HOML: Hierarchical Object Mapping Language

Luis Henrique da Hora Nascimento (Universidade Salvador, Bahia, Brasil) – henrique_ssa@hotmail.com

Jorge Alberto Prado de Campos (Universidade Salvador, Bahia, Brasil) – jorge@unifacs.br

Abstract. *The semantic gap between data storage engines and object-oriented languages is a recurring problem that programmers have to face at every new project. Many of the proposed solutions perform the mapping between the entities in memory and database through programming features and rigidly defined rules, which makes such solutions non-extensible and linked to specific mechanisms. This paper introduces HOML (Hierarchical Object Mapping Language) as a resource for making the program´s code independent from data sources. HOML allows the creation of flexible and extensible mappings decoupled from any storage mechanisms or programming languages. HOML aims at providing transparent access to data by removing all complexity to access the data sources from source code while ensuring full ability of expression to programmers.*

Key Word: *object-relational mapping, database abstraction, query methods.*

Resumo. A separação semântica entre os mecanismos de armazenamento de dados e as linguagens orientadas a objetos é um problema recorrente com o qual os programadores precisam lidar a cada novo projeto. Muitas das soluções propostas realizam o mapeamento entre as entidades em memória e do banco de dados através de recursos de programação e regras rigidamente definidas, o que torna as soluções não extensíveis e vinculadas a mecanismos específicos. Este artigo apresenta a HOML (Hierarchical Object Mapping Language) como um recurso para promover a independência de fontes de dados, permitindo a criação de mapeamentos flexíveis, extensíveis e desvinculado de quaisquer mecanismos de armazenamento ou linguagens de programação. A HOML busca promover o acesso transparente aos dados ao retirar do código a complexidade da integração entre o programa e as fontes de dados ao tempo em que assegura programador a sua capacidade de expressão.

Palavras Chave: Mapeamento objeto-relacional, abstração de base de dados, métodos de consulta.

1. Introdução

O aumento da complexidade dos programas de computador e o crescente volume de informação a ser armazenado e manipulado transformaram a persistência dos dados em uma das questões mais críticas no ciclo de desenvolvimento dos sistemas computacionais [Reese, G. 1997]. De forma a endereçar essa crescente complexidade, a infraestrutura de programação e os mecanismos de armazenamento persistentes seguiram caminhos independentes, dando origem a uma separação semântica [Holder et al. 2008] entre as tecnologias dominantes de cada área. De um lado surgiram poderosas plataformas de desenvolvimento para linguagens de alto nível baseadas na orientação a objetos (OO), do outro lado são notáveis os avanços dos SGBDs baseados no modelo relacional [Codd, EF 1990]. Interligando os dois ambientes, a Structured Query Language (SQL) tornou-se o padrão adotado quase que universalmente [Soukup 1998]. Outras tecnologias têm surgido como alternativas viáveis, porém ainda não possuem a mesma penetração de mercado das tecnologias citadas.

A maioria do SGBDs modernos se baseia no modelo relacional e adota o padrão SQL. A despeito dessa base comum, as diferenças entre eles são enormes. Muitos fabricantes adotam soluções particulares para diversos recursos definidos nos padrões que regem a SQL. Além disso, cada fabricante também cria seus próprios recursos para suprir especificidades não contempladas pelos padrões. Isso torna extremamente difícil a decisão de abandonar um SGBD e adotar um novo, pois o código responsável pelo acesso aos dados acaba incluindo particularidades que geram forte acoplamento entre o programa e o SGBD adotado.

As modernas linguagens OO, por outro lado, permitem utilizar estruturas complexas que precisam ser traduzidas em estruturas compatíveis com o SGBD para viabilizar a persistência. Esta tradução é frequentemente realizada através de rotinas de programação, utilizando sentenças SQL. Neste cenário, a maioria dos programas trata o problema da separação semântica através do mapeamento objeto-relacional [Vidal 2005; Bauer e King 2005; Holder et al. 2008]. Grande parte das soluções adotadas costuma ignorar possíveis substituições do SGBD, outras deixam a solução deste problema como decisão de projeto, de modo que muitas peças especializadas de código precisam ser escritas para contornar essa deficiência.

Apesar dos avanços na busca para minimizar ou mesmo eliminar a separação semântica entre os programas e os SGBDs, não foi encontrada na literatura nenhuma solução que alcançasse a abstração completa das fontes dados no desenvolvimento de softwares. Este artigo apresenta uma contribuição neste sentido, buscando alcançar uma tecnologia que seja verdadeiramente WOPA (write once, persist anywhere).

Com o objetivo de implementar da tecnologia WOPA, foi concebido o Modelo de Desenvolvimento Orientado a Dados (MDOD), que visa promover o isolamento completo e transparente entre o código do aplicativo e os mecanismos de armazenamento, recuperação e controle de dados. Deste modo, retira-se do código o mapeamento entre o programa e o banco de dados. Para isso, utiliza-se uma abordagem orientada a objetos para dividir o programa em camadas que encapsulam estruturas gradualmente especializadas para atingir a completa abstração dos mecanismos de armazenamento de dados. O objetivo do MDOD é ajudar os desenvolvedores a superar a separação semântica entre os programas e as fontes de dados (SGBDs e outros mecanismos), fornecendo orientações para a construção de um software capaz de lidar com esta questão de forma transparente, flexível e portável. Dentre os recursos

apresentados pelo MDOD, um dos mais importantes é a Hierarchical Object Mapping Language (HOML). Seu propósito é permitir a definição de mapeamentos e expressões de consulta para a execução de todas as operações necessárias sobre as fontes de dados de forma abstrata e independente do mecanismo utilizado. Deste modo, a estrutura da HOML visa proporcionar flexibilidade, extensibilidade e portabilidade, sem prejuízo à capacidade de expressão necessária para que o programador possa tirar o máximo proveito dos recursos oferecidos por cada mecanismo.

O restante deste trabalho está organizado da seguinte forma: a seção 2 apresenta alguns trabalhos relacionados e discute suas abordagens. A seção 3 aborda brevemente o MDOD, apresenta a arquitetura que dá suporte à HOML. A seção 4 apresenta a estrutura da HOML e discute suas características. A seção 5 apresenta as considerações finais e conclusões.

2. Trabalhos Relacionados

A separação semântica entre dados e código motivou a criação de diversas soluções para endereçar o problema. Estas soluções possuem abordagens variadas, o que torna impossível elencar todas em um curto espaço. O foco deste artigo é a abstração de dados com ênfase em linguagens orientadas a objetos. Por este motivo, o escopo dos trabalhos relacionados foi limitado somente aos trabalhos que buscam o mesmo objetivo e que possuem notório destaque.

A HOML foi criada como resultado da análise dos problemas recorrentes no processo de desenvolvimento de softwares orientados a objetos e das soluções mais utilizadas para tratar estes problemas. A maior dificuldade neste processo decorre da diferença entre a forma como os dados são representados em memória e a forma como eles são armazenados, resultando na falta de transparência entre estes dois elementos (separação semântica).

A SQL foi uma das primeiras tentativas bem sucedidas para lidar com o problema da separação semântica, por volta da década de 70 [Date 2003]. Esta linguagem possui como vantagem o fato das operações sobre os dados serem especificadas de forma não procedural. Isso permite que estas operações sejam executadas através de instruções simples e de fácil compreensão, dispensando a execução de rotinas complicadas e a manipulação direta de estruturas de armazenamento complexas.

As linguagens orientadas a objetos trouxeram um complicador adicional, elevando o nível de abstração utilizado no desenvolvimento de software e requerendo a criação de mecanismos de mapeamento objeto-relacional (MOR). Muitas soluções independentes foram criadas para tratar este problema, cada uma delas com suas particularidades, pontos fortes e pontos fracos.

As soluções mais representativas de MOR baseiam-se em duas abordagens principais: o padrão Expert e o padrão Database Broker [Molz 1990]. A primeira abordagem assume que cada entidade deve ser responsável pela sua própria persistência, provendo os métodos necessários para leitura e armazenamento no repositório de dados, tendo como exemplo o Active Records [Fowler, 2002]. A segunda abordagem assume que o objeto não deve manter qualquer informação sobre a sua persistência, limitando-se a modelar as entidades do domínio de aplicação, delegando as operações de persistência a entidades especializadas. Foge ao escopo deste trabalho exaurir cada uma das soluções existentes, por isso serão abordados somente as três mais utilizadas, que juntas representam mais de 80% do mercado: A

Language-INtegrated Query (LINQ) , o Entity Framework e o Hibernate, todos baseados no padrão Database Broker.

A LINQ funciona como uma extensão da linguagem de programação, associando a simplicidade das expressões em SQL com o mapeamento objeto-relacional. As expressões em LINQ descrevem as operações definidas no modelo relacional através de expressões textuais que são tratadas pelo compilador e transformadas em código executável. O seu diferencial é a integração com a linguagem de programação, permitindo a interação direta entre as expressões de consulta e os modelos definidos pelo programador de forma rápida e eficiente. Como resultados, são obtidos objetos em vez de tabelas cruas ou conjuntos de registros. Um problema desta abordagem é o forte acoplamento entre o código e a fonte de dados, já que as operações são rigidamente definidas.

O Entity Framework propõe uma solução mais transparente que a LINQ, pois nenhum código precisa ser escrito para que o mapeamento seja estabelecido. Em vez disso utilizam-se apenas algumas entidades básicas para executar as operações sobre os dados. Os mapeamentos são gerados de forma automatizada e armazenados em arquivos XML. Estes arquivos são tratados pelo compilador e integrados ao executável sob a forma de recursos embutidos. Esta técnica eleva o nível de abstração, porém reduz o nível de controle e vincula a estrutura das entidades a uma representação específica no banco de dados. Isso significa que se houver alguma alteração na estrutura da fonte de dados o programa precisará ser recompilado, mesmo que esta alteração seja pequena.

O Hibernate utiliza uma técnica muito semelhante ao Entity Framework, porém os arquivos de configuração definem membro a membro, as correspondências entre os atributos das classes e os atributos das tabelas. Este método é um pouco mais flexível, mas ainda é limitado a um conjunto restrito de representações e a um conjunto específico de fontes de dados. Mapeamentos com entidades mais complexas precisam ser definidas em código e não é possível mapear a mesma entidade para mais de uma fonte de dados.

Tanto o Entity Framework quanto o Hibernate vinculam fortemente o programa a uma só fonte de dados. Nenhum deles permite definir mais de um mapeamento para uma mesma entidade, o que dificulta a persistência de uma entidade em múltiplas fontes de dados, necessidade comum em situações de integração ou em sistemas modulares. Uma variação desta situação é a persistência de partes de uma mesma entidade em bancos de dados diferentes (Figura 1), cenário muito comum na interoperabilidade com sistemas legados, mas negligenciado por ambas as soluções. Nestes casos o programador precisa criar suas próprias soluções para contornar as limitações dos frameworks e desenvolver as funcionalidades necessárias ao funcionamento do programa.

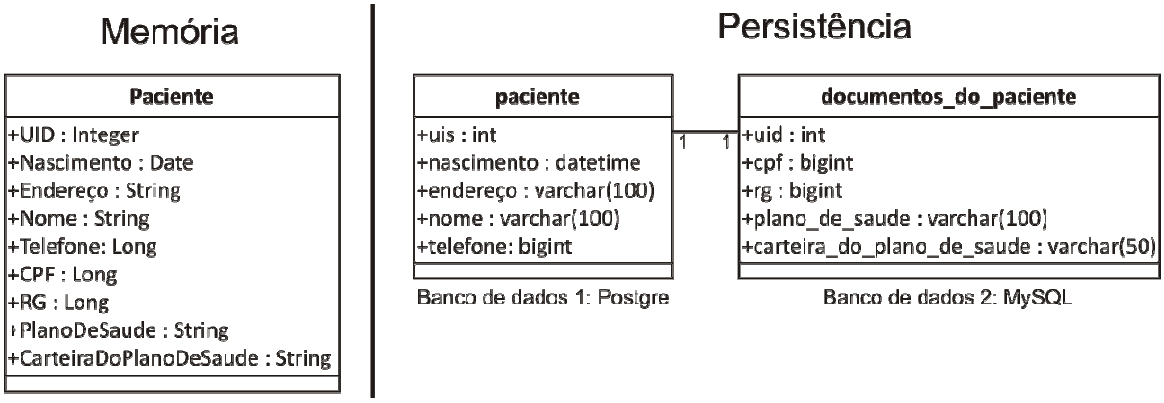

Figura 1: Persistência das partes de um mesmo objeto em dois bancos de dados distintos

Diante deste cenário, a HOML busca solucionar os problemas apresentados provendo uma forma de declaração de mapeamentos para fontes de dados heterogêneas [Shivakumar 1998], de maneira independente do mecanismo de processamento das operações. Isso permite aumentar a flexibilidade e eliminar o acoplamento do código.

3. Arquitetura

As atividades de manutenção de sistemas consomem de 50% a 70% de todo o esforço de desenvolvimento de um software, impactando seriamente em seu custo [Sommerville, 2011]. Este esforço é decorrente de mudanças corretivas ou preventivas nos sistemas, mudanças na legislação, mudanças nas regras de negócios e outras alterações, frequentemente implicando em mudanças nos dados e na sua persistência. A utilização de mecanismos que minimizem o esforço dedicado a esta tarefa é, portanto, essencial.

A HOML busca oferecer transparência na manipulação dos dados ao retirar do código a complexidade da integração entre o programa e as fontes de dados sem retirar do programador a sua capacidade de expressão. Toda a complexidade da operação fica encapsulada em drivers reutilizáveis que permitem o acesso transparente e integrado a fontes de dados heterogêneas. Isso reduz o esforço necessário para o desenvolvimento da camada de dados e ao mesmo tempo facilita o processo de manutenção e a integração com sistemas legados.

O que se pretende com a utilização do HOML é prover uma forma abstrata de definir mapeamentos para qualquer fonte capaz de fornecer informação e não simplesmente o mapeamento entre as entidades na memória e suas contrapartes em um banco de dados. Desta forma, as diversas fontes de dados são então tratadas genericamente como recursos que integram um repositório onde as operações são executadas de forma simplificada, padronizada e unificada, não importando se são bancos de dados, serviços, sistemas de arquivos ou qualquer outra origem.

A HOML foi projetada para dar suporte a diversas situações não contempladas por outras soluções, incluindo a persistência de uma mesma entidade em múltiplas fontes de dados, mesmo quando as representações dos dados divergem entre elas. A linguagem possui uma estrutura declarativa, onde as entidades são apresentadas de forma hierárquica. Isso permite definir quais drivers precisam ser carregados, como estes drivers referenciam cada fonte de dados, quais entidades estão armazenadas em cada fonte, como estas entidades se relacionam e como são mapeadas.

A estrutura da linguagem foi planejada para representar uma árvore de sintaxe abstrata (AST do inglês *Abstract Syntax Tree*), permitindo sua interpretação através de um processador de linguagem simplificado, dotado de um analisador léxico e um analisador sintático. As estruturas são carregadas para a memória e consultadas por meio de drivers previamente compilados. Por este motivo não existe interpretação durante a execução do programa, dispensando os estágios de análise contextual e compilação. Isso facilita a construção de processadores especializados para prover extensões da linguagem, requerendo pouco esforço de programação se comparado a um processador de linguagem completo. Esta extensibilidade é crucial para que se possa tirar proveito dos recursos já existentes sem limitar o poder de expressão e a capacidade de adaptação aos avanços posteriores. A análise semântica dos elementos básicos da linguagem é feita pelo próprio mecanismo de processamento, enquanto as extensões são processadas pelos respectivos drivers.

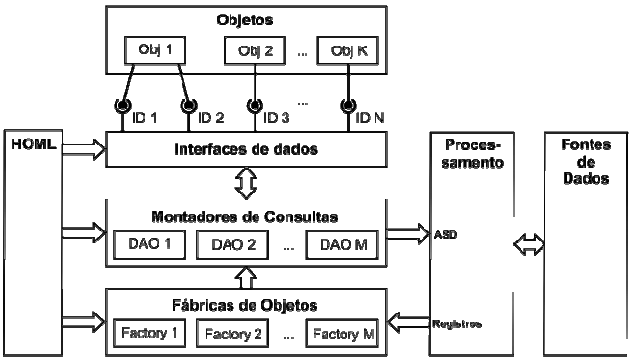

Figura 2: Estrutura de um sistema utilizando HOML

O ambiente de execução necessário para dar suporte ao processamento da HOML é organizado conforme visto na Figura 2. Os objetos consistem em estruturas de dados, definidas pelo programador, que podem ser compatíveis com uma ou mais interfaces de dados. Estas interfaces definem visões abstratas sobre o repositório, que permitem a persistência de um mesmo objeto de várias formas diferentes. Cada interface pode ser associada a um montador de consulta específico, permitindo que uma mesma entidade possa ser mapeada para múltiplas fontes de dados ou possa utilizar múltiplas representações, conforme o contexto de uso. Cada montador é responsável por construir expressões de consulta a partir das definições feitas em HOML para uma determinada interface dentro de uma fonte de dados específica. A expressão gerada consiste em uma AST que será enviada ao processador de linguagem. A expressão será então processada e ação correspondente será executada na fonte de dados pelo driver específico. O resultado da operação é então devolvido ao montador através de uma fábrica de objetos especializada e depois encaminhado para a interface de dados requisitante.

A vantagem da abordagem apresentada está no nível de abstração obtido, pois não é necessário interagir diretamente com a fonte de dados ou conhecer o mecanismo onde os dados são persistidos. Em vez disso, o programador executa métodos especializados que são associados com cada uma das operações definidas nos mapeamentos específicos de cada entidade para manipulação dos dados.

Para cada fonte de dados diferente deve existir um driver capaz de compreender a sua organização interna e traduzi-la para uma interface comum, permitindo o mapeamento de dados brutos para estruturas complexas. Cada driver deve ser capaz de enumerar o conteúdo

do repositório mapeado de maneira padronizada, permitindo que o mecanismo possa interagir com esse repositório através de entidades abstratas. Uma vez que um driver seja declarado, ele pode ser utilizado para mapear diversas fontes de dados do mesmo tipo, sendo responsável pela identificação e interpretação dos parâmetros necessários para o acesso aos dados.

4. Estrutura da Linguagem

A HOML tem sua estrutura baseada na XML e seus lexemas básicos são inspirados no modelo relacional e na SQL. As expressões são compostas por elementos declarativos, que informam quais fontes de dados estão sendo mapeadas, quais drivers fornecem o suporte para a manipulação de cada fonte de dados, quais entidades são mapeadas e como estas entidades são lidas e gravas em cada fonte de dados (operações). Para desempenhar estas funções, quatro seções básicas são definidas, conforme apresentado na Figura 3:

a) cabeçalho;

b) referências;

c) declarações;

d) mapeamentos.

```
<?xml version="1.0" encoding="utf-8" ?>
<homl  version="1.0">
  <header>
    <description>Documento de exemplo</description>
  </header>
  <references>
    <drivers><!--Lista de drivers que estão sendo declarados--></drivers>
  </references>
  <declarations>
    <datasources><!--Lista de fontes de dados a serem mapeadas com os drivers declarados--></datasources>
    <sets><!--Grupos personalizados para referenciar múltiplas fontes de dados em conjunto--></sets>
    <entities><!--Lista de entidades que estão sendo mapeadas--></entities>
  </declarations>
  <mappings>
    <all><!—Definições de mapeamentos para todas as fontes de dados--></all>
    <setName><!-- Definições de mapeamentos para um grupo de fontes de dados personalizado --></setName >
    <DataSourceName><!-- Definições exclusivas para uma fonte de dados específica--></DataSourceName >
  </mappings>
</homl>
```

Figura 3: Exemplo genérico de documento HOML

O cabeçalho tem propósito informativo e serve para documentação do arquivo. Ele permite que o usuário defina suas próprias marcas de forma livre, sem nenhum tipo de semântica predefinida. As demais seções do documento permitem declarar as fontes de dados que serão utilizadas, definir agrupamentos para simplificação dos mapeamentos, declarar as entidades que serão mapeadas, definir quais as operações que estarão disponíveis para cada entidade em cada fonte de dados e determinar a regra de processamento destas operações.

A primeira seção funcional do documento contém a área de referências e permite declarar as bibliotecas externas que serão utilizadas como drivers para permitir processamento de das fontes de dados. Para cada driver, esta seção deve conter um nome que servirá para identificar a fonte de dados declarada e um caminho indicando a localização da biblioteca. Cada nome deve ser exclusivo, pois ele permite que sejam feitas referências ao Driver mapeado sempre que necessário.

```
<references>
  <drivers>
    <driver name="MySQL" driver="c:\drivers\MySQL.dll"/>
    <driver name="GPS" driver="c:\drivers\GPS.dll"/>
  </drivers>
</references>
```

Figura 4: Exemplo de referências

O caminho do driver deve indicar a localização da biblioteca (arquivo) que contém as funcionalidades necessárias para dar suporte à fonte de dados declarada. Um exemplo de utilização desta seção é apresentado na Figura 4, onde são referenciadas as bibliotecas para mapeamento do MySQL e para um dispositivo GPS. Este exemplo é particularmente importante, pois demonstra que um SGBD e um dispositivo GPS são tratados como fonte de dados, sem qualquer distinção. Cabe ao criador do driver prover as funcionalidades necessárias para garantir a transparência, permitindo que as definições feitas em HOML possam funcionar de maneira apropriada.

Cada um dos drivers definidos na seção de referências pode ser associado a uma fonte de dados específica, conforme demonstrado na Figura 5. Isso é feito pela simples associação de um nome exclusivo, um driver e um conjunto de parâmetros de conexão. Cada driver distinto pode ter seu conjunto específico de parâmetros que permitem definir como a conexão com a fonte de dados será estabelecida. Estes parâmetros são definidos pelo criador do driver, conforme a necessidade específica, sendo possível utilizar o mesmo driver para mais de uma fonte de dados, conforme demonstrado no exemplo apresentado, onde são mapeadas duas fontes de dados MySQL e uma fonte de dados GPS.

```
<datasources>
  <datasource name="MySQL1" driver="MySQL">
    <attribute name="server" value="localhost"/>
    <attribute name="userid" value="usuario"/>
    <attribute name="password" value="senha"/>
    <attribute name="database" value="banco_de_dados"/>
  </datasource>
  <datasource name="MySQL2" driver="MySQL"/>
    <attribute name="server" value="localhost"/>
    <attribute name="userid" value="usuario"/>
    <attribute name="password" value="senha"/>
    <attribute name="database" value="banco_de_dados"/>
  </datasource>
  <datasource name="GPS1" driver="GPS"/>
    <attribute name="port" value="COM1"/>
    <attribute name="protocol" value="NMEA 0183"/>
    <attribute name="databits" value="8"/>
    <attribute name="stopbits" value="0"/>
    <attribute name="parity" value="none"/>
    <attribute name="baudrate" value="115200"/>
  </datasource>
</datasources>
```

Figura 5: Exemplo de mapeamento de fontes de dados

Para evitar o uso de declarações redundantes, a HOML prevê a declaração de grupos. Isso permite a definição de mapeamentos comuns a múltiplas fontes de dados através de uma só declaração, conforme a Figura 6. Cada grupo pode assumir qualquer nome, exceto pela

palavra reservada “All”, que possui semântica própria e representa um grupo que referencia todas as fontes de dados ao mesmo tempo. Não é permitida também a utilização de nomes que já tenham sido atribuídos a outros grupos e/ou fontes de dados.

```
<sets>
  < set name="Grupo1">
     <item name="MySQL1"/>
     <item name="MySQL2"/>
  </set>
</sets>
```

Figura 6: Exemplo de declaração de grupos

Ainda na seção de declarações, as entidades que serão mapeadas também precisam ser referenciadas. Cada entidade é identificada pela sua identificação completa, incluindo o namespace (package para o Java). Tomando como exemplo a Figura 7, temos as classes “meuprojeto.entidades.Usuario” e “meuprojeto.entidades.Location” sendo declaradas. Cada entidade deve receber um nome exclusivo que não precisa ser o mesmo nome da classe referenciada. Este nome tem como propósito identificar a classe dentro da estrutura do HOML, porém não afeta o funcionamento do mecanismo. Isto significa que não é possível atribuir dois nomes diferentes para a mesma classe. Apesar desta restrição, é possível definir diversos mapeamentos para a mesma classe. Para isso, dois nomes distintos podem ser atribuídos declarados para fontes de dados iguais ou diferentes. Para cada um deles, pode ser associado um conjunto de mapeamentos diferenciado referenciando as mesmas classes.

```
  <entities>
    <entity name="Usuario" namespace="meuprojeto.entidades"/>
    <entity name="Location" namespace="meuprojeto.entidades"/>
  </entities>
</declarations>
```

Figura 7: Exemplo de declaração de entidades

Uma vez que todas as referências e declarações tenham sido feitas os mapeamentos podem ser criados. Estes mapeamentos podem ser declarados tanto de forma individual quanto através de grupos. Como existe a possibilidade de mapeamentos distintos feitos para a mesma entidade resultarem em ambiguidades, o critério de solução de conflitos leva em consideração a especificidade do mapeamento e a sua precedência. Deste modo uma definição de mapeamentos feita para uma entidade específica tem prioridade sobre outro que tenha sido feito em grupo. Caso o critério da especificidade não possa ser utilizado, prevalece aquele mapeamento que for definido primeiro (na ordem de leitura do arquivo HOML).

A Figura 8 apresenta um exemplo de mapeamento da entidade “Usuario” para o grupo de fontes de dados “Grupo1”. Neste exemplo foram declaradas as operações “select”, “insert”, “update” e “delete”, porém não foram especificadas como estas operações serão processadas. Tais operações possuem uma semântica padrão, deste modo não é necessário definir detalhadamente como deverão funcionar, exceto quando o comportamento desejado for diferente do comportamento padrão.

```
<mappings>
  < Grupo1>
    <usuario>
      <attributes>
        <attribute name="id" field="id_usuario" behaviour="autoincrement" keytype="primary"/>
```

```
        <attribute name="nome"/>
        <attribute name="login" behaviour="writeonce" keytype="exclusive"/>
        <attribute name="senha"/>
      </attributes>
      <source entity="tabela_usuarios"/>
      <operations>
        <select/>
        <insert/>
        <delete/>
        <update/>
      </operations>
    </usuario>
  </ Grupo1>
</mappings>
```

Figura 8: Exemplo de mapeamento

A semântica padrão assume que o nome da entidade é o mesmo na memória e na fonte de dados, assim como os nomes dos seus atributos. Do mesmo modo as operações Insert, Delete, Update e Select possuem a mesma semântica das respectivas operações no modelo relacional. Estas definições básicas permitem simplificar o mapeamento das entidades para casos triviais, poupando o usuário da criação de expressões detalhadas para cada operação.

A HOML define não apenas um conjunto próprio de semânticas, mas também algumas regras para lidar de forma simples com os mais diversos casos de mapeamentos, permitindo definir todas as operações do modelo relacional utilizando um conjunto mínimo de declarações. Foge ao escopo deste artigo tratar cada uma destas regras, no entanto é importante mencioná-las uma vez que fazem parte do conjunto de funcionalidades provido pela linguagem. Tais facilidades criam um ambiente que facilita o uso ferramentas capazes de automatizar o processo de construção dos mapeamentos, permitindo, por exemplo, que a estrutura de um banco de dados seja inferida a partir de um documento HOML ou que um documento HOML seja inferido a partir de um banco de dados. Isso permite reduzir sensivelmente o esforço de programação e pode facilitar a distribuição do software produzido.

5. Conclusões

A HOML apresenta uma forma para simplificar o mapeamento de fontes de dados heterogêneas. As fontes de dados podem ser tratadas como recursos em um repositório onde as operações são executadas de forma transparente. Isso permite que os recursos possam ser bancos de dados, serviços, sistemas de arquivos ou qualquer outra fonte de informação, sendo necessária apenas a criação de um driver especializado para cada fonte.

A estrutura da HOML facilita o seu processamento e permite a sua extensibilidade de maneira relativamente simples, dispensando diversas etapas do processamento. Isso permite que a linguagem possa ser estendida facilmente pelo programador e ao mesmo tempo possa ser processada de modo automatizado.

Os mapeamentos definidos em HOML podem ser modificados de maneira simples, sem a necessidade de modificações no programa principal, facilitando a adoção de mudanças e até mesmo a modificação no repositório de dados. O resultado disso é um enorme ganho em produtividade obtido pela intensa reutilização de código.

Para proporcionar maior familiaridade dos programadores com estas expressões, seus elementos foram inspirados na linguagem SQL e no modelo relacional, reduzindo drasticamente a curva de aprendizado. A intenção neste caso não é de que o programador crie

ou mantenha manualmente estes arquivos, mas que ele possa entendê-los depois de gerados. A geração efetiva destes mapeamentos pode ser feita de forma automatizada, eliminando erros e reduzindo o tempo necessário para a realização de uma tarefa maçante e enfadonha.

Na etapa atual de desenvolvimento, está sendo estudada a utilização da HOML para a simplificação da composição automática de serviços. Neste caso, a HOML permitirá ao programador utilizar sua própria estrutura para tratar a composição de serviços em memória obtendo informações dos serviços declarados e devolvendo a estes serviços as informações processadas. Através das operações definidas por meio da HOML, o programa poderá converter automaticamente as informações processadas para as linguagens nativas de cada serviço.

6. REFERÊNCIAS

Bauer, C. E King G. Hibernate in Action. Greenwich: MANNING, 2005. 430 p.

Codd, E. F. The Relational Model for Database Management: Version 2. Washington: Addison-Wesley, 1990. 538 p. ISBN 0-201-14192-2

DATE, C. J. An Introductionj to Database Systems. Rio de Janeiro: Campus, 1991-c1990. 674 p.

Fowler, Martin; Rice, Davi; Foemmel, Matthew; Hieatt, Edward; Mee, Robert; Stafford, Randy. Patterns Of Enterprise Application Architecture. Addison Wesley. 2002. 389p. ISBN: 0-321-12742-0.

Holder, S., Buchan, J., & MacDonell, S.G. (2008) Towards a metrics suite for object-relational mappings, in Proceedings of the 1st International Workshop on Model-Based Software and Data Integration.Berlin, Germany, Springer (Communicationsin Computer and Information Science v.8), pp.43-54.

Shivakumar Venkataraman; Tian Zhang. Heterogeneous Database Query Optimization in DB2 Universal DataJoiner. New York City: 24th VLDB Conference, 1998. 5p. Available at: <http://www.vldb.org/conf/1998/p685.pdf>. Access at: nov. 14 2011.

Sommerville, Ian,. Engenharia de software. São Paulo: Pearson Education do Brasil, 2011. xiii, 529 p. ISBN 9788579361081.

SOUKUP, Ron. Desvendando o Microsoft SQL Server 6.5. Rio de Janeiro: Campus. 1998. 883 p.

Vidal, Vania; Santos, Lineu; Araujo, Valdiana; Lemos, Fernando. Geração Automática de Visões de Objeto de Dados Relacionais. Uberlândia: XX Simpósio Brasileiro de Banco de Dados, 2005. 15p. Available at: <http://www.lbd.dcc.ufmg.br:8080/colecoes/sbbd/2005/011.pdf>. Access at: jan. 14 2012.