PUC-Rio/CCE

Especialização em Análise e Projeto de Sistemas

RTP - Redes e Teleprocessamento (Programação para a Web)

Trabalho Prático - VinhosWeb

André Pfeiffer
Carlos Lucio Lampert
Igor Borges
Leticia Bacoccoli
Luís Claudio R Junior

03/12/2017

Índice

Introdução

O objetivo deste trabalho foi complementar um sistema de venda de vinhos a partir de uma implementação inicial fornecida (VinhosWeb). O sistema completo é composto por um CRUD (*Create*, *Read*, *Update* e *Delete*) de Vinhos e um de Clientes, criação de pedidos e consulta de pedidos por status. Além destes componentes, o grupo optou por incluir, como uma funcionalidade adicional, um painel gerencial do negócio.

Este relatório apresenta e descreve de forma geral como as principais funcionalidades foram implementadas ou adaptadas a partir do sistema fornecido. Também são apresentadas instruções para distribuição e configuração do projeto em um novo computador.

Configurando o projeto em um novo computador

Primeiro é necessário pegar o código do repositório do Github. Para tanto faça o clone do projeto usando o seguinte comando:

```
git clone git@github.com:luiscrjr/sysAdega.git
```

Dúvidas em relação a fazer o clone de um projeto consultar:
https://help.github.com/articles/cloning-a-repository/

Depois de clonar o repositório é necessário configurar o mesmo em seu computador, para tanto é necessário modificar os seguintes arquivos:

/.classpath

Linha 4: a versão do jre instalado no computador.
Linha 9: a versão do Apache Tomcat instalado no computador

/src/META-INF/persistence.xml

Linha 20: substituir com o endereço do local do arquivo adega.db3

Banco de dados

O banco de dados utilizado é o SQL Lite. O nome do arquivo que possui o banco de dados é adega.db3 e está localizado na raiz do projeto.
O banco de dados do sistema possui 4 tabelas:

Vinho - armazena os vinhos em estoque

- `idVinho`: chave primária da tabela
- `nomeVinho`: o nome do vinho
- `anoVinho`: o ano de produção do vinho
- `corVinho`: a cor do vinho
- `precoVinho`: o preço armazenado em double do vinho
- `qtdEstoque`: a quantidade disponível em estoque

Pedido - os pedidos realizados no sistema

- `idPedido`: chave primária da tabela
- `idCliente`: chave estrangeira para a tabela Cliente
- `dtPedido`: quando o pedido foi realizado
- `valorTotal`: o valor total do pedido incluindo todos os seus produtos
- `dtEncerramento`: data em que o pedido foi encerrado (pode ser nulo caso o pedido ainda não tenha sido encerrado)
- `estadoPedido`: o estado atual do pedido

Cliente - os clientes criados no sistema

- `idCliente`: chave primária da tabela
- `nomeCliente`: o nome completo do cliente
- `cpf`: o CPF do cliente
- `cep`: o CEP de residência do cliente
- `endereco`: o endereço de residência do cliente
- `complemento`: complemento do endereço de residência do cliente
- `numero`: o número de residência do cliente
- `bairro`: o bairro de residência do cliente
- `cidade`: a cidade de residência do cliente
- `estado`: o Estado de residência do cliente (sigla)
- `pais`: o país de residência do cliente

ItemPedido - tabela que cria o relacionamento muitos para muitos entre pedidos e vinhos. Armazena os itens de cada pedido.

- `id`: chave primária da tabela
- `idPedido`: chave estrangeira para a tabela Pedido
- `idVinho`: chave estrangeira para a tabela Vinho
- `qtdVinho`: quantidade pedida deste vinho
- `valorTotalItem`: valor total (qtdVinho * valorVinho da tabela Vinho) deste item do pedido

Hibernate

Foi utilizado neste projeto o Hibernate que é um framework para o mapeamento objeto-relacional que facilita o mapeamento dos atributos entre a base de dados que segue o padrão relacional e a aplicação que segue modelo de orientação a objetos.

Foi utilizado o mapeamento em anotações Java, que possui 4 classes que correspondem às 4 tabelas existentes no banco de dados adega.db3. Além de todas as colunas das tabelas foram mapeadas os seguintes relacionamentos:

/src/bebidas/model/Cliente.java
/src/bebidas/model/ItemPedido.java
relacionamento ManyToOne com Pedido
relacionamento ManyToOne com Vinho
/src/bebidas/model/Pedido.java
relacionamento ManyToOne com Cliente
relacionamento ManyToMany com Vinhos
relacionamento OneToMany com ItemPedido
/src/bebidas/model/Vinho.java

Foram implementados as seguintes funções:

/src/bebidas/model/ClienteManager.java
`public static boolean isCPF(String CPF)`: verifica se o CPF é válido
`public static String cadastrarCliente (String nomeCliente, String cpf, String cep, String endereco, String numero, String complemento, String bairro, String cidade, String estado, String pais )`: cadastra o cliente
`public static String editarCliente(int idCliente, String nomeCliente, String cpf, String cep, String endereco, String numero, String complemento, String bairro, String cidade, String estado, String pais )`: edita o cliente
`public static String apagarCliente( int idCliente )`: remove o cliente
`public static List<Cliente> consultarTodosClientes()`: retorna todos os clientes
`public static Cliente consultarClientePorId( int idCliente )`: retorna 1 cliente pela sua chave primária

/src/bebidas/model/PedidoManager.java
`public static String criarPedido(String[] vinhos, int idCliente, String[] qtdVinhos)`: cadastra um novo pedido
`public static String encerrarPedido(int idPedido)`: modifica o status do pedido para encerrado
`public static List<`**`Pedido`**`> consultarPedidoPorEstado(String estadoPedido)`: retorna todos os pedidos do estado solicitado

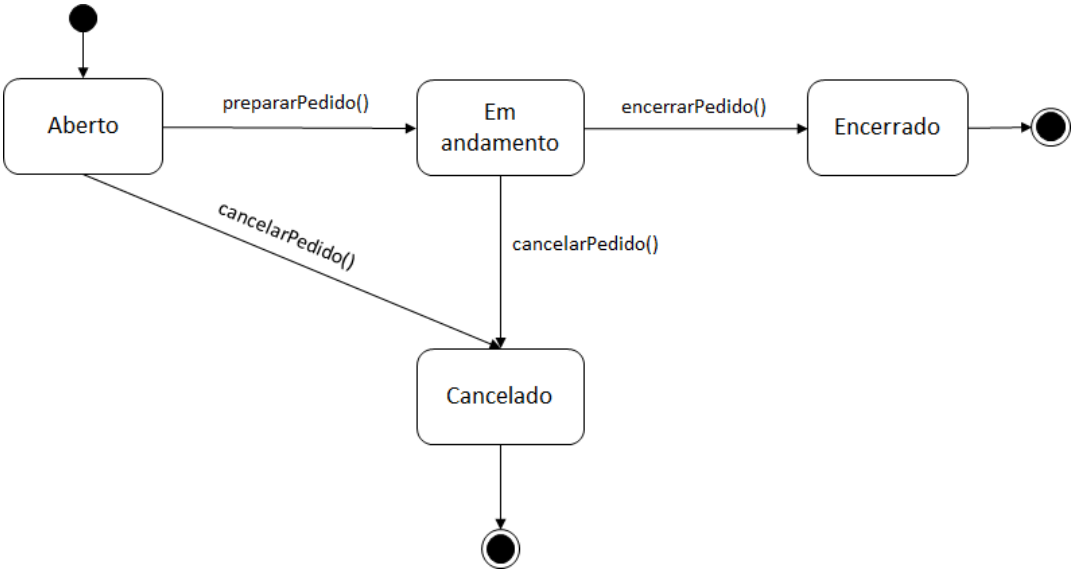
`public static String prepararPedido(int idPedido)`: modifica o status do pedido para preparar
`public static String cancelarPedido(int idPedido)`: modifica o status do pedido para cancelado

/src/bebidas/model/VinhoManager.java
`public static String cadastrarVinho( String nomeVinho, int anoVinho, String corVinho, double precoVinho, int qtdEstoque )`: cadastra o vinho
`public static String editarVinho( int idVinho, String nomeVinho, int anoVinho, String corVinho, double precoVinho, int qtdEstoque )`: edita o vinho
`public static String apagarVinho( int idVinho )`: remove o vinho
`public static List<Vinho> consultarTodosVinhos()`: retorna todos os vinhos
`public static Vinho consultarVinhoPorId( int idVinho )`: retorna 1 vinho pela sua chave primária
`public static void limparBD()`: remove todos os vinhos do banco de dados
`public static void popularBD()`: insere os vinhos padrões no banco de dados

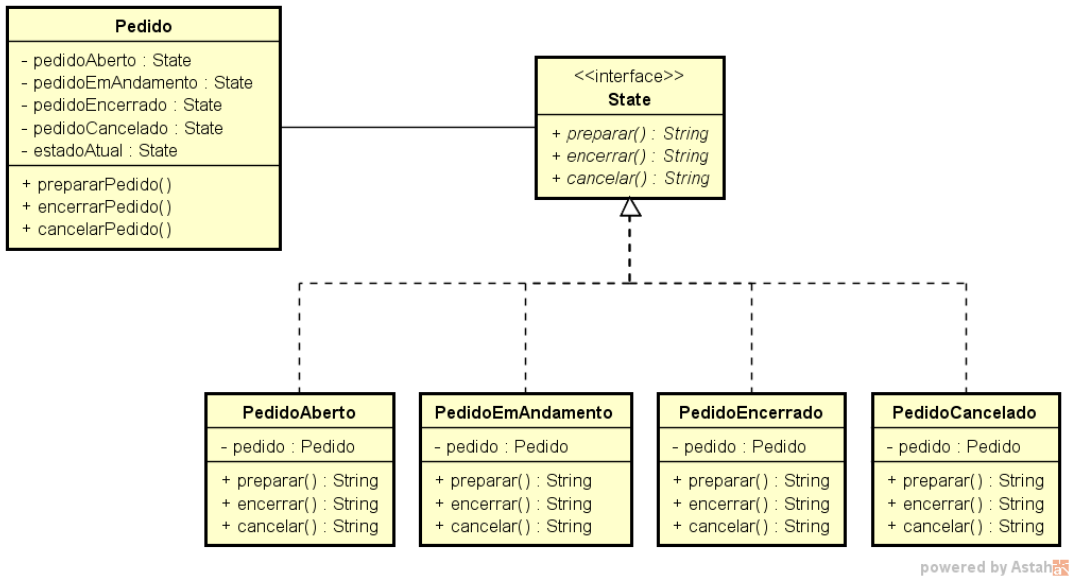
É possível também manipular todos os modelos diretamente incluindo a utilização de seus relacionamentos através das seguintes classes:
/src/bebidas/dao/ClienteDAO.java
/src/bebidas/dao/ItemPedidoDAO.java
/src/bebidas/dao/PedidoDAO.java
/src/bebidas/dao/VinhoDAO.java

Estados do pedido

Cada pedido pode variar entre 4 estados diferentes: Aberto, Cancelado, Em andamento e Encerrado, conforme a Figura abaixo.



Para facilitar a implementação possível de novos estados e manter o código organizado foi escolhido o padrão de projeto *State* para a classe Pedido, inicialmente de acordo com o diagrama de classes abaixo.



No entanto, além das variáveis e métodos acima, optamos por criar um variável de instância do tipo String “estadoPedido”, que será persistida no Banco de Dados em vez do objeto State em si. Para que isto funcione, cada vez que um objeto do tipo Pedido é retornado a partir do Banco

de Dados, é necessário redefinir sua variável de instância “estadoAtual” para o objeto State adequado.

Esta lógica foi implementada no método “setEstado(Object pedidoObj)” da classe “PedidoDAO.java”, que é utilizada também para fazer o *cast* do objeto resultado da consulta para um objeto do tipo Pedido., conforme abaixo:

```
public Pedido setEstado(Object pedidoObj) {
        Pedido pedido = null;

        if(pedidoObj instanceof Pedido) {
                pedido = (Pedido) pedidoObj;

                String estadoPedido = pedido.getEstadoPedido();
                switch (estadoPedido) {
                case "Aberto":
                        pedido.setEstadoAtual(pedido.getPedidoAberto());
                        break;
                case "Em andamento":
                        pedido.setEstadoAtual(pedido.getPedidoEmAndamento());
                        break;
                case "Encerrado":
                        pedido.setEstadoAtual(pedido.getPedidoEncerrado());
                        break;
                case "Cancelado":
                        pedido.setEstadoAtual(pedido.getPedidoCancelado());
                        break;
                }
        }

        return pedido;
}
```

Para que o usuário possa atualizar o pedido passando pelos estados apresentados acima, foram incluídos botões na página de resultado de “Consultar pedidos por status”. Em função do estado atual do pedido, são habilitados somente os botões relativos às ações possíveis para aquele pedido, conforme mostrado na imagem a seguir.

| Sobre | Ajuda | Perguntas frequentes | Entre em contato

SisAdega 2.0

Nova consulta

Pedidos no estado **todos** — Filtro

Cliente	Vinhos | Qtd	Valor total	Dt Pedido	Dt Encerramento	Estado	Ações
Roberto	Santa Helena | 1 Santa Sara | 2 Santa Maria | 3	R$ 1.120,68	24/11/2017	01/12/2017	Encerrado	
Roberto	Santa Maria | 1	R$ 345,78	26/11/2017	N/A	Cancelado	
Alberto Roberto	Santa Sara | 1 Santa Helena | 1	R$ 60,56	26/11/2017	N/A	Em andamento	
Norberto	Santa Carolina | 2 Santa Maria | 3	R$ 1.123,14	26/11/2017	N/A	Em andamento	
Letícia	Santa Sara | 5	R$ 113,90	26/11/2017	26/11/2017	Encerrado	
Norberto	Santa Carolina | 2 Santa Carolina | 2	R$ 171,60	26/11/2017	26/11/2017	Encerrado	
Norberto	Santa Carolina | 1 Santa Maria | 1	R$ 388,68	26/11/2017	N/A	Aberto	
Roberto		R$ 0,00	30/11/2017	N/A	Cancelado	
Roberto	Santa Helena | 1	R$ 37,78	30/11/2017	N/A	Aberto	

Tratamento de erros

Todas as funções do modelo tem tratamentos de erros incorporados na sua lógica de negócios e, sempre ao encontrar um erro, o código retorna uma String com a mensagem de erro. Para este tratamento funcionar é necessário que o Servlet que chama as funções de modelo receba essa String e envie para o usuário.

Na parte da visualização existe um arquivo preparado para ser incluído e ele recebe a variável da mensagem e visualiza de forma padronizada o erro ao usuário:
`/WebContent/_containerMensagens.jsp`

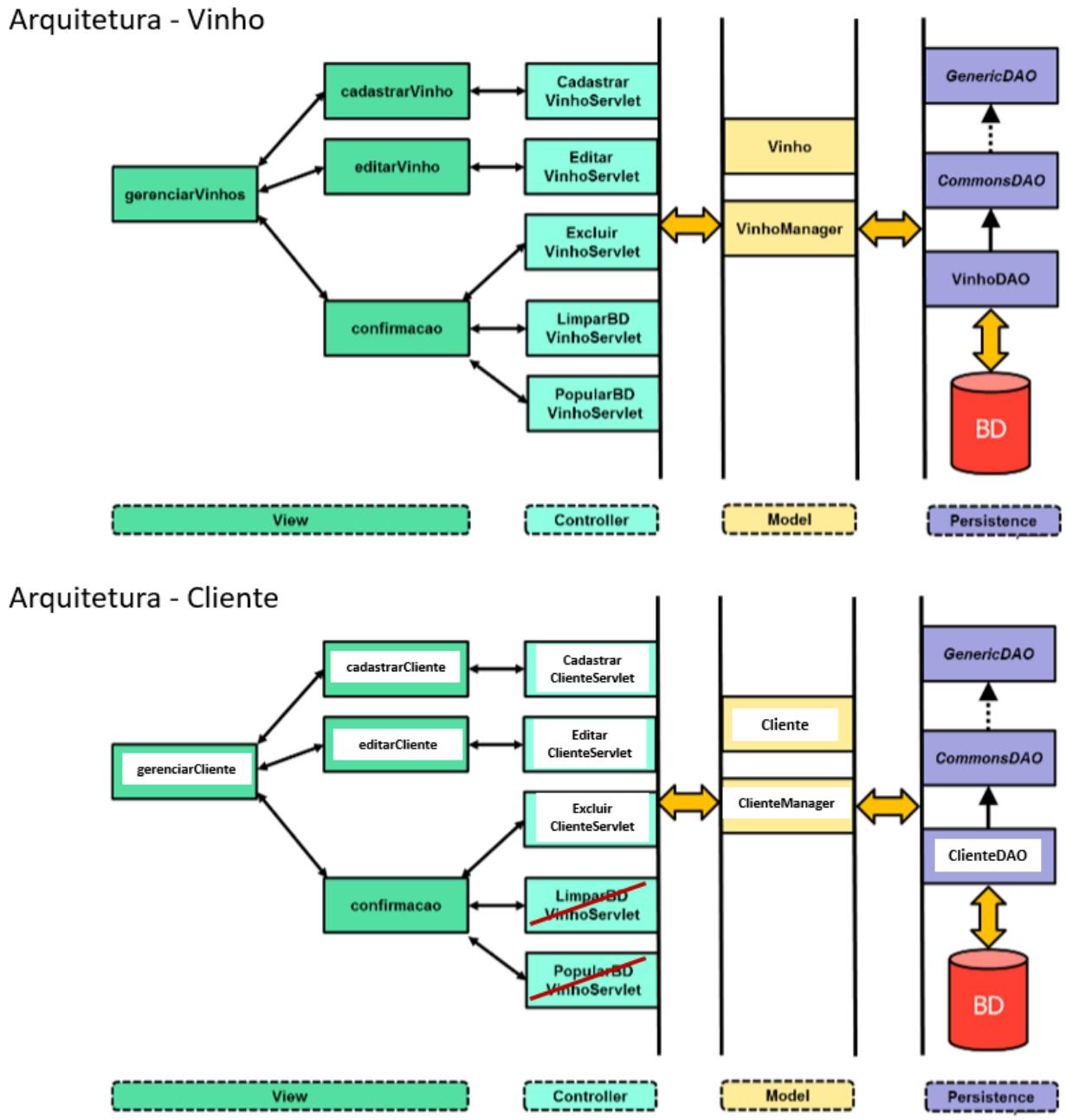
Abaixo parte do código do servlet CriarPedidoServlet.java onde o resultado da função do modelo criarPedido() é recebido em uma String que na linha abaixo é passado para o JSP com o nome “mensagem”. É exatamente a variável mensagem que o arquivo _containerMensagens.jsp está verificando e visualizando.

```
String result = PedidoManager.criarPedido(vinhos, idCliente, qtdVinhos);

request.setAttribute("mensagem", result);
```

No caso de formulários que são preenchidos pelo usuário, seja de criação ou edição, além de enviar a mensagem de erro, o servlet retorna para a página as informações preenchidas anteriormente e a página JSP se encarrega de utilizar estas informações para popular os campos do formulário, poupando o usuário do retrabalho de digitação. Esta recuperação foi feita para o cadastrarCliente.jsp, cadastrarVinho.jsp, editarCliente.jsp, editarVinho.jsp e criarPedido.jsp.

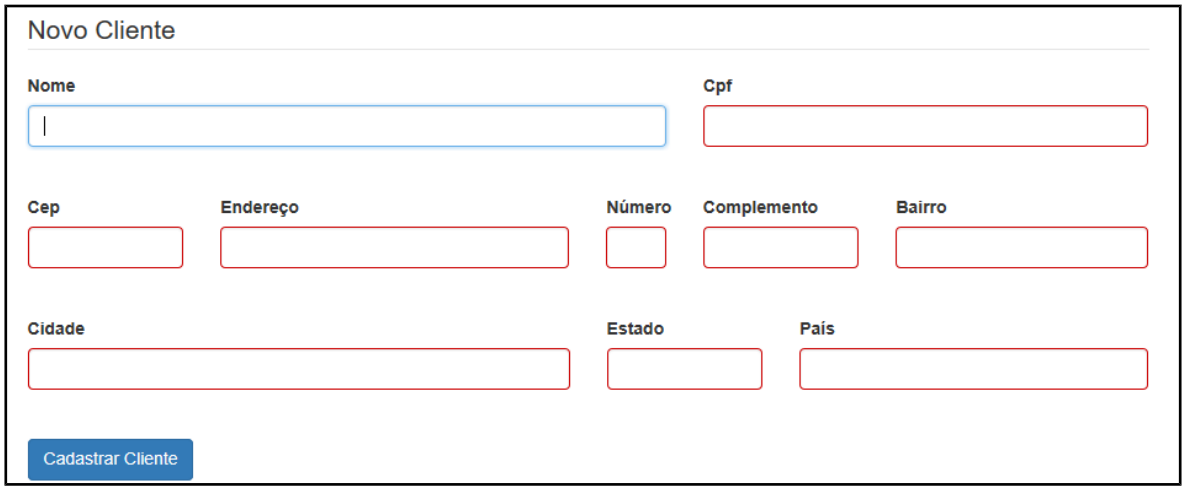
CRUD completo de clientes

O CRUD de clientes foi implementado baseado na mesma arquitetura MVC (Model, Vew, Controller) utilizada para o CRUD de vinhos do sistema original, conforme imagem abaixo.



As funcionalidades de limpar e popular Banco de Dados não foram replicadas para o sistema final, pois foram necessárias somente para testes no início do desenvolvimento da versão original fornecida.

Para preenchimento das informações do objeto Cliente, descritas no Capítulo Banco de dados, foi criado uma página JSP com o formulário abaixo.



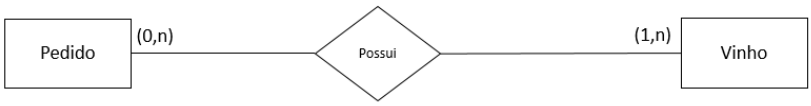
Foram incluídas verificações de preenchimento de todos os campos obrigatórios, assim como validação do Cpf, tanto nas páginas JSP (View) quanto no ClienteManager.java (Model). Para facilitar o preenchimento e minimizar erros, foi implementado o auto preenchimento do endereço a partir do preenchimento do Cep utilizando o serviço do ViaCEP (https://viacep.com.br/).

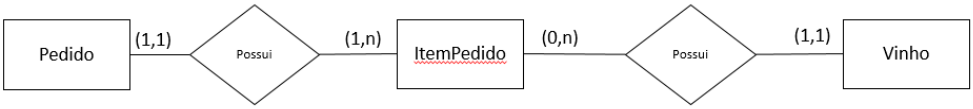
Criação do pedido

A implementação da funcionalidade de criação do pedido no sistema original permitia a inclusão de somente um tipo de vinho por pedido. Para permitir a inserção de mais de um tipo de vinho por pedido foi criada uma nova classe ItemPedido:

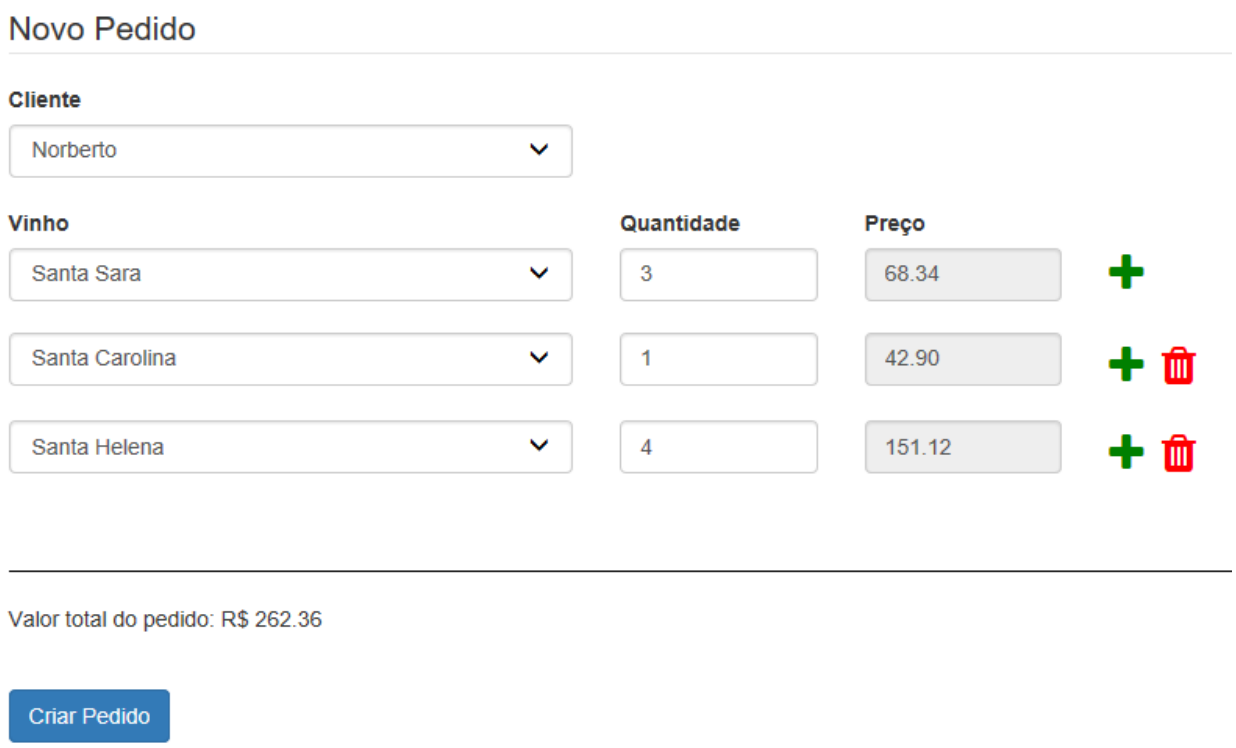
```
@Entity
@Table(name="ItemPedido")
public class ItemPedido {
        @Id
        @Column(name = "id", nullable = false)
        @GenericGenerator(name="generator", strategy="increment")
        @GeneratedValue(generator="generator")
        private int id;

        @Column(name = "qtdVinho")
        private int qtdVinho;

        @Column(name = "valorTotalItem")
        private double valorTotalItem;

        @ManyToOne(fetch = FetchType.EAGER)
        @JoinColumn(name="idPedido")
        private Pedido pedido;

        @ManyToOne(fetch = FetchType.EAGER)
        @JoinColumn(name="idVinho")
        private Vinho vinho;
        ...
```

Conforme visto no Capítulo Banco de dados, esta classe fica responsável por fazer o relacionamento “muitos para muitos” entre Pedido e Vinho transformando em dois relacionamentos “muitos para um” entre Pedido e ItemPedido e entre ItemPedido e Vinho, conforme mostrado na imagem a seguir (DER).





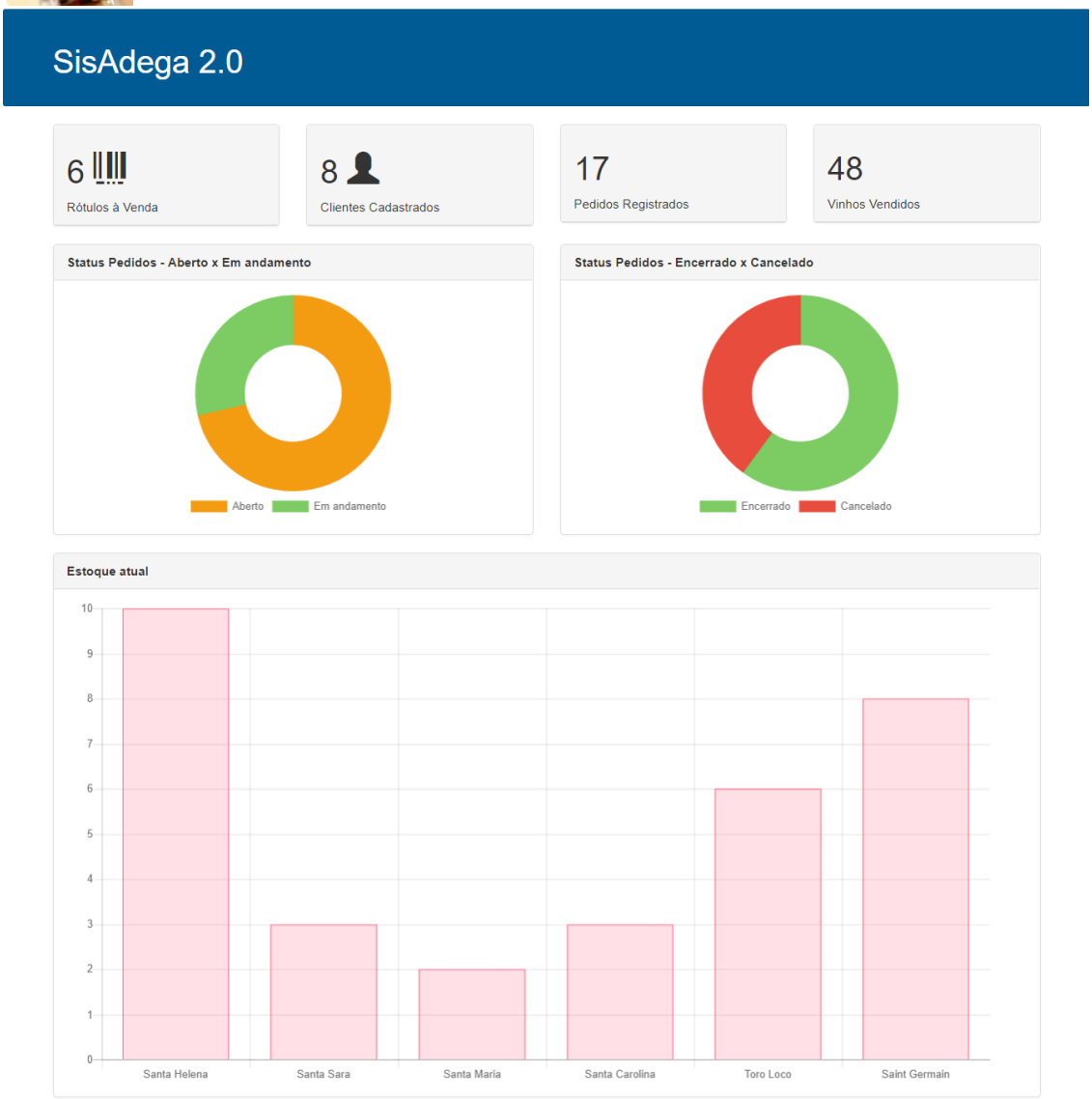
Além das alterações no modelo e banco de dados, foi necessário também adequar a página “criarPedido.jsp”, que agora permite ao usuário adicionar linhas de vinho x quantidade de forma livre, conforme imagem abaixo.



Funcionalidade adicional

Como funcionalidade adicional ao sistema, foi implementado um “Painel Gerencial” que tem como objetivo apresentar uma visão geral do negócio. Foram selecionadas algumas informações consideradas relevantes e foi implementada uma página que apresenta estas informações através de números e gráficos de forma resumida e objetiva.

A funcionalidade pode ser acessada a partir da página inicial do sistema, na opção “Painel Gerencial” e, em seguida, é apresentada uma nova página como a da figura abaixo.



Para a implementação do painel foi criada uma página JSP e foram utilizadas as classes Java existentes no sistema para acesso ao banco de dados e tratamento das informações. Para a criação dos gráficos foi utilizada a biblioteca Chart.js (http://www.chartjs.org/).

Especificamente para retornar a informação da quantidade de vinhos vendidos, foi necessário implementar dois métodos extras na classe VinhoManager:

a) uma para consultar a quantidade vendida por vinho:

```
public static int consultarQtdeVendidaVinho(Vinho vinho) {
	ItemPedidoDAO itemPedidoDAO = new ItemPedidoDAO();
	List<ItemPedido> listaItemPedido = itemPedidoDAO.selecionarPorVinho(vinho);
	if(listaItemPedido == null) {
		return 0;
	} else {
		int qtdeTotal = 0;
		for (ItemPedido itemPedido : listaItemPedido) {
			int qtdeItem = itemPedido.getQtdVinho();
			// se o pedido for cancelado, não conta como vendido
			if(itemPedido.getPedido().getEstadoPedido().equals("Cancelado")) {
				qtdeItem = 0;
			}
			qtdeTotal = qtdeTotal + qtdeItem;
		}
		return qtdeTotal;
	}
}
```

b) e outra para calcular a quantidade total de vinhos vendidos:

```
public static int consultarQtdeTotalVinhosVendidos() {
	List<Vinho> vinhos = VinhoManager.consultarTodosVinhos();
	int qtdTotal = 0;
	for (Vinho vinho : vinhos) {
		int qtdVendidaVinho = VinhoManager.consultarQtdeVendidaVinho(vinho);
		qtdTotal += qtdVendidaVinho;
	}
	return qtdTotal;
}
```