Multi Locale Entity: Um padrão de projeto para persistência de entidades com internacionalização

Carlo Giovano S. Pires [1]
cgiovano@atlantico.com.br
Instituto Atlântico

Resumo

A ampla adoção de aplicações Web e distribuídas requer processos para internacionalização e localização que permitam que as aplicações possam dar suporte à diferentes línguas e regiões. Esse trabalho apresenta um padrão de projeto (padrão Multi Locale Entity) que fornece suporte para internacionalização e localização. O padrão apresenta uma solução para a camada de negócio, camada de integração e camada de dados.

Abstract

The large adoption of Web and distributed application requires processes of internationalization and localization in order to enable applications to support different languages and regions. This paper presents a design pattern (Multi Locale Entity pattern) that supports internationalization and localization. It presents a solution for the business tier, the integration tier and the data tier.

Introdução

A larga difusão de aplicações Web e distribuídas exige cada vez mais o suporte de internacionalização e localização de aplicações. Internacionalização [4] é o processo de projetar uma aplicação para se adaptar a várias línguas e regiões. Localização [4] é o processo de adaptar uma aplicação para uma língua e região.

A internacionalização de uma aplicação pode ser feita ao nível de elementos da interface e dos próprios dados informados. Apesar da existência de outros modelos de dados (modelo orientado a objetos [1], hierárquico, entre outros), o modelo relacional representa a grande maioria das aplicações com acesso à base de dados. O padrão proposto tem o objetivo de fornecer um mecanismo de representação e persistência para entidades com suporte a internacionalização, facilitando a localização de aplicações e otimizando o acesso aos dados.

[1] Este trabalho foi suportado pelo Instituto Atlântico (*www.atlantico.com.br*).

Copyright 2003, Carlo Giovano S. Pires. Permissão de cópia concedida para a Conferência Sugarloaf-PLoP 2003. Todos os outros direitos reservados. Agradecimentos ao shepherd Sérgio Soares por sua revisão e valiosas sugestões

1 Contexto

Desenvolvimento de aplicações Web e distribuídas (N-Camadas como, por exemplo, CORBA ou J2EE) com bancos de dados relacionais que devem ser utilizadas em diversas regiões devem dar suporte aos processos de localização e internacionalização dos dados. Nesse tipo de aplicações, uma mesma entidade (instância, ou *pool* de instâncias) é compartilhada por diversos clientes. Cada cliente pode estar em alguma região do mundo e necessitar consultar e editar a entidade na sua própria língua.

Quanto aos dados, as tabelas em um banco relacional e as classes de negócio e persistência devem ser projetadas de forma a racionalizar o suporte a várias línguas.

2 Problema

Como desenvolver objetos para representação de entidades de negócio, a camada de acesso a dados e o modelo de dados relacional de forma a dar suporte à persistência de dados em várias línguas?

3 Forças

- O acesso à informação sensível a língua (localizada) deve ser simples para fornecer boa usabilidade para as classes localizadas;
- O processo de localização (por exemplo, adicionar novas línguas à aplicação e atualizar informações sensíveis à língua) deve ser feito sem recompilação da aplicação, oferecendo manutenibilidade à aplicação;
- O mecanismo de acesso aos dados deve minimizar as consultas à informação para cada língua de forma a não prejudicar o desempenho da aplicação;
- O mecanismo de acesso a dados deve estar separado da entidade de negócio para oferecer extensibilidade com relação ao modelo de dados e tecnologia de acesso ao repositório.

4 Solução

O padrão fornece uma solução que abrange a camada de negócios, camada de integração e a camada de dados. A solução para camada de negócio e integração mostra como projetar classes para representar a entidade que deve ser localizada e como projetar uma classe para armazenar e recuperar a informação dessa entidade em uma base de dados. A solução para a camada de dados mostra como projetar tabelas em um banco de dados relacional para armazenar informações de entidades com suporte a internacionalização.

Quanto às camadas de negócio, o padrão fornece a classe *Language* para representação da língua, as classes *MultiLocaleEntity* e *LocaleSensitiveValue* para representar a entidade internacionalizada com informação não dependente de língua e com informação dependente de língua. A classe *MultiLocaleDataAccess* é utilizada na camada de integração para gerenciar a persistência da informação. A classe de entidade *MultLocaleEntity* deve fornecer um método para leitura de cada atributo sensível a língua. Esse método recebe como parâmetro o objeto que representa a língua e retorna a informação localizada. O objeto de persistência realiza a interface com a base de dados, preenchendo a informação da entidade para cada língua, atualizando, criando e excluindo a informação quando necessário.

Quanto à camada de acesso a dados, deve-se fornecer uma tabela com a informação que não é sensível à língua e possui o ID da entidade, e uma outra que é o relacionamento entre essa primeira e a tabela de línguas. A tabela de relacionamento deve ter um campo para cada atributo sensível à língua.

5 Estrutura

A figura 1 apresenta o diagrama de classes da camada de aplicação do padrão *Entidade Multilíngua*. Uma descrição de cada participante é apresentada na seção *Participantes*.

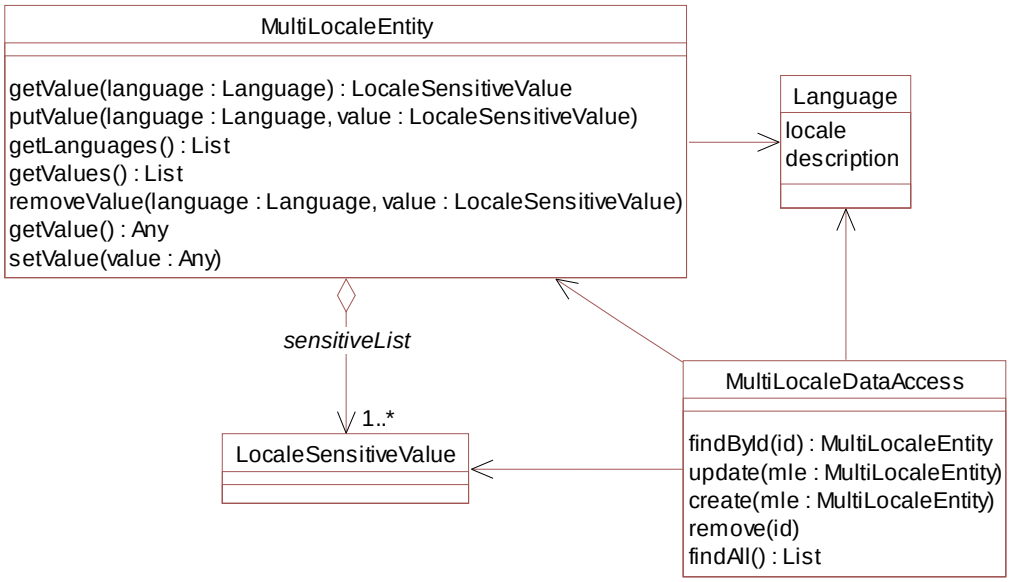

Figura 1 – Diagrama de Classes do Padrão Multi Locale Entity na camada de negócio e integração

A figura 2 apresenta o esquema da camada de dados do padrão *Entidade Multilíngua*. Uma descrição de cada participante é apresentada na seção *Participantes*.

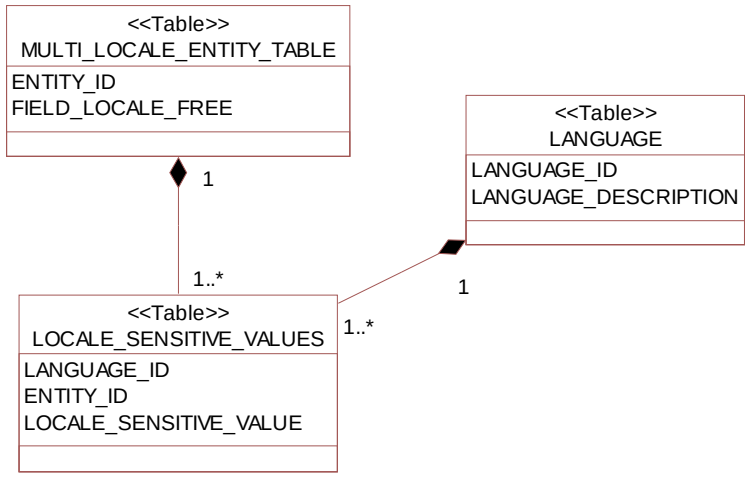

Figura 2 – Esquema do Padrão Multi Locale Entity na camada de dados

6 Participantes

6.1 Camada de Negócio

- **MultiLocaleEntity**
  - Fornece operações para cada atributo para consultar a informação (*getValue*) e outro para atualizar a informação (*putValue*). As operações permitem a passagem da língua como parâmetro
  - Possui uma lista de objetos *LocaleSensitiveValue* que podem ser indexados pela língua (Language)
  - Fornece operação (*removeValues*) para remover um elemento de acordo com a língua
  - Fornece operação (*getLanguages*) para listar as línguas suportadas pela entidade/aplicação

- **LocaleSensitiveValue**
  - Representa o valor dependente da língua
  - Pode ser uma classe complexa (por exemplo, classe que representa o cálculo de imposto dependendo do país) ou tipos primitivos como caracteres e números (por exemplo, descrições e valores monetários)

- **Language**
  - Representa a língua
  - Possui atributo (*locale*) para identificar a língua. Esse atributo identifica a língua propriamente dita e a região. Por exemplo, en_US, representa inglês dos Estados Unidos
  - Possui atributo (*description*) para descrever a língua. Por exemplo, Inglês

6.2 Camada de Integração

- **MultiLocaleDataAccess**
  - Implementa o mecanismo de persistência da entidade sensível à língua
  - Define instruções SQL [2] que fazem o mapeamento da entidade em tabelas com suporte a várias línguas

6.3 Camada de Dados

- **Multi_Locale_Entity_Table**
  - Tabela que possui como chave primária o atributo identificador da entidade (*Entity_Id*) e os atributos que não dependem de língua (*Field_Locale_Free*)
  - Funciona como entidade forte para a tabela fraca *Locale_Sensitive_Values*

- **Locale_Sensitive_Values**
  - Tabela que possui como chave primária o identificador da entidade (*Entity_Id*) e o identificador da língua (*Language_Id*), permitindo armazenar valores diferentes, para línguas diferentes em uma mesma entidade
  - Possui atributos para armazenar dados que dependem da língua (*Locale_Sensitive_Value*)

- **Language**

  - Tabela que armazena as línguas suportadas pela aplicação
  - Possui como chave primária o identificador da língua (*language_id*) e possui um atributo com a descrição da língua (*Language_Description*)

7 Dinâmica

Veja fluxo da consulta para o primeiro acesso de uma entidade multilíngua (ver figura 3):

1. Cliente consulta entidade por seu identificador (*Id*)
2. *MultiLocaleDataAccess* constrói novo objeto *MultiLocaleEntity* e executa consulta por *Id*, recuperando os dados da tabela independente de língua e da tabela sensível a língua. *Para cada linha recuperada*
   2.1 Preenche atributo não-sensível à língua em *MultiLocaleEntity*
   2.2 Adiciona à lista *sensitiveList* o objeto sensível à lista, passando *Language* como parâmetro (*Language* é construído com base no id da língua armazenada na tabela sensível a língua - *LocaleSensitiveValues*)
3. *MultiLocaleDataAccess* retorna objeto *MultiLocaleEntity*
4. Cliente obtém língua do usuário através de algum contexto e obtém valor de um atributo sensível à língua através do método *getValue(contextLanguage)*. Atributos não sensíveis à língua são recuperados através de algum método do tipo *getValue()*
5. O cliente atualiza algum atributo sensível à língua através de algum método *putValue(contextLanguage, newLocaleSensitiveValue)*, passando o novo valor e a língua selecionada como parâmetro
6. O cliente confirma atualização do objeto através do objeto *MultiLocaleEntityDataAccess*
7. *MultiLocaleEntity* executa (dentro de uma única transação)
   7.1 Instrução de atualização em *Multi_Locale_Entity_Table* para os atributos não sensíveis à língua modificados
   7.2 Obtém lista de línguas disponíveis em *MultiLocaleEntity* através do método *getLanguages*
   7.3 Para cada língua da lista, recupera objeto modificado através de *getValue* de *MultiLocaleEntity* e executa instrução de atualização na tabela *Locale_Sensitive_Values*

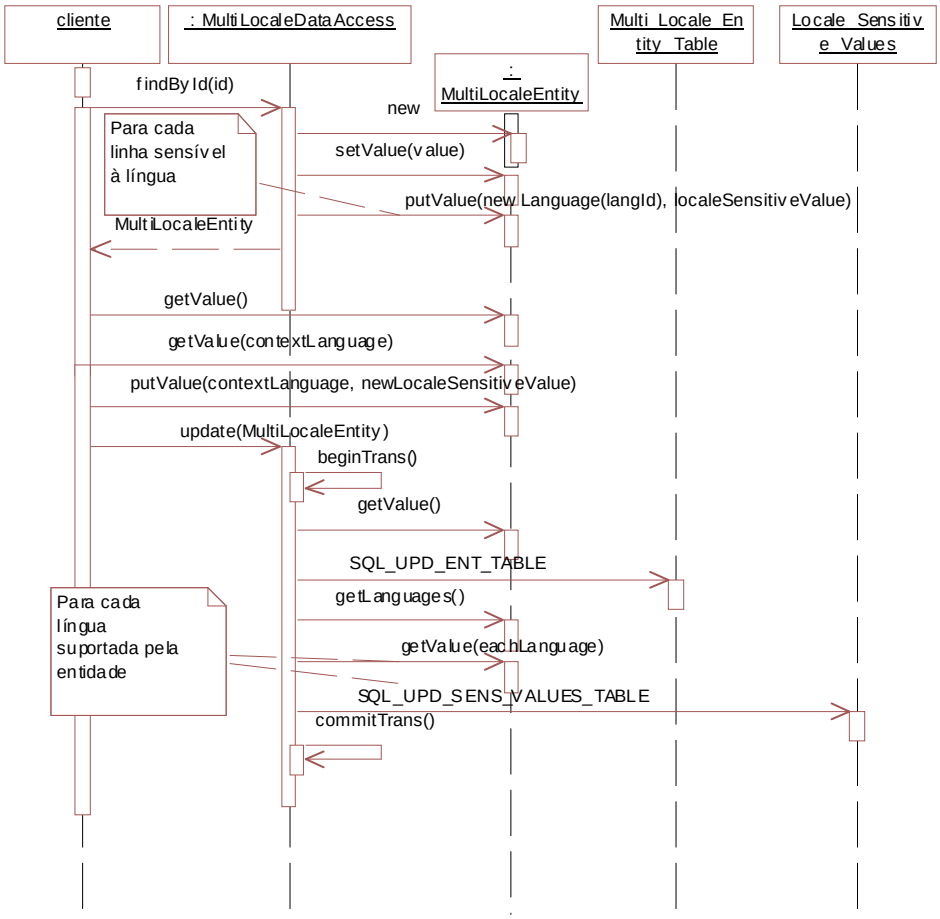

Figura 3: Consulta e atualização de entidade

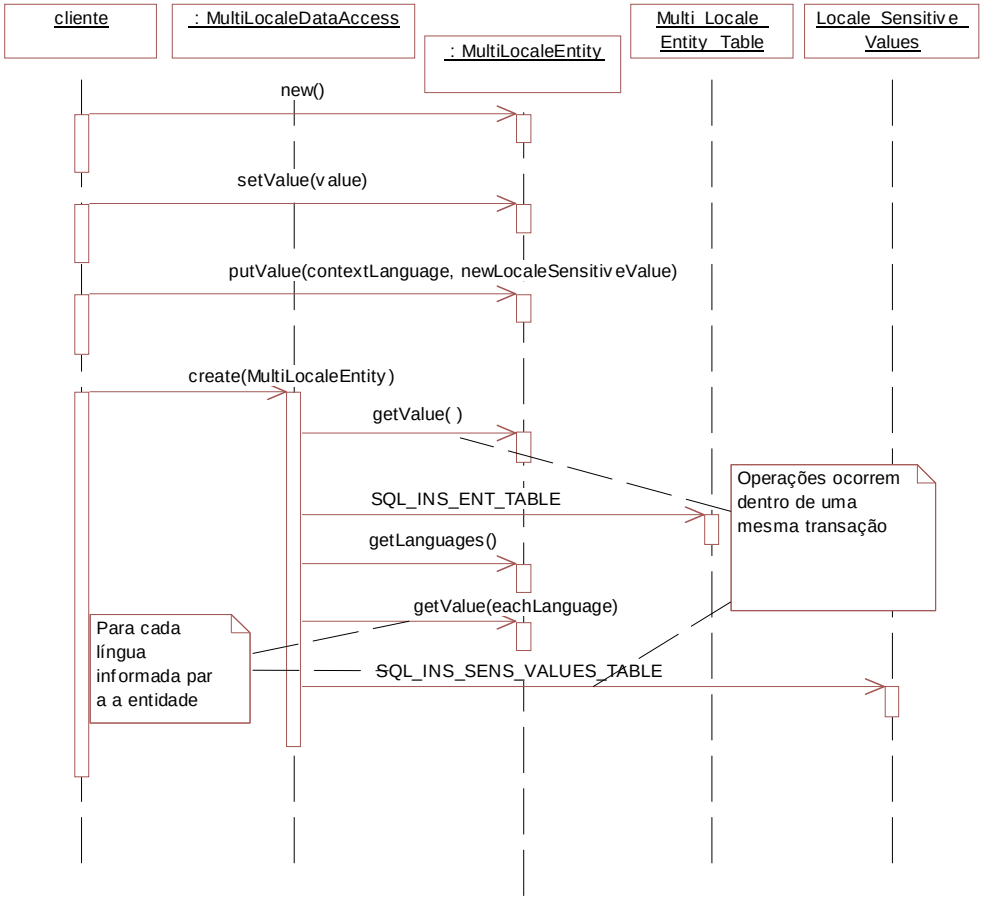

Figura 4: Criação de entidade

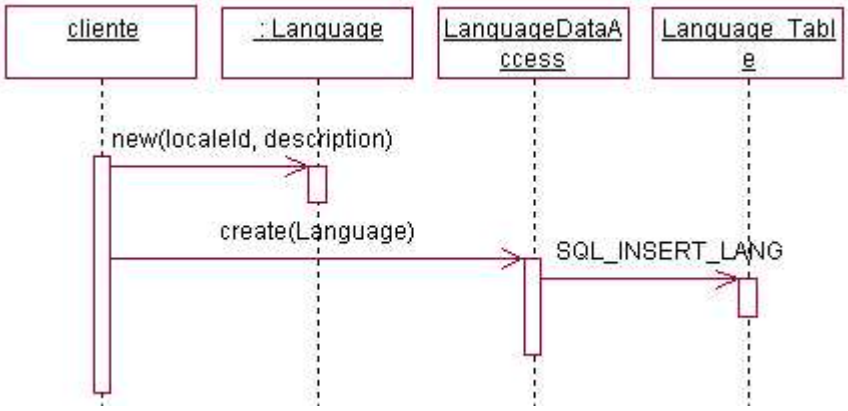

Figura 5: Adicionando uma nova língua

As figuras 4 e 5 apresentam a criação de uma nova entidade e a adição de uma nova língua à aplicação.

8 Conseqüências

8.1 Vantagens

- A classe que representa a entidade (*MultiLocaleEntity*) fornece uma interface simples e bem definida para acessar a informação/atributos de acordo com a língua do contexto da aplicação, facilitando o acesso a informações localizadas e aumentando a usabilidade.
- A configuração de novas línguas pode ser feita via classe *Language* e novas informações (ou atualizações) referentes a uma língua podem ser registradas através da interface *putValue* da classe *MultiLocaleEntity* sem recompilação da aplicação, oferecendo boa manutenibilidade para a aplicação
- O acesso a dados é minimizado através da recuperação e *cache* das informações para cada língua no objeto *MultiLocaleEntity* melhorando o desempenho das operações de localização da aplicação
- A classe *MultiLocaleEntityDataAccess* isola a entidade de negócio do modelo e tecnologia de acesso a dados, fornecendo extensibilidade com relação ao modelo de dados e tecnologia de acesso.
- A separação de tabelas do modelo relacional entre informações sensíveis à língua e informações insensíveis à língua fornece maior racionalização do armazenamento das informações

8.2 Desvantagens

- O *cache* das informações por língua pode gerar maior utilização de memória
- Se os dados da aplicação forem atualizados por outros meios diferentes das classes do padrão, as informações em *cache* podem ficar desatualizadas e inconsistentes com a base de dados
- O padrão gera um número maior de classes e pode gerar maior complexidade para entendimento

9 Variantes

Apesar do padrão dar ênfase ao modelo relacional, o padrão também pode ser utilizado para outros modelos de dados, basta utilizar outras implementações de *MultiLocaleDataAccess* para cada modelo, utilizando instruções de consulta equivalentes.

10 Exemplo

O exemplo exibe parte do código utilizado para implementar uma entidade sensível à língua para um portal na *Web*. Essa entidade representa os tipos de conteúdos que podem ser apresentados no portal. Os dados da entidade são utilizados na página principal do portal para que o usuário possa selecionar qual tipo de conteúdo deseja ver: Esportes, Notícias, Diversões, Tecnologia, Cinema, etc. O portal fornece uma aplicação de *backoffice* que permite que administradores possam criar, remover, atualizar, habilitar e desabilitar tipos de conteúdo.

A entidade possui os seguintes atributos que não dependem de língua: i*d, enabled* e *totalHits*. O atributo *id* é o identificador único do tipo de conteúdo, o atributo *enabled* indica que um tipo de conteúdo pode ser habilitado ou desabilitado. Conteúdos desabilidades não

aparecem no portal. Outro atributo que não depende da língua é *totalHits*. Esse atributo indica o número total de acessos a determinado tipo de conteúdo, independemente da língua ou país.

A entidade também possui os seguintes atributos que são sensíveis à língua: *description* e *priority*. O atributo *description* indica a descrição do tipo de conteúdo em cada língua suportada pelo portal. Dessa forma, o usuário que acessar o portal pode ver os tipos de conteúdo disponíveis na língua mais adequada. O atributo *priority* é um objeto que descreve a ordem (*order*) e cor (*color*) que podem caracterizar um tipo de conteúdo dependendo da língua do usuário. Dessa forma, o administrador do portal pode indicar, por exemplo, que um usuário que seleciona a língua *Português/Brasil* veja o conteúdo Diversão no topo da página e em vermelho, enquanto um usuário com língua *Japonês/Japão* veja Tecnologia no topo com cor vermelha e Diversão no meio da lista e com cor azul.

Na aplicação do padrão *Multi Locale Entity* para a entidade de tipo de conteúddo, temos as classes *ContentType*, *Priority, ContentTypeDataAccess*. A primeira funciona como *MultiLocaleEntity,* a segunda como *LocaleSensitiveValue* e a terecenira como *MuliLocaleEntityDataAccess*. Para o atributo *description* de *ContentType,* o *LocaleSensitiveValue* é apenas o tipo *String*. Veja na figura 6 que na classe *ContentType* temos as operações *getDescription(language : Language)* e *getPriority(language : Language)* como aplicações da operação *MultiLocaleEntity::getValue(language : Language)* do padrão. Também temos as operações *putDescription(language : Language, value: String)* e *putPriority(language : Language, value: Priority)* como aplicações da operação *MultiLocaleEntity::putValue(language : Language, value : LocaleSensitiveValue)* do padrão. Finalmente temos as operações *removeDescription(language : Language, value: String)* e *removePriority(language : Language, value: Priority)* como aplicações da operação *MultiLocaleEntity::removeValue(language : Language, value : LocaleSensitiveValue)* do padrão.

Para os atributos que não dependem de língua, temos *getEnabled()* e *setEnabled(e : boolean)*, *getTotalHits()* e *setTotalHists(h : Integer)* como aplicações de *MultiLocaleEntity::getValue()* e *MultiLocaleEntity::setValue(v: Any)* do padrão.

As operações *getDescriptions()* e *getPriorities()* são aplicações de *MultiLocaleEntity::getValues().*

As classes do exemplo foram implementadas em *Java*.

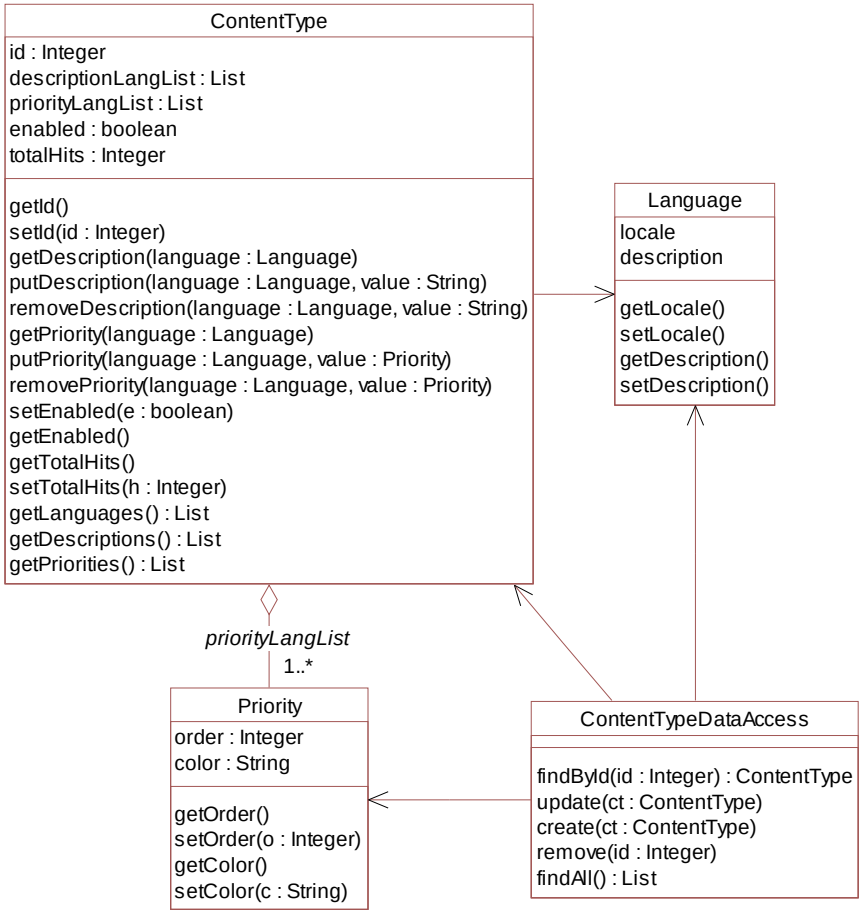

Figura 6: Diagrama de classes para entidade ContentType

As tabelas criadas foram CONTENT_TYPE, CONTENT_TYPE_LOCALE_SENSITIVE_VALUES representado, respectivamente, MULTI_LOCALE_ENTITY_TABLE e LOCALE_SENSITIVE_VALUES. As classes para *Language* e a tabela LANGUAGE também foram criadas. Veja na figura 7 que apenas os campos que não dependem de língua estão na tabela CONTENT_TYPE, os campos sensíveis à língua estão na tabela CONTENT_TYPE_LOCALE_SENSITIVE_VALUES juntamente com a chave primária composta por CONTENT_TYPE_ID e LANGUAGE_ID.

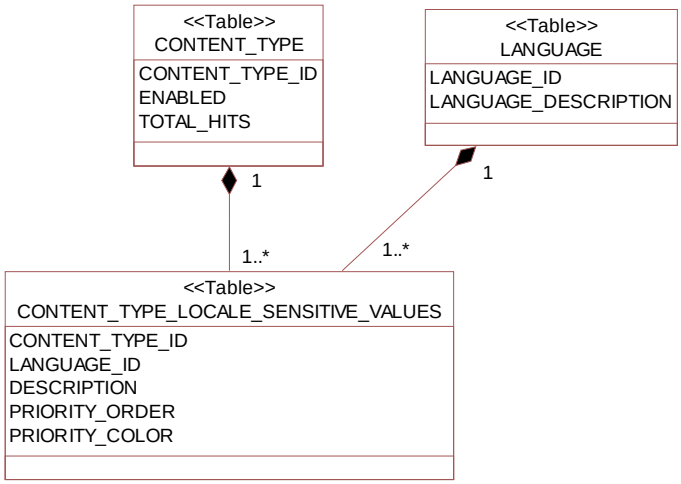

Figura 7: Esquema de dados para entidade ContentType

O trecho de código na Figura 8 apresenta variáveis da classe *ContentTypeDataAccess* que representam as consultas SQL para armazenamento e recuperação da entidade na base de dados:

```
private static final String SQL_FIND_BY_ID = "SELECT CONTENT_TYPE.CONTENT_TYPE_ID,
CONTENT_TYPE.ENABLED, CONTENT_TYPE.TOTAL_HITS,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.LANGUAGE_ID,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.DESCRIPTION,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.PRIORITY_ORDER,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.PRIORITY_COLOR FROM CONTENT_TYPE,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES WHERE CONT_TYPE_ID=? AND
CONTENT_TYPE.CONT_TYPE_ID = CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.CONT_TYPE_ID ";

private static final String SQL_FIND_ALL = "SELECT CONTENT_TYPE.CONTENT_TYPE_ID,
CONTENT_TYPE.ENABLED, CONTENT_TYPE.TOTAL_HITS,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.LANGUAGE_ID,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.DESCRIPTION,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.PRIORITY_ORDER,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.PRIORITY_COLOR FROM CONTENT_TYPE,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES WHERE CONTENT_TYPE.CONT_TYPE_ID =
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.CONT_TYPE_ID ";

private static final String SQL_CREATE_ENT = "INSERT INTO CONTENT_TYPE
(CONT_TYPE_ID, ENABLED, TOTAL_HITS) VALUES (?, ?, ?)";

private static final String SQL_CREATE_LSENS = "INSERT INTO
(CONTENT_TYPE_LOCALE_SENSITIVE_VALUES CONTENT_TYPE_ID, LANGUAGE_ID,DESCRIPTION,
PRIORITY_ORDER, PRIORITY_COLOR) VALUES (?,?,?,?,?)";

private static final String SQL_DELETE_ENT = "DELETE FROM CONTENT_TYPE WHERE
CONTENT_TYPE_ID=?";

private static final String SQL_DELETE_LSENS = "DELETE FROM
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES WHERE CONTENT_TYPE_ID=?";
```

Figura 8 – Consultas de ContentTypeDataAccess

O trecho de código na Figura 9 apresenta o código para recuperar uma lista de todos os tipos de conteúdo. Veja que logo abaixo do comentário “//Verifica se já não está na lista”, uma condição é testada para evitar que a instância de *ContentType* seja adicionada à lista, caso ela já tiver sido adicionada. Isso evita que as repetições de informações não sensíveis à língua gerada pela junção das duas tabelas no SQL_FIND_ALL sejam inseridas na lista. Logo

após a condição, o método *putDescription* de *ContentType* é acionado, colocando as informações sensíveis à língua, no caso a descrição do tipo de conteúdo. Depois que todas as informações sensíveis à língua para um mesmo *id* são colocadas na instância, uma nova instância, para novo *id* é criada e o processo se repete.

```
public List findAll() throws DataAccessException {

        Connection conn = null;
        try {
            conn = getDBConnection();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(SQL_FIND_ALL);
            List list = new ArrayList();
            ContentType contentType = null;
            String id;
            while (rs.next()) {
                id = rs.getString("CONT_TYPE_ID");
                //Verifica se já não está na lista
                if (list.size()==0 || !((contentType)list.get(list.size()- 1
                    )).getId().equals(id)) {
                   contentType = new ContentType(id);
                   contentType.setEnabled(rs.getString("ENABLED").equals("YES"));
                   contentType.setTotalHits(rs.getString("TOTAL_HITS"));

                   list.add(ContentType);
                }

                Priority priority = new Priority();
                Priority.setOrder(rs.getInteger("PRIORITY_ORDER"));
                priority.setColor(rs.getString("PRIORITY_COLOR"));

                contentType.setPriority(priority);
               contentType.putDescription(LocaleFormat.parse(rs.getString("LANGUAG
               E_ID")),rs.getString("DESCRIPTION"));
            }
            rs.close();
            stmt.close();
            return list;
        }
        catch (Exception ex) {
            throw DataAccessException.buildException(ex);
        }
        finally {
            closeDBConnection(conn);
        }

    }
```

Figura 9 – Operação findAll de ContentTypeDataAccess

O trecho de código na Figura 10 apresenta o código para inserir um novo tipo de conteúdo no portal. Note que o primeiro SQL (SQL_CREATE_ENT) é executado uma única vez para criar a nova entidade e o segundo SQL (SQL_CREATE_LSENS) é executado para todas as línguas informadas (*contentType.getLanguages*) para a nova entidade, inserindo várias linhas na tabela, para cada novo *id* da entidade sensível à língua. Todas as operações são feitas dentro de uma única transação.

```
public Integer create(ContentType contentType) throws DataAccessException {

        Connection conn = null;
        try {
            conn = getDBConnection();
            int id = getNextId();
            conn.setAutoCommit(false);
            try {
                 PreparedStatement pstmt = conn.prepareStatement(SQL_CREATE_ENT);
                pstmt.setInteger(1,id);
                pstmt.setString(2, (contentType.isEnable()?"YES":"NO"));
                pstmt.setInteger(3, (contentType.totalHits()));

                pstmt.executeUpdate();
                pstmt.close();

                pstmt = conn.prepareStatement(SQL_CREATE_LSENS);

                pstmt.setInt(1,id);

                Locale[] locales = contentType.getLanguages();
                for (int index=0; index<locales.length; index++) {
                    pstmt.setString(2,LocaleFormat.format(locales[index]));
                    pstmt.setString(3,contentType.getDescription(locales[index]));
                   pstmt.setInteger(4,contentType.
                               getPriority(locales[index]).getOrder());
                   pstmt.setString(5,contentType.
                               getPriority(locales[index]).getColor());
                    pstmt.executeUpdate();
                }
                pstmt.close();
                conn.commit();
            }
            catch (SQLException ex) {
                conn.rollback();
                throw new DataAccessException(ex);
            }

            contentType.setId(id);
            return id;
        }
        catch (Exception ex) {
            throw DataAccessException.buildException(ex);
        }
        finally {
            closeDBConnection(conn);
        }

}
```

Figura 10 – Operação create de ContentTypeDataAccess

11 Usos Conhecidos

Por motivos de confidencialidade, mais detalhes dos usos conhecidos abaixo não podem ser fornecidos.

- Um portal Web de acesso a informações de lista telefônica
- Um portal Web de informações sobre restaurantes e gastronomia
- Uma aplicação de Web-TV
- Uma aplicação de correio eletrônico segura na Web

12 Padrões Relacionados

- *Data Access Object* [3]:
    - *Pode ser utilizado para implementação do MultiLocaleDataAccess*
- *ValueObject* [3]:
    - Um *ValueObject* pode representar uma entidade sensível à língua e que deve ser transmitida via rede

Referências

[1] R. Cattel, D. Barry, editors. *The Object Database Standard: ODMG 93*. Morgan Kaufman, 1997.

[2] ISO/IEC JTC1/SC21 N10489, ISO//IEC 9075, Part 2, Committee Draft (CD), *Database Language SQL - Part 2: SQL/Foundation*, ftp://speckle.ncsl.nist.gov/isowg3/dbl/BASEdocs/cd-found.pdf, July, 1996.

[3] http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html

[4] http://l10n.openoffice.org/