Desenvolvendo uma aplicação pas so a passo

O artigo aborda a criação passo a passo de uma pequena aplicação GUI com acesso a banco de dados. Veremos como são criadas as classes do domínio da aplicação e as classes necessárias para a persistência.

Demais posts desta série:

Desenvolvendo uma aplicação passo a passo – Parte 3

Do que se trata o artigo

O artigo aborda a criação passo a passo de uma pequena aplicação GUI com acesso a banco de dados. Nesta primeira parte são criadas as classes do domínio da aplicação e as classes necessárias para a persistência.

Em que situação o tema é útil

O tema abordado é fundamental para quem precisa criar aplicações com GUI e que implementam acesso a banco de dados. Além disso, é útil para o desenvolvimento em camadas, que neste caso é a interface com o usuário, as classes do domínio de negócio e a camada de persistência.

Desenvolvendo uma aplicação passo a passo

A maneira de abordar a solução de um problema de desenvolvimento se inicia com o entendimento da questão, passando pela criação do modelo do domínio do problema e do diagrama de classes. Nessa etapa, um dos passos, caso a aplicação necessite implementar persistência, consiste em fazer o mapeamento objeto-relacional. Após isso, finalmente inicia-se a codificação, que é a última etapa do processo, antes de efetuar os testes junto com o usuário. Assim, nesta primeira parte da série de artigos buscamos apresentar um problema, cuja solução segue esses passos.

Na Easy Java Magazine nº 4 foi proposta pelo autor uma metodologia simples para resolver problemas de programação, no artigo "Solucionando Problemas usando Java". Na matéria que ora iniciamos será apresentada uma aplicação mais complexa que os exemplos usados no artigo citado, que demanda uma metodologia mais adequada à sua solução. No entanto, em linhas gerais, o processo é similar e tentaremos acompanha-lo, visto que foge do escopo deste artigo a utilização de processos estudados na Engenharia de Software.

Na implementação do programa optou-se por criar uma interface gráfica utilizando a API Swing, à qual já foi dedicado o artigo "Swing: O Desktop Java" na Esay Java Magazine nº 12. Com o objetivo de facilitar a criação das telas será empregado o *plug-in WindowBuilder*, disponível para o Eclipse. Entretanto, se o leitor tiver habilidade com o NetBeans, este também poderá ser usado. É importante observar que ambas as tecnologias já foram objeto de matérias também nesta revista.

A camada de persistência foi desenvolvida usando uma solução apresentada na Edição nº 4 da Easy Java, no artigo "Solucionando problemas usando Java". Poderíamos ter optado por usar o padrão DAO (*Data Access Object*) ou um *framework* tal como Hibernate, mas isso traria uma complexidade maior ao desenvolvimento. Todavia, a solução adotada ainda consegue uma separação entre as camadas de persistência, regras de negócio e interface gráfica.

Seguindo esta linha, inicialmente será apresentado o problema. Depois disso será iniciado o processo de solução, começando com a análise do problema. Após a análise, o problema será dividido em partes que possam ser solucionadas individualmente e depois integradas na solução final. Em seguida será implementada cada uma das partes nas quais o problema foi separado.

Nesta primeira parte da matéria serão criadas as classes do domínio do problema e as classes necessárias para implementar a persistência.

Definição do Problema

É um fato corriqueiro que pessoas que possuem uma biblioteca particular emprestem livros aos amigos. E infelizmente é bastante comum que os amigos dificilmente devolvam as obras, e as pessoas terminam mesmo por esquecer a quem foram emprestadas. Pensando nisso, o autor propõe o desenvolvimento de um pequeno aplicativo que registre as obras mantidas em uma biblioteca pessoal. E, para evitar o esquecimento, que os dados dos empréstimos sejam também registrados.

Análise do Problema

O primeiro passo na análise é identificar o domínio do problema, que consiste em definir as classes e seus relacionamentos. A partir da descrição do problema, pode-se inferir que existem as entidades **Livro** e **Emprestimo**. Sabendo-se que um **Livro** possui pelo menos um autor, determinaremos também como parte do domínio, a entidade **Autor**. Quanto aos relacionamentos, pode-se afirmar que um livro tem um ou mais autores e um autor pode escrever mais de um livro. Define-se também que um livro pode ser emprestado mais de uma vez. Dessa maneira criamos um modelo do domínio, mostrado na **Figura 1**.

Utilizamos a UML (*Unified Modeling Language*) para criar o diagrama de classes que irá representar o modelo do domínio do problema. Assim, conforme a **Figura 1**, foram representadas as classes, seus relacionamentos e multiplicidades. Note que a multiplicidade do relacionamento entre **Autor** e **Livro** determina que um autor pode ter um ou mais livros publicados e um livro deve ter pelo menos um autor. Um livro pode ter um ou muitos empréstimos e um empréstimo é feito para um único livro.

O passo seguinte consiste em definir os dados relevantes a cada uma das classes identificadas. Na classe **Autor**, por exemplo, consideramos como importante o **nome** do mesmo. Na classe **Livro**, identificamos os campos **isbn** (um número identificador único para livros), **titulo**, **ano de edição**, **edição** e **editora**. **Nome do emprestador**, **data do empréstimo** e **data da devolução** são dados importantes a serem registrados na classe **Emprestimo**.

Na etapa seguinte do processo será apresentada uma possível solução. É o que podemos chamar de um projeto.

Solução Proposta

Antes de projetar a solução do problema deve-se pensar na persistência dos dados, e uma alternativa muito utilizada pelos desenvolvedores é o padrão DAO (*Data Access Object*). DAO é um padrão de persistência que permite separar as regras de negócio das regras de acesso ao banco de dados. Ele funciona tanto para pegar os objetos Java, e enviá-los ao banco de dados usando comandos SQL, quanto para buscar os dados do banco e convertê-los em objetos que possam ser tratados pela aplicação.

Para se conseguir implementar esse padrão, uma das tarefas do processo é mapear as classes para entidades do modelo relacional. Mas falaremos sobre isso com mais detalhes em seguida. Por ora apenas consideramos necessário fazer uma modificação no modelo do domínio, para que o mapeamento seja feito adequadamente. A modificação consiste em mudar o relacionamento entre **Autor** e **Livro** para agregação, e adicionar um atributo do tipo **Set** para receber os autores na classe **Livro**. Essa modificação visa facilitar a implementação da solução, pois um relacionamento muitos-para-muitos requer a criação de uma entidade associativa no banco de dados, para representar o relacionamento. Dessa forma, quando salvarmos os dados de um **Livro**, a classe persistente se encarregará também de salvar os autores na tabela associativa. Isso ficará mais claro quando forem apresentadas as implementações das classes persistentes. Na **Figura 2** mostramos esse novo modelo, já incluídos os atributos.

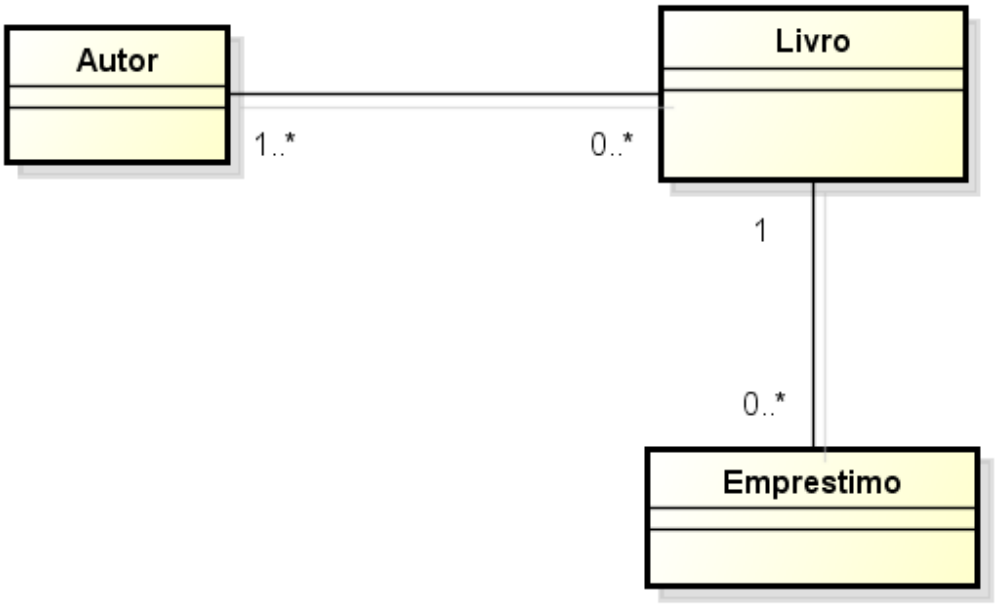


Figura 1. Modelo do domínio do problema.

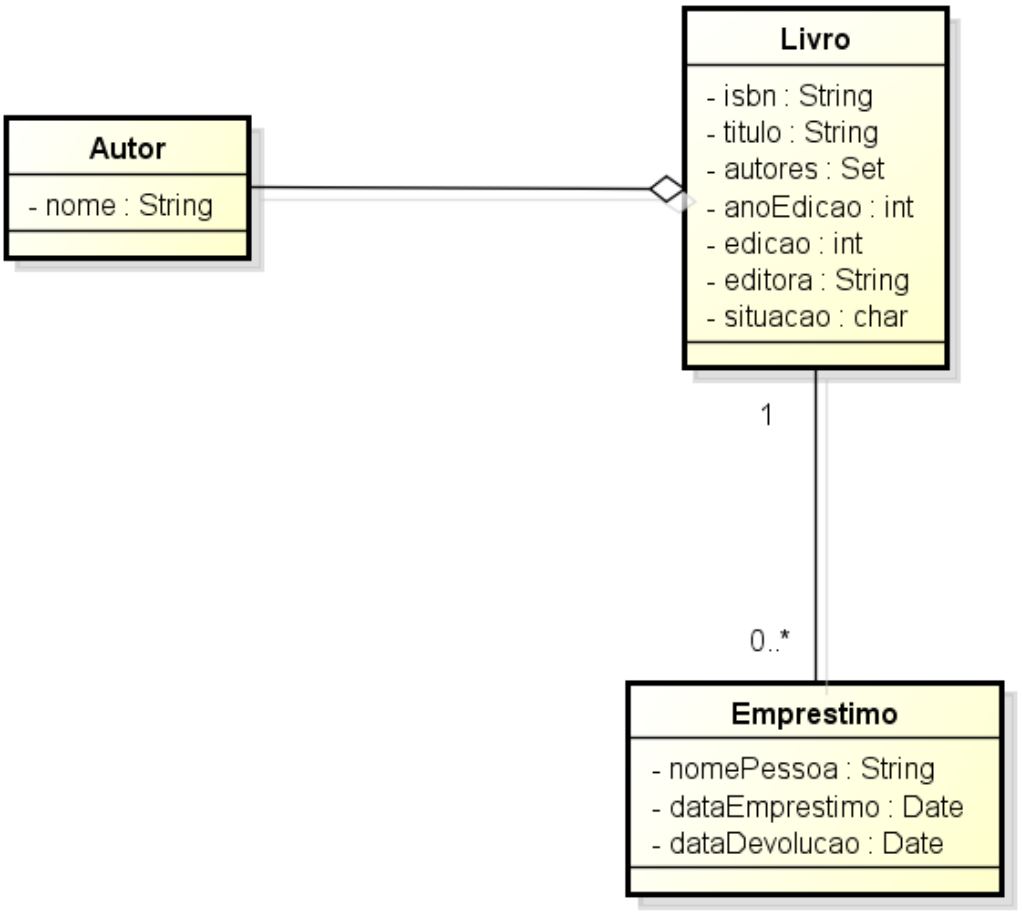


Figura 2. Diagrama de classes do domínio do problema.

Observe que, além dos atributos identificados anteriormente, foi definido na classe **Livro** o campo **situacao**, responsável para indicar se um livro está emprestado ou não. Será considerado o seguinte: se o livro estiver emprestado, **situacao** terá o valor *2*, caso contrário, assumirá o valor *1*.

Agora que já se tem um modelo de classes, podemos definir como será o modelo ER (Entidade Relacionamento) correspondente ao mapeamento objeto-relacional.

Para obter esse novo modelo, precisamos rever algumas estratégias do mapeamento objeto-relacional, de uma perspectiva bastante simples, voltada para a solução do nosso problema atual:

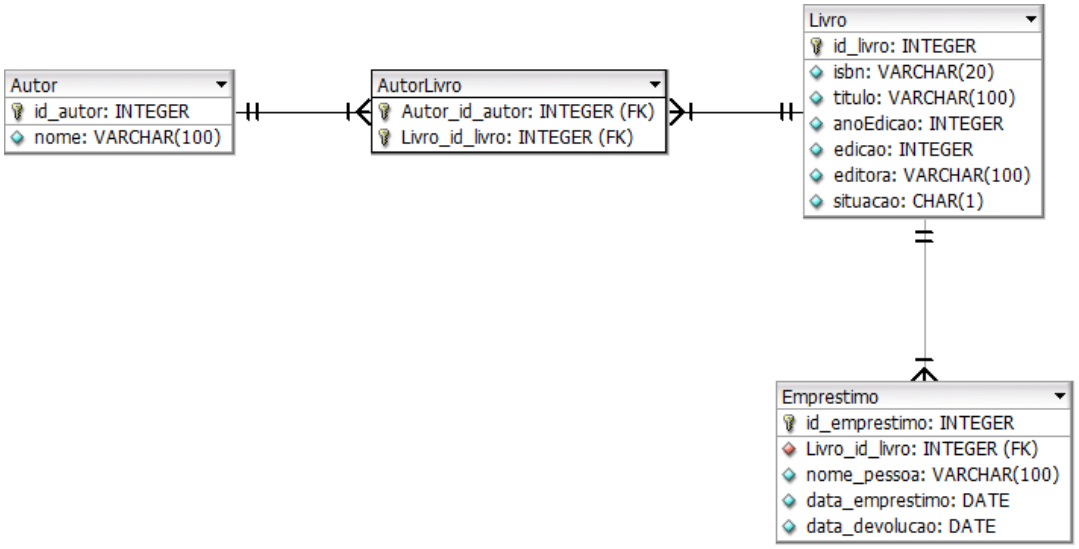
· Um atributo de uma classe pode ser mapeado para zero ou mais colunas em um banco de dados relacional. O mais comum é o mapeamento um-para-um, no entanto, podem-se citar dois extremos. Um caso é quando se tem um atributo de classe que não é persistente, tal como a média de compras da classe **Cliente**. Visto que esse campo pode ser calculado, então não precisa ser salvo no banco de dados. O outro extremo é quando um atributo é um objeto, tal como **Endereco** na classe **Cliente**. Isso levaria a uma associação entre duas classes e consequentemente seria necessário mapear os atributos de **Endereco**;

· Em geral cada classe é mapeada para uma tabela. No entanto existem casos em que isso não acontece, tal como nos relacionamentos de generalização/especialização, que não estudaremos neste artigo;

· Nos relacionamentos um-para-muitos, o mapeamento é feito definindo-se uma chave estrangeira na tabela muitos. Na situação do problema ora exposto, as classes **Livro** e **Emprestimo** são um exemplo disso. Dessa forma, no modelo relacional, a tabela **Emprestimo** terá uma chave estrangeira referenciando a tabela **Livro**;

· Nos relacionamentos muitos-para-muitos é necessário o conceito de uma tabela associativa, cuja finalidade é manter o relacionamento entre duas ou mais tabelas. Este é o caso do relacionamento **Livro-Autor**. Para mapear esse relacionamento será criada a tabela **AutorLivro**, que terá chaves estrangeiras referenciando **Autor** e **Livro**.

Após essas considerações pode-se então apresentar o modelo ER, mostrado na **Figura 3**, derivado do diagrama de classes da **Figura 2**.



abrir imagem em nova janela

Figura 3. Diagrama ER produzido a partir do mapeamento objeto-relacional

Pode-se observar na **Figura 3** que foram definidas chaves primárias para as entidades, pois essa também é uma tarefa do mapeamento objeto-relacional. Nesse exemplo, preferimos escolher números sequenciais – sem significado nas regras do negócio – para as chaves. A outra alternativa é escolher chaves naturais – atributos que fazem parte do objeto – tal como **isbn** da classe **Livro**. Um artigo, citado nos links, discute essa questão de escolher a chave primária das entidades.

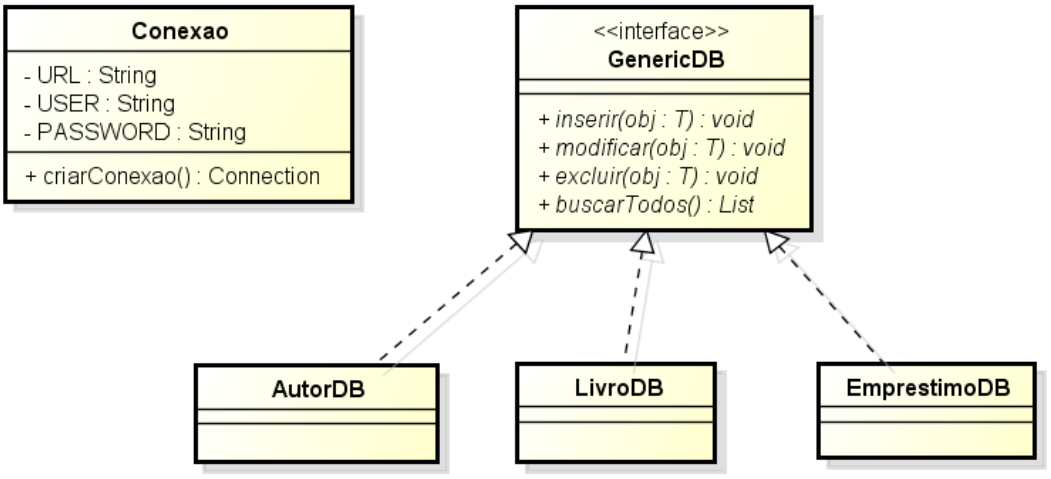
O passo seguinte no projeto corresponde a definir as classes necessárias para implementar a persistência na aplicação. Uma solução muito utilizada é o padrão DAO (*Data Access Object*). DAO implementa o encapsulamento e a centralização do acesso a fontes de dados – banco de dados, arquivos texto, XML, entre outros. Tal implementação permite que se separe a lógica de negócio da persistência. A camada de lógica de negócio consiste das classes do sistema, que no presente exemplo correspondem a **Autor**, **Livro** e **Emprestimo**. Já a camada de persistência se refere às classes responsáveis pelo acesso às fontes de dados.

No entanto, deixaremos essa solução para outra matéria em virtude da sua complexidade, posto que isso nos forçaria a expor toda a sua teoria antes de apresentar a solução. Por ora, então, será utilizada uma proposta apresentada no artigo "Solucionando problemas usando Java", publicado na Easy Java Magazine nº 4.

Na solução que foi apresentada nesse artigo foram criadas a classe **Conexao**, responsável por fazer a conexão com o banco de dados, e uma classe persistente para cada tabela no diagrama ER. Com isso chegou-se ao diagrama de classes persistentes mostrado na **Figura 4**.

Veja nesta figura que foi definida uma interface denominada **GenericDB**, a qual servirá para determinar os métodos que deverão ser obrigatoriamente implementados pelas classes **AutorDB**, **LivroDB** e **EmprestimoDB**. O leitor deve ter notado que não criamos uma classe correspondente à entidade **AutorLivro** – que foi utilizada para implementar o relacionamento muitos-para-muitos entre **Livro** e **Autor**. Esta decisão foi tomada porque a gravação/alteração dos dados na tabela **AutorLivro** será feita através dos métodos correspondentes da classe **LivroDB**.

Para concluir esta fase, definiremos que a interface com o usuário será construída usando Swing e o banco de dados será o Apache Derby. A opção pelo Derby se deve ao fato de que esta é uma aplicação simples e de uso pessoal, não exigindo maiores recursos, e que pode ser implementada utilizando um banco de dados embutido.



abrir imagem em nova janela

Figura 4. Diagrama de classes persistentes.

Codificação da Solução

Antes de começar a implementação é necessário preparar o ambiente, que consiste fundamentalmente em obter o *driver* JDBC para o Apache Derby e depois configurá-lo no IDE que será utilizado – que neste caso será o Eclipse. Se o leitor estiver acompanhando o desenvolvimento, é importante destacar que qualquer banco de dados pode ser usado em substituição ao Derby, desde que você obtenha o *driver* JDBC correspondente.

Acesse o endereço do Apache Derby que consta na relação de links, baixe o pacote do banco de dados e instale-o no diretório de sua preferência.

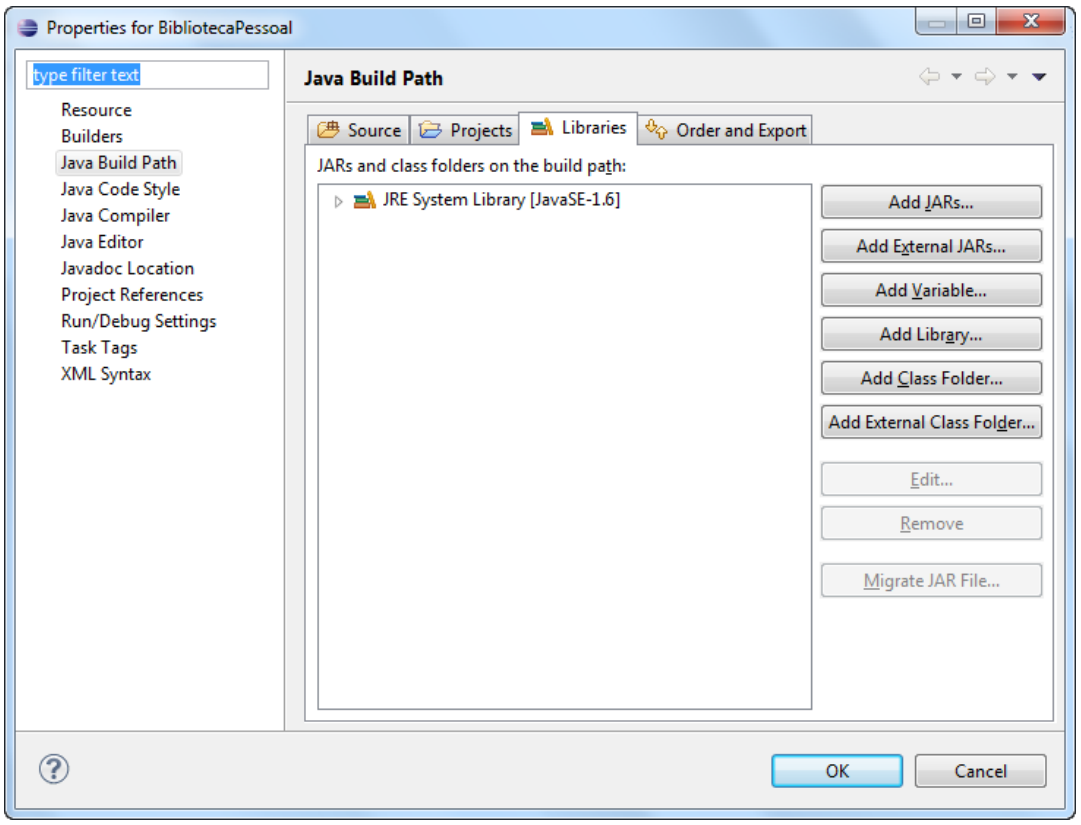
Em seguida crie um projeto Java no Eclipse, denominando-o *BibliotecaPessoal*, por exemplo. Após o projeto ter sido criado, precisamos adicionar a biblioteca do banco de dados. Para isso, com o projeto selecionado, pressione **ALT + ENTER**. Essa ação irá abrir a janela de *Propriedades do Projeto*, de acordo como mostra a **Figura 5**.

Nessa janela, selecione na caixa do lado esquerdo o item *Java Build Path* e, logo após, escolha a aba *Libraries*. Feito isso, clique no botão *Add External JARs*. Localize então o arquivo *Derby.jar* sob a pasta *lib* localizada no diretório onde a biblioteca do banco de dados foi instalada. Por fim, pressione o botão *Abrir* e depois o botão *OK* para confirmar a inclusão da biblioteca.

Concluída esta parte podemos partir para o desenvolvimento propriamente dito. Inicialmente vamos dividir nosso projeto em três pacotes: **br.com.biblioteca.dominio**, **br.com.biblioteca.persistencia** e **br.com.biblioteca.ui**. Portanto, crie esses três pacotes no menu de contexto do projeto. Para fazer isso, pressione o botão direito do mouse sobre o nome do projeto e selecione *New* | *Package*. Digite o nome do pacote e pressione *Finish*.

Criação das classes de domínio

As primeiras classes a serem criadas são as classes do domínio do problema: **Autor**, **Livro** e **Emprestimo**. Na **Listagem 1** mostramos o código da classe **Autor**. A criação desta classe deve ser feita no pacote **br.com.biblioteca.dominio**. Portanto, acesse o menu de contexto sobre esse pacote e escolha *New* | *Class*. Na janela que será aberta digite o nome da classe e pressione *Finish*. Em seguida digite o código da **Listagem 1** no editor.



abrir imagem em nova janela

Figura 5. Janela de propriedades do projeto.

Listagem 1. Classe de domínio Autor.

```
package br.com.bibliotecapessoal.dominio;
public class Autor implements Comparable<Autor> {
  private int id;
```

```
    private String nome;

    public Autor() {}

    public Autor(String nome) {
      this.nome = nome;
    }

    public Autor(int id, String nome) {
      this.id = id;
      this.nome = nome;
    }

    // métodos de acesso (getters e setters) vão aqui

    @Override
    public String toString() {
      return "Autor [id=" + id + ", nome=" + nome + "]";
    }

    @Override
    public int hashCode() {
      return this.nome.hashCode();
    }

    @Override
    public boolean equals(Object obj) {
      return true;
    }

    public int compareTo(Autor autor) {
      return this.nome.compareTo(autor.getNome());
    }

}
```

Note que foi definido o campo **id** de tipo **int**. Isso é importante para que o mapeamento funcione adequadamente, visto que esta tabela no banco de dados relacional possuirá uma chave primária que corresponderá a esse atributo no objeto. As demais classes do domínio do problema igualmente incluirão um atributo chamado **id** com o mesmo objetivo.

A classe **Autor** também implementa a interface **Comparable** – que define o método **compareTo()** – e sobrescreve os métodos **hashCode()** e **equals()** de **Object**. A justificativa para isso será dada mais adiante.

Após isso, repete-se o mesmo procedimento para criar as classes **Livro** e **Emprestimo**, cujas implementações são apresentadas nas **Listagens 2** e **3**, respectivamente.

Listagem 2. Classe de domínio Livro.

```
package br.com.bibliotecapessoal.dominio;
import java.util.HashSet;
import java.util.Set;

public class Livro {
  private int id;
  private String isbn;
  private String titulo;
  private int anoEdicao;
  private int edicao;
  private String editora;
  private Set<Autor> autores;
  private char situacao;

  public Livro() {
    this.autores = new HashSet<Autor>();
  }

  public Livro(String isbn, String titulo, int anoEdicao, int edicao,
      String editora, char situacao)
  {
    this.isbn = isbn;
    this.titulo = titulo;
    this.anoEdicao = anoEdicao;
    this.edicao = edicao;
    this.editora = editora;
    this.autores = new HashSet<Autor>();
    this.situacao = situacao;
  }

  public Livro(int id, String isbn, String titulo, int anoEdicao, int edicao,
      String editora, Set<Autor> autores, char situacao)
  {
    this.id = id;
    this.isbn = isbn;
    this.titulo = titulo;
    this.anoEdicao = anoEdicao;
    this.edicao = edicao;
    this.editora = editora;
    this.autores = new HashSet<Autor>();
    this.situacao = situacao;
  }

  // métodos de acesso (getters e setters) vão aqui

  @Override
  public String toString() {
    return "Livro [id=" + id + ", titulo=" + titulo + "]";
  }
```

```
}
```

Na classe **Livro** precisa ser destacado o campo **autores** do tipo **Set**. A explicação para isso iniciou com a apresentação da **Figura 2**, onde falamos como seria implementado o relacionamento muitos-para-muitos. A decisão de escolher a interface **Set** é porque este tipo de conjunto não permite objetos duplicados, e não queremos que um livro tenha autores repetidos.

Essa característica de **Set**, no entanto, requer que a classe dos objetos que são adicionados a essa *collection* implemente a interface **Comparable** – para definir como dois objetos serão comparados. Além disso, visto que se optou por usar a implementação **HashSet** da interface **Set**, deve-se definir os métodos **hashCode()** e **equals()**. Isso é fundamental porque **HashSet** precisa do código *hash* para inserir e depois localizar objetos na *collection*.

Observando a classe **Autor**, vemos que o código dos métodos citados é muito simples. No entanto, essa simplicidade se deve ao fato de que a classe tem apenas o atributo **nome** do tipo **String**. Por causa disso, apenas usamos os métodos correspondentes já disponíveis em **String**. Mas, deve-se destacar que o Eclipse oferece assistentes para implementar **equals()** e **hashCode()**, no menu *Source* | *Generate hashCode and equals*. Sugere-se usar esses assistentes, caso o leitor precise implementar tais métodos em classes que possuam mais atributos.

Consulte a Easy Java Magazine nº 1 para estudar um pouco mais sobre *collections* e obter mais detalhes sobre a interface **Set** e sua implementação **HashSet**.

Listagem 3. Classe de domínio Emprestimo.

```
package br.com.bibliotecapessoal.dominio;
  import java.util.Date;
  public class Emprestimo {

    private int id;
    private String nomePessoa;
    private Date dataEmprestimo;
    private Date dataDevolucao;
    private Livro livro;

    public Emprestimo() {}

    public Emprestimo(String nomePessoa, Date dataEmprestimo, Date dataDevolucao,
        Livro livro)
    {
      this.nomePessoa = nomePessoa;
      this.dataEmprestimo = dataEmprestimo;
      this.dataDevolucao = dataDevolucao;
```

```
    this.livro = livro;
  }

  public Emprestimo(int id, String nomePessoa, Date dataEmprestimo,
      Date dataDevolucao, Livro livro)
  {
    this.id = id;
    this.nomePessoa = nomePessoa;
    this.dataEmprestimo = dataEmprestimo;
    this.dataDevolucao = dataDevolucao;
    this.livro = livro;
  }

  // métodos de acesso (getters e setters) vão aqui

  @Override
  public String toString() {
    return "Emprestimo [nomePessoa=" + nomePessoa + ", dataEmprestimo="
        + dataEmprestimo + ", dataDevolucao=" + dataDevolucao + "]";
  }

}
```

A classe **Emprestimo** pode ser vista na **Listagem 3**, mas nada merece atenção especial nesse código.

Criação das classes persistentes

O passo seguinte no processo de desenvolvimento constitui-se da criação da interface e classes necessárias para a persistência: **Conexao**, **GenericDB**, **AutorDB**, **LivroDB** e **EmprestimoDB**. Essas classes devem ser criadas no pacote **br.com.bibliotecapessoal.persistencia**.

Na **Listagem 4** pode ser vista a classe **Conexao**, responsável por fazer a conexão com o banco de dados. Os membros desta classe são definidos como **static** de maneira que não seja necessário criar um objeto para que o método **criarConexao()** seja executado. O atributo **URL** define a *url* de conexão com o Derby, onde, além do *driver* JDBC, informamos o nome do banco de dados a ser criado – *biblioteca*. A propriedade *create=true* define que o banco será criado na primeira conexão, se ele não existir. Visto que o caminho do banco não foi informado na *url*, o banco de dados será criado no diretório corrente. Nesse caso, no próprio diretório do projeto Eclipse.

Listagem 4. Código da classe Conexao.

```
package br.com.bibliotecapessoal.persistencia;

import java.sql.Connection;
```

```
import java.sql.DriverManager;
import java.sql.SQLException;

public abstract class Conexao {
  public static final String URL = "jdbc:derby:biblioteca;create=true";

  public static Connection criarConexao() throws SQLException {
    Connection con = (Connection) DriverManager.getConnection(URL);
    return con;
  }

}
```

Com o objetivo de definir o contrato que as classes persistentes deverão cumprir, optou-se por criar uma interface, a qual denominamos **GenericDB**. Tal interface declara quais são os métodos que deverão ser implementados pelas classes **AutorDB**, **LivroDB** e **EmprestimoDB**. O código da interface é mostrado na **Listagem 5**.

Listagem 5. Código da interface GenericDB.

```
package br.com.bibliotecapessoal.persistencia;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.List;

public interface GenericDB<T, ID> {
  public void inserir(T obj);
  public void modificar(T obj);
  public void excluir(T obj);
  public List<T> buscarTodos();
  public T buscarPorID(ID id);

  public void fecha(ResultSet rs, Statement stm, Connection con);
}
```

É importante observar que **GenericDB** é uma interface genérica. Por isso ela foi definida com os parâmetros **T** e **ID**, onde **T** poderá assumir qualquer classe de domínio e **ID** será o tipo do **id** da classe persistente correspondente. Definida dessa maneira, esta interface poderá ser implementada por qualquer classe persistente da aplicação.

Listagem 6. Código da classe AutorDB.

```
package br.com.bibliotecapessoal.persistencia;

import br.com.bibliotecapessoal.persistencia.Conexao;
import br.com.bibliotecapessoal.dominio.Autor;
```

```
//imports omitidos...

public class AutorDB implements GenericDB<Autor, Integer> {

  private Connection con;
  private ResultSet rs;
  private Statement stm;

  public AutorDB() {
    try {
      con = Conexao.criarConexao();
      DatabaseMetaData dbmd = con.getMetaData();
      // verifica se a tabela AUTOR já existe no BD
      rs = dbmd.getTables(null, null, "AUTOR", new String[]{"TABLE"});
      if (!rs.next()) {
        stm = con.createStatement();
        // se não houver uma tabela, ela é criada
        stm.executeUpdate("CREATE TABLE autor
        (id int generated always as identity, nome VARCHAR(40))");
      }
    } catch (SQLException e) {
      System.out.println("Erro ao criar conexão/tabela");
    } finally {
      // fecha os recursos e a conexão com o BD
      this.fecha(rs, stm, con);
    }
  }

  public void inserir(Autor autor) {
    // cria um comando INSERT com os atributos de Autor
    String s = "insert into autor values
    (DEFAULT,'" + autor.getNome() + "')";
      try {
        con = Conexao.criarConexao();
        stm = con.createStatement();
        // executa o comando para inserir os dados na tabela
        stm.executeUpdate(s);
      } catch (SQLException e) {
        System.out.println("Erro ao inserir na tabela");
      } finally {
        this.fecha(rs, stm, con);
      }

  }

  public void excluir(Autor autor) {
    // cria um comando DELETE usando o id do Autor
    String s = "delete from autor where id = " + autor.getId();
    try {
      con = Conexao.criarConexao();
      stm = con.createStatement();
      // executa o comando para excluir o autor da tabela
```

```
      stm.executeUpdate(s);
    } catch (SQLException e) {
      System.out.println("Erro ao tentar excluir na tabela");
    } finally {
      this.fecha(rs, stm, con);
    }
  }

  public void modificar(Autor autor) {
    // cria um comando UPDATE usando os atributos de Autor
    String s = "update autor set nome = '" + autor.getNome() +
    "' where id = " + autor.getId();
    try {
      con = Conexao.criarConexao();
      stm = con.createStatement();
      // executa o comando para modificar os dados na tabela
      stm.executeUpdate(s);
    } catch (SQLException e) {
      System.out.println("Erro ao inserir na tabela");
    } finally {
      this.fecha(rs, stm, con);
    }

  }

  public List<Autor> buscarTodos() {
    // declara um ArrayList para receber os dados da tabela
    List<Autor> lista = new ArrayList<Autor>();
    String s = "select * from autor";
    try {
      con = Conexao.criarConexao();
      stm = con.createStatement();
      // executa a consulta de todos os registros
      rs = stm.executeQuery(s);
      // percorre o ResultSet lendo os dados de Autor
      while (rs.next()){
        int id = rs.getInt("id");
        String nome = rs.getString("nome");
        // cria um Autor com os dados de um registro
        Autor autor = new Autor(id, nome);
        // adiciona o Autor no ArrayList
        lista.add(autor);
      }
    } catch (SQLException e) {
      System.out.println("Erro ao consultar tabela");
    } finally {
      this.fecha(rs, stm, con);
    }
    return lista;
  }

  public void fecha(ResultSet rs, Statement stm, Connection con) {
```

```
      if (rs != null) {
        try {
          rs.close();
        } catch (SQLException e){

        }
      }
      if (stm != null) {
        try {
          stm.close();
        } catch (SQLException e){

        }
      }
      if (con != null) {
        try {
          con.close();
        } catch (SQLException e){

        }
      }
    }

    public Autor buscarPorID(Integer id) {
      // cria um SELECT para retornar um Autor pelo id
      String s = "select * from autor where id = " + id;
      String nome;
      Autor autor = null;
      try {
        con = Conexao.criarConexao();
        stm = con.createStatement();
        // executa a consulta
        rs = stm.executeQuery(s);
        // cria um objeto Autor com os dados retornados
        if (rs.next()) {
          id = rs.getInt("ID");
          nome = rs.getString("NOME");
          autor = new Autor(id, nome);
        }
      } catch (SQLException e) {
        System.out.println("Erro ao consultar na tabela");
      } finally {
        this.fecha(rs, stm, con);
      }
      return autor;
    }

  }
```

A classe persistente **AutorDB** é apresentada na **Listagem 6**. Veja que esta classe implementa a interface **GenericDB** e deve definir os tipos genéricos declarados, os quais são **Autor** e **Integer**. Os tipos declarados correspondem respectivamente à classe de domínio mapeada pela classe persistente e o tipo de dado da chave primária da tabela. Note também que a chave primária foi definida como **Integer**, porque em *generics* apenas são permitidas classes e não tipos primitivos. Portanto, mesmo que o método **buscarPorID()** receba um **Integer**, ele o trata como **int** devido à operação **unboxing**, descrita no artigo "Explorando as classes Wrapper" na Easy Java Magazine nº 17.

Nesta classe destaca-se que o construtor de **AutorDB** verifica se existe uma tabela com o nome **AUTOR** no banco de dados. Caso não exista, a tabela é criada. A verificação da existência da tabela é responsabilidade do método **getTables()** da interface **DatabaseMetaData**. Esta interface deve ser implementada pelo *driver* JDBC com o objetivo de obter informações sobre os metadados do sistema gerenciador de banco de dados ao qual a aplicação está conectada. Além da relação de tabelas definidas no banco, pode-se retornar a chave primária e as colunas criadas em cada tabela, o nome e a versão do banco do banco de dados, entre outras.

Outra observação importante é que o método **buscarTodos()** retorna um **List**, que é uma interface. Isto é considerada uma boa prática de programação, pois o usuário tem a liberdade de usar a implementação da interface que julgar adequada, quando chamar o método. Por exemplo, se quisermos usar um **ArrayList** como uma lista de autores na aplicação, pode-se escrever o seguinte código: **ArrayList<Autor> lista = (ArrayList<Autor>) autorDB.buscarTodos()**, onde **autorDB** é uma instância de **AutorDB**. Comentários mais detalhados sobre a implementação da classe podem ser encontrados na própria **Listagem 6**.

Listagem 7. Código da classe LivroDB.

```
package br.com.bibliotecapessoal.persistencia;

//imports omitidos...

import br.com.bibliotecapessoal.dominio.Autor;
import br.com.bibliotecapessoal.dominio.Livro;

public class LivroDB implements GenericDB<Livro, Integer> {
  private Connection con;
  private ResultSet rs;
  private Statement stm;

  public LivroDB() {
    try {
      con = Conexao.criarConexao();
      DatabaseMetaData dbmd = con.getMetaData();
      // verifica se a tabela LIVRO já existe no BD
```

```
    rs = dbmd.getTables(null, null, "LIVRO", new String[] { "TABLE" });
    if (!rs.next()) {
      stm = con.createStatement();
      // se não houver uma tabela, ela é criada
      String s = "CREATE TABLE livro (id int not null generated always
       as identity primary key, isbn varchar(20), " +
                 "titulo varchar(100), anoedicao int, edicao int,
                  editora varchar(100), situacao char(1))";
      stm.executeUpdate(s);
      // verifica se a tabela LIVROAUTOR já existe no BD
      rs = dbmd.getTables(null, null, "AUTORLIVRO", new String[] { "TABLE" });

      if (!rs.next()) {
        stm = con.createStatement();
        // se não houver uma tabela, ela é criada
        s = "CREATE TABLE AUTORLIVRO (ID_AUTOR INT REFERENCES AUTOR, "
                   + "ID_LIVRO INT REFERENCES LIVRO)";
        stm.executeUpdate(s);
      }
    }
  } catch (SQLException e) {
    System.out.println("Erro ao criar conexão/tabela LIVRO");
  }
  finally {
    // fecha os recursos e a conexão com o BD
    this.fecha(rs, stm, con);
  }
}

public void inserir(Livro livro) {
  // cria um comando INSERT com os atributos de Livro
  String s = "insert into livro values(DEFAULT,'" + livro.getIsbn()
      + "', '" + livro.getTitulo() + "'," + livro.getAnoEdicao() + ",
       " + livro.getEdicao()
      + ", '" + livro.getEditora() + "', '" + livro.getSituacao() + "')";
  try {
    con = Conexao.criarConexao();
    stm = con.createStatement();
    // executa o comando para inserir os dados na tabela LIVRO
    stm.executeUpdate(s);
    // grava autores do livro na tabela AUTORLIVRO
    Livro novoLivro = this.buscarPorISBN(livro.getIsbn());
    Set<Autor> autores = livro.getAutores();
    Iterator<Autor> it = autores.iterator();
    while (it.hasNext()) {
      Autor autor = it.next();
      s = "INSERT INTO AUTORLIVRO VALUES (" + autor.getId() + ",
      " + novoLivro.getId() + ")";
      stm.executeUpdate(s);
    }

  }
```

```
  catch (SQLException e) {
    System.out.println("Erro ao inserir na tabela LIVRO/AUTORLIVRO");
  }
  finally {
    this.fecha(rs, stm, con);
  }

}

public void excluir(Livro livro) {
  // cria um comando DELETE usando o id do Livro
  String s = "delete from livro where id_livro = " + livro.getId();
  try {
    con = Conexao.criarConexao();
    stm = con.createStatement();
    // executa o comando para excluir o livro da tabela
    stm.executeUpdate(s);
  }
  catch (SQLException e) {
    System.out.println("Erro ao tentar excluir na tabela");
  }
  finally {
    this.fecha(rs, stm, con);
  }
}

public void modificar(Livro livro) {
  // cria um comando para excluir todos os autores do livro
  String s1 = "DELETE FROM autorlivro where id_livro = " + livro.getId();
  // cria um comando UPDATE usando os atributos de livro
  String s2 = "UPDATE livro set isbn = '" + livro.getIsbn()
      + "', titulo = '" + livro.getTitulo()
      + "', anoedicao = " + livro.getAnoEdicao()
      + ", edicao = " + livro.getEdicao()
      + ", editora = '" + livro.getEditora()
      + "', situacao = '" + livro.getSituacao()
      + "' where id = "
      + livro.getId();
  try {
    con = Conexao.criarConexao();
    stm = con.createStatement();
    // exclui autores do livro
    stm.executeUpdate(s1);
    // executa o comando para modificar os dados na tabela
    stm.executeUpdate(s2);
    // grava autores do livro na tabela AUTORLIVRO
    Set<Autor> autores = livro.getAutores();
    Iterator<Autor> it = autores.iterator();
    while (it.hasNext()) {
      Autor autor = it.next();
      s1 = "INSERT INTO AUTORLIVRO VALUES (" + autor.getId() + ",
      " + livro.getId() + ")";
```

```
        stm.executeUpdate(s1);
      }
    }
    catch (SQLException e) {
      System.out.println("Erro ao modificar a tabela");
    }
    finally {
      this.fecha(rs, stm, con);
    }

  }

  public List<Livro> buscarTodos() {
    // declara um ArrayList para receber os dados da tabela
    List<Livro> lista = new ArrayList<Livro>();
    String s = "select * from livro";
    Livro livro = null;
    try {
      con = Conexao.criarConexao();
      stm = con.createStatement();
      // executa a consulta de todos os registros
      rs = stm.executeQuery(s);
      // percorre o ResultSet lendo os dados de Livro
      while (rs.next()) {
        // cria um Autor com os dados de um registro
        int id = rs.getInt("id");
        String isbn = rs.getString("isbn");
        String titulo = rs.getString("titulo");
        int anoEdicao = rs.getInt("anoedicao");
        int edicao = rs.getInt("edicao");
        String editora = rs.getString("editora");
        Set<Autor> autores = new HashSet<Autor>();
        String situacao = rs.getString("situacao");
        // busca os autores do livro
        s = "select b.id, b.nome from autorlivro a, autor b
        where a.id_autor = b.id and a.id_livro = " + id;
        rs = stm.executeQuery(s);
        while (rs.next()) {
          int idAutor = rs.getInt("id");
          String nomeAutor = rs.getString("nome");
          autores.add(new Autor(idAutor, nomeAutor));
        }
        livro = new Livro(id, isbn, titulo, anoEdicao, edicao, editora,
            autores, situacao);
        // adiciona o Livro no ArrayList
        lista.add(livro);
      }
    }
    catch (SQLException e) {
      System.out.println("Erro ao consultar tabela");
    }
    finally {
```

```
    this.fecha(rs, stm, con);
  }
  return lista;
}

// código do método fecha()

public Livro buscarPorID(Integer id) {
  // cria um SELECT para retornar um Livro pelo id
  String s = "select * from livro where id = " + id;
  Livro livro = null;
  try {
    con = Conexao.criarConexao();
    stm = con.createStatement();
    // executa a consulta
    rs = stm.executeQuery(s);
    // cria um objeto Autor com os dados retornados
    if (rs.next()) {
      // cria um Autor com os dados de um registro
      String isbn = rs.getString("isbn");
      String titulo = rs.getString("titulo");
      int anoEdicao = rs.getInt("anoedicao");
      int edicao = rs.getInt("edicao");
      String editora = rs.getString("editora");
      Set<Autor> autores = new HashSet<Autor>();
      String situacao = rs.getString("situacao");
      // busca os autores do livro
      s = "select b.id, b.nome from autorlivro a, autor b
      where a.id_autor = b.id and a.id_livro = " + id;
      rs = stm.executeQuery(s);
      while (rs.next()) {
        int idAutor = rs.getInt("id");
        String nomeAutor = rs.getString("nome");
        autores.add(new Autor(idAutor, nomeAutor));
      }
      livro = new Livro(id, isbn, titulo, anoEdicao, edicao, editora,
          autores, situacao);
    }
  }
  catch (SQLException e) {
    System.out.println("Erro ao consultar na tabela LIVRO");
  }
  finally {
    this.fecha(rs, stm, con);
  }
  return livro;
}

private Livro buscarPorISBN(String isbn) {
  // cria um SELECT para retornar um Livro pelo isbn
  String s = "select * from livro where isbn = '" + isbn + "'";
  Livro livro = null;
```

```
    try {
      // executa a consulta
      rs = stm.executeQuery(s);
      // cria um objeto Autor com os dados retornados
      if (rs.next()) {
        // cria um Autor com os dados de um registro
        int id = rs.getInt("id");
        String titulo = rs.getString("titulo");
        int anoEdicao = rs.getInt("anoedicao");
        int edicao = rs.getInt("edicao");
        String editora = rs.getString("editora");
        String situacao = rs.getString("situacao");
        livro = new Livro(id, isbn, titulo, anoEdicao, edicao, editora,
            null, situacao);
      }
    }
    catch (SQLException e) {
      System.out.println("Erro ao consultar na tabela LIVRO");
    }
    return livro;
  }

}
```

A seguir, falaremos sobre a classe **LivroDB**, a qual pode ser vista na **Listagem 7**. O que difere **LivroDB** de **AutorDB** é que, como será constatado, na **Listagem 7** existe código referente ao tratamento dos dados dos autores de um livro. No construtor, por exemplo, são criadas – se não existirem – tanto a tabela **Livro** quanto a tabela **LivroAutor**.

Visto que foi decidido que um livro é composto de autores, foi preciso implementar nos métodos **inserir()**, **modificar()** e **excluir()** uma parte do código responsável por gravar/excluir os autores. Dessa maneira garante-se que, ao excluir um livro – por exemplo – exclui-se também seus autores, para manter a integridade do banco de dados.

Nos métodos **buscarTodos()** e **buscarPorID()** – que fazem a pesquisa de livros no banco de dados – foi necessário implementar uma parte correspondente à busca dos autores de um livro. As buscas de autores sempre retornam um **Set**, de acordo com o atributo definido na classe **Livro**. Verifique mais detalhes da codificação nos comentários da **Listagem 7**.

A classe **EmprestimoDB** é a última classe persistente a ser analisada. Além do que já foi comentado para as classes anteriores, ressaltaremos apenas o tratamento dos campos do tipo **Date**. Isso é importante destacar porque o banco de dados Derby requer um campo data no formato "MM/dd/yyyy" informado como caractere. Por isso foi utilizada a classe **SimpleDateFormat**, cuja finalidade é formatar datas. Dessa maneira, foi criado um objeto com o formato "MM/dd/yyyy", denominado **dateFormat**, o qual chamará o método **format()** encarregado de efetivar a formatação da data. Veja a **Listagem 8**.

Listagem 8. Código da classe EmprestimoDB.

```
package br.com.bibliotecapessoal.persistencia;

//imports omitidos...

import br.com.bibliotecapessoal.dominio.Emprestimo;
import br.com.bibliotecapessoal.dominio.Livro;

public class EmprestimoDB implements GenericDB<Emprestimo, Integer> {
  private Connection con;
  private ResultSet rs;
  private Statement stm;
  // define o formato do campo Data
  SimpleDateFormat dateFormat = new SimpleDateFormat("MM/dd/yyyy");

  public EmprestimoDB() {
    try {
      con = Conexao.criarConexao();
      DatabaseMetaData dbmd = con.getMetaData();
      // verifica se a tabela EMPRESTIMO já existe no BD
      rs = dbmd.getTables(null, null, "EMPRESTIMO", new String[] { "TABLE" });
      if (!rs.next()) {
        stm = con.createStatement();
        // se não houver uma tabela, ela é criada
        String s = "CREATE TABLE EMPRESTIMO (ID INT NOT NULL GENERATED
         ALWAYS AS IDENTITY PRIMARY KEY," +
            " ID_LIVRO INT REFERENCES LIVRO, NOME_PESSOA VARCHAR(40),
            DATA_EMPRESTIMO DATE, DATA_DEVOLUCAO DATE)";
        stm.executeUpdate(s);
      }
    }
    catch (SQLException e) {
      System.out.println("Erro ao criar conexão/tabela EMPRESTIMO");
    }
    finally {
      // fecha os recursos e a conexão com o BD
      this.fecha(rs, stm, con);
    }
  }
```

```
public void inserir(Emprestimo emprestimo) {
  // cria um comando INSERT com os atributos de Emprestimo
  String s = "insert into emprestimo (id, id_livro, nome_pessoa,
  data_emprestimo) values(DEFAULT,"
      + emprestimo.getLivro().getId()
      + ", '"+  emprestimo.getNomePessoa()
      + "', '" + new String(dateFormat.format(emprestimo.getDataEmprestimo().getTime()))
      + "')";
  try {
    con = Conexao.criarConexao();
    stm = con.createStatement();
    // executa o comando para inserir os dados na tabela
    stm.executeUpdate(s);
  }
  catch (SQLException e) {
    System.out.println("Erro ao inserir na tabela Emprestimo");
  }
  finally {
    this.fecha(rs, stm, con);
  }

}

public void excluir(Emprestimo emprestimo) {
  // cria um comando DELETE usando o id de Emprestimo
  String s = "delete from emprestimo where id = " + emprestimo.getId();
  try {
    con = Conexao.criarConexao();
    stm = con.createStatement();
    // executa o comando para excluir o autor da tabela
    stm.executeUpdate(s);
  }
  catch (SQLException e) {
    System.out.println("Erro ao tentar excluir na tabela");
  }
  finally {
    this.fecha(rs, stm, con);
  }
}

public void modificar(Emprestimo emprestimo) {
  // cria um comando UPDATE usando os atributos de Emprestimo
  String s = "update emprestimo set nome_pessoa = '" + emprestimo.getNomePessoa()
      + "', data_emprestimo = '" + new String(dateFormat.format
       (emprestimo.getDataEmprestimo().getTime()));
  if (emprestimo.getDataDevolucao() == null) {
    s += "' where id = " + emprestimo.getId();
  } else {
    s += "', data_devolucao = '" + new String(dateFormat.format
   (emprestimo.getDataDevolucao().getTime()))
      + "' where id = " + emprestimo.getId();
  }
```

```
  System.out.println(s);
  try {
    con = Conexao.criarConexao();
    stm = con.createStatement();
    // executa o comando para modificar os dados na tabela
    stm.executeUpdate(s);
  }
  catch (SQLException e) {
    System.out.println("Erro ao modificar a tabela");
  }
  finally {
    this.fecha(rs, stm, con);
  }

}

public List<Emprestimo> buscarTodos() {
  // declara um ArrayList para receber os dados da tabela
  List<Emprestimo> lista = new ArrayList<Emprestimo>();
  String s = "select * from emprestimo";
  try {
    con = Conexao.criarConexao();
    stm = con.createStatement();
    // executa a consulta de todos os registros
    rs = stm.executeQuery(s);
    // percorre o ResultSet lendo os dados de Emprestimo
    while (rs.next()) {
      int id = rs.getInt("id");
      int idLivro = rs.getInt("id_livro");
      String nomePessoa = rs.getString("nome_pessoa");
      Date dataEmprestimo = rs.getDate("data_emprestimo");
      Date dataDevolucao = rs.getDate("data_devolucao");
      Livro livro = new LivroDB().buscarPorID(idLivro);
      // cria um Emprestimo com os dados de um registro
      Emprestimo emprestimo = new Emprestimo(id, nomePessoa,
      dataEmprestimo, dataDevolucao, livro);
      // adiciona o Emprestimo no ArrayList
      lista.add(emprestimo);
    }
  }
  catch (SQLException e) {
    System.out.println("Erro ao consultar tabela");
  }
  finally {
    this.fecha(rs, stm, con);
  }
  return lista;
}

// código do método fecha()

public Emprestimo buscarPorID(Integer id) {
```

```
    // cria um SELECT para retornar um Emprestimo pelo id
    String s = "select * from Emprestimo where id = " + id;
    Emprestimo emprestimo = null;
    try {
      con = Conexao.criarConexao();
      stm = con.createStatement();
      // executa a consulta
      rs = stm.executeQuery(s);
      // cria um objeto Emprestimo com os dados retornados
      if (rs.next()) {
        id = rs.getInt("ID");
        int idLivro = rs.getInt("id_livro");
        String nomePessoa = rs.getString("nome_pessoa");
        Date dataEmprestimo = rs.getDate("data_emprestimo");
        Date dataDevolucao = rs.getDate("data_devolucao");
        Livro livro = new LivroDB().buscarPorID(idLivro);
        // cria um Emprestimo com os dados de um registro
        emprestimo = new Emprestimo(id, nomePessoa, dataEmprestimo,
        dataDevolucao, livro);
      }
    }
    catch (SQLException e) {
      System.out.println("Erro ao consultar na tabela");
    }
    finally {
      this.fecha(rs, stm, con);
    }
    return emprestimo;
  }

}
```

Este procedimento foi implementado tanto no método **inserir()** quanto em **modificar()**. A classe inclui alguns comentários em seu corpo de maneira que o leitor pode obter mais informações sobre sua codificação.

Nas classes **LivroDB** e **EmprestimoDB** não foi apresentado o código do método **fecha()**, o qual foi mostrado em **AutorDB**.

Com isso encerramos esta parte da matéria, deixando para a próxima a criação da interface gráfica que fará uso das classes aqui criadas.

Conclusões

Costumamos insistir em nossos artigos que tratam da solução de problemas, sobre a importância de ter um entendimento da questão que está sendo apresentada e a necessidade de seguirmos um método que oriente o desenvolvimento. O método que empregamos neste e em outros artigos anteriores não pretende substituir os processos de Engenharia de Software, mas é um bom ponto de partida para o desenvolvedor adquirir a prática da análise de requisitos.

Para o problema ora exposto, buscamos apresentar uma solução que, longe de ser a melhor, funciona adequadamente para os objetivos da matéria, que é implementar uma aplicação usando interface gráfica do usuário e que seja simples.

Nesta primeira parte do artigo buscou-se implementar as classes de domínio e as persistentes, deixando para a próxima a criação da interface gráfica. Igualmente deixamos para a parte final da matéria a integração entre as camadas de domínio, de persistência, e de interface com o usuário.

Links

Estudo em detalhes do mapeamento objeto-relacional.
agiledata.org/essays/mappingObjects.html

Discussão sobre a escolha da chave primária para uma tabela em um banco de dados relacional.
agiledata.org/essays/keys.html

Definição de DAO.
oracle.com/technetwork/java/dataaccessobject-138824.html

Site oficial do Apache Derby.
db.apache.org/derby/index.html

Uso de *collections* para implementar relacionamentos muitos-para-muitos.
ibm.com/developerworks/webservices/library/ws-tip-objrel4/

Parte II

Veja abaixo a segunda parte do artigo - Agora as partes I e II foram compiladas em um único artigo. Bons estudos :)

Desenvolvendo uma aplicação passo a passo – Parte 2

Desenvolvendo uma aplicação passo a passo – Parte 2

O presente artigo dá continuidade a uma série que aborda a criação passo a passo de uma pequena aplicação GUI com acesso a banco de dados. Nesta etapa, introduzimos o artigo com a apresentação do mapa navegacional da solução e em seguida criamos as primeiras janelas responsáveis pelas operações de inserção e alteração de dados em uma tabela do banco de dados.

Em que situação o tema é útil

O tema abordado é importante para quem precisa criar aplicações com GUI utilizando Eclipse e WindowBuilder. Principalmente, trata da utilização do componente JTable, empregado neste caso para visualização dos registros de uma tabela de banco de dados.

Na primeira parte desta matéria foi iniciado o desenvolvimento de uma aplicação utilizando um processo simples, que foi proposto na Easy Java Magazine nº 4. Tal processo consiste de quatro etapas assim definidas: 1) entenda o problema; 2) resolva-o manualmente; 3) defina o algoritmo; e, 4) codifique a solução. Considera-se estas etapas como sendo uma metodologia mínima necessária para implementar qualquer solução.

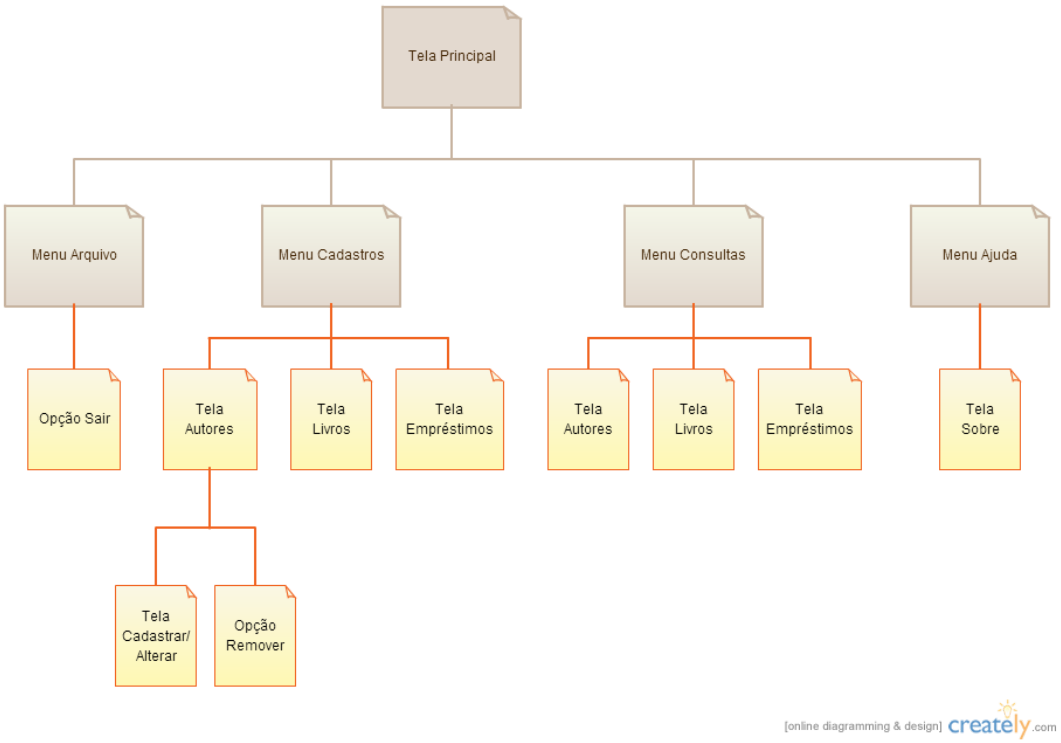
A aplicação em questão foi denominada Biblioteca Pessoal, a qual irá registrar os livros do usuário e os eventuais empréstimos efetuados. Da análise feita do problema, foram identificadas as classes de domínio **Autor**, **Livro** e **Emprestimo**, e as classes persistentes **AutorDB**, **LivroDB** e **EmprestimoDB**, além da classe **Conexao**, responsável pela conexão com o banco de dados.

Tais classes já foram implementadas no artigo publicado na Easy Java Magazine 27, o que deixou para esta parte a criação das interfaces do usuário, as quais serão desenvolvidas com o auxílio do *plug-in WindowBuilder*. Este *plug-in* oferece ao desenvolvedor as ferramentas necessárias para criar mais facilmente aplicações com GUI (Interface Gráfica do Usuário).

O *plug-in WindowBuilder* foi apresentado aos leitores na edição nº 27 da Easy Java Magazine, na qual foi descrito o procedimento para implementar um pequeno programa, usando a mesma ferramenta. O estudo daquela matéria seria importante, mas não imprescindível, para a leitura e entendimento do desenvolvimento que ora iniciaremos. De qualquer maneira, estamos considerando que o leitor já tem o *plug-in* devidamente instalado no IDE. Qualquer dúvida sobre instalação de *plug-ins* no Eclipse pode ser resolvida consultando a série de artigos sobre a ferramenta publicados nesta mesma revista.

Projeto de Interface Gráfica

Antes de iniciarmos a criação da interface, é importante definirmos que telas serão implementadas e como elas estarão interligadas, o que foi feito com o auxílio de um mapa navegacional, mostrado na **Figura 1**.



abrir imagem em nova janela

Figura 1. Mapa navegacional da aplicação.

Um mapa navegacional serve para representar a maneira como o usuário irá navegar entre as telas da aplicação. Assim, observando a **Figura 1**, pode-se notar que nossa aplicação terá uma tela principal com um menu de opções **Arquivo**, **Cadastros**, **Consultas** e **Ajuda**. No menu **Arquivo** haverá a opção **Sair** e em **Ajuda** teremos a tela **Sobre**. No menu **Consultas** haverá chamadas para três telas de consultas: **Autores**, **Livros** e **Empréstimos**. No menu **Cadastros**, onde serão chamadas as telas responsáveis por fazer a inserção e alteração dos registros nas tabelas do banco de dados, haverá três opções com denominações semelhantes ao que apresenta o menu **Consultas**. Entretanto, cada tela chamada no menu **Cadastros** mostrará uma lista de registros da tabela correspondente e terá uma chamada a uma tela que nomeamos como **Cadastrar/Alterar**, efetivamente responsável por fazer modificações nos registros, e uma opção denominada **Remover**, a qual fará a exclusão de um registro. Esta mesma tela **Cadastrar/Alterar** e a opção **Remover** se repetem nas demais telas do menu **Cadastros**, e por essa razão não foram mostradas no mapa.

Agora que se tem uma visão geral da interface do sistema, vamos iniciar o seu desenvolvimento, o qual está organizado nas seções seguintes.

Tela Principal

Para iniciar a criação da interface gráfica da aplicação, precisamos retomar o projeto criado na primeira parte desta matéria. Naquela oportunidade foram criados três pacotes de classes: **br.com.bibliotecapessoal.dominio**, **br.com.bibliotecapessoal.persistencia** e **br.com.bibliotecapessoal.ui**. É neste último o local onde ficarão as classes de interface do programa. Dessa forma, clique com o botão direito do mouse sobre o pacote **br.com.bibliotecapessoal.ui** para abrir o menu de contexto. Em seguida, selecione a opção *New* | *Other*, para selecionar o assistente desejado. Neste caso expanda o grupo *WindowBuilder* e, depois, *Swing Designer*. Escolha então a opção **JFrame**, conforme mostra a **Figura 2**. Com esse assistente nós criaremos um contêiner Swing de alto nível, que será a tela de entrada da aplicação, ou seja, será a primeira janela a ser mostrada quando o programa for executado.

São três os contêineres de alto nível no Swing: **JFrame**, **JDialog** e **JApplet**. **JFrame** possui uma barra de título e os botões maximizar, minimizar e fechar. Ele é frequentemente usado para criar as telas de uma aplicação, as quais irão conter outros objetos, tais como botões, caixas de textos, entre outros. **JDialog** é usado para construir caixas de diálogo para apresentar informações referentes a um **JFrame**. Por sua vez, **JApplet** é usado para criar *applets* – pequenos aplicativos que executam na web. Apesar de **JInternalFrame** ter comportamento e características semelhantes ao **JFrame**, ele é usado para criar aplicações MDI (*Multiple Document Interface*) e não é considerado um contêiner de alto nível.

Agora que sabemos por que foi escolhido **JFrame**, pressione o botão *Next*. Na próxima janela digite "BibliotecaPessoal" na caixa *Name*. Em seguida pressione *Finish* para criar a classe. Com isso teremos no editor do Eclipse o código da classe **BibliotecaPessoal** juntamente com o desenho do **JFrame** correspondente, acessíveis através das abas *Source* e *Design*, respectivamente, de acordo com o que pode ser visto na **Figura 3**. Antes de qualquer coisa, localize no Painel de Propriedades a propriedade **title** e digite nela "Biblioteca Pessoal", o que irá definir um título para a janela.

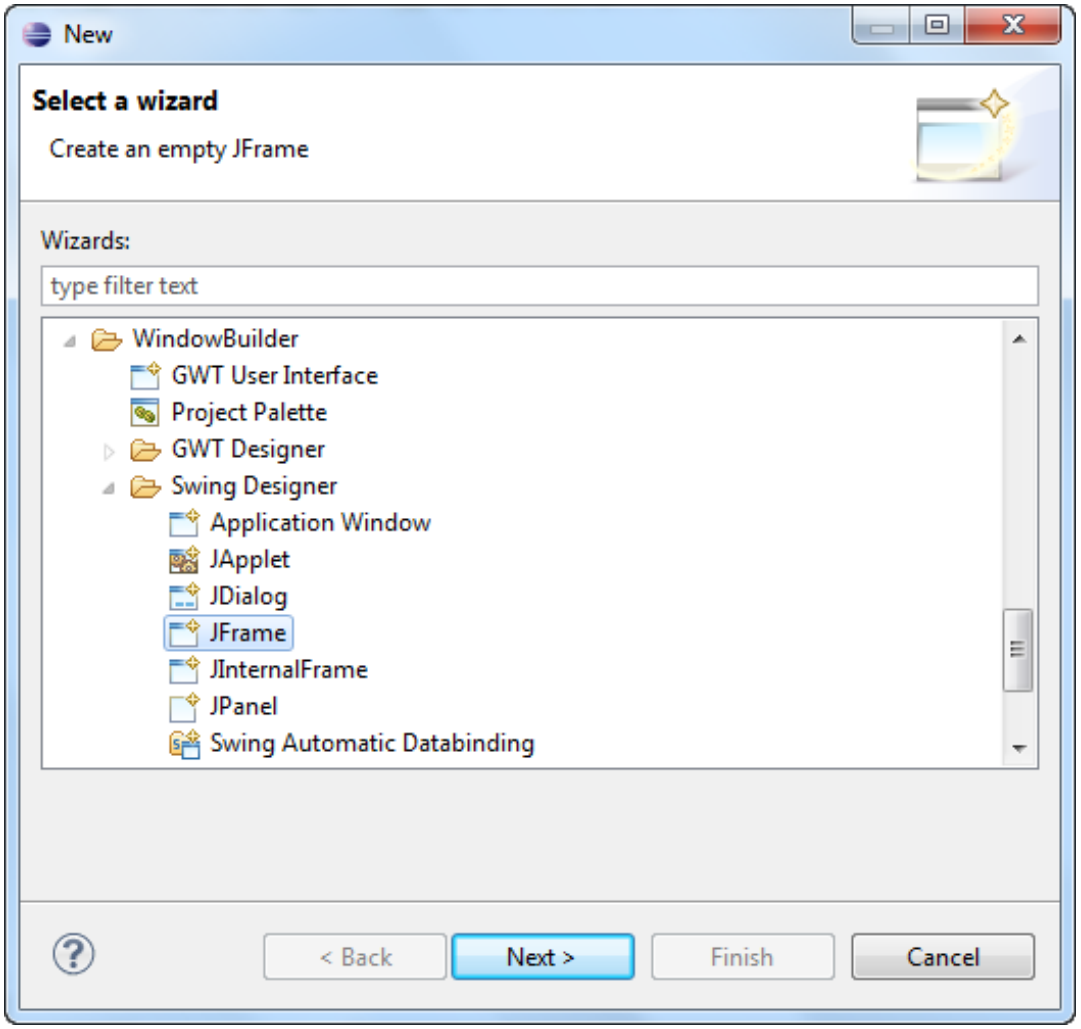
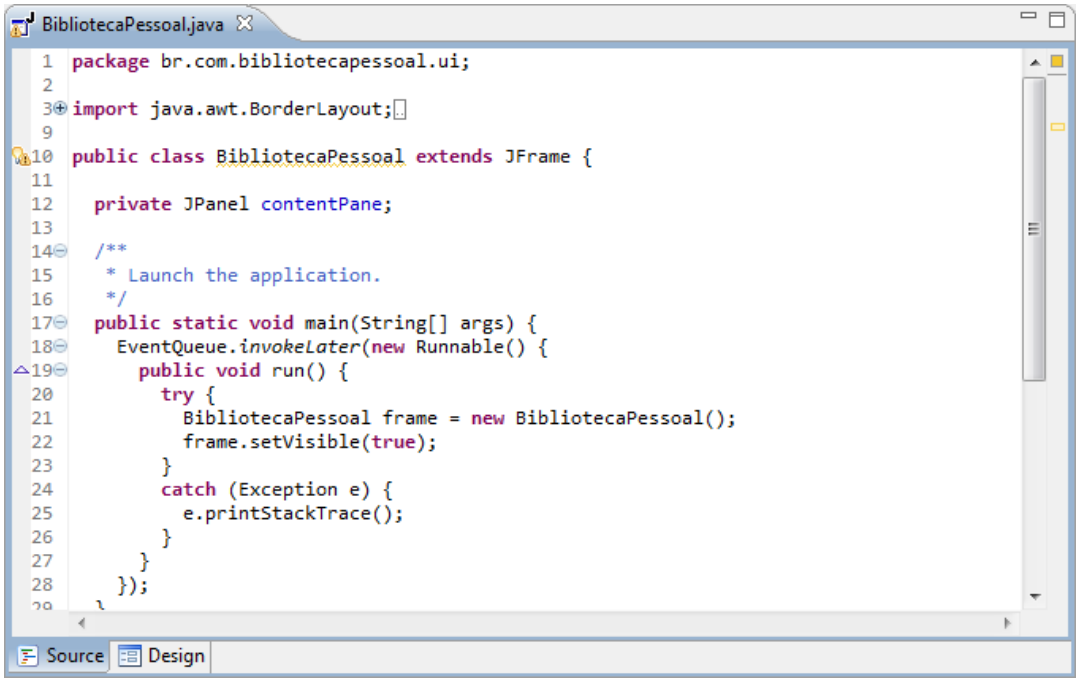


Figura 2. Selecionando um assistente para criação de janela.



abrir imagem em nova janela

Figura 3. Visão do código fonte da classe BibliotecaPessoal.

Nesta janela principal do programa haverá uma barra de menus, conforme mostramos no modelo navegacional. Uma barra de menus no Swing é criada com o componente **JMenuBar**. Em seguida este elemento é preenchido com objetos **JMenu**, um para cada uma das opções: **Arquivo**, **Cadastros**, **Consultas** e **Ajuda**. Dentro de cada um desses **JMenu**, as opções são criadas usando **JMenuItem**. Assim, por exemplo, a opção *Sair* do menu *Arquivo* é criada com um **JMenuItem**.

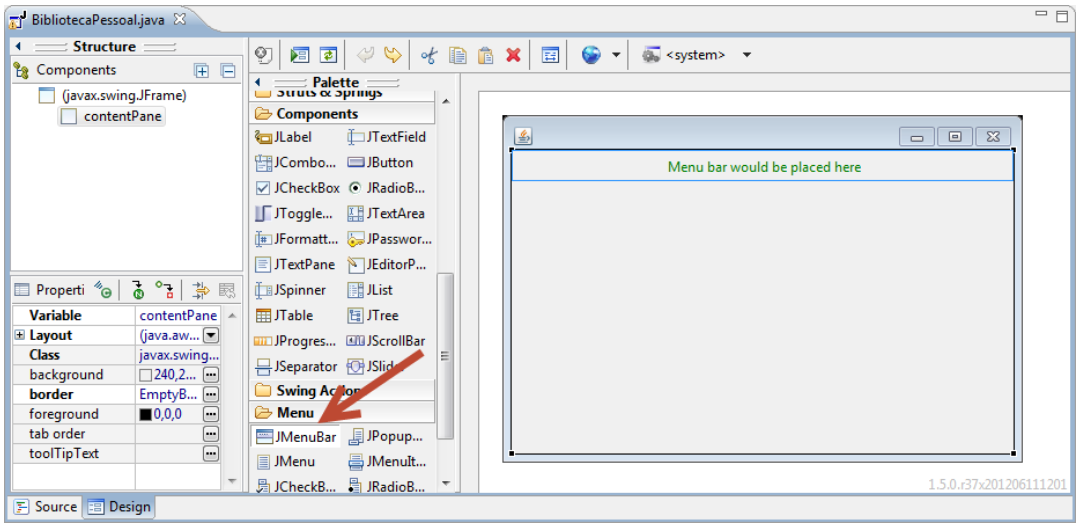
Depois desta breve introdução à criação de menus no Swing, podemos iniciar o desenho da tela propriamente dito – que consiste em adicionar componentes ao **JFrame** e modificar suas propriedades. Tais operações são feitas por meio do acesso à aba *Design* – a qual deve estar selecionada neste momento – localizada ao lado da aba *Source*. Para criar a barra de menus, será utilizado o componente **JMenuBar**, disponível no grupo *Menu* da Paleta. A localização deste componente pode ser vista na **Figura 4**.

Assim, clique no componente **JMenuBar** e posicione o ponteiro do mouse sobre o contêiner. Não importa a posição do **JFrame** onde você clica com o mouse, o menu será sempre colocado no topo da janela, onde aparece a mensagem "*Menu bar would be placed here*", também vista na **Figura 4**.

Depois de termos incluído a barra de menus, serão criados os **JMenu**. Para isto, selecione um **JMenu** na Paleta, localizado de acordo como mostra a **Figura 5**. Em seguida clique sobre o **JMenuBar**, na posição onde deseja inserir o menu. Ao fazer isso, a janela ficará com a aparência que pode ser vista na **Figura 5**, isto é, com o texto do **JMenu** selecionado. Portanto, digite a *string* correspondente ao menu desejado, que neste caso será "Arquivo".

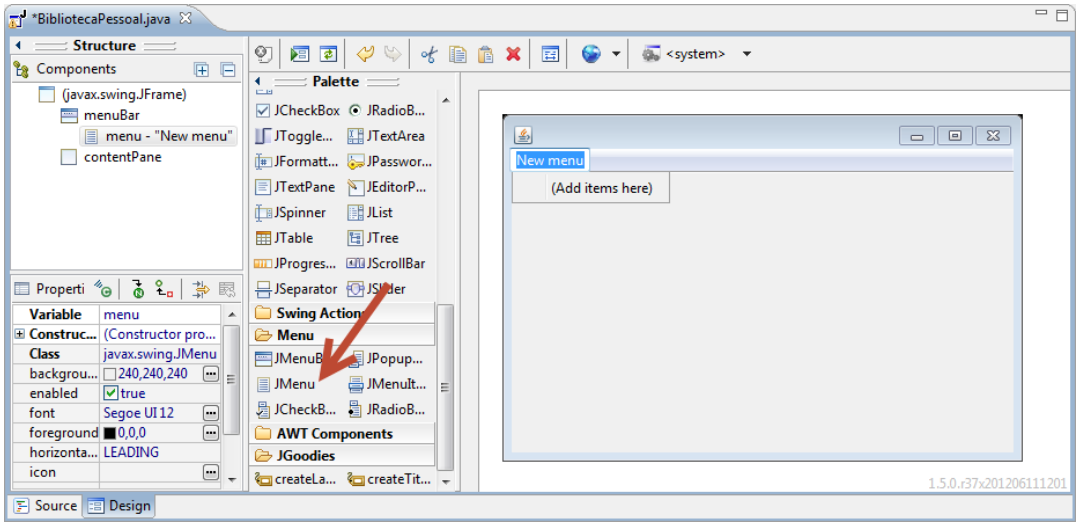
Após colocarmos um objeto na janela, é possível definir/modificar as propriedades desse objeto através do Painel de Propriedades. Uma das propriedades que podemos alterar para um menu é a **mnemonic**. Esta propriedade declara um caractere (**char**) que permite selecionar o referido menu apenas pressionando a tecla correspondente ao caractere escolhido – ou *ALT* + *mnemônico*. O mnemônico é indicado visualmente na barra de menus com uma letra sublinhada. Portanto, ainda com o menu **Arquivo** selecionado, clique na propriedade **mnemonic** do Painel de Propriedades e digite a letra "A". Com isso, ficará definido que, se pressionarmos a tecla correspondente à letra "A", o menu **Arquivo** será escolhido.

Na sequência será inserida a opção de menu **Sair** em **Arquivo**. Conforme pode ser observado na **Figura 5**, na posição onde aparece o texto "*(Add items here)*" é onde devem ser inseridas as opções **JMenuItem**. Deste modo, selecione um **JMenuItem** na Paleta e clique no local indicado.



abrir imagem em nova janela

Figura 4. Localização de JMenuBar na Paleta.

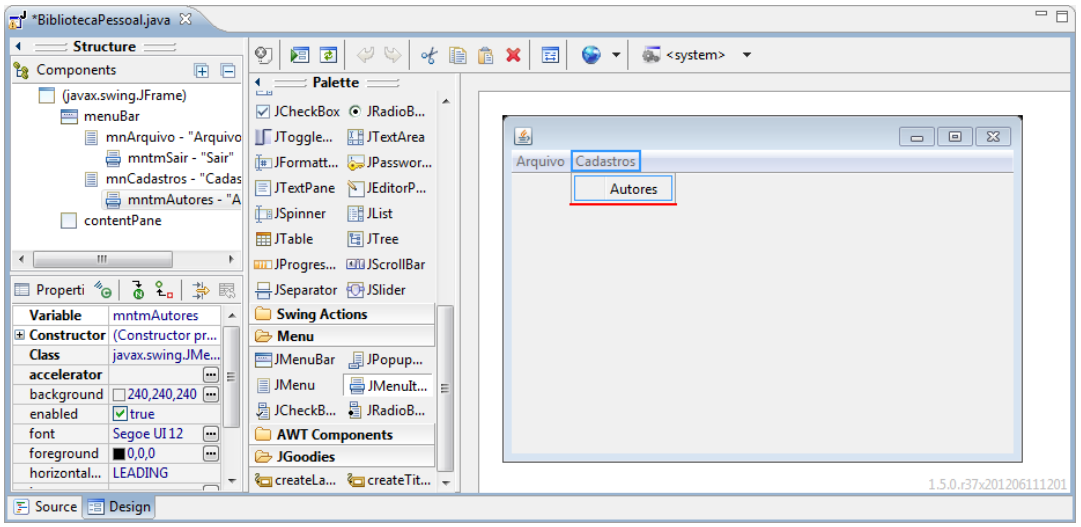


abrir imagem em nova janela

Figura 5. Inclusão de um JMenu na barra de menus.

Como próximo passo, adicione o menu **Cadastros**, colocando outro **JMenu** no lado direito de **Arquivo** e informe a letra "C" como **mnemonic**. Na etapa seguinte, adicione os **JMenuItem**: **Autores**, **Livros** e **Empréstimos**. Após incluir o **JMenuItem Autores**, os demais itens deverão ser adicionados na posição onde aparece uma linha vermelha, como podemos observar na **Figura 6**.

Em uma aplicação GUI, uma opção de menu pode ser acionada através de atalhos de teclado, denominados aceleradores de teclado. Num **JMenuItem** podemos definir aceleradores através da propriedade **acelerator**. Sendo assim, pressione o botão elipse junto a esta propriedade para abrir a caixa de diálogo onde será escolhido o conjunto de teclas aceleradoras.



abrir imagem em nova janela

Figura 6. Criação de um menu com mais de um JMenuItem.

Na janela da **Figura 7** é onde se escolhe as teclas aceleradoras para um item de menu. Suponha então que queremos definir *CTRL + A* como aceleradoras do menu **Autores**. Para isto, marque a caixa *Ctrl* em *Modifiers* e localize a letra *A* em *Key code*. Em seguida, irá aparecer *Ctrl-A* na caixa de texto *Press key stroke combination*. Pressione o botão *OK* para confirmar e a sequência *CTRL+A* será mostrada ao lado da opção **Autores**.

Repita os procedimentos anteriores para criar o menu **Consultas** e **Ajuda**, finalizando assim a criação da parte visual da janela principal. Mais adiante retornaremos a esta janela para escrever algum código referente às funcionalidades dos itens de menu.

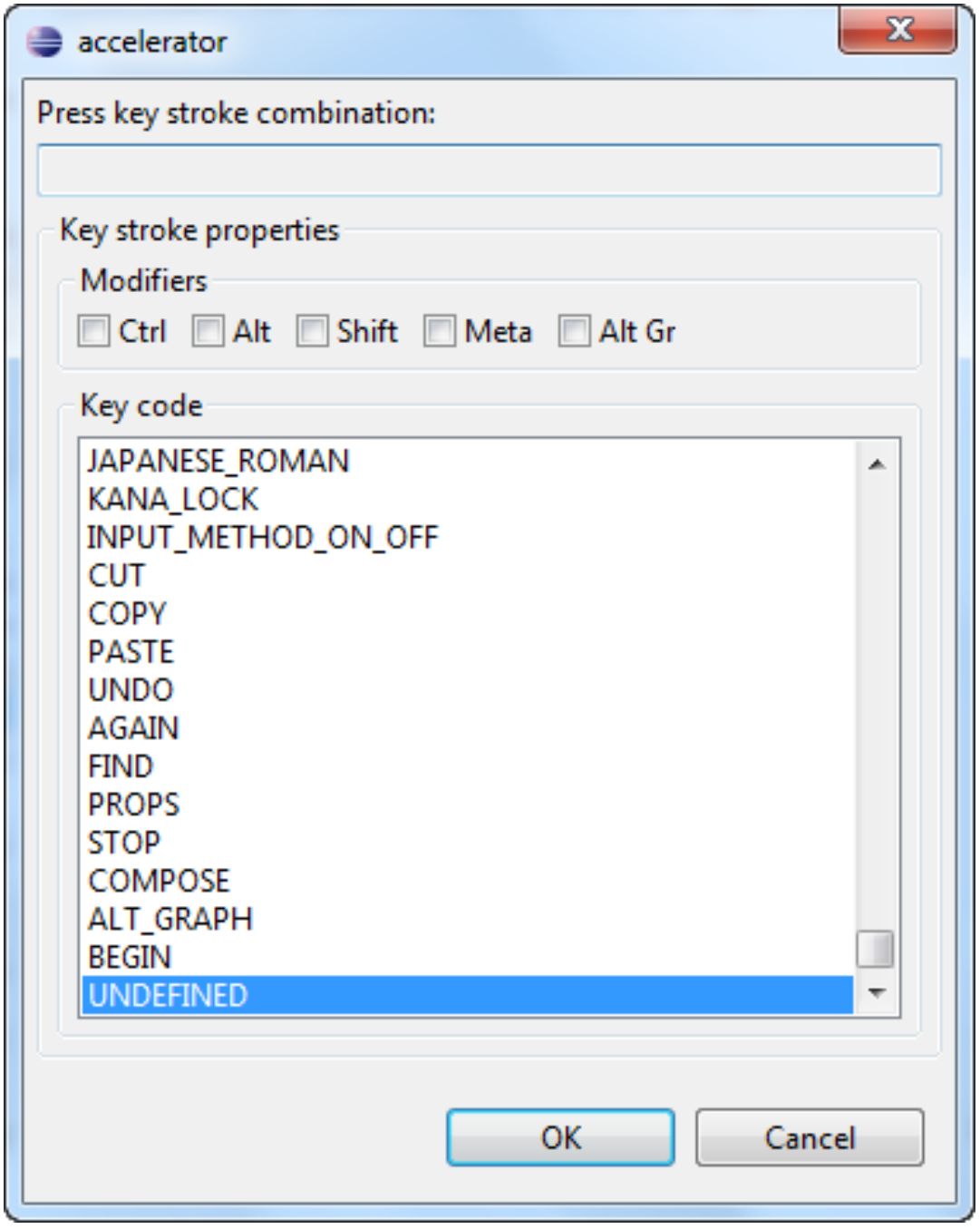


Figura 7. Definição de teclas aceleradoras para opções de menu.

Tela Autores

Na sequência do desenvolvimento será implementada a tela **Autores**, a qual deverá apresentar a aparência mostrada na **Figura 8**. Ou seja, esta interface terá um **JTable**, onde poderão ser visualizados os autores cadastrados, e quatro botões: **Atualizar**, **Novo**, **Alterar** e **Remover**.

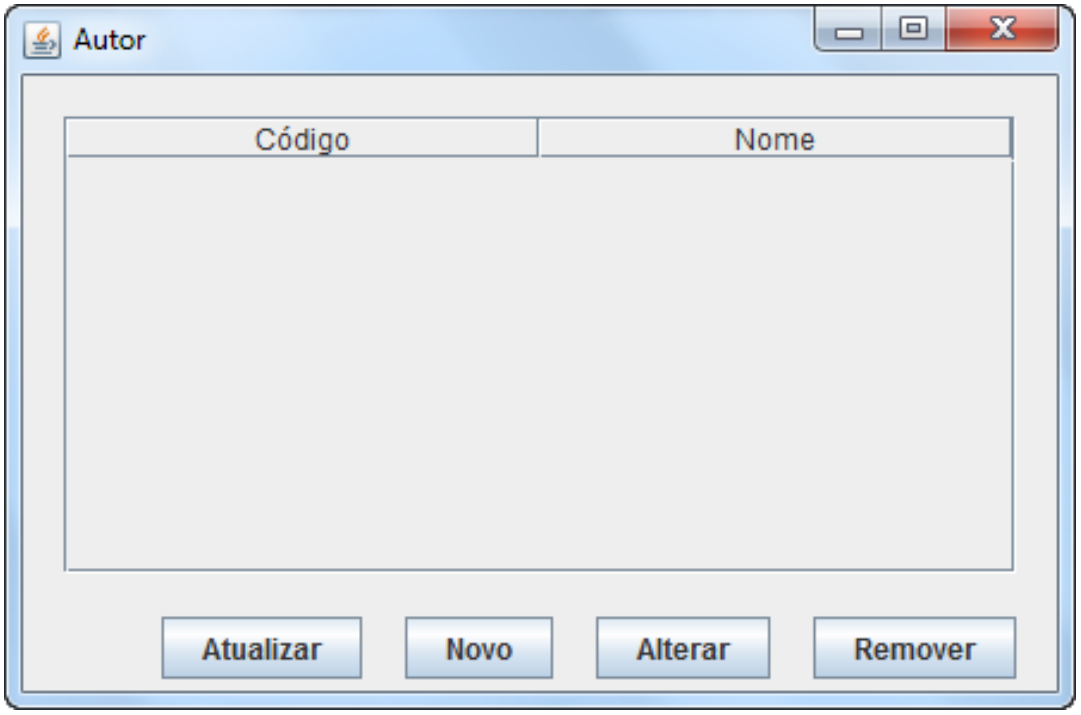


Figura 8. Tela de Autores do menu Cadastros.

O botão **Atualizar** faz a atualização dos dados visualizados no **JTable**, através de uma nova leitura na tabela do banco de dados. Com a finalidade de cadastrar um novo autor, utiliza-se o botão **Novo**. Após selecionar um autor no **JTable**, pode-se pressionar o botão **Alterar** para modificar os dados ou **Remover** para excluir o registro do banco de dados.

Após termos tido uma visão geral desta nova janela, podemos então iniciar a sua criação. A fim de criar esta e as demais telas da solução, sempre será utilizado o **JFrame**. Portanto, clique com o botão direito sobre o pacote **br.com.bibliotecapessoal.ui** e escolha *New* | *Other*. Na caixa de diálogo aberta, selecione o assistente *WindowBuilder* > *Swing Designer* > *JFrame* e pressione *Next*. Em seguida, digite "AutorUI" na caixa *Name* e pressione o botão *Finish*, e o código da classe **AutorUI**, recém-criada, será aberta na tela do editor Java.

Observe no código desta classe que existe um método **main()**. Isso acontece porque o *plug-in* sempre insere esse método quando cria uma classe que estende **JFrame**. Sabendo-se que só precisamos de um método **main()** na aplicação, e ele já existe na classe **BibliotecaPessoal**, o leitor deve excluir tal método.

Como próximo passo, vamos alterar algumas propriedades recorrendo ao Painel de Propriedades. Inicialmente, localize a propriedade **title** e digite "Autor". Depois, na propriedade **defaultCloseOperation**, selecione **HIDE_ON_CLOSE**. Esta última definição indica que a janela será ocultada ao ser fechada, visto que a opção **EXIT_ON_CLOSE** provoca o fechamento da aplicação e deve ser estabelecida apenas na janela principal.

A tarefa seguinte do nosso trabalho consiste em definir o *layout* do contêiner, que seja mais adequado para a solução do problema. Como já foi estudado em artigos anteriores, para a criação de interfaces gráficas, o Swing oferece os gerenciadores de *layout*, responsáveis pelo posicionamento dos componentes em um contêiner, independentemente da plataforma e tamanho de tela do computador.

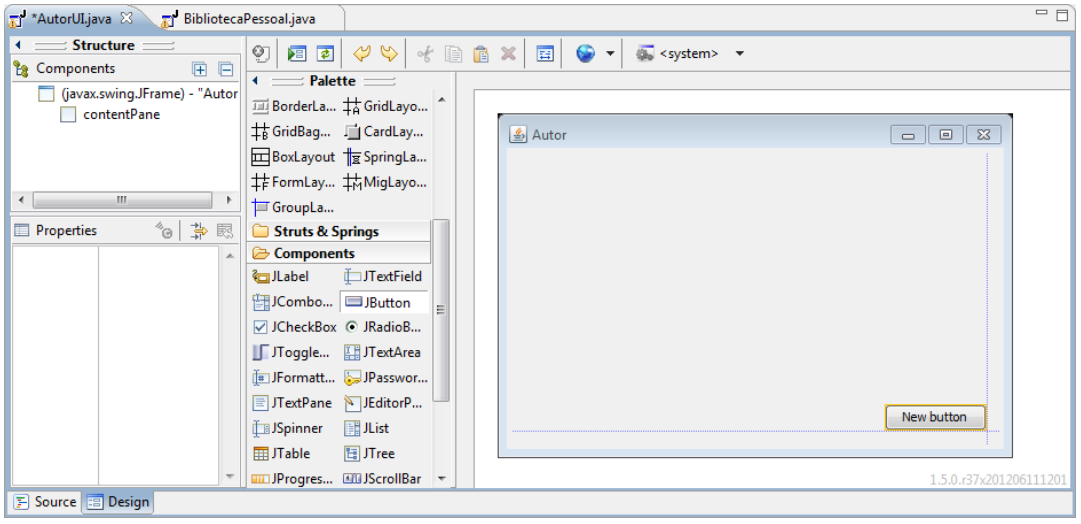
Por padrão, o *WindowBuilder* define que o **JFrame** usará o *layout BorderLayout*. No entanto, para facilitar o desenho da interface gráfica optamos por empregar *GroupLayout*, criado especialmente para ser utilizado em ferramentas de construção de GUIs (Interface Gráfica do Usuário), tais como a que existe no NetBeans e no *WindowBuilder*.

Entretanto, antes de fazer a alteração no *layout*, é preciso entender algo. Não se pode adicionar componentes diretamente no **JFrame**. Em vez disso, eles são adicionados a um **JRootPane**, contido no **JFrame**. **JRootPane** é uma classe contêiner usada nos bastidores, por exemplo pelas classes **JFrame**, **JApplet**, **JDialog**, entre outras, e para obter a instância dela devemos invocar o método **getContentPane()**. Visto que esse procedimento inviabiliza o desenho da tela de maneira visual, pois não seria possível adicionar um objeto no **JFrame** a partir do Painel de Propriedades, o WindowBuilder, ao criar a janela, adiciona a ela um **JPanel** denominado **contentPane**. E é sobre esse **JPanel** que os *widgets* são adicionados.

Por essa razão, com o objetivo de modificar o *layout* da janela, deve-se fazer a definição no **JPanel**. Portanto, clique na aba *Design* para que seja mostrada o **JFrame** e, a seguir, selecione o **JPanel**. No Painel de Propriedades localize a propriedade **Layout** e, após clicar no botão à direita, escolha *GroupLayout*.

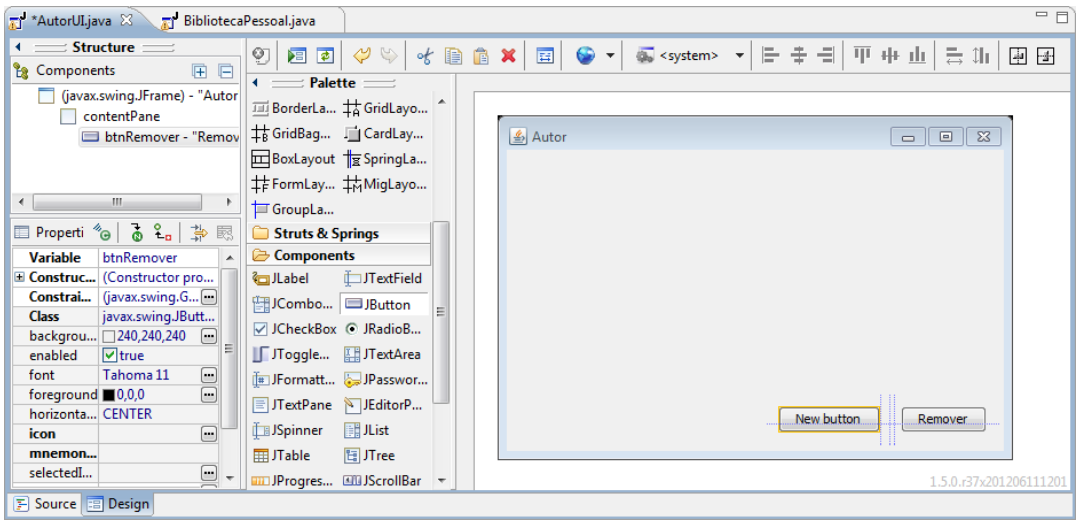
Após isso, podemos iniciar a adição dos *widgets* no contêiner, começando pelos botões na parte inferior da tela. Portanto, coloque um **JButton** no canto inferior direito, assegurando-se de que ele fique alinhado de acordo com as linhas guia, conforme a **Figura 9**. Quando confirmar a posição, você será solicitado a digitar o rótulo do botão. Digite "Remover".

Coloque agora o próximo botão, à esquerda do anterior, alinhando-o da maneira que mostra a **Figura 10**. Defina seu rótulo como "Alterar". De maneira similar, adicione mais dois botões, e atribua a eles os rótulos "Novo" e "Atualizar".



abrir imagem em nova janela

Figura 9. Posicionando um JButton na janela de acordo com as linhas guia.



abrir imagem em nova janela

Figura 10. Posicionando o segundo JButton na janela.

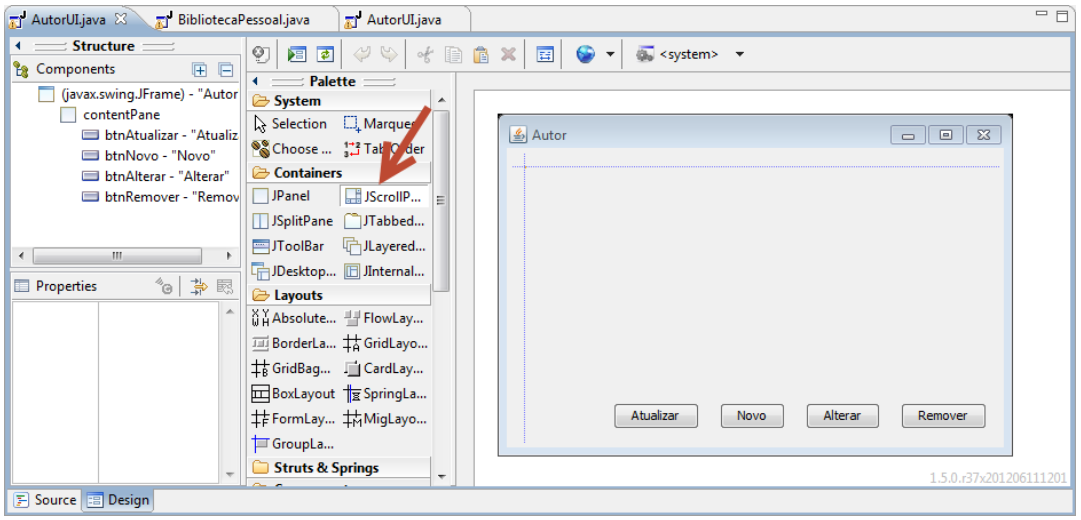
O próximo *widget* a ser adicionado à interface é um **JTable**. Este é o componente Swing para apresentar dados bidimensionais, em forma de grade. É ideal, portanto, para visualizar dados de tabelas de banco de dados. Entretanto, antes de iniciar a implementação, vejamos um pouco mais sobre este objeto.

JTable é um componente com arquitetura MVC (*Model View Controller*). Ou seja, para que ele cumpra sua função de apresentar dados em grade, são necessários três elementos. O **Model**, que é quem cuida da distribuição dos dados no **JTable**, e cuja tarefa é definida pela interface **TableModel**. As classes **AbstractTableModel** e **DefaultTableModel** são implementações desta interface. O **View**, que é a parte responsável pela apresentação, implementada pela interface **CellRenderer**. E, finalmente, o **Controller**, que irá controlar a apresentação dos dados na **View**. Esta última parte é o próprio **JTable**.

A classe **JTable** vem implementada para utilizar tipos padrão existentes em Java, tais como **String**, **Integer** e **Double**. Dessa forma é possível criar facilmente uma tabela muito simples, mas sem muita flexibilidade. Isto é, se desejarmos recursos mais elaborados, tais como exibir a tabela com linhas em cores alternadas, será necessário recorrer à criação de um *model* personalizado a partir da classe **AbstractTableModel**.

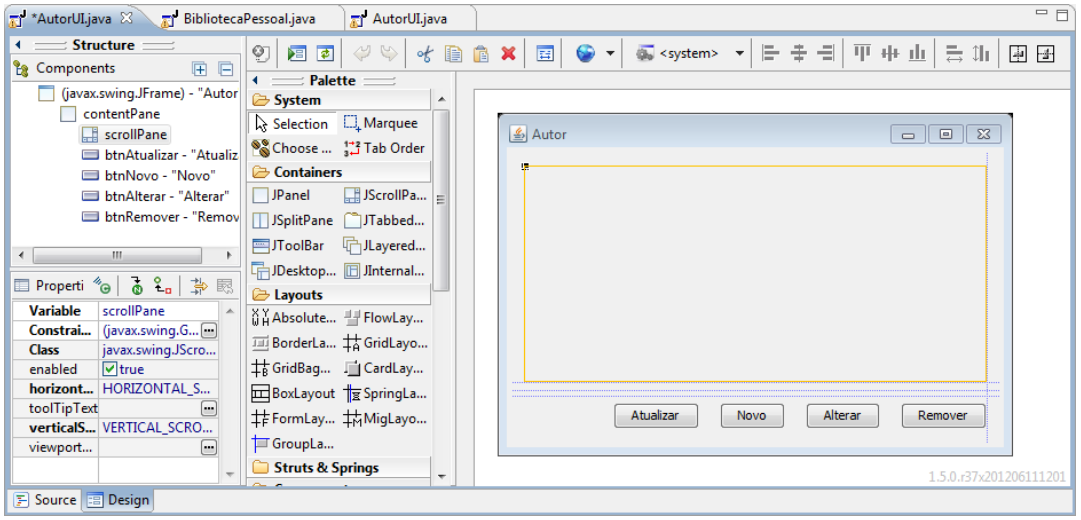
Outra questão a respeito de **JTable** é que a quantidade de informação a ser visualizada geralmente é maior do que cabe na tela. Caso isso aconteça será necessário implementar o recurso de rolagem, através do uso de *scrollbars*. A fim de obter essa funcionalidade, uma tabela deve ser colocada dentro de um **JScrollPane**. Dessa forma, quando a tabela for maior do que o espaço disponível, as barras de rolagem irão aparecer no **JScrollPane**. Além das barras de rolagem, este *widget* permite que sejam definidos nomes para as colunas, chamados de cabeçalho de coluna, e nomes para as linhas, chamados de cabeçalho de linha.

Visto que já temos uma visão geral do **JTable**, pode-se iniciar a sua criação na interface **Autores**. Assim, localize o **JScrollPane** no grupo *Containers* na Paleta, de acordo com a **Figura 11**, e posicione-o no canto superior esquerdo do **JFrame**, no ponto onde se cruzam as linhas guias. Em seguida arraste o mouse até que o painel ocupe o espaço conforme mostra a **Figura 12**.



abrir imagem em nova janela

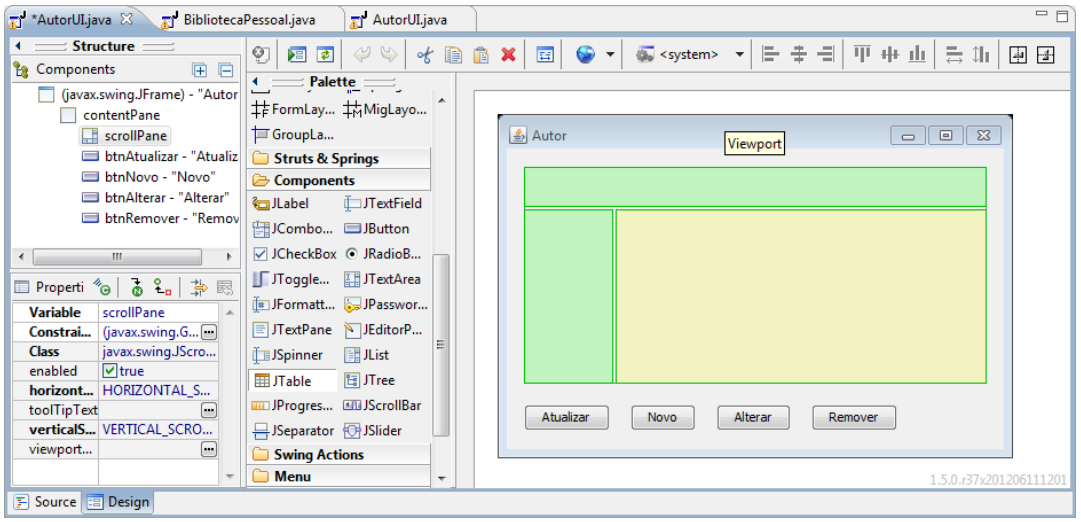
Figura 11. Posicionando o JScrollPane na janela.



abrir imagem em nova janela

Figura 12. Redimensionando o JScrollPane até a borda do JFrame.

Um **JScrollPane** é dividido em três partes: o cabeçalho de coluna, o cabeçalho de linha e o *Viewport*, como pode ser visto na **Figura 13**. O *Viewport* é uma visão retangular que se movimenta de acordo com as barras de rolagem, para mostrar os dados. Esta visão em formato retangular é mantida pela classe **JViewport**.



abrir imagem em nova janela

Figura 13. As três regiões de um JScrollPane.

Na **Figura 13**, a área horizontal em verde é o cabeçalho de coluna, a área vertical da mesma cor é o cabeçalho de linha, e a área restante corresponde à *Viewport*. É nesta região que será adicionado o **JTable**.

Selecione, portanto, um **JTable** no grupo *Components* da Paleta e posicione-o sobre a região **Viewport**. Agora, para atribuir um nome mais significativo ao componente, localize a propriedade **Variable** do *widget* **JTable** e digite "tbAutor".

Nesse ponto será necessário algum código para definir os rótulos no cabeçalho de coluna e como as células serão preenchidas com os dados da tabela **Autor**.

Inicialmente é necessário criar um **TableModel**, e aqui temos duas opções. Ou usamos **DefaultTableModel**, e ficamos limitados às possibilidades que esta classe oferece, ou criamos nosso próprio **TableModel**, estendendo a classe abstrata **AbstractTableModel**, o que nos oferece mais flexibilidade. Apesar desta flexibilidade, isso só seria obtido com a criação de uma classe e a implementação dos métodos definidos nessa classe abstrata. Como o foco da matéria é apresentar a solução de um problema passo a passo, preferiu-se utilizar **DefaultTableModel** e deixar para o leitor a tarefa de pesquisar a outra solução. Dessa maneira, a primeira coisa que precisamos fazer é criar uma instância desta classe, o que será realizado especificando-a como um atributo da classe **AutorUI**. A fim de acrescentar essa definição, digite, logo após os atributos de **AutorUI** declarados até então, o seguinte código: **private DefaultTableModel modelo = new DefaultTableModel();** e **private List<Autor> lista;**. Estamos aproveitando o momento para declarar mais um objeto na última linha, o qual será utilizado posteriormente. Na **Listagem 1** pode-se ver como ficará o início da declaração da classe.

Listagem 1. Classe AutorUI após declaração do DefaultTableModel.

```
public class AutorUI extends JFrame {

  private JPanel contentPane;
  private JTable tbAutor;
  private DefaultTableModel modelo = new DefaultTableModel();
  private List<Autor> lista;

  /**
   * Create the frame.
   */
  public AutorUI() {
    // Código do construtor
  }
}
```

Localize agora, no código da classe **AutorUI**, a linha onde é instanciado o **JTable**. Você deve ter encontrado o seguinte código: **tbAutor = new JTable()**. Modifique-o para que o construtor receba como parâmetro o objeto **modelo** definido anteriormente, e acrescente, antes da instanciação do **JTable**, as duas linhas mostradas no código da **Listagem 2**.

Listagem 2. Criação dos cabeçalhos de coluna no JTable.

```
  String[] nomesColuna = {"Código", "Nome"};
modelo.setColumnIdentifiers(nomesColuna);
tbAutor = new JTable(modelo);
```

As duas primeiras linhas na **Listagem 3** definem o cabeçalho de coluna do **JTabel**, de forma que a janela **Autores** deverá ter a aparência da **Figura 14**, quando o programa executar.

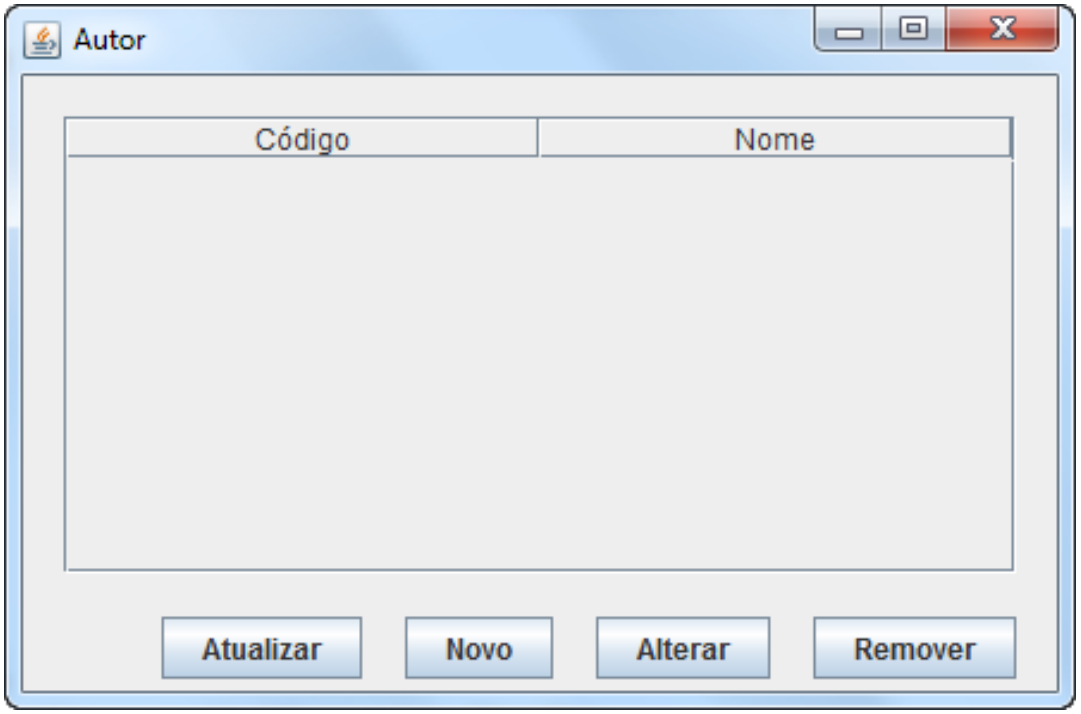


Figura 14. Janela Autores com os cabeçalhos de coluna.

O próximo passo é escrever o código necessário para povoar o **JTable** com os dados da tabela Autor. Tal código será implementado em um método membro da classe **AutorUI**, que foi denominado **atualizaTabela()**. Observe sua implementação na **Listagem 3**.

Listagem 3. Método cuja função é povoar o JTable.

```
private void atualizaTabela() {
```

```
    AutorDB autorDB = new AutorDB();
    lista = autorDB.buscarTodos();
    Iterator<Autor> it = lista.iterator();
    Autor a;
    while (modelo.getRowCount() > 0) {
        modelo.removeRow(0);
    }
    while (it.hasNext()) {
      a = it.next();
      modelo.addRow(new Object[] {a.getId(), a.getNome()});
    }
  }
```

No código do método **atualizaTabela()**, primeiro é criada uma instancia da classe persistente **AutorDB**, e na sequencia o método **buscarTodos()** retorna os registros da tabela Autor, guardando-os em **lista**. Em seguida, se existirem linhas de dados no **JTable**, estas são removidas. Isso é importante para que não ocorra duplicidade de dados na visualização. Somente após isso é que **lista** é percorrida com o uso de um **Iterator**, e o método **addRow()** adiciona um vetor de **Object** ao modelo. Cada elemento do vetor é formado por um campo declarado na classe **Autor**.

Concluída a implementação do método, precisa-se definir onde ele será chamado. Um desses locais é no botão **Atualizar**, através do evento **ActionEvent**, pois assim, sempre que pressionarmos este botão os dados atualizados da tabela Autor serão visualizados na tabela. O outro caso que nos interessa é o **WindowEvent**, que define os eventos de janela, tais como abrir, fechar, ativar, entre outros. Mais precisamente, temos interesse na abertura da janela; ou seja, sempre que a janela **Autores** abrir, os dados do **JTable** serão atualizados.

Para definir o tratamento do evento de **JButton**, clique com o botão direito sobre o botão **Atualizar** e escolha a opção *Action event handler > Action > ActionPerformed*. Feito isso, um tratador de eventos será criado e o editor de código será aberto no ponto onde será inserido o código a ser executado. Veja a **Listagem 4**.

Listagem 4. Adicionando um tratador de eventos para o botão Atualizar.

```
    JButton btnListar = new JButton("Atualizar");
  btnListar.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      // linha a ser inserida
      atualizaTabela();
    }
  });
```

O **ActionListener** criado define o método **actionPerformed()**, o qual é chamado quando se pressionar o botão. É o corpo desse método que precisa ser implementado. Neste caso, simplesmente inclua a chamada ao método **atualizaTabela()**.

Selecione agora o **JFrame** para adicionar um **WindowListener**, o tratador de eventos da janela. Aqui é necessária certa cautela, pois só podemos selecionar a janela clicando na sua barra de títulos, visto que a área restante está ocupada por um **JPanel**. Outra maneira de selecionar a janela é na Árvore de Componentes. Clique então com o botão direito sobre o **JFrame** e escolha a opção *Add event handler* > *Window* > *WindowOpened*. Observe na **Listagem 5** a parte da classe onde o editor irá abrir.

Listagem 5. Adicionando um tratador de eventos para o JFrame.

```
addWindowListener(new WindowAdapter() {

    @Override
    public void windowOpened(WindowEvent arg0) {
      // linha a ser inserida
      atualizaTabela();
    }
  });
```

Insira a chamada ao método **atualizaTabela()** no método **windowOpened()**. Apenas para conhecimento do leitor, os demais métodos que irão responder a cada um dos eventos de janela serão todos implementados nesta mesma classe anônima **WindowAdapter**.

Nesse momento precisa-se definir como a opção de menu na janela principal irá chamar a tela **AutorUI**. Com o objetivo de fazer isso, selecione a tela **BibliotecaPessoal**, abra o menu **Cadastros** e clique com o botão direito sobre a opção **Autores**. Agora, selecione a opção *Add event handler* > *Action* > *ActionPerformed*. Como se pode notar, o evento de **JMenuItem** é tratado da mesma maneira que o evento de **JButton**. Na verdade, vários outros componentes também disparam um evento **ActionEvent**, os quais podem ser a alteração de estado de um **JCheckBox**, teclar **ENTER** em um **JTextField**, além de cliques em **JButton** e **JMenuItem**, por exemplo.

Após selecionar a opção acima, o editor de código será aberto para digitarmos a resposta ao ato de clicarmos na opção **Atores** do menu **Cadastro**. Veja o código do *listener* na **Listagem 6**.

Listagem 6. Definindo um tratador de eventos para o item de menu Atores.

```
    JMenuItem mntmAutores = new JMenuItem("Autores");
  mntmAutores.addActionListener(new ActionListener() {
```

```
    public void actionPerformed(ActionEvent arg0) {
      // linhas a serem inseridas
      AutorUI autorUI = new AutorUI();
      autorUI.setVisible(true);
    }
  });
```

No método **actionPerformed()**, nesta situação, é necessário instanciar uma janela **AutorUI** e em seguida torná-la visível, através de uma chamada ao método **setVisible()**.

Finalmente, para que seja concluída a implementação da janela **AutorUI**, é necessário criar os *listeners* dos botões **Novo**, **Alterar** e **Remover**, mas para isso precisamos da tela que fará a inserção e modificação de um registro. Por isso, vamos fazer uma pausa na implementação de **AutorUI** para trabalharmos na criação da próxima janela.

Tela Cadastrar/Alterar Autores

A próxima tela a ser criada é a responsável por fazer as modificações na tabela Autor, sendo que tais modificações podem ser inserções de novos autores ou alterações de dados daqueles já existentes. Esta tela terá a aparência que vemos na **Figura 15**.

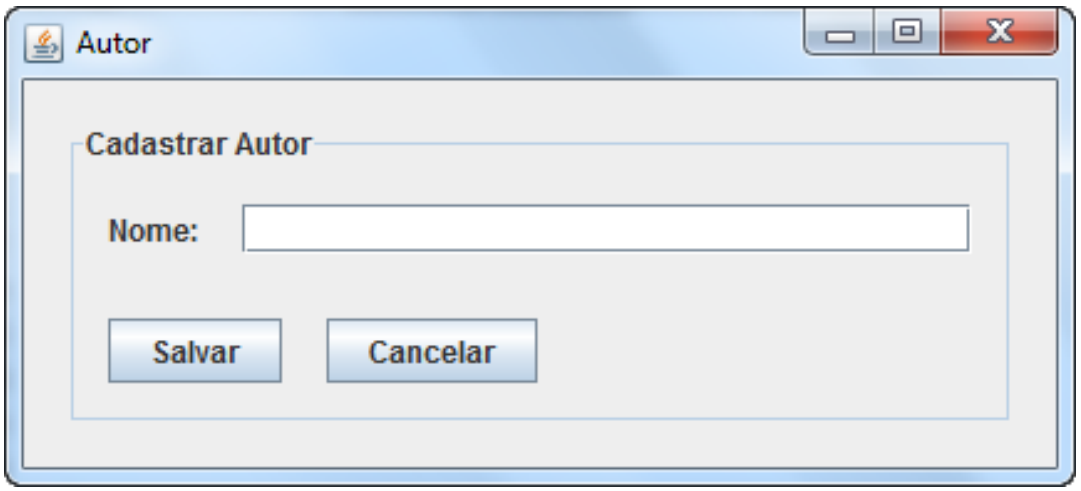


Figura 15. Tela para Cadastrar/Alterar autores.

A tela terá um **JPanel** onde haverá um **JLabel**, um **JTextField** e dois **JButton** denominados **Salvar** e **Cancelar**. O botão **Salvar** executará a função de gravar os dados do registro no banco de dados. Por sua vez, o botão **Cancelar** irá fechar a janela sem confirmar as modificações solicitadas.

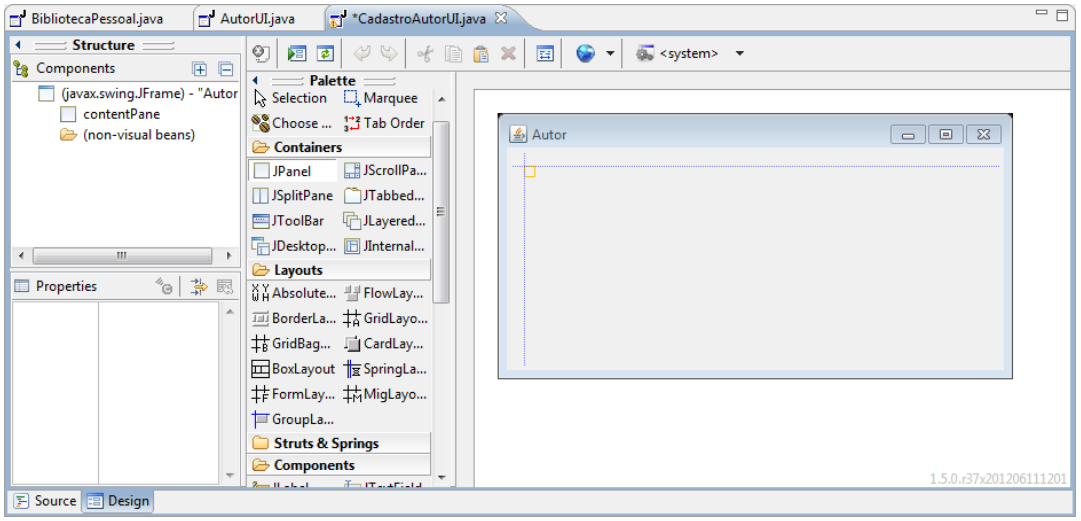
Um detalhe importante é que a implementação desta classe será feita de maneira a atender às duas funcionalidades: cadastrar um novo autor e alterar um autor existente. Mais adiante será mostrado como isso será obtido.

Vamos então iniciar a criação da tela, que será feita da mesma maneira que a tela **Autores** implementada anteriormente.

Para isso, clique com o botão direito sobre o pacote **br.com.bibliotecapessoal.ui** e escolha a opção *New* > *Other*. Na caixa de diálogo aberta, selecione o assistente *WindowBuilder* > *Swing Designer* > *JFrame* e pressione *Next*. Em seguida, digite "CadastroAutorUI" na caixa *Name* e pressione o botão *Finish*. Com isso, o código da classe **CadastroAutorUI** será aberta na tela do editor Java.

Semelhante ao que fizemos na tela anterior, exclua o código correspondente ao método **main()** que foi criado. Depois, com a aba **Design** e o **JFrame** selecionados, localize a propriedade **defaultCloseOperation** e altere-a para **HIDE_ON_CLOSE**. Digite "Autor" na propriedade **title** e defina que o gerenciador de *layout* será o **GroupLayout**.

Agora vamos adicionar o **JPanel**, conforme se vê na **Figura 16**. Assim, posicione o **JPanel** no canto superior esquerdo, no ponto onde cruzam as linhas guias.

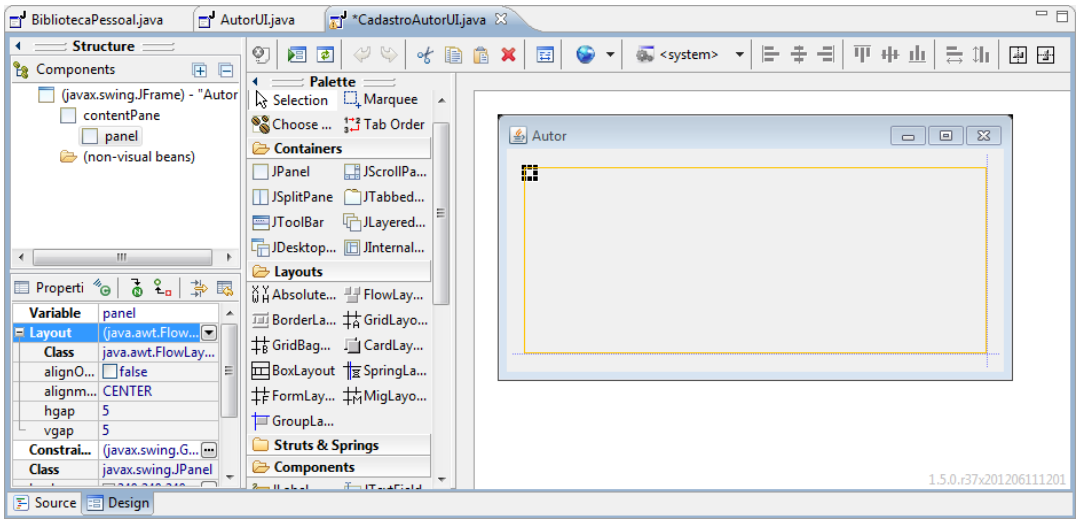


abrir imagem em nova janela

Figura 16. Posicionando JPanel no JFrame CadastroAutorUI.

Na sequência, arraste o painel até que ele fique como mostra a **Figura 17**. Após isso, gostaríamos de definir uma borda para o **JPanel**. Sendo assim, localize a propriedade **border** e pressione o botão elipse para chamar a caixa de diálogo que permite escolher uma borda para o painel. Veja a **Figura 18**.

Clique na caixa *Border type* e escolha a opção **TitledBorder**. Isso define uma borda com título, onde podemos estabelecer um título na caixa *Title*. Nesta mesma janela pode-se selecionar um alinhamento, posição e cor do título. Vamos deixar com as opções padrão e não definiremos um título agora, pois queremos que ele seja determinado durante a execução da aplicação. Ou seja, se o formulário for chamado para cadastrar um autor, então o título será "Cadastrar Autor". Caso seja para modificar um autor, então será "Alterar Autor". E isso deve ser feito através de código, mais adiante. Depois da definição da propriedade **border**, vamos inserir no **JPanel** os *widgets*. Entretanto, antes de adicionar os componentes ao painel, é necessário modificar o seu gerenciador de *layout* para **GroupLayout**, o que pode ser feito usando a propriedade **Layout**.



abrir imagem em nova janela

Figura 17. Redimensionando o JPanel.

Neste momento adicionaremos os componentes no interior do **JPanel**, iniciando pelo **JLabel** que ficará situado no canto superior esquerdo do painel, assim como vemos na **Figura 19**. Após posicionar o *widget*, digite "Nome:". Logo após, adicione um **JTextField** da forma que mostra a **Figura 20**. Arraste sua borda direita até encontrar a linha guia e modifique a propriedade **Variable** para "textNome".

Com isso feito, adicione um **JButton** no canto inferior esquerdo e digite "Salvar" (veja a **Figura 21**.), e adicione um segundo botão à direita do primeiro e digite "Cancelar".

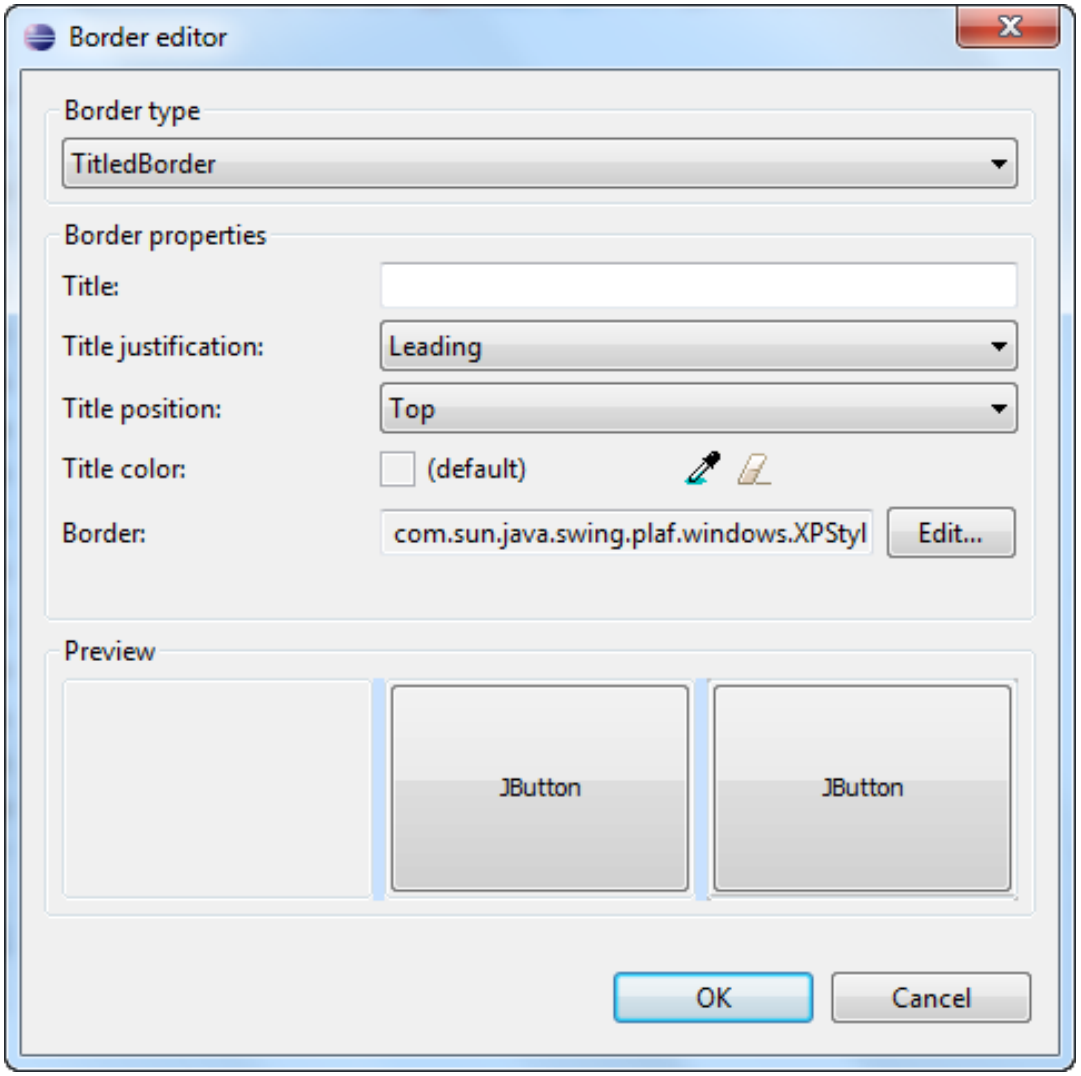
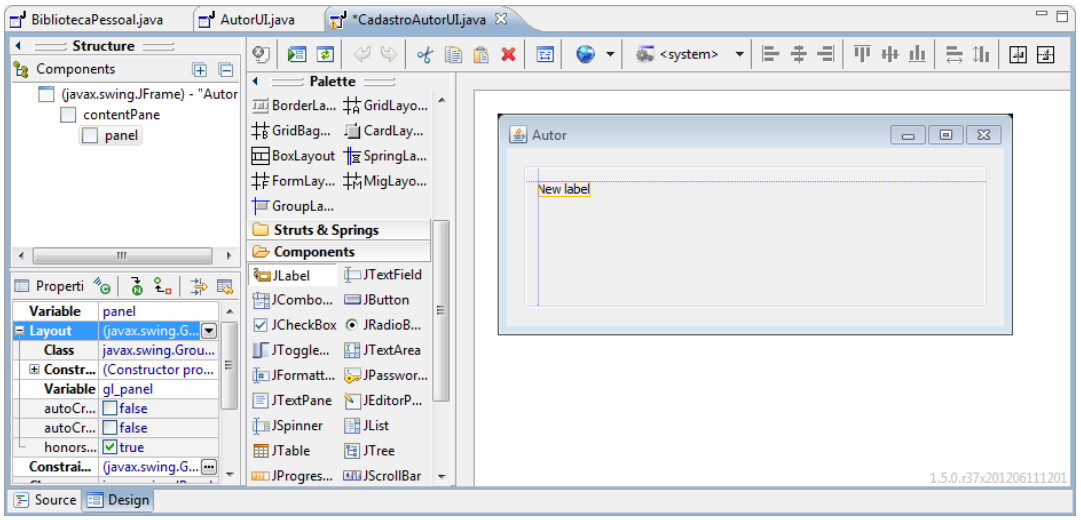
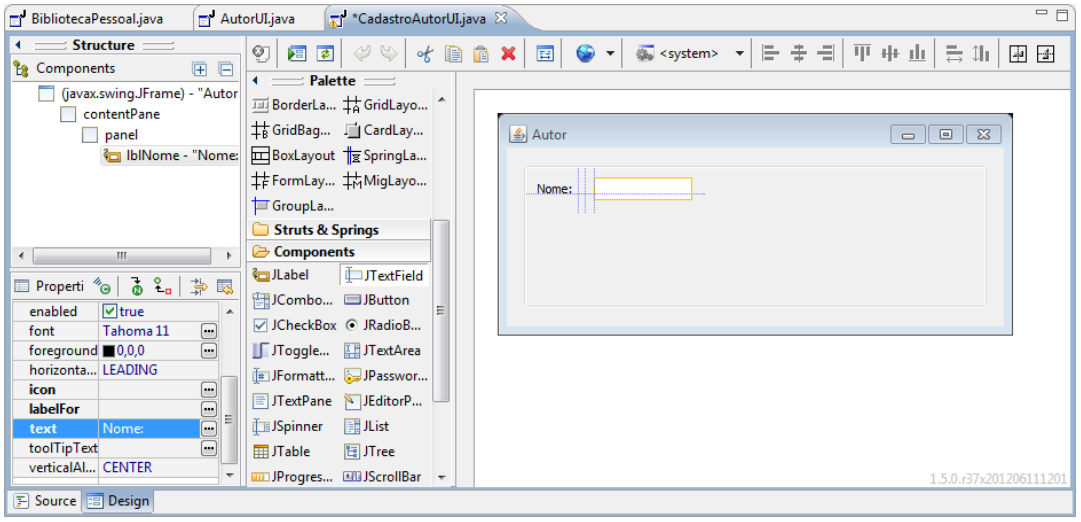


Figura 18. Selecionando uma borda para o JPanel.



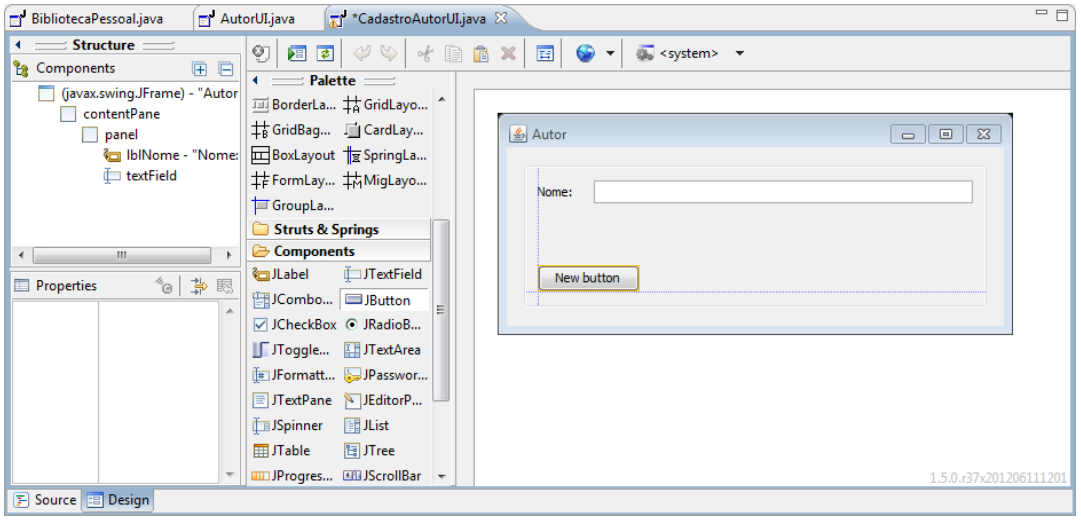
abrir imagem em nova janela

Figura 19. Adicionando um JLabel ao JPanel.



abrir imagem em nova janela

Figura 20. Adicionando um JTextField ao JPanel.



abrir imagem em nova janela

Figura 21. Adicionando um JButton ao JPanel.

Finalizada a criação da interface, vamos fazer as modificações necessárias no código da classe. Inicialmente, declare dois novos atributos na classe **CadastroAutorUI**. São eles: **private int tipo** e **private Autor autor**. Esses dois atributos serão passados como parâmetro pela classe **AutorUI** da seguinte maneira. Quando **CadastroAutorUI** for requisitada para inserir um novo autor, **tipo** será **0** e **autor** será **null**. Quando a operação for de modificação de um autor, **tipo** será **1** e **autor** será o objeto a ser modificado. Assim o código ficará como o da **Listagem 7**. As demais alterações de código serão feitas no construtor da classe.