UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

IURE SAUZEN CARNEIRO

SISTEMA MULTICAMADAS PARA GERENCIAMENTO DE BARES E RESTAURANTES DESENVOLVIDO COM TECNOLOGIA DATASNAP

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2014

IURE SAUZEN CARNEIRO

SISTEMA MULTICAMADAS PARA GERENCIAMENTO DE BARES E RESTAURANTES DESENVOLVIDO COM TECNOLOGIA DATASNAP

Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de Trabalho de Diplomação, do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, da Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Profa. Beatriz T. Borsoi

PATO BRANCO
2014

ATA Nº: **254**

DEFESA PÚBLICA DO TRABALHO DE DIPLOMAÇÃO DO ALUNO **IURE SAUZEN CARNEIRO**.

Às 14:30 hrs do dia 18 de dezembro de 2014, Bloco V da UTFPR, Câmpus Pato Branco, reuniu-se a banca avaliadora composta pelos professores Beatriz Terezinha Borsoi (Orientadora), Edilson Pontarolo (Convidado) e Andrei Carniel (Convidado), para avaliar o Trabalho de Diplomação do aluno Iure Sauzen Carneiro, matrícula 606456, sob o título **Sistema multicamadas para gerenciamento de bares e restaurantes desenvolvido com tecnologia DataSnap**; como requisito final para a conclusão da disciplina Trabalho de Diplomação do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, COADS. Após a apresentação o candidato foi entrevistado pela banca examinadora, e a palavra foi aberta ao público. Em seguida, a banca reuniu-se para deliberar considerando o trabalho **APROVADO**. Às 15:05 hrs foi encerrada a sessão.

Profa. Beatriz Terezinha Borsoi, Dr.
Orientadora

Prof. Edilson Pontarolo, Dr.
Convidado

Prof. Andrei Carniel, Esp.
Convidado

Profa. Eliane Maria de Bortoli Fávero, M.Sc
Coordenadora do Trabalho de Diplomação

Prof. Edilson Pontarolo, Dr.
Coordenador do Curso

RESUMO

CARNEIRO, SAUZAN IURE. Sistema multicamadas para gerenciamento de bares e restaurantes desenvolvido com tecnologia DataSnap. 2014. 58 f. Monografia de trabalho de Conclusão de Curso - Curso de Tecnologia em Análise e Desenvolvimento de Sistemas. Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. Pato Branco, 2014.

A grande demanda de informação processada por sistemas computacionais e requisitada por seus usuários tem gerado a necessidade de desenvolvimento de aplicativos com maior capacidade de processamento e armazenamento de dados. E, que, além disso, ofereçam maior segurança de acesso aos dados e na realização das operações, maior agilidade, confiabilidade e disponibilidade de informação. Para atender a esses requisitos, além da escolha de um banco de dados consistente e capaz de armazenar e gerenciar grande volume de dados e de uma linguagem de programação eficiente é necessário utilizar uma arquitetura de software que permita atender as necessidades dos usuários. Dentre essas arquiteturas está a *DataSnap* que é multicamadas. Neste trabalho é apresentado o desenvolvimento de um software para gerenciamento de bares e restaurantes implementado utilizando a tecnologia *DataSnap*. Dessa forma, um exemplo de aplicação dessa tecnologia é disponibilizado para desenvolvedores de software.

Palavras-chave: DataSnap. Arquitetura multicamadas. Software para gerenciamento de bares e restaurantes.

ABSTRACT

CARNEIRO, SAUZAN IURE. Multi-tier system for managing bars and restaurants developed with DataSnap technology. 2014. 57 f. Monografia - Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas. Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. Pato Branco, 2014.

The great demand of information processed by computer systems and requested by its users has created the need for developing applications with greater processing power and data storage. And that, in addition, provide security of data access and conduct of the operations, flexibility, reliability and availability of information. To meet these requirements, in addition to choosing a database able to store and manage large amounts of data and an efficient programming language it is need to use a software architecture that allows meeting the system requirements. Among these architectures is DataSnap that is multi-tier. In this text it is presented the development of management software for bars and restaurants implemented using DataSnap technology. Thus, an example of application of this technology is made available to software developers.

Palavras-chave: DataSnap. Multi-tier architecture. Management of bars and restaurants.

LISTA DE FIGURAS

Figura 1 - Estrutura DataSnap 20
Figura 2 - Objeto JSON 21
Figura 3 - Array JSON 21
Figura 4 - Valor JSON........ 22
Figura 5 - SQL Developer Data Modeler – representação gráfica do banco de dados........ 26
Figura 6 - Visual Paradigm - Diagrama de Classes........ 28
Figura 7 - Diagrama de casos de uso do sistema proposto 36
Figura 8 - Diagrama de classes do sistema 37
Figura 9 - Modelo de dados desenvolvido para o sistema 38
Figura 10 - Tela Principal do Sistema 39
Figura 11 - Tela de Cadastro Padrão........ 40
Figura 12 - Comanda da mesa 41
Figura 13 - Divisão da Conta da Comanda 42
Figura 14 - Controle de Caixa 43
Figura 15 - Opção de Criação do Servidor DataSnap 44
Figura 16 - ServerContainer da Aplicação 45
Figura 17 - Tela de processos do servidor de aplicação 55

LISTA DE QUADROS

Quadro 1 – Tecnologias e ferramentas 23
Quadro 2 - Detalhamento do requisito funcional Cadastrar Comanda 32
Quadro 2 - Detalhamento do requisito funcional Dividir Conta 32
Quadro 3 - Detalhamento do requisito funcional Manutenção do Estoque 33
Quadro 4 - Detalhamento do requisito funcional Pedidos para Cozinha............................... 33
Quadro 5 - Detalhamento do requisito funcional formação de preço dos Produtos 33
Quadro 6 - Detalhamento do requisito funcional Composição de Produtos......................... 34
Quadro 7 - Detalhamento do requisito funcional Encerrar Comanda do Cliente................... 34
Quadro 8 - Detalhamento do requisito funcional Transferir Mesa... 35
Quadro 9 - Casos de uso do Sistema 35

LISTAGENS DE CÓDIGO

Listagem 1 - Métodos do cadastro padrão 41
Listagem 2 - Unit de conexão ao banco de dados 46
Listagem 3 - Implementação do Método CreatePrivate 46
Listagem 4 - Método Singleton TConexão.GetInstance 47
Listagem 5 - Implementação da Classe TBaseClass 48
Listagem 6 - Implementação da unit uAtributos.pas 49
Listagem 7 - Código da unit uPedido.pas 50
Listagem 8 - Código da classe de Persistência *uGenericDAO.pas* 51
Listagem 9 - Implementação do Método Insert da classe TGenericDAO 53
Listagem 10 - Implementação do evento *OnGetClass* 53
Listagem 11 - Implementação da unit uDsProduto.pas 54
Listagem 12 - Implementação do método Insert da classe TDSProduto 55

LISTA DE SIGLAS

API	*Application Programming Interface*
COM	*Component Object Model*
CRUD	*Create, Retrieve, Update, Delete*
DDL	*Data Definition Language*
GUI	*Graphical User Interface*
HTTP	*HyperText Transfer Protocol*
IDE	*Integrated Development Environment*
JSON	*JavaScript Object Notation*
MIDAS	*Middle-tier Distributed Applications Services*
RAD	*Rapid Application Development*
REST	*Representational State Transfer*
RTTI	*Runtime Type Information*
SDL	*Software Development Labs*
SGDB	Sistema Gerenciador de Banco de Dados
SOAP	*Simple Object Access*
SQL	*Structured Query Language*
TCP/IP	*Transmission Control Protocol/Internet Protocol*
UC	*User Case*
UML	*Unified Modeling Language*
XE	*Express Edition*

SUMÁRIO

1 INTRODUÇÃO 10
1.1 CONSIDERAÇÕES INICIAIS 10
1.2 OBJETIVOS 12
1.2.1 Objetivo Geral 12
1.2.2 Objetivos Específicos 12
1.3 JUSTIFICATIVA 12
1.4 ESTRUTURA DO TRABALHO 13
2 REFERENCIAL TEÓRICO 14
2.1 ARQUITETURA DE SOFTWARE 14
2.2 SISTEMAS DISTRIBUÍDOS 16
2.3 MODELO CLIENTE/SERVIDOR 17
2.3.1 Sistemas Multicamadas (*n-tier*) 18
2.4 ARQUITETURA MULTICAMADAS EM DELPHI 19
2.5 JSON 20
3 MATERIAIS E MÉTODO 23
3.1 MATERIAIS 23
Oracle 10g XE 23
Embarcadero® RAD Studio XE2 23
Sql Developer Data Modeler 23
Visual Paradigm for UML 23
3.1.1 Oracle 10g XE 24
3.1.2 Embarcadero® RAD Studio XE2 24
3.1.3 Sql Developer Data Modeler 24
3.1.4 Visual Paradigm for UML 27
3.2 MÉTODO 29
4 RESULTADOS 30
4.1 ESCOPO DO SISTEMA 30
4.2 MODELAGEM DO SISTEMA 31
4.3 APRESENTAÇÃO DO SISTEMA 39
4.4 IMPLEMENTAÇÃO DO SISTEMA 43
5 CONCLUSÃO 56

1 INTRODUÇÃO

A seguir são apresentadas as considerações iniciais, os objetivos e a justificativa do trabalho. Por fim está a apresentação do texto por meio da organização dos seus capítulos.

1.1 CONSIDERAÇÕES INICIAIS

Quase todas as áreas de atividade humana, como, por exemplo, as voltadas para negócios, entretenimento, saúde e educação, possuem auxílio direto ou indireto de programas computacionais. Dessa forma, exige-se dos programas cada vez mais no que se refere à qualidade, pois em geral busca-se nos sistemas melhor desempenho e gerenciamento das atividades realizadas, aumento de receitas, redução de despesas e maior segurança dos dados trabalhados.

Existem diversas tecnologias e tecnologias (ferramentas para análise e projeto, implementação, testes e gerenciamento, linguagens de programação e modelagem, *frameworks*, componentes, entre outros) para o desenvolvimento de sistemas computacionais. Cada uma delas com suas vantagens e desvantagens, aplicabilidade mais específica ou mais genérica. Associadas às ferramentas e tecnologias estão os conceitos, padrões e modelos que possuem o objetivo de desenvolver sistemas com qualidade. A qualidade está associada, também, ao atendimento dos requisitos definidos para o sistema e que correspondam às funcionalidades esperadas pelos usuários.

Dentre as tecnologias e conceitos existentes com o objetivo de desenvolvimento de sistemas com qualidade e redução do trabalho de desenvolvimento está o conceito de aplicações em camadas. Esse conceito está relacionado à possibilidade de uma aplicação, geralmente chamada de servidor de aplicação, conter todas as suas regras de negócio e permitir que essas regras sejam utilizadas por aplicações clientes (MOURAO, 2014).

A arquitetura multicamadas pode ser um diferencial na criação de software com maior qualidade, pois, as técnicas utilizadas nesta estrutura visam proporcionar a construção de um software melhor estruturado, com mais segurança. Isso porque

a camada de apresentação não tem ligação direta com a camada de armazenamento de dados (banco de dados). Desta forma também é possível realizar a troca de SGBD (Sistema Gerenciador de Banco de Dados) da aplicação com mais facilidade.

Com o crescente aumento da quantidade de dados a serem processados, torna-se necessário desenvolver sistemas em camadas, ou seja, com sua distribuição de processos dividida, evitando uso desnecessário de equipamentos e possibilitando facilitar o acesso aos dados. Assim, gera-se a possibilidade de alterações muito mais fáceis e seguras, pois as alterações serão feitas somente nas camadas necessárias no servidor de aplicação.

A tecnologia *DataSnap* facilita o desenvolvimento multicamadas porque provê uma forma facilitada de transferência de dados entre cliente e servidor. Como forma de exemplificar o uso dessa tecnologia, o presente trabalho apresenta o desenvolvimento de um software para gerenciamento de bares, restaurantes e similares utilizando-se da tecnologia *Datasnap* para ambientes multicamadas. A intenção do software é atender qualquer público desse ramo de atuação que atualmente conta com soluções de grande porte, que muitas vezes são disponíveis a valores incompatíveis com as possibilidades dos usuários e em muitos casos com interfaces de difícil usabilidade.

Bares, restaurantes e similares são caracterizados por prestação de serviços relacionados à elaboração de alimentos e atendimento a clientes no próprio estabelecimento. A rotina básica de um restaurante a *lacarte* ou de um bar, por exemplo, é o cliente fazer o pedido, que é encaminhado por um funcionário (garçom, por exemplo) para a cozinha e depois de preparado é enviado ao cliente. O controle dos pedidos geralmente é realizado por mesa que pode ser composta por vários clientes. O sistema permitirá o registro dos pedidos dos clientes, realizados em comandas em papel (em uma versão futura o sistema contará com dispositivo móvel utilizado pelo garçom para realizar o pedido), que são enviados para preparo. O sistema auxiliará o atendente de caixa no fechamento da conta e permitirá o controle das mesas que estão ocupadas, em processo de limpeza ou livres, dentre outras funcionalidades.

1.2 OBJETIVOS

A seguir está o objetivo geral e os objetivos específicos do trabalho.

1.2.1 Objetivo Geral

Desenvolver um software multicamadas utilizando a tecnologia *DataSnap* para gerenciar bares e restaurantes.

1.2.2 Objetivos Específicos

- Apresentar o uso de boas práticas de programação, tais como orientação a objetos e *design patterns* para desenvolver um software multicamadas no ambiente de desenvolvimento Embarcadero® Delphi XE2.
- Oferecer uma forma de organização dos processos realizados em bares e restaurantes, por meio de um software que permita gerenciar as principais atividades realizadas.

1.3 JUSTIFICATIVA

A crescente necessidade de acesso a informações leva as empresas a investirem em servidores maiores com hardwares robustos e links dedicados, com o objetivo de ter uma maior confiabilidade e agilidade nas informações processadas para seus clientes.

Com isso surge a necessidade de sistemas distribuídos de forma que permitam realizar atualizações e ajustes de acordo com a necessidade do cliente, pois uma aplicação com um número elevado de clientes não pode ficar presa à obrigatoriedade de atualizar todos os clientes para cada implementação ou correção efetuada em suas regras de negócio.

Com a tecnologia Embarcadero® *Datasnap* (incluído nas versões Enterprise, Ultimate e Architect do Delphi) tem-se uma forma fácil e ágil de implementar um sistema multicamadas (*Rapid Application Development* (RAD) por meio de

componentes específicos para esse tipo de implementação). É possível criar serviços de dados e de aplicação, como *Application Programming Interfaces* (API) personalizados, que podem ser acessados a partir de aplicações cliente em Windows, Web, iOS, Android, dentre outros.

Os servidores *DataSnap* suportam *Representational State Transfer* (REST), *JavaScript Object Notation* (JSON), *Component Object Model (*COM), *HyperText Transfer Protocol* (HTTP), *Transmission Control Protocol/Internet Protocol* (TCP/IP), dentre outros padrões, para uma conectividade a partir de virtualmente qualquer plataforma. As aplicações podem ser hospedadas em um servidor que pode ser servidor no conceito de nuvem.

Desta forma, viu-se oportuno empregar essa tecnologia para desenvolvimento do software proposto neste trabalho, apresentando também boas técnicas de programação orientada a objetos e notação JSON. Essa notação pode ser acessada por diversas linguagens, como Delphi, Java, PHP, Java Android, Java iOS, Visual Basic, .Net, Flex, por exemplo.

1.4 ESTRUTURA DO TRABALHO

O Capítulo 2 apresenta o referencial teórico sobre arquitetura de software, sistemas distribuídos, modelo cliente/servidor e sistemas multicamadas. No Capítulo 3 são indicados o método e os materiais utilizados durante a elaboração do estudo proposto, ou seja, a sequência geral de passos empregados e os recursos necessários para implementação do sistema. Os resultados obtidos com o trabalho, utilizando a metodologia proposta no Capítulo 3 estão contidos no Capítulo 4. Esse capítulo apresenta a modelagem, telas do sistema e partes de código. Por fim, o Capítulo 5 apresenta a conclusão.

2 REFERENCIAL TEÓRICO

Neste capítulo são apresentados conceitos relacionados à arquitetura de software, sistemas distribuídos, modelo cliente/servidor, sistemas multicamadas e tecnologia *DataSnap*.

2.1 ARQUITETURA DE SOFTWARE

"A arquitetura de sistemas é uma descrição compacta e administrável de como um sistema é organizado e de como os componentes operam entre si" (SOMMERVILLE, 2003, p. 182).

Por meio da arquitetura de software é obtida a estrutura de dados e a estrutura hierárquica de todos os componentes do software (MAZZOLA, 2003). Para Booch, Rumbaugh e Jacobson (2005) a arquitetura de software é bem mais do que a descrição dos componentes e das relações pertinentes a eles.

> "A arquitetura de software não está apenas relacionada à estrutura e ao comportamento, mas também ao uso, à funcionalidade, ao desempenho, à flexibilidade, à reutilização, à abrangência, a adequações e a restrições de caráter econômico e tecnológico, além de questões estéticas" (BOOCH; RUMBAUGH; JACOBSON, 2005, p. 34).

Ainda segundo esses autores, a arquitetura é o conjunto de decisões significativas acerca dos seguintes itens:

- A organização do sistema de software;
- Os elementos estruturais e suas interfaces que irão compor o sistema, bem como os seus respectivos comportamentos especificados a partir da colaboração entre esses elementos;
- A composição dos elementos estruturais e comportamentais em cada sub-sistema;
- O estilo arquitetural que direciona a organização desses elementos, suas colaborações e composições.

Pressman (2006) diz que a arquitetura de software deriva de um processo de divisão em partições que relaciona elementos de uma solução de software a partes de um problema do mundo real implicitamente definida durante a análise de requisitos.

Sendo assim pode-se entender que a estrutura padrão do software deve ser definida no início do projeto, a divisão do problema ou melhoria será feita em partes menores, ou seja, “pequenos problemas”. A junção da solução de cada um desses “pequenos problemas resultará na solução do problema como um todo”. Segundo Sommerville (2003), a arquitetura do sistema afeta o desempenho, a robustez e a facilidade de distribuição e de manutenção de um sistema. A estrutura e o estilo específico escolhidos para uma aplicação podem, portanto, depender dos requisitos não funcionais do sistema (SOMMERVILLE, 2003, p. 183).

Alguns desses requisitos não funcionais são (SOMMERVILLE, 2003, p. 183 e 184):

- Desempenho - a arquitetura deve ser projetada para abranger as operações mais importantes, com poucos subsistemas e menor comunicação possível entre os subsistemas;
- Proteção - uma arquitetura desenvolvida em camadas em que os itens mais importantes devem estar dispostos nas camadas mais internas e com um alto nível de validação para garantir a proteção;
- Segurança - os itens e as operações relacionados com a segurança ficam localizados em um único subsistema ou em poucos subsistemas, para que, dessa forma, sejam reduzidos os custos e os problemas de validação de segurança;
- Disponibilidade - inclusão de componentes redundantes para possibilitar a substituição e atualização dos componentes sem precisar interromper o uso do sistema;
- Facilidade de manutenção - arquitetura projetada utilizando componentes encapsulados, que possam ser facilmente e rapidamente modificados. Devem ser separados os consumidores de dados dos produtores de dados, evitando a criação de estruturas de dados compartilhadas.

O processo de projeto de arquitetura se preocupa em estabelecer um *framework* estrutural básico para um sistema. Ele envolve a identificação dos componentes principais do sistema e das comunicações entre esses componentes (SOMMERVILLE, 2003, p. 182).

Projetar e documentar uma estrutura de software tem algumas vantagens, algumas delas é a possibilidade de comunicação com os interessados, tornar

explícita a análise de sistema e a reutilização do software em grande escala.

O projeto de arquitetura pode ser desenvolvido de modos diferentes dependendo do enfoque do projetista. Porém, algumas atividades são comuns aos processos de arquitetura, podem-se destacar os seguintes (SOMMERVILLE, 2003, p. 182):

- Estruturação de sistema: o sistema é dividido em vários subsistemas principais independentes, sendo que a comunicação entre esses subsistemas é identificada;
- Modelagem de controle: os subsistemas são organizados de acordo com um modelo de controle, que diz respeito ao controle de fluxo entre os subsistemas. O controle pode ser centralizado ou baseado em eventos;
- Decomposição modular: cada subsistema é decomposto em módulos, sendo possível a utilização de dois modelos: o orientado a objetos e o de fluxo de dados.

2.2 SISTEMAS DISTRIBUÍDOS

Até o fim da década de 60 os sistemas eram em sua maioria centralizados, ou seja, dependiam de um único servidor com grande capacidade de processamento para obtenção de informações. Esta estrutura tinha um custo bastante elevado e o hardware era na maioria das vezes um *mainframe* e os terminais ligados a ele tinham pouca capacidade de processamento.

A partir da década de 70 iniciou-se a pesquisa de sistemas distribuídos a fim de dividir essa carga de processamento e melhorar dentre outros aspectos o custo/ benefício dos sistemas.

Tanenbaum (2007) define um sistema distribuído como um conjunto de computadores independentes entre si, mas que se apresenta aos seus usuários como um sistema único e coerente.

A definição de sistema distribuído pode ser apresentada de forma simplória como uma coleção de máquinas autônomas e software fornecendo a abstração de uma única máquina. Os sistemas distribuídos caracterizam-se como sistema de informação, dos quais destacam-se três tipos principais (SOMMERVILLE, 2003, p.

203):

- Sistemas pessoais: software projetado para utilização em computadores e dispositivos pessoais e que por sua vez não são distribuídos. São exemplos: jogos em celulares, agendas.
- Sistemas embutidos: são softwares desenvolvidos com funções específicas, baseados em microcontroladores. São exemplos: eletrodomésticos, dispositivos eletrônicos e brinquedos que possuem sensores e atuadores.
- Sistemas distribuídos: são sistemas que divididos em diversas máquinas interagem entre si através de uma rede. São exemplos: redes bancárias, empresas de varejo com inúmeras filiais, dentre outros.

2.3 MODELO CLIENTE/SERVIDOR

O surgimento dos computadores pessoais no início da década de 1970 trouxe a necessidade de uma utilização cada vez mais efetiva da capacidade de processamento dessas máquinas. Com a notável expansão das redes de computadores, o desenvolvimento de software tomou um novo rumo evoluindo para uma arquitetura descentralizada. Desse modo, a carga de processamento era dividida entre servidor e cliente.

Segundo Sommerville (2003, p. 187), “o modelo de arquitetura cliente-servidor é um modelo de sistema distribuído, que mostra como os dados e o processamento são distribuídos em uma série de processadores”. A grande vantagem desse sistema é que é uma arquitetura distribuída, o que permite o uso de sistemas de rede com muitos processadores distribuídos, sendo fácil, por exemplo, incluir um novo servidor na rede e integrá-lo com o restante do sistema.

No modelo cliente-servidor os clientes não recebem serviços de outros clientes, somente dos servidores. Já os servidores podem solicitar e/ou receber serviços de outros servidores, mas nunca de clientes. O cliente deve saber quais serviços são oferecidos e quais servidores os oferecem, pois quando solicitar algum serviço deve constatar qual servidor o oferece, para então fazer a requisição.

O software ou camada cliente é responsável pela interação entre o usuário e

o sistema. Nele está contida a interface por meio da qual o usuário realiza a manipulação das informações.

As regras de negócio podem ficar armazenadas no cliente, no servidor ou em ambos. Quando armazenadas no cliente, as regras são implementadas por meio de código da linguagem de programação utilizada (funções, validações, por exemplo). Já quando armazenadas no servidor, geralmente as regras de negócio são implementadas por meio de recursos do banco de dados (*triggers*, *procedures*, por exemplo).

2.3.1 Sistemas Multicamadas (*n-tier*)

A arquitetura multicamadas é considerada como uma evolução da arquitetura cliente-servidor. Essa arquitetura tem como princípio básico a eliminação da conexão direta com o servidor de banco de dados. Essa conexão fica a cargo de uma ou mais camadas responsáveis por recuperar as informações no servidor de banco de dados e disponibilizá-las a outras camadas.

De forma geral uma aplicação multicamadas é composta por pelo menos três camadas denominadas: camada de apresentação, camada de regras ou lógica de negócio e camada de banco de dados. A comunicação entre as camadas segue uma ordem na qual uma camada interage apenas com a camada imediatamente contígua a ela, sendo que cada camada tem suas funções e características.

Com essa estrutura, o processamento das regras de negócio continua sendo centralizado, mas ao contrário da arquitetura cliente-servidor, esse processamento não será feito pelo servidor de banco de dados e sim pela camada de regras de negócio que pode estar em um servidor diferente do banco de dados. Desta forma, as aplicações cliente tornam-se mais leves em termos de processamento pelo fato de que as regras e o acesso a dados serão implementados no servidor de aplicação (camada de regra de negócios) e exigindo assim menos manutenção. No caso de alteração nas regras de negócio não é necessário realizar manutenção ou reinstalar as aplicações nos clientes.

Por utilizar objetos distribuídos aliados a uma boa implementação de interfaces para execução de seus procedimentos, o sistema torna-se independente

de localização, desta forma o sistema pode estar em uma máquina apenas ou em máquinas separadas. A seguir estão listadas as características das camadas mais conhecidas de um sistema multicamadas.

A camada de apresentação, interface do usuário, camada GUI (*Graphical User Interface*) ou simplesmente camada cliente, fica localizada na máquina cliente e é responsável por promover a interação do usuário com as outras camadas do sistema. Essa camada é responsável pelas validações de telas e campos e quase que na maioria dos sistemas essa camada tem acesso somente à camada de regras de negócio. As consultas a bancos de dados são realizadas pela camada de regras de negócio que devolve à camada de apresentação o resultado das consultas.

A camada de regras de negócio, também conhecida como camada de lógica de negócio, camada de acesso a dados, camada intermediária ou servidor de aplicação, tem a função de realizar a ligação entre a camada de apresentação e o banco de dados. Geralmente é a camada que tem ligação direta com o banco de dados, sendo responsável por fazer as requisições a este e todo o seu tratamento. Pode estar conectada a diversas aplicações clientes. É chamado de servidor de aplicação, pelo fato de, geralmente ser uma máquina dedicada e com recursos de hardware elevados, sendo que nele ficam os métodos remotos e é onde são realizados o tratamento e processamento desses métodos.

A camada de banco de dados ou dados contém o sistema gerenciador de banco de dados, responsável pelo armazenamento dos dados.

2.4 ARQUITETURA MULTICAMADAS EM DELPHI

Segundo Mourao (2014), o desenvolvimento de aplicações em camadas sempre foi um capítulo à parte no Delphi. A partir do Delphi 6 o *Middle-tier Distributed Applications Services* (MIDAS) mudou de nome passando a se chamar *DataSnap*, mas na realidade o DataSnap é o MIDAS com suporte a SOAP-HTTP/XML (*Simple Object Access/Extensible Markup Language*), sendo assim o *DataSnap* é uma coletânea de tecnologias que funcionam em conjunto para facilitar o desenvolvimento de aplicações distribuídas.

Quando começou, a tecnologia era dependente de COM, porém com o lançamento do Delphi 2009 ela passou a funcionar sobre o framework DBExpress. A partir do lançamento do Delphi 2010 a arquitetura *DataSnap* evoluiu novamente, passando a permitir a criação de servidores DataSnap REST e tráfego de dados com notação JSON, tudo isso nativamente no Delphi (MOURAO, 2014).

Com o lançamento do RAD Studio XE outras novidades foram disponibilizadas no *DataSnap*. Assim, a tecnologia *DataSnap* representa uma forma de facilitar a implementação de ambientes multicamadas no desenvolvimento de software. Mesmo sendo uma forma de engenharia de software já antiga, a arquitetura multicamadas por vezes é pouco aplicada devido a sua dificuldade de desenvolvimento em algumas linguagens.

A Borland define *DataSnap* como uma tecnologia para o particionamento de aplicações utilizando a "e - infra-estrutura" do BizSnap. Sendo que BizSnap é a "e - infra-estrutura" para construção de *Web Services* (RODRIGUES, 2002, p. 8). A Figura 1 apresenta a estrutura *DataSnap*.

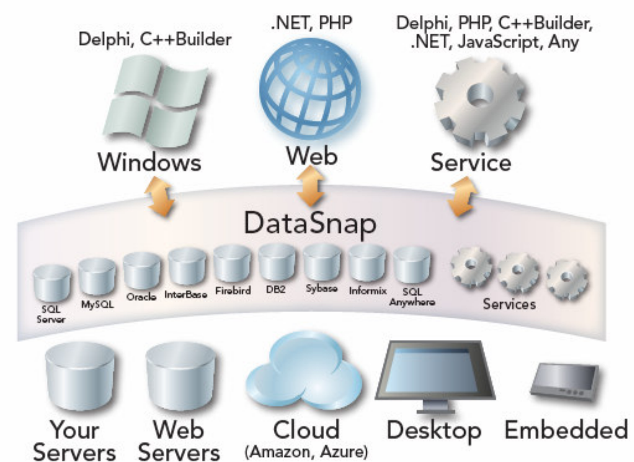

Figura 1 - Estrutura DataSnap
Fonte: LANUSSE (2014).

2.5 JSON

O JSON (notação de objetos em JavaScript) é uma formatação de troca de dados, uma espécie de padronização para transportar informações em ambientes heterogêneos. É baseado na linguagem JavaScript.

JSON é um formato de texto e tem independência de linguagens de programação. Sua principal vantagem com relação ao XML (muito utilizado para o compartilhamento de informações através da internet) é a notação do JSON ser muito mais simples, pois não utiliza *tags*, não tem regras complexas para escrita e não utiliza mecanismos complexos de *parser* (componente que faz a leitura de um documento XML e analisa a sua estrutura).

JSON está constituído em duas estruturas de dados universais que, virtualmente todas as linguagens de programação modernas a suportam, são elas (JSON, 2014):

- Uma coleção de pares nome/valor. Em várias linguagens, isto é caracterizado como um *object*, *record*, *struct*, dicionário, *hash table*, *keyed list*, ou *arrays* associativas.
- Uma lista ordenada de valores. Na maioria das linguagens isto é caracterizado como uma *array*, vetor, lista ou sequência.

No JSON um objeto é um conjunto desordenado de pares nome/valor. Um objeto começa com { (chave de abertura) e termina com } (chave de fechamento). Cada nome é seguido por: (dois pontos) e os pares nome/valor são seguidos por vírgula, como apresentado na Figura 2.

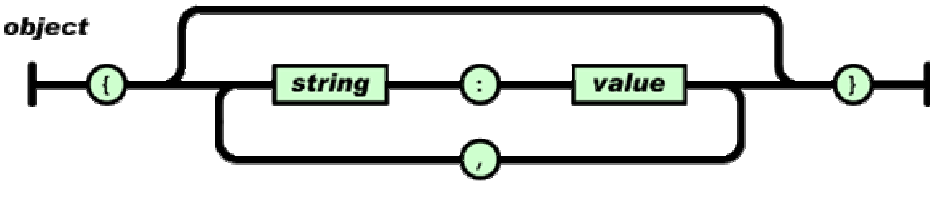

Figura 2 - Objeto JSON
Fonte: JSON (2014).

Um *array* JSON é uma coleção de valores ordenados (JSON, 2014). O *array* começa com [(colchete de abertura) e termina com] (colchete de fechamento). Os valores são separados por vírgula, como apresentado na Figura 3.

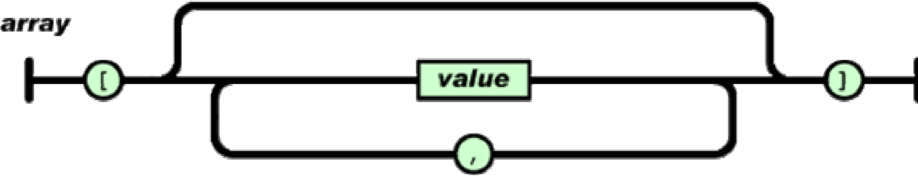

Figura 3 - Array JSON
Fonte: JSON (2014).

Como apresentado na Figura 4, um valor JSON (*value*, na Figura 3) pode ser uma cadeia de caracteres (*string*) ou um número, ou ter o valor *true/false*, ou *null*, ou um objeto ou um *array* (JSON, 2014). Essas estruturas podem estar aninhadas.

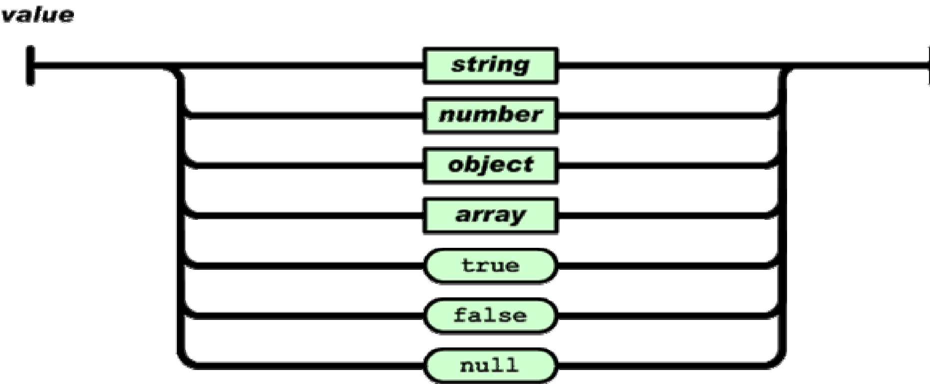

Figura 4 - Valor JSON
Fonte: JSON (2014).

3 MATERIAIS E MÉTODO

Este capítulo apresenta os materiais e o método utilizados para realizar a análise e desenvolvimento do sistema proposto. Os materiais se referem às ferramentas e às tecnologias e o método se refere aos procedimentos utilizados no ciclo de vida do sistema, abrangendo da definição dos requisitos à implementação do sistema.

3.1 MATERIAIS

As ferramentas e as tecnologias utilizadas para as atividades de modelagem, criação do banco de dados e implementação do sistema são apresentadas a seguir. O Quadro 1 apresenta um resumo das tecnologias e ferramentas utilizadas na modelagem e implementação do sistema.

Tecnologia/ Ferramenta	Versão	Tipo licença	Aplicabilidade no projeto	Vantagens	Desvantagens
Oracle 10g XE	10g XE	Free	SGBD Principal da aplicação	Livre, desempenho e confiabilidade	Armazenamento max 4GB, 1 núcleo de processamento apenas.
Embarcadero® RAD Studio XE2	XE2	Paga	IDE de desenvolvimento utilizada para confecção do protótipo	Ambiente RAD, desenvolvimento rápido e ágil utilizando VCL e demais *frameworks*	Custo de aquisição elevado. Versões somente para Windows
Sql Developer Data Modeler	4.0.3.853	Free	Desenvolvimento do Diagrama de Entidade e Relacionamento	Ambiente visual, desenvolvimento ágil e de fácil manuseio	Somente gera DDL para banco de dados Oracle
Visual Paradigm for UML	10.2	Paga	Desenvolvimento de diagramas de casos de uso e de classes	Ambiente visual de fácil compreensão e boa produtividade	Custo de aquisição e geração de códigos das classes para poucas linguagens exemplo C++ e Java

Quadro 1 – Tecnologias e ferramentas

3.1.1 Oracle 10g XE

Um banco de dados Oracle é composto pela instância e pelo banco de dados propriamente dito. Esse por sua vez consiste em uma ou mais unidades de armazenamento lógico denominadas *Tablespaces,* que armazenam coletivamente todos os dados do banco. As *Tablespaces* ficam armazenadas em arquivos físicos locais denominados *Datafiles*, que são estruturas físicas compatíveis com o sistema operacional no qual o Oracle está instalado.

Para todos os bancos de dados Oracle existe uma *Tablespace* denominada SYSTEM que armazena as informações da instância e dos controles do Oracle. Se essa *Tablespace* for excluída ou danificada a base de dados pode ser perdida.

Com o intuito de atender pequenas e médias empresas a Oracle criou a versão XE (*Express Edition*) de seus gerenciadores de bancos de dados, que é uma versão gratuita para desenvolvimento e distribuição. A versão XE tem algumas vantagens e restrições.

Vantagens: gratuidade, disponibilidade para Windows e Linux, diversos componentes de conectividade(Data Provider for .Net e OLE DB, ODBC, JDBC, PHP, C e C++) e linguagem PL/SQL compatível com a versão paga.

Desvantagens: armazenamento máximo de 4GB, memória RAM limitada a 1GB e processamento limitado a um núcleo.

3.1.2 Embarcadero® RAD Studio XE2

A Interface de Desenvolvimento Integrado, do inglês *Integrated Development Environment,* (IDE) do RAD Studio XE2 é uma evolução dos ambientes de desenvolvimento baseados em linguagem Delphi. Como diferenciais dessa versão estão: compilador para plataformas 32 bits e de 64 bits, MacOSX, iOs e Firemonkey.

3.1.3 Sql Developer Data Modeler

Sql Developer Data Modeler é uma ferramenta de modelagem de dados

grátis utilizada para aumentar a produtividade e organização do desenvolvedor.

Através dessa ferramenta é possível importar estruturas de dados, bem como exportar DDL (*Data Definition Language*) dos bancos criados. A Figura 5 ilustra a tela da ferramenta com o modelo de dados desenvolvido neste trabalho.

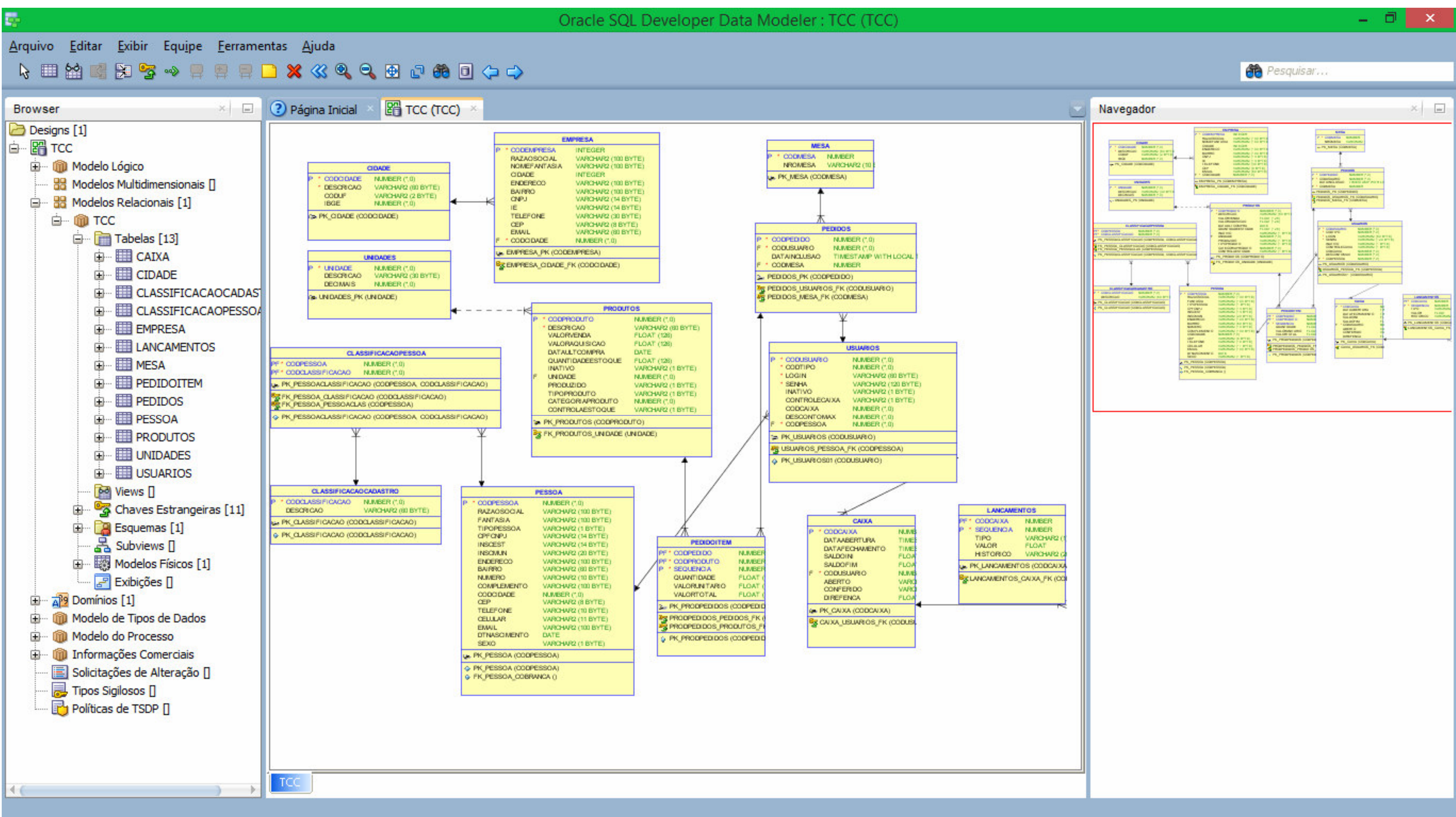

Figura 5 - SQL Developer Data Modeler – representação gráfica do banco de dados

3.1.4 Visual Paradigm for UML

O Visual Paradigma é uma ferramenta para desenvolvimento de diagramas *Unified Modeling Language* (UML) tais como casos de uso, diagrama de classes, sequência e atividades dentre outros. Com essa ferramenta também é possível em determinadas versões gerar diagramas de entidade e relacionamento bem como sua DDL. A Figura 6 ilustra a tela do Visual Paradigm com o diagrama de classes do sistema.

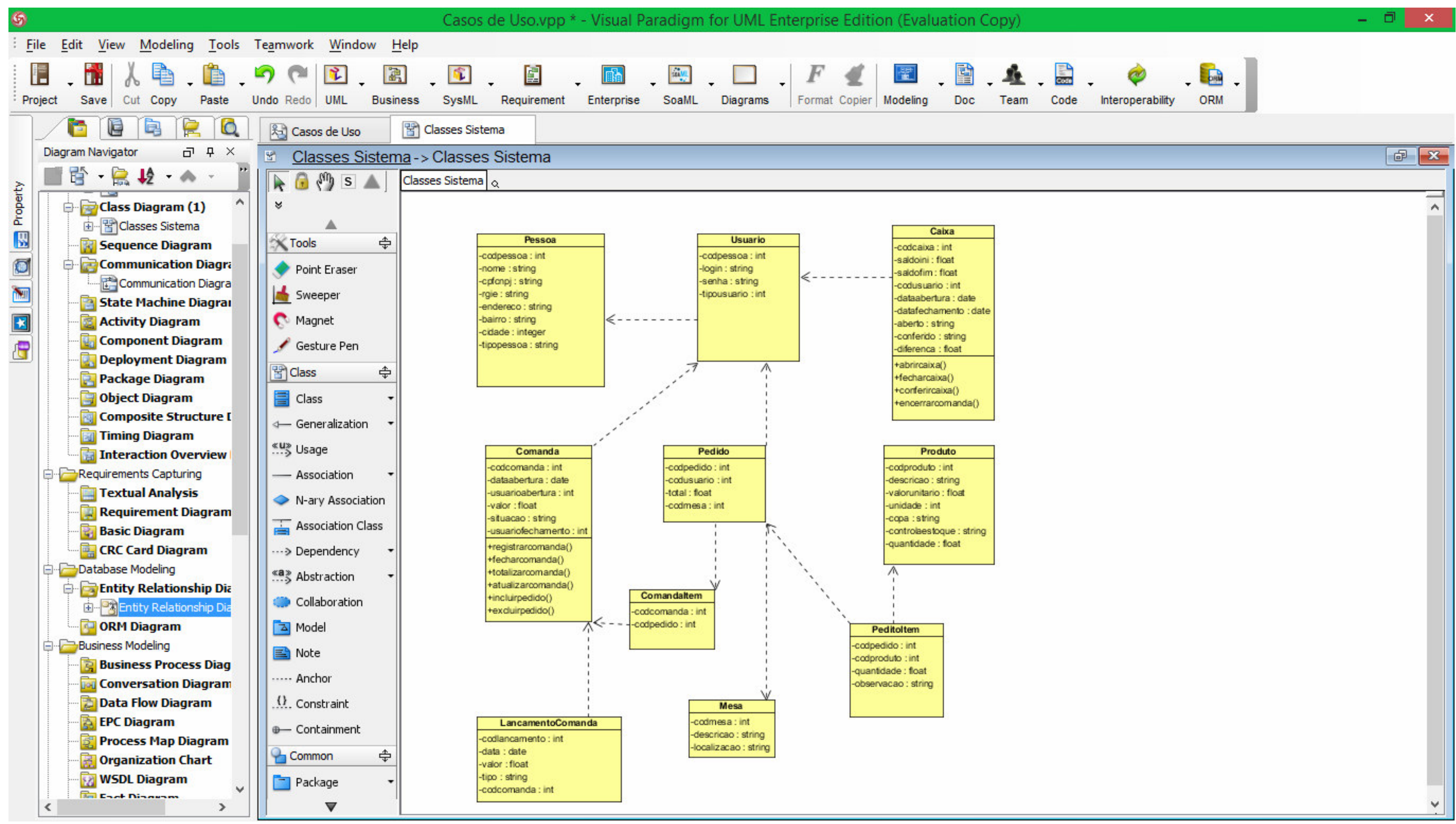

Figura 6 - Visual Paradigm - Diagrama de Classes

3.2 MÉTODO

O presente trabalho teve como base a identificação da necessidade de um software para controle de bares, restaurantes e similares que seja flexível, de fácil manuseio e baixo custo de implantação.

Os requisitos foram levantados e definidos tendo como base a necessidade de desenvolver um sistema que auxilie administradores de bares, restaurantes e similares na gestão de processos do dia a dia. Os requisitos foram levantados com base na observação da rotina diária de estabelecimentos do ramo de bares e restaurantes. Foi identificada a necessidade de maior mobilidade no atendimento ao cliente e para isso será aplicada a implementação em *tablet* para uso pelos garçons.

Com base na definição de requisitos gerada foram implementados os diagramas da UML para ilustrar o processo a ser implementado pelo sistema.

O levantamento dos requisitos e a modelagem dos mesmos, além do referencial teórico, foram realizados como atividade de estágio do autor deste trabalho.

O desenvolvimento do software foi realizado com Delphi XE2 utilizando tecnologia *DataSnap* para implementação do servidor e cliente multicamadas.

Os testes da aplicação foram realizados pelo autor deste trabalho. Os testes foram informais e com o objetivo de identificar erros de código.

4 RESULTADOS

Neste capítulo são apresentados os resultados da realização deste trabalho, que é a modelagem de um sistema para gerenciamento de bares e restaurantes.

4.1 ESCOPO DO SISTEMA

Com o sistema proposto pretende-se atender as necessidades de bares, restaurantes e similares no que se refere ao controle de estoque, pedidos, caixa e atendimento ao cliente através de um *tablet* manuseado pelo garçom.

Desta forma faz-se necessária a criação de rotinas para manutenção de estoque de produtos realizados por meio de compras, as quais movimentam o estoque e realizam também movimentações financeiras. Outras rotinas necessárias são: o controle das comandas e dos pedidos de clientes e a rotina para fechamento da comanda do cliente. Ao realizar o fechamento de uma comanda, o usuário caixa alterará a situação da comanda para fechada e então os lançamentos financeiros da comanda serão devidamente ligados ao caixa.

Como funcionalidade desejável tem-se a integração realizada pelo *tablet* com o sistema. Por meio deste dispositivo o garçom realizará a inclusão de novas comandas bem como de novos pedidos para os clientes. Também será possível informar ao cliente os preços de produtos e se solicitado o valor de sua conta totalizando a comanda do cliente.

O sistema será desenvolvido de forma que atenda aos mais diversos portes de empresas do segmento de bares e restaurantes, sendo necessário realizar uma análise no cliente sobre a infraestrutura adotada para servidor, a fim de escolher o melhor banco de dados para executar a aplicação.

O fechamento da comanda do cliente será o principal foco da aplicação, pois essa rotina faz a principal movimentação do estabelecimento, ou seja, o controle financeiro. No momento em que o usuário de caixa realizar o fechamento da comanda serão listados todos os pedidos do cliente totalizando o consumo e juntamente com isso quais foram os garçons que incluíram o pedido e observações

que tiveram em cada pedido.

4.2 MODELAGEM DO SISTEMA

A seguir são apresentados os requisitos definidos para o sistema e os diagramas modelados a fim de representar a estrutura e os processos do sistema proposto nesse trabalho.

O levantamento de requisitos desenvolvido para o sistema em questão é apresentado na forma de requisitos funcionais e não funcionais. Os requisitos funcionais representam as funcionalidades consideradas necessárias para o funcionamento do sistema. Os requisitos não funcionais explicitam regras de negócio, restrições, entre e outros.

Por convenção, a referência a requisitos pode ser feita por meio do identificador do requisito, usando a sigla F para requisitos funcionais e NF para requisitos não funcionais. A identificação desses é feita indicando o tipo de requisito seguido de um número identificador do requisito. Neste esquema o número utilizado é criado sequencialmente, que determina que aquele requisito seja único.

O Quadro 2 apresenta o requisito funcional Cadastrar Comanda e os requisitos não funcionais relacionados.

F1 Cadastrar Comanda	**Oculto ( )**			
Descrição: O sistema deve oferecer ao usuário atendente a possibilidade de cadastrar e alterar as comandas das mesas.				
Requisitos Não-Funcionais				
Nome	**Restrição**	**Categoria**	**Desejável**	**Permanente**
NF 1.1 Incluir Produtos na comanda	Sistema deve possibilitar ao usuário incluir produtos na comanda da mesa.	Interface	( )	(x)
NF 1.2 Tablet	Será necessário a disponibilização de um tablet para o garçom incluir os produtos na comanda do cliente.	Hardware	(x)	(x)
NF 1.3 Apresentação das mesas	As mesas devem ficar dispostas com seu número e status visíveis.	Usabilidade	(X)	(X)
NF 1.4 Conta do cliente	Ao acessar a mesa o sistema deve mostrar o total da conta do cliente.	Usabilidade	(X)	(X)
NF 1.5 Pedido para cozinha	Ao incluir um produto da cozinha o sistema deve enviar o pedido para a cozinha imprimindo o mesmo.	Usabilidade	(X)	()
NF 1.6 Não permitir exclusão de produto da comanda	O sistema deve validar para que os garçons não possam excluir os produtos inseridos na comanda.	Segurança	(X)	(X)
NF 1.7 Baixa do estoque	Ao Inserir produto na comanda o sistema deve fazer a baixa automática dos produtos do estoque.	Usabilidade	(X)	(X)

Quadro 2 - Detalhamento do requisito funcional Cadastrar Comanda

O Quadro 3 apresenta o requisito funcional Dividir Conta e os requisitos não funcionais relacionados.

F2 Dividir Conta	Oculto ( )			
Descrição: O sistema deve possibilitar a divisão da conta pela quantidade de pessoas da mesa.				
Requisitos Não-Funcionais				
Nome	Restrição	Categoria	Desejável	Permanente
NF 2.1 baixar conta de pagamentos parciais	Sistema deve proporcionar funcionalidade de efetuar baixa parcial da conta de acordo com a quantidade de pessoas	Usabilidade	(X)	( )
NF 2.2 Salvar Histórico de baixas parciais	Ao realizar baixa parcial de conta sistema deve salvar histórico de baixas	Usabilidade	(X)	( )
NF 2.3 Calcular saldo da comanda	Ao ser solicitado sistema deve efetuar o cálculo do saldo da comanda considerando as baixas parciais.	Usabilidade	(X)	( )

Quadro 3 - Detalhamento do requisito funcional Dividir Conta

O Quadro 4 apresenta o requisito funcional Manutenção de Estoque e os requisitos não funcionais relacionados.

F3 Manutenção de Estoque	**Oculto ()**			
Descrição: Sistema deve fazer a manutenção do estoque.				
Requisitos Não-Funcionais				
Nome	**Restrição**	**Categoria**	**Desejável**	**Permanente**
NF 3.1 Entrada/saída de produtos	Disponibilizar possibilidade do usuário realizar a entrada e a saída de produtos no estoque.	Interface	(X)	(X)
NF 3.2 Compra de produto	Sistema deve disponibilizar a compra de produtos realizando a entrada dos produtos no estoque.	Usabilidade	(X)	(X)
NF 3.3 Gerar conta a pagar	Sistema deve gerar fatura a pagar caso a compra seja gerada a prazo ou debitar do caixa se paga no ato.	Usabilidade	(X)	(X)
NF 3.4 Saldo do caixa	Sistema deve validar para permitir compras somente com valor menor do que o saldo do caixa.	Segurança	(X)	(X)

Quadro 4 - Detalhamento do requisito funcional Manutenção do Estoque

O Quadro 5 apresenta o requisito funcional Pedidos para Cozinha e os requisitos não funcionais relacionados. Esse requisito

F4 Pedidos para Cozinha	**Oculto ( )**			
Descrição: Imprimir pedidos diretamente na cozinha ao incluir os mesmos na comanda.				
Requisitos Não-Funcionais				
Nome	**Restrição**	**Categoria**	**Desejável**	**Permanente**
NF 4.1 Impressão do pedido	Sistema deve gerar um pedido que será impresso diretamente na cozinha para informar o que deve ser feito e a mesa que deve ser entregue.	Interface	(X)	(X)

Quadro 5 - Detalhamento do requisito funcional Pedidos para Cozinha

O Quadro 6 apresenta o requisito funcional Formação de Preço dos Produtos.

F5 Formação de Preços dos Produtos	**Oculto ( )**			
Descrição: Sistema deve possibilitar ao usuário fazer a formação de preço dos produtos.				
Requisitos Não-Funcionais				
Nome	**Restrição**	**Categoria**	**Desejável**	**Permanente**
NF 5.1 Alterar Preços	Possibilitar usuário alterar preços de uma determinada faixa de produto.	Interface	( )	(X)
NF 5.2 Alterar preço por grupo de Produtos	Possibilitar usuário selecionar um determinado grupo de produtos e atualizar os valores.	Usabilidade	( )	(X)
NF 5.3 Gerar combo	Sistema deve ter a possibilidade de trabalhar com combos de produtos.	Usabilidade	(X)	( )

Quadro 6 - Detalhamento do requisito funcional formação de preço dos Produtos

O Quadro 7 apresenta o requisito funcional Composição de Produtos e os requisitos não funcionais relacionados.

F6 Composição de Produtos	**Oculto ( )**			
Descrição: Sistema deve possibilitar ao usuário formar a composição dos produtos.				
Requisitos Não-Funcionais				
Nome	**Restrição**	**Categoria**	**Desejável**	**Permanente**
NF 6.1 Composição do produto	Sistema deve apresentar ao usuário a possibilidade de formar a composição dos produtos feitos na cozinha.	Interface	( )	(X)
NF 6.2 Separar produtos por categoria	Os produtos para composição devem ser separados por categoria.	Usabilidade	(X)	( )
NF 6.3 Validar tipo de Produto para composição	Sistema deve validar para que somente os produtos marcados como de cozinha possam ser compostos e os produtos que irão compor o mesmo também não possam ser de venda avulsa.	Segurança	( )	(X)

Quadro 7 - Detalhamento do requisito funcional Composição de Produtos

O Quadro 8 apresenta o requisito funcional Encerrar Comanda do Cliente.

F7 Encerrar Comanda do cliente	**Oculto ( )**			
Descrição: Sistema deve possibilitar ao usuário registrar o encerramento da comanda do cliente.				
Requisitos Não-Funcionais				
Nome	**Restrição**	**Categoria**	**Desejável**	**Permanente**
NF 7.1 Fechar conta do cliente	Sistema deve encerrar comanda do cliente somando todos os pedidos da comanda.	Interface	( )	(X)
NF 7.2 Realizar lançamentos financeiros dos recebimentos	Sistema deve realizar lançamentos financeiros dos recebimentos feitos pelo cliente na baixa da comanda.	Usabilidade	(X)	(X)
NF 7.3 Impressão do comprovante da comanda	Sistema deve realizar a impressão da comanda do cliente com os produtos consumidos e os recebimentos efetuados	Usabilidade	(X)	( )

Quadro 8 - Detalhamento do requisito funcional Encerrar Comanda do Cliente

O Quadro 9 apresenta o requisito funcional Transferir Mesa e os requisitos não funcionais relacionados.

F8 Transferir Mesa	**Oculto ( )**			
Descrição: Sistema deve possibilitar a troca de mesas pelo cliente.				
Requisitos Não-Funcionais				
Nome	**Restrição**	**Categoria**	**Desejável**	**Permanente**
NF 8.1 Transferir comanda	Sistema deve possibilitar a troca de comanda de uma mesa para outra.	Interface	( )	(X)
NF 8.2 Bloquear	Sistema deve bloquear a troca de	Segurança	(X)	( )

troca por mesa ocupada	uma mesa por outra já ocupada.			

Quadro 9 - Detalhamento do requisito funcional Transferir Mesa

A partir da definição dos requisitos do sistema, foram elaborados os casos de uso de forma simplificada. No Quadro 10 são apresentados os casos de uso identificados por meio dos requisitos levantados. A sigla UC significa *User Case* (Caso de Uso).

Nome	Atores	Descrição
UC 01 Registrar comanda	Garçom	Ao realizar atendimento ao cliente, o garçom deve efetuar o registro da comanda bem como os pedidos feitos pelo cliente.
UC 02 Dividir conta	Caixa	Ao ser solicitado o acerto da conta, se houver mais de uma pessoa para pagamento o caixa pode realizar a divisão da comanda mostrando ao cliente o valor que cada um deve acertar.
UC 03 Manutenção de estoque	Administradores	Usuários com privilégios administrativos farão inclusão e baixa de produtos do estoque.
UC 04 Pedidos para cozinha	Garçom	Ao realizar a inclusão do pedido do cliente o sistema fará a impressão dos pedidos diretamente na cozinha para produtos definidos no seu cadastro como “Copa”.
UC 05 Formação de preços de vendas	Administradores	Administradores farão a formação de preços de vendas de determinadas categorias de produtos.
UC 06 Composição de produtos	Administradores	Usuários com perfil administrativo farão a composição dos produtos feitos pela cozinha tais como lanches, pizzas, etc..
UC 07 Registrar pagamento de conta	Caixa	Usuário do caixa fará a baixa da comanda do cliente registrando o pagamento da conta do mesmo.
UC 08 Transferir conta	Garçom	Garçom poderá através do *tablet* realizar a troca de mesa de uma comanda para outra mesa disponível.

Quadro 10 - Casos de uso do Sistema

Na Figura 7 está representado o diagrama de casos de uso do sistema. Nela, os casos de uso são relacionados com os atores: Garçom, Caixa e Supervisor.

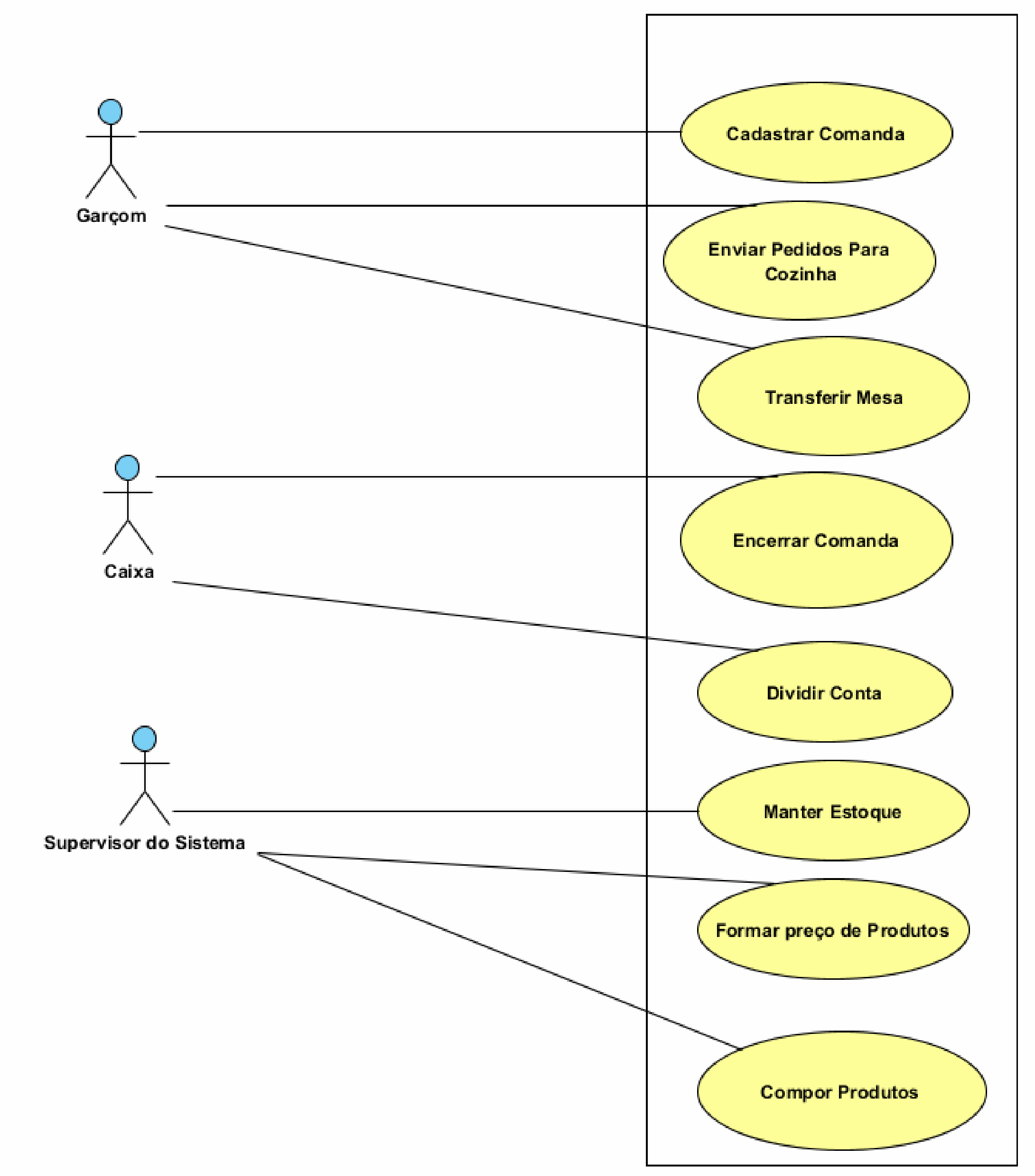

Figura 7 - Diagrama de casos de uso do sistema proposto

A Figura 8 apresenta o diagrama de classes elaborado para o sistema. Esse diagrama foi elaborado utilizando UML.

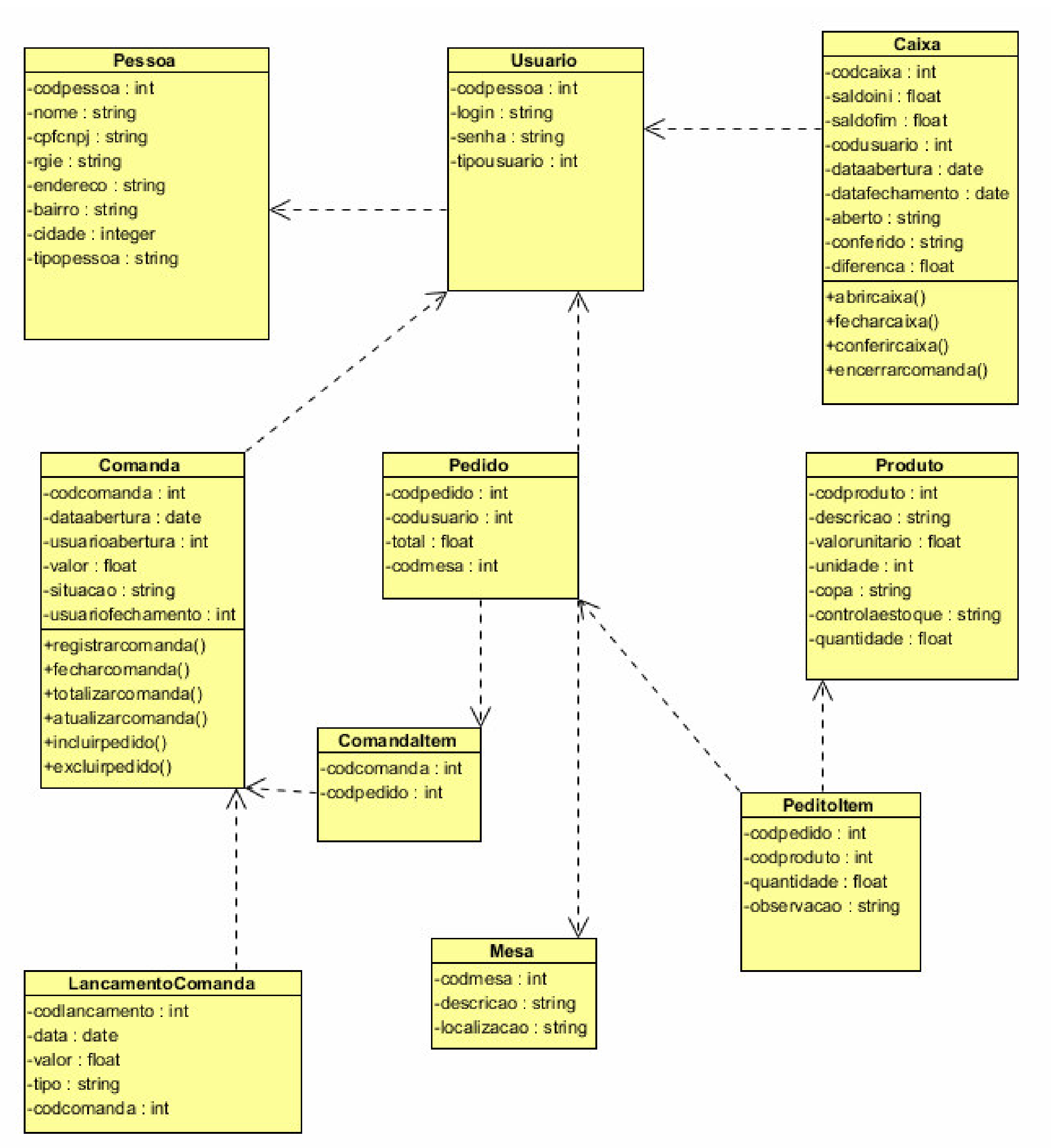

Figura 8 - Diagrama de classes do sistema

A Figura 9 representa o modelo de dados que deve ser aplicado para construção do sistema. Esse diagrama foi concebido com auxílio da ferramenta Sql Developer Data Modeler. Esse diagrama foi elaborado utilizando UML.

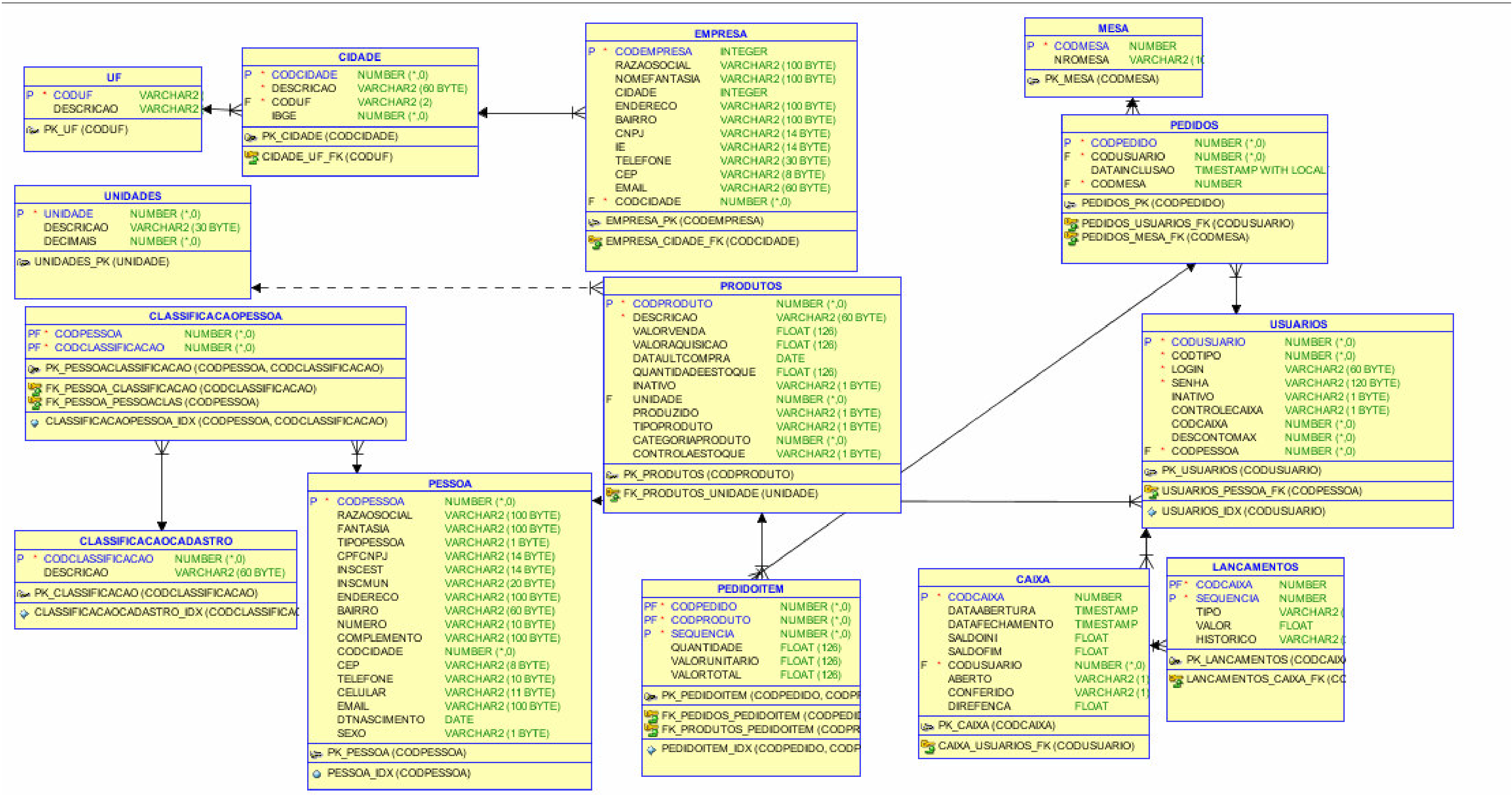

Figura 9 - Modelo de dados desenvolvido para o sistema

4.3 APRESENTAÇÃO DO SISTEMA

Após a conexão com o servidor de aplicação sistema mostrará a tela principal a qual permitirá ao usuário acesso a todas as funcionalidades do software. A tela principal está apresentada na Figura 10**.** O software foi desenvolvido utilizando o componente *TdxRibbon* o qual disponibiliza um leiaute semelhante ao do Microsoft Office®, desta forma separam-se as funcionalidades do sistema em abas.

Todas as telas do sistema serão abertas em abas. Assim, o usuário não precisa fechar uma tela para acessar outras rotinas do sistema, somente alternará entre as abas.

Na parte superior são apresentados os menus também em forma de abas que são formados por:

a) Principal: permite ao usuário sair do sistema, realizar troca de usuário, abrir e fechar o caixa.

b) Cadastros: cadastros de pessoas, empresas, classificações de pessoa, mesas, entre outros.

c) Estoque: cadastros de unidades, categorias de produtos, produtos e, ainda, rotinas de compras, ajustes de estoque e formação de preços.

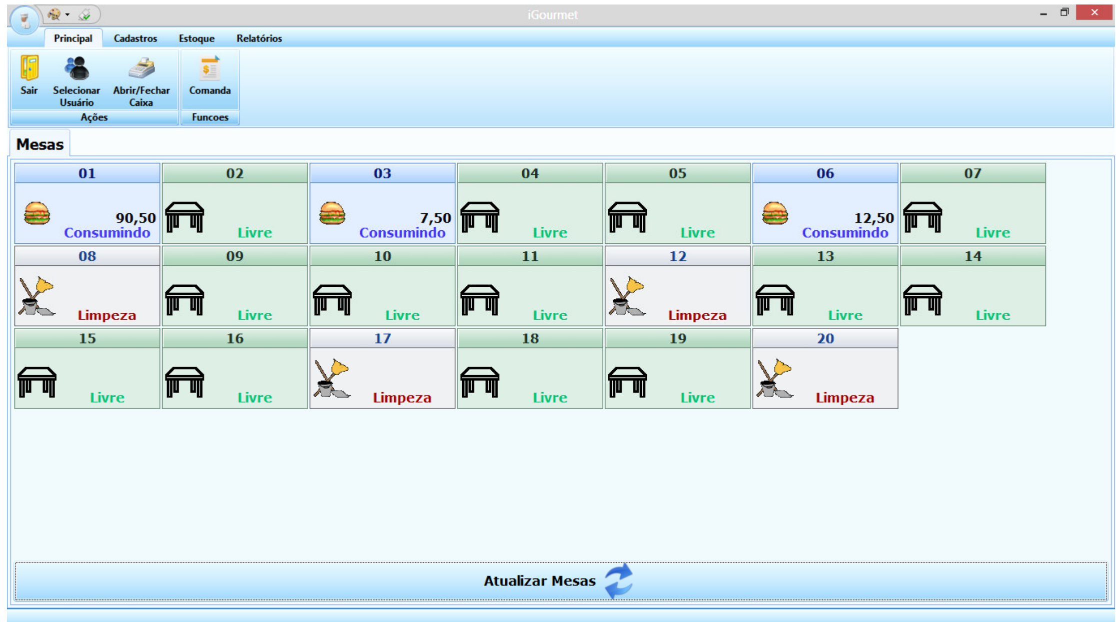

Figura 10 - Tela Principal do Sistema

Utilizando o paradigma de orientação a objetos todos os cadastros implementados na aplicação herdam funcionalidades de uma classe base denominada *TfCadPadrao.*

A tela de cadastro base implementa as principais funcionalidades comuns a qualquer cadastro na aplicação cliente, como validações, leiaute e métodos do tipo *virtual e abstract* que permitem que em classes que herdam desta, possam ser sobrescritos ou sobrecarregados conforme os padrões de polimorfismo.

A Figura 11 representa a tela de cadastro padrão, que é utilizada como superclasse para as demais telas de cadastro que implementam as especializações necessárias.

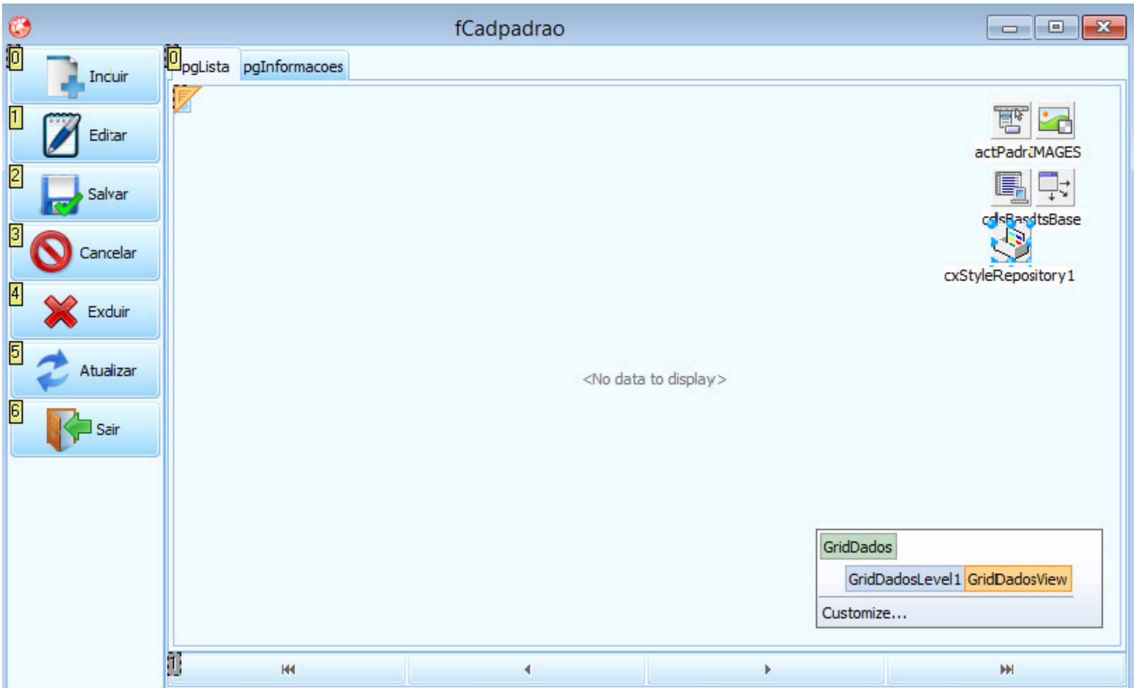

Figura 11 - Tela de Cadastro Padrão

A Listagem 1 apresenta alguns dos métodos que a tela de cadastro padrão possui e estes podem ser reescritos na utilização de formulários filhos. O escopo *protected* permite que os formulários filhos deste possam utilizar e sobrescrever seus métodos.

```
protected
  {Variável Disponível aos filhos que indica o estado do formulário}
  property EstadoDoForm : TEstadoForm read EstadoForm;
```

```
{Método responsável por realizar a consulta das Informações no servidor}
procedure GetDadosTabela; virtual; abstract;
{Método responsável por atribuir informações do ClientDataSet para um Objeto}
procedure CdsToObject; virtual; abstract;
{Método responsável pelas validações de tela}
procedure Validate; virtual; abstract;
{Eventos ligados aos Botões do Cadastro}
function ExcluirRegistro : boolean ; virtual; abstract;
procedure IncluirRegistro; virtual; abstract;
function SalvarRegistro : boolean; virtual; abstract;
```

Listagem 1 - Métodos do cadastro padrão

Por meio da comanda da mesa são lançados os pedidos de produtos feitos pelo cliente. Na comanda o usuário do sistema selecionará o produto, a quantidade para incluir na comanda. É também por meio dessa mesma tela que o usuário fará o recebimento e o fechamento da comanda do cliente realizando a movimentação do caixa.

Outra funcionalidade que esta tela implementa é a de prover a divisão da conta por um determinado número de pessoas, auxiliando o usuário no fechamento da conta como demonstra a Figura 12.

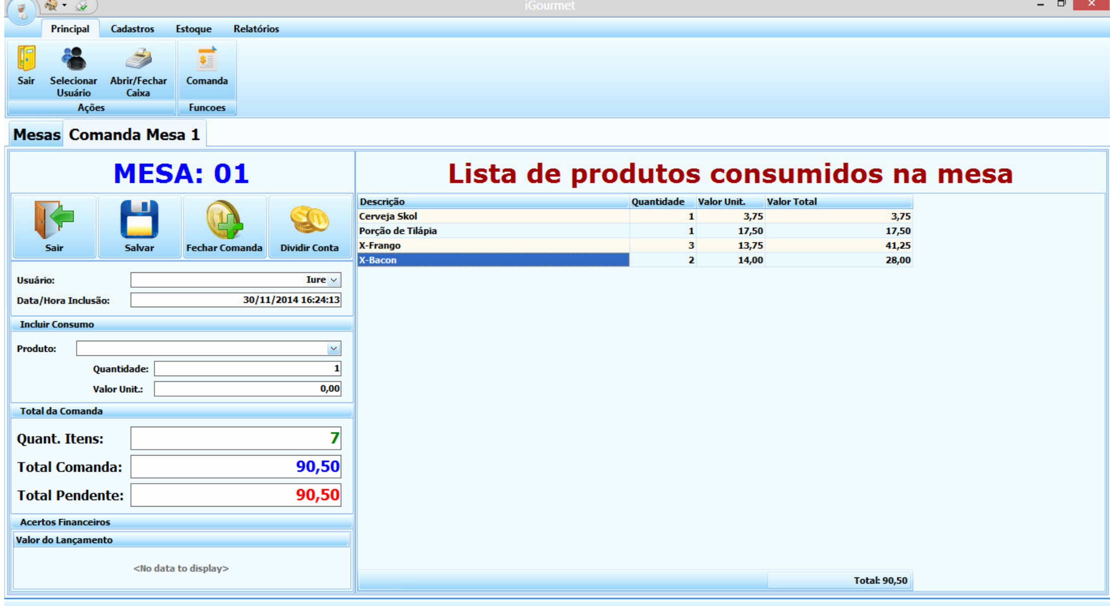

Figura 12 - Comanda da mesa

Se o cliente solicitar divisão da conta entre as pessoas que farão o pagamento, o usuário deve acessar a rotina para dividir conta que está na tela da comanda da mesa. Dessa forma, o sistema fará a divisão da conta pela quantidade de pessoas informadas, ao marcar "quitar" cada registro sistema fará o cálculo do valor que já foi quitado. A Figura 13 ilustra a tela de divisão da conta.

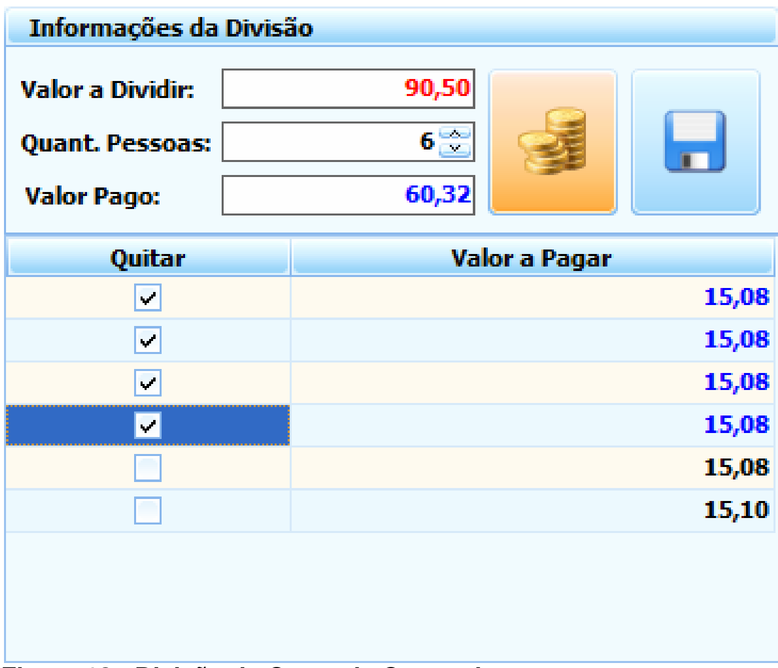

Figura 13 - Divisão da Conta da Comanda

Considerando que todos os pedidos encerrados devem fazer parte de um caixa, assim que realizado o encerramento do pedido, os lançamentos serão armazenados no caixa e mostrados ao usuário. Também no caixa o usuário poderá visualizar cada lançamento feito em cada encerramento.

Para encerrar um pedido deve obrigatoriamente haver um caixa aberto para que os valores sejam inseridos neste caixa. A inclusão é realizada pela tela de controle de caixa. Nessa tela o usuário pode realizar a abertura e fechamento de caixa e outras operações pertinentes. A Figura 14 apresenta a tela de controle de caixa.

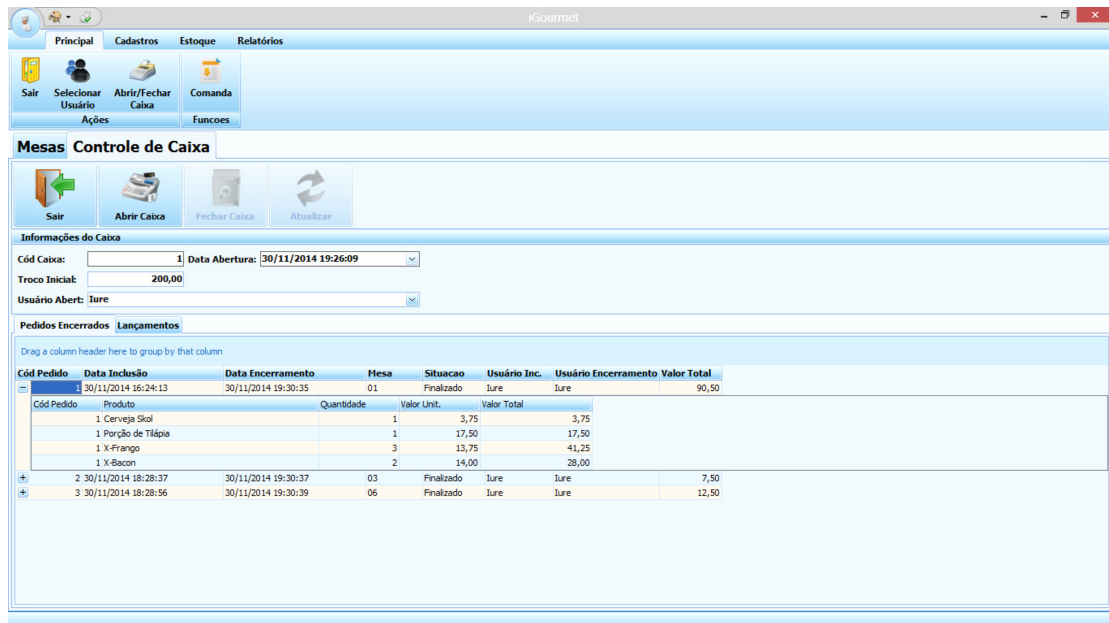

Figura 14 - Controle de Caixa

4.4 IMPLEMENTAÇÃO DO SISTEMA

O sistema foi desenvolvido com auxílio do *RAD Studio XE2,* que agiliza o processo de criação de projetos. Para a criação de um novo projeto *DataSnap* deve-se acessar o menu *File/New/Other,* selecionar a opção *DataSnap Server* e selecionar as opções desejadas conforme mostra a Figura 15.

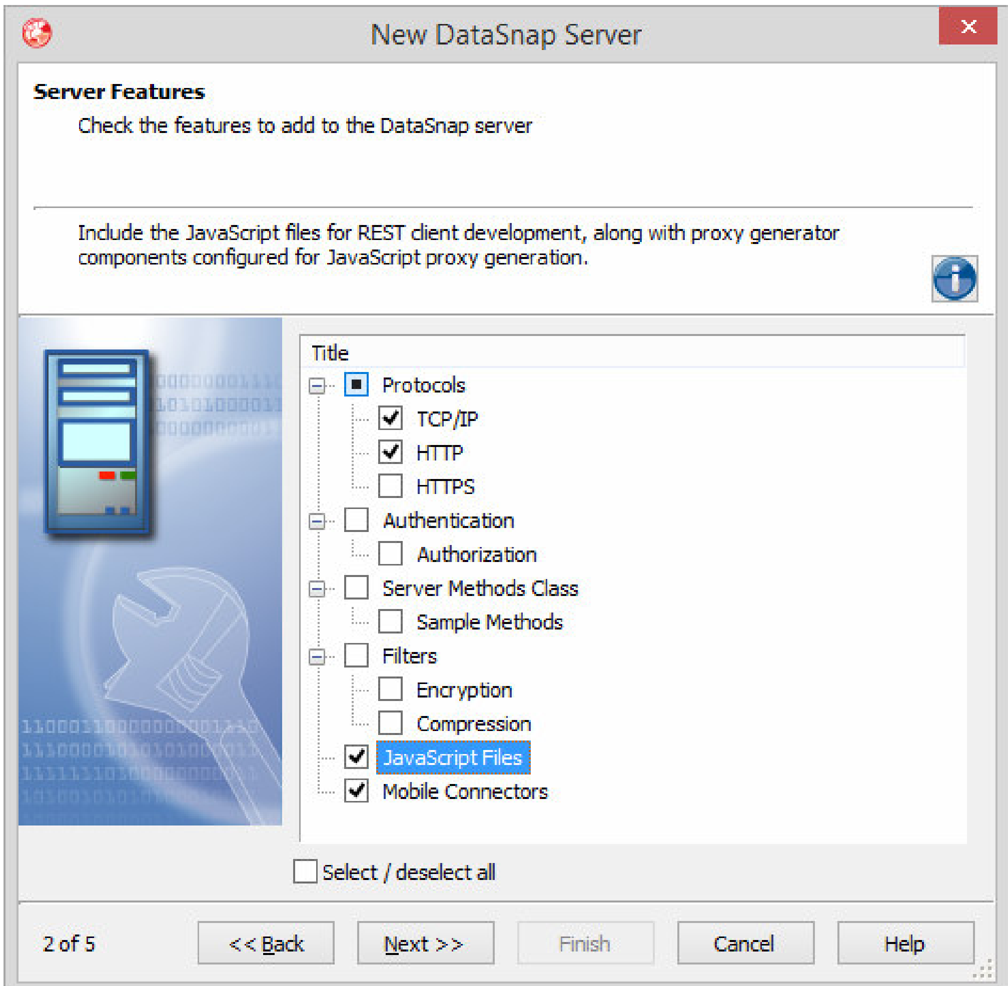

Figura 15 - Opção de Criação do Servidor DataSnap

Na configuração do DataSnap (Figura 15) são disponibilizadas opções para que ao criar o servidor com o RAD Studio XE2 sejam apresentadas configurações prontas agilizando o desenvolvimento. Resumindo cada opção tem a seguinte funcionalidade:

- TCP/IP: prover comunicação via rede, com base no protocolo de transporte TCP/IP entre a camada cliente e o servidor.
- HTTP: prover comunicação via http por REST com clientes (Delphi, C++, JavaScript e PHP).
- HTTPS: comunicação segura para os mesmos clientes de http.
- Authentication: controle de acesso ao servidor mediante usuário e senha.
- Server Methods Class: Implementa uma *unit* para serem alocados métodos que serão disponibilizados as aplicações clientes.

- Mobile Connectors: incluem suporte à geração de classes para integração de dispositivos móveis tais como Android, iOS e BlackBerry.

Após a criação do projeto o *RAD Studio XE2* gera uma classe chamada *ServerContainer* que é um *DataModule* no qual estão situados os componentes responsáveis pela comunicação entre servidor e clientes, ilustrado na Figura 16.

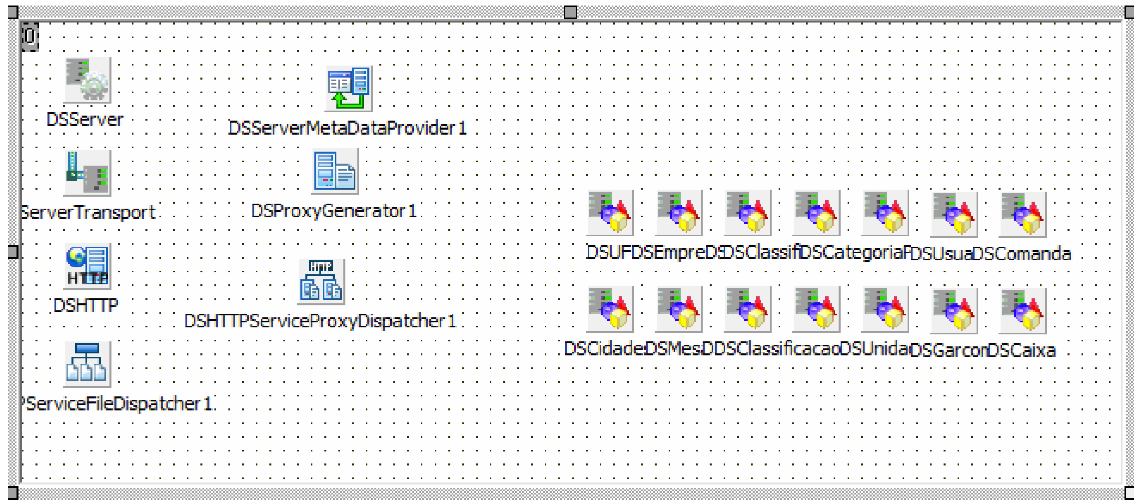

Figura 16 - ServerContainer da Aplicação

O servidor de aplicação tem como funcionalidade principal prover a comunicação entre o banco de dados e as requisições efetuadas pelos clientes. Dentre essas requisições destacam-se as operações *CRUD* (*Create, Retrieve, Update, Delete*), para tais operações deve-se configurar o acesso ao banco de dados.

No *RAD Studio XE2* existem componentes para essa funcionalidade, porém no desenvolvimento deste projeto foi utilizada uma classe de conexão abordando técnicas de orientação a objetos e padrão *singleton*.

A classe de conexão foi criada a partir do menu *File/New/Unit* e salva com o nome de "*uConnection .pas*". A Listagem 2 apresenta a codificação da *Unit*.

```
unit uConnection;
interface
uses
 DbxCommon, DbxOracle, DbxFirebird, Classes, sysUtils, uConfigDB, DataSnap.dbclient,
uConstantes;
type
 TConexao = class
 strict private
   class var FInstance : TConexao;
   constructor CreatePrivate;
 private
```

```
    FConexao: TDBXConnection;
    FComando: TDBXCommand;
    FProperties : TConfDB;
    FTipoBanco: TTipoBanco;
    procedure SetComando(const Value: TDBXCommand);
    procedure setConexao(const Value: TDBXConnection);
  public
    constructor Create;
    property Comando : TDBXCommand read FComando write SetComando;
    property Conexao : TDBXConnection read FConexao write setConexao;
    property TipoBanco  : TTipoBanco read FTipoBanco;
    // Singleton da Conexao
    class function GetInstance : TConexao;
    function ExecuteQuery(cSql : string) : TDBXReader;
    function GetNextID(cTableName, cField : string; cWhere : string = '') : Integer;
    function GetConsulta(cName : string) : string;
  end;
```

Listagem 2 - Unit de conexão ao banco de dados

Analisando o código da Listagem 2 pode-se observar que foram importadas para a classe outras três *units, DBXCommon,* DbxOracle *e* DbxFirebird. A *unit* DBXCommon permite utilizar uma fábrica de conexão e com a mesma é possível criar conexões conforme houver necessidade. As *units* DbxOracle e DbxFirebird, são utilizadas para permitir a conexão com os bancos Oracle e Firebird como é apresentado no código que está na Listagem 3, para conexão com outros bancos de dados é necessário somente importar a class DBX correspondente, por exemplo: DbxDb2, DbxInformix , DbxMSSQL e DbxMySql.

```
constructor TConexao.CreatePrivate;
begin
  // Informa no Log a Criação da Classe de Conexão
  FServidor.Logar('Criando Conexão Com banco de Dados!');
  // Le as Configurações gravadas no Arquivo .Ini
  FProperties := TConfDB.Create;
  // Valoriza o Tipo de Banco da conexão
  FTipoBanco  := FProperties.TipoBanco;
  // Cria a Conexão
  FConexao                                                                    :=
TDBXConnectionFactory.GetConnectionFactory.GetConnection(FProperties.NomeConexao,
FProperties.Usuario, FProperties.Senha);
  // Cria o Comanda que sera utilizado no acesso aos dados
  FComando    := FConexao.CreateCommand;
  // Valoriza o Tipo de Comanda para SQL
  FComando.CommandType := TDBXCommandTypes.DbxSQL;
  // Informa no Lod o Tipo de Banco Conectado
  case FProperties.TipoBanco of
    tFirebird : fServidor.Logar('Conectado a Base Firebird');
    tOracle   : fServidor.Logar('Conectado a Base Oracle');
  end;
end;
```

Listagem 3 - Implementação do Método CreatePrivate

Na Listagem 4 é realizada a chamada ao método T*DBXConnectionFactory.GetConnectionFactory.GetConnection* que é responsável por criar um objeto de conexão com o banco de dados. Nessa chamada são passados os parâmetros de nome da conexão, usuário e senha os quais foram armazenados em um arquivo *ini*. O resultado da chamada deste método é valorizado na propriedade FConexao da classe TConexao.

Para melhor utilização desta classe e para garantir unicidade desta conexão foi utilizado o padrão de projeto *Singleton*, que é definido como um padrão de projeto de criação de objetos cujo objetivo é garantir uma única instância e acessível de forma global e uniforme para toda classe que implemente este padrão (FREEMAN; FREEMAN, 2005, p. 150). Conforme esta definição a Listagem 4 apresenta o código para a implementação da chamada *Singleton* da classe TConexao.

```
class function TConexao.GetInstance: TConexao;
begin
  if not Assigned(FInstance) then
    FInstance := TConexao.CreatePrivate;
  Result := FInstance;
end;
```

Listagem 4 - Método Singleton TConexão.GetInstance

Um dos objetivos deste trabalho é apresentar benefícios da tecnologia multicamadas. Uma questão que é sempre levantada nesta arquitetura é a transferência de informações entre aplicações clientes e servidoras. Tendo em vista que essa comunicação é feita pela rede é importante procurar a forma mais leve de transferência de informações. A Listagem 5 apresenta a classe *TBaseClass* que implementa métodos de conversão de objetos para *JSON* e vice-versa. Esses métodos são utilizados em todos os objetos para transferência entre cliente e servidor.

```
unit uBaseClass;

interface
uses
  DBXJSON, DBXJSONReflect;

type
  TBaseClass = class
  public
    class function ObjectToJSON<T: class>(aObject: T): TJSONValue;
    class function JSONToObject<T: class>(aJSON: TJSONValue): T;
  end;

implementation

{ TBaseClass }

class function TBaseClass.JSONToObject<T>(aJSON: TJSONValue): T;
var
  UnMarshal: TJSONUnMarshal;
begin
  if aJSON is TJSONNull then
    Exit(nil);
  UnMarshal := TJSONUnMarshal.Create;
  Result := T(UnMarshal.Unmarshal(aJSON));
  UnMarshal.Free;
end;

class function TBaseClass.ObjectToJSON<T>(aObject: T): TJSONValue;
var
  Marshal: TJSONMarshal;
begin
  if Assigned(aObject) then
    begin
    Marshal := TJSONMarshal.Create(TJSONConverter.Create);
    Result := Marshal.Marshal(aObject);
    Marshal.Free;
  end
  else
    Exit(TJSONNull.Create);
end;

end.
```

Listagem 5 - Implementação da Classe TBaseClass

A partir do Delphi 2001 foi implementada a notação JSON abordada na seção 2.5 que é uma notação padrão para comunicação leve e rápida.

Utilizando-se de JSON para comunicação entre cliente e servidor foi possível apresentar de forma mais interessante técnicas de orientação a objetos. Assim, quando o servidor recebe um objeto do cliente é necessário que o mesmo seja processado de maneira específica gerando as instruções Structured Query Language (SQL) para persistência na camada de dados.

Para resolver esses casos foi utilizado no desenvolvimento do projeto a *Runtime Type Information* (RTTI) que consiste em manter informações de um tipo de dado em memória durante o tempo de execução.

Com a *RTTI* é possível extrair de um objeto todas as suas propriedades, respectivos valores e métodos, gerando uma instrução *SQL* sem a necessidade de implementação dos comandos básicos para cada nova classe criada no projeto.

Para aproveitar o máximo das funcionalidades da *RTTI* foi criada uma *unit* denominada *uAtributos.pas* que contém as classes utilizadas em todas as demais classes do projeto para persistência dos dados. A Listagem 6 ilustra a implementação destas classes.

```
unit uAtributos;

interface

uses
  Classes;

type
  TableName = class(TCustomAttribute)
  private
    FName: String;
  public
    constructor Create(aName: String);
    property Name: String read FName write FName;
  end;

  KeyField = class(TCustomAttribute)
  private
    FName: String;
  public
    constructor Create(aName: String);
    property Name: String read FName write FName;
  end;

  FieldName = class(TCustomAttribute)
  private
    FName: String;
  public
    constructor Create(aName: String);
    property Name: String read FName write FName;
  end;
```
Listagem 6 - Implementação da unit uAtributos.pas

Na *unit uAtributos.pas* foram criadas três classes para utilizá-las com *RTTI*, são estas *TableName*, *KeyField* e *FieldName*. Essas classes foram utilizadas nas demais classes para apresentar cada propriedade da classe corresponde a um

campo do banco de dados e se ele é campo chave, bem como a classe é espelho de uma entidade do banco. A Listagem 7 apresenta uma classe nesta formação.

```
unit uPedido;

interface
uses
  uBaseClass, uAtributos;

type
  [TableName('PEDIDOS')]
  TPedido = class(TBaseClass)
  private
    FCodPedido: Integer;
    FCodMesa: Integer;
    FDataInclusao: TDateTime;
    FCodUsuario: Integer;
    FDateFechamento: TDateTime;
    FSituacao: string;
    FValorTotal: Double;
    procedure SetCodMesa(const Value: Integer);
    procedure SetCodPedido(const Value: Integer);
    procedure SetCodUsuario(const Value: Integer);
    procedure SetDataInclusao(const Value: TDateTime);
    procedure SetDateFechamento(const Value: TDateTime);
    procedure SetSituacao(const Value: string);
    procedure SetValorTotal(const Value: Double);
  public
    [KeyField('CODPEDIDO')]
    [FieldName('CODPEDIDO')]
    property CodPedido: Integer read FCodPedido write SetCodPedido;
    [FieldName('DATAINCLUSAO')]
    property DataInclusao: TDateTime read FDataInclusao write SetDataInclusao;
    [FieldName('CODMESA')]
    property CodMesa: Integer read FCodMesa write SetCodMesa;
    [FieldName('SITUACAO')]
    property Situacao : string read FSituacao write SetSituacao;
    [FieldName('CODUSUARIO')]
    property CodUsuario: Integer read FCodUsuario write SetCodUsuario;
    [FieldName('DATAFECHAMENTO')]
    property DateFechamento : TDateTime read FDateFechamento write SetDateFechamento;
    [FieldName('VALORTOTAL')]
    property ValorTotal : Double read FValorTotal write SetValorTotal;
  end;
```

Listagem 7 - Código da unit uPedido.pas

Sobre a classe *TPedido* foi informado [TableName('PEDIDOS')] que armazena o nome correspondente a tabela do banco de dados, [KeyField('CODPEDIDO')] isso indica que o campo *CODPEDIDO* é chave primária da tabela e as demais propriedades são anotadas com [FieldName('CODPEDIDO')] representando o nome físico desta propriedade junto ao banco de dados. Esse

procedimento foi realizado para todas as classes que serão persistidas no banco de dados.

Neste projeto foi criada uma classe que é responsável por gerar as instruções *SQL* e executar as mesmas junto ao banco de dados, sem que seja necessário implementar código algum para as demais tabelas do banco de dados.

Fazendo uso da *RTTI* foi desenvolvida uma classe específica para os procedimentos de persistência. Para persistir um objeto é chamado um método dessa classe passando o tipo do Objeto e a instância do mesmo. Esse método é responsável por extrair as propriedades da classe e gerar a instrução *SQL* correspondente. Na Listagem 8 está a implementação da classe *uGenericDAO.pas*.

```
unit uGenericDAO;

interface

uses
  DBXJSON, DBXJSONReflect, DBXCommon, RTTI, System.SysUtils,
  System.Classes, Datasnap.DSServer, Datasnap.DSAuth, System.TypInfo,
  uAtributos, uBaseClass, uConnection, Datasnap.DBclient, Data.SqlExpr,
  Datasnap.Provider, Forms, uFuncoes, uConstantes;

type
  TGenericDAO = class
  private
    class function GetTableName<T: class>(Obj: T): String;
  public
    class function Insert<T: class>(Obj: T) : boolean;
    class function Delete<T: class>(Obj: T) : Boolean;
    class function Update<T: class>(Obj: T) : Boolean;
    class function GetAll<T: class>(Obj: T): TDBXReader; overload;
  end;
```

Listagem 8 - Código da classe de Persistência *uGenericDAO.pas*

A classe *TGenericDAO* possui quatro *class function (Insert, Delete, Update, GetAll)* as quais podem ser chamadas sem necessidade de instanciar a classe. Esses métodos são utilizados por todas as classes que precisarem ser persistidas. Esses quatro métodos recebem como parâmetro o tipo do objeto e a instância do mesmo. Todos os objetos serão herdados de *TBaseClass*. A Implementação do método *Insert<T: class>(Obj : T)* está descrita na Listagem 9.

```
class function TGenericDAO.Insert<T>(Obj: T) : boolean;
var
  Contexto: TRttiContext;
  TypObj: TRttiType;
  Prop: TRttiProperty;
  strInsert, strFields, strValues: String;
  Atributo: TCustomAttribute;
  IsKey: Boolean;
begin
  try
    Result := True;
    strInsert := '';
    strFields := '';
    strValues := '';
    Contexto := TRttiContext.Create;
    TypObj := Contexto.GetType(TObject(Obj).ClassInfo);
    for Prop in TypObj.GetProperties do
      begin
      IsKey := False;
      for Atributo in Prop.GetAttributes do
        if Atributo is KeyField then
          IsKey := True;

      strFields := strFields + FieldName(Atributo).Name + ',';
      case Prop.GetValue(TObject(Obj)).Kind of
        tkWChar, tkLString, tkWString,
        tkString, tkChar, tkUString : strValues := strValues +
TFuncoes.StringToSql(Prop.GetValue(TObject(Obj)).AsString) + ',';
        tkInteger, tkInt64 : strValues := strValues +
TFuncoes.ZeroAsNull(Prop.GetValue(TObject(Obj)).AsInteger) + ',';
        tkFloat :
          begin
          // se for campo de Data entao faz a Conversao
          if (Prop.PropertyType.Name = 'TDateTime') then
            begin
            // se houver data informada
            if (Prop.GetValue(TObject(Obj)).AsExtended > 0) and
(Prop.GetValue(TObject(Obj)).AsExtended <> DateNull) then
              begin
              case TConexao.GetInstance.TipoBanco of
                tFirebird : strValues := strValues + TFuncoes.StringToSql(FormatDateTime('DD.MM.YYYY
HH:MM:SS', FloatToDateTime(Prop.GetValue(TObject(Obj)).AsExtended))) + ',';
                tOracle : strValues := strValues + Format('TO_TIMESTAMP(%s, ''DD/MM/YYYY
HH24:MI:SS'')', [QuotedStr(FormatDateTime('DD/MM/YYYY HH:MM:SS',
FloatToDateTime(Prop.GetValue(TObject(Obj)).AsExtended)))]) + ','
              end;
            end
            else
              strValues := strValues + 'NULL' + ',';
          end
          else
            strValues := strValues + TFuncoes.FloatToSQl(Prop.GetValue(TObject(Obj)).AsExtended) +
',';
        end;
        tkEnumeration : strValues := strValues +
TFuncoes.StringToSql(TFuncoes.SimNao(Prop.GetValue(TObject(Obj)).AsBoolean)) + ',';
      else
        strValues := strValues + 'NULL' + ',';
      end;
```

```
    end;
    strFields := Copy(strFields, 1, Length(strFields) - 1);
    strValues := Copy(strValues, 1, Length(strValues) - 1);
    strInsert := Format('INSERT INTO %s (%s) VALUES (%s) ', [GetTableName(Obj), strFields,
strValues]);
    TConexao.GetInstance.Comando.Text := strInsert;
    TConexao.GetInstance.Comando.ExecuteUpdate;
    FServidor.Logar(Format('%d Registros Inseridos na Tabela %s',
[TConexao.GetInstance.Comando.RowsAffected, GetTableName(Obj)]));
  except
    on E : Exception do
      begin
      Result := False;
      fServidor.Logar('Erro Ao Inserir Registro: ' + E.Message);
    end;
  end;
end;
```

Listagem 9 - Implementação do Método Insert da classe TGenericDAO

O método TGenericDAO.Insert<T: class>(Obj: T) utilizando a *RTTI* percorre os atributos do objeto e constrói a instrução *SQL* dinamicamente, após a instrução formada utiliza a classe de conexão para executar o comando junto ao banco de dados. Se o comando for executado com sucesso o retorno será verdadeiro caso contrário se um erro ocorrer, o mesmo será interceptado pelo bloco *Try Except* para que não se propague para a aplicação cliente devendo esse resultado ser validado na aplicação cliente.

Para que as aplicações clientes tenham acesso aos métodos implementados pelo servidor de aplicação, é necessário que o servidor disponibilize as classes e métodos acessíveis, utilizando *DataSnap* isso é feito por meio de um componente do tipo *TDSServerClass. Esse* componente foi apresentado na Figura 16.

Pode-se verificar que para cada classe implementada no servidor existe um componente do tipo *TDSServerClass* e por meio deste no evento *OnGetClass* aponta-se a classe responsável e seus métodos que ficarão visíveis para as aplicações clientes. A Listagem 10 contém a implementação do evento *OnGetClass*.

```
procedure TServerContainer.DSProdutoGetClass(DSServerClass: TDSServerClass;
  var PersistentClass: TPersistentClass);
begin
  PersistentClass := uDsProduto.TDSProduto;
end;
```

Listagem 10 - Implementação do evento *OnGetClass*

Visando uma melhor organização do sistema foram criadas classes específicas para cada objeto representado e que será trafegado entre cliente e

servidor, para isso essas classes são derivadas de *TDSServerModule* as quais já implementam as diretivas *{$MethodInfo ON}* e *{$MethodInfo OFF}* que são responsáveis por deixar as classes visíveis para as aplicações clientes.

O acesso para criar um classe do tipo *TDSServerModule*, que possui a mesma funcionalidade de um *DataModule*, é feito pelo menu *File/New/Other* e selecionar na lista a opção *DataSnap Server* e selecionar *Server Module,* uma nova *unit* será criada. A Listagem 11 apresenta a classe responsável pela tabela *produtos*.

```
unit uDsProduto;

interface

uses
  System.SysUtils, System.Classes, Datasnap.DSServer, Datasnap.DSAuth, Data.DBXJSON,
Data.DBXCommon,
  uGenericDao, uProduto;

type
  TDSProduto = class(TDSServerModule)
  private
    { Private declarations }
  public
    { Public declarations }
    function Insert(aObj : TJSONValue) : Integer;
    function Update(aObj : TJSONValue) : Boolean;
    function Delete(aObj : TJSONValue) : Boolean;
    function GetProdutoById(nCodProduto : Integer) : TJSONValue;
    function GetAllProdutos : TDBXReader;
  end;
```

Listagem 11 - Implementação da unit uDsProduto.pas

A Listagem 11 apresenta o código que contém a utilização da notação *JSON* no processo de comunicação entre clientes e servido. Uma vez que os métodos são invocados diretamente na aplicação cliente e o cliente envia ao servidor um objeto do tipo *TProduto* o qual será serializado na notação JSON conforme mostra a Listagem 12.

```
function TDSProduto.Insert(aObj: TJSONValue): Integer;
var
  oProduto : TProduto;
begin
  try
    Result := -1;
    oProduto := TProduto.JSONToObject<TProduto>(aObj);
    if oProduto.CodProduto = -1 then
      oProduto.CodProduto := TConexao.GetInstance.GetNextID('PRODUTOS', 'CODPRODUTO');

    if TGenericDAO.Insert<TProduto>(oProduto) then
      Result := oProduto.CodProduto;
  finally
    if Assigned(oProduto) then
      FreeAndNil(oProduto);
  end;
end;
```

Listagem 12 - Implementação do método Insert da classe TDSProduto

A variável *oProduto* é criada a partir do método de conversão de objeto serializado *JSON*, implementado na classe *TBaseClass*. Esse objeto é recebido por parâmetro vindo da aplicação cliente. É verificado se a propriedade código recebido no objeto é -1 que foi adotado como padrão para registros novos. Após isso é definido o próximo código válido para a propriedade *oProduto.CodProduto* e então invocado o método da classe de persistência TGenericDAO.Insert<TProduto>(oProduto). Esse método, como informado anteriormente, fará a persistência junto ao banco de dados.

Para que o usuário tenha uma interface que atenda critérios de usabilidade com as funcionalidades implementadas no sistema foi desenvolvida uma tela na qual são apresentados todos os processos executados na aplicação servidora. A Figura 17 apresenta essa tela.

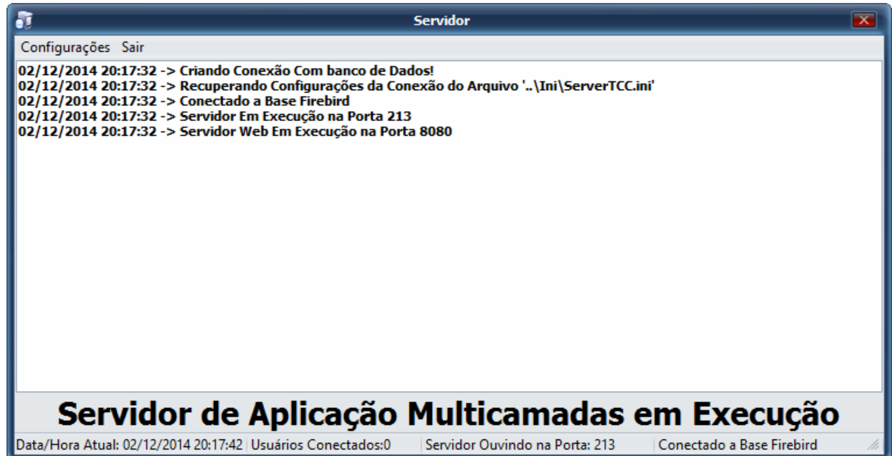

Figura 17 - Tela de processos do servidor de aplicação

5 CONCLUSÃO

O trabalho desenvolvido teve como objetivo realizar a implementação de um sistema para controle de bares, restaurantes e similares utilizando a tecnologia de desenvolvimento *DataSnap* para implementação de sistemas multicamadas.

Os objetivos buscados com este trabalho foram devidamente alcançados, pois proporcionaram o estudo e um melhor entendimento no que se refere à análise de sistemas utilizando como base a UML e a implementação em camadas utilizando a tecnologia DataSnap.

O sistema pode ser aplicado a qualquer porte de estabelecimento do ramo de bares e restaurantes, pois utilizando multicamadas facilita a escalabilidade e a manutenção do sistema.

A utilização de ferramentas para modelagem e a implementação do sistema foram relevantes pelos recursos oferecidos. Permitindo, assim, uma visão mais generalizada dos processos facilitando a implementação do protótipo e a manutenção do mesmo.

Utilizando a tecnologia *DataSnap* tem-se grande vantagem no desenvolvimento de ambientes multicamadas pela sua agilidade e facilidade. Desta forma, há uma grande economia de tempo, pois trata-se de uma implementação baseada em componentes visuais deixando a tarefa mais intuitiva e simples para o desenvolvedor.

Sendo a comunicação entre cliente e servidor baseada em *JSON* o sistema pode facilmente ser acessado por qualquer linguagem de programação que faz uso dessa notação. Assim, caso seja necessário implementar um cliente em outra plataforma ou linguagem de programação, não será preciso qualquer alteração no servidor de aplicação.

O ambiente multicamadas assegura ao sistema maior eficiência e facilidade na sua manutenção porque as regras de negócio estão implementadas no servidor de aplicação. E, assim, para realizar alguma manutenção ou melhoria no sistema é necessário somente atualizar a aplicação servidor. Os clientes deste ambiente são executáveis com pouca codificação porque são compostos de componentes visuais com validações de campos e telas.

A tecnologia *DataSnap* chegou para quebrar um paradigma na utilização e implementação de ambientes multicamadas utilizando *Delphi*, que até então era muito difícil de implementar e limitada no que se refere a transferência entre aplicações clientes e servidores.

A implementação de funcionalidades para uso de dispositivos móveis é uma dos trabalhos indicados como continuidade do sistema implementado. Com uso de um dispositivo móvel os garçons teriam mais facilidade para o registro dos pedidos, agilizando o atendimento aos clientes. Também poderia ser implementado um 'módulo' que apresentasse os pedidos para a cozinha. Em um monitor (terminal) seriam listados os pedidos que estão na fila para serem atendidos e à medida que atendidos os próprios funcionários da cozinha alteram o estado do respectivo pedido. Esse módulo listaria os pedidos pendentes e permitira alterar o estado dos mesmos. A visualização dos pedidos pendentes poderia agilizar o preparo dos alimentos uma vez que permitiria identificar mais facilmente quantidades maiores de alimentos que poderiam ser preparadas para atender diversos pedidos e atividades de preparo que podem ser realizadas simultaneamente.

REFERÊNCIAS

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **The unified software development process**. Addison Wesley, 1998.

FREEMAN, Eric; FREEMAN, Elisabeth. **Use a cabeça! Padrões de projetos**. São Paulo: Alta Books, 2005.

JSON. **Introducing JSON**. Disponível em <http://www.json.org>. Acesso em: 30 out 2014.

LANUSSE. **Teste nosso servidor DataSnap XE instalado no Amazon Cloud**. Disponível em: <http://www.andreanolanusse.com/blogen/wp-content/uploads/2010/08/datasnapoverall_2814.png>. Acesso em: 30 out. 2014.

MAZZOLA, Vitório Bruno. **Engenharia de software e sistemas de informação**. 6 ed. Brasport, 2003.

MOURAO, Rodrigo C. **DataSnap no Delphi XE**. Artigo Clube Delphi 125. Disponível em: <http://www.devmedia.com.br/datasnap-no-delphi-xe-artigo-clube-delphi-125/18850#ixzz3GifsGetl>. Acesso em: 20 out. 2014.

PRESSMAN, Roger S. **Engenharia de software**. 6 ed. McGraw-Hill Interamericana do Brasil, Rio de Janeiro, 2006.

RODRIGUES, Anderson H. **Sistemas multicamadas com delphi datasnap e dbexpress**. Florianópolis: BookStore, 2002.

SOMMERVILLE, Ian. **Engenharia de software**. São Paulo: Pearson Education do Brasil, 2003.

TANENBAUM, Andrew S. **Organização estruturada de computadores**. 5. ed. São Paulo: Prentice Hall, 2007.