2
Trabalhos Relacionados

Os trabalhos relacionados apresentados neste capítulo, estão associados ao processo proposto nesta dissertação devido à utilização da abordagem MDA para efetuar transformações em um modelo PIM, com o intuito de gerar modelos PSM. O foco está na maneira como cada trabalho efetua essa transformação.

O único trabalho que não atua nesta área é o Appfuse, que está relacionado diretamente à geração de um sistema por completo, gerando diversas etapas do desenvolvimento, mas sem se preocupar na geração de modelos.

2.1.
Odyssey-MDA

A ferramenta Odyssey-MDA [Odyssey-MDA05] é baseada em um framework onde é possível o desenvolvedor especificar e executar transformações sobre modelos UML. O enfoque desta ferramenta está na transformação de modelos independentes de plataforma (PIM – Platform Independent Model) em modelos específicos para uma plataforma (PSM – Platform Specific Model).

Conforme é apresentado na Figura 3, o cenário típico de utilização da ferramenta considera um usuário desenvolvendo o seu modelo UML em uma ferramenta CASE e exportando o mesmo no formato XMI para ser importado pela ferramenta.

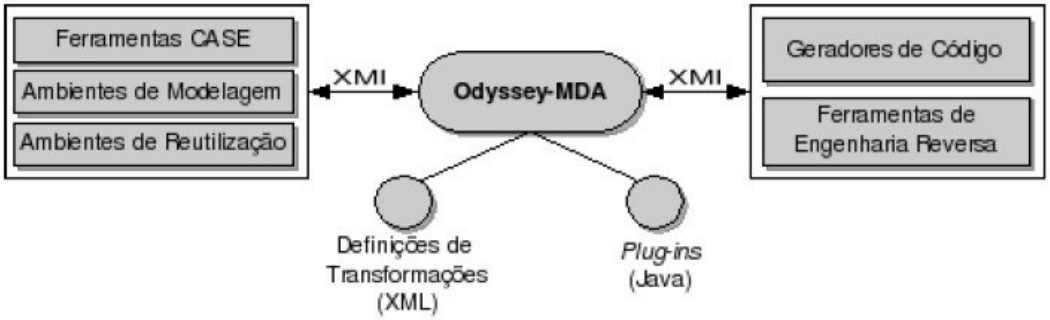

Figura 3 - Cenário de utilização da Odyssey-MDA [Odyssey-MDA05]

As transformações disponíveis podem ser executadas sobre o modelo, resultando em um PSM de saída, que pode ser exportado no formato XMI para posterior importação em uma ferramenta CASE.

É possível também utilizar uma ferramenta de engenharia reversa para gerar, a partir de um código-fonte existente, um modelo UML, e posteriormente exportar sua representação no padrão XMI, para então ser utilizada na Odyssey-MDA.

Cada transformação na Odyssey-MDA é formada por uma especificação declarativa e um conjunto de mecanismos. A especificação declarativa é responsável por definir os mapeamentos entre os elementos dos modelos de entrada e saída. Tais mapeamentos são definidos através de critérios de busca para a seleção dos elementos a serem transformados e pela atribuição de um mecanismo responsável pela realização da transformação dos elementos.

Os mapeamentos são específicos para cada tipo de elemento a ser mapeado. Para mapear subtipos de Classifier no meta-modelo da UML (ex: classes, interfaces), é proposto o mapeamento classifier-map. Para mapear subtipos de Features (ex: atributos e operações), é proposto o sub-mapeamento feature-map. Para mapear um Classifier para Feature, ou o contrário, é proposto o mesmo sub-mapeamento classifier-featuremap. A figura 4 mostra um exemplo de um arquivo XML utilizado para armazenar as regras de transformações.

```
1  <transformation-mapping name="Simple EJB Transformation">
2    <classifier-map name="Entity to EntityBean" type="ClassClass">
3      <finder direction="left" type="stereotype" value="Entity"/>
4      <finder direction="right" type="stereotype" value="EJBEntityBean"/>
5      <property name="stereotype" direction="forward" value="EJBEntityBean"/>
6      <property name="nameTransformation" direction="forward" value="#CLASSIFIER_NAME#Bean"/>
7      <property name="stereotype" direction="reverse" value="Entity"/>
8      <property name="nameTransformation" direction="reverse">
9        <property name="input" value="#CLASSIFIER_NAME#"/>
10       <property name="regex" value="(.*)Bean$"/>
11       <property name="subst" value="$1"/>
12     </property>
13     <feature-map name="Copy attributes: Entity - EntityBean" type="AttributeAttribute"> ... </feature-map>
14     <feature-map name="EntityBean getter operation" type="AttributeOperation"> ... </feature-map>
15     <feature-map name="EntityBean setter method" type="AttributeOperation"> ... </feature-map>
16     <classifier-feature-map name="EntityBean ejbCreate method" type="ClassOperation"> ...
17     </classifier-feature-map>
18   </classifier-map>
19   ...
20 </transformation-mapping>
```

Figura 4 - Exemplo de um arquivo XML contendo as regras de transformações [Odyssey-MDA05]

A Odyssey-MDA provê uma infra-estrutura de mecanismos genéricos, denominados built-ins, que realizam transformações simples sobre elementos UML, como gerar uma classe a partir de uma classe existente, gerar uma operação a partir de um atributo, entre outras transformações. Como exemplos desses mecanismos, podemos citar ClassClass, ClassInterface e AttributeOperation, conforme ilustrados na Figura 5.

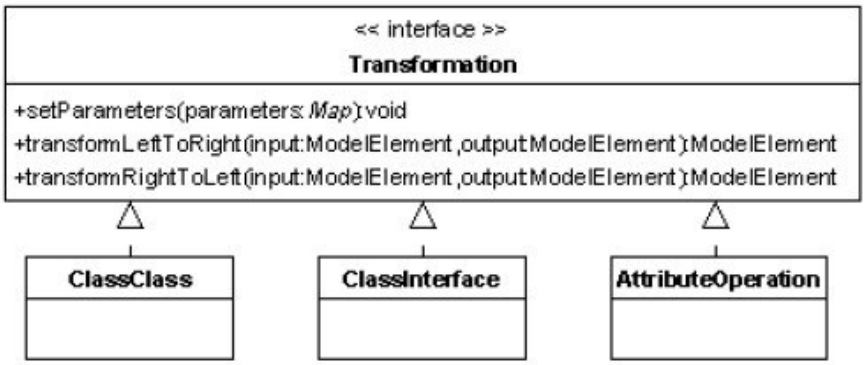

Figura 5 - Mecanismos genéricos de transformação (built-ins) [Odyssey-MDA05]

Os built-ins disponíveis no framework são suficientes para a definição de qualquer transformação entre os tipos suportados. Caso o usuário necessite de alguma transformação não disponível no framework, ele pode desenvolvê-la criando um mecanismo (plug-in) que implementa a interface Transformation e manipula os elementos através das interfaces JMI.

2.2. UML Model Transformation Tool (UMT)

UML Model Transformation Tool (UMT) [UMT04] é uma ferramenta de suporte à transformações de modelos e geração de código, baseados em modelos UML. Ela utiliza um formato intermediário, chamado de XMI Light, para efetuar suas transformações.

Os modelos devem ser importados pela ferramenta para que a mesma possa efetuar as transformações. Caso o modelo esteja no formato XMI padrão, ele deverá ser transformado para o XMI Light, antes das transformações ocorrerem. A partir daí um novo modelo é gerado, podendo ser exportado no formato XMI padrão novamente.

A figura 6 ilustra a arquitetura da ferramenta UMT durante a fase de importação dos modelos independentes de plataforma. Onde é possível visualizar os componentes que compõem a UMT, como sendo elementos independentes de plataforma.

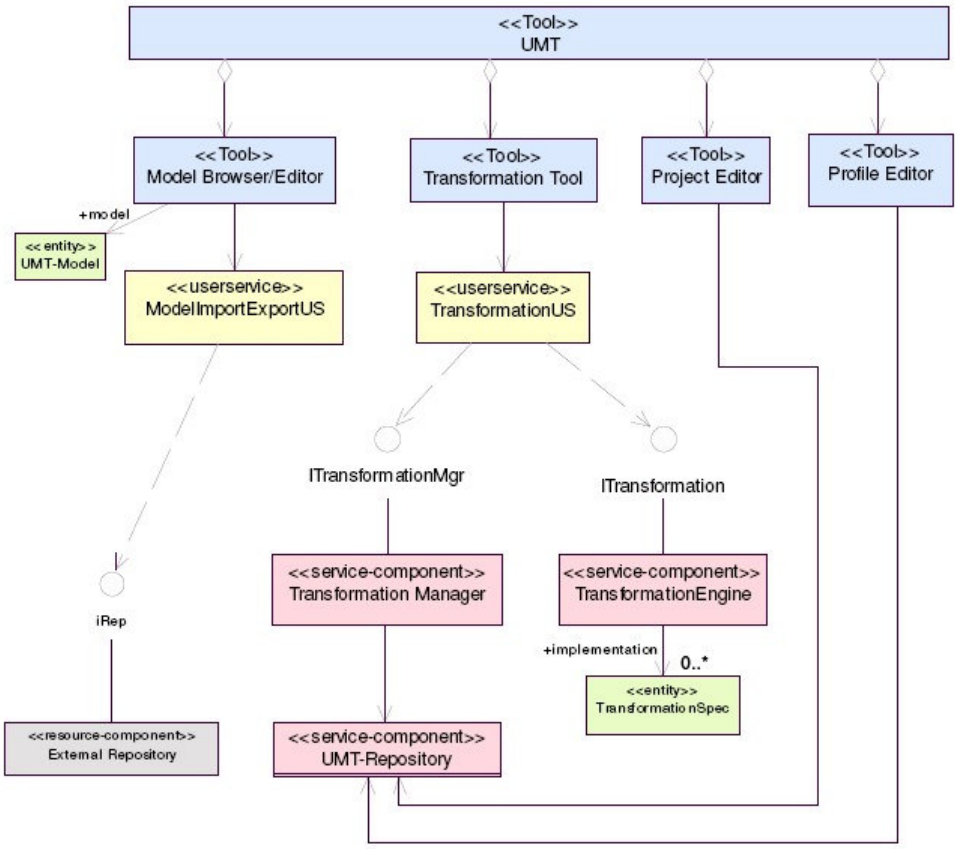

Figura 6 - Arquitetura da Ferramenta UMT (Independente de Plataforma) [UMT]

Já a figura 7 mostra a arquitetura da ferramenta UMT durante a fase de geração dos modelos específicos para uma plataforma, onde os diversos componentes estão definidos para uma plataforma específica. Por exemplo, o esteriótipo <<java>> significa uma implementação java, o <<xml>> um arquivo XML, <<xslt>> um arquivo XSLT e <<javaclass>> uma classe java compilada.

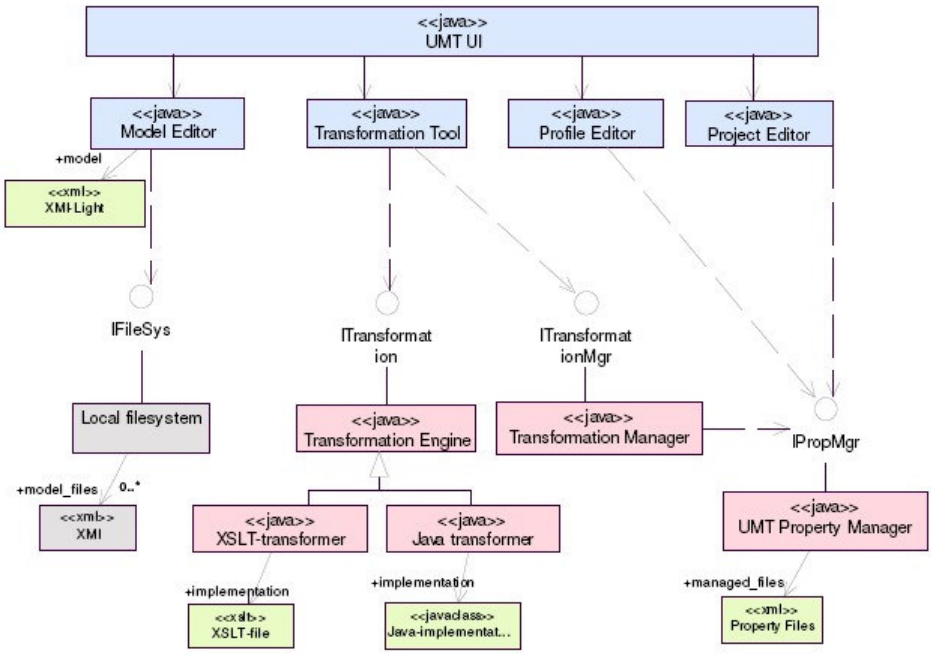

Figura 7 - Arquitetura da Ferramenta UMT (Específica para uma Plataforma) [UMT]

O transformador XSLT da UMT é um XSLT Stylesheet [XSLT], que produz transformações baseadas no padrão XMI Light de modelos importados pela ferramenta. Atualmente, existem dois tipos de transformadores XSLT: o de um único arquivo e o de múltiplos arquivos. A figura 8 abaixo mostra um exemplo de um arquivo contendo regras de transformação.

```
<!--
***     template match class
-->
<xsl:template match="class">
  <file>
    <xsl:attribute name="type">javaclass</xsl:attribute>
    <xsl:attribute name="filename"><xsl:value-of select="@name"/>
    </xsl:attribute>
    <xsl:attribute name="location">interfaces/<xsl:value-of select="../@name"/>
    </xsl:attribute>
    <xsl:attribute name="stereotype">interface</xsl:attribute>
    <xsl:attribute name="extension">java</xsl:attribute>
```

Figura 8 - Exemplo de um arquivo XSLT contendo as regras de transformação [UMT]

O primeiro produz somente um arquivo de saída a partir do modelo de entrada, como, por exemplo, um arquivo WSDL [WSDL] ou SQL. Já o segundo pode produzir diversos arquivos, sendo que para isso é necessário que o arquivo de entrada seja pré-processado pela UMT Java Transformer Class e posteriormente processado por uma das classes que implementam a classe 'transformer.DefaultMultiFileTransformer'.

2.3. AppFuse

AppFuse [AppFuse] é uma aplicação desenvolvida para ajudar na criação de projetos Web Java. Sua principal característica é criar e configurar todo um novo projeto incluindo no mesmo, por exemplo, frameworks de apresentação, negócio e persistência, um banco de dados e um servidor de aplicação.

Com isso, toda a parte de configuração e adaptação das tecnologias fica sob a responsabilidade da ferramenta, o usuário só precisa adicionar informações simples para que a configuração possa ser concluída.

Um exemplo seria uma aplicação Web Java que utilize os frameworks Struts [Struts], Spring [Spring] e Hibernate, o banco de dados MySQL [MySQL] e o servidor de aplicação Tomcat [Apache]. Ela irá gerar um projeto para essas tecnologias e irá configurá-los em conjunto, gerando arquivos de configuração e classes voltados para as tecnologias em questão. Por último, irá fazer o deployment no Tomcat.

A figura 9 mostra um exemplo de configuração para a geração de uma aplicação. A AppFuse se encarrega de gerar os arquivos necessários e testar o deployment no Tomcat, por exemplo.

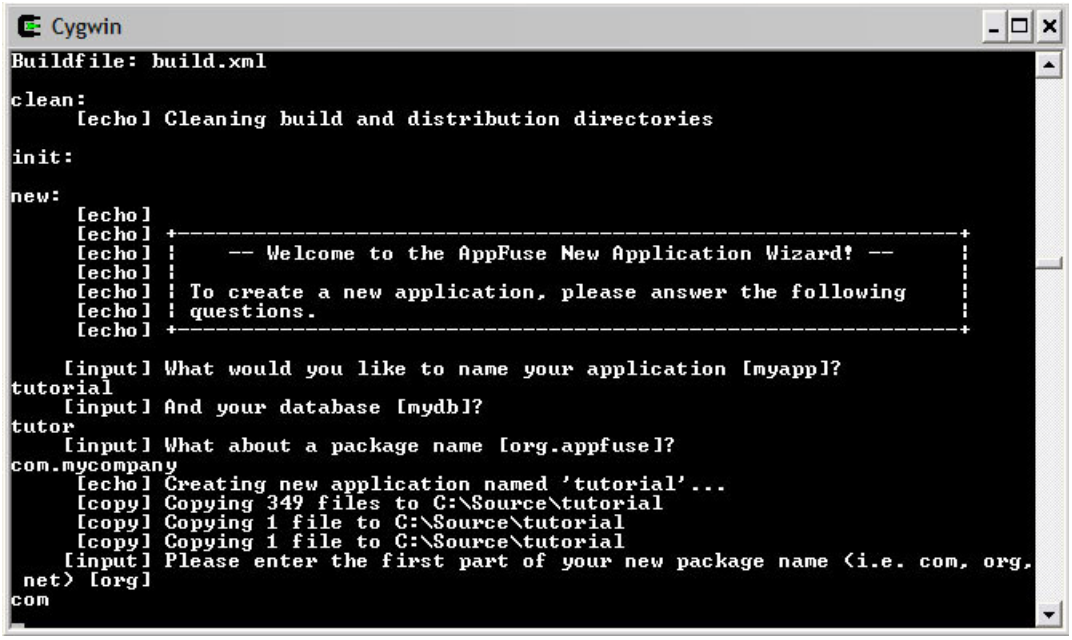

Figura 9 - Exemplo de configuração do Appfuse, para a geração de uma aplicação [AppFuse]

A limitação do AppFuse está em ser uma ferramenta para aplicações Web que utilizem a plataforma Java, e seu suporte às tecnologias ainda é bem limitado, com isso ela é voltada para casos bem específicos onde a necessidade de geração se encaixa com as tecnologias suportadas pela mesma.

Um ponto importante que a AppFuse não atua é a parte de documentação e modelagem das informações geradas. Toda essa parte fica a cargo do desenvolvedor, sendo necessário fazer uma engenharia reversa de tudo que foi gerado.

Esse tipo de ferramenta se encaixa na etapa posterior a geração do PSM, onde as opções relacionadas a tecnologias (persistência, interface e negócio), banco de dados e servidor de aplicação já estão definidos. A partir daí fica a cargo da ferramenta gerar em conjunto, código para todas essas tecnologias.

2.4. Model in Action (Mia)

Model in Action [MIA] é uma ferramenta para transformações de modelos para modelos. Seu grande diferencial está na forma de construir as regras de transformações, sendo possível fazê-las da forma tradicional (incluindo as regras no arquivo diretamente, utilizando a linguagem específica para as regras de transformações) ou utilizando uma interface gráfica oferecida pela mesma (não sendo necessário conhecer a linguagem que descreve as regras de transformações). A figura 10 ilustra de um modo geral como a ferramenta Model in Action atua.

Figura 10 - Visão Geral da Ferramenta Model in Action [MIA]

A MIA trabalha com três tipos de linguagens para armazenar as regras de transformações: para as mais simples e que necessitam de funções básicas, a RL-TL é utilizada, para as que necessitam de um pouco mais de complexidade, a MIA-TL oferece uma sintaxe mais rebuscada, e para as regras muito complexas é possível desenvolver regras de transformações na linguagem Java.

Ela se divide em cenários, regras e serviços. Os cenários determinam o fluxo das transformações, sendo os mesmos responsáveis pela ordem de execução, por exemplo, da criação de novas classes, novos métodos, novos relacionamentos, entre outras coisas relativas ao modelo. Diversos cenários podem ser definidos para uma única aplicação. Eles podem executar tanto regras quanto serviços. A figura 11 mostra um exemplo de criação de um serviço para a criação de um método, utilizando a parte gráfica da ferramenta MIA.

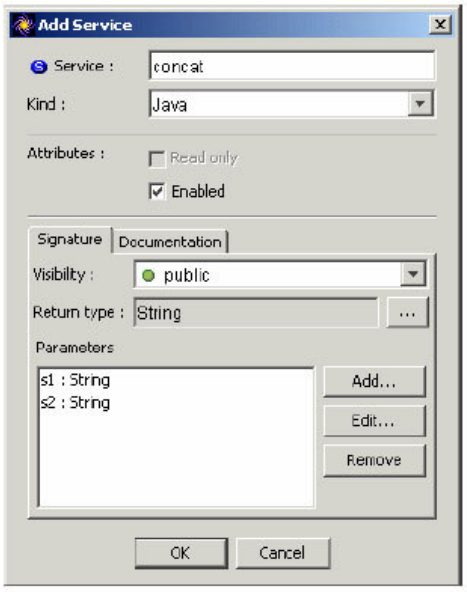

Figura 11 - Exemplo de criação de um serviço, utilizando a parte gráfica da ferramenta Model in Action [MIA]

A figura 12 mostra um exemplo de criação de regras de transformação sem utilizar a parte gráfica da ferramenta MIA.

```
operation = createInstance(uml, "Operation");
addLink("name", operation, name);
addLink("visibility", operation, visi
addLink("owner", operation, owner);
```


Figura 12 - Exemplo da criação de regras de transformação, sem utilizar a parte gráfica da ferramenta MIA [MIA]

Cada regra é responsável por uma única transformação em um modelo, podendo ser desde a criação de métodos, classes e atributos, até a adição de visibilidade em um relacionamento existente. Quando uma transformação complexa é necessária, diversas regras são criadas, deixando a cargo dos cenários determinarem à ordem de execução das mesmas.

Já os serviços, determinam que transformações poderão ser utilizadas e o que deverá ser retornado após a execução de cada uma. É possível retornar desde objetos comuns, até modelos e arquivos específicos em Java. A figura 13 mostra a arquitetura da ferramenta.

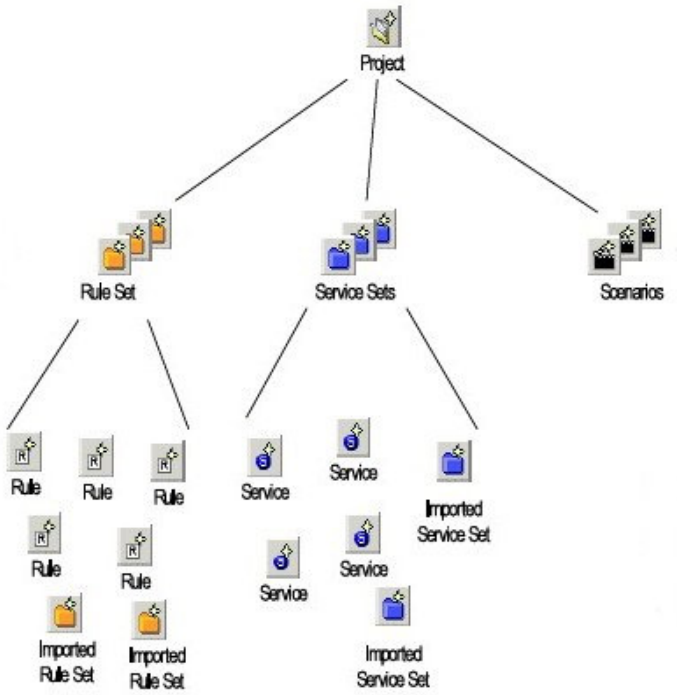

Figura 13 - Arquitetura da Ferramenta Model in Action [MIA]

2.5. Metastorage

Metastorage [Metastorage05] é uma aplicação voltada para a geração e adaptação de camadas de persistência, sendo a mesma baseada no módulo de persistência da máquina de compilação MetaL. Seu principal foco está na geração de componentes de software para a camada de persistência, descritos na forma de um arquivo XML. Ambos foram desenvolvidos utilizando a linguagem de scripts PHP.

Ela reduz o esforço necessário para implementar a camada de persistência, sem que haja perda de flexibilidade da mesma. A geração é feita utilizando as melhores estratégias, com o intuito de gerar sempre um código otimizado.

MetaL é uma máquina de compilação para meta-programming. Esse método de programação consiste no desenvolvimento de aplicações utilizando linguagens de alto nível, que posteriormente serão traduzidas em uma ou mais linguagens de programação.

O formato utilizado para descrever os componentes é baseado no XML. O objetivo é utilizar um formato simples de ser escrito, mesmo por aqueles que não estão familiarizados com linguagens de programação específicas, necessitando apenas que os mesmos conheçam a estrutura e a forma de se trabalhar com arquivos XMLs.

A figura 14 mostra um exemplo de criação de uma classe, seus atributos e métodos utilizando o padrão XML. Esse arquivo será utilizado posteriormente para gerar as classes em diversas linguagens, como Java, PHP e Perl, entre outros.

```
<class>
  <id>1</id>
  <name>article</name>

  <variable>
    <name>title</name>
    <type>text</type>
    <initialvalue>[no title]</initialvalue>
    <length>64</length>
  </variable>

  <variable>
    <name>lead</name>
    <type>text</type>
    <multiline>1</multiline>
  </variable>

  <variable>
    <name>body</name>
    <type>text</type>
    <multiline>1</multiline>
  </variable>
  <function>
    <name>getauthor</name>
    <type>getreference</type>
    <parameters>
      <variable>author</variable>
    </parameters>
  </function>

  <function>
    <name>setauthor</name>
    <type>setreference</type>
    <parameters>
      <variable>author</variable>
      <reference>
        <argument>author</argument>
      </reference>
    </parameters>
  </function>
</class>
```

Figura 14 - Exemplo de criação de uma classe e seus atributos e métodos utilizando XML [Metastorage]

A figura 15 mostra um exemplo de criação de um report, que é como a parte relacionada ao banco é descrita, podendo ser uma consulta seguida pelo preenchimento de objetos com os dados originários do banco. O type getallreportdata indica que essa função irá buscar os dados no banco e preenchê-los nos objetos.

```
<function>

  <name>getlatestarticles</name>

  <type>getallreportdata</type>

  <argument>
    <name>startdate</name>
    <type>timestamp</type>
  </argument>

  <parameters>
    <query>latestarticles</query>
    <data>
      <argument>articles</argument>
    </data>
    <first>
      <integer>0</integer>
    </first>
    <limit>
      <integer>10</integer>
    </limit>
    <count>
      <argument>total</argument>
    </count>
  </parameters>

</function>
```

Figura 15 - Exemplo da criação de um report [Metastorage]

O padrão XML do Metastorage segue as restrições do formato Simplified XML, que é um formato simplificado que não contêm atributos. Como os atributos não podem conter várias tags dentro de si mesmo, não há necessidade de utilizá-los.

O Metastorage também gera diagramas de classes da UML quando necessário, sendo o mesmo é gerado no formato DOT [Graphviz], que é um formato para gráficos que podem ser visualizados através do programa Graphviz [Graphviz]. É possível também gerá-los na forma de figuras (GIF e JPEG, por exemplo) e outros formatos como Postscript e PDF.

Apesar desta ferramenta não atuar diretamente sobre modelos, ou seja, não transformar modelos PIM em modelos PSM, ela também trabalha na adaptação e geração de camadas de persistência para aplicações novas ou já existentes. Ela foca na construção de arquivos XML contendo os componentes a serem gerados para diversos tipos de camadas de persistência, e para diversas linguagens de programação.