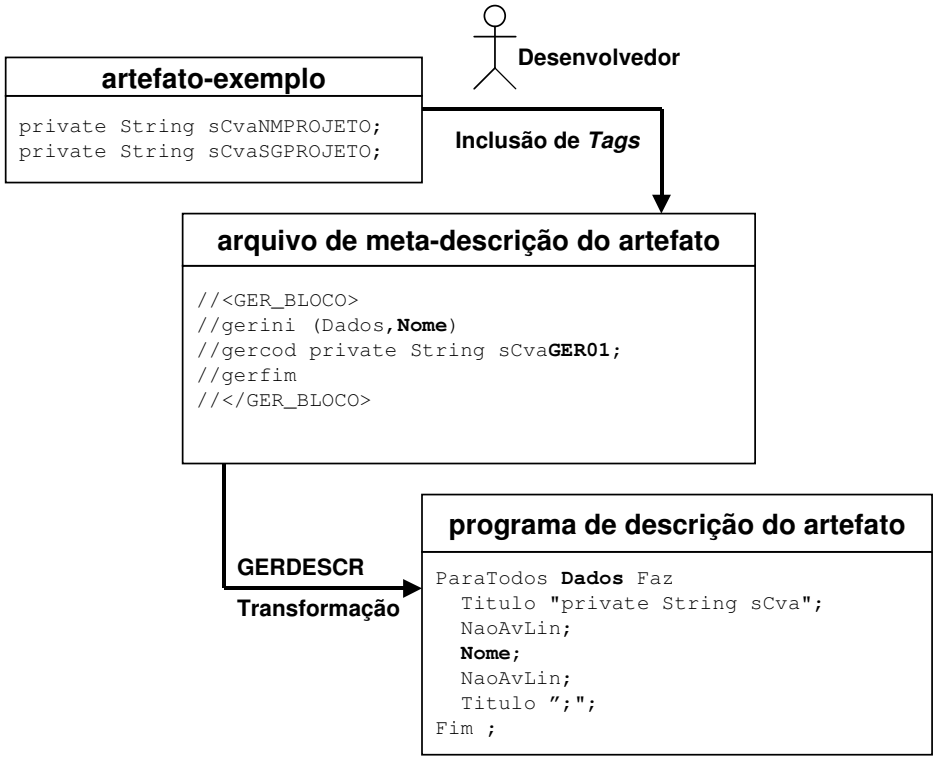


Figura 3.18 Exemplo de *tag* de Bloco

3.4.3 – *Tag* Condicional

A regra de transformação referente a um *tag* condicional (Figura 3.19 e 3.20) 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 Talisman 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.

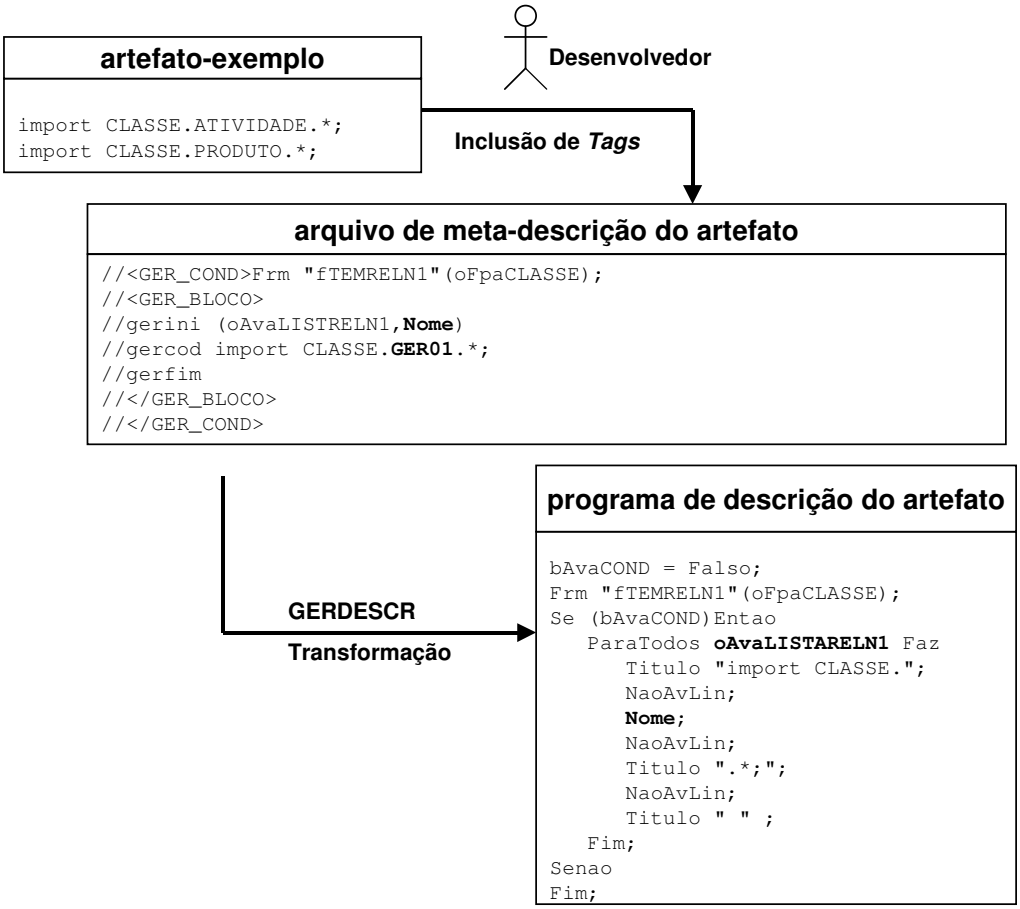


Figura 3.19 Exemplo de *tag* Condicional

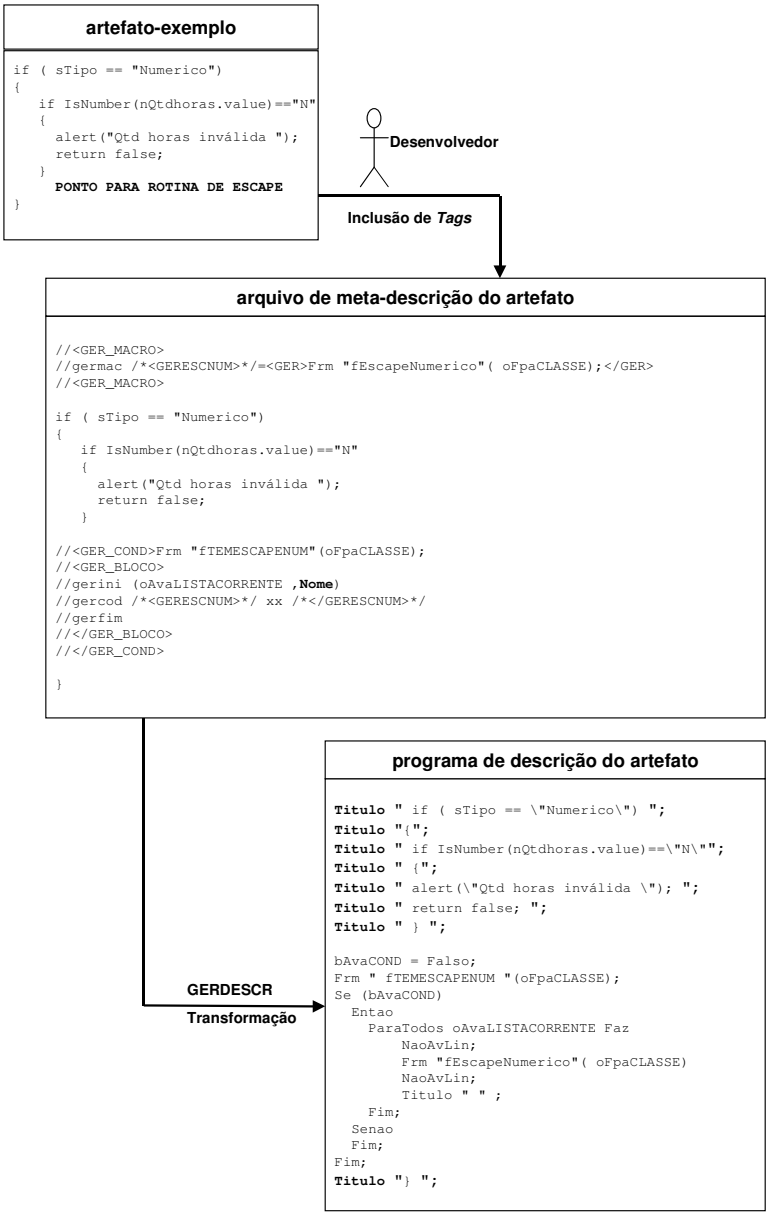


Figura 3.20 Exemplo de *tag* Condicional (rotina de escape)

3.5 – Resumo

Este capítulo procurou dar uma visão geral dos métodos, técnicas e ferramentas utilizados na construção de geradores de artefatos.

As justificativas para a adoção de ferramenta CASE como ferramenta-chave do nosso processo de construção de geradores foram apresentadas. A ferramenta CASE, dentro do processo proposto, exerce o papel de um meta-gerador de artefatos. Por possuir as características desejáveis para um ambiente de geração e pela experiência acumulada na sua utilização, em nosso trabalho, adotamos o CASE Talisman [Staa 1993] como gerador de geradores de artefatos. Ressaltamos que qualquer CASE que possua as características acima relacionadas pode ser utilizado com ambiente de geração. Em [Hohenstein 2000, Milicev 2000] é relatada a utilização do CASE Rational Rose [Rational 1998] como ambiente de geração.

A identificação de *hot-spots* é uma atividade importante para a identificação das partes fixas e variáveis que serão geradas pelo gerador de artefatos. O método de *hot-spot mining* empregado é baseado na análise das modificações efetuadas na evolução dos protótipos que foram criados para prover um maior conhecimento dentro do domínio da aplicação.

O utilitário GERDESCR foi desenvolvido para realizar as transformações de generalização dos *hot-spots* identificados. Estas transformações são extraídas de arquivos denominados de meta-descrição do artefato, que são compostos do código- fonte original marcados com *tags* que servem para orientar o utilitário na substituição dos *hot-spots* por suas respectivas regras de transformação. O GERDESCR recebe como entrada um arquivo de meta-descrição do artefato e produz como saída o programa de descrição do artefato

Capítulo 4 – O Processo de Construção de Geradores de Artefatos

Este capítulo descreve o processo de construção de geradores. Inicialmente é apresentada uma visão geral do processo e, em seguida, cada passo do processo é detalhado.

4.1 – Características do Processo Proposto

O processo de construção de geradores foi refinado e a sua aplicabilidade foi avaliada por intermédio de diversos experimentos de construção de geradores de protótipos a partir de modelo de dados. Dentre as principais características deste processo, destacamos: a abordagem *bottom-up*, o desenvolvimento evolutivo e o reúso de regras de transformação.

O processo de construção proposto utiliza uma abordagem do tipo *bottom-up* para construção do gerador de artefatos. O ponto de partida do processo de construção é a implementação de um artefato-exemplo que servirá de modelo para o gerador. O artefato-exemplo pertence ao domínio de problema alvo dos artefatos que poderão vir a ser gerados. Através da técnica de *hot-spot mining* (seção 3.3), as partes variáveis (*hot-spot*) do artefato-exemplo são identificadas. Estes *hot-spots* são pontos dependentes de informações contidas na especificação. Em seguida, cada *hot-spot* encontrado é generalizado através de sua substituição por uma regra de transformação. Esta regra, quando aplicada, coloca no lugar do *hot-spot* informações extraídas da especificação (Fig. 4.1). No caso ideal, a especificação somente conterá as informações necessárias ao processo de geração. No caso real, pode conter mais informações do que as estritamente necessárias, por exemplo, descrições e outros comentários dos itens sendo especificados. Esta estratégia de desenvolver geradores a partir de um artefato-exemplo é encontrada na literatura em artigos referentes a desenvolvimento de *frameworks*. Em [RJ 1996] é apresentado a técnica de desenvolvimento de *frameworks* a partir da construção de três exemplos de aplicação dentro do domínio do problema.

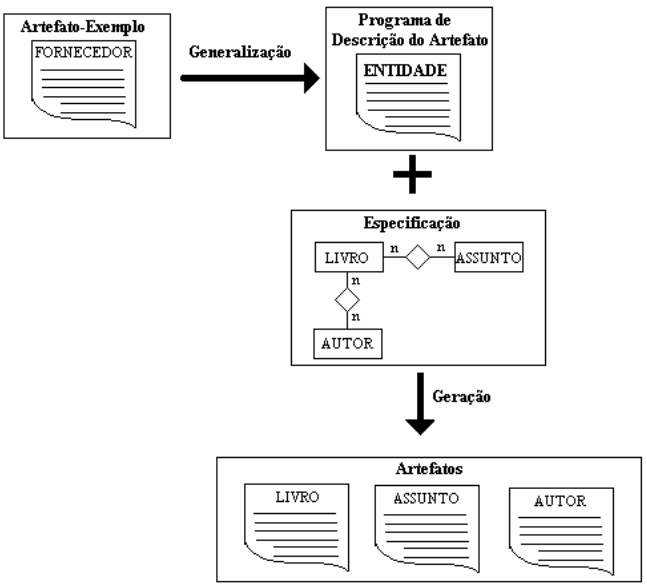


Figura 4.1 Visão esquemática da generalização/geração

No processo proposto, o gerador de artefatos não é construído através de um processo de desenvolvimento tradicional, e sim, é utilizando um processo evolutivo, através de ciclos sucessivos de criar e alterar o artefato-exemplo, identificando e inserindo *hot-spots* bem como as correspondentes transformações, produz-se versões sucessivas do gerador (Fig. 4.2). Cada ciclo produz uma versão executável do gerador. O objetivo de cada ciclo é aumentar o escopo do gerador. À medida que os ciclos vão se repetindo, as versões geradas vão ficando mais completas e mais genéricas. Cada ciclo realizado, também serve para aumentar o conhecimento do desenvolvedor em relação ao domínio do problema. Isto é conseguido através da avaliação dos artefatos gerados, quanto a sua adequação às necessidades do usuário. Este processo pode ser visto como um processo baseado em sucessivos protótipos conforme sugerido por [Boehm 1988].

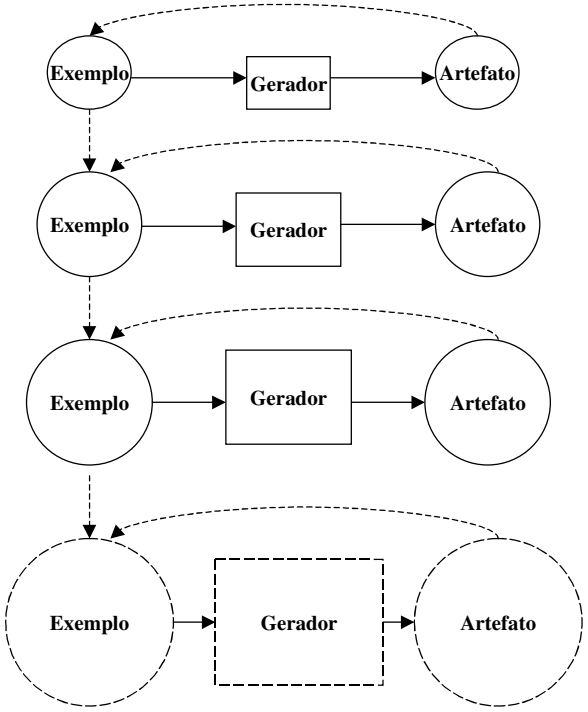


Figura 4.2 Processo evolutivo de construção

Como foi descrito no capítulo anterior, as regras de transformação são implementadas através de funções que buscam informações contidas na especificação armazenada na base de software da ferramenta CASE. Nos experimentos realizados, observamos que uma mesma regra de transformação era utilizada em diferentes arquivos de meta-descrição de artefatos, sendo que estes arquivos podiam ser de um mesmo gerador ou de geradores diferentes. Para facilitar o reúso, criamos uma biblioteca de funções que implementam o conjunto de regras de transformação. Esta biblioteca vai sendo expandida, com as novas regras identificadas e implementadas durante a realização do ciclo de evolução do gerador. Nos experimentos realizados, após a criação das funções básicas, o reúso das funções tornou-se elevado, e a biblioteca passou a crescer lentamente a cada ciclo evolutivo, eventualmente parando de crescer com relação a um determinado domínio de problema, mesmo que este seja expandido. Atualmente, a biblioteca conta com cerca de 70 regras de transformação (Apêndice C).

4.2 – Visão Geral do Processo de Construção

A Figura 4.3 apresenta uma visão esquemática do processo proposto. Para simplificar o diagrama, estão representados apenas os principais artefatos manipulados e produzidos ao longo do processo. A notação utilizada foi extraída de uma proposta de um modelo de processo para medição de software baseada em GQM ("Goal/Question/Metric") [GHW 1995].

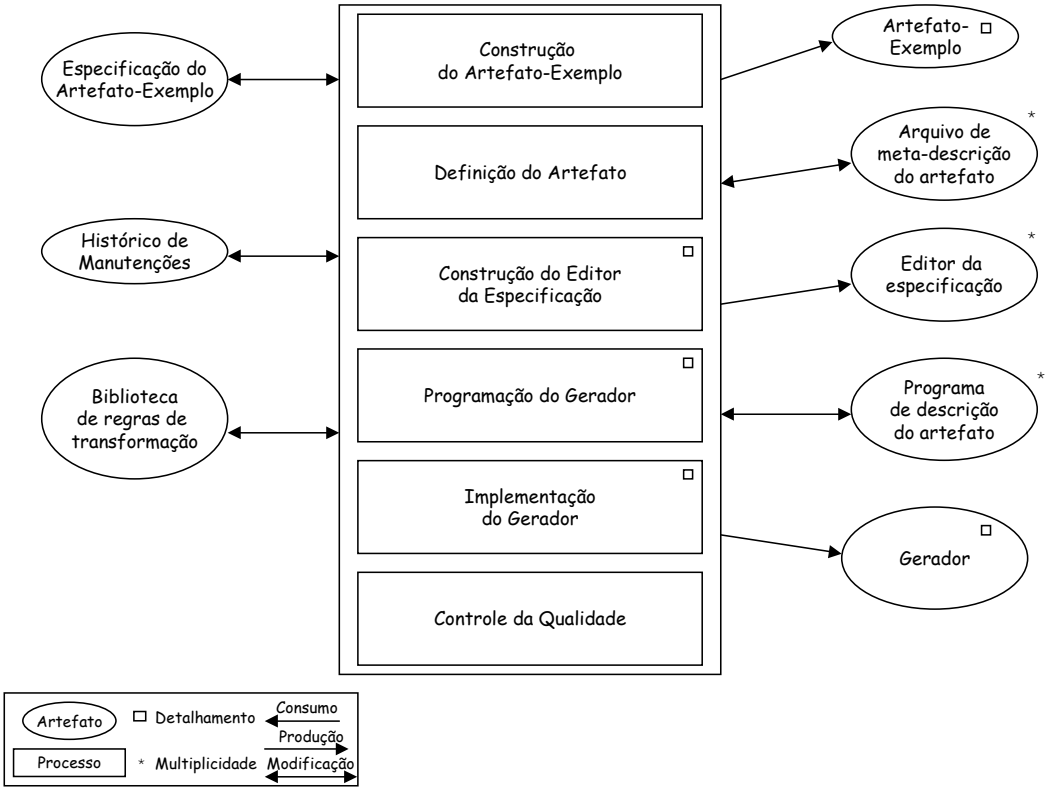


Figura 4.3 Visão resumida do processo de construção de geradores de artefatos

O ponto de partida no desenvolvimento do gerador de artefatos é a construção de um artefato-exemplo no domínio de problema-alvo. Este artefato servirá de modelo para o gerador. Durante esta etapa, devem ser registradas todas as modificações realizadas na evolução do artefato-exemplo. Por servir de protótipo, este artefato-exemplo permite que o desenvolvedor expanda seus conhecimentos no domínio do problema-alvo à medida que vai desenvolvendo e avaliando o gerador.

A partir de uma versão correta do artefato-exemplo, o desenvolvedor realiza a sua generalização, no sentido de criar os programas de descrição que

compõem o artefato final. Esta generalização envolve identificar os *frozen-spots* (parte fixa) e os *hot-spots* (parte variável) e incluir as correspondentes regras de transformação nos arquivos do artefato-exemplo.

A etapa seguinte envolve a construção do módulo de edição da especificação. Os formulários de edição das especificações terão campos referentes às informações utilizadas pelas regras de transformação contidas nos programas de descrição do artefato. Além da entrada de dados textual, a especificação pode ser informada por meio de diagramas. O módulo de edição da especificação aproveita as ferramentas disponibilizadas pela ferramenta CASE para realizar a edição de diagramas.

No processo de programação do gerador são codificados quatro tipos de programas: descrição do artefato, regras de transformação, coordenação da geração e verificação da especificação.

Os programas de descrição do artefato são gerados automaticamente a partir dos arquivos de meta-descrição criados na etapa de definição do artefato. Se necessário, novas regras são programadas e incluídas na biblioteca de regras. O programa de coordenação da geração é codificado com objetivo de estabelecer a seqüência correta da geração dos diferentes arquivos que compõem o gerador. Já o programa de verificação tem como objetivo verificar se todas as informações necessárias à geração foram fornecidas.

No processo de montagem, o gerador é composto a partir dos programas de descrição do artefato, da biblioteca das regras de transformação, do programa de coordenação da geração e o programa de verificação da especificação. Após a sua composição, o gerador deve ser compilado pela ferramenta CASE, habilitando-o, assim, a gerar artefatos.

Na etapa final, o gerador passa por diversos testes, para verificar se os artefatos gerados estão de acordo com sua especificação e para detectar possíveis erros presentes no gerador. Estes testes correspondem à execução do gerador pela equipe de desenvolvimento e posteriormente por um grupo externo de testadores. O gerador é liberado para uso generalizado somente após passar por esta seqüência de testes.

A Figura 4.4 mostra a seqüência de execução dos sub-processos de construção utilizada nos nossos experimentos (tabela 4.1). Nas próximas seções detalharemos cada um destes processos.

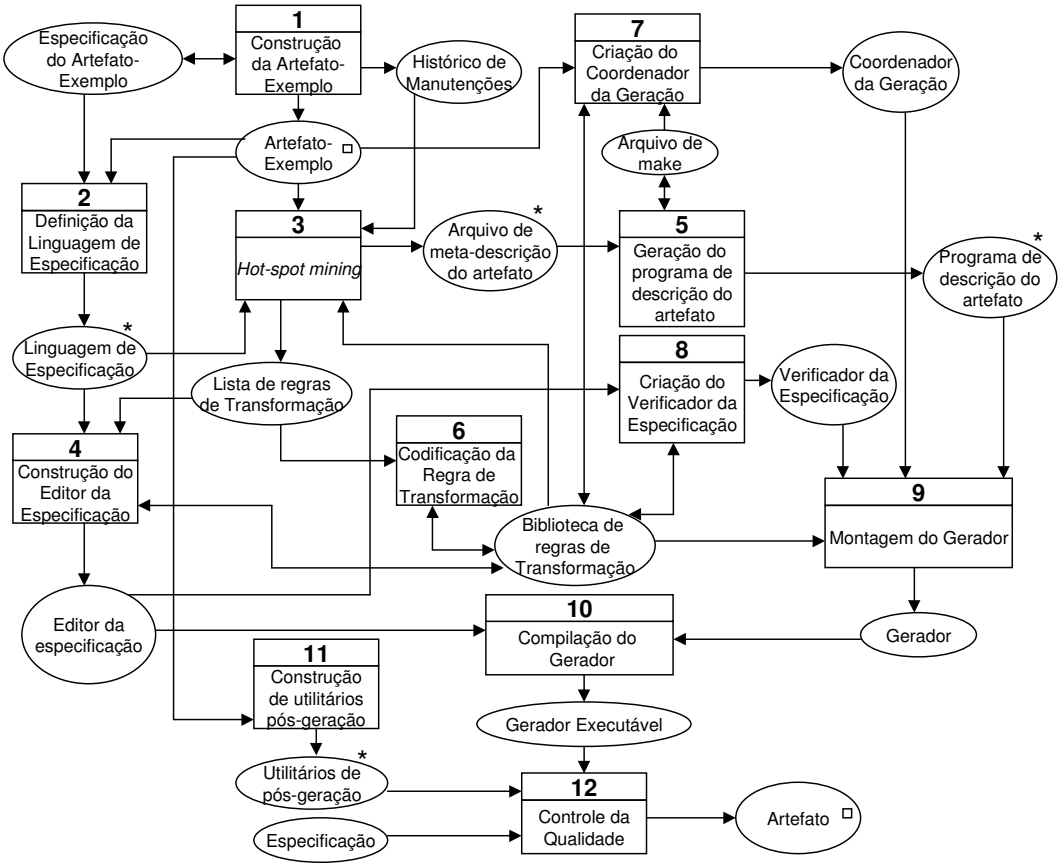


Figura 4.4 Visão detalhada do processo de construção de geradores de artefatos

• **Processo**	• **Sub-Processo**
• Construção do artefato-exemplo	• **1** Construção do artefato-exemplo
• Definição do artefato	• **2** Definição da linguagem de especificação • **3** *Hot-spot mining*
• Construção do editor da especificação	• **4** Construção do editor da especificação
• Programação do gerador	• **5** Geração do programa de descrição do artefato • **6** Codificação das regras de transformação • **7** Criação do coordenador da geração • **8** Criação do verificador da especificação
• Implementação do gerador	• **9** Montagem do gerador • **10** Compilação do gerador • **11** Construção de utilitários pós-geração
• Controle da qualidade	• **12** Controle da Qualidade

Tabela 4.1 Seqüência de execução do processo de construção

4.3 – Detalhamento do Processo de Construção

4.3.1 – Construção do Artefato-Exemplo

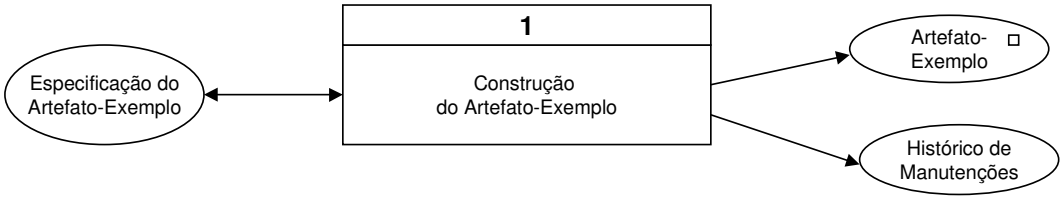


Figura 4.5 Construção do artefato-exemplo

Nesta etapa do processo, é construído o artefato-exemplo que servirá de modelo para o gerador de artefatos do domínio de problema-alvo. O artefato-exemplo pode ser composto por um ou mais arquivos, que podem estar codificados em diferentes linguagens. Este processo é concluído quando o artefato-exemplo alcança uma versão correta e estável. Para desenvolver um artefato-exemplo de elevada qualidade, são utilizadas as técnicas convencionais da engenharia de software. O processo utilizado está, portanto, fora do contexto do processo de desenvolvimento do gerador. É importante enfatizar que é muito importante que o artefato-exemplo possua elevada qualidade, uma vez que o gerador replicará a sua implementação. Conseqüentemente, faltas e deficiências porventura existentes, serão replicadas nos artefatos gerados. Portanto, o processo de software utilizado para construir o artefato-exemplo deve possuir mecanismos rigorosos de controle de qualidade. Por exemplo, nos experimentos realizados, a adoção de um padrão de nomeação de variáveis facilitou a inspeção dos arquivos do artefato-exemplo (Apêndice E). O artefato-exemplo construído é composto pelos seus arquivos-fonte e pela sua documentação

Todas as alterações na especificação do artefato-exemplo que ocorreram durante a sua construção e os seus respectivos impactos na implementação devem ser registrados. Este histórico de manutenção servirá de guia para as generalizações.

Se o gerador a ser construído tem por finalidade gerar artefatos baseados num artefato já desenvolvido (ex.: sistema legado), o objetivo deste processo passa a ser a recuperação da documentação existente do artefato.

É importante ressaltar que o domínio de problema do artefato escolhido para ser modelo, deve possuir características que favoreçam a geração, como por exemplo:

- **Abstrações comuns**. O artefato-exemplo deve compartilhar abstrações comuns com outros artefatos do mesmo domínio de problema. Em [RJ 1996], a construção de três exemplos de aplicações de um mesmo domínio é apresentada como sendo a quantidade mínima de experimentos necessária para tornar possível a identificação de abstrações comuns. O desenvolvedor deve procurar por padrões de repetição e similaridades no *design* e na arquitetura dos artefatos.
- **Rastreabilidade da especificação.** Deve ser fácil verificar a correspondência da especificação com a implementação do artefato-exemplo. Caso não o seja, além de ser um indicador da possibilidade de uma implementação com problemas, é também um indicador de uma organização deficiente do artefato-exemplo. Ambos os casos sugerem que se reveja tanto o projeto com a sua implementação.

4.3.2 – Definição do artefato

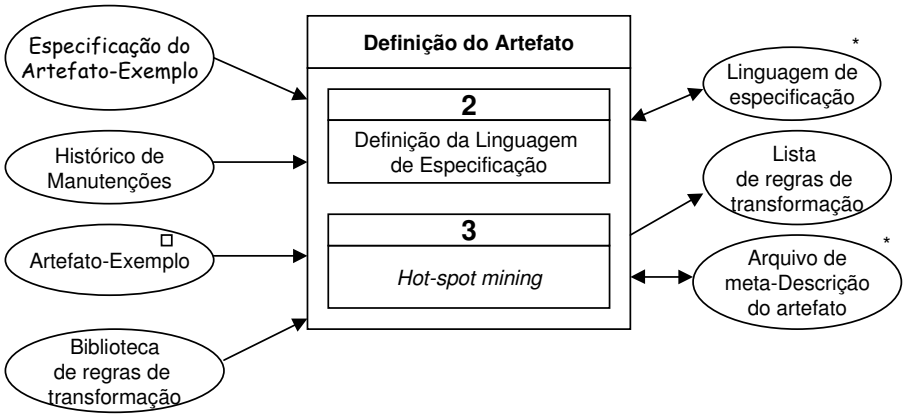


Figura 4.6 Definição do artefato

Nesta etapa do processo, os arquivos que compõem o artefato-exemplo são generalizados. O processo de descrição do artefato subdivide-se nos processos: definição da linguagem de especificação, e *hot-spot mining.*

4.3.2.1 – Definição da Linguagem de Especificação

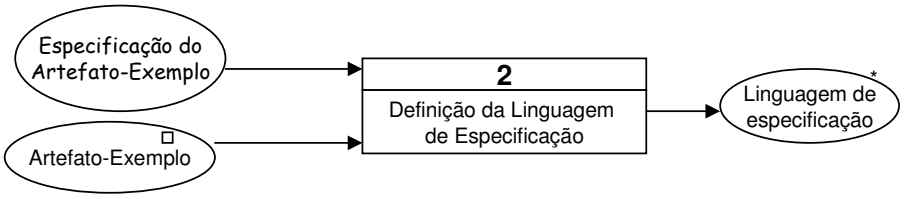


Figura 4.7 Definição da linguagem de especificação

Nesta etapa do processo, são definidas as linguagens utilizadas para especificar o artefato a ser gerado. A definição da linguagem de especificação é importante para as etapas seguintes do processo de construção do gerador. A linguagem de especificação serve para determinar os meta-dados (ex.: No caso do diagrama E-R: entidade, relacionamento, atributo, etc.) que serão utilizados nas regras de transformação e para definir as características do programa de edição da especificação. A linguagem de especificação está associada ao modelo no qual o artefato a ser gerado está baseado (ex.: modelo de dados, modelo orientado a objetos, máquina de estados e transições, etc.).

Dependendo do gerador, pode ser necessária a utilização de mais de uma linguagem de especificação. Nos nossos experimentos com prototipadores de modelo de dados (seção 6.1), utilizamos duas linguagens de especificação: diagrama de entidade-relacionamento para especificação da base de dados da aplicação e o diagrama hierárquico para definição da estrutura do menu da aplicação.

A análise da especificação e da implementação do artefato-exemplo pode apontar para a utilização de uma linguagem de especificação diferente da utilizada na especificação do artefato-exemplo. Por exemplo, para construir o arterfato-exemplo foi utilizado um diagrama de Entidade-Relacionamento, já o gerador utiliza o diagrama de Classes como linguagem de especificação do artefato-alvo.

4.3.2.2 –*Hot-spot Mining*

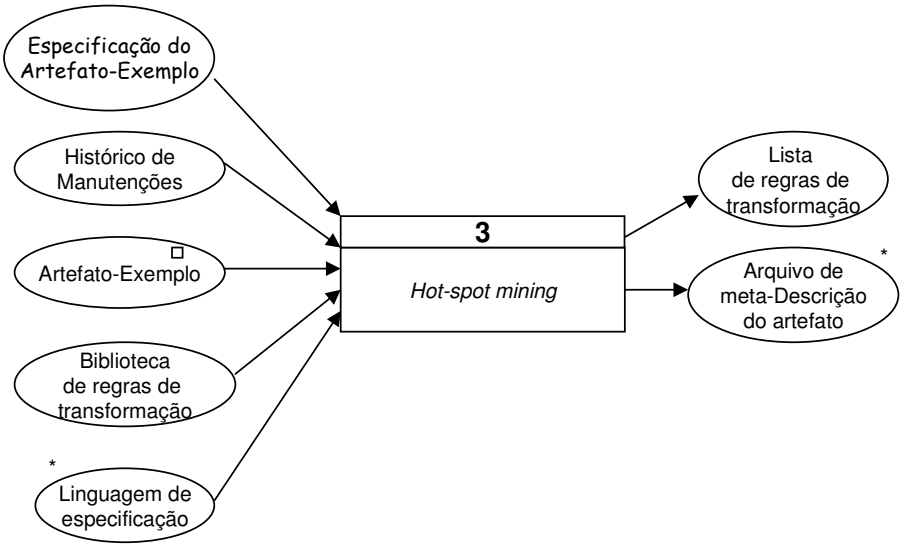


Figura 4.8 *Hot-spot Mining*

Nesta etapa do processo, são identificados os *hot-spot*s (partes variáveis) e os *frozen-spots* (partes fixas) do artefato-exemplo segundo os princípios de *hot-spot mining* descritos na seção 3.3. Esta identificação é realizada para cada arquivo que compõe o código do artefato-exemplo, por meio da inspeção do arquivo, da verificação das relações entre especificação e implementação (rastreabilidade) bem como da análise do histórico de modificações (ex.: impactos de uma determinada alteração). Durante esta identificação, cada *hot-spot* encontrado é marcado por um *tag* de transformação (seção 3.4). Estas marcações irão transformar os arquivos do artefato-exemplo no seu arquivo de meta-descrição. Ao final deste processo, cada arquivo do artefato-exemplo terá dado origem ao seu respectivo arquivo de meta-descrição. A ferramenta básica utilizada neste processo é um editor de texto.

Cada *tag* utilizado para marcar um *hot-spot* está associado a uma ou mais regras de transformação. As regras de transformação manipulam os meta-dados utilizados pelas linguagens de especificação definidas na etapa anterior. Durante a inclusão de marcadores, o desenvolvedor deve procurar reutilizar as regras já implementadas, consultando a biblioteca de regras de transformação. As regras existentes e as novas requeridas devem ser registradas numa lista,

assinalando, além das propriedades a serem satisfeitas pela regra, se é uma regra nova ou não. Esta lista será insumo para os processos seguintes de codificação das regras de transformação e da definição do editor da especificação.

A lista de regras de transformação contém a especificação de cada uma das regras de transformação inserida no arquivo de meta-descrição de produto. Esta especificação define o protótipo da chamada da função que implementa a regra, os meta-dados sobre os quais a regra de transformação vai ser aplicada e as pré e pós-condições.

Durante a criação do arquivo de meta-descrição, pode ocorrer a necessidade do desenvolvedor solicitar a reestruturação do arquivo do artefato-exemplo, provocando assim um retorno para o processo inicial. A reestruturação tem como objetivo facilitar a identificação dos *hot-spots* e a aplicação da regra de transformação. Como exemplos de sinais da necessidade de reestruturação podemos citar:

- **Regras de transformação complexas.** Nos experimentos realizados, as regras em sua maioria eram bastante simples, correspondendo, em geral, à busca de informações simples no repositório de dados. Quando havia necessidade de utilizar uma regra mais complexa, sempre avaliávamos a possibilidade de modificarmos o arquivo da aplicação simplificando as transformações, possivelmente através da adição de mais *hot-spots*.

- **Dificuldade de visualizar a correspondência entre especificação e implementação.** Durante a identificação de partes variáveis do artefato-exemplo, o desenvolvedor procura localizar as conexões entre a implementação e a especificação do artefato-exemplo. Quando o arquivo do artefato-exemplo apresenta algum tipo de padronização, como por exemplo, os nomes das variáveis estão relacionados com elementos da especificação do artefato, a identificação dos *hot-spots* é bastante facilitada.

4.3.3 – Construção do editor da especificação

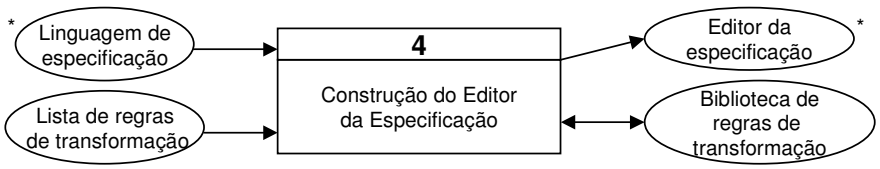


Figura 4.9 Construção do editor da especificação

Nesta etapa do processo, é definido o programa de edição da especificação do gerador. Através deste módulo a especificação do artefato a ser gerado é registrada no repositório de dados. Este módulo corresponde ao conjunto de formulários de edição do dicionário de dados e dos editores de diagramas utilizados para capturar e manter a especificação.

Os formulários do dicionário de dados são definidos por meio do programa de edição da especificação (Figura 3.5). Este arquivo é escrito utilizando a linguagem disponibilizada pela ferramenta CASE. Esta facilidade de programação permite que o formulário do dicionário seja completamente customizado.

Para definir o conteúdo de um formulário do dicionário de dados, o desenvolvedor deve analisar as linguagens de especificação definidas para o gerador e a lista de regras de transformação utilizadas na construção do gerador, identificando relações e atributos necessários para realizar cada uma das transformações. Estas relações e atributos são obtidos através de campos de formulários. Os campos dos formulários de edição do dicionário de dados referem-se aos meta-dados da linguagem de especificação do gerador (seção 4.3.2.1). Cada linguagem de especificação tem seu conjunto de meta-dados (ex.: linguagem de fluxo de dados: processo, fluxo de dados, depósito de dados). Caso o gerador utilize mais de uma linguagem de especificação, cada linguagem terá o seu programa de edição da especificação.

Em relação aos diagramas, uma ferramenta CASE disponibiliza editores de diagramas para uma variedade de linguagens de especificação suportadas pela ferramenta (ex.: entidade e relacionamento, fluxo de dados, etc.), tornando desnecessária a codificação de um editor de diagramas para o gerador que está sendo construído.

4.3.4 – Programação do Gerador

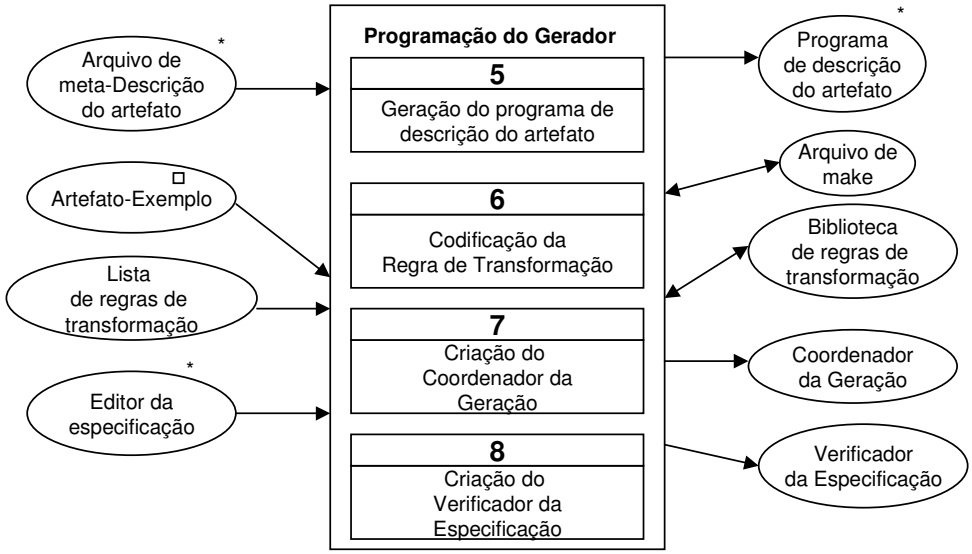


Figura 4.10 Programação do Gerador

Esta etapa do processo concentra todas as atividades relacionadas à programação do componente de geração do gerador. Os programas são codificados na linguagem de programação disponibilizada pela ferramenta CASE. Ao final desta etapa, os programas de descrição do artefato terão sido gerados automaticamente, a biblioteca de regras de transformação estará atualizada com a inclusão de novas regras e as funções que fazem a coordenação da geração e da verificação da especificação terão sido codificadas. Este processo divide-se em quatro sub-processos: *geração do programa de descrição do artefato, codificação das regras de transformação*, *criação do coordenador de geração* e *criação do verificador da especificação*.

No sentido de aumentar a qualidade da codificação é importante à adoção de padrões de programação [Staa 2000]. Por exemplo, nos experimentos realizados, a adoção de um padrão de nomeação de variáveis facilitou a inspeção dos programas do gerador (Apêndice E).

4.3.4.1 – Geração do programa de geração do artefato

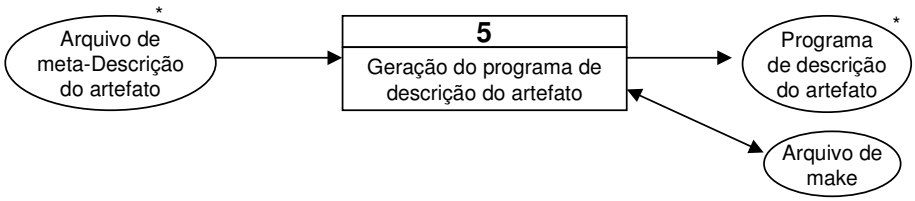


Figura 4.11 Geração do programa de geração do artefato

Nesta etapa do processo, através da utilização do utilitário GERDESCR (seção 3.4), os arquivos de meta-descrição do artefato são transformados automaticamente nos programas de descrição do artefato correspondente.

O utilitário GERDESCR é um programa escrito em Java com interface em linha de comando. Para executar a transformação, GERDESCR precisa receber 5 parâmetros (Figura 4.12). A linha de comando completa do GERDESCR para cada arquivo de meta-descrição é incluída no arquivo de *make.* O arquivo de *make* é, na realidade, um arquivo de lote (`.BAT` no ambiente DOS). Cada arquivo de meta-descrição do artefato tem uma linha correspondente no arquivo de *make* (Fig. 4.13).

```
GERDESCR BD.MJAVA BD.TXT "GERA BD" "NomeBD.java" "###"
```

• **Parâmetro**	• **Exemplo**	• **Decrição**
• 1º	`BD.MJAVA`	• nome do arquivo de meta-descrição do artefato
• 2º	`BD.TXT`	• nome do programa de descrição do artefato
• 3º	`"GERA BD"`	• nome da função que vai gerar o programa de descrição do artefato
• 4º	`"NomeBD.java"`	• nome do arquivo do artefato a ser gerado. A string Nome é uma palavra-reservada do GERDESCR que sinaliza um substituição pelo nome da instância do meta-dado que está sendo gerado (ex.: Entidade Produto).
• 5º	`"###"`	• marcador de separação do arquivo gerado

Figura 4.12 Linha de comando do utilitário GERDESCR

```
GERDESCR INDEX.MHTML      INDEX.TXT  "GERA INDEX"     "C:\TEMP\DOC\INDEX.HTML" "###"
GERDESCR ENTIDADE.MHTML   ENT.TXT    "GERA ENTIDADE"  "C:\TEMP\DOC\ENT_Nome.HTML" "###"
GERDESCR RELAC.MHTML      REL.TXT    "GERA RELAC"     "C:\TEMP\DOC\REL_Nome.HTML" "###"
GERDESCR ATRIBUTO.MHTML   ATRIB.TXT  "GERA ATRIBUTO"  "C:\TEMP\DOC\ATRIB_Nome.HTML" "###"
```

Figura 4.13 Arquivo de *make*

4.3.4.2 – Codificação das Regras de Transformação

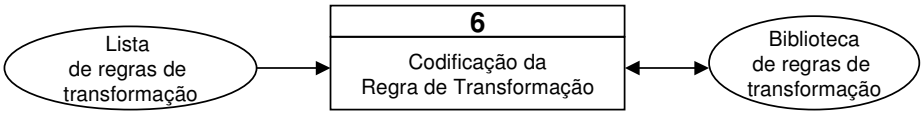


Figura 4.14 Codificação da Regra de Transformação

Nesta etapa do processo, as regras de transformação utilizadas são implementadas através de funções que acessam o repositório da ferramenta CASE.

Nos geradores que construímos, a complexidade das funções que implementavam as regras de transformação era bem baixa (Apêndice C). Geralmente as funções retornavam valores dos atributos dos objetos do repositório de dados da ferramenta (Fig. 4.15). Estes valores antes de serem retornados passavam por algum tipo de formatação (ex.: concatenação de prefixos ou sufixos) .

Função que retorna o nome de uma instância do meta-dado

```
InicFrm "fNomeClasse"( Objeto oObj)
    ComObjeto oObj Faz
        Nome
    Fim;
FimFrm
```

Figura 4.15 Exemplo de regra de transformação

Em relação à manipulação de meta-dados, é possível escrever funções que "navegam" pelo repositório da ferramenta através das relações entre esses meta-dados. Esta navegação é feita utilizando as relações definidas nos dicionários de dados ou através das relações definidas nos diagramas. Por exemplo, no meta-CASE Talisman dada uma entidade, a determinação das entidades com que ela se relaciona, pode ser feita através da relação gráfica *LigInstância* que é o conjunto de ligações que atingem determinada instância contida em um diagrama. Talisman atualiza esta relação à medida que se edita o diagrama.

Durante a codificação de uma nova função de regra de transformação, deve-se procurar utilizar as funções já existentes. A organização da biblioteca das regras de transformação deve facilitar a busca das funções reutilizáveis [Prieto-Diaz 1991]. Caso o desenvolvedor utilize editores de texto comuns para codificação das regras de transformação, a organização da biblioteca deverá

ser feita por uma ferramenta à parte. Caso o desenvolvedor utilize editores especializados, a organização da biblioteca pode ser garantida pelo próprio ambiente de edição. Por exemplo, o meta-CASE Talisman disponibiliza um editor de estruturas para codificação de programas escritos na sua linguagem nativa. Este editor automaticamente povoa a base de dados do Talisman com as informações do programa que está sendo editado, gerando a sua documentação e disponibiliza mecanismos de composição e de busca do código editado.

Dependendo da ferramenta CASE, a linguagem de programação utilizada pode ter recursos adicionais que facilitam a codificação das regras de transformação. Por exemplo, a linguagem nativa do meta-CASE Talisman oferece diferentes comandos para o processamento de listas. Muitas vezes, a implementação de uma função de transformação é simplificada através da utilização de listas auxiliares. Por exemplo, suponha que a aplicação de uma regra de transformação varie conforme o tipo do atributo da entidade (Fig. 4.16). Uma possível implementação desta regra seria utilizar a relação *Dados* que contém todos os atributos da entidade e através de comandos condicionais executar a ação específica da regra relacionada ao tipo de atributo. Um outra alternativa de implementação seria a criação de listas auxiliares para cada tipo de atributo. Estas listas seriam montadas a partir da relação *Dados.* Neste caso, o desenvolvedor poderia desmembrar a regra original, em regras menores que manipulassem cada lista específica de tipo .

Regra	***Regra decomposta***
InicFrm "Criticar dados " ParaTodos **Dados** Faz Se [Aliás Tipo] == "String" Entao ... Fim; Se [Aliás Tipo] == "Numerico" Entao ... Fim; Se [Aliás Tipo] == "Texto" Entao ... Fim; Fim; **FimFrm**	**InicFrm "Criticar dados String"** ParaTodos **auxListaDadosString** Faz Fim; **FimFrm** **InicFrm "Criticar dados Numerico"** ParaTodos **auxListaDadosNumerico** Faz Fim; **FimFrm** **InicFrm "Criticar dados Texto"** ParaTodos **auxListaDadosTexto** Faz Fim; **FimFrm**

Figura 4.16 Exemplo de decomposição da regra de transformação

4.3.4.3 – Criação do Coordenador da Geração

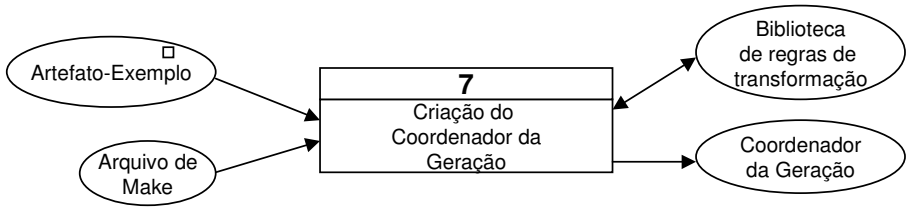


Figura 4.17 Criação do coordenador da geração

O objetivo desta etapa do processo é a implementação da função responsável pela coordenação da geração de um artefato. Esta função garante a seqüência correta da geração dos diferentes arquivos que compõem o gerador. A função de coordenação (Figura 4.18) tem a seguinte estrutura padrão:

- **Iteração sobre uma lista.** Dentro de uma repetição de leitura de uma lista, são feitas chamadas às funções de geração de cada arquivo do artefato, cada iteração vai gerar o conjunto de arquivos do artefato referentes ao elemento corrente da lista. Os arquivos a serem gerados estão definidos no arquivo de *make* (seção 4.3.2.2). A lista que comanda a "repetição" é escolhida a partir da análise da estrutura do artefato. Por exemplo, na geração do arquivo de criação das tabelas de um banco de dados (arquivo de *script* de criação), uma navegação adequada seria: para cada *Entidade* da especificação execute a função *CriarTabela*, na qual a função *CriarTabela* está definida através de um programa de descrição do artefato.
- **Preparação de lista auxiliares.** No caso das regras de transformação utilizarem listas auxiliares (seção 4.3.4.1), elas devem ser preparadas antes da chamada das funções de geração.

```
InicExterno "Gera APLICACAO"
  ParaTodos Dicionario DicEntidade Faz
   Frm "fSetListaTipoDado"(Corrente); /* montagem de lista aux */

   Frm "GERA CLASSE HTML"(Corrente); /* geração arquivo artefato*/
   Frm "GERA CLASSE SCRIPT"(Corrente);/* geração arquivo artefato*/
  Fim;
FimFrm
```

Figura 4.18 Exemplo de uma Função de Coordenação de Geração

4.3.4.4 – Criação do verificador da especificação

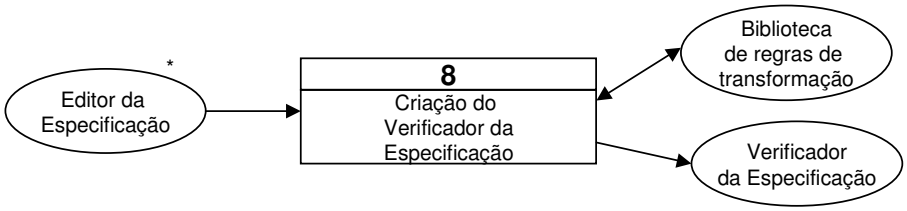


Figura 4.19 Criação do verificador da especificação

O objetivo desta etapa do processo é a implementação da função responsável pela verificação da especificação. Esta função verifica se todas as informações necessárias para a geração estão disponíveis na especificação. O processo de geração só deve prosseguir, quando a função de verificação não encontra mais erros na especificação. Por exemplo, no caso de um gerador baseado em modelo de dados, a função de verificação poderia realizar as seguintes verificações:

- Todas as entidades têm pelo menos um atributo;
- Todos os atributos têm tipo
- Pelo menos um atributo é obrigatório e identifica a entidade

No caso do meta-CASE Talisman, o programa de verificação pode ser aplicado em diferentes níveis: meta-dado, diagrama, dicionário de dados, e especificação completa. Cada nível tem um programa de verificação específico. Por exemplo, cada meta-dado pode ter um programa de verificação (ex.: verificador de uma entidade, verificador de um atributo). No nosso trabalho utilizamos a verificação no nível de especificação completa, no qual o programa de verificação examina de uma vez os meta-dados e diagramas utilizados na especificação.

4.3.5 – Implementação do Gerador

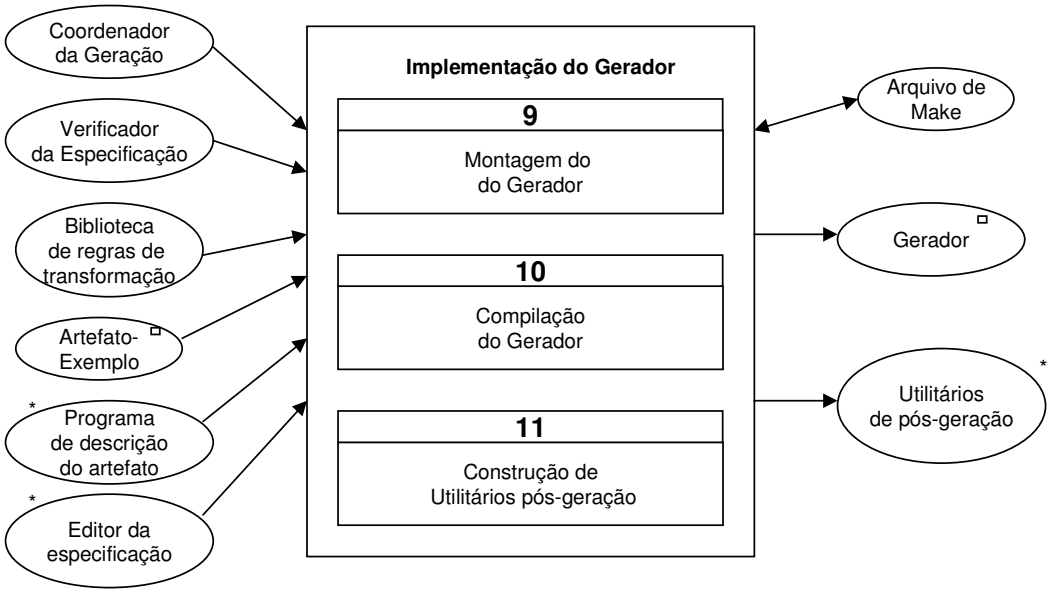


Figura 4.20 Implementação do gerador

O objetivo desta etapa do processo é tornar operacional o gerador. Primeiro é necessário montar o gerador completo para permitir a sua compilação pela ferramenta CASE. Após a compilação, o gerador estará pronto para ser executado. Dependendo da aplicação gerada, pode ser necessária a utilização de utilitários que complementam a sua implementação.

4.3.5.1 – Montagem do Gerador

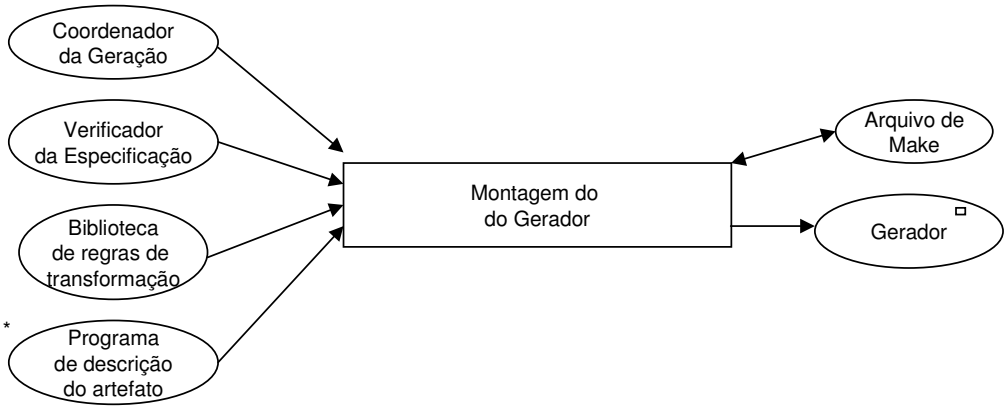


Figura 4.21 Montagem do Gerador

O objetivo deste passo é a composição do gerador formado pelos programas de descrição do artefato, programas de coordenação de geração e verificação da especificação e pela biblioteca de regras de transformação. A forma de montagem depende dos recursos da ferramenta CASE utilizada como ambiente de geração. Por exemplo, como o meta-CASE Talisman não possui mecanismos de ligação com biblioteca externas, a montagem do gerador é feita pela concatenação dos seus arquivos componentes. O arquivo montado corresponde a um único grande arquivo texto a ser compilado pelo Talisman. O arquivo de *make* criado na etapa de geração do programa de descrição do artefato é expandido com a inclusão das linhas referentes a concatenação dos arquivos (Fig 4.22).

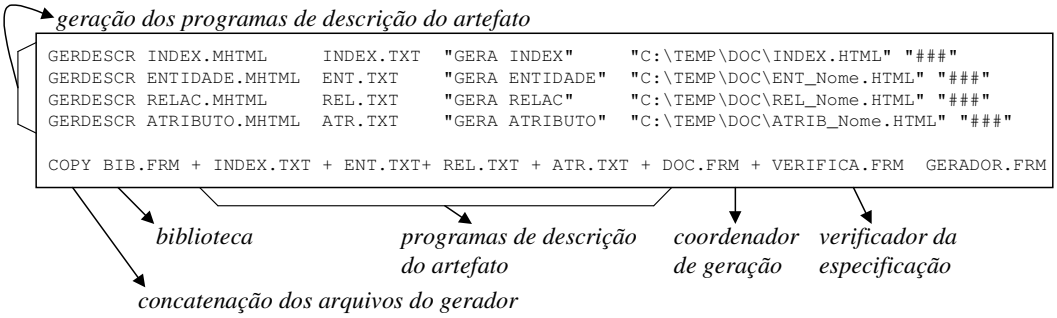


Figura 4.22 Arquivo de *make* completo

4.3.5.2 – Compilação do Gerador

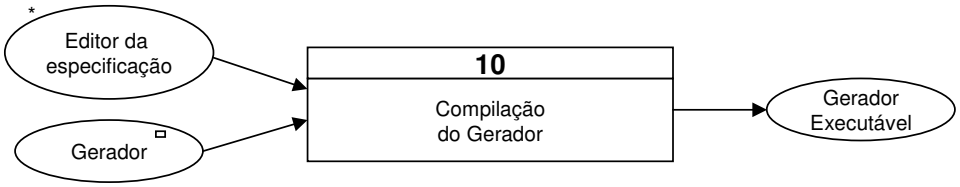


Figura 4.23 Processo de Compilação do Gerador

O objetivo deste passo é produzir a forma executável do gerador. Tanto os programas de edição da especificação como o programa completo do gerador devem ser compilados pela ferramenta CASE. Cada ferramenta CASE tem um modo de integrar o executável do gerador com os demais módulos da ferramenta. Por exemplo, o compilador do meta-CASE Talisman compila os programas do gerador e armazena o executável (*byte code*) do módulo do editor da especificação e do gerador de artefato na base de conhecimento da ferramenta. Após a compilação, tanto as novas telas de entrada de dados como o novo módulo de geração ficam disponíveis para serem instanciados.

4.3.5.3 – Construção de utilitários pós-geração

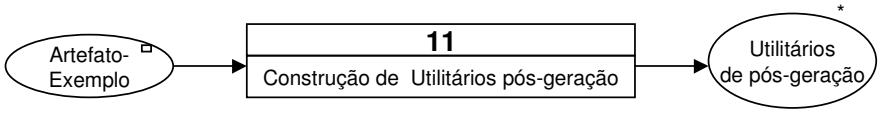


Figura 4.24 Processo de Construção de Utilitários pós-geração

O objetivo desta etapa do processo é a construção de utilitários que preparam a implementação do artefato gerado. Caso o artefato seja uma aplicação ou um programa, antes de compilar os arquivos componentes do artefato pode ser necessário preparar estes arquivos para compilação. Em nossos experimentos com o gerador de artefatos em Java foi necessário criar um utilitário que criasse o arquivo de projeto e copiasse os arquivos gerados para a estrutura de diretórios definida por este arquivo de projeto. Floch em [Floch 1995] trabalhando com geradores na área de telefonia relata a necessidade de criar uma linguagem para definição de módulos para facilitar a composição da aplicação gerada.

Dependendo do formato dos arquivos gerados pode ser necessário a criação de utilitários que modifiquem os arquivos para um formato desejado. Por exemplo, o resultado da geração através do meta-CASE Talisman é um único arquivo contendo todos os arquivos do artefato separados por um marcador (parâmetro do GERDESCR). Este arquivo contém os *tags* de transformação na forma de comentários. Estes *tags* são extraídos do arquivo através do utilitário *Extrator*. Após a extração, os arquivos do artefato são separados do arquivo gerado através do utilitário *Separador* (Fig. 4.25).

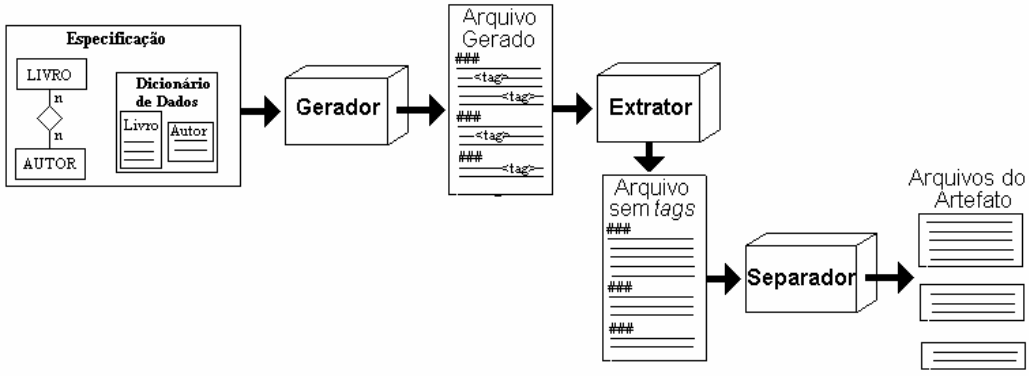


Figura 4.25 Exemplos de Utilitários pós-geração

4.3.6 – Controle da qualidade

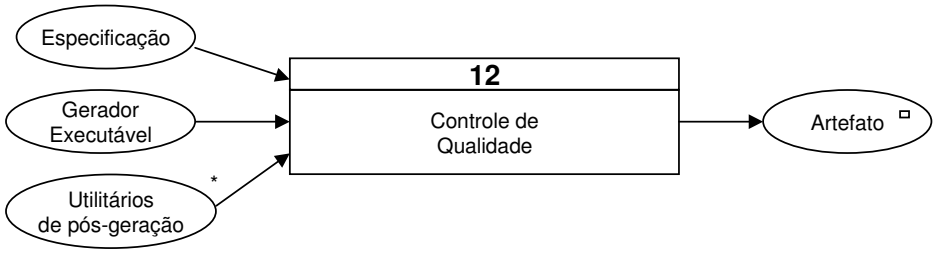


Figura 4.26 Controle da qualidade

Nesta etapa, o gerador passa por diversos testes, para verificar se o artefato gerado está de acordo com sua especificação e para detectar possíveis erros do gerador. Estes testes correspondem à execução do gerador pela equipe de desenvolvimento, e posteriormente por um grupo externo de testadores. O gerador é liberado para uso generalizado somente após passar por esta seqüência de testes.

O primeiro teste a ser executado é a geração do artefato-exemplo que serviu de modelo para a construção do gerador. Os arquivos do artefato gerado são comparados com os arquivos do artefato-exemplo através de um utilitário do tipo "*file compare"* (ex.: FC do ambiente DOS). A existência de diferenças entre esses arquivos é um sinal de problemas no processo de construção.

Após o teste de comparação de arquivo, a equipe de desenvolvimento deve verificar a instanciação de cada *hot-spot* do gerador. Para cada *hot-spot* é criado um caso de teste. O caso de teste consiste em criar uma especificação contendo as informações necessárias para a aplicação da regra de transformação associada ao *hot-spot*. Os casos de testes podem ser criados a partir da análise das telas do módulo de edição da especificação do gerador. A título de exemplo, a tabela 4.2 apresenta uma lista parcial dos casos de teste criados para verificar a qualidade do gerador de aplicações de manutenção/consulta das tabelas de um de modelo de dados criado em nossos experimentos.

Casos de teste montados a partir do dicionário de dados	
1º	Entidade com título a ser exibido preenchido ex.: Entidade: METRICA => tíitulo a ser apresentado nas telas: Métrica
2º	Entidade sem título a ser exibido preenchido
3º	Entidade com 1 campo obrigatório
4º	Entidade com mais de 1 campo obrigatório
5º	Entidade com 1 campo do tipo caracter
6º	Entidade com mais de 1 campo do tipo caracter
7º	Entidade com 1 campo do tipo numérico
8º	Entidade com mais de 1 campo do tipo numérico
9º	Entidade com 1 campo do tipo data
10º	Entidade com mais de 1 campo do tipo data
11º	Entidade com 1 campo do tipo hora
12º	Entidade com mais de 1 campo do tipo hora
13º	Entidade com 1 campo do tipo texto
14º	Entidade com mais de 1 campo do tipo texto
15º	Entidade com campos de diferentes tipos
16º	Entidade com atributo com título a ser exibido preenchido ex.:Atributo: ENDERECO=>título a ser exibido nas telas: Endereço completo
17º	Entidade com atributo sem título a ser exibido preenchido
18º	Entidade com rotina de escape para tratamento da inclusão
19º	Entidade sem rotina de escape para tratamento da inclusão
Casos de teste montados a partir dos diagramas Entidade-Relacionamento	
20º	Modelo de dados com 1 entidade
21º	Modelo de dados com mais de 1 entidade
22º	Modelo de dados com 2 entidades e 1 relacionamento 1:N
23º	Modelo de dados com 2 entidade e mais de 1 relacionamento 1:N
24º	Modelo de dados com 3 entidades 2 relacionamentos 1:N
25º	Modelo de dados com 2 entidades e 1 relacionamento N:N
26º	Modelo de dados com 2 entidade e mais de 1 relacionamento N:N
27º	Modelo de dados com 3 entidades e 2 relacionamentos N:N
28º	Modelo de dados com 1 entidade e 1 auto-relacionamento 1:N
29º	Modelo de dados com 1 entidade e mais de 1 auto-relacionamento 1:N
30º	Modelo de dados com 1 entidade e 1 auto-relacionamento N:N
31º	Modelo de dados com 1 entidade e mais de 1 auto-relacionamento N:N
32º	Modelo de dados com várias entidades e vários relacionamentos

Tabela 4.2 Casos de teste para um prototipador de modelo de dados

Após a realização dos testes de verificação da implementação dos *hot-spots*, a equipe de desenvolvimento deve realizar a geração para pelo menos 3 artefatos completos. Este teste de aceitação realizado pela equipe de desenvolvimento é chamado de alfa teste [Pfleeger 1998]. Em seguida, deve ser realizado um teste de aceitação externo denominado beta teste. Este teste é realizado por um grupo de potenciais usuários do gerador. Somente após a

realização dos testes de aceitação, o gerador deve ser liberado para uso generalizado.

Caso o processo de construção do gerador não introduza faltas no gerador, a qualidade dos artefatos gerados será equivalente à qualidade do artefato-exemplo utilizado como modelo para o gerador. Por isto, na etapa inicial de construção do artefato-exemplo deve ser realizado um rigoroso controle da qualidade deste artefato. Somente após este controle, deve-se prosseguir com o processo de construção do gerador. Como foi mencionado anteriormente, caso o artefato- exemplo possua faltas, elas serão propagadas pelo gerador e estarão presentes em todos os artefatos gerados.

O capítulo 5 é dedicado para um maior detalhamento da questão do controle da qualidade. Neste capítulo, são apresentados alguns aspectos da abordagem empírica adotada neste trabalho, e também são apresentadas algumas referências de uma abordagem mais formal.

4.4 – Processo de Manutenção do Gerador de Artefatos

A figura 4.27 apresenta a composição do gerador construído através do processo de construção proposto.

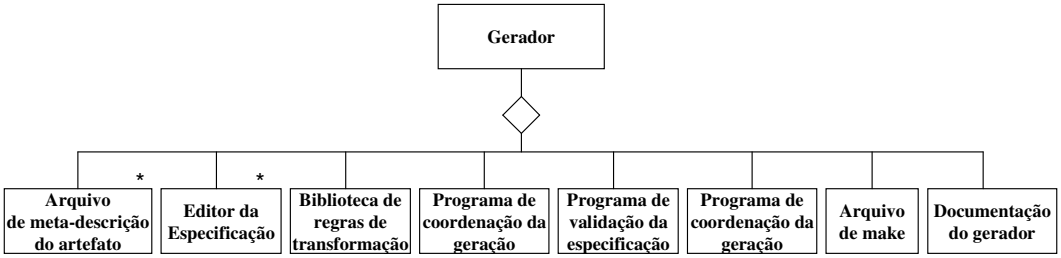


Figura 4.27 Composição do gerador

A inclusão de algumas novas funcionalidades no artefato gerado corresponde a alterações no seu gerador. Estas alterações têm impacto sobre todos os componentes do gerador. Esta evolução pode ser feita de forma organizada através da adoção do próprio processo de construção do gerador.

Na primeira etapa de construção do artefato-exemplo, as alterações são feitas num artefato gerado. A vantagem de utilizar um artefato gerado é a facilidade de mapeamento das alterações sobre os *hot-spots* marcados nos arquivos de meta-descrição do artefato.

Na etapa seguinte de definição do artefato, o desenvolvedor modifica os arquivos de meta-descrição envolvidos com as novas funcionalidades do artefato. A identificação dos *hot-spots* é guiada pelas alterações realizadas para suportar as novas funcionalidades.

Na etapa de construção do editor da especificação, os programas do editor da especificação são alterados no sentido de permitir que a especificação passe a conter as informações necessárias para implementação das novas funcionalidades.

Na etapa de programação do gerador, as novas regras de transformação são codificadas e os programas de coordenação da geração bem como de verificação da especificação são alterados para tratarem as novas funcionalidades.

Na etapa de implementação do gerador, o novo gerador é montado e recompilado.

Na etapa de controle de qualidade, são desenvolvidos casos de testes para verificar a implementação das novas funcionalidades e são adaptados os casos-teste das funcionalidades alteradas. É importante a realização de um rigoroso teste de regressão para verificar se o comportamento anterior do gerador não foi prejudicado com as alterações realizadas [GJM 1991].

O processo de evolução do gerador depende da qualidade da documentação existente sobre o gerador. Muitos trabalhos na área de *framework* [Johnson 1992, SLB 1998] apontam que as dificuldades no aprendizado de um *framework* estão associadas à baixa qualidade de sua documentação. A documentação do gerador pode ser dividida em: documentação do artefato gerado, documentação dos *hot-spots* e documentação do mapeamento dos hot-spots sobre o artefato gerado. Em [Guerriei 1998, DCGMM 1999] são apresentadas propostas de documentação através de hipertexto que podem ser aplicadas para documentação de um gerador. No Apêndice D, é mostrado um exemplo da documentação utilizada para os *hot-spots*. A documentação de hot-spots está baseada na proposta de [FHLS 1997, FHLS 1999].

4.5 – Resumo

Neste capítulo foi apresentado um processo de construção de geradores baseado em uma ferramenta CASE. O processo proposto é caracterizado pela sua abordagem *bottom-up*, pelo desenvolvimento evolucionário e pelo reúso das regras de transformação. O processo está dividido em seis sub-processos:

- **Construção da artefato-exemplo:** É desenvolvido um artefato-exemplo que vai servir de modelo para geração. Este artefato é construído usando métodos e técnicas convencionais.
- **Definição do artefato:** São definidas as linguagens de especificação do artefato-alvo. É feita a generalização do artefato através da identificação de seus *hot-spots (hot-spot mining)*. O resultado da generalização são os arquivos de meta-descrição do artefato.

- **Construção do editor da especificação:** Os programas do editor da especificação são construídos. Estes programas definem os formulários de edição do dicionário de dados utilizadas para capturar a especificação necessária para geração do artefato.
- **Programação do gerador:** Os programas do gerador são codificados utilizando a linguagem de programação disponibilizada pela ferramenta CASE. Nesta etapa são codificados quatro tipos de programas: descrição do artefato, regras de transformação, coordenação da geração, e verificação da especificação. Os programas de descrição de artefato são gerados automaticamente a partir dos arquivos de meta-descrição resultantes da generalização do artefato-exemplo. Os demais tipos de programas são codificados manualmente.
- **Implementação do Gerador:** Os arquivos que compõem o gerador são agrupados num único arquivo. Este arquivo, juntamente com os programas de edição da especificação, são compilados e os executáveis produzidos passam a fazer parte do CASE.
- **Controle da Qualidade:** O gerador somente é liberado para uso generalizado após passar por diversos testes.

A evolução do gerador deve ser resultante da adoção do próprio processo de construção do gerador. Em cada etapa do processo, os componentes do gerador são alterados para implementar a inclusão das novas funcionalidades.

Capítulo 5- Controle da Qualidade

Este capítulo apresenta uma abordagem para a prevenção de defeitos como forma de melhorar o processo proposto. São apresentadas também algumas propostas selecionadas da literatura de garantia da qualidade através da formalização do processo e do gerador.

5.1 – Introdução

Um dos objetivos almejados por um processo de software é a garantia, por construção, da qualidade dos resultados gerados. Porém, a intervenção do desenvolvedor ao longo do processo, pode introduzir faltas nos artefatos produzidos. Visto que o processo proposto de construção de geradores de artefatos depende da participação do desenvolvedor, o processo proposto por si só não poderá garantir a qualidade, em particular a corretude, do gerador construído.

Embora faltas possam ser inseridas em vários componentes do processo de desenvolvimento empregado ao desenvolver artefatos com o apoio de geradores, muitas das conseqüentes falhas serão observadas somente no artefato final gerado. Faltas podem ser sistêmicas quando forem introduzidas pelas ferramentas de geração do artefato. Faltas sistêmicas serão geradas sistematicamente sempre que ocorrerem determinadas condições de especificação do artefato-alvo.

Faltas fortuitas correspondem a erros humanos cometidos ao desenvolver a especificação. Através do uso de verificadores da especificação e de outras ferramentas, procura-se minimizar a ocorrência de faltas fortuitas que afetem o código do artefato gerado. Sobram, então, inadequações, ou seja, erros de especificação que levem à geração de um artefato diferente do esperado pelo usuário. Muitos destes erros são decorrentes da falta de entendimento da aplicação a desenvolver. O uso de um gerador permite agora a geração de uma série de protótipos que têm por objetivo entender melhor o problema a resolver. Caso não se disponha de um gerador de artefatos, a produção de

sucessivos protótipos costuma ter um custo proibitivo, levando muitas vezes à produção de artefatos inadequados ou de difícil uso.

Neste capítulo interessa-nos como prevenir a inserção de faltas sistêmicas e, caso sejam inseridas, como identificar o mais cedo possível os correspondentes problemas. O objetivo é assegurar a corretude do gerador, ou seja, desejamos assegurar que o gerador produza artefatos em exata correspondência com as suas especificações. É objetivo também satisfazer outros critérios de qualidade, tais como, adequação[2] e utilizabilidade[3].

É importante ressaltar mais uma vez que é imprescindível o artefato-exemplo possuir elevada qualidade, uma vez que o gerador replicará a sua implementação.

No sentido de assegurar a qualidade do gerador produzido, os compiladores utilizados ao longo do processo são ferramentas que ajudarão na identificação de problemas sistemáticos. Além das ferramentas, dentro do contexto de processo de software, introduzimos uma etapa de verificação da especificação fornecida (seção 4.3.4.4), e adotamos a abordagem de prevenção de problemas nos moldes do CMM [PWCC 1995, FSR 1998] que será descrita na próxima seção.

Dentro deste capítulo apresentaremos também algumas propostas de garantia da qualidade através da formalização do processo e do gerador. A formalização possibilita uma maior automação do processo de construção e permite a construção de ferramentas para realizar verificações complexas, que seriam difíceis de realizar manualmente. Como a formalização do processo não é foco do nosso trabalho, selecionamos algumas referências cujas idéias podem ser aproveitadas no processo proposto.

5.2 – Ferramentas de Identificação de Faltas

A identificação da causa de um problema em um artefato produzido por um gerador não é uma tarefa trivial, uma vez que as causas podem ser uma ou

[2] *adequação*: o conjunto de solução satisfaz as necessidades e expectativas dos diversos usuários [Staa 2000]

[3] *utilizabilidade*: o artefato possui interfaces claras, completas e corretamente estabelecidas e pode ser utilizado com segurança por pessoas possuindo o nível de formação previamente estipulado [Staa 2000]

mais das seguintes (Fig. 5.1): a especificação do artefato-exemplo, o próprio artefato-exemplo, o gerador do artefato-alvo, o ambiente de geração (CASE) e a especificação do artefato-alvo fornecida para a geração.

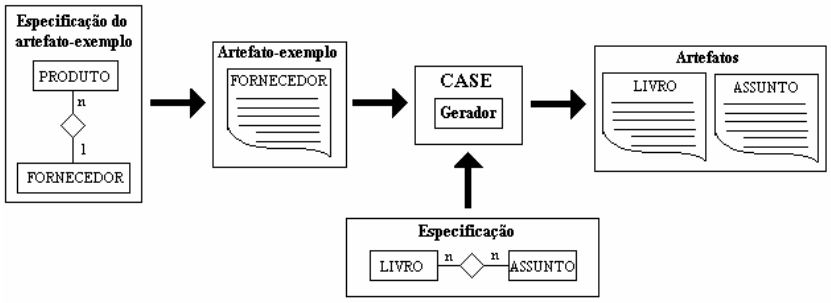


Figura 5.1 Origens das faltas de um artefato gerado

As ferramentas utilizadas ao longo do processo ajudam a antecipar a identificação de vários problemas no artefato final. São utilizadas as seguintes ferramentas: compilador da ferramenta CASE, programa de verificação de especificação, compiladores das linguagens de programação utilizadas pelo artefato, e utilitário comparador de arquivos. Além das ferramentas automatizadas, o controle da qualidade é complementado através da realização de testes e do emprego de técnicas de revisões e inspeções.

Durante as etapas de construção do gerador, o compilador da ferramenta CASE é a principal ferramenta para a identificação de erros de sintaxe nos programas que compõem o gerador escritos na linguagem nativa da ferramenta (Fig 5.2). A ferramenta CASE disponibiliza uma ambiente de execução de *scripts* que permite a criação de *drivers*[4] para testar os diversos programas que compõe o gerador. Em nossos experimentos utilizando o meta-CASE Talisman, foram criados *drivers* para testar as regras de transformação.

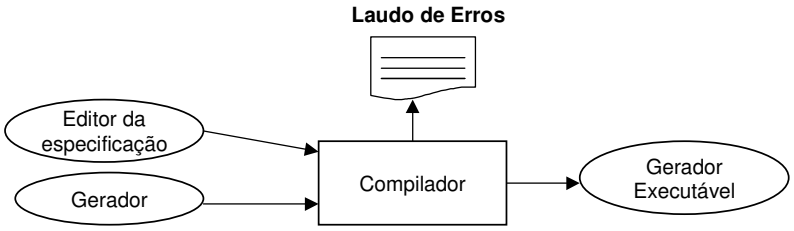


Figura 5.2 Ferramenta de identificação de faltas nas etapas de construção

[4] Programa que simula a utilização do módulo que está sendo testado [GJM 1991]

Na fase de geração do artefato, o módulo de verificação do gerador analisa a completeza da especificação fornecida quanto às informações necessárias para geração, emitindo um laudo apontando as faltas na especificação (Fig 5.3).

Após a geração, a identificação de problemas é realizada com relação aos arquivos dos artefatos gerados (Fig 5.3). Caso o artefato gerado tenha algum erro de sintaxe, o compilador da linguagem de programação (ex.: Java, VisualBasic) do artefato emite um laudo exibindo os erros de sintaxe. Caso sejam utilizados interpretadores (ex.: SQL) erros de sintaxe serão apontados somente em tempo de execução. É necessário, então, que o verificador controle os elementos da especificação que somente serão tratados nas porções de código interpretadas.

O processo de construção por ter como ponto de partida um artefato-exemplo, possibilita comparar o artefato-exemplo com um artefato gerado a partir de exatamente a mesma especificação que deu origem ao artefato-exemplo. Esta comparação pode ser feita através de um programa de comparação de arquivos (ex.: Windiff, FC do DOS, etc.). A existência de diferenças entre os arquivos do artefato gerado e os arquivos do artefato-exemplo é um indicativo de problemas no gerador.

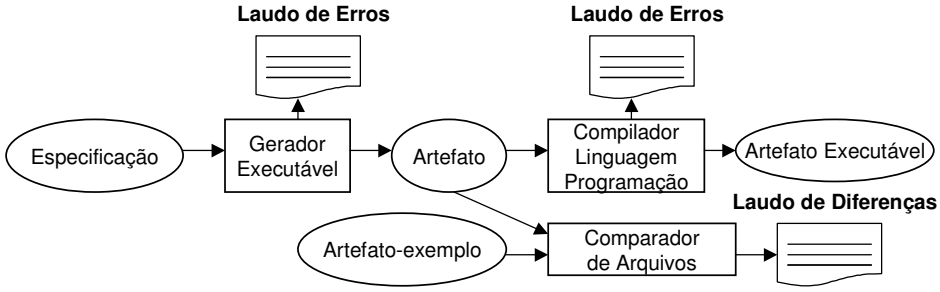


Figura 5.3 Ferramentas de identificação de faltas nas etapas de geração

5.3 – Prevenção de Problemas

A prevenção de problemas é uma *KPA* (*Key Process Area*) do nível de maturidade mais elevado do CMM (Capability Maturiy Model) da SEI (Software Engineering Institute) [PWCC 1995, FSR 1998]. O objetivo da prevenção de problemas é identificar os problemas e localizar as sua causas bem como evitar

que problemas semelhantes aos identificados ocorram novamente. A análise dos problemas encontrados fornece subsídios para melhoria do processo de software. A atividade de prevenção de problemas pode ser dividida em [Humphrey 1989]:

1. **Registro do problema:** Registrar informações suficientes para possibilitar a classificação e a determinação das causas do problema. A organização deve adotar um padrão de formulário para o registro do problema.
2. **Análise das causas:** Determinar as causas dos problemas.
3. **Ações de correção:** Estabelecer as ações necessárias para eliminar completamente as causas dos problemas.

A adoção de um processo de construção bem definido facilita a identificação da causa do problema, na medida em que a diagnose da causa passa a ficar associada à determinação da provável etapa do processo em que a falta foi introduzida.

Na figura 5.4, podemos identificar duas principais áreas de introdução de faltas no gerador que está sendo construído. A primeira delas é atividade de generalização do artefato-exemplo, caso o desenvolvedor não consiga fazer uma generalização correta, as faltas criadas durante esta atividade fazem com que o gerador construído produza sistematicamente artefatos com desvios em relação ao artefato-exemplo. A geração é a segunda área de introdução de defeitos. As faltas introduzidas nesta atividade são resultantes de erros na codificação dos programas do gerador.

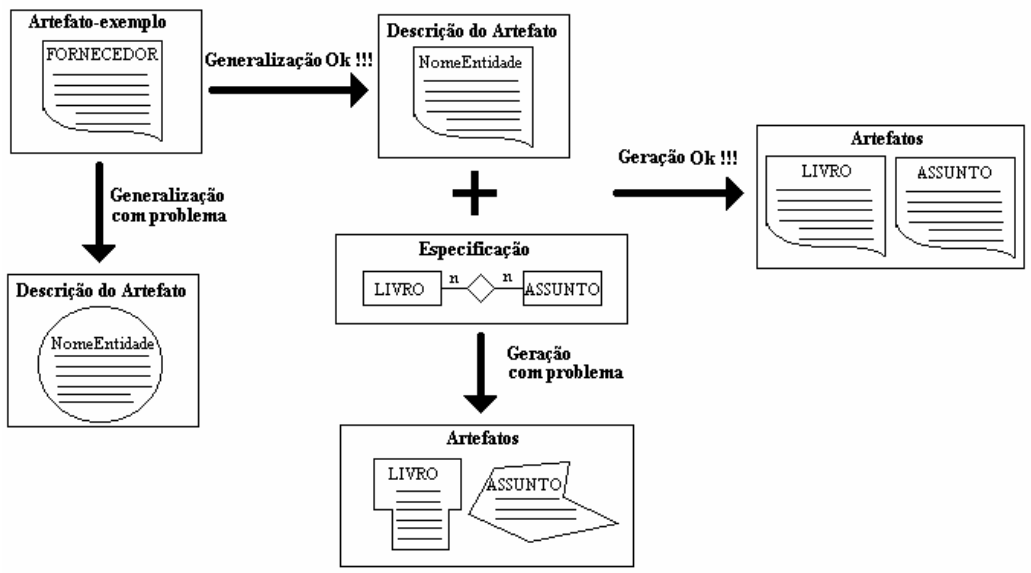


Figura 5.4 Visão esquemática da introdução de faltas

As faltas introduzidas durante a generalização do artefato-exemplo são difíceis de serem identificadas automaticamente, uma vez que o ambiente de geração não tem como verificar as interdependências dos diversos pontos que implementam um *hot-spot* (Fig. 5.5). Por exemplo, suponha que num programa de entrada de dados escrito em C, o *hot-spot Habilitar crítica de CPF*, envolva a colocação de um `#include` da biblioteca de CPF no início do arquivo e uma chamada da função de CPF após a perda de foco do campo. Durante a construção do arquivo de meta-descrição, caso o desenvolvedor somente coloque o *tag* condicional relativo à chamada da função CPF, o arquivo C quando for compilado irá apresentar um erro de `#include`. Para identificar este erro durante a construção do arquivo de meta-descrição seria necessária uma ferramenta que "conhecesse" a semântica do artefato a ser gerado. Em nossa proposta, não foi possível utilizar este tipo de ferramenta porque o gerador construído produz os artefatos através da composição de texto. No sentido de minimizar este tipo de erro, adotamos a técnica de inspeção [Staa 2000] para verificar os arquivos de meta-descrição criados.

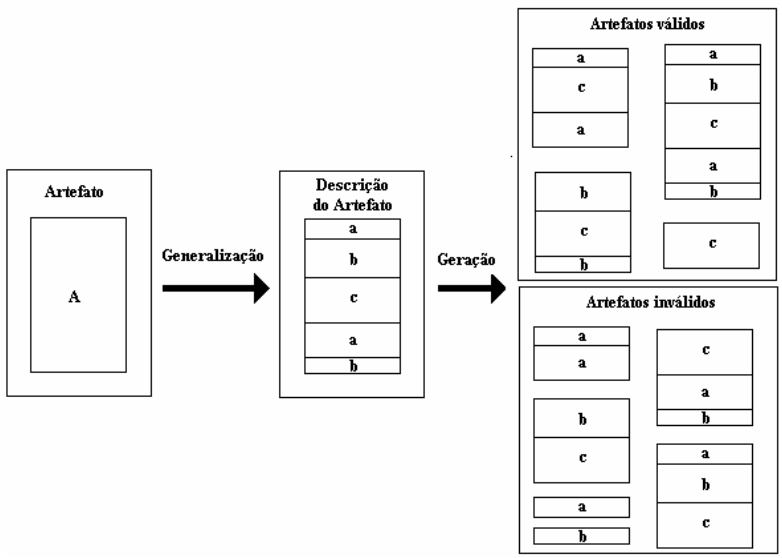


Figura 5.5 Composição do artefato gerado

Na tabela 5.1 são apresentados alguns dos principais problemas encontrados nos experimentos realizados para avaliar o processo de construção proposto.

Problema	**Erro na execução do artefato gerado**
Origem	Etapa de construção do artefato-exemplo
Causa	O artefato-exemplo possui uma falta que não foi identificada durante a sua construção. Esta falta é propagada pelo gerador para todos os artefatos gerados.
Defeito	**Artefato gerado não apresenta todas as funcionalidades do artefato-exemplo.**
Origem	Etapa de *Hot-spot mining*
Causa	Identificação parcial dos hot-spots. Os arquivos de meta-descrição criados pelo desenvolvedor não relacionavam todos os hot-spots com isto o gerador não consegue reproduzir um artefato similar ao artefato-exemplo.
Problema	**Artefato gerado apresenta erros de compilação**
Origem	Etapa de *Hot-spot mining*
Causa	Identificação parcial dos *hot-spots*. Os arquivos de meta-descrição criados pelo desenvolvedor não relacionavam todos os aspectos de um *hot-spot* levando o gerador a produzir artefatos incompletos.
Problema	**Artefato gerado apresenta erros de compilação**

Origem	Etapa de Criação do Coordenador da Geração
Causa	O programa de coordenação produz artefatos incompletos caso não contenha todas as chamadas das funções necessárias para gerar cada um dos arquivos componente do artefato.
Problema	**Artefato gerado apresenta erros de compilação**
Origem	Etapa de Criação do Verificador da Especificação
Causa	O programa de verificação da especificação não está verificando corretamente a completeza da especificação fornecida, com isto o gerador está produzindo artefatos a partir de especificações incompletas.
Problema	**Artefato gerado apresenta erros de compilação**
Origem	Etapa de Construção de Utilitários de Pós-Geração
Causa	O código dos utilitários de pós-geração contém faltas, e a sua utilização está introduzindo faltas no artefato gerado.
Problema	**Artefato gerado apresenta erros de execução**
Origem	Etapa de Criação do Verificador da Especificação
Causa	O programa de verificação da especificação não está verificando corretamente a sintaxe dos elementos da especificação fornecida, permitindo que código contendo faltas seja submetido a interpretadores (ex. caracteres não ascii em comandos SQL).
Problema	**Artefato gerado apresenta erros de execução**
Origem	Artefato-exemplo não previu restrições da plataforma
Causa	O artefato permite o usuário fornecer dados conflitantes com a plataforma de execução (ex. caracteres especiais ao recuperar textos HTML).

Tabela 5.1 Principais Problemas encontrados no processo de construção

Diversos erros cometidos ao desenvolver o artefato-exemplo deveram-se à falta de conhecimento quanto às restrições da plataforma. Caso os artefatos alvo tivessem sido desenvolvidas de forma tradicional, o custo da correção seria muito elevado. Por outro lado, a correção em um único lugar no gerador e a subseqüente geração de uma nova versão permite uma correção completa e rápida do problema identificado.

5.4 – Prototipação

O processo proposto viabiliza uma evolução sistemática do gerador que permite um baixo custo na construção de protótipos. Os protótipos criados servem para verificar os aspectos de adequação e de utilizabilidade do artefato-gerado.

A prototipação consiste na utilização do gerador para criar diferentes membros de uma família de artefatos até que o artefato gerado atenda as necessidades do seu usuário. Caso a geração não consiga produzir um artefato satisfatório, é necessário iniciar a criação de uma nova versão do gerador. Um artefato gerado é modificado até que as novas necessidades do usuário sejam atendidas. Após a estabilização das alterações, o artefato alterado passa a ser o artefato-exemplo para construção da nova versão do gerador (Fig. 5.6). A evolução de geradores permite que se produza artefatos cada vez mais próximos dos anseios do usuário por meio do aumento do escopo do gerador.

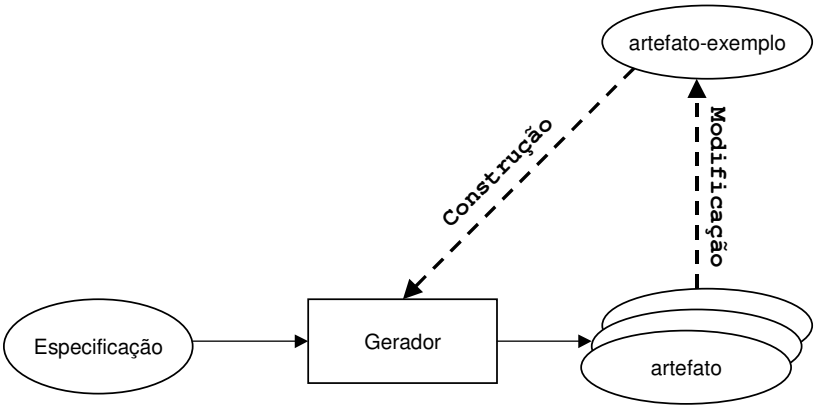


Figura 5.6 Prototipação + Evolução do gerador

5.5 – Abordagem Formal

Como garantir que o gerador construído realiza corretamente as transformações da especificação até o artefato gerado? Como garantir que a composição do artefato esteja correta (Fig. 5.5)?

Na literatura, respostas para estas questões estão relacionadas com a abordagem empregada na geração.

Sistemas transformacionais formais. A geração é verificada através da formalização da linguagem de especificação e de seus mapeamentos nas linguagens de implementação.

Framewoks. A geração é verificada através da preservação da semântica do *framework* durante a sua instanciação.

Componentes. Uma forma de verificar a geração é através da descrição do processo de composição através de linguagens de composição de módulos e de descrição da arquitetura.

A seguir, detalharemos cada uma destas abordagens.

5.4.1 – Sistemas Transformacionais Formais

A geração baseada em sistemas transformacionais formais consiste numa série de refinamentos (transformações) a que uma especificação formal é submetida culminando na sua versão de implementação. A verificação deste tipo de geração é baseada na teoria utilizada para formalizar a especificação e as transformações.

A ferramenta SPECWARE do Kestrel Institute [SJ 1994, SMcD 1996] suporta a construção sistemática de programas executáveis a partir de especificações axiomáticas através de refinamentos sucessivos. Esta ferramenta acumula as experiências anteriores deste centro de pesquisa na construção de sistemas transformacionais formais [Smith1990]. A fundamentação teórica da ferramenta SPECWARE é a teoria de categorias que apresenta recursos para descrever a manipulação de estruturas em diferentes níveis de granularidade. Ao contrário da teoria de conjunto que é baseada na relação de pertinência, que induz a um estudo da estrutura interna das entidades do conjunto, a teoria de categorias estuda as propriedades externas dos objetos. Para definir um objeto, basta descrever suas interações com os demais objetos. Esta teoria é utilizada para descrever os diagramas e os refinamentos dos diagramas até a geração do código, mostrando que as propriedades semânticas são preservadas ao longo destes refinamentos.

Em [FM 1995] a teoria de categorias é apontada como um *framework* matemático que pode uniformizar os diferentes formalismos utilizados no

processo de desenvolvimento de software. Esta abordagem, além de fornecer mecanismos para relacionar os diferentes tipos de objetos (programas, especificações, etc.) intrínsecos a cada variedade de formalismo presente no processo de software, também fornece mecanismos para estruturação dos formalismos utilizados.

Em [Moreira 1998] é apresentado um trabalho com especificações algébricas que propõem a fundamentação teórica para a construção de uma ferramenta para a parametrização de componentes de software a partir de sua especificação formal. Através desta fundamentação teórica, pode-se garantir a validade das propriedades semânticas dos componentes a partir dos quais eles foram gerados. O princípio desta base teórica, conforme descrito por Moreira é:

> *"Dado um componente e um conjunto de propriedades semânticas satisfeitas por ele e que se quer preservar, encontrar um componente parametrizado do qual o componente original é uma instanciação possível e tal que possamos garantir que toda nova instanciação satisfará as propriedades citadas"*

Este princípio é verificado por meio da utilização da técnica de generalização de provas.

5.4.2 – Framewoks

Em [Fontoura 1999] as verificações das transformações de implementação, instanciação, e manutenção de *framewoks* OO são baseadas na semântica da linguagem proposta (extensão da UML). Neste trabalho, a linguagem de *design* é formalizada através da teoria de conjuntos. No contexto de *frameworks* OO, a semântica de um *design* é a combinação das semânticas de todas as suas classes. Já a semântica de uma classe, é baseada na semântica de seus métodos. O trabalho também descreve a semântica do *hot-spot*. Baseado na formalização da linguagem de *design*, o trabalho descreve o esforço computacional para verificação de cada tipo de transformação. Na figura 5.7, a função $Significado$ faz o mapeamento da descrição do *design* no plano sintático em sua representação semântica. A representação semântica de um *design* corresponde ao conjunto de aplicações que pode ser instanciada a partir do *design*.

As transformações do tipo implementação e manutenção de um *framework*, são transformações que preservam semântica:

$$Significado(\ framework\ ) = Significado(\ Transformação(\ t, framework\ ))$$

Um exemplo de transformação que não preserva semântica é a transformação do tipo instanciação. Neste caso, a função *Instância* é utilizada para verificar se uma aplicação é uma instância válida de um *framework*.

$$Se\ Significado(\ aplicação\ ) \subset Significado(\ framework\ )$$

$$Instância(\ framework, aplicação\ ) = Verdadeiro$$

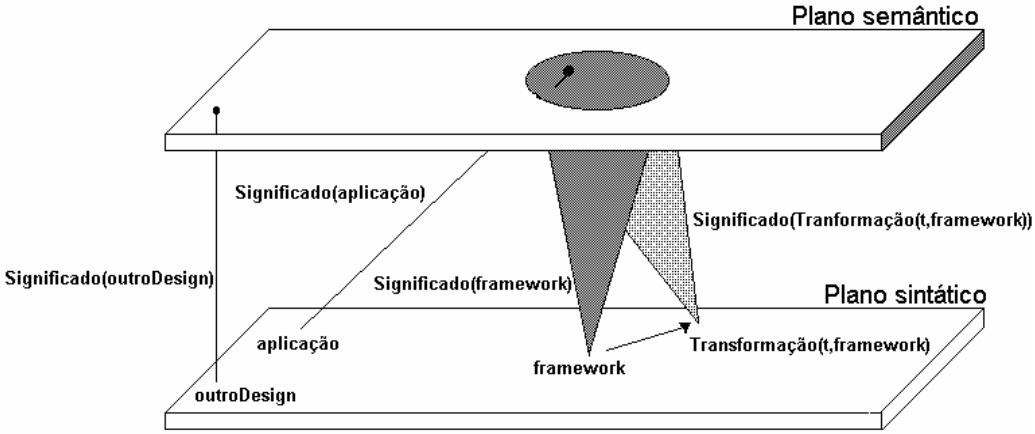


Figura 5.7 Abordagem de formalização [Fontoura 1999]

5.4.3 – Componentes

Como verificar se uma composição de componentes é válida?

Esta é a questão central da validação no contexto de geração baseada em componentes. A validação assume que os componentes estão implementados corretamente.

Os geradores do tipo GenVoca (seção 2.4.5) sintetizam sistemas de software através da composição de componentes de uma biblioteca. Nestes geradores, a composição é guiada por uma equação de tipo (Fig. 2.9), que especifica um componente do seu nível mais abstrato até a sua forma final de implementação. Uma composição é considerada correta caso a sua equação

de tipo respeite as regras de *design* dos componentes envolvidos. *Design rule checking* (DRC) [BG 1997] é o processo de aplicar regras de *design* para validar as equações de tipo. As regras de *design* correspondem às pré- e pós-condições referentes à utilização de um componente. Alguns algoritmos foram desenvolvidos para implementar o DRC, e foram testados em alguns dos geradores do tipo GenVoca [BG 1997]. O DRC é baseado nos trabalhos do Prof. D. Perry [Perry 1987, 1989a, 1989b] com modelos de interconexão de módulos. Nestes trabalhos, a construção e a evolução de sistemas são feitas através da composição de módulos e a validação da composição é feita por meio de modelos sintáticos e semânticos para interconexão de módulos. O esquema de verificação utilizado, tanto nos trabalhos do Prof. D. Perry como nos trabalhos com os geradores GenVoca não pretende ser completo (*full-fledged verification*). A verificação adotada é do tipo *shallow consistency checking*, que segundo o Prof. D. Perry é capaz de identificar e prevenir uma larga gama de erros lógicos. Os erros restantes devem ser identificados por outros mecanismos de verificação devidamente integrados ao processo (ex.: teste).

Uma outra abordagem de validação da composição de componentes é através da utilização de linguagens de interconexão de módulos MIL (*module interconnection language*) [PM 1987]. A especificação do *design* de um sistema através deste tipo de linguagem permite a verificação do *design* em relação a sua completeza e ausência de inconsistências. As linguagens de interconexão convencionais apresentam construtores para descrever as informações da arquitetura do sistema. Já as informações referentes a especificação do sistema e ao *design* detalhado não são descritas nestas linguagens. Em [ZSGS 1993] é apresentado um trabalho de extensão da MIL para tratar a utilização de *templates*, onde as informações intramodular passam a ser importantes para garantir a correta expansão de um *template* dentro de um módulo. Seguindo esta mesma linha, artigos referentes à arquitetura de software, tratam da questão da composição de diferentes arquiteturas através da utilização de linguagens de descrição e arquiteturas ADL (*architecture description language*) [MR 1997, MRT 1999].

Em [Broy 1997] é apresentado um modelo matemático para a utilização de componentes. O modelo proposto apresenta conceitos para a interface sintática e semântica de um componente, composição, desenvolvimento e refinamento sucessivo, interoperabilidade de sistemas heterogêneos bem como arquitetura de software.

EM [MQ 1994, MQR 1995] é apresentado um trabalho específico para verificação da corretude no desenvolvimento baseado em refinamentos sucessivos. Neste tipo de desenvolvimento, uma arquitetura mais abstrata é refinada numa arquitetura mais concreta que deve ser correta por construção. A idéia básica deste trabalho é o conceito de padrão de refinamento. Um padrão de refinamento é composto por um esquema de arquitetura abstrato e por um esquema mais detalhado que implementa o esquema abstrato. Este par de esquemas de arquitetura são corretos para um dado mapeamento. Por exemplo, um padrão pode mostrar como implementar um diagrama de fluxo de dados através de uma arquitetura cliente/servidor. A prova da corretude é feita com relação ao padrão; se o padrão estiver correto, as instâncias deste padrão de refinamento também estarão corretas. Neste caso, a prova é feita apenas uma vez para o padrão, não há necessidade de repeti-la para cada instanciação. O trabalho apresenta detalhadamente as técnicas de prova para verificação da arquitetura refinada. O objetivo futuro deste trabalho é o desenvolvimento de ferramentas de síntese de arquitetura do tipo daquelas utilizadas no *design* de circuito integrados.

5.5 – Resumo

A identificação das causas de problemas em artefatos produzidos por geradores não é uma tarefa fácil, visto que existem várias fontes potenciais de introdução de faltas: a especificação do artefato-exemplo, o próprio artefato-exemplo, os fragmentos de código que compõem o gerador, o ambiente de geração e a especificação fornecida para a geração.

Os compiladores utilizados ao longo do processo de construção exercem o papel de ferramentas que ajudam a antecipar a identificação de algumas das faltas sistemáticas no artefato final gerado. Durante as etapas de construção, o compilador do CASE verifica os erros sintáticos dos componentes do gerador. Na etapa de geração, o programa de validação de especificação evita a geração de artefatos a partir de especificações incompletas. Após a sua geração, o compilador da linguagem de programação do artefato é utilizado para identificar erros nos arquivos fontes do artefato gerado. Utilizando ferramentas de comparação de arquivos, pode-se comparar os arquivos gerados com os arquivos originais do artefato-exemplo, diferenças são sinais de problemas na construção do gerador.

A prevenção de problemas nos moldes propostos pelo CMM foi empregada para a identificação/análise dos principais problemas encontrados nos artefatos gerados. No processo proposto, as faltas são introduzidas tanto nas atividades de generalização como nas de geração. No caso da generalização, como o processo não considera a semântica dos artefatos gerados, a maioria das faltas introduzidas nesta etapa somente serão identificados posteriormente na compilação ou na utilização dos artefatos gerados.

Uma abordagem mais formal para descrever tanto o processo como gerador é a base para construção de ferramentas que automatizem o controle da qualidade. Apresentamos algumas referências nas áreas de sistemas transformacionais formais, *frameworks* e componentes cujos formalismos podem ser empregados no contexto do nosso trabalho.

Capítulo 6- Estudos de Caso

Este capítulo apresenta sete estudos de caso sobre a utilização do processo de construção proposto. O apêndice F apresenta algumas medições que foram realizadas durante os estudos de caso.

6.1 – ProtoBD: Prototipador de modelo de dados

O objetivo deste estudo de caso foi o de definir e delimitar as etapas do processo de construção de gerador de artefatos. ProtoBD é um gerador de protótipos de aplicações de manutenção de base de dados. A geração é feita a partir da especificação do modelo de dados da aplicação (Fig. 6.1).

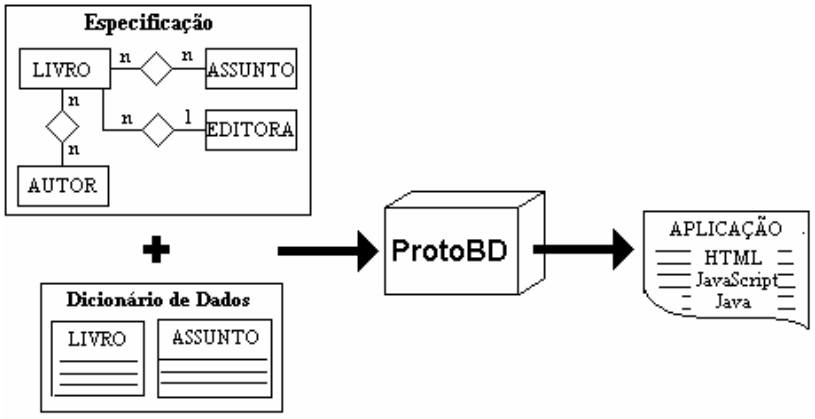


Figura 6.1 ProtoBD: Visão Esquemática da Geração

A primeira versão do ProtoBD gerava uma aplicação com as seguintes características (Fig. 6.2):

- Tela de manutenção para cada entidade do modelo;
- Validação dos campos utilizando informações da especificação;
- Navegação sobre os relacionamentos 1:N;
- Navegação sobre os relacionamentos N:N;
- Tela de manutenção para cada relacionamento N:N;

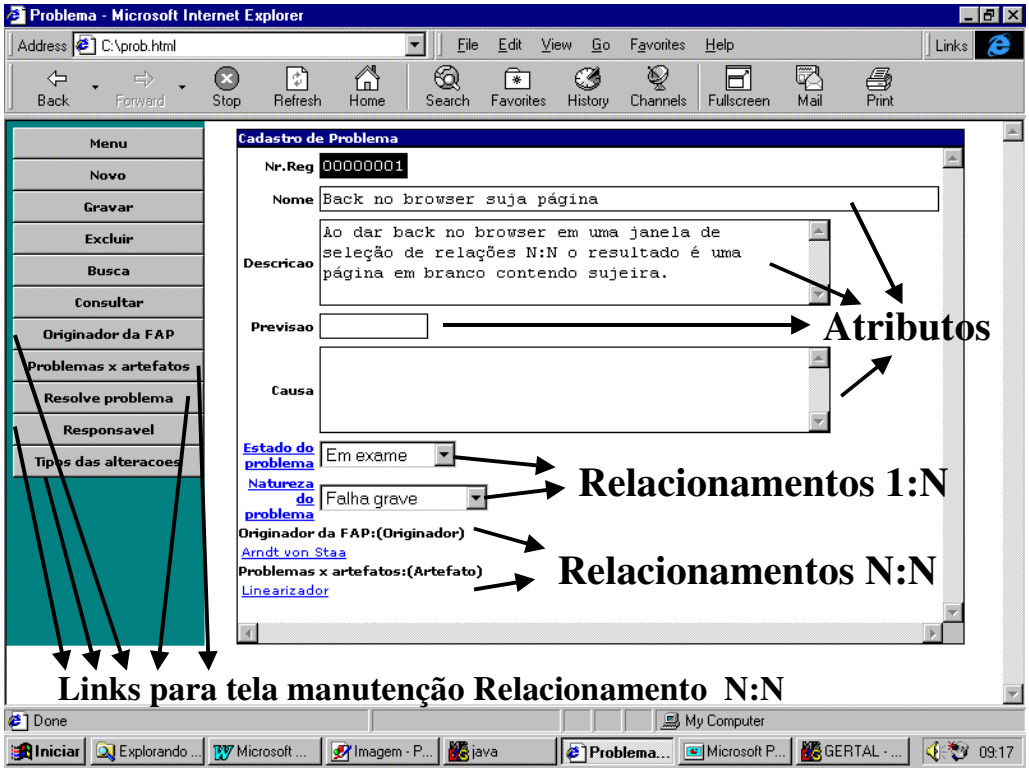


Figura 6.2 Tela de Manutenção de uma Entidade

A aplicação gerada utiliza uma arquitetura em três camadas (Fig. 6.3) [ED 1997, Caroli 1999]. A camada de interface foi codificada em HTML e JavaScript, a camada de negócio foi escrita em Java utilizando Servlet e camada de banco de dados foi escrita em Java e JDBC. Na arquitetura proposta por [Caroli 1999], a estrutura de diretório da aplicação (Fig. 6.4) reflete a arquitetura em três camadas, onde as classes em Java e os arquivos HTML estão colocados em sub-diretórios específicos e cada entidade do modelo de dados tem um programa em Java representando sua estrutura interna bem como um programa correspondente para cada camada da arquitetura.

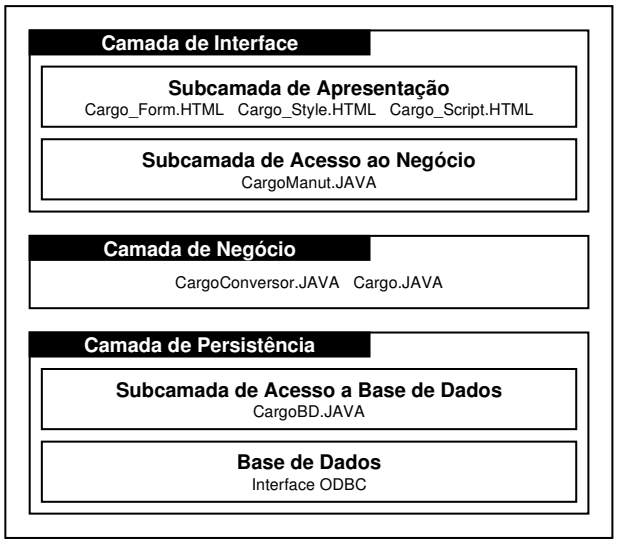


Figura 6.3 Implementação em três camadas de uma entidade do modelo de dados

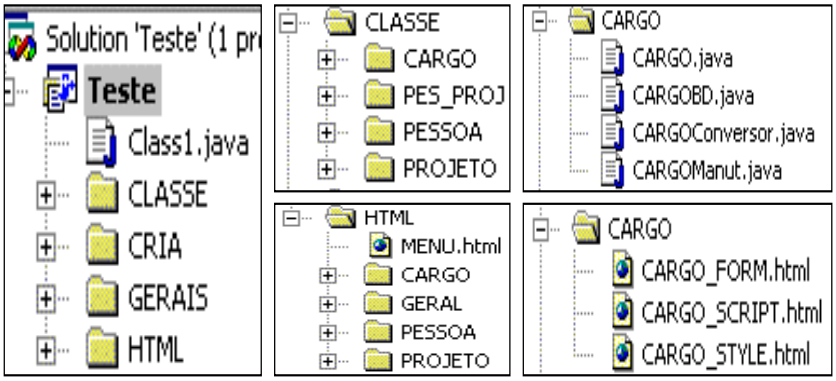


Figura 6.4 Estrutura de diretórios da aplicação gerada

Etapa de construção do artefato-exemplo: Inicialmente foi criado um artefato-exemplo correspondendo à aplicação de manutenção de uma base de dados para um modelo composto por 2 entidades, 1 relacionamento 1:N e 1 relacionamento N:N.

Etapa de construção do editor de especificação: Foi codificado o programa ERESP.FRM (linguagem Talisman) de edição do dicionário de dado associado ao diagrama de entidade-relacionamento. Os seguintes formulários de edição foram codificados: entidade, relacionamento, atributo e rótulo de ligação (ex.: cardinalidade do relacionamento).

Etapa de definição do artefato: Os arquivos Java e HTML que compunham o artefato-exemplo serviram de base para criação dos arquivos de meta-descrição do artefato. No sentido de organizar estes arquivos, a extensão .MJAVA foi utilizada para os arquivos de meta-descrição de programas em Java e a extensão .MHTML para os arquivos de meta-descrição dos arquivos em HTML (Figura 6.5).

Etapa de programação do gerador: Através do utilitário GERDESCR, os meta-arquivos foram transformados nos correspondentes programas de descrição de artefato. As regras de transformação foram implementadas através de funções escritas na linguagem Talisman. Foram escritas cerca de 70 regras (Apêndice C), sendo que a regra cuja função é obter o nome de um objeto, foi a regra mais utilizada, sendo aplicada cerca de 1300 vezes. A segunda regra mais utilizada que retornava o tipo do atributo foi aplicada cerca de 80 vezes (Apêndice F). Nesta etapa, foram escritos também o programa de coordenação da geração e o de verificação da especificação.

Etapa de implementação do gerador: O arquivo GERADOR.FRM foi montado a partir da concatenação dos arquivos de descrição do artefato, da biblioteca de regras e do programa de coordenação da geração. O programa de verificação, o programa de edição do dicionário de dados (interface de entrada) ERESP.FRM e o programa GERADOR.FRM foram compilados pelo Talisman sendo incorporados como ferramentas disponibilizadas pelo meta-CASE. Foram criados 2 utilitários de pós-geração, o primeiro decompõe o arquivo gerado nos diferentes arquivos da aplicação enquanto o segundo utilitário copia os arquivos da aplicação para a estrutura de diretórios definida pelo arquivo de projeto.

Arquivos gerais
Cria.mjava: Descreve o programa que executa a criação do banco de dados.
Menu.mhtml: Descreve o arquivo HTML com os links para cada entidade.
Camada de Interface
Classe.mhtml: Descreve o arquivo HTML com o formulário de entrada de dados de uma entidade.
Style.mhtml: Descreve a seção HTML que descreve o formato dos campos do formulário de entrada de dados de uma entidade.
Classescr.mhtml: Descreve o arquivo HTML com as funções JavaScript utilizadas pelo formulário de entrada de dados de uma entidade.
Classeponta.mhtml: Descreve o arquivo HTML com o formulário de entrada de dados de um relacionamento N:N.
RelNNscr.mhtml: Descreve o arquivo HTML com as funções JavaScript utilizadas pelo formulário de entrada de dados de um relacionamento N:N.
Manut.mjava: Descreve o programa que trata as ações disparadas pelo usuário para manipular uma entidade do modelo.
RelNNManut.mjava: Descreve o programa que trata as ações disparadas pelo usuário para manipular um relacionamento N:N do modelo.
Camada de Negócio
Classe.mjava: Descreve o programa dos construtores da classe da entidade
Conversor.mjava: Descreve o programa de conversão entre as camadas de banco de dados e de interface que implementam uma entidade.
RelNN.mjava: Descreve o programa dos construtores da classe do relacionamento N:N.
RelNNco.mjava: Descreve o programa de conversão entre as camadas de banco de dados e de interface que implementam um relacionamento N:N.
Camada de Banco de Dados
BD.mjava: Descreve o programa com as funções SQL de manutenção e acesso à tabela referente a uma entidade do modelo.
RelNNBD.mjava: Descreve o programa com as funções SQL de manutenção e acesso à tabela referente a um relacionamento N:N do modelo.

Figura 6.5 Arquivos de meta-descrição do artefato

6.2 – Utilização de diferentes linguagens de representação

O objetivo deste estudo de caso foi demonstrar a independência do processo de construção da linguagem de representação utilizada para especificar a aplicação.

No ProtoBD (seção 6.1), a linguagem de representação utilizada foi o diagrama de entidade-relacionamento. A especificação é registrada através dos diagramas e dos dicionários de dados associados ao diagrama.

Com o objetivo de utilizar outra linguagem de representação em uma mesma especificação, utilizamos o diagrama de fluxo de dados (DFD) disponível em Talisman como linguagem para definir os menus da aplicação (Figura 6.6, 6.8). Foi utilizada esta linguagem simplesmente pelo fato de estar disponível e ser diferente da linguagem entidade-relacionamento. A semântica utilizada para os diagramas é a seguinte:

- **Processo**: Corresponde a um arquivo HTML que tem links para outras páginas. Estas páginas podem ser outros menus ou páginas de manutenção de entidades do modelo. Cada processo pode referenciar zero ou mais entidades definidas pelo diagrama entidade-relacionamento. As entidades relacionadas estarão disponíveis no item de menu correspondente ao processo. Uma mesma entidade pode ser relacionada com vários processos.
- **Fluxo de Dados:** Corresponde a um link HTML. Fluxo bi-direcional significa que a página chamada tem um link para página chamadora.

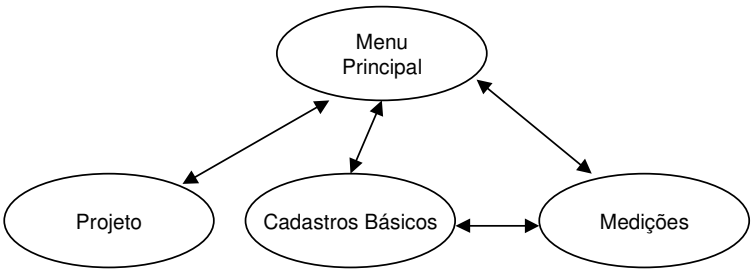


Figura 6.6 Definição da estrutura de menus da aplicação

O arquivo de meta-decrição MLINK.MHTML foi criado a partir de um arquivo de HTML correspondente a um menu da aplicação.

O arquivo MENUESP.FRM contém o programa o editor do dicionário de dados (Figura 6.7) associado ao diagrama de fluxo de dados. Os *links* anteriores e posteriores são determinados pelos fluxos de dados do diagrama. Já os *links* com os cadastros são selecionados pelo usuário a partir da lista de entidades do modelo.

Nome: **Cadastros Básicos**

Links Anteriores:
Menu Principal
Links Posteriores:
Medições
Links com Cadastros:
Metrica
Atividade

Figura 6.7 Dicionário de dados de um processo do diagrama de fluxo de dados.

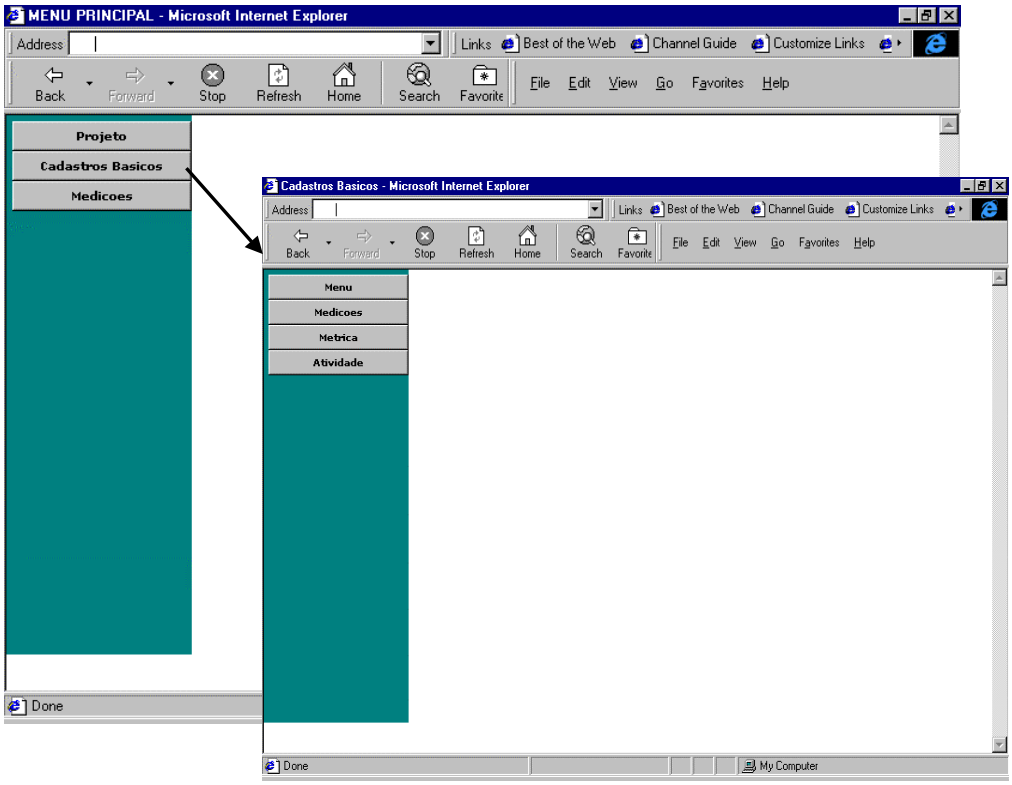


Figura 6.8 Estrutura de menus

Foi criado um programa de coordenação para criação da estrutura de menus. Este programa prepara uma lista dos *links* do processo para todos os processos do modelo e, em seguida, executa a função de geração de *link* definida pelo arquivo de meta-descrição MLINK.MHTML. Este programa de coordenação juntamente com o arquivo de descrição do link html foram concatenados ao arquivo original do gerador. O novo gerador e a tela de dicionário de dados do processo foram compilados pelo Talisman e incorporados como ferramentas disponibilizadas pelo metaCASE.

Este estudo de caso serviu para mostrar a independência do processo em relação à linguagem de representação utilizada na especificação da aplicação. Nesta versão o ProtoBD utiliza 3 linguagens de representação: Textual (dicionário de dados) e Diagramática (diagrama entidade-relacionamento e diagrama de fluxo de dados).

6.3 – Independência do processo em relação à linguagem de programação da aplicação final

A primeira versão do ProtoBD produzia aplicações escritas em Java, JavaScript, HTML e SQL, apesar da aplicação gerada ser composta por arquivos escritos em diferentes linguagens de programação, resolvemos realizar 3 experimentos para reforçar a verificação da independência do processo em relação à linguagem de programação do artefato gerado (Fig. 6.9).

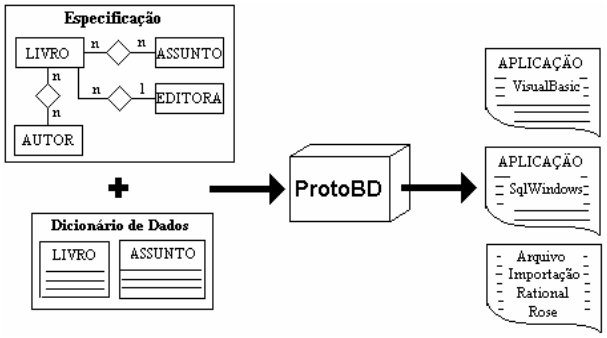


Figura 6.9 Geração de diferentes linguagens

O primeiro experimento correspondeu a uma nova versão do ProtoBD visando à geração de aplicações com arquitetura cliente-servidor utilizando VisualBasic como linguagem de programação. Cada entidade do modelo de dados tem um arquivo FRM para sua tela de manutenção. O processo de construção foi executado, e as aplicações geradas possuíam as mesmas funcionalidades da versão em três camadas. Durante a construção, observamos uma elevada reutilização das regras de transformação criadas anteriormente para o gerador da versão em três camadas.

O segundo experimento correspondeu a uma nova versão do ProtoBD visando à geração de aplicações com arquitetura cliente-servidor utilizando

SqlWindows como linguagem de programação. A aplicação final corresponde a um único arquivo, onde cada entidade possui uma janela do tipo *DialogBox* para manutenção da tabela referente à entidade do modelo. Neste experimento também foi observada uma elevada reutilização das regras de transformação criadas anteriormente para o gerador da versão em três camadas.

O terceiro experimento correspondeu à criação do arquivo de importação para o CASE Rose da Rational [Rational 1998]. A partir de um arquivo MDL gerado pelo CASE Rose foi criado um arquivo de meta-descrição de artefato. O programa de coordenação da geração percorre o dicionário das entidades e relacionamentos executando a função definida pelo meta-arquivo. O gerador produzido é capaz de exportar o modelo de dados armazenado no Talisman para ser importado pelo CASE Rose através de um arquivo de formato MDL.

6.4 – Utilização de Rotinas de Escape

Como foi descrito no capítulo 2, alguns geradores permitem que o usuário informe trechos de códigos que irão ser expandidos no código da aplicação gerada. Estes trechos, denominados de rotinas de escape, são inseridos em pontos especiais da aplicação como por exemplo, antes da atualização do banco de dados e depois da consulta ao banco de dados (Fig. 6.10, 6.14).

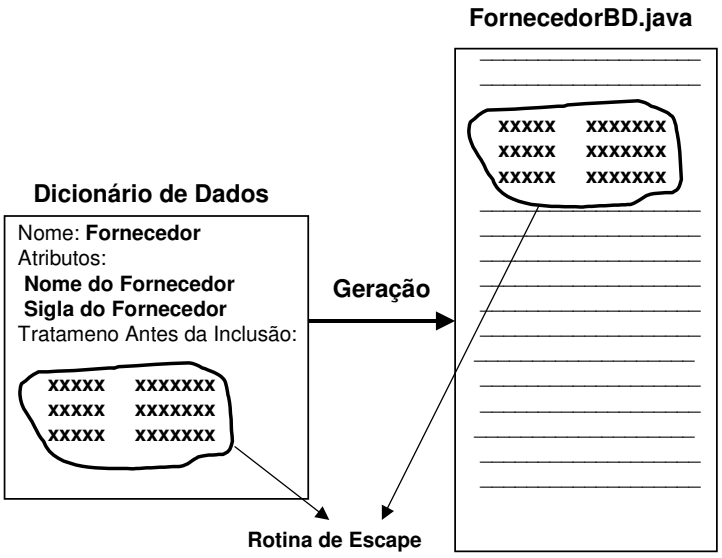


Figura 6.10 Rotina de Escape

No processo de construção proposto, a inclusão de uma rotina de escape no gerador, envolve as seguintes alterações:

- **Inclusão de campo no dicionário de dados:** Cada rotina de escape passa a ser um atributo de um meta-dado do Talisman. Por exemplo, no ProtoBD foi criado um novo atributo para o objeto entidade do modelo de dados. O atributo **Texto TxtEscapeIncluir** é um campo que o usuário pode digitar um trecho de código em java (Fig. 6.11). Este código será expandido dentro do arquivo do artefato na posição definida pelo *tag* condicional associado à rotina de escape.

- **Criação de *tag* condicional:** No arquivo de meta-descrição do artefato, no local de expansão da rotina de escape é inserido um *tag* condicional (Fig. 6.12). Durante a geração, se o campo da rotina estiver preenchido, a regra de expansão da rotina de escape é aplicada.

- **Regras da Rotina de Escape:** O *tag* condicional associado à rotina de escape utiliza 2 regras de transformação (Fig. 6.13). A primeira regra verifica se o campo da rotina de escape está preenchido, a verificação é feita por meio da checagem do valor do atributo referente à rotina de escape. Já a segunda regra, retorna o valor do atributo referente à rotina de escape, para ser expandido na posição desejada.

As figuras abaixo exemplificam a inclusão de rotina de escape em Java para limitar em apenas 2, o total de fornecedores cujos nomes comecem com a mesma letra.

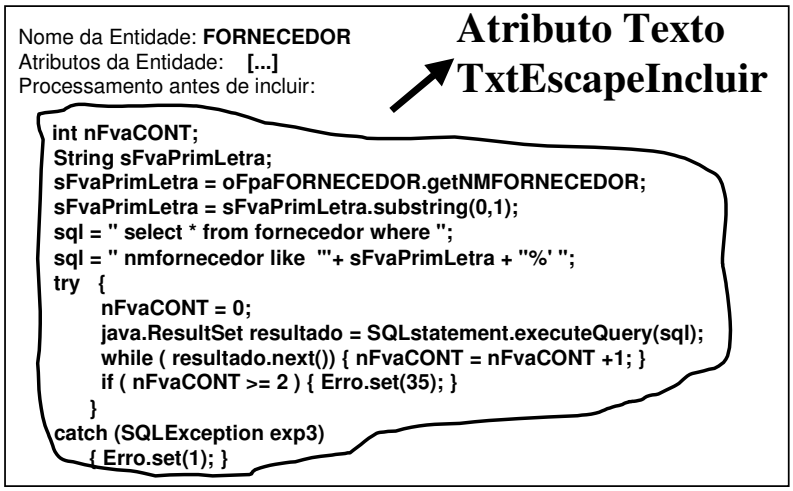


Figura 6.11 Dicionário de dados para rotina de escape.

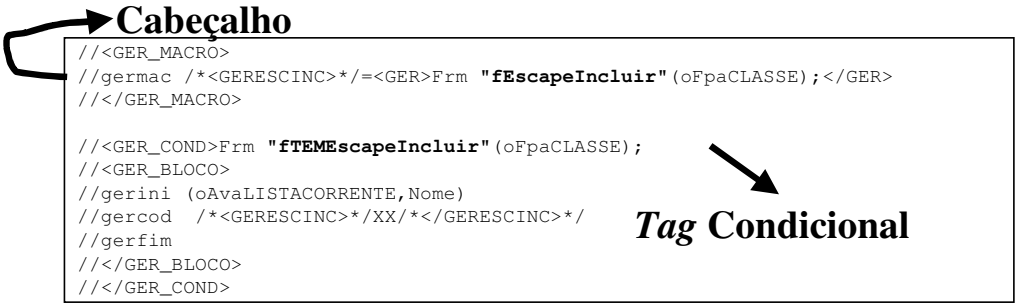
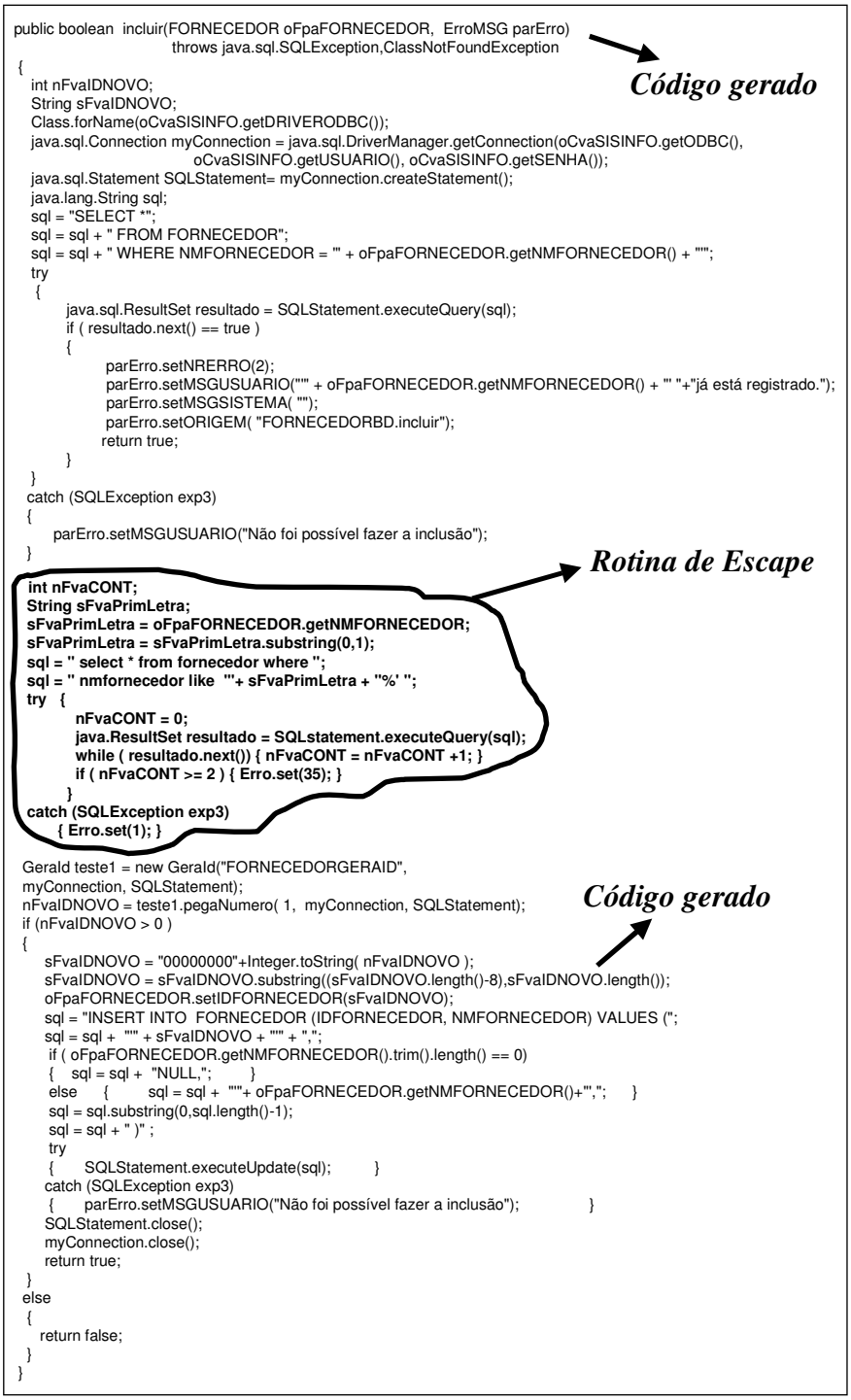


Figura 6.12 Trecho do meta-arquivo de descrição BD.MJAVA

```
InicFrm "fTEMEscapeIncluir" (Objeto oFpaENTIDADE)
   Se Existe ( [Texto TxtEscapeIncluir])
      Entao bAvaCOND = verdadeiro;
      Senao bAvaCOND = falso;
   Fim;
FimFrm

InicFrm "fEscapeIncluir" (Objeto oFpaENTIDADE)
   ComObjeto oFpaENTIDADE Faz
      Texto TxtEscapeIncluir;
   Fim;
FimFrm
```

Figura 6.13 Regras de transformação associadas a rotina de escape




```
public boolean  incluir(FORNECEDOR oFpaFORNECEDOR,  ErroMSG parErro)
                        throws java.sql.SQLException,ClassNotFoundException
{
   int nFvaIDNOVO;
   String sFvaIDNOVO;
   Class.forName(oCvaSISINFO.getDRIVERODBC());
   java.sql.Connection myConnection = java.sql.DriverManager.getConnection(oCvaSISINFO.getODBC(),
                              oCvaSISINFO.getUSUARIO(), oCvaSISINFO.getSENHA());
   java.sql.Statement SQLStatement= myConnection.createStatement();
   java.lang.String sql;
   sql = "SELECT *";
   sql = sql + " FROM FORNECEDOR";
   sql = sql + " WHERE NMFORNECEDOR = '" + oFpaFORNECEDOR.getNMFORNECEDOR() + "'";
   try
    {
          java.sql.ResultSet resultado = SQLStatement.executeQuery(sql);
          if ( resultado.next() == true )
          {
                 parErro.setNRERRO(2);
                 parErro.setMSGUSUARIO("'" + oFpaFORNECEDOR.getNMFORNECEDOR() + "' "+"já está registrado.");
                 parErro.setMSGSISTEMA( "");
                 parErro.setORIGEM( "FORNECEDORBD.incluir");
                return true;
          }
   }
   catch (SQLException exp3)
   {
        parErro.setMSGUSUARIO("Não foi possível fazer a inclusão");
   }
   int nFvaCONT;
   String sFvaPrimLetra;
   sFvaPrimLetra = oFpaFORNECEDOR.getNMFORNECEDOR;
   sFvaPrimLetra = sFvaPrimLetra.substring(0,1);
   sql = " select * from fornecedor where ";
   sql = " nmfornecedor like  '"+ sFvaPrimLetra + "'%' ";
   try   {
           nFvaCONT = 0;
           java.ResultSet resultado = SQLstatement.executeQuery(sql);
           while ( resultado.next()) { nFvaCONT = nFvaCONT +1; }
           if ( nFvaCONT >= 2 ) { Erro.set(35); }
          }
   catch (SQLException exp3)
        { Erro.set(1); }
  GeraId teste1 = new GeraId("FORNECEDORGERAID",
  myConnection, SQLStatement);
  nFvaIDNOVO = teste1.pegaNumero( 1,  myConnection, SQLStatement);
  if (nFvaIDNOVO > 0 )
  {
      sFvaIDNOVO = "00000000"+Integer.toString( nFvaIDNOVO );
      sFvaIDNOVO = sFvaIDNOVO.substring((sFvaIDNOVO.length()-8),sFvaIDNOVO.length());
      oFpaFORNECEDOR.setIDFORNECEDOR(sFvaIDNOVO);
      sql = "INSERT INTO  FORNECEDOR (IDFORNECEDOR, NMFORNECEDOR) VALUES (";
      sql = sql +  "'" + sFvaIDNOVO + "',";
       if ( oFpaFORNECEDOR.getNMFORNECEDOR().trim().length() == 0)
       {    sql = sql +  "NULL,";        }
       else     {        sql = sql +  "'"+ oFpaFORNECEDOR.getNMFORNECEDOR()+"',";     }
       sql = sql.substring(0,sql.length()-1);
       sql = sql + " )" ;
       try
       {       SQLStatement.executeUpdate(sql);         }
      catch (SQLException exp3)
       {       parErro.setMSGUSUARIO("Não foi possível fazer a inclusão");               }
      SQLStatement.close();
      myConnection.close();
      return true;
  }
  else
  {
     return false;
  }
}
```

Figura 6.14 Exemplo de expansão da rotina de escape

6.5 – Evolução do Gerador de Artefatos

O objetivo deste estudo de caso foi verificar se o processo de manutenção proposto suportava diferentes tipos de alteração no gerador de *Browser* de modelo de dados. As seguintes alterações foram realizadas:

Exibição de títulos customizados

Na primeira versão do gerador, os títulos dos campos das telas correspondiam aos nomes dos objetos do modelo de dados, por exemplo, NMFORNECEDOR, QTDPRODUTO, etc. Esses nomes eram usados tanto para os títulos como para nomear as colunas e as tabelas do banco de dados (Fig. 6.15).

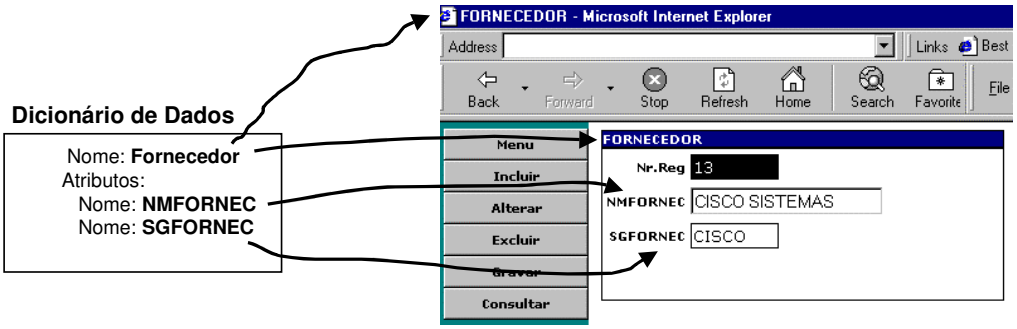


Figura 6.15 Atributo Nome = Titulo de janela/campo

A especificação foi alterada no sentido de permitir ao usuário fornecer, para cada objeto, o nome a ser exibido na tela, por exemplo, Nome do Fornecedor, Sigla do Fornecedor, etc. Na especificação foram criados 2 campos novos (Fig. 6.16). O primeiro campo registra o título a ser exibido por uma entidade ou relacionamento. Já o segundo, campo registra o título do campo na tela referente ao atributo de uma entidade ou relacionamento. Esta alteração implicou na criação das regras de transformação fTituloClasse e fTituloAtributo, que nos arquivos de meta-descrição passaram a substituir a regra fNomeClasse. Caso o nome para título não tivesse sido fornecido, o gerador continuaria utilizando o nome do objeto.

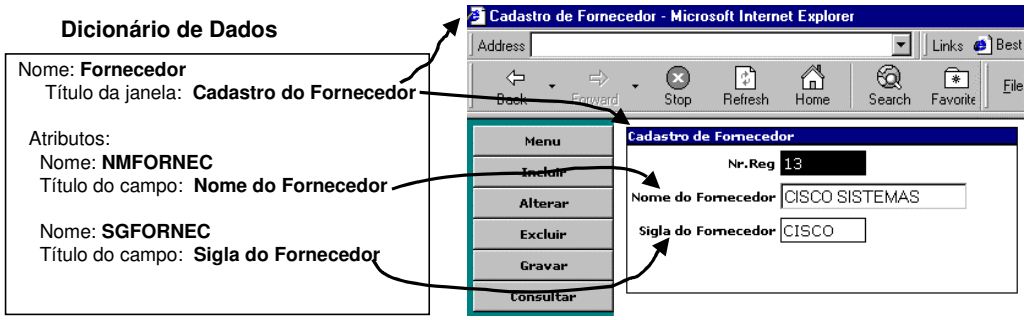


Figura 6.16 Atributo título de janela/campo

Crítica dos campos do formulário de entrada de dados

Os campos de entrada de dados passaram a ser criticados em relação a: campo obrigatório, valor compatível com o tipo do campo e valor dentro da faixa especificada. A crítica foi implementada através de uma função em JavaScript disparada quando o campo da tela é alterado (Fig. 6.17).

```
function ESPvalida(oFpacampo, sFpaantes,
          sFpaTIPO, sFpaFLOBRIG,
          nFpaVALMIN, nFpaVALMAX, nFpaTIPORET)
{
//Descrição:
// Funcao que valida o campo do formulário.
// As seguintes validações são feitas:
//   - obrigatoriedade do campo
//   - valor dentro do domínio do tipo
//   - valor numérico dentro da faixa especificada
//
//Parâmetros:
// oFpacampo => objeto corresponde ao campo a ser validado
// oFpaantes => valor anterior do campo
// sFpaFLOBRIG => flag de campo obrigatório: S ou N
// sFpaTIPO  => tipo do campo: N numérico
//                D data
//                H hora
//                S string
// nFpaVALMIN => valor mínimo de um campo numérico
// nFpaVALMAX => valor máximo de um campo numérico
// nFpaTIPORET => flag para setar foco no campo inválido: 0 ou
1
//
```

Figura 6.17 Função javascript de validação de campo

A introdução da função de validação na aplicação gerada causou as seguintes alterações (Fig. 6.18):

- Formulário do dicionário de dados da entidade: Foram criados três novos atributos.
    - Atributo de obrigatoriedade do campo;

- Atributo de valor mínimo de campo numérico;
- Atributo de valor máximo de campo numérico

- Arquivo de meta-descrição Classe.mhtml: A mensagem onchange para cada campo da tela foi incluída no arquivo que descreve o formulário HTML da tela de manutenção de um entidade. Nesta mensagem foi colocada a chamada da função ESPvalida cujos parâmetros da função são montados a partir das informações do dicionário de dados.

- Arquivo de meta-descrição Classescr.mhtml: A função ESPvalida foi incluída no arquivo que descreve as funções JavaScript do formulário HTML da tela de manutenção de um entidade.

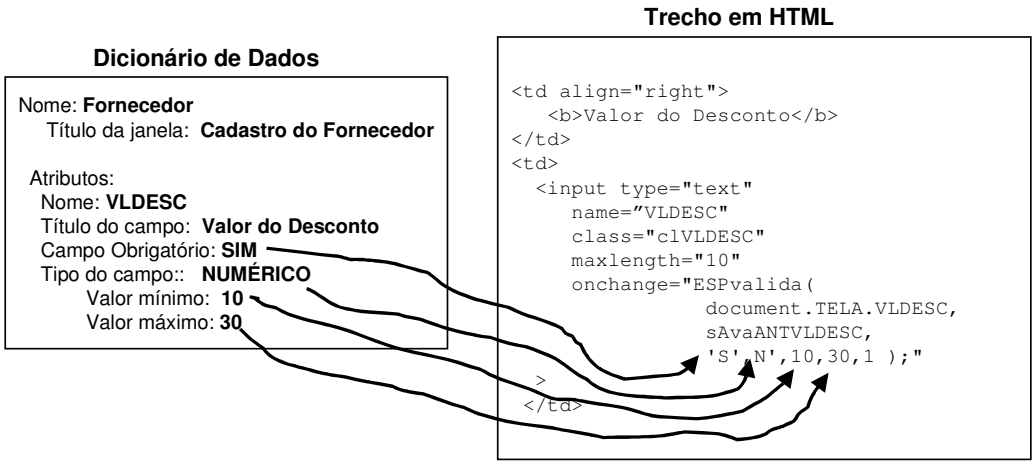


Figura 6.18 Relação dicionário de dados X função de validação de campo

Implementação de múltiplos relacionamentos

Nas primeiras versões do ProtoBD, uma entidade poderia ter apenas um relacionamento 1:N com outra entidade, o mesmo ocorrendo para o relacionamento N:N. Por exemplo, a entidade Produto poderia ter apenas um relacionamento 1:N com a entidade Fornecedor. Esta limitação foi resultado de um erro de projeto do gerador que estava utilizando o nome da Entidade da ponta do relacionamento para definir as colunas e tabelas do banco de dados (Fig. 6.18). A utilização do nome da ponta do relacionamento provocava a duplicidade de nomes quando uma mesma entidade participava de mais de um relacionamento. Para solucionar este problema, o gerador passou a utilizar o

nome do relacionamento para definir as colunas e as tabelas do banco de dados. Esta alteração permitiu que a aplicação gerada apresentasse uma multiplicidade de relacionamentos por entidade (Fig. 6.19).

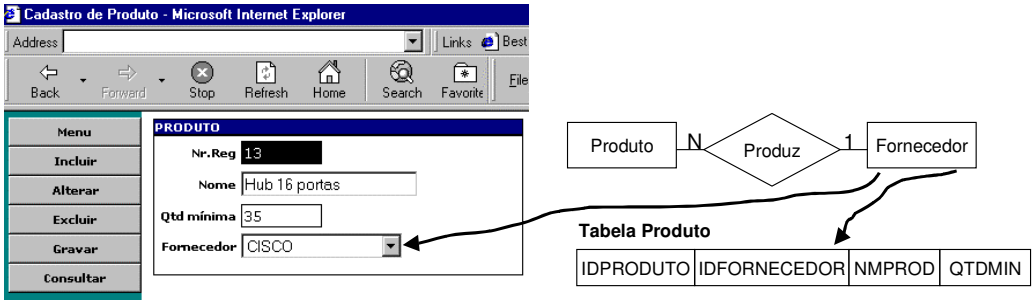


Figura 6.19 Limite de 1 relacionamento com uma mesma entidade

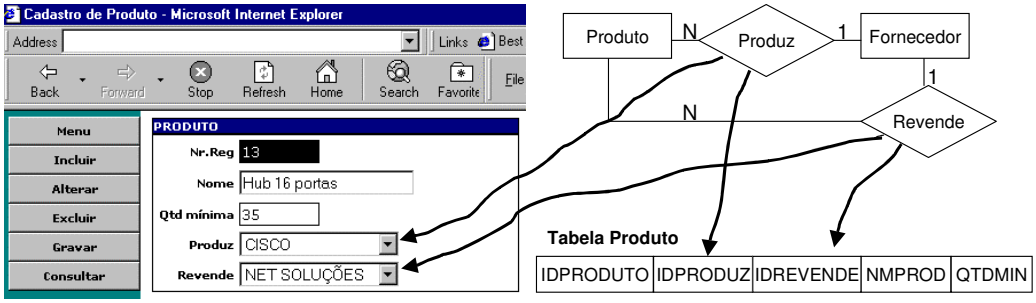


Figura 6.20 Múltiplos relacionamentos com uma mesma entidade

Implementação de auto-relacionamentos 1:N

A primeira versão do ProtoBD, não implementava os auto-relacionamentos 1:N. A partir de uma aplicação gerada, foi criado um exemplo de auto-relacionamento 1:N. Observamos que as alterações na aplicação estavam sempre próximas dos trechos de códigos que tratavam os relacionamentos 1:N. A especificação não precisou ser alterada pois o editor de diagramas de entidade-relacionamento do Talisman já permitia a criação de auto-relacionamentos 1:N. Em relação aos arquivos de meta-descrição, nos pontos próximos dos trechos de códigos que tratavam os relacionamentos 1:N foi incluído o tratamento dos auto-relacionamentos. No programa de coordenação da geração, foram criadas listas auxiliares para armazenar os auto-relacionamentos da entidade que estava sendo processada. A implementação do auto-relacionamento envolveu a alteração de sete arquivos de meta-descrição.

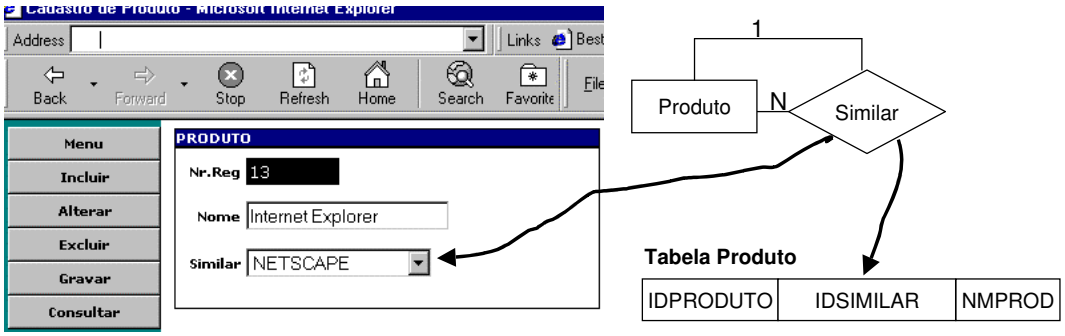


Figura 6.21 Auto-relacionamento 1:N

Implementação de auto-relacionamentos N:N

A primeira versão do gerador, não implementava os auto-relacionamentos N:N. A partir de uma aplicação gerada, foi criado um exemplo de auto-relacionamento N:N. Observamos que as alterações na aplicação envolveram a criação de arquivos similares aos arquivos que tratavam os relacionamentos N:N. A especificação não precisou ser alterada pois o editor de diagramas de entidade-relacionamento do Talisman já permitia a criação de auto-relacionamentos N:N. Foram criados seis novos arquivos de meta-descrição para realizar o tratamento dos auto-relacionamentos N:N. O programa de coordenação da geração foi alterado para realizar a chamada das funções definidas pelos novos meta-arquivos.

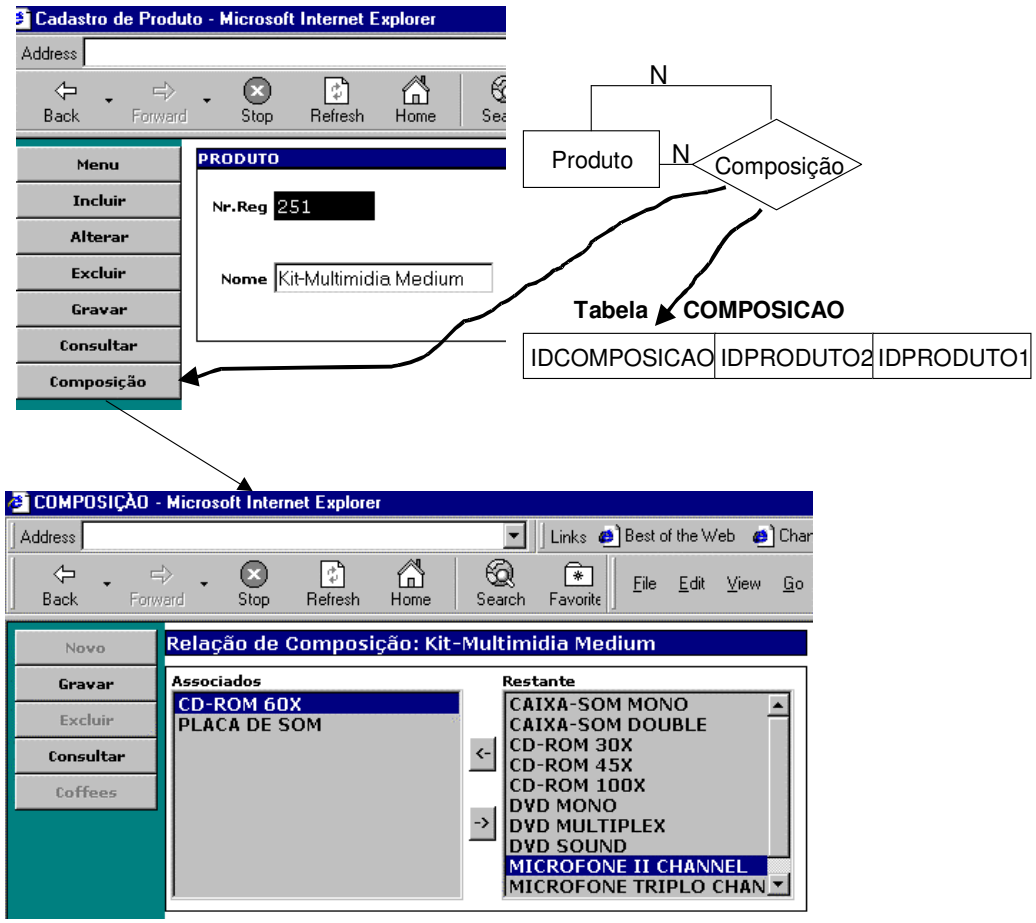


Figura 6.22 Auto-relacionamento N:N

6.6 – Utilização do ProtoBD

O objetivo deste estudo de caso foi utilizar o ProtoBd para criar aplicações para diferentes modelos de dados, no sentido de verificar alguma limitação quanto à diversidade e ao tamanho do modelo de dados.

Foram criados diferentes modelos de dados, contendo todos os tipos de relacionamentos binários suportados pelo gerador. Os modelos criados apresentavam relacionamentos e auto-relacionamentos 1:N e N:N. Tanto as entidades como os relacionamentos apresentavam diferentes tipos de atributos. As aplicações para estes modelos funcionaram corretamente.

Em relação ao tamanho do modelo de dados suportado pelo gerador, foi criada uma aplicação de gerência de projeto de software composta por 16

entidades e 21 relacionamentos. A aplicação gerada funciona corretamente e tem cerca de 99.000 linhas distribuídas em 650 arquivos organizados em 80 diretórios (Fig. 6.23).

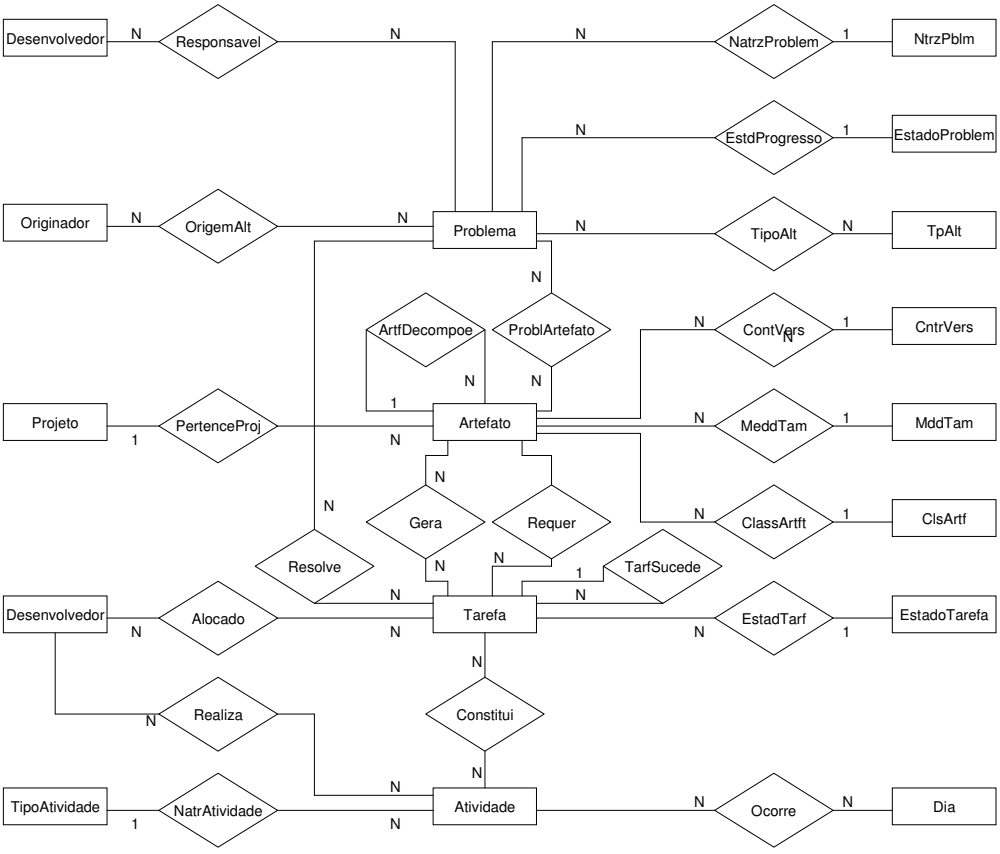


Figura 6.23 Exemplo de modelo de dados testado no ProtoBD

6.7 – Documentador HTML de Modelo de Dados

O objetivo deste estudo de caso foi construir um gerador que produzisse uma aplicação com funcionalidades diferentes daquelas das aplicações geradas pelo ProtoBd. O gerador escolhido foi o Documentador HTML de modelo de dados, a partir do modelo de dados são geradas páginas HTML do dicionário de dados (Fig 6.24).

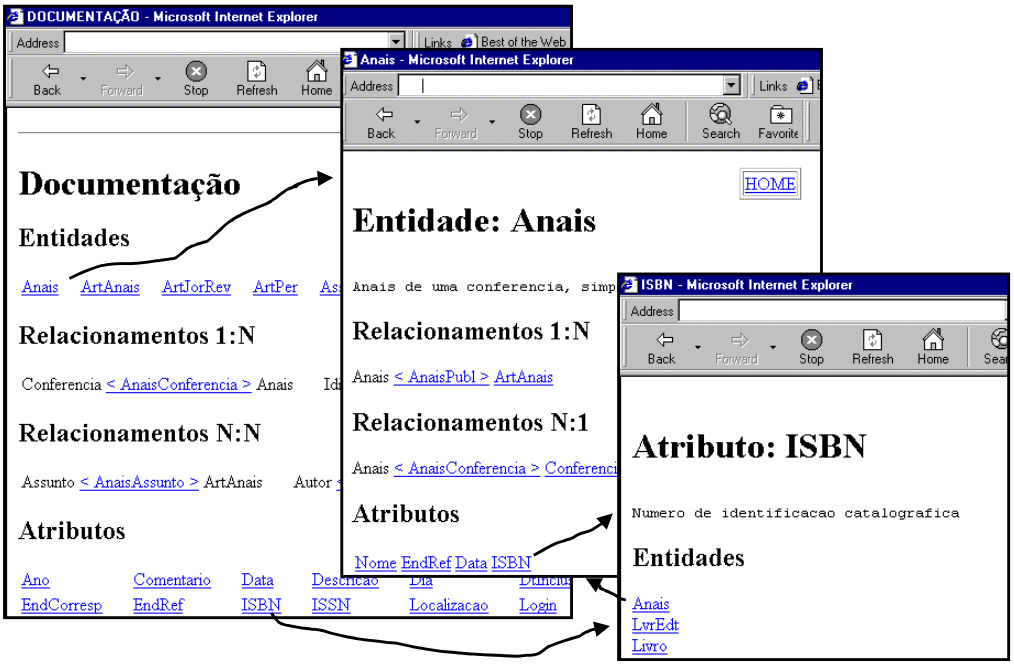


Figura 6.24 Documentação em HTML

Os arquivos HTML eram gerados segundo a seguinte lógica:

```
Se conjunto não é vazio
   Escrever título do conjunto
Fim-se
Listar conjunto
```

O Documentador foi construído executando os passos do processo de construção proposto. Foram criados quatro arquivos de meta-descrição do artefato (Fig. 6.25) contendo *tags* condicionais que implementam a lógica de geração acima descrita (Fig. 6.26).

Index.mhtml: Descreve o arquivo da home da documentação

Entidade.mhtml: Descreve o arquivo HTML com da documentação da entidade

Relacionamento.mhtml: Descreve o arquivo HTML com da documentação do relacionamento

Atributo.mhtml: Descreve o arquivo HTML com da documentação do atributo de uma entidade.

Figura 6.25 Arquivos de meta-descrição

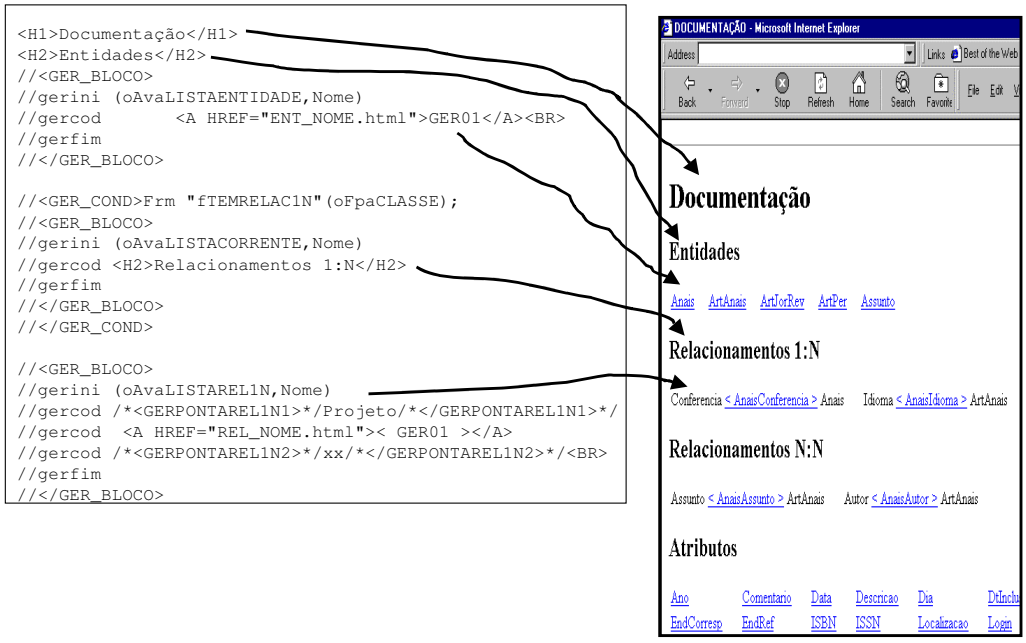


Figura 6.26 Relação arquivo de meta-descrição Index.mhtml
X
arquivo do artefato Index.html

6.8 – Resumo

Este capítulo apresentou sete diferentes estudos de casos que serviram para definir e delimitar as etapas do processo de construção proposto:

1. Envolveu a construção do ProtoBD prototipador de modelo de dados. 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 [Barosa 2000]. O gerador criado era composto por treze arquivos de meta-descrição de artefato. Este estudo de caso serviu para definir as etapas do processo.

2. Serviu para verificar a independência do processo de construção em relação a linguagem de representação utilizada na especificação. O diagrama de fluxo de dados foi utilizado para definir a estrutura de menus da aplicação gerada.

3. Mostrou a independência do processo de construção em relação à linguagem de programação da aplicação gerada. 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 [Rational 1998]. Este gerador permite a exportação de um modelo de dados armazenado no repositório do Talisman para ser importado no CASE Rose.

4. 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.

5. Permitiu 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.

6. 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. 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.

7. Foi criado um gerador de documentação HTML de um modelo de dados. Este estudo de caso serviu para avaliar a independência do processo de construção em relação ao tipo da aplicação gerada (manutenção, exploração, etc).

O apêndice F apresenta algumas medições que foram realizadas durante os estudos de caso.

Capítulo 7- Conclusão

Neste capítulo apresentamos um resumo da dissertação. Em seguida, apresentamos as principais contribuições, finalizando com sugestões de trabalhos futuros.

7.1 – Resumo do Trabalho

Neste trabalho definimos um processo de construção de geradores de artefatos. A experiência adquirida com o desenvolvimento do ambiente de geração de sistemas de medição (COMPASSO) [FS 1998, FSL 1998, FSF 1999] e a experiência acumulada em 10 anos de desenvolvimento de aplicativos, foram as fontes das duas principais características do processo proposto. A primeira característica é a utilização de uma ferramenta CASE como ambiente de geração. A ferramenta utilizada foi o meta-CASE Talisman que possui facilidades de customização que permitem uma ampla edição e manipulação do seu repositório de dados. A segunda característica é adoção de um processo "dirigido por exemplo" para construção do gerador. Esta idéia de começar a construção do gerador a partir de um exemplo do artefato a ser gerado, foi extraída da prática disseminada de reúso do tipo "cortar/colar/modificar", em que um programa antigo e estável serve de base para construção de novos programas.

No sentido de avaliar se uma ferramenta CASE possuía os requisitos necessários para servir de ambiente de geração, no capítulo 2 estão descritas algumas propostas de geradores de artefatos. Enquanto numa ferramenta CASE, a geração é através da manipulação de seus meta-dados, a maioria dos geradores analisados segue a linha de geração a partir de estruturas do tipo árvores de sintaxe abstrata. No capítulo 2, também são descritos trabalhos recentes com *frameworks*, cuja utilização e construção têm muita similaridade com o nosso trabalho.

No capítulo 3 foram identificados os requisitos que ferramentas CASE devem possuir para que se tornem adequadas ao processo de desenvolvimento de geradores preconizado. A ferramenta CASE deve disponibilizar mecanismos que permita a customização de seus formulários de

entrada de dados e seus arquivos de saídas. Geralmente, este tipo de customização é feito por meio de programas codificados na linguagem nativa da ferramenta CASE. Através dos formulários de entrada de dados, é registrado e mantido o dicionário de dados da especificação. Uma outra forma de capturar a especificação é por meio dos editores de diagramas disponibilizados pela ferramenta.

No capítulo 3, também é apresentado detalhadamente o método de identificação e de generalização dos *hot-spots*. Uma atividade importante dentro do processo de construção é a identificação das partes fixas e variáveis do gerador. O método de identificação utilizado é o *hot-spot mining* (termo da área de *frameworks*). Através deste método, os *hot-spots* do artefato-exemplo são identificados, e em seguida são generalizados. Para implementar a generalização, foi criado um conjunto de *tags* (marcadores) que são inseridos nas posições dos *hot-spots*. Construímos um utilitário (GERDESCR) que transforma um arquivo com marcadores (denominado de arquivo de meta-descrição de artefato) no programa de descrição do artefato. Este programa está codificado na linguagem nativa da ferramenta CASE utilizada como ambiente de geração.

As etapas do processo de construção são descritas no capítulo 4. Como o ponto de partida do processo é o artefato-exemplo, é necessário enfatizar que é muito importante que ele possua elevada qualidade, uma vez que o gerador replicará a sua implementação. Conseqüentemente faltas e deficiências porventura existentes serão replicados em todos os artefatos gerados. O resultado do processo de construção é um gerador capaz de gerar artefatos similares ao artefato-exemplo. Ressaltamos a necessidade de uma etapa final de controle de qualidade para certificar se o artefato gerado está em conformidade com especificação fornecida ao gerador. O capítulo 5 é dedicado ao assunto controle da qualidade.

Para ajustar as etapas do processo, foram realizados uma série de experimentos (capítulo 6). Os experimentos serviram para verificar a independência do processo em relação à linguagem de representação da especificação e em relação à linguagem de programação do artefato gerado.

Com auxílio dos experimentos também foi possível verificar a aplicabilidade do processo de manutenção proposto.

7.2 – Comparação com trabalhos relacionados

7.2.1 - Processo de construção

De forma análoga ao desenvolvimento de *frameworks*, o nosso processo de construção apresenta um comportamento evolutivo. O escopo do gerador é ampliado através da execução de sucessivos ciclos de desenvolvimento.

Similarmente ao processo baseado em meta-Geradores (seção 2.1.1) e em Famílias de Aplicações, o processo proposto apresenta um etapa de identificação das partes fixas e variáveis do gerador. As heurísticas utilizadas nesta identificação estão baseadas na técnica de *hot-spot mining* empregadas no desenvolvimento de *frameworks*. Diferentemente das propostas de desenvolvimento de *frameworks* analisadas, no processo proposto, o produto da etapa de *hot-spot mining* está precisamente definido (arquivo de meta-descrição do artefato).

De forma similar ao processo de desenvolvimento de [LN 1995], o processo proposto possui explicitamente uma etapa dedicada ao controle de qualidade. As diferenças entre o controle de qualidade do processo proposto e o controle dos demais processos estudados estão nos testes de verificação de conformidade e de especificação que foram incluídos nesta etapa. No teste de conformidade, o desenvolvedor compara os arquivos do artefato-exemplo com os arquivos de um artefato gerado pelo gerador que acabou de ser construído. Este tipo de teste é possível em virtude do processo proposto ter como ponto de partida um exemplo do artefato a ser gerado. Já o teste de verificação da especificação, consiste na execução do programa de verificação da especificação antes da execução do gerador.

Em relação a atividade de codificação do *pretty-printer*, de forma similar a proposta do STAGE [Cleaveland 1988], desenvolvemos uma linguagem que transforma um arquivo-fonte num programa de descrição do artefato. Na linguagem de marcadores desenvolvida, as partes invariantes (frozen-spot)

estão fora dos marcadores, e as partes variantes(hot-spot) estão entre marcadores. Os nossos tags de bloco e condicional, correpondem aos operadores de repetição *(%foreach*) e condicional (*%if*) da linguagem do Stage. Já o tag de substituição não tem operador correspondente na linguagem do Stage. O tag de sustituição permitiu um maior reúso do arquivo-fonte do artefato-exemplo.

7.2.2. Ambiente de geração

Os sistemas de transformacionais [Cleaveland 1988, Neighbors 1989, MM1993] estudados utilizam apenas especificações textuais. Em nosso trabalho, por utilizarmos a ferramenta CASE como ambiente de geração, os geradores produzidos estão preparados para receber entrada de dados textual ou diagramática.

Em relação a entrada de dados, alguns sistemas transformacionais não possuem interface para uma captura amigável da especificação, a entrada de dados do gerador corresponde a um arquivo texto contendo a especificação do artefato a ser gerado. Por estarmos utilizando um CASE, os geradores que construímos não apresentam limitação quanto a interface com usuário, as telas de decionário de dados e os editores de diagramas são utilizados para captura da especificação.

Geralmente a estrutura de armazenamento de um sistema transformacional é uma árvore de sintaxe abstrata (AST) que armazena a sintaxe e a semântica das linguagens de especificação utilizadas. Em nosso trabalho a estrutura de armazenamento é o repositório da ferramenta CASE. A organização do repositório da ferramenta CASE corresponde aos meta-modelos das linguagens de representação suportadas pela ferramenta.

Em relação a transformação da especificação na estrutura de armazenamento do gerador, enquanto nos sistemas transformacionais é necessário a codificação de um *parser* para realizar esta transformação, no nosso caso, não foi necessário a codificação deste transformador porque a ferramenta CASE realiza implicitamente esta transformação através de seus dicionários de dados e editores de diagramas.

Em relação a transformação da estrutura de armazenamento no artefato desejado, de forma análoga aos sistemas transformacionais, em nosso trabalho também é necessário a codificação de um transformador do tipo *pretty-printer*. Enquanto nos sistemas transformacionais, a geração é resultante da exploração do AST, em nosso trabalho a geração é resultante da exploração da base de dados do CASE através da definição de seu meta-modelo. Este tipo de exploração baseada em meta-dados reduz a complexidade da codificação do *pretty-printer*. O meta-dado apresenta uma granularidade maior dos que os elementos de um AST, correspondendo a abstrações mais próximas da realidade do desenvolvedor.

7.3 – Principais Contribuições

Definição do processo de construção de geradores de artefatos. No sentido de facilitar a replicação do processo proposto por outros grupos de pesquisa e a sua comparação com outros processos de construção, definimos o nosso processo de construção do geradores de artefatos na forma de um processo de software. Os insumos e as saída de cada etapa do processo proposto são descritos, permitindo um visão geral do fluxo de construção do gerador.

Processo de construção "dirigido por exemplo". 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.

CASE como ambiente de geração. 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. Nossos experimentos confirmaram a lista das características que uma ferramenta CASE deve ter para ser utilizada como ambiente de geração apresentada em [Hohenstein 2000]:

- 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 identificamos que a capacidade de criar novas linguagens de representação mais próximas do domínio da aplicação é uma característica adicional que a ferramenta CASE pode ter para facilitar ainda mais o processo de construção do gerador. Por exemplo, o nosso gerador de menus utiliza o diagrama de fluxo de dados como linguagem de especificação, o fluxo de dados foi utilizado dentro de um contexto diferente em virtude da dificuldade que encontramos para criar uma nova linguagem diagramática.

Meta-dado como unidade de transformação. Os geradores construídos serviram para comprovar os benefícios da utilização de meta-dado como unidade de transformação. A utilização do meta-dado uniformiza tanto a entrada como a saída do gerador. Os programas de edição da especificação (entrada) e os programas de descrição do artefato (saída) utilizam comandos que manipulam os mesmos meta-dados. Os meta-dados passíveis de manipulação pertencem ao conjunto de meta-dados das linguagens de representação utilizadas para capturar a especificação do artefato a ser gerado. Recentemente, alguns trabalhos foram publicados [PR 1999, Hohenstein 2000, Milicev 2000] confirmando a viabilidade desta abordagem.

Ciclo evolutivo do gerador. No processo proposto, o gerador de artefatos não é construído através de um processo de desenvolvimento tradicional e sim,

utilizando um processo evolutivo, através de ciclos sucessivos criar/alterar o artefato, identificando e inserindo *hot-spots* e as correspondentes transformações, produz-se o gerador. Cada ciclo produz uma versão executável do gerador. O artefato gerado pela última versão do gerador pode ser utilizado como artefato-exemplo para construção da próxima versão. Em cada ciclo, o artefato-exemplo pode apresentar novas funcionalidades ou correções em relação ao artefato-exemplo do ciclo anterior. O objetivo de cada ciclo é aumentar o escopo do gerador, aumentando a variedade de membros da família do artefato-exemplo que podem ser gerados. À medida que os ciclos vão se repetindo, as versões geradas vão ficando mais completas e mais genéricas. Este processo pode ser visto como um processo baseado em sucessivos protótipos conforme sugerido por [Boehm 1988].

Sistematização da generalização dos *hot-spots*. Os *tags* de substituição, de bloco e condicional foram criados para implementar a generalização dos *hot-spots* identificados nos arquivos fontes do artefato-exemplo. Estes *tags* são pontos de integração do artefato com sua especificação. Esta integração é feita através de funções que acessam o repositório de dados do CASE. A transformação dos *tags* em suas funções correspondente é feita de forma automática através do utilitário GERDESCR. O utilitário GERDESCR lê o arquivo de meta-descrição do artefato e substitui cada *tag* pela regra de transformação correspondente escrita na linguagem Talisman. O resultado final deste utilitário é o programa de descrição do artefato. Desta forma, o GERDESCR elimina a necessidade de codificação manual dos programas de descrição do artefato.

Criação do arquivo de meta-descrição do artefato. O arquivo de meta-descrição do artefato reduz o problema de impedância [Hohenstein 2000, Milicev 2000] 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.

Facilidade de inclusão de rotinas de escape. 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. Finalmente, as duas regras que implementam o *tag* condicional são codificadas: regra de verificação de existência, e regra de expansão da rotina de escape.

Verificação da conformidade do gerador. A utilização do artefato-exemplo como base de geração possibilita a realização de uma verificação simples de conformidade do gerador. Após construir o gerador, o artefato-exemplo deve ser re-gerado através do gerador. A especificação do artefato-exemplo é fornecida ao gerador, e em seguida a geração é disparada. Os arquivos do artefato-exemplo original são comparados com os arquivos do artefato-exemplo gerado, qualquer divergência é sinal de problemas no gerador.

Verificação automática da especificação. Antes de um artefato ser gerado, sua especificação é submetida a um programa de verificação. Este programa verifica a completeza da especificação em relação às informações necessárias para geração. O programa de verificação é codificado na linguagem nativa da ferramenta CASE e manipula os mesmos elementos utilizados nos programas de edição da especificação e de descrição do artefato. Esta verificação automática da completeza da especificação evita que o gerador produza artefatos incompletos.

Independência da linguagem do artefato gerado. O processo proposto é capaz de produzir geradores que produzam artefatos codificados em diferentes linguagens. Esta independência é possível porque a geração é feita através da composição de trechos dos arquivos do artefato-exemplo. Para o processo de construção, os arquivos do artefato-exemplo são tratados como arquivos-texto, independente da linguagem que estão escritos. Esta abordagem já foi utilizada

com sucesso em outro contexto de geração [Basset 1987, 1996, 1997, GM 1994].

7.4 – Trabalhos Futuros

Execução completa do processo de construção do gerador através da ferramenta CASE. Realização de experimentos nos quais as etapas de construção do artefato-exemplo, criação dos arquivos de meta-descrição e programação do gerador sejam implementadas dentro da ferramenta CASE. Neste caso, como todos os produtos intermediários do processo de construção estão armazenados no repositório de dados do CASE, possivelmente surgirão novas oportunidades de reúso e de integração da documentação do artefato-exemplo com a documentação do gerador.

Inclusão de mecanismos de detecção de erros no utilitário GERDESCR. Na versão atual do GERDESCR, não há mecanismos de verificação de erros na composição dos *tags* presentes no arquivo de meta-descrição do artefato. A detecção de erros do arquivo de meta-descrição foi deixada para ser feita pelo compilador do CASE, quando da compilação do programa de descrição de artefato correspondente ao arquivo de meta-descrição. A idéia é incluir no GERDESCR, mecanismos de detecção de erros sintáticos na composição de *tags* (ex.: *tag* de bloco sem o *tag* de fim de bloco).

Utilização de outras ferramentas CASE como ambiente de geração. Com objetivo de mostrar a independência do processo em relação ao ambiente de geração utilizado, experimentos devem ser realizados utilizando uma outra ferramenta no lugar do meta-CASE Talisman.

Estudo de alternativas de documentação do gerador. Diferentes alternativas de documentação devem ser examinadas, no sentido de buscar a redução do tempo de aprendizado e de facilitar a manutenção do gerador. Uma documentação que possua mecanismos de hiperdocumento integrando os arquivos do artefato-exemplo, os tags de um *hot-spot*, e a especificação do artefato a ser gerado parece ser bem adequada para os geradores construídos pelo nosso processo.

Verificação automática dos arquivos de meta-descrição de artefato. A identificação parcial de *hot-spots* ou a generalização parcial de um *hot-spot*

pode produzir um arquivo de meta-descrição sintaticamente correto porém incorreto do ponto de vista semântico. Na versão atual, este tipo de erro somente pode ser detectado após a construção do gerador, quando os artefatos gerados podem apresentar problemas de conformidade com sua especificação. O utilitário GERDESCR de alguma forma deve passar a receber informações complementares que permitam a detecção de erros semânticos nos arquivos de meta-descrição.

Criação de geradores para outros domínios de problema. Os geradores criados em nossos experimentos são fortemente baseados em modelos de dados. No sentido de mostrar a independência do processo de construção em relação ao domínio do problema novos geradores serão construídos. Por exemplo, gerador para construção de sites, e gerador para exploração de base de dados estatísticas (ex.: medição de software).

Melhorias no ProtoBD. O ProtoBD vai ser estendido para a partir de uma mesma especificação gerar o sistema de autoria e o sistema de exploração. Será também incluída a possibilidade de geração do módulo de controle de acesso (login e senha) com definição dos direitos de uso.

Avaliação quantitativa da utilização do gerador. É necessário realizar mais experimentos para quantificar os benefícios evidenciados com a adoção da tecnologia de geradores. As medições podem ser baseadas nos experimentos realizados em [GM 1994, BMJH 1996, Kitchenham 1996**,** McKBH 1996].

Apêndice A - Glossário

Aplicação: Coleção de componentes organizada para realizar uma ou mais funções interdependentes, visando apoiar atividades de um ou mais usuários.

Arquitetura: Macro organização (estrutura) do software.

Arquivo de meta-descrição do artefato: Arquivo intermediário entre o arquivo do artefato-exemplo e o programa de descrição do artefato. O arquivo de meta-descrição contém partes fixas (frozen-spots) que são cópias de trechos do arquivo do artefato-exemplo, e partes variáveis (hot-spots) que dependem da especificação. As partes variáveis são assinaladas por tags de transformação.

Artefato: 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, componente, módulo, aplicação, documentação para o usuário. Artefatos podem ou não ser entregues a um cliente ou usuário final.

Artefato-alvo: Artefato final a ser obtido por meio de um processo de desenvolvimento.

Artefato-exemplo: Artefato que serve de modelo para o tipo de artefato a ser produzido por um gerador.

CASE (*Computer-Aided Software Engineering*): Uma coletânea integrada de ferramentas utilizadas para desenvolver, controlar a qualidade e manter planos, especificações, projetos, código e documentação de sistemas de programação quaisquer. As facilidades de customização do CASE permitem que os usuários adaptem o ambiente às condições particulares da organização, do projeto, e da etapa no processo de desenvolvimento.

Componente: Um conjunto de um ou mais módulos, formando um todo coerente e implementando uma funcionalidade (conjunto de funções, classe, tipo abstrato) bem definida. Componentes são incorporados ao programa sem sofrerem alterações.

Domínio: Uma área de aplicação. São exemplos: pilhas, árvores, fluxo de caixa, planejamento e acompanhamento de projetos, controle de um

determinado processo químico, gestão acadêmica, edição de diagramas, etc.

Editor de especificação: Programa que permite a edição dos elementos da especificação armazenada no repositório de dados do CASE. O editor é capaz de manipular especificações escritas em diferentes linguagens de especificação.

Especificação: Um documento que determina: o que deve ser feito, sem dizer como fazê-lo; por que deve ser feito; todos os artefatos a serem entregues; os critérios de aceitação desses artefatos.

Formulário: Corresponde a uma tela de entrada de dados (ex.: tela do cadastro de clientes).

Framework: 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* [Fontoura 99].

Frozen-spots: São as partes do *framework* cujas implementações não variam com o uso do *framework*. No contexto de um gerador de artefatos, os *frozen-spots* definem as partes fixas do artefato gerado que independem da especificação fornecida ao gerador.

Gerador de artefatos: Uma ferramenta que produz um artefato completo e pronto para o uso a partir de sua especificação.

Hot-spots: São os pontos de flexibilização do *framework* e que podem ter uma implementação diferente para cada uso do *framework*. No contexto de um gerador de artefatos, os *hot-spots* definem as partes variáveis do artefato gerado que dependem da especificação fornecida ao gerador.

***Hot-spot* mining**: Método de identificação dos *hot-spots* que irão fazer parte do *framework* que está sendo construído. Em nosso processo de construção de geradores, o resultado da aplicação deste método é o arquivo de meta-descrição do artefato.

Linguagem de representação: Descreve os elementos, as estruturas (sintaxe) e o significado (semântica) destas estruturas. Linguagens de representação podem ser textuais (ex.: português, C, C++, Java), ou gráficas (por exemplo, diagramas de classes, diagramas de fluxo de dados, máquinas de estado).

Linguagem de especificação: Uma linguagem de representação utilizada para descrever uma especificação.

Meta-Gerador: Um ambiente de construção de geradores.

Módulo: Um artefato de programação que pode ser desenvolvido e compilado independentemente dos demais artefatos que compõem um determinado programa.

Programa de descrição do artefato: Programa que produz um artefato a partir de informações extraídas do repositório de dados do CASE.

Repositório de dados: Base de dados na qual o CASE armazena, de forma organizada, todos os artefatos desenvolvidos durante a vida útil de um software.

Rotina de escape: Um fragmento escrito na linguagem de implementação (ex.: C, Java, português) que é registrado na especificação e expandido pelo gerador em pontos predefinidos do artefato gerado. São exemplos: validação específica de um campo, processamento para formatação de uma saída, detalhe de uma especificação, etc.

Tag de transformação: Marcador inserido no arquivo de meta-descrição do artefato para definir um ponto de correspondência entre a especificação e a implementação do artefato.

Apêndice B - BNF do arquivo de meta-descrição

Elementos léxicos

$Caractere: é um caractere ASCII qualquer.

$Palavra: é uma seqüência de 1 ou mais caracteres diferentes de branco. Cada palavra pode ter até 250 caracteres.

$Identificador: é uma seqüência de 1 ou mais caracteres, começando por letra e contendo letras, dígitos ou o caracter sublinhado('_'). Cada identificador pode ter até 250 caracteres.

$Lista: é uma lista de objetos de uma especificação. Esta lista pode ser uma rela-ção pré-definida entre objetos do CASE ou uma lista montada por um programa.

$Atributo: é um atributo do objeto de uma lista (ex.: nome, aliás, ou texto).

arquivo de meta-descrição ::= cabeçalho corpo

cabeçalho::=

//**<GER_MACRO>**

{ regra_ substituição}

</GER_MACRO>

regra_de_substituição ::=

tag_substituição **=** **<GER>** chamada_função**</GER>**

tag_substituição::= seqüência_ substituição | /***<**ident_ substituição**>***/

seqüência_substituicão::= **$palavra**

ident_ substituição::= **GER $palavra**

chamada_função::= padrão_chamada [parâmetros_função] **;**

padrão_chamada[5]::= **Frm** "nome_função"

nome_função ::= **$identificador**

parâmetros_função ::= **($identificador** { **, $identificador** } **)**

corpo::=

{linha } |

{tag_bloco} |

{tag_condicional}

linha::=

{linha_fixa } |

{linha_com_tag_substituição} |

linha_fixa::= { seqüência }

seqüência::= **$caractere** { **$caractere** }

linha_com_tag_substituição::= {seqüência}{ seqüência_ substituição }{subst_ident}

subst_ident::= { /***<**ident_ substituição**>***/ seqüência /***</**ident_ substituição**>***/ }

tag_bloco::=

//**<GER_BLOCO>**

//**gerini**[6] **($lista, $atributo**[**, $atributo**] [**, $atributo**] [**, $atributo**] [**, $atributo**])

//**gercod**[7] {seqüência} { seqüência_ substituição}{subst_ident}{subst_bloco}

//**gerfim**

//**</GER_BLOCO>**

subst_bloco::= [**GER01**] [**GER02**] [**GER03**] [**GER04**] [**GER05**]

tag_cond::=

//**<GER_COND>** chamada_função

tag_bloco

//**</GER_COND>**

---

[5] O padrão de chamadas depende da sintaxe da linguagem nativa do CASE. Frm "nome_função" é o padrão de chamada de função escrita na linguagem nativa do meta-CASE Talisman.

[6] A construção gerini além de definir a lista que vai orientar o processamento do bloco, define também as substituições que serão feitas no escopo do bloco com relação aos atributos declarados da lista.

[7] O GER0n é substituído pelo n-ésimo atributo (parâmetro) do construtor //gerini,

Apêndice C- Regras de Transformação

Neste trabalho as regras de transformação são implementadas através de funções escritas na linguagem nativa do Talisman. A maior parte destas funções realizam consultas à especificação armazenada na base de dados do Talisman. A seguir apresentamos a lista das 68 funções utilizadas no gerador ProtoBD.

Nr	Nome
1	**fAtribNome**(Objeto oFpaDADO,Inteiro nFpaTIPO) Exibe[8] o valor do atributo Nome do objeto oFpaDADO. Tem opção de tirar os caracteres especiais do nome a ser exibido.
2	**fAtribTam**(Objeto oFpaDADO) Exibe o valor do atributo Alias AliasTamTot do objeto oFpaDADO
3	**fAtribTitulo**(Objeto oFpaDADO) Exibe o valor do atributo Alias AliasCampoTITULO do objeto oFpaDADO. Se o valor do atributo é vazio, a função retorna o valor do atributo Nome do objeto oFpaDADO
4	**fGetATRIBUTOS**(Objeto oFpaENTIDADE) Carrega[9] as listas auxiliares de tipo de dado com os dados do objeto oFpaENTIDADE. Cada dado é colocado na lista correspondente ao seu tipo. Ex: oAvaLISTAATRIBNUMERO
5	**fGetCardinalidade**(Objeto oFpaOrigem, Objeto oFpaLigacao) Atribui as variáveis globais da cardinalidade do relacionamento corrente
6	**fGetLinks(Objeto oFpaMENU)** Carrega as listas auxiliares dos links anteriores, posteriores, e de cadastro do objeto oFpaMENU.
7	**fGetRel** Atribui as variáveis globais dos objetos envolvidos (entidades das pontas e cardinalidades) com o relacionamento corrente.
8	**fGetRelacionamento**(Objeto oFpaREL) Carrega as listas auxiliares do tipo de relacionamento do objeto oFpaREL. Ex:(oAvaLISTARELN1,oAvaLISTAPONTAN1)
9	**fGetRelEntidade**(Objeto oFpaENTIDADE) Carrega as listas auxiliares dos tipos de relacionamentos que o objeto oFpaENTIDADE está envolvido.
10	**fGrafSQL**(Objeto oFpaVISAO,Sequencia sFpaSTRANTES, Sequencia sFpaSTRDEPOIS) Exibe o valor do atributo Texto TxtGraficoSQL do objeto oFpaVISAO. O valor é concatenado com o prefixo sFpaSTRANTES e com o sufixo sFpaSTRDEPOIS.
11	**fGrafTIPO** (Objeto oFpaVISAO) Exibe o valor do atributo Alias AliasGraficoTIPO do objeto oFpaVISAO. Ex: Gráfico de Barras, Pizza, etc.
12	**fGrafTITULO**(Objeto oFpaVISAO) Exibe o valor do atributo Alias AliasGraficoTITULO do objeto oFpaVISAO.

[8] Exibe: Colocar o dado no dispositivo de saída (ex.: mostrar um valor na tela)
[9] Carrega: Atribuir valores para os elementos de uma lista

13	**fIDCHAVETipoAtributo**(Sequencia sFpaEscopo, Inteiro nFpaMONTAGEM, Inteiro nFpaFLNOME) Exibe o valor do ID do objeto corrente. O valor exibido é formatado segundo os parâmetros da função. Exemplo de valores exibidos para entidade PRODUTO: String, IDPRODUTO, String sFpaIDPRODUTO, sCvaIDPRODUTO
14	**fIniInd** (Inteiro nFpaBASE) Inicializa o contador global
15	**fJAVAEscapeIncluir** (Objeto oFpaENTIDADE) Exibe o valor do atributo Texto TxtJAVAEscapeIncluir do objeto oFpaENTIDADE.
16	**fNomeClasse** (Objeto oFpaENTIDADE, Inteiro nFpaTIPO) Exibe o valor do atributo Nome do objeto oFpaENTIDADE. Tem opção de tirar os caracteres especiais do nome a ser exibido.
17	**fNomeDescr**(Objeto oFpaENTIDADE , Inteiro nFpaTIPO) Exibe o nome do atributo escolhido pelo para ser o campo a ser listado nos relacionamentos 1:N que a o objeto oFpaENTIDADE participa. Este campo vai ser carregado na combobox do relacionamento 1:N
18	**fNomeDescrRELN1**(Objeto oFpaRELN1) Exibe o nome do atributo escolhido pelo para ser o campo a ser listado nos relacionamentos 1:N da entidade da ponta N do relacionamento oFpaRELN1.
19	**fNomeID**(Objeto oFpaENTIDADE, Inteiro nFpaTIPO) Exibe o identificador padrão de uma entidade utilizando o padrão: ID + Nome do objeto oFpaENTIDADE (ex: IDPRODUTO)
20	**fNomeIDPONTAN1**(Objeto oFpaRELN1) Exibe o identificador padrão da entidade da ponta do relacionamento oFpaRELN1
21	**fNomePONTA01** Exibe o nome da entidade da ponta 1 do relacionamento corrente
22	**fNomePONTA01DESCR** Exibe o campo descritor da entidade da ponta 1 do relacionamento corrente
23	**fNomePONTA01ID** Exibe o identificador padrão da entidade da ponta 1 do relacionamento corrente
24	**fNomePONTA02** Exibe o nome da entidade da ponta 2 do relacionamento corrente
25	**fNomePONTA02DESCR** Exibe o campo descritor da entidade da ponta 2 do relacionamento corrente
26	**fNomePONTA02ID** Exibe o identificador padrão da entidade da ponta 2 do relacionamento corrente
27	**fNomePONTA1N(**Objeto oFpaREL1N) Exibe o nome da entidade da ponta do relacionamento oFpaRELN1
28	**fNomePONTAAUTO** Exibe o nome da entidade da ponta 1 do auto-relacionamento corrente
29	**fNomePONTAAUTODESCR** Exibe o campo descritor da entidade da ponta do auto-relacionamento corrente
30	**fNomePONTAAUTOID** Exibe o identificador padrão da entidade da ponta do auto-relacionamento corrente
31	**fNomePONTAN1**(Objeto oFpaRELN1) Exibe o nome da entidade da ponta do relacionamento oFpaRELN1
32	**fNomeREL** Exibe o nome do relacionamento corrente
33	**fPadBranco**(Inteiro nFpaTAM) Concatena um número n (nFpaTAM) de espaços em branco.

34	**fParametrosAlteracao**(Objeto oFpaENTIDADE, Inteiro nFpaDESLOC, Inteiro nFpaMAXPORLINHA, Inteiro nFpaMONTAGEM, Sequencia sFpaESCOPO, Sequencia sFpaSUFIXO, Inteiro nFpaFLNOME) Exibe uma lista de parâmetros com os atributos de uma entidade (identificador, dados e chaves estrangeiras). A formatação da lista é definida pelos parâmetros da função. Exemplos de lista: sFpaIDPRODUTO, sFpaNMPRODUTO String sFpaIDPRODUTO, String sFpaNMPRODUTO
35	**fParametrosInclusao**(Objeto oFpaENTIDADE, Inteiro nFpaDESLOC, Inteiro nFpaMAXPORLINHA, Inteiro nFpaMONTAGEM, Sequencia sFpaESCOPO, Sequencia sFpaSUFIXO, Inteiro nFpaFLNOME) Exibe uma lista de parâmetros com os atributos de uma entidade (dados e chaves estrangeiras). Esta lista não contém o identificador (ex:IDPRODUTO). A formatação da lista é definida pelos parâmetros da função. Exemplos de lista: sFpaNMPRODUTO, nFpaQTDMIN String sFpaNMPRODUTO, Integer nFpaQTDMIN
36	**fParametrosRELNN**(Objeto oFpaENTIDADE, Inteiro nFpaDESLOC, Inteiro nFpaMAXPORLINHA, Inteiro nFpaMONTAGEM, Sequencia sFpaESCOPO, Sequencia sFpaSUFIXO, Inteiro nFpaFLNOME) Exibe uma lista com os identificadores das entidades da ponta do relacionamento N:N corrente. Esta lista não contém o identificador do relacionamento (ex:IDRELFORNECE). A formatação da lista é definida pelos parâmetros da função. Exemplos de lista: sFpaIDPRODUTO, sFpaIDFORNECEDOR String sFpaIDPRODUTO, String sFpaIDFORNECEDOR
37	**fRELNNParametrosAlteracao**(Objeto oFpaENTIDADE, Inteiro nFpaDESLOC, Inteiro nFpaMAXPORLINHA, Inteiro nFpaMONTAGEM, Sequencia sFpaESCOPO, Sequencia sFpaSUFIXO, Inteiro nFpaFLNOME) Exibe uma lista com os identificadores das entidades da ponta do relacionamento N:N corrente e o identificador do relacionamento. A formatação da lista é definida pelos parâmetros da função. Exemplos de lista: sFpaIDREL, sFpaIDPROD, sFpaIDFOR String sFpaIDREL, String sFpaIDPROD, String sFpaIDFOR
38	**fSETTPOBJENT** Atribui a variável de flag do tipo de objeto corrente igual à entidade.
39	**fSETTPOBJRELAUTONN** Atribui a variável de flag do tipo de objeto corrente igual à auto-relacionamento N:N.
40	**fSETTPOBJRELNN** Atribui a variável de flag do tipo de objeto corrente igual à relacionamento N:N
41	**fSomaInd** Incrementa de uma unidade o contador global
42	**fTagFimCampo** Exibe o tag HTML de fim de campo (</textarea>)no caso do atributo corrente for do tipo texto longo
43	**fTagIniCampo** Exibe o tag HTML de início de campo no caso do atributo corrente for do tipo texto longo
44	**fTamanhoCampoSTR** Exibe o tamanho padrão de cada tipo de dado. No caso do tipo String é exibido o tamanho fornecido pelo usuário.
45	**fTEMATRIBDATA** (Objeto oFpaENTIDADE) Verifica se objeto oFpaENTIDADE tem um atributo do tipo data.

46	**fTEMATRIBHORA** (Objeto oFpaENTIDADE) Verifica se objeto oFpaENTIDADE tem um atributo do tipo hora.
47	**fTEMATRIBNUMERO** (Objeto oFpaENTIDADE) Verifica se objeto oFpaENTIDADE tem um atributo do tipo numérico.
48	**fTEMATRIBSTRING** (Objeto oFpaENTIDADE) Verifica se objeto oFpaENTIDADE tem um atributo do tipo string.
49	**fTEMATRIBTEXTO** (Objeto oFpaENTIDADE) Verifica se objeto oFpaENTIDADE tem um atributo do tipo texto longo.
50	**fTEMGRAFICO** (Objeto oFpaENTIDADE) Verifica se objeto oFpaENTIDADE tem algum gráfico associado.
51	**fTEMJAVAEscapeIncluir**(Objeto oFpaENTIDADE) Verifica se objeto oFpaENTIDADE tem rotina de escape de inclusão.
52	**fTemREL1N**(Objeto oFpaENTIDADE) Verifica se objeto oFpaENTIDADE participa de algum relacionamento 1:N
53	**fTemRELAUTO1N**(Objeto oFpaENTIDADE) Verifica se objeto oFpaENTIDADE participa de algum auto-relacionamento 1:N
54	**fTemRELAUTON1**(Objeto oFpaENTIDADE) Verifica se objeto oFpaENTIDADE participa de algum auto-relacionamento N:1
55	**fTemRELAUTONN**(Objeto oFpaENTIDADE) Verifica se objeto oFpaENTIDADE participa de algum auto-relacionamento N:N
56	**fTemRELN1**(Objeto oFpaENTIDADE) Verifica se objeto oFpaENTIDADE participa de algum relacionamento N:1
57	**fTemRELNN**(Objeto oFpaENTIDADE) Verifica se objeto oFpaENTIDADE participa de algum relacionamento N:N
58	**fTipoAtributo** (Sequencia sFpaEscopo, Inteiro nFpaMONTAGEM, Inteiro nFpaFLNOME) Exibe o tipo de dado do atributo corrente. A formatação do tipo de dado é definida pelos parâmetros da função. Exemplos para um atributo do tipo String: sFpa , s, String
59	**fTipoDescr** (Objeto oFpaENTIDADE, Sequencia sFpaEscopo, Inteiro nFpaMONTAGEM, Inteiro nFpaFLNOME) Exibe o tipo de dado do atributo que descreve o objeto oFpaENTIDADE. A formatação do tipo de dado é definida pelos parâmetros da função. Exemplos para um atributo do tipo String: sFpa , s, String
60	**fTipoID**(Objeto oFpaENTIDADE, Sequencia sFpaEscopo, Inteiro nFpaMONTAGEM, Inteiro nFpaFLNOME) Exibe o tipo de dado do identificador do objeto oFpaENTIDADE. A formatação do tipo de dado é definida pelos parâmetros da função. Exemplos para um atributo do tipo String: sFpa , s, String
61	**fVerificaAtributo** (Objeto oFpaENTIDADE,Objeto oFpaATRIBUTO) Faz a verificação da especificação do atributo oFpaATRIBUTO do objeto oFpaENTIDADE
62	**fValidaCARDINALIDADE** Verifica a especificação das cardinalidades de todos os relacionamentos do modelo. As cardinalidade válidas são: 1:N, N:1, e N:N
63	**fValidaDados** (Objeto oFpaENTIDADE, Sequencia sFpaTPOBJ) Verifica se o objeto oFpaENTIDADE tem atributos.
64	**fVerificaNumero**(Sequencia sFpaSTRCAMPO) Verifica se a string sFpaSTRCAMPO é um número.

Apêndice D –Documentação de *Hot-Spot*

Cada hot-spot é escrito num documento divido em sete seções:

- **Nome**: Identifica o *hot-spot*
- **Descrição**: Descrição textual do problema que o *hot-spot* resolve.
- **Tipo**: Identifica o método de adaptação utilizado e o grau de apoio fornecido.
- **Editor de especificação:** Lista os meta-dados e funções do editor de especificação envolvidos com o *hot-spot*.
- **Arquivo de meta-descrição do artefato**: Lista dos arquivos de meta-descrição envolvidos. Para cada arquivo é descrito os tags transformação associados ao *hot-spot*.
- **Regras de transformação**: Lista as regras de envolvidas no instanciamento do *hot-spot*.
- **Hot-spots relacionados**: Lista dos *hot-spots* necessários para implementação deste *hot-spot*.
- **Exemplos da instanciação:** Lista de arquivos do artefato que contém exemplos da instanciação do *hot-spot*.

A figura D.1 corresponde ao diagrama de classes da documentação proposta.

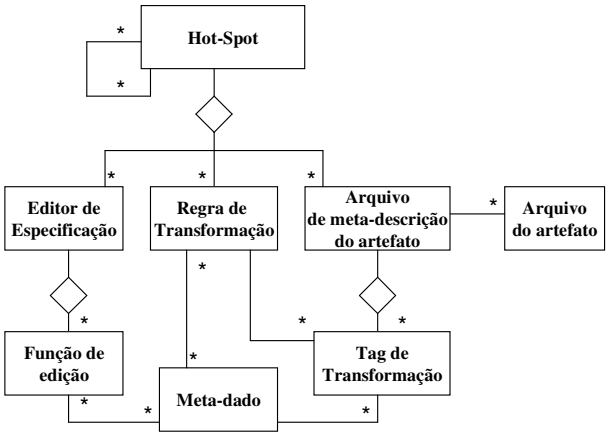


Figura D.1 Diagrama de Classes da Documentação do *Hot-spot*

A seguir é apresentado um exemplo de documentação (ex: *hot-spot* Validação de Campo.

Validação de Campo

Descrição

A validação do campo é executada em dois momentos: o campo perde o foco e o formulário é submetido.

É feita a validação de tipo de dado, campo numérico dentro da faixa de valores fornecida e campo obrigatório não preenchido.

A implementação da validação é feita por meio da função ESPValida que é disparada no evento `onchange` do campo do formulário HTML. A validação antes da submissão do formulário é feita por meio da função `ESPvalidaTELA`.

Este hot-spot é responsável pelo preenchimento da lista de parâmetros da chamada da função ESPValida para validação de cada campo do formulário HTML. A lista de parâmetros depende da especificação do campo.

Tipo

Modo de Adaptação: Substituir **Apoio**: Pattern

Arquivos de meta-descrição do artefato

CLASSE.MHTML

```
//germac /*<TALCAMPOVALID>*/=<TALISMAN>Frm "fValidaCampo";</TALISMAN>

//gercod onchange="ESPvalida(document.TELA.GER01, sAvaANTGER01,
//gercod                          /*<GERCAMPOVALID>*/8/*</GERCAMPOVALID>*/,1 );">
```

CLASSEPONTA.MHTML

```
//germac /*<TALCAMPOVALID>*/=<TALISMAN>Frm "fValidaCampo";</TALISMAN>
//gercod onchange="ESPvalida(document.TELA.GER01, sAvaANTGER01,

//gercod                          /*<GERCAMPOVALID>*/8/*</GERCAMPOVALID>*/,1 );">
```

CLASSEPONTAAUTO.MHTML

```
//germac /*<TALCAMPOVALID>*/=<TALISMAN>Frm "fValidaCampo";</TALISMAN>
//gercod onchange="ESPvalida(document.TELA.GER01, sAvaANTGER01,
//gercod                          /*<GERCAMPOVALID>*/8/*</GERCAMPOVALID>*/,1 );">
```

Editor da especificação

Meta-dados

Alias AliasObrigatorio

Alias AliasIntMin

Alias AliasIntMax

Funções

Editar definicao de atributo

Regras de Transformação

BIB.FRM

fValidaCampo

Hot-spots relacionados

Manutenção de campo

Apêndice E – Padrão para a escolha de nomes

Neste apêndice é descrito o padrão para a escolha de nomes utilizado tanto na codificação do artefato-exemplo como na codificação dos componentes do gerador.

Um padrão de nomes tem por objetivos [Staa 2000]:

- Uniformizar a forma de escolher nomes ao desenvolver ou manter programas.
- Tornar a criação de novos nomes independente do redator do programa.
- Facilitar a lembrança e o entendimento dos nomes dos elementos, reduzindo a necessidade de consulta a documentação complementar
- Facilitar a verificação do correto uso dos elementos do programa.
- Facilitar a localização do documento e a correspondente seção, contendo a especificação e a declaração completa do elemento denominado.

Regra 1: Todas as funções têm um prefixo. O valor do prefixo é “f”. Este prefixo permite diferenciar as funções criadas pelo programador das funções nativas da linguagem de programação.

Regra 2: Todas as variáveis e parâmetros devem identificar o seu tipo computacional através de um prefixo. O prefixo de tipo pode ter os seguintes valores:

s: String
n: Numérico
dt: Data, Hora
l: Long String
b: Booleana
o: Objeto

Regra 3: Todas as variáveis e parâmetros devem identificar o seu local de criação através de um prefixo. O prefixo de local de criação pode ter os seguintes valores:

A: Aplicação (global)

W: Janela

F: Função

C: Classe

Regra 4: Todas as variáveis e parâmetros devem identificar o seu tipo de utilização através de um prefixo. O prefixo de utilização pode ter os seguintes valores:

va: Variável

pa: Parâmetro por valor

pr: Parâmetro por referência

Exemplos:

fNomeCorrente: função

fAtribNome(Objeto oFpaDADO, Inteiro nFprTIPO): função

oFpaDADO: parâmetro do tipo objeto passado por valor.

nFprTIPO: parâmetro do tipo numérico passado por referência

bAvaCOND: Variável Global do tipo booleana

bFvaCOND: Variável do tipo booleana criada dentro de uma função.

bWvaCOND: Variável do tipo booleana criada dentro de uma janela.

Apêndice F – Medições Realizadas

Durante os estudos de casos foram realizadas as seguintes medições:

Métrica	**Esforço**
Descrição	Total de horas dedicadas para construção do gerador
Protocolo de medição	1 semana = 20 horas

Gerador	**Total de Horas**
ProtoBD versão Java, HTML, JavaScript	240
ProtoBD simplificado versão VisualBasic	60
ProtoBD simplificado versão SqlWindows	60
Gerador de Documentação em HTML	10
Gerador de arquivo de Rational Rose	2

Tabela F.1 Esforço em tempo

Métrica	Tamanho
Descrição	Total de linhas dos *frozen-spots* dos programas de descrição de artefato de um gerador. Total de linhas dos *hot-spots* dos programas de descrição de artefato de um gerador.
Protocolo de medição	1) Montar um único arquivo composto por todos os programas de descrição do artefato. 2) Montar um único arquivo composto por todos os programas de descrição do artefato. 3) Contabilizar o total de linhas/caracteres de frozen-spot do arquivo 4) Contabilizar o total de linha/caracteres de hot-spot do arquivo 5) A linha contendo a string Titulo "xxxx" é considerada uma linha de frozen-spot. O total de caracteres da linha de fronzen-spot é contabilizado. Para cada linha são abatidos os seis caracteres da string Titulo. 6) A linha cujo conteúdo seja diferente de Titulo "xxxx" e diferente de NaoAvLin é considerada uma linha de hot-spot .

	Total de Linhas		Total de Caracteres	
Gerador	**Frozen-spot**	**Hot-spot**	**Frozen-spot**	**Hot-spot**
ProtoBD versão Java, HTML, JavaScript	7901	5008	263432	97987
ProtoBD simplificado versão VisualBasic	1379	844	34298	14183
ProtoBD simplificado versão SqlWindows	2143	405	73678	8519
Gerador de Documentação em HTML	176	352	3277	6548
Gerador de arquivo de Rational Rose	23	45	427	1097

Tabela F.2 Total de Linhas de Frozen-spot e Hot-spot

Métrica	Complexidade
Descrição	Total de chamadas de funções de transformação nos programas de descrição de artefato de um gerador.
Protocolo de Medição	1) Montar um único arquivo composto por todos os programas de descrição do artefato. 2) Para cada chamada de função de transformação encontrada no arquivo, contabilizar o total de chamadas desta função no restante do arquivo.

Função	Chamadas
fNomeClasse	1295
fNomeID	83
fNomePONTA01ID	63
fNomePONTA02ID	63
fNomeDescr	62
fNomeREL	55
fTipoAtributo	51
fNomePONTAAUTO02ID	45
fTituloClasse	44
fNomePONTAAUTO01ID	43
fTEMRELN1	33
fGetRelEntidade	29
fTEMRELAUTO1N	29
fNomePONTAN1	28
fTEMRELAUTON1	26
fNomePONTA	23
fTEMRELAUTONN	20
fTEMRELNN	19
fParametrosAlteracao	18
fNomePONTAAUTO01DESCR	17
fNomePONTA01DESCR	16
fNomePONTA02DESCR	16
fNomePONTA01	14
fNomePONTA02	14
fNomePONTAAUTO01	14
fTipoID	12
fTEMGRAFICO	9
fTipoDescr	9
fNomePONTAAUTO02	8
fTEMREL1N	8
fNomePONTA1N	7
fNomePONTAAUTO02DESCR	7
fValidaCampo	6
fAtribTamCAMPOHTML	5
fParametrosInclusao	5
fGrafTITULO	4
fTituloREL	4

Função	Chamadas
FAtribTam	3
FAtribTitulo	3
FDadosCont	3
FIniInd	3
FSomaInd	3
FTagFimCampo	3
FTagIniCampo	3
FContador01Ini	2
FContador01Mais	2
FGetRelacionamento	2
FGrafSQL	2
fGrafTIPO	2
fNomePONTANN	2
fRELNNParametrosAlteracaoLINHA	2
fTituloPONTA	2
fGetLinks	1
fJAVAEscapeIncluir	1
fNomeDescrRELN1	1
fNomeIDPONTAN1	1
fParametrosAlteracaoLINHA	1
fParametrosRELNN	1
fSETTPOBJENT	1
fSETTPOBJRELAUTONN	1
fSETTPOBJRELNN	1
fTEMATRIBDATA	1
fTEMATRIBHORA	1
fTEMATRIBNUMERO	1
fTEMATRIBSTRING	1
fTEMATRIBTEXTO	1
fTEMJAVAEscapeIncluir	1
fTituloPONTA01	1
fTituloPONTA02	1
fTituloPONTAAUTO01	1
fTituloPONTAAUTO02	1
fTituloPONTAAUTO1N	1

Tabela F.4 ProtoBD completo versão Java, HTML, JavaScript

Função	Chamadas
fNomeClasse	130
fNomePONTA	39
fNomeREL	23
fContador02Get	20
fContador02Mais	20
fContador01Get	19
fContador01Mais	19
fTEMRELN1	12
fParametrosInclusao	6
fContador01Ini	5
fAtribTam	4
fGetRelEntidade	4
fNomeDescr	3
fNomeID	3
fTEMRELNN	3
fContador02Ini	2
fTamDescr	1

Tabela F.5 ProtoBD simplificado versão VisualBasic

Função	Chamadas
fNomeClasse	81
fNomeID	44
fContador01Get	18
fContador01Mais	9
fParametrosInclusao	9
fContador02Get	4
fContador02Mais	4
fGetRelEntidade	4
fContador01Ini	3
fValidaCampo	3
fAtribTam	2
fNomeDescr	2
fContador02Ini	1
fSetListaDados	1

Tabela F.6 ProtoBD simplificado versão SqlWindows

Função	Chamadas
fNomeClasse	43
fNomeRELPONTA1N	6
fNomeRELPONTAAUTO1	6
fNomeRELPONTAAUTON	6
fNomeRELPONTAAUTON	6
fNomeRELPONTAN1	6
fNomeRELPONTANN	6
fGetRelEntidade	4
fTEMRELACAUTO1N	3
fTEMRELAUTONN	3
fTEMRELNN	3
fTxtDescr	3
fTEMRELAC1N	2
fNomeARQPONTA1N	1
fNomeARQPONTAN1	1
fNomeARQPONTANN	1
fNomePONTA1N	1
fNomePONTAN1	1
fNomePONTANN	1
fTEMREL1N	1
fTEMRELN1	1

Tabela F.7 Gerador de Documentação em HTML

Função	Chamadas
fContador01Get	3
fContador01Mais	3
fNomeClasse	3
fNomeID	3
fGetRelEntidade	2
fNomeDescr	2
fContador01Ini	1
fSetListaDados	1
fTipoAtributo	1

Tabela F.8 Gerador de importação CASE Rational Rose

Referências Bibliográficas

[BG 1997] Batory, D.; Geraci, B.; "Validating Compositions and Subjectivity and GenVoca Generators"; *IEEE Transactions on Software Engineering 23(2);* 1997; pp 67-82

[Balzer 1989] Balzer, R.; "A Fifteen-Year Perspective on Automatic Programming"; In Ted J. Biggerstaff and Alan Perlis (Eds.); *Software Reusability*; Addison-Wesley/ACM Press; 1989; pp 289-311.

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

[Basset 1987] Basset, P.; "Frame-Based Software Engineering"; *IEEE Software 4(4);* 1987; pp 9-16

[Basset 1996] Basset, P.; *Framing Software Reuse;* Ed. Prentice-Hall; 1996

[Basset 1997] Basset, P.; "The Theory and Practice of Adaptive Reuse"; *Symposium on Software Reusability;* 1997; pp 2-9

[Batory 1996] Batory, D.; "Subjectivity and GenVoca Generators"; *Proceedings of 4º.International Conference on Software Reuse;* 1996; pp 166-175

[Batory 1998] Batory, D.; "Domain Analysis for GenVoca Generators"; *Proceedings of 5º.International Conference on Software Reuse;* 1998; pp 350-357

[Baxter 1992] Baxter, I.; "Design Maintenance Systems"; *Communications of the ACM*; 35(4); 1992; pp 73-89

[Baxter 1999] Baxter, I.; "Scaling for the Design Maintenance System"; *Proceedings of International Workshop on Software Transformation Systems;* 1999; pp 20-25

[BCRW 1998] Batory, D.; Chen, G.; Robertson, E.; Wang, T. "Design Wizards and Visual Programming Environments for Generators"; *Proceedings of 5º.International Conference on Software Reuse;* 1998; pp 255-267

[BFHLPRRB 1999] Buntine, W.; Fischer, B.; Havelund, K.; Lowry, M.; Pressburger, T.; Roach, S.; Robinson, P.; Baalen, J.; "Transformations Systemns at NASA Ames"; *Proceedings of International Workshop on Software Transformation Systems;* 1999; pp 8-13

[BGKLRZ 1997] 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

[Biggerstaff 1998] Biggerstaff, T.; "A Perspective of Generative Reuse"; *Annals of Software Engineering*; 5; 1998; pp 169-226

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

[BMJH 1996] Bruckhaus, T.; Madhavji, H.; Janssen, I.; Henshaw, J.;"Frame-Based Software Engineering"; *IEEE Software 13(5);* 1996; pp 29-38

[BMMB 1997] Bosch, J.; Molin, P.; Mattson, M.; Bengtson, P. "Object-Oriented Frameworks – Problems & Experiences"; http://www.ide.hk-r.se/~michaelm/papers/ex-frame.ps

[Boehm 1988] Boehm, B.W.; "A Spiral Model of Software Development and Enhancement"; *IEEE Computer 21(5);* 1988; pp 61-72

[BPB 1999] Barrére, T.; Prado, A.; Bonafé, V.; "CASE Orientada a Objetos com Múltiplas Visões de Requisitos e Implementação Automática de Sistemas"; *Anais do XIII Simpósio Brasileiro de Engenharia de Software;* Ed. SBC; 1999; pp 113-128

[BPL 1996] Bergmann, U.; Prado A.; Leite, J. C. S. P.; "Desenvolvimento de Sistemas Orientados a Objetos Utilizando o Sistema Transformacional Draco-PUC"; *Anais do X Simpósio Brasileiro de Engenharia de Software;* Ed. SBC; 1996; pp 173-188

[Broy 1997] Broy, M.; "Towards a Mathematical Concept of a Component and its Use"; *Software- Concepts and Tools*; 18; 1997; pp 137-148

[BS 1999] Batory, D.; Smaragdakis, Y.; "Application Generators"; *Wiley Encyclopedia of Electrical and Electronics Engineering – Software Engineering Volume;* Ed. John Wiley and Sons; 1999; http://www.cs.utexas. edu/users/schwartz

[BSL 1998] Batory, D.; Smaragdakis, Y.; Lofaso, B.; "JTS: Tools for Implementing Domain-Specific Languages"; *Proceedings of 5º. International Conference on Software Reuse;* 1998; pp 143-153

[Caroli 1999] Caroli, P.; *Uma Metodologia de Projeto de Software Orientado a Objetos*; Tese de Mestrado; Depto. Informática PUC-Rio; 1999

[CHSV 1997] 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

[CHW 1998] Coplien, J.; Hoffman, D.; Weiss, D.; "Commonality and Variability in Software Engineering"; *IEEE Software* 15(6); 1998; pp 37-45

[Cleaveland 1988] Cleaveland, C.; "Building Application Generators"; *IEEE Software* 5(4); 1988; pp 25-33

[Cleaveland 2000] Cleaveland, C.; *Program Generators using Java and XML*; Ed. Prentice-Hall; 2000 (em preparação) http://craigc.com/craigc/book.html

[CK 1988] Cleaveland, C.; Kintala, C.; “Tools for Building Application Generators”; *AT&T Technical Journal* 67(4); 1988; pp 46-58

[DCGMM 1999] Devanbu, P.; Chen, Y.; Gansner, E.; Müller, H.; Martin, J.; “CHIME: Customizable Hyperlink Insertion and Maintenance Engine for Software Engineering Environments”; In *Proceedings of the 21th International Conference on Software Engineering*; 1999; pp 473-482.

[DMNS 1997] Demeyer, S.; Meijler, T.; Nierstrasz, O.; Steyaert, P.; “Design Guidelines for 'Tailorable Frameworks' ”; *Communications of the ACM*; 40(10); 1997; pp 60-64

[ED 1997] Edwards, J.; DeVoe, D.; *3-Tier Client/Server At Work*; Ed. John Wiley and Sons; 1997

[FHLS 1997] Froehlich, G.; Hoover, H.; Liu, L.; Sorenson, P.; “Hooking into Object-Oriented Application Frameworks”; *Proceedings of International Conference on Software Engineering*; 1997; pp 491-501http://www.cs.ualberta.ca/~softeng/papers/papers.htm

[FHLS 1999] 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

[FL 1997] Freitas, F.; Leite, J. C. S. P.; “Aplicando reuso de software na construção de ferramentas de engenharia reversa”; *Anais do XI Simpósio Brasileiro de Engenharia de Software;* Ed. SBC; 1997; pp 265-280

[Floch 1995] Floch, J.; “Supporting Evolution and Maintenance by Using a Flexible Automatic Code Generator”; In *Proceedings of the 17th International Conference on Software Engineering*; 1995; pp 211-219.

[FM 1995] Fiadeiro, J.L.; Maibaum, T.; "Interconnecting Formalisms: Supporting Modularity, Reuse and Incrementality"; *ACM SIGSOFT Software Engineering Notes 20(4);* 1995; pp 72-80

[FMCM 1996] 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

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

[FS 1997] Fayad, M.; Schmidt, D.; "Object-Oriented Application Frameworks - Introduction"; *Communications of the ACM*; 40(10); 1997; pp 32-38

[FS 1998] Franca, L.P.A.; Staa A.; "Software Measurement for Small Organization"; *Proceedings of CAiSE98 Doctoral Consortium;* Pisa; 1998; pp xxx

[FSF 1999] 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

[FSL 1998] Franca, L.P.A.; Staa A.; Lucena, C.J.P.; "Medição de Software para Pequenas Empresas: Uma Solução Baseada na Web"; *Anais do XII Simpósio Brasileiro de Engenharia de Software;* Ed. SBC; 1998; pp 71-84

[FSR 1998] Fiorini, S.; Staa A.; Baptista, R.; *Engenharia de Software com CMM;* Ed. Brasport; 1998;

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

[GHW 1995] Gresse, C.; Hoisl, B.; Wüst, J.; "A Process Model fo GQM-Based Measurement"; *Software Technology Transfer Initiative Technical Report STTI-95-04-E*, University of Kaiserslautern, Department of Computer Science; 1995.

[GJM 1991] Ghezzi, C.; Jazayeri, M.; Mandrioli, D.; *Fundamentals of Software Engineering*; Ed. Prentice-Hall; 1991

[GM 1994] Grossman, I.; Mah, M.; "Independent Research Study of Software Reuse (Using Frame technology)"; *QSM Associates*; Sept. 1994; 75 pp.

[Guerrieri 1998] Guerrieri, E.; "Software Document Reuse with XML"; *Proceedings of 5º. International Conference on Software Reuse;* 1998; pp 246-254

[HBI 1997] Hester, A.M.; Borges, R.; Ierusalimschy, R.; "CGILua: A Multi-Paradigmatic Tool for Creating Dynamic WWW Pages"; *Anais do XI Simpósio Brasileiro de Engenharia de Software;* Ed. SBC; 1997; pp xxx

[Hohenstein 2000] 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.

[Humphrey 1989] Humphrey, W.S.; *Managing the Software Process;* Ed. Addison Wesley; 1989;

[Jenkins 1985] Jenkins, M.; "Surveying The Software Generator Market"; *Datamation*; Sept. 1985; pp 105-120

[Johnson 1992] Johnson, R.; "Documenting Frameworks Using Patterns"; *OOPSLA'92*; 1992; pp 63-76

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

[Kitchenham 1996] Kitchenham, B.; "Evaluating Software Engineering Methods and Tool"; *Software Engineering Notes 21(1,2,3,4,5);* 1996;

[KK 1999] Khriss, I.; Keller, R.; "Transformations for Pattern-Based Forward-Engineering"; *Proceedings of International Workshop on Software Transformation Systems;* 1999; pp 50-58

[KMcKBH 1996] Kieburtz, R.; McKinney, L.; Bell, J.; Hook, J.; "A Software Engineering Experiment in Software Component Generation"; In *Proceedings of the 18th International Conference on Software Engineering*; 1996; pp 542-552

[KPCLG 1996] Kirner, T.; Prado, A.; Costa, C.; Lima, M.; Gratão, R.; "Ambiente para Representação de Múltiplas Visões de Requisitos: O Metamodelo e uma Linguagem de Transformação"; *Anais do X Simpósio Brasileiro de Engenharia de Software;* Ed. SBC; 1997; pp 207-222

[LD 1998] Laitenberger, O.; Dreyer, H.; "Automated Software Engineering Data Collection Activities via the World Wide Web: A Tool Development Strategy applied in the Area of Software Inspection"; *ISERN Technical Report 98-12t*; 1998; http://www.iese.fhg.de/network/ISERN/pub/isern_biblio_tech.html

[Levy 1986] Levy, L.; "A Metaprogramming Method and Its Economic Justification"; *IEEE Transactions on Software Engineering 12(2);* 1986; pp 272-277

[LG 1989] 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

[LN 1995] Landin, N.; Niklasson, A.; *Development of Object-Oriented Frameworks*; Master Thesis; Department of Communication Systems; Lund University; Sweden; 1995; http://www.tts.lth.se/Personal/bjornr/Papers/OOFW.ps

[LPBC 1997] Lima, M.; Prado, A.; Barrére, T.; Couto, A.; "Ambiente CASE com Múltiplas Visões de Requisitos e Implementação Automática utilizando Draco"; *Anais do XI Simpósio Brasileiro de Engenharia de Software;* Ed. SBC; 1997; pp 65-79

[LSF 1994] Leite, J. C. S. P.; Sant'Anna, M.; Freitas, F; "Draco-PUC: a Technology Assembly for Domain Oriented Software Development"; *Proceedings of 2o.International Conference on Software Reuse*; 1994; pp 94-100

http://www.sei.cmu.edu/activities/cbs/icse98/papers/

[LTV 1996] Lindén, G.; Tirri, H.; Verkamo, I.; "ALCHEMIST: A General Purpose Transformation Generator"; *Software - Practice and Experience 26(6);* 1996; pp 653-675

[Milicev 2000] 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.

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

[Moreira 1998] Moreira, A.; "Parametrização de Componentes de Especificação com Preservação de Semântica"; *Anais do XII Simpósio Brasileiro de Engenharia de Software;* Ed. SBC; 1998; pp 155-170

[MQ 1994] Moriconi, M.; Qian, X.; "Correctness and Composition of Software Architectures"; *ACM SIGSOFT Software Engineering Notes*; 19(5); 1994; pp 164-174

[MQR 1995] Moriconi, M.; Qian, X.; Riemenschneider, R.; "Correct Architecture Refinement"; *IEEE Transactions on Software Engineering*; 21(4); 1995; pp 356-372

[MR 1997] Medvidovic, N.; Rosenblum, S.; “Domains of Concern in Software Architectures and Architecture Description Languages”; *In Proceedings of the USENIX Conference on Domain-Specific Languages*; 1997. http://www.ics.uci.edu/~dsr/Selected.html

[MRT 1999] Medvidovic, N.; Rosenblum, S.; Taylor, R.; “A Language and Environment for Architecture-Based Software Development and Evolution”; *In Proceedings 21th International Conference on Software Engineering*; 1999. http://www.ics.uci.edu/~dsr/Selected.html

[Neighbors 1989] Neighbors, J. M.; “Draco: A Method for Engineering Reusable Software Systems”; In Ted J. Biggerstaff and Alan Perlis (Eds.); *Software Reusability*; Addison-Wesley/ACM Press; 1989; pp 295-319

[Perry 1987] Perry, D.; “Software Interconnection Models”; In *Proceedings of the 9th International Conference on Software Engineering*; 1987; pp 61-71

[Perry 1989a] Perry, D.; “The Inscape Environment”; In *Proceedings of the 10th International Conference on Software Engineering*; 1987; pp 2-12

[Perry 1989b] Perry, D.; “The Logic of Propagation in The Inscape Environment”; In *Proceedings of SIGSOFT '89: Testing, Analysis and Verification Symposium*; 1989; pp 114-121

[PM 1987] Prieto-Diaz, R.; Neighbors, J. M.; ”Module Interconnection Languages”; *J. Systems Software 6;* 1987; pp 307-334

[Pfleeger 1998] Pfleeger, S.L.; *Software Engineering Theory and Practice*; Prentice-Hall; NJ; 1998

[PR 1999] Plantec, A.; Ribaud, V.; "Using and Re-using Application Genetarors"; *In Proceedings of the First International Symposium on Constructing Software Engineering Tools*; 1999; pp 59-65.

[Prado 1992] Prado, A.; *Estratégia de Re-Engenharia de Software Orientada a Domínios*; Tese de Doutorado; Depto. Informática PUC-Rio; 1992

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

[Pree 1999] 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.

[Prieto-Diaz 1991] Prieto-Diaz, R.; "Implementing Faceted Classification for Software Reuse"; *Communications of the ACM*; 34(5); 1991; pp 88-97

[PWCC 95] Paulk, M. C.; Weber, C.V.; Curtis, W.; Chrissis, M.B.; *The Capability Maturity Model – Guidelines for Improving the Software Process*; Ed. Addison Wesley; 1995

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

[RJ 1996] 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

[Sant'Anna 1999] Sant'Anna, M.; *Circuitos Transformacionais*; Tese de Doutorado; Depto. Informática PUC-Rio; 1999

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

[Schmid 1997] Schmid, H.; "Systematic Framework Design by Generalization"; *Communications of the ACM*; 40(10); 1997; pp 48-51

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

[Sindelar 1990] Cleaveland, C.; "Specification-Driven Tool Technology"; *Proceedings of Sun User Group Conference*; 1990; pp 209-219

[SJ 1994] Srinivas, Y.; Jüllig, R.; "Formal Support for Composing Software"; *Kestrel Institute Technical Report 94.5*; 1994

[SKC 1993] Setliff, D.; Kant, E.; Cain, T.; "Practical Software Synthesis"; *IEEE Software 10(5)*; 1993; pp 6-10

[SL 1999] Sant'Anna, M.; Leite, J. C. S. P.; "An Architectural Framework for Software Transformation"; *Proceedings of International Workshop on Software Transformation Systems;* 1999; pp 33-38

[SLB 1998] Shull, F.; Lanubile, F.; Basili, V.; "Investigating Reading Techniques for Framework Learning"; *ISERN Technical Report 98-16*; 1998; http://www.iese.fhg.de/network/ISERN/pub/isern_biblio_tech.html

[SLP 1998] Sant'Anna, M.; Leite, J. C. S. P.; Prado, A.; "A Generative Approach to Componentware"; *Proceedings of International Workshop on Component Based Software Engineering*; 1998; http://www.sei.cmu.edu/activities/cbs/icse98/papers/

[SMcD 1996] Srinivas, Y.; McDonald, J.; "The Architecture of SPECWARE, a Formal Software Development System"; *Kestrel Institute Technical Report 96.7*; 1996

[Smith1990] Smith, D.; “KIDS: A Semi-Automatic Program Development System”; *IEEE Transactions on Software Engineering*; 16(9); 1990; pp 1024-1043

[Staa 1993] Staa, A.; Manual de Referência: Talisman: Ambiente de Engenharia de Software Assistido por Computador; Rio de Janeiro; 1993

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

[SWJ 1998] Schneider, G.; Winters, J.; Jacobson, I.; *Applying Use Cases : A Practical Guide*; Ed. Addison-Wesley; 1998.

[Winter 1999] Winter, V.; “Program Transformation in HATS”; *Proceedings of International Workshop on Software Transformation Systems;* 1999; pp 26-32

[ZSGS 1993] Zand, M.; Saiedian, H.; George, K.; Samadzadeh, M.; ”An Interconnection Language for Reuse at the Template/Module Level”; *J. Systems Software 23;* 1993; pp 9-25