Java SE: Aprendendo o padrão MVC

Este artigo apresenta o padrão de arquitetura chamado MVC (Model-View-Controller) utilizando a plataforma Java SE, mostrando como um diagrama de classes construído segundo esse padrão é transformado em código Java.

Artigo do tipo **Tutorial**
Recursos especiais neste artigo:
Artigo no estilo Soução Completa

Porque este artigo é útil

Este artigo apresenta o padrão de arquitetura chamado MVC (Model-View-Controller), mostrando como um diagrama de classes construído segundo esse padrão é transformado em código Java. O foco principal é dado à camada Model, que é implementada utilizando-se o padrão DAO e o banco de dados MySQL.

Os desenvolvedores que desejam aprender sobre o modelo MVC e estudantes que estão iniciando em modelagem e programação orientada a objetos encontrarão neste artigo explicações de como transformar a modelagem de um sistema orientado a objetos em código Java funcionando.

Este artigo é direcionado a iniciantes em programação que já aprenderam lógica de programação e conseguem implementar algoritmos simples na linguagem Java. A partir desse ponto deve-se aprender a modelar sistemas maiores de maneira profissional. Atualmente isso é feito através da modelagem Orientada a Objetos, utilizando, por exemplo, a UML (*Unified Modeling Language*).

A atividade de modelagem de um sistema consiste em transformar as necessidades definidas pelos clientes, documentadas nos requisitos do sistema, em uma série de representações desses requisitos e das estruturas de software que deverão ser implementadas para atendê-los. É comum construirmos o Diagrama de Casos de Uso para representar os requisitos do sistema, incluindo sua documentação para indicar os cenários de utilização. Em seguida construímos o Diagrama de Classes, que representa a estrutura do sistema em termos das entidades que ele deve manipular. Outro diagrama muito importante é o de Sequência. O objetivo de se construir esse diagrama é o de validar se as classes estão adequadas para suportar todas as interações previstas nos Casos de Uso e consequentemente revisar e melhorar o diagrama de classes. A construção de outros diagramas pode ser necessária dependendo do tipo de sistema, sua complexidade e seu tamanho, no entanto, os três considerados básicos são: casos de uso, classes e sequência.

Mesmo considerando apenas o diagrama de classes, sabemos que aprender a modelar quando ainda temos pouca experiência com desenvolvimento de software (principalmente com programação) não é tarefa fácil. E fica ainda mais difícil quando não conseguimos visualizar como aquele modelo que acabamos de construir vai ser transformado efetivamente em código. Este artigo pretende reduzir a distância entre a modelagem e a implementação, mostrando como um diagrama de classes é transformado em código.

Estudo de Caso: Sistema Bancário Simples

Vamos trabalhar sobre um exemplo para entender o processo de transformação do projeto do sistema representado em um diagrama de classes para o código fonte. Escolhemos para tanto um Sistema Bancário simples, uma vez que ele possui requisitos amplamente conhecidos. Basicamente teremos o cadastro de clientes e suas contas no banco, que podem ser dos tipos comum ou especial. Além disso, serão suportadas operações de depósito e saque sobre as contas do banco.

Veja a seguir a lista mais detalhada de requisitos:

· O sistema deve permitir que clientes fossem cadastrados, incluindo seu nome e e-mail;

· O sistema deve permitir que clientes possuíssem contas no banco. Cada cliente pode ter várias contas, e pode também não ter nenhuma conta criada;

· Cada conta do banco deve pertencer a um único cliente, não sendo permitidas contas conjuntas;

· As contas são divididas em dois tipos: comum e especial;

· As contas do tipo comum possuem um saldo e sobre elas é possível realizar saques e depósitos. Saques que venham a superar o valor do saldo da conta não são autorizados;

· As contas especiais possuem um saldo e um limite, e sobre elas é possível realizar saques, depósitos e modificação do limite da conta. Saques que superem o valor do saldo acrescido do valor do limite não são autorizados.

Não foi considerado, entre muitas outras coisas, que o sistema bancário deveria incluir vários bancos e que transferências de valores podem ser realizadas entre contas de bancos diferentes. No nosso exemplo temos modelado apenas um banco e as contas de seus clientes. Esse pequeno conjunto de requisitos levou à construção do diagrama de classes apresentado na **Figura 1**.

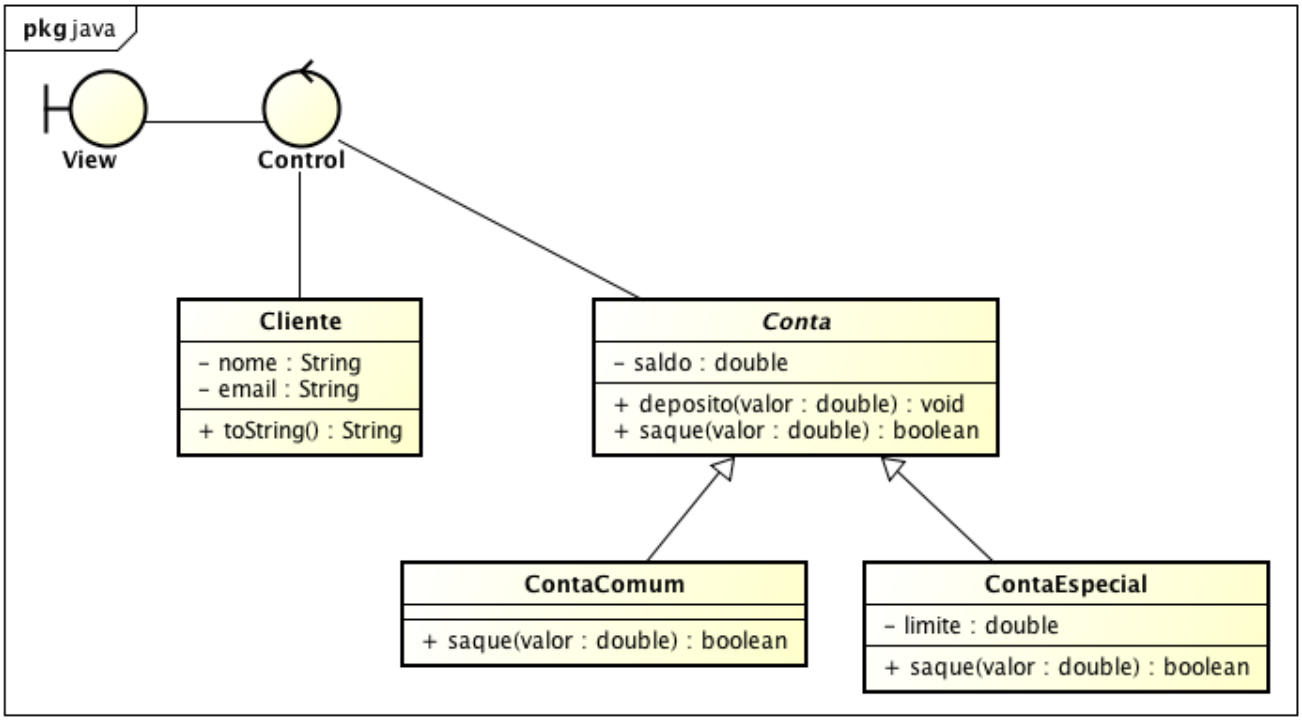

Figura 1. Diagrama de Classes do Sistema Bancário (Criado utilizando ASTAH – Community Edition).

O diagrama enfatiza a camada *Model*, composta pelas classes **Cliente**, **Conta**, **ContaComum** e **ContaEspecial**, sendo apresentados os seus atributos, os principais métodos e as associações entre classes. As duas outras camadas, *Controller* e *View*, estão apresentadas de forma resumida através dos estereótipos **Control** e **View**. Essa é a representação típica da arquitetura de um sistema organizado em três camadas, como a que é utilizada no padrão MVC (Model-View-Controller).

Ainda considerando o diagrama da **Figura 1**, alguns detalhes são importantes de se destacar sobre as classes da camada Model. A classe **Cliente** é bem simples, possuindo apenas dois atributos, o nome e o e-mail. A classe **Conta** é uma classe abstrata que contém atributos e métodos que são comuns entre os tipos de contas manipulados pelo sistema. Ela é abstrata porque nunca teremos nenhuma conta cadastrada no banco que não seja ou comum ou especial. Portanto, essa classe não será instanciada diretamente, sempre teremos instâncias de classes derivadas dela.

As classes **ContaComum** e **ContaEspecial**, por sua vez, fornecem as implementações e atributos específicos para cada um dos tipos de conta que podem ser manipuladas no nosso sistema bancário. Observe que **ContaEspecial** tem um atributo a mais para lidar com a definição do limite de crédito do cliente.

Essas são as classes da camada **Model** apresentadas em detalhes. Assim já temos um diagrama de classes que pode suportar o início da implementação. Em uma situação real, seria importante também fazer protótipos de telas da aplicação e revisar a lista de requisitos para verificar se as telas e a camada de controle vão suportar a implementação de todos os requisitos.

Implementação da Camada Model

Mesmo no nosso sistema bancário simplificado podemos ter um grande número de clientes cadastrados, cada um com várias contas. É natural se imaginar que essas informações estariam armazenadas em um banco de dados e não residindo exclusivamente em memória. A implementação da camada Model deve, portanto, suportar o carregamento de informações do banco de dados para objetos em memória e vice-versa, gravando novos dados no banco.

No entanto, os sistemas gerenciadores de banco de dados mais utilizados comercialmente são baseados no modelo relacional, ou seja, armazenam as informações em colunas de tabelas conectadas entre si através de chaves; enquanto que em nosso programa, teremos as informações encapsuladas em objetos das classes da camada *Model*. Desta forma, precisamos criar uma tradução entre esses dois mundos, chamada Mapeamento Objeto Relacional.

Existem frameworks para facilitar a implementação do mapeamento objeto relacional, como é o caso do Hibernate, que utiliza anotações específicas nas classes da camada *Model* para gerar a tradução para as tabelas do banco de dados. No entanto, para realmente entender o que está acontecendo, é melhor aprender a fazer esse mapeamento manualmente primeiro, para depois evoluir para o uso de um framework.

Um padrão muito utilizado para se fazer manualmente o mapeamento entre objetos em memória e dados armazenados em um banco de dados relacional é o **DAO** – **Data Access Object**. Nesse padrão, cada classe definida na camada de dados tem uma classe **DAO** correspondente, responsável por implementar métodos para gravar, recuperar, atualizar e apagar dados do banco de dados. Esses métodos são genericamente conhecidos como CRUD, do Inglês: *Create*, *Retrieve*, *Update* e *Delete*.

Construindo o Banco de Dados

Para suportar a necessidade de persistência de dados do projeto exemplo, criaremos um banco de dados com duas tabelas: *i*) “cliente”, que deve ser criada para armazenar os nomes, e-mails e os identificadores únicos dos clientes do banco; e, *ii*) “conta”, para armazenar as contas comuns e especiais cadastradas no sistema. Esta tabela contém o identificador do cliente ao qual a conta pertence “idCliente” (no nosso caso temos apenas um cliente para cada conta, ou seja, não existem contas conjuntas), o tipo da conta (se comum ou especial), o saldo da conta e o seu limite (preenchido somente para o caso de ser uma conta especial).

Por questões de praticidade apenas uma tabela foi utilizada para armazenar os dois tipos de conta que serão manipuladas pelo sistema. Isso facilita a criação do banco de dados e a implementação dos acessos às informações para um sistema simples como esse. No entanto, introduz uma dependência semântica que pode não ser interessante em um sistema maior; a saber: o limite da conta é uma informação que deve ser preenchida quando o tipo da conta corresponder a uma conta especial, mas não deve ser preenchido se tivermos uma conta comum. No nosso caso forçamos, para evitar problemas, que o limite seja zero no caso de se tratar de uma conta comum. Ou seja, tivemos que lidar com a dependência semântica em código, o que torna o sistema mais complexo.

A **Listagem 1** apresenta o script de criação do banco de dados com as duas tabelas descritas anteriormente. Utilizamos o MySQL como sistema gerenciador, uma vez que é robusto, simples e com versão gratuita disponível.

Listagem 1. Script de criação das tabelas do banco de dados.

```
    CREATE TABLE `cliente` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `nome` varchar(60) NOT NULL,
    `email` varchar(60) DEFAULT NULL,
    PRIMARY KEY (`id`)
  )

  CREATE TABLE `conta` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `idCliente` int(11) NOT NULL,
    `tipo` int(11) NOT NULL,
    `saldo` double NOT NULL DEFAULT '0',
    `limite` double NOT NULL DEFAULT '0',
    PRIMARY KEY (`id`),
    FOREIGN KEY (`idCliente`) REFERENCES `cliente` (`id`)
  )
```

Implementando as classes de dados da camada Model

Essa parte da implementação é relativamente rápida, uma vez que a estrutura das classes é muito parecida umas com as outras. Todas as classes dessa camada serão criadas dentro de um mesmo pacote no projeto Java. Assim, utilizando a sua IDE favorita, crie o pacote **model** e também os pacotes **view** e **controller**, uma vez que a mesma situação se aplica às duas outras camadas.

No pacote **model** vamos criar as classes que correspondem à camada de dados mostrada no diagrama da **Figura 1**: **Cliente**, **Conta**, **ContaComum** e **ContaEspecial**.

A **Listagem 2** apresenta a implementação da classe **Cliente**. Observe que os mesmos atributos que aparecem no banco de dados são refletidos nessa implementação. Além disso, um construtor que recebe todos os atributos da classe é codificado, como também os **getters** e **setters** tradicionais e o método **toString()**, seguindo assim o padrão JavaBeans.

Listagem 2. Código da classe Cliente.

```
    public class Cliente {

      private int id;
      private String nome;
      private String email;
```

```
    public Cliente(int id, String nome, String email){
        this.id = id;
        this.nome = nome;
        this.email = email;
    }

    public int getId(){return id;}
    public String getNome() {return nome;}
    public void setNome(String nome) {this.nome = nome;}
    public String getEmail() {return email;}
    public void setEmail(String email) {this.email = email;}
    public String toString(){return nome+" ["+email+"]";}

}
```

A **Listagem 3** apresenta a implementação da classe **Conta**. Alguns detalhes são muito importantes em relação à implementação dessa classe, conforme analisado a seguir. Observa-se, por exemplo, que existe o método **getId()**, mas não existe o método **setId()**. A implementação está assim de propósito, uma vez que o **id** é recuperado do banco de dados e definido no objeto a partir do construtor. Como o **id** nunca será modificado durante a execução do programa, não há a necessidade de um método **setId()**. De forma similar, o **idCliente** só possui o **getter** e não o **setter**.

Temos também um **getSaldo()**, mas não temos um **setSaldo()**. Ao invés disso, temos dois métodos que irão modificar o saldo da conta, **deposito()** e **saque()**. Essa estruturação vem do entendimento da lógica de negócios do ambiente onde o sistema irá funcionar. No contexto de um sistema bancário nunca modificamos o saldo de uma conta diretamente. As modificações são sempre realizadas através das operações de saque ou depósito.

Listagem 3. Código da classe Conta.

```
public abstract class Conta {

    private int id;
    private int idCliente;
    private double saldo;

    public enum TipoDeConta {
      Comum, Especial
    }

    public Conta(int id, int idCliente){
        this.id = id;
```

```
            this.idCliente = idCliente;
            saldo = 0;
        }

        public int getId(){return id;}
        public int getIdCliente() {return idCliente;}
        public double getSaldo(){return saldo;}

        public void deposito(double valor){
            saldo += valor;
        }

        public boolean saque(double valor){
            saldo -= valor;
            return true;
        }

    }
```

A **Listagem 4** apresenta a implementação da classe **ContaComum**, derivada de **Conta**. Como pode ser observado, seu código é muito simples. Temos apenas o construtor e o método **saque()** sobrescrito. Na classe pai esse método realiza a retirada de dinheiro da conta não importando se existe fundo ou não. A proteção para não realizar a operação caso não exista saldo suficiente só é realizada em **ContaComum**. Caso não exista, o método retorna **false**. Se existir, o método **saque()** do pai é chamado para efetivamente realizar o débito.

Ainda na **Listagem 4**, o comando ((valor>super.getSaldo())?false:super.saque(valor)) é um operador ternário que retorna **false** se a expressão que aparece antes do ponto de interrogação "**?**" for verdadeira, ou seja, se o valor a ser sacado superar o saldo. Caso o valor não supere o saldo, é executado o método **saque()** da classe pai (**super.saque(valor)**).

Listagem 4. Código da classe ContaComum.

```
    public class ContaComum extends Conta{

        public ContaComum(int id, intidCliente){
            super(id, idCliente);
        }

        public boolean saque(double valor){
            return ((valor>super.getSaldo())?false:super.saque(valor));
        }

    }
```

A **Listagem 5** apresenta o código da classe **ContaEspecial**, também derivada de **Conta**. Nessa classe é adicionado o atributo **limite**, com os *getters* e *setters* correspondentes e o método **saque()** é também sobrescrito. Observa-se que a única diferença da implementação de **saque()** fornecida por **ContaComum** e a fornecida por **ContaEspecial** é que, neste último caso, um saque pode ser executado se o valor a ser sacado não superar o saldo somado ao limite.

Listagem 5. Código da classe ContaEspecial.

```
    public class ContaEspecial extends Conta{

      private double limite;

      public ContaEspecial(int id, int idCliente, double limite){
          super(id,idCliente);
          this.limite = limite;
      }

      public double getLimite(){return limite;}
      public void setLimite(double limite){this.limite = limite;}

      public boolean saque(double valor){
          return ((valor>(super.getSaldo()
          +this.limite))?false:super.saque(valor));
      }

  }
```

Com isso as classes básicas que representam nosso domínio da informação estão implementadas. Vamos passar agora para a implementação do acesso ao banco de dados.

Implementando o acesso ao Banco de Dados

Para conseguirmos executar comandos de leitura e escrita em um banco de dados temos que inicialmente conseguir uma conexão com o banco que será utilizado, o que envolve compreender e saber utilizar uma série de conceitos. É necessário saber sobre strings de conexão, drivers de acesso ao banco, comandos específicos, etc. Entender profundamente todos esses detalhes não é o foco deste artigo, mas de qualquer forma os conceitos são apresentados resumidamente aqui para permitir sua utilização no programa exemplo apresentado.

Existem algumas funcionalidades que são comuns independentemente do banco de dados específico que esteja sendo utilizado. Em geral, separamos o código que implementa essas funcionalidades em uma classe genérica para permitir que ele seja utilizado pelas várias classes específicas derivadas dela. Esse é o caso da classe **GenericCONN** apresentada na **Listagem 6**, que implementa os métodos: **getResultSet()**, que recupera registros do banco de dados;

executeUpdate(), que insere ou atualiza registros; além do **lastId()**, que recupera o identificador do último registro inserido em uma tabela do banco.

No caso deste exemplo, usaremos apenas o banco de dados MySQL. A **Listagem 7** mostra o código de **MySQLCONN**, única classe derivada de **GenericCONN** que implementaremos. Essa classe apresenta o método **getConnection()** específico para acessar o banco de dados MySQL.

Listagem 6. Código da classe GenericCONN.

```
public abstract class GenericCONN {

  public static final String DAOName = "MySQLCONN";
  protected static Connection con;

  public abstract Connection getConnection();

  protected ResultSet getResultSet(PreparedStatement queryStatement)
   throws SQLException {
      ResultSet rs = null;
      return queryStatement.executeQuery();
  }

  protected int executeUpdate(PreparedStatement queryStatement)
  throws SQLException {
      return queryStatement.executeUpdate();
  }

  protected int lastId(String tableName, String primaryKey)
  throws SQLException {
      Statement s;
      ResultSet rs;
      int lastId = -1;
      this.getConnection();
      s = con.createStatement();
      s.executeQuery("SELECT MAX(" + primaryKey + ")
       AS \"key\" FROM " + tableName);
      rs = s.getResultSet();
      if (rs.next()) {
          lastId = rs.getInt("key");
      }
      return lastId;
  }

}
```

O próximo passo é estabelecer o acesso ao banco de dados. Para isso, precisamos definir a **String** de conexão, o nome de usuário, a senha e importar o driver JDBC para o projeto. Em geral ele é disponibilizado no site do fornecedor do gerenciador de banco de dados que estiver sendo utilizado. No caso do MySQL, o driver é chamado de Connector/J (veja a seção **Links**). Para importar o driver no NetBeans, por exemplo, após fazer o download do driver (em geral um arquivo .jar), clique com o botão direito sobre o nome do projeto e acesse o item *Propriedades*. Em seguida, clique em *Bibliotecas* e em *Adicionar JAR/Pastas*. Procure então pelo .jar que contém o driver e confirme a importação do arquivo.

Feito isso, precisamos implementar uma classe derivada de **GenericCONN** para fazer o acesso específico ao banco MySQL. Na **Listagem 7**, conforme citado anteriormente, temos a classe **MySQLCONN**. Ela fornece a implementação para o método **getConnection()**, que estabelece uma conexão com o banco de dados e a atribui à variável **con** (atributo da classe pai). Para tanto é utilizada uma chamada a **getConnection()** da classe **DriverManager**. Este procura por um driver JDBC adequado para acessar o banco de dados definido na **String** de conexão **DBURL**, utilizando-o para estabelecer a conexão com o banco.

Listagem 7. Código da classe MySQLCONN.

```
    public class MySQLCONN extends GenericCONN {

      private static final String DBURL =
      "jdbc:mysql://localhost:3306/DBbanco";
      private static final String user = "vilela";
      private static final String pass = "vilela";

      public Connection getConnection() {
          if (con == null) {
              try {
                  con = DriverManager.getConnection(DBURL, user, pass);
              } catch (Exception e) {
                 System.err.println("Não foi possível conectar
                  com o banco de dados.");
        }
          }
          return con;
      }

  }
```

Essas duas classes implementam recursos genéricos que podem ser reutilizados na maioria de seus projetos. **GenericCONN** pode ser reusada praticamente sem nenhuma modificação, uma vez que foi construída para ser genérica. A classe **MySQLCONN** pode ser utilizada diretamente

se o banco de dados for o MySQL. Caso não seja, na maioria dos casos a única coisa que precisa ser modificada é a **String** de conexão que está na variável **DBURL**.

As classes que serão vistas a seguir utilizam o método **getConnection()** de **MySQLCONN** para estabelecer uma conexão com o banco e executar os comandos de escrita e gravação. Apresentaremos essas classes seguindo o padrão DAO – *Data Access Object*.

Implementando o Padrão DAO – Data Access Object

Para implementar o padrão DAO, para cada classe da camada *model* será criada outra de acesso ao banco associada a ela. Por padrão elas recebem o nome da original acrescidos da terminação DAO. Portanto, teremos **ClienteDAO** e **ContaDAO**, coletivamente chamadas de classes DAO, além das classes originais **Cliente** e **Conta** que continuam existindo. Novamente temos a situação onde algumas funcionalidades são reusadas por todas as DAO. Assim, uma classe abstrata **GenericDAO** foi implementada para encapsular o código que é comum.

A **Listagem 8** apresenta a **GenericDAO**. No seu construtor utilizamos o mecanismo de reflexão (que permite criar uma instância de uma classe a partir do seu nome) para carregar uma instância da classe de conexão com o banco de dados e estabelecer a conexão. A classe que será carregada depende do nome definido para a variável **DAOName** em **GenericCONN** – **Listagem 6**. Alterando o valor dessa variável podemos carregar classes diferentes e assim acessar gerenciadores de banco de dados diferentes quando necessário.

Na classe **GenericCONN** um método abstrato **buildObject()** é definido. Ele deve montar um objeto a partir de um **ResultSet** e é abstrato pois a instanciação de objetos depende da DAO específica que estiver executando. Portanto, cada classe filha deve oferecer a sua própria implementação para esse método.

Define-se também o método **retrieveListOfObjects()**, que recebe uma **query** e a envia para o banco de dados recebendo de volta uma instância de **ResultSet** que encapsula as linhas recuperadas do banco correspondendo à **query** executada. O método então percorre essas linhas invocando **buildObject()** para instanciar os objetos adequados à consulta realizada e inseri-los em um **ArrayList**. Por exemplo, esse método é chamado em **ClienteDAO** com uma query do tipo: “SELECT * FROM clientes” para recuperar todos os campos de todos os clientes. O **buildObject()** definido em **ClienteDAO** sabe processar uma **ResultSet** e instanciar objetos de **Cliente**, como expõe a **Listagem 9**.

Listagem 8. Código da classe GenericDAO.

```
public abstract class GenericDAO {

  protected GenericCONN con;

  protected GenericDAO() {
      Class<?> daoClass = Class.forName
      ("model." + GenericCONN.DAOName);
```

```
            con = (GenericCONN) daoClass.newInstance();
            con.getConnection();
        }

        protected abstract Object buildObject(ResultSet rs);

        protected ArrayList<Object>
         retrieveListOfObjects(String query) throws SQLExpection{
            PreparedStatement stmt;
            List<Object> list = new ArrayList<Object>();
            ResultSet rs;
            stmt = con.getConnection().prepareStatement(query);
            rs = con.getResultSet(stmt);
            while (rs.next()) {
                list.add(buildObject(rs));
            }
            rs.close();
            return list;
        }

        protected Object retrieveById(int id, String tableName)
         throws SQLExpection{
            PreparedStatement stmt;
            Object obj = null;
            ResultSet rs;
            stmt = con.getConnection().prepareStatement
            ("SELECT * FROM " + tableName + " where id=?");
            stmt.setInt(1, id);
            rs = con.getResultSet(stmt);
            if (rs.next()) {
                obj = buildObject(rs);
            }
            rs.close();
            return obj;
        }

        protected Object retrieveLastId(String tableName) {
            int id = con.lastId(tableName, "id");
            return retrieveById(id, tableName);
        }

        protected void delete(int id, String tableName)
          throws SQLExpection{
```

```
            PreparedStatement stmt;
            stmt = con.getConnection().prepareStatement
            ("DELETE FROM "+tableName+" WHERE id = ?");
            stmt.setInt(1, id);
            con.executeUpdate(stmt);
        }

    }
```

Os dois próximos métodos retornam apenas um objeto. O método **retrieveById()** retorna o objeto correspondente ao **id** enviado como parâmetro e o **retrieveLastId()** recupera o objeto com o maior **id** inserido em uma dada tabela. O último método apresentado é o **delete()**, que remove a linha correspondente ao **id** passado como parâmetro na tabela indicada.

Todos esses métodos serão utilizados nas classes DAO específicas, como por exemplo, **ClienteDAO** que implementa o CRUD para recuperar **Clientes** do banco de dados e **ContaDAO**, utilizada tanto para instâncias de **ContaComum** como de **ContaEspecial**.

Vale lembrar que não existem instâncias de **Conta**, uma vez que **Conta** é uma classe abstrata. Veremos a seguir a implementação de **ClienteDAO** e **ContaDAO**.

Listagem 9. Código da classe ClienteDAO.

```
    public class ClienteDAO extends GenericDAO {

        private static ClienteDAO instance;

        private ClienteDAO() {
            super();
        }

        public static ClienteDAO getInstance() {
            if (instance == null) {
                instance = new ClienteDAO();
            }
            return instance;
        }

        protected Object buildObject(ResultSet rs) throws SQLException{
            Cliente obj = null;
            obj = new Cliente(rs.getInt("id"), rs.getString("nome"),
            rs.getString("email"));
            return obj;
        }
```

```
        public void create(String nome, String email) throws SQLException{
            PreparedStatement stmt;
            stmt=con.getConnection().prepareStatement("INSERT INTO cliente
            (nome, email) VALUES (?,?)");
            stmt.setString(1, nome);
            stmt.setString(2, email);
            con.executeUpdate(stmt);
        }

        public List<Object> retrieveAll() {
            return retrieveListOfObjects
             ("SELECT * FROM cliente ORDER BY nome");
        }

       public List<Object> retrieveLike(String nome) {
           return retrieveListOfObjects
           ("SELECT * FROM cliente WHERE nome LIKE '%"+nome+"%'");
       }

       public Cliente retrieveById(int id) {
           return (Cliente) retrieveById(id, "cliente");
       }

       public Cliente retrieveLastId() {
           return (Cliente) retrieveLastId("cliente");
        }

        public boolean update(Cliente cliente) throws SQLException{
            PreparedStatement stmt;
            stmt=con.getConnection().prepareStatement
            ("UPDATE cliente SET nome=?,email=? WHERE id=?");
            stmt.setString(1, cliente.getNome());
            stmt.setString(2, cliente.getEmail());
            return (myCONN.executeUpdate(stmt) == 1);
        }

        public void delete(Conta conta) {
            this.delete(conta.getId(), "cliente");
        }

    }
```

A **Listagem 9** apresenta a implementação da classe **ClienteDAO**. Ela é basicamente dividida em duas partes. A primeira implementa o padrão **Singleton**, uma vez que só será utilizada uma única instância de **ClienteDAO** no programa. Isso é conseguido através do uso de um construtor privado e de um método estático chamado **getInstance()**, que retorna sempre a mesma instância dessa classe. O próximo método a ser implementado corresponde ao método abstrato requerido pela classe pai **GenericDAO**, da qual **ClienteDAO** é filha, ou seja, é fornecida uma implementação para o método **buildObject()**. Neste caso é instanciado um **Cliente** a partir das informações do **ResultSet**, obtido de uma consulta ao banco de dados. Assim estamos transformando uma linha do banco em um objeto do nosso sistema. Essa é basicamente a implementação do mapeamento objeto-relacional que foi discutido no início do artigo.

Os próximos métodos implementam o CRUD. O método **create()** recebe as informações de um novo cliente que deve ser cadastrado no banco e envia essas informações para a tabela adequada. Observe que o **id** do novo cliente não é enviado, pois estamos usando o incremento automático da chave primária do MySQL. Observe a **Listagem 1**.

Quando realizamos leituras de dados a partir de um banco, existe a possibilidade de definirmos diversos filtros. Por exemplo, podemos recuperar todos os clientes cadastrados, ou somente os clientes com nomes começados por “P”, ou o cliente que tenha o id = 56. Esses filtros são implementados nos métodos denominados coletivamente de *retrieves*. Dito isso, continuando na apresentação da classe **ContaDAO**, em seguida tem-se a implementação de quatro desses *retrieves*. Todos utilizam os métodos já definidos na classe pai, alterando apenas a query específica que deve ser executada ou o nome da tabela que deve ser consultada.

Por fim, são implementados os métodos **update()** e **delete()**. No caso do **update()** não é utilizado nenhum método da classe pai. O código necessário ao **update()** é muito específico para cada classe **DAO**, portanto não houve como criar nenhum método de uso genérico na classe pai. Já o método **delete()** se utiliza de um método já criado na classe pai, identificando apenas o nome da tabela específica que terá um elemento apagado.

A última classe apresentada na camada model é a **ContaDAO**, conteúdo da **Listagem 10**.

Listagem 10. Código da classe ContaDAO.

```
public class ContaDAO extends GenericDAO {

  private static ContaDAO instance;

  private ContaDAO() {
      super();
  }

  public static ContaDAO getInstance() {
      if (instance == null) {
          instance = new ContaDAO();
```

```
        }
        return instance;
    }

    public void create(int idCliente) {
        this.create(TipoDeConta.Comum, idCliente, 0);
    }

    public void create(int idCliente, double limite) {
        this.create(TipoDeConta.Especial, idCliente, limite);
    }

    private void create(int tipo, int idCliente, double limite)
     throws SQLException{
        PreparedStatements;
        String str;
        str = new String("INSERT INTO conta
        (tipo,idCliente,saldo,limite) VALUES(?,?,?,?)");
        s=con.getConnection().prepareStatement(str);
        s.setInt(1, tipo);
        s.setInt(2, idCliente);
        s.setDouble(3, 0);
        s.setDouble(4, limite);
        con.executeUpdate(stmt);
    }

    protected Object buildObject(ResultSetrs) throws SQLException{
        Conta obj = null;
        switch (rs.getInt("tipo")) {
        case 0:
            obj=new ContaComum(rs.getInt("id"), rs.getInt("idCliente"));
            break;
        case 1:
            obj=new ContaEspecial(rs.getInt("id"),rs.getInt("idCliente"),
            rs.getDouble("limite"));
            break;
        }
        if (obj != null) {
            obj.deposito(rs.getDouble("saldo"));
        }
        Return obj;
    }
```

```
    public ArrayList<Object> retrieveAll() {
     return retrieveListOfObjects("SELECT * FROM conta");}

    public ArrayList<Object> retrieveByIdCliente(int idCliente) {
        return retrieveListOfObjects("SELECT * FROM conta
         WHERE idCliente = " + idCliente);
    }

    public ContaretrieveById(int id) {return (Conta) retrieveById
     (id, "conta");}
    public Conta retrieveLastId() {return (Conta) retrieveLastId
      ("conta");}

    public boolean update(Conta conta) throws SQLException{
        PreparedStatement s;
        if (conta instanceof ContaComum) {
            s=con.getConnection().prepareStatement
            ("UPDATE conta SET saldo=? WHERE id = ?");
            s.setDouble(1,conta.getSaldo());s.setInt(2,conta.getId());
        } else {
            s=con.getConnection().prepareStatement("UPDATE conta
            SET saldo=?,limite=? WHERE id=?");
            s.setDouble(1,conta.getSaldo());s.setDouble(2,
            ((ContaEspecial) conta).getLimite());
            s.setInt(3,conta.getId());
        }
        int update = con.executeUpdate(stmt);
        if (update == 1) return true;
        return false;
    }

    public void delete(Conta conta) {
        this.delete(conta.getId(), "conta");
    }

}
```

ContaDAO segue o mesmo padrão definido em **ClienteDAO**. Deste modo, também é feita a implementação do padrão Singleton logo no início da definição da classe e em seguida o método **buildObject()** e os métodos que implementam o **CRUD**.

A maior diferença é que **ContaDAO** não monta objetos de **Conta** e sim objetos de **ContaComum** ou **ContaEspecial**, dependendo das informações carregadas do banco de dados. Além disso, ela também deve inserir corretamente o código do tipo de conta, dependendo de qual conta estiver

sendo criada no banco de dados. Isso é feito através dos métodos sobrecarregados **create(int idCliente)**, que cria uma conta comum ao invocar o método "**this.create(TipoDeConta.Comum, idCliente, 0);**", ou seja, o tipo da conta é Comum e, portanto, seu limite é zero, e **create(int idCliente, double limite)**, que cria uma conta especial através de uma chamada ao método "**this.create(TipoDeConta.Especial, idCliente, limite);**", especificando que o tipo da conta é Especial e o limite é definido pela variável recebida como parâmetro. Assim finalizamos a implementação da camada *model*.

Considerações finais sobre a Camada Model

A camada **Model** possui uma hierarquia de classes definida por **Conta**, **ContaComum** e **ContaEspecial**, conforme o diagrama apresentado na **Figura 1**. Essa hierarquia foi reproduzida na implementação em Java, no entanto não houve necessidade de serem criadas tabelas separadas no banco de dados. O mapeamento objeto relacional, discutido na seção que descreve a camada Model e implementado na classe **ContaDAO**, é o responsável por instanciar os objetos corretos de **ContaComum** ou **ContaEspecial** a partir dos dados carregados de uma única tabela do banco de dados. Com relação à associação entre **Cliente** e **Conta**, temos que cada conta pertence a um único cliente. Portanto, a associação entre essas classes pode ser implementada pela presença do atributo **idCliente** na classe **Conta**.

Ainda em relação ao mapeamento objeto relacional, observe que ao carregar um **Cliente** do banco de dados pela classe **ClienteDAO**, não são carregadas as contas correspondentes a esse **Cliente**. Elas só serão recuperadas do banco quando for realmente necessário. A leitura das contas de um dado cliente é feita através de uma chamada ao método **retrieveByIdCliente()** de **ContaDAO**. Essa abordagem reduz o uso de memória e melhora o desempenho do programa, uma vez que apenas as informações realmente necessárias serão carregadas do banco de dados.

Implementação da Camada Controller

A camada Controller é a responsável por implementar a lógica de negócios do sistema. É ela que faz a interligação entre a interface com o usuário e os dados da camada Model. No nosso exemplo essa camada é implementada em uma única classe, chamada **Control** e apresentada na **Listagem 11**. Essa implementação é bem simples. Ela recebe informações da **View** e faz as chamadas adequadas à camada **Model** obtendo as informações requeridas e repassando de volta para a **View**. Em sistemas mais complexos essa camada ganha uma participação mais intensa, podendo até mesmo dispor de várias classes. Por exemplo, em um sistema maior poderíamos ter uma classe controller apenas para gerenciar clientes e outra somente para contas.

A classe **Control** possui três métodos para o controle de clientes e quatro para o de contas. No controle de clientes é possível adicionar um novo cliente, recuperar todos os clientes cadastrados ou recuperar uma lista de clientes que tenham o nome parecido com uma dada **String**. No controle de contas é possível adicionar uma conta ao banco, comum ou especial, recuperar todas as contas de um cliente (identificado pelo seu **id**) e fazer o **update** de uma conta (por exemplo, modificando seu limite ou mesmo atualizando seu saldo no banco de dados).

Listagem 11. Código da classe Control – Implementação da Camada de Controle.

```
public class Control {

  // Control Clientes:
  public static void addCliente(String nome, String email){
      ClienteDAO.getInstance().create(nome, email);
  }

  public static ArrayList<Object> getAllClientes(){
      return ClienteDAO.getInstance().retrieveAll();
  }

  public static ArrayList<Object> getAllClientesLike(String nome){
      return ClienteDAO.getInstance().retrieveLike(nome);
  }

  // Control Contas:
  public static void addContaComum(int idCliente){
      ContaDAO.getInstance().create(idCliente);
  }

  public static void addContaEspecial(int idCliente, double limite){
      ContaDAO.getInstance().create(idCliente, limite);
  }

  public static ArrayList<Object>
    getAllContasByCliente(intidCliente){
      return ContaDAO.getInstance().retrieveByIdCliente(idCliente);
  }

  public static void updateConta(Conta conta){
      ContaDAO.getInstance().update(conta);
  }

}
```

As classes DAO definidas na camada Model implementam o padrão Singleton para fornecer instâncias únicas das classes de acesso a dados. Na classe **Control** podemos observar como essas instâncias são obtidas. Por exemplo, se precisarmos acessar os métodos disponíveis em **ClienteDAO**, fazemos uma chamada ao seu método estático **getInstance()**, que retorna o único objeto dessa classe disponível no sistema. A partir daí podemos invocar os métodos que desejarmos.

O mesmo acontece para a manipulação de contas. A classe **Control** fornece para a camada *View* dois métodos com nomes diferentes, **addContaComum()** e **addContaEspecial()**, um para criar contas comuns e outro para criar contas especiais. Esses dois métodos utilizam o **create()** de **ContaDAO** para efetivamente criar a conta no banco de dados, mas cada um usa uma versão diferente desse método, como pode ser visto na **Listagem 10**. Um dos métodos recebe apenas o identificador do cliente e cria uma conta comum e o outro recebe também o limite de crédito da conta e cria uma conta especial.

Implementação da Camada View

A última parte que deve ser implementada para concluirmos a construção do sistema é a camada *View*. Ela é dividida em duas partes: uma visual, que define a aparência e o posicionamento dos vários componentes de interface disponíveis para o usuário, e outra formada pelo código fonte associado aos componentes visuais, onde é implementado o comportamento desses componentes em resposta às ações dos usuários sobre eles.

Iniciamos a construção desta camada pelo projeto visual da interface gráfica, como o que é apresentado na **Figura 2**. Esse projeto visual foi desenvolvido com a IDE NetBeans versão 7.2, que permite a construção da interface gráfica a partir de componentes que podem ser arrastados e posicionados conforme a necessidade do desenvolvedor. O processo de construção dessa interface gráfica é relativamente trabalhoso, mas não necessariamente difícil. Assim, explicar detalhadamente esse processo está fora do escopo deste artigo. Serão apresentados apenas alguns detalhes da implementação como forma de ilustrar o uso da camada *Controller* dentro da *View*.

Nesse momento é importante destacar que no padrão MVC recomenda-se que a camada *View* não tenha acesso direto à *Model*. Os acessos a esta camada devem ser feitos através dos métodos disponibilizados pela camada *Controller*.

A **Listagem 12** mostra o código do tratamento de eventos do tipo **KeyReleased** no **jTextField1** (usado para filtrar a lista de clientes cadastrados no sistema). A partir desse evento, toda vez que uma tecla é pressionada e solta na caixa de texto um evento é disparado fazendo com que esse método seja executado. Este método verifica inicialmente se a caixa de texto está vazia. Se estiver, o botão **jButton1** (que equivale ao “OK” na **Figura 2**) é desabilitado e a **jList1** (lista de cliente, que está do lado esquerdo da tela) é carregada com todos os clientes cadastrados no banco de dados. Essa lista é obtida através de uma chamada ao método **getAllClientes()** na camada **Controller**. Como esse é um método estático, não há necessidade de se instanciar um objeto **Control** para poder chamá-lo.

Listagem 12. Carregando dados na View.

```
    private void jTextField1KeyReleased(java.awt.event.KeyEventevt) {
      if (jTextField1.getText().equals("")) {
          jButton1.setEnabled(false);
          loadListModel(jList1, Control.getAllClientes());
```

```
        } else {
            jButton1.setEnabled(true);
            if (evt.getKeyChar() == java.awt.event.KeyEvent.VK_ENTER) {
                loadListModel(jList1,
                 Control.getAllClientesLike(jTextField1.getText()));
            }
        }
    }
```

Ainda na **Listagem 12**, caso o campo de texto não esteja vazio e a tecla *ENTER* tenha sido pressionada, é feita uma chamada ao método **getAllClientesLike()** da **Controller**, recuperando do banco de dados todos os clientes que tenham o nome parecido com o texto inserido na **jTextField1**. Essa lista é carregada no **jList1** através de uma chamada a **loadListModel()**, método esse que é apresentado na **Listagem 13** e, basicamente, percorre o **ArrayList** recuperado do banco inserindo cada elemento no **DefaultListModel** do **JList**.

Listagem 13. Carregando dados na JList.

```
    private void loadListModel(JList list,
      ArrayList<Object>vObjects) {
      DefaultListModel listModel =
      (DefaultListModel) list.getModel();
      listModel.clear();
      for (Object o : vObjects) {
          listModel.addElement(o);
      }
  }
```

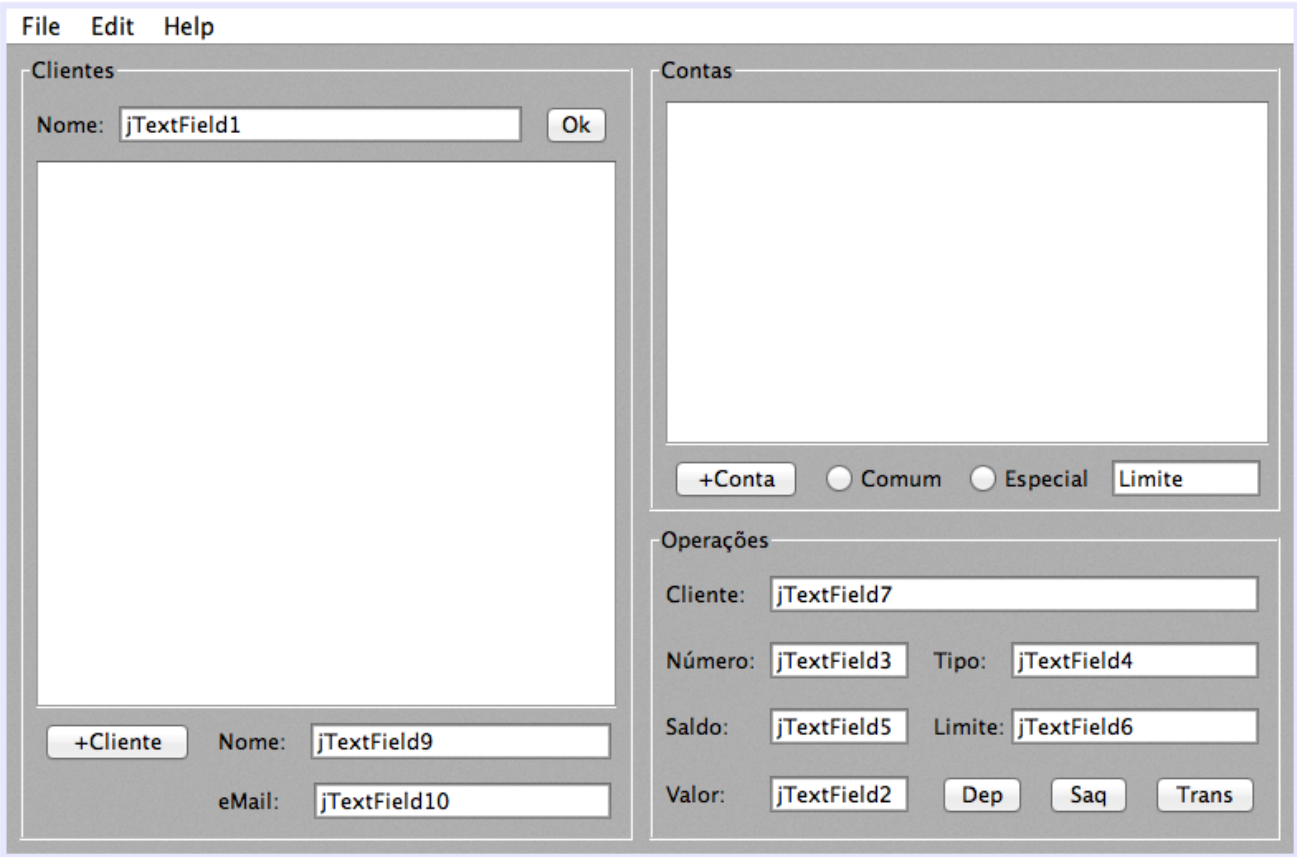

Figura 2. Interface Gráfica – Design da Camada View (criado utilizando o NetBeans 7.2).

A construção da interface gráfica é uma das partes mais demoradas de um projeto de desenvolvimento Java para desktop. O uso de ferramentas como a IDE NetBeans pode acelerar o processo, mas ainda assim ele continua demorado.

Mesmo para um projeto pequeno como este do artigo, o desenvolvimento da interface tomaria quase 60% do tempo total de desenvolvimento do projeto. Portanto, é importante planejar com antecedência as telas que serão construídas para evitar o retrabalho e aos poucos desenvolver suas próprias ferramentas (APIs, classes úteis que podem ser reutilizadas em outros projetos, etc.) para acelerar o desenvolvimento.

Uma dica importante para quem está começando é procurar ao máximo manter o código da *View* livre de implementações relativas à lógica de negócios. Esse tipo de código deve ficar na camada *Controller*. A *View* deve lidar única e exclusivamente com a realização da interface gráfica.

Conclusões

A estruturação do projeto no padrão MVC permite que uma camada seja substituída sem a necessidade de modificações radicais nas outras partes do sistema. No exemplo apresentado, poderíamos modificar o banco de dados utilizado na camada Model praticamente sem nenhuma alteração no restante do código. Para isso precisaríamos apenas implementar outras classes derivadas de **GenericCONN**, uma para cada novo banco de dados suportado pelo sistema, similares a **MySQLCONN**, mudando apenas a string de conexão, drivers e informações de acesso (usuário e senha).

A *View* também poderia ser substituída por uma versão Web, por exemplo, mantendo-se intactas as camadas *Controller* e *Model*, lembrando que o desenvolvimento dessa parte do sistema é demorado e requer planejamento. Uma boa sugestão é desenvolver um protótipo das telas antes de definir como será a versão final e só depois de testadas várias opções iniciar o desenvolvimento propriamente dito.

O padrão DAO implementado facilita a estruturação da camada *Model*, permitindo que o acesso aos dados seja feito de uma maneira organizada e com menos possibilidades de erros de codificação. Uma vez que o mecanismo de construção das classes desse padrão esteja dominado, o desenvolvimento dessa camada passa a ser bem rápido.

Analisando novamente o diagrama de classes da **Figura 1**, podemos observar que ele foi implementado fielmente. As classes adicionais são auxiliares para permitir o mapeamento entre os objetos do sistema e as tabelas do banco de dados.

Com isso chegamos ao objetivo do artigo que era o de mostrar em detalhes o processo de transformação de um modelo, representado em um Diagrama de Classes, em uma implementação Java.

Links

Gerenciador de banco de dados MySQL
http://www.mysql.com

Connector/J - Driver para acessar o MySQL via Java
http://www.mysql.com/downloads/connector

Ambiente integrado de desenvolvimento NetBeans
http://www.netbeans.org

Livros

UML 2 Uma Abordagem Prática, Gilleanes Guedes, Novatec, 2009
Livro sobre a linguagem de modelagem UML com vários exemplos práticos.