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

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

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 (*MultiLocaleEntity pattern*) that supports internationalization and localization. It presents a solution for the business tier and business entity, the integration tier accessing data resources and the data tier.

Introdução

A larga difusão de aplicações Web e distribuídas exige cada vez o suporte de internacionalização e localização de aplicações. Internacionalização (i18n) [OPEN] é o processo de projetar uma aplicação para se adaptar a várias línguas e regiões. Localização (l10n) [OPEN] 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 [CB97], 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.

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 suportar os 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 (considerar língua como a língua propriamente dita mais a região, por exemplo, Português/Brasil).

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 suportar a persistência de dados em várias línguas?

Forças

- O acesso à informação sensível a língua (localizada) deve ser simples para fornecer boa usabilidade para as classes localizadas

[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.

- 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 aumentar 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

Solução

A solução é dividida em dois grupos: A camada de aplicação e a camada de acesso a dados.

Quanto à camada de aplicação, deve-se fornecer uma classe para representação da língua, uma classe para conter a informação da entidade para cada língua e uma classe para gerenciar a persistência da informação. A classe de entidade 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.

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*.

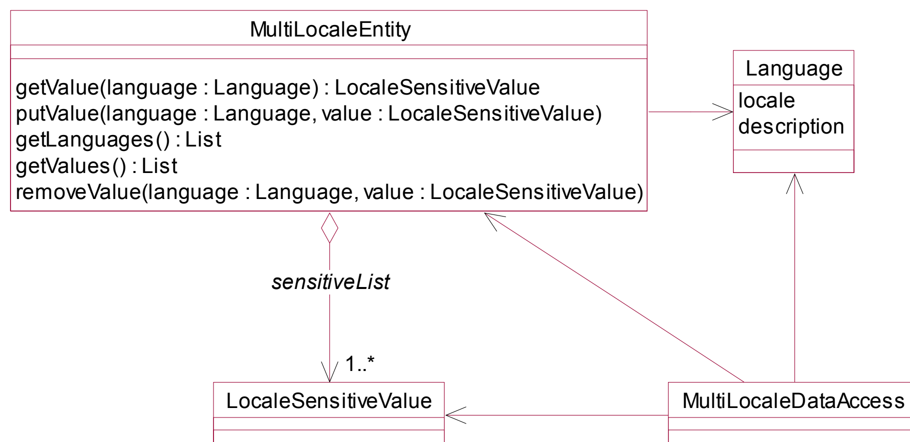

Figure 1 – Diagrama de Classes do Padrão Multi Locale Entity na camada de aplicação

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

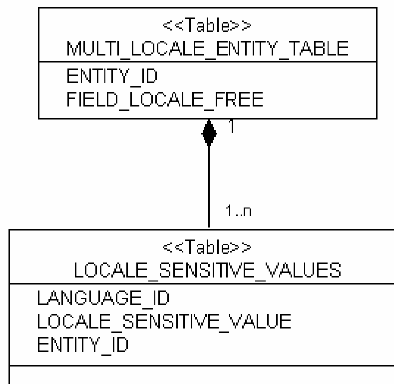

Figure 2 – Digrama de Classes do Padrão MultiLocale Entity na camada de dados

Participantes

- **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)

- **MultiLocaleDataAccess**

  - implementa o mecanismo de persistência da entidade sensível à língua.
  - *define instruções SQL [ISO96] que fazem o mapeamento da entidade em tabelas com suporte a várias línguas.*

- **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.

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* (atributos não sensíveis à língua são recuperados através de algum método *getXXX* comum)
5. O cliente atualiza algum atributo sensível à língua através de algum método *putValue*, passando o novo valor e *Language* 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*

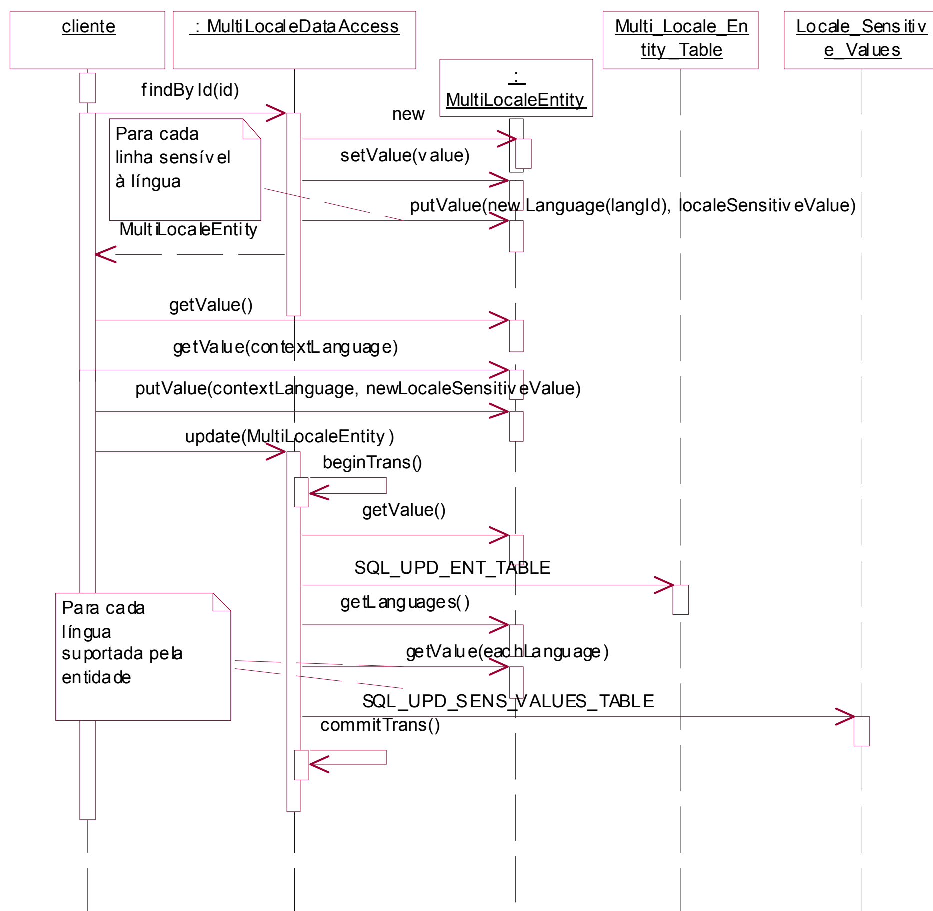

Figura 3: Consulta e atualização de entidade

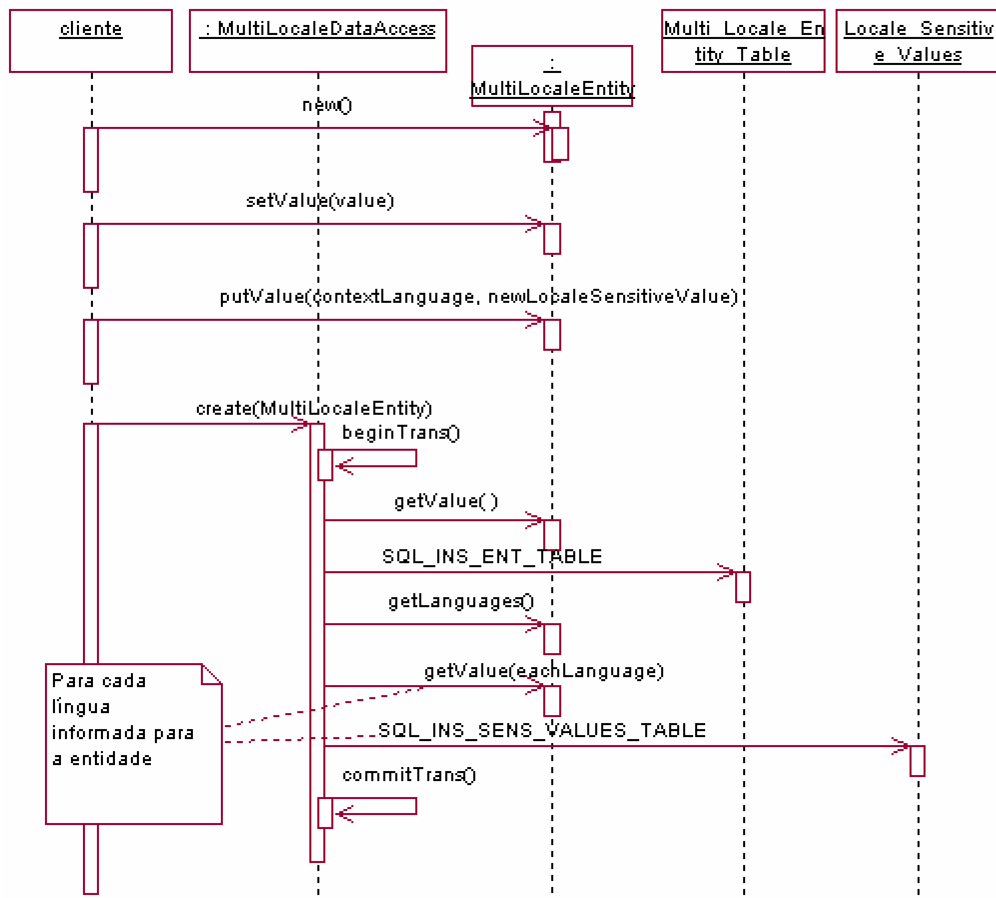

Figura 4 : Criação de entidade

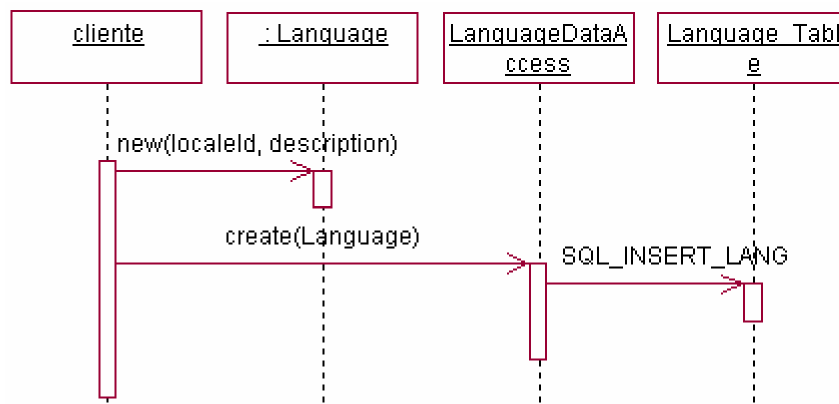

Figura 5: Adicionando uma nova língua

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

Conseqüências

- 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.

- O *cache* das informações por língua por gerar maior utilização de memória e 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

- A separação de tabelas do modelo relacional entre informações sensíveis à língua e informações insensíveis à língua fornece maior racionalização do armazenamento das informações

Implementação

O Padrão para implementação dos SQLs de persistência deve utilizar a estrutura a seguir:

SQL de criação de entidade:

– Deve-se utilizar uma instrução para criar dados na tabela livre de língua e outra para a tabela sensível a língua:

```
INSERT INTO MULTI_LOCALE_ENTITY_TABLE( ENTITY_ID, FIELD_LOCALE_FREE)
VALUES(?, ?)

INSERT INTO LOCALE_SENSITIVE_VALUES( ENTITY_ID, LANGUAGE_ID,
LOCALE_SENSITIVE_VALUE)  VALUES(?, ?, ?)
```

– O SQL de consulta por *id* deve recuperar todos os dados (sensíveis e não-sensíveis à língua)

```
SELECT MULTI_LOCALE_ENTITY_TABLE.ENTITY_ID,
MULTI_LOCALE_ENTITY_TABLE.FIELD_LOCALE_FREE,  LANGUAGE_ID,
LOCALE_SENSITIVE_VALUES.LOCALE_SENSITIVE_VALUE FROM
MULTI_LOCALE_ENTITY_TABLE, LOCALE_SENSITIVE_VALUES WHERE
MULTI_LOCALE_ENTITY_TABLE.ENTITY_ID = LOCALE_SENSITIVE_VALUES.ENTITY_ID AND
MULTI_LOCALE_ENTITY_TABLE.ENTITY_ID = ?
```

– A atualização pode ser composta de dois comandos SQL.

Padrões Relacionados

- *Data Access Object* [J2EE]:
  - Pode ser utilizado para implementação do *MultiLocaleEntity*

- *ValueObject* [J2EE]:
  - Um *ValueObject* pode representar uma entidade sensível à língua e que deve ser transmitida via rede

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.

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 entrada do portal para que o usuário possa selecionar qual tipo de conteúdo deseja ver: Esportes, Notícias, Tecnologia, Cinema, etc. Os tipos de conteúdo podem ser habilitados ou desabilitados, assim a entidade possui o atributo booleano *enabled* para indicar essa característica. Esse atributo não é sensível à língua, mas a descrição do tipo de conteúdo é sensível à língua. Assim temos duas classes *ContentType* e *ContentTypeDataAccess*. A primeira funciona como *MultiLocaleEntity* e a segunda como *MuliLocaleEntityDataAccess*. O *LocaleSensitiveValue* é apenas o tipo *String* para o atributo *description* de *ContentType*. As classes foram implementadas em *Java*. As tabelas criadas foram CONTENT_TYPE e CONTENT_TYPE_NAME representado, respectivamente, MULTI_LOCALE_ENTITY_TABLE e LOCALE_SENSITIVE_VALUES. As classes para *Language* e a tabela LANGUAGE também foram criadas.

O trecho abaixo 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.CONT_TYPE_ID,
CONTENT_TYPE.ENABLED, CONTENT_TYPE_NAME.LANGUAGE, CONTENT_TYPE_NAME.DESCRIPTION
FROM CONTENT_TYPE, CONTENT_TYPE_NAME WHERE CONT_TYPE_ID=? AND
CONTENT_TYPE.CONT_TYPE_ID = CONTENT_TYPE_NAME.CONT_TYPE_ID ";

private static final String SQL_FIND_ALL = "SELECT CONTENT_TYPE.CONT_TYPE_ID,
CONTENT_TYPE.ENABLED, CONTENT_TYPE_NAME.LANGUAGE, CONTENT_TYPE_NAME.DESCRIPTION
FROM CONTENT_TYPE, CONTENT_TYPE_NAME WHERE CONTENT_TYPE.CONT_TYPE_ID =
CONTENT_TYPE_NAME.CONT_TYPE_ID ";

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

private static final String SQL_CREATE_LSENS = "INSERT INTO CONTENT_TYPE_NAME
CONT_TYPE_ID, LANGUAGE,DESCRIPTION) VALUES (?,?,?)";

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

private static final String SQL_DELETE_LSENS = "DELETE FROM
CONTENT_TYPE_NAME WHERE CONT_TYPE_ID=?";
```

O trecho abaixo 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 se 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"));
                    list.add(ContentType);
                }

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

    }
```

O trecho abaixo 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 String 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.setInt(1,id);
                pstmt.setString(2, (contentType.isEnable()?"YES":"NO"));
                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.executeUpdate();
                }
                pstmt.close();
                conn.commit();
            }
            catch (SQLException ex) {
                conn.rollback();
                throw new DataAccessException(ex);
            }
```

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

Usos Conhecidos

- 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 eletrônica segura na Web e em Desktop

Referências

[CB97] R. Cattel, D. Barry, editors. *The Object Database Standard: ODMG 93*. Morgan Kaufman, 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[ISO96] 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.

[J2EE] http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html

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