Sistema de Gestão da Indústria Mineira de Jóias

Documentação Técnica
(Rascunho)

Júlio César e Melo

24 de Dezembro de 2005

Este é um documento meramente técnico, servindo de referência para desenvolvimento complementar ou manutenção do sistema. Não deve ser usado como manual do sistema, já que não há nenhuma instrução de como utilizar os programas ou componentes do sistema.

Índice

Capítulo 1

Requisitos Tecnológicos

- **.Net Framework 2.0 ou compatível**

  Descrição: Framework para a plataforma .Net.

  Autoria: Microsoft

  Licença: -

  Custo: Gratuíto

  URL: http://msdn.microsoft.com/netframework

- **MySQL 4.1**

  Descrição: Banco de dados relacional.

  Autoria: MySQL AB.

  Licença: GPL

  Custo: Gratuíto

  URL: http://www.mysql.com/

- **MySQL Connector/Net 1.0**

  Descrição: ADO.Net para banco de dados MySQL.

  Autoria: MySQL AB.

  Licença: GPL

  Custo: Gratuíto

  URL: http://www.mysql.com/products/connector/net/

- **Validating Edit Controls***

  Descrição: Controles para aplicações em .Net com finalidade de formatar TextBox's.

  Autoria: Alvaro Mendez

  Licença: desconhecida

  Custo: Gratuíto

  URL: http://www.codeproject.com/cs/miscctrl/ValidatingTextBoxControls.asp

* Disponível no código.

- **Ballon Windows for .NET***

  Descrição: Controles para aplicações em .Net para criação de janelas em formatos de balões.

  Autoria: Rama Krishna Vavilala

  Licença: desconhecida

  Custo: Gratuíto

  URL: http://www.codeproject.com/cs/miscctrl/balloonnet.asp

* Disponível no código.

Capítulo 2

Organização Do Sistema

Foi adotada a técnica de divisão em três camadas: acesso, negócio e apresentação. A primeira é responsável pelo armazenamento físico dos dados. A segunda trata as regras de negócio da empresa, realizando todos os cálculos e operações relacionados ao funcionamento da empresa. Já a terceira camada está vinculada à interface de usuário (UI – *User Interface*).

Padrões adotados

Para a compreensão do sistema, deve-se antes conhecer alguns padrões adotados nos códigos e no modelo de classes.

Padrão Singleton

Classes *Singleton* são classes de instância única, ou seja, só pode existir um único objeto de classes *singleton* ou nenhuma durante a execução do programa.

Padrão Observador (Observer Pattern)

Contém a implementação adaptada do padrão *Observer*, permitindo que objetos *sujeitos* disparem eventos para observadores de forma assíncrona, protegida contra falhas e com independência entre bibliotecas. Estas últimas características não correspondem ao padrão, mas sim à adaptação.

A implementação está contida no componente *Observador.dll*, possuindo classes no namespace *Observador*. São elas: (vide também diagrama de classes na página 43)

Encaminhador	Classe que encaminha mensagens para aplicações da camada de apresentação, utilizando uma *thread* particular. *(Observador\Encaminhador.cs)*
EventoObservação	Assinatura da função que deverá ser usada para tratar os eventos de observação. *(Observador\ISujeito.cs)*

ExceçãoDelegate	Assinatura da função que será chamada para tratar exceções ocorridas durante a observação. *(Observador\ExceçãoDelegate.cs)*
ExceçãoObservadorDuplic...	Exceção levantada quando um observador é inserido duas vezes em um mesmo encaminhador. *(Observador\ExceçãoObservadorDuplicado.cs)*
IPopulação	Interface para população de sujeitos observáveis. *(Observador\IPopulação.cs)*
ISujeito	Interface para sujeito observável. *(Observador\ISujeito.cs)*
Mensagem	Estrutura interna contendo a mensagem a ser inserida na fila de mensagens para observadores. *(Observador\Mensagem.cs)*
MensagensObservador	Classe interna que implementa uma tupla contendo mensagens de um observador. *(Observador\MensagensObservador.cs)*
PopulaçãoMarshalByRef	Classe abstrata que implementa uma população de sujeitos observáveis. Esta classe herda a classe *System.MarshalByRef*. *(Observador\PopulaçãoMarshalByRef.cs)*
ReceptorMensagens	Classe para receber mensagens de forma protegida, tornando desnecessárias referências entre projetos do objeto sujeito observado e do objeto observador. Com esta classe, obtém-se independência entre bibliotecas DLLs e executáveis EXEs. Esta classe herda de *System.MarshalByRef*.
ReceptorMensagensSujeito	Classe que herda de *ReceptorMensagens*. Esta classe facilita a observação de um único sujeito. Caso mais de um sujeito seja observado por uma mesma função, é melhor utilizar a classe *ReceptorMensagens*.
SujeitoProtegidoMarshal...	Classe abstrata que implementa um sujeito observável, encaminhando mensagens por uma *Thread* estática, garantindo ordem de eventos. Esta classe herda de *System.MarshalByRef.* *(Observador\SujeitoProtegidoMarshalByRef.cs)*

Capítulo 3

Camada De Acesso

A camada de acesso é a única camada do sistema a conter implementação de códigos vinculados ao armazenamento e recuperação de dados relevantes da empresa.

Namespaces

Acesso	Contém classes para manipulação de banco de dados.
Entidades	Contém classes que representam o modelo entidade-relacionamento.
Entidades.Administrativo	Contém classes que representam o modelo entidade-relacionamento de entidades meramente administrativas.
Entidades.Pessoa	Contém classes que representam o modelo entidade-relacionamento de entidades relacionadas a pessoas.

Classes e tipos do namespace Acesso

Acesso	Classe abstrata, responsável pela manipulação de todos os dados relevantes da empresa. *(Acesso\Acesso.cs)*
AcessoMySQL	Classe herdada de *Acesso*, com implementação de manipulação de dados para banco de dados MySQL. *(Acesso – MySQL\AcessoMySQL.cs)*
DadosPessoa	Enumeração de atributos da entidade *Pessoa* que podem ser usados para consulta. *(Acesso\Acesso.cs)*
DadosPessoaFísica	Enumeração de atributos da entidade *PessoaFísica* que podem ser usados para consulta. *(Acesso\Acesso.cs)*

*Classes e tipos do namespace Entidades.**

Administrativo.Reclamação	Classe para armazenamento de reclamações de clientes ou funcionários a respeito de funcionários ou setores. *(Entidades\Administrativo\Reclamação.cs)*

IEntidade	Interface usada para identificar entidades. *(Entidades\IEntidade.cs)*
MotivoContato	Enumeração para identificar o motivo com que uma pessoa entrou em contato físico (visitante) ou por telefone com a empresa. *(Entidades\MotivoContato.cs)* (Melhor posicionada na camada de negócio)
Pessoa.Funcionário	Classe que herda de *Pessoa.Pessoa* e implementa *IEntidade* para armazenamento de dados de um funcionário. *(Entidades\Pessoa\Funcionário.cs)*
Pessoa.Pessoa	Classe genérica que implementa *IEntidade* para armazenamento de dados a respeito de uma pessoa física ou jurídica. Esta classe contém somente atributos que estão comumente presentes em pessoa física e jurídica. *(Entidades\Pessoa\Pessoa.cs)*
Pessoa.PessoaFísica	Classe que herda de *Pessoa.Pessoa* e implementa *IEntidade* para armazenamento de dados de uma pessoa física. *(Entidades\Pessoa\PessoaFísica.cs)*
Pessoa.PessoaJurídica	Classe que herda de *Pessoa.Pessoa* e implementa *IEntidade* para armazenamento de dados de uma pessoa jurídica. *(Entidades\Pessoa\PessoaJurídica.cs)*
Setor	Classe que implementa *IEntidade* para armazenamento de dados relativos a um setor. *(Entidades\Setor.cs)*
Telefonema	Classe que implementa *IEntidade* para armazenamento de dados relativos a telefonemas realizados a partir da empresa ou recebidos à cobrar. *(Entidades\Telefonema.cs)*
Visita	Classe que implementa *IEntidade* para armazenamento de dados relacionados a uma visita de um cliente ou de algum visitante que adentrou na empresa por alguma razão. *(Entidades\Visita.cs)*
VisitaAtendimento	Classe que implementa *IEntidade* relativa ao mapeamento do relacionamento Visita<->Atendimento. *(Entidades\VisitaAtendimento.cs)* (Melhor se reimplementada)

Capítulo 4

Camada De Negócio

Toda regra de negócio é de responsabilidade da camada de negócio. Ela realiza operações, cálculos e manutenção dos objetos e dados relevantes à empresa.

Como a camada de negócio intermedia a camada de acesso e a camada de apresentação e, principalmente, por ela servir de base para o funcionamento da camada de apresentação, ela deve ser persistente, ou seja, os valores devem se manter mesmo que nenhum programa da camada de apresentação esteja funcionando. Portanto, esta camada precisa ser mantida por um serviço responsável por armazenar os dados em um meio persistente (camada de acesso) e manter as instâncias de objetos "vivas" enquanto necessárias. Estes objetos são considerados objetos relacionados a um "contexto" e estão todos organizados no namespace *Negócio.Contexto* (vide diagrama de classes na página 45).

O intermédio da camada de apresentação e a camada de negócio é realizada sempre por interfaces, de forma que a camada de apresentação desconheça como as rotinas da camada de negócio são executadas. Para tanto, as classes da camada de negócio implementam interfaces do projeto "Interface – Negócio", que é compartilhado com todas as camadas. Assim, garante-se que a interoperabilidade da camada de apresentação com a camada de negócio não seja prejudicada caso aconteça manutenções nas mesmas.

Todo objeto da camada de negócio que pode ser acessada externamente a ela ou pelos objetos de controles possui uma interface pública. Esta interface está contida em um componente (DLL) aparte, que é posteriormente importada pelos componentes da camada de apresentação. Desta maneira, uma suposta alteração na camada de negócio pode não afetar em nada os componentes compilados da camada de apresentação. Apenas alterações na interface acarretarão em recompilação dos programas para usuários.

A implementação das classes são distribuídas em outras bibliotecas, agrupadas pelo ambiente de funcionamento ou funcionalidade, conforme *namespace*. Entretanto, *namespaces* não muito complexas são incorporadas em outras bibliotecas cuja finalidade seja semelhante.

Os componentes da camada de negócio são extremamente críticos, necessitando ser rigorosamente estáveis e recuperáveis, mesmo em caso de falhas, já que a sua função é análoga ao coração do sistema de informação.

Namespaces

Negócio	Contém todas as classes destinadas às regras de negócio e aos elementos/pessoas que interagem com o sistema.
Negócio.Comparadores	Contém classes para comparação dos objetos da camada de negócio.
Negócio.Contexto	Contém classes que manipulam os objetos do contexto envolvido no momento, como visitantes na empresa, funcionários trabalhando, rodízio de setores, etc.
Negócio.Controle	Contém classes de controle para intermédio da camada de apresentação.
Negócio.Estoque*	Contém classes para controle de estoque e mercadoria, ademais de impostos vinculados a eles (estoque, grupo e mercadoria).
Negócio.Observador	Contém enumerações de ações possíveis de *sujeitos* (vide namespace *Observador*) da camada de negócio.

Classes e tipos do namespace Negócio

EstadoFuncionário	Enumeração dos possíveis estados de um funcionário. *(Interface – Negócio\EstadoFuncionário.cs)*
Funcionário	Classe que implementa *IFuncionário* e herda de *Observador.SujeitoProtegidoMarshalByRef* contendo a representação computacional de um funcionário da empresa. *(Negócio\Funcionário.cs)*
IFuncionário	Interface para a representação computacional de um funcionário da empresa. *(Interface – Negócio\IFuncionário.cs)*
ISetor	Interface para a representação computacional de um setor da empresa. *(Interface – Negócio\ISetor.cs)*
ISetorRodízio	Interface que permite acesso ao rodízio de um setor. *(Interface – Negócio.Contexto\ISetorRodízio.cs)* (Nome de diretório ruim e a interface deveria ser mesclada com *ISetor*)

* Contém biblioteca reutilizável para outros projetos, cuja implementação é genérica.

IVisitante	Interface para a representação computacional de um visitante que está ou esteve na empresa. *(Interface – Negócio\IVisitante.cs)*
Mapeamento	Classe abstrata contendo função estática para copiar todos os atributos de um objeto origem para um destino, não necessariamente de mesmo tipo. Bastante útil para copiar atributos de uma classe para outra em um nível hierárquico abaixo. *(Interface - Negócio\Mapeamento.cs)*
Setor	Classe que implementa *ISetor* e *ISetorRodízio*, representando computacionalmente um setor da empresa.
Visitante	Classe que implementa *IVisitante* e herda de *Observador.SujeitoProtegidoMarshalByRef*, representando computacionalmente um visitante que está ou esteve na empresa. *(Negócio\Visitante.cs)*

Classes e tipos do namespace Negócio.Comparadores

FuncionárioOrdemRodízio	Compara dois funcionários conforme sua prioridade no rodízio. *(Interface – Negócio\Comparadores\FuncionárioOrdemRodízio.cs)*

Classes e tipos do namespace Negócio.Contexto

AcessoLocal	Classe permitindo acesso à camada de acesso por parte dos componentes da camada de negócio. Não deve ser usada para controles. *(Negócio\Contexto\AcessoLocal.cs)*
Funcionários	Classe que implementa *IFuncionários,* sendo uma coleção de funcionários da empresa. *(Negócio\Contexto\Funcionários.cs)*
IFuncionários	Interface para a coleção de funcionários da empresa. *(Interface – Negócio.Contexto\Contexto\IFuncionários.cs)*
IRodízio	Interface para o controle de rodízio de um setor. *(Interface – Negócio.Contexto\Contexto\IRodízio.cs)*
ISetores	Interface para a coleção de setores da empresa. *(Interface – Negócio.Contexto\Contexto\ISetores.cs)*
IVisitantes	Interface para a coleção de visitantes que estão atualmente na empresa ou que estiveram no presente dia. *(Interface – Negócio.Contexto\Contexto\IVisitantes.cs)*
Rodízio	Classe que implementa *IRodízio*, permitindo o controle de rodízio de um determinado setor. *(Negócio\Contexto\Rodízio.cs)*

Setores	Classe que implementa *ISetores*, sendo uma coleção de setores da empresa. *(Negócio\Contexto\Setores.cs)*
Visitantes	Classe que implementa *IVisitantes* e herda de *Observador.PopulaçãoMarshalByRef*, sendo uma coleção de visitantes que estiveram no dia presente na empresa. *(Negócio\Contexto\Visitantes.cs)*

Classes e tipos do namespace Negócio.Controle

Nem todas as classes e tipos estão mostradas aqui. Somente aquelas principais que servem de base para as demais classes de controle estão explicitadas aqui.

Contextos	Classe que implementa *IContextos,* agrupando todos os objetos de coleção do contexto da camada de negócio. *(Negócio\Controle\Contextos.cs)*
Comum	Classe abstrata com implementação de métodos e atributos comuns dos controles. *(Interface – Negócio.Contexto\Controle\Comum.cs)*
IContextos	Interface que agrupa todos os objetos de coleção do contexto da camada de negócio.
IControle	Interface para controles. *(Interface – Negócio\Controle\IControle.cs)*
IControleCriptografado	Interface para controles com transmissão de parâmetros criptografados. *(Interface – Negócio\Controle\IControleCriptografado.cs)*
InterfaceControleCriptografado	Classe abstrata contendo métodos comuns de comunicação criptografada. *(Interface – Negócio\Controle\InterfaceControleCriptografado.cs)*

Classes e tipos do namespace Negócio.Estoque

Este namespace contém biblioteca reutilizável (*Negócio.Estoque.dll*), cuja implementação é genérica (vide diagrama de classes nas páginas e).

Estoque	Classe de representação do(s) estoque(s) da empresa. Implementa *ISujeitoImposto*. *(Negócio.Estoque.dll)*
Grupo	Classe que agrupa referências de mercadorias, para fins de classificação. Implementa *ISujeitoImposto*. *(Negócio.Estoque.dll)*
GrupoHierárquico	Classe que herda de grupo, permitindo um agrupamento hierárquico. *(Negócio.Estoque.dll)*
IEstocador	Interface para empresas que possuem estoques. Deve ser implementado para uso na classe *Estoque*. *(Negócio.Estoque.dll)*

Imposto	Classe que representa um imposto a ser cobrado sobre um *ISujeitoImposto*. *(Negócio.Estoque.dll)*
ISujeitoImposto	Interface para objetos sujeitos a impostos, como *Estoque*, *Grupo* ou *Mercadoria*.
Mercadoria	Classe abstrata que representa uma mercadoria da empresa. Implementa *ISujeitoImposto*. *(Negócio.Estoque.dll)*
Referência	Classe abstrata que representa uma referência/código de mercadoria. *(Negócio.Estoque.dll)*
TabelaReferências	Classe *singleton* para armazenar todas as referências adotadas nos estoques da empresa.

Classes e tipos do namespace Negócio.Observador

AçãoFuncionário	Possíveis ações que podem ser observadas de um objeto do tipo *Negócio.Funcionário*.
AçãoVisitante	Possíveis ações que podem ser observadas de um objeto do tipo *Negócio.Visitante*.

Capítulo 5

Estrutura De Um Programa Da Camada De Apresentação

Existe todo um conjunto de classes que podem ser usadas para desenvolver um sistema para a camada de apresentação, seguindo um único padrão. Até mesmo o início do programa, que no C# normalmente se dá pelo comando "Application.Run(new Form1())" é substituído por uma função estática que mostra a *Splash Screen* (tela de entrada), efetua o login do usuário, conecta o programa na fachada (camada de negócio) e submete erros para o sistema de bugs.

Para tanto, deve-se introduzir alguns conceitos acerca do framework dotNet. Toda aplicação necessita de um contexto para seu funcionamento. Na chamada *Application.Run*, cria-se um contexto padrão para um formulário específico. Neste contexto, será executado o *loop* local do programa, que trata eventos como click do mouse, entrada de teclado ou requisições/interrupções do sistema. Na estrutura dos programas da camada de apresentação, o contexto padrão do dotNet é expandido pela classe ***AplicaçãoIntegrada***, no namespace ***Integração.Apresentação***.

Esta classe é responsável por identificar erros não tratados durante a execução do programa e preparar o formulário principal para ser utilizado, a partir do método estático ***Executar***. Assim, uma função principal típica de um programa seguindo esta estrutura seria:

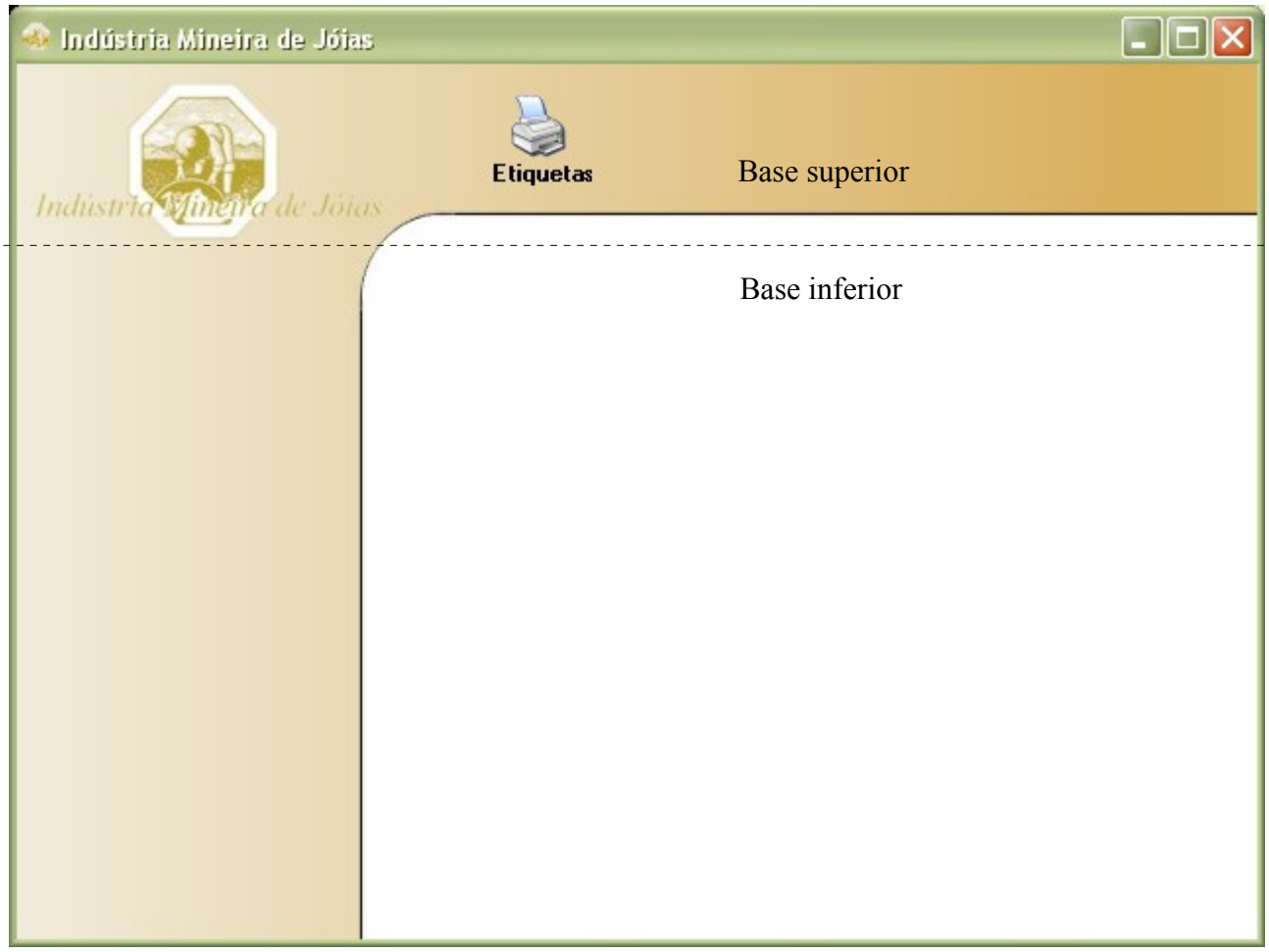
```
private static void Main(string [] args)
{
     Integração.Apresentação.AplicaçãoIntegrada.Executar(
           typeof(Principal),      // Tipo do formulário principal
           "Cofre",                // Nome da fachada
           8111,                   // Porta padrão da fachada
           true,                   // Efetuar login
           true);                  // Mostrar splash (tela de entrada)
}
```

O primeiro parâmetro especificado é uma janela que deve herdar de ***BaseFormulário***. Isso é necessário, pois esta classe possui suporte a fachadas. Durante os primeiros passos do método *Executar*, a propriedade *Fachada* da instância de *BaseFormulário* é atribuída ao objeto remoto da fachada correspondente, especificada no segundo e terceiro parâmetro. No exemplo acima, *Principal* é o nome de uma classe que herda de *BaseFormulário*.

Janela Principal

A janela principal é composta por duas partes: base superior e base inferior. A base inferior contém botões genéricos para acessar a base inferior. O sistema da recepção, por exemplo, contém os botões Agenda, Entrada e Saída – para registro de entrada e saída de clientes ou visitantes –, Telefonemas, Rodízio e Agendamentos.

Os botões que compõem o topo são objetos da classe ***Botão*** do namespace ***Apresentação.Formulários*** e estão inseridos dentro do componente ***BarraBotões***. O botão está associado a uma controlador de base inferior que, quando clicado, automaticamente substitui a base inferior do formulário em que está inserido pela base inferior associada a ele. Para tanto, deve-se inserir a base inferior na construtora do formulário. O trecho de código a seguir mostra como associar o botão *botãoEtiquetas* à base inferior *Imprimir*:



```
botãoEtiquetas.AdicionarBaseInferior(new Imprimir());
```

Toda vez que um botão for clicado, será pedido ao seu controlador que exiba a base inferior (vide detalhes do controlador mais à frente).

Base inferior

A base inferior, por sua vez, é um controle que herda de ***BaseInferior*** do namespace ***Apresentação.Formulários***. Ela também é divida em duas regiões: painel da esquerda e painel da direita. O painel da esquerda contém opções específicas vinculadas ao trabalho que o usuário está exercendo. Logo, deve mostar opções específicas da base inferior e, inclusive, das seleções realizadas pelo usuário dentro da mesma, de forma dinâmica.

Sempre que a base inferior precisar de atribuir dados no momento em que é carregado, seja a seus componentes ou às suas variáveis internas, que dependem que o usuário tenha se identificado ou efetuado conexão com o banco de dados, deve-se sobrescrever o método ***AoCarregarCompletamente***. Este é o único método que garante que existe uma conexão com o banco de dados e que o usuário já encontra-se identificado. Quando disparado, este método verifica se existe algum controle isnerido na base que implemente a interface ***IPósCargaSistema***. Caso encontre, o método ***AoCarregarCompletamente*** é executado também nestes controles.

Logo que a base inferior é exibida pela primeira vez, é disparado o método virtual ***AoExibir***, sendo que da primeira vez que é exibida, é chamado antes o método ***AoExibirDaPrimeiraVez***. Tais métodos podem ser sobrescritos para prepararem a exibição da base inferior.

A base inferior pode ainda trabalhar em conjunto com outras bases, requisitando a substituição da própria base por outra, por meio do método ***SubstituirBase***. Assim que chamado, a

base inferior requerará a substituição da base, que só ocrrerá caso esta base esteja em exibição. A regra de subsituição e validação de base consta no controlador de base inferior (vide a seguir).

Controlador de base inferior

O controlador da base inferior é responsável pela exibição de base inferiores correlacionadas no formulário. Sempre que um botão associado a ele é clicado, o controlador substitui a base inferior do formulário base. É possível implementar uma regra para subsituição da base, introduzindo um controlador personalizado que especialize a classe ***ControladorBaseInferior*** e atribuindo ao botão na construtora do formulário. Todo botão possui um controlador associado e, se nenhum for definido pelo implementador, o controlador padrão utilizado é uma instância do ***ControladorBaseInferior***.

O intuíto do controlador é deixar o código da base inferior mais enxuto, voltado somente à interface gráfica, permitindo ao usuário, assim, a partir de um mesmo botão, acessar diferentes bases inferiores.

As bases inferiores podem ser correlacionadas de duas maneiras: i) por meio do método ***SubstituirBase***, da base inferior; ii) por meio do método ***Controlador.InserirBaseInferior***.

Quando se utiliza o método ***SubstituirBase*** de uma base inferior, não estando ela inserida no controlador pelo método ***InserirBaseInferior***, a base a ser exibida será considerada temporária, tendo seus recursos liberados logo que também, substituída. Quando a base inferior for inserida no controlador, ela será considerada permanente e nunca terá seus recursos liberados por ele.

Em um mesmo controlador, pode-se ter várias bases inferiores inseridas.

Componentes Da Base Inferior

Existem diversos controles de formulários implementados para facilitar o desenvolvimento da base inferior.

Título

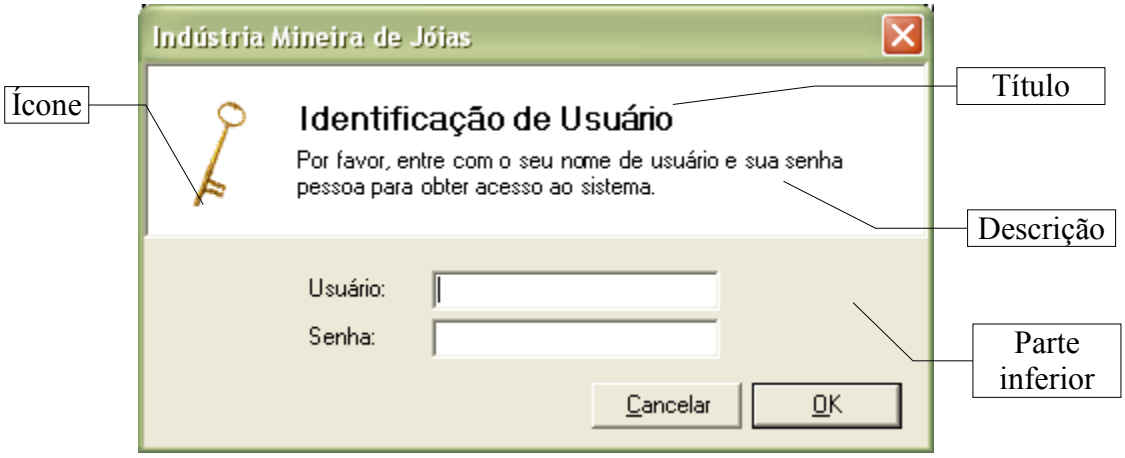
O título de uma base inferior pode ser introduzido pelo controle ***TítuloBaseInferior***. Nele podem ser atribuídos uma imagem ilustrativa, um título e uma descrição ou texto auxiliar. Exemplo:

Janelas De Apoio

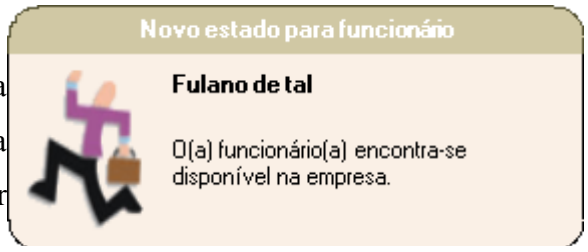
Existem algumas janelas de apoio que facilitamo desenvolvimento padrozinado do sistema.

Janela Explicativa

A janela explicativa é composta por um título, uma descrição e um ícone na parte superior, reservando à parte inferior o conteúdo da janela. Exemplo:



Janela de Notificação

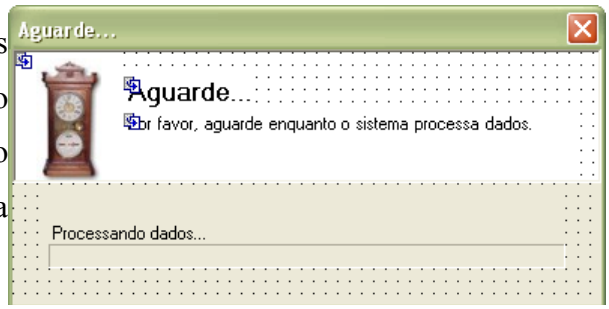


A janela de notificação é utilizada para notificar o usuário de algum evento ocorrido. A janela pode ser configurada para que exiba só por determinados segundos.

Para garantir a notificação sem interrupção do trabalho do usuário, utilize o método estático Mostrar da classe Notificação. Neste método, a janela de notificação é construída em segundo plano e exibida em *thread* separada, sem mudar a janela ativa.

Janela Aguarde

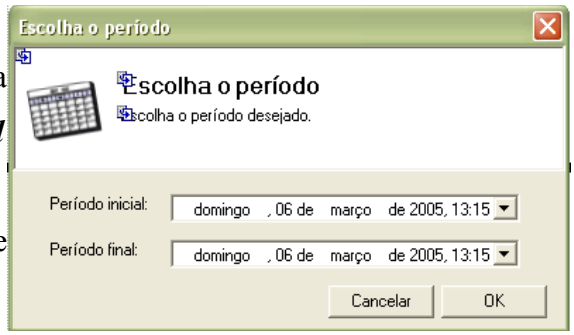
A janela ***Aguarde*** tem como função informar ao usuário que um processamento está sendo realizado e é necessário que ele aguarde o término do mesmo.



Para utilizá-la, é necessário informar quantos passos serão realizados. A cada passo do processamento, o desenvolvedor deve chamar o método ***Passo***, passando como parâmetro opcional a descrição do passo. Exemplo:

```
using (Aguarde aguarde = new
Aguarde("Preparando...", 2))
{
        aguarde.Show();
        // (...)
        aguarde.Passo("Realizando primeiro passo...");
        // (...)
        aguarde.Passo("Realizando passo final...");
        // (...)
}
```

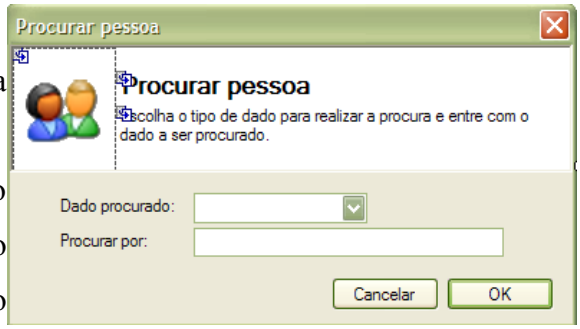
Seleção de período



Para a seleção simples de período, existe a janela ***SeleçãoPeríodo*** com duas propriedades: ***PeríodoInicial*** e ***PeríodoFinal***.

Esta janela pode ser utilizada como janela de diálogo, poupando trabalho do desenvolvedor.

Procurar pessoa cadastrada



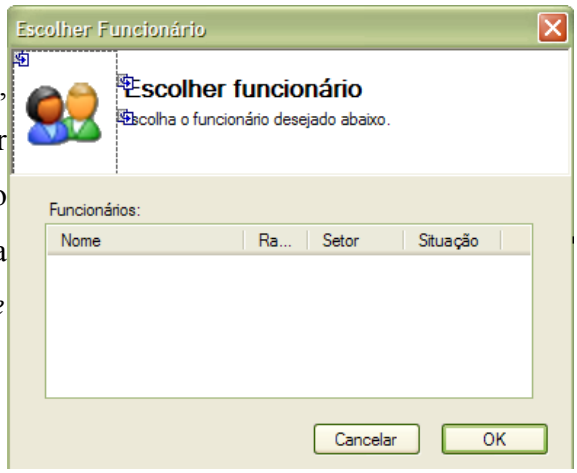
Para consulta de pessoa cadastrada, a janela ***Apresentação.Pessoa.Consultas.ProcurarPessoa*** permite a busca por diferentes tipos de dados, como nome, documento, cidade, etc. Nesta consulta, parte do sobrenome pode ser omitido. Exemplo: Júlio Melo ao invés de Júlio César e Melo.

Digitação de nome de pessoa cadastrada

A digitação de nome de pessoa cadastrada pode ser facilitada utilizando o TextBox ***Apresentação.Pessoa.Consulta.TextBoxPessoa*** que exibe lista de pessoas cadastradas conforme prefixo digitado.

Escolha de funcionário

Se necessário realizar escolha de funcionários, com possibilidade de enfatizar um setor, pode-se optar por utilizar a janela ***EscolherFuncionário*** do *namespace* ***Apresentação.*****Pessoa**, *ou* utilizar a ListView ***ListViewFuncionários*** do *namespace* ***Apresentação.Pessoa.Consultas.***.



Capítulo 6

Estrutura De Uma Fachada

As fachadas dão acesso ao funcionamento do servidor de camada de negócio do sistema e ao banco de dados. Desta maneira, o sistema do servidor está protegido, tendo suas funções acessadas somente pelas fachadas.

Toda fachada implementa a classe abstrata ***FachadaBásica*** do namespace ***Negócio.Fachada***, presente no projeto ***Negócio – Controle.Compartilhado***[1]. Por sua vez, a *FachadaBásica* herda de ***AcessoUsuário***, que provê a propriedade ***UsuárioRemoto*** para acesso ao usuário conectado que está chamando algum método da fachada. *AcessoUsuário* é uma classe *ContextBoundObject*, que sempre tem seu código executado no servidor quando chamado remotamente e pode acessar objetos em contextos diferentes. A referência do objecto de conexão do usuário, por exemplo, é armazenada no contexto do cliente, e não do servidor. Assim, para ter acesso à mesma, é necessário que o objecto de fachada tenha acesso a contextos externos, característica esta permitida a objetos *ContextBoundObject*.

A *FachadaBásica* permite o *log* de erros e o acesso pela fachada por meio de propriedades protegidas aos objetos de contexto, além do controle de usuário, abrindo um caminho para que o usuário possa se identificar no sistema, criando na fachada uma conexão com o banco de dados.

A única exigência para uma classe que herda de *FachadaBásica* é que ela implemente a o método ***Preparar()***. Este método é executado toda vez que algum usuário obtém o objeto remoto, utilizando o método ***Conectar*** da classe ***Fachada*** do namespace ***Integração.Apresentação***. Este método é automaticamente chamado quando executado o método ***Executar*** da classe ***AplicaçãoIntegrada***, como visto no capítulo anterior.

Normalmente uma fachada é executada em um serviço a parte, facilitando a sua manutenção sem prejudicar os demais componentes do sistema. O desenvolvimento do sistema, entretanto, é dificultado ao se trabalhar com serviços, já que estes não podem ser executados pelo ambiente de desenvolvimento como *SharpDevelop* ou *Microsoft Visual Studio*. Para facilitar tal desenvolvimento, foi desenvolvido um método estático da classe ***FachadaServiçoBásico*** também

1 O nome do projeto é inadequado e o *namespace* padrão para o projeto é “Negócio”.

do namespace *Fachada.Negócio* que, quando compilado com a configuração **DEBUG**, cria o objeto remoto da fachada e segura a aplicação enquanto o usuário não pressiona <Enter> no console, e que se compilado com a configuração **RELEASE**, executa o serviço criado por ele mesmo.

Assim, a criação de uma fachada tradicional, sem implementações específicas, contém somente as seguintes linhas, que podem ser usadas como molde para qualquer fachada:

```
/// <summary>
/// Fachada de exemplo
/// </summary>
class Fachada : Negócio.Fachada.FachadaBásica
{
      /// <summary>
      /// Prepara a fachada
      /// </summary>
      public override void Preparar()
      {
            // Nada aqui
      }


      /// <summary>
      /// Início do sistema
      /// </summary>
      static void Main(string[] args)
      {
            Negócio.Fachada.FachadaServiçoBásico.Executar(
                  typeof(Fachada),
                  "Exemplo",
                  5555);
      }
}
```

Esta fachada está ***inserida em um pro***jeto de aplicação console. Para a instalação do serviço, não basta implementá-lo. Deve-se ter uma classe que herde de ***Installer*** do namespace ***System.Configuration.Install***, com o atributo ***RunInstaller*** marcado como verdadeiro. Para facilitar o processo de criação, basta implementar uma classe no mesmo projeto conforme molde a seguir:

```
/// <summary>
/// Instalação do serviço
/// </summary>
[System.ComponentModel.RunInstaller(true)]
class Instalação : Negócio.Fachada.FachadaInstalação
{
      public Instalação() : base("Exemplo")
      { /* Vazio */ }
}
```

A classe abstrata ***FachadaInstalação*** requer somente o nome da fachada passada na construtora.

Capítulo 7

Código De Barras

Existem diversos padrões para o formato do código de barras. O padrão escolhido é o Interleaved25 2:1, devido ao código reduzido gerado. Com ele é possível codificar até 8 dígitos, no tamanho atual das etiquetas da Indústria Mineira de Jóias, sem prejudicar a leitura óptica do código de barras. Quanto mais dígitos, maior é o código de barras. Mantendo-se o tamanho fixado proporcional ao tamanho da etiqueta, quanto mais dígitos, menor é o tamanho da barra, dificultando a leitura óptica. À medida que o tamanho dela diminui, menor é a distância tolerada do leitor de código de barras em relação à etiqueta.

Uma importante característica do Interleaved25 é a utilização de códigos de tamanho par com pelo menos 6 dígitos. A codificação ocorre intercalando pares de dígitos, em que cada par é impresso com um dígito utilizando barras escuras (pretas) e o outro com barras claras (brancas ou simplesmente espaçamento).

Tal característica permite uma maior intervalo de codificação numérico, visto que o código 000000 é impresso diferente de 00000000. O segundo código possui dois dígitos a mais que o primeiro, mesmo algebricamente representando o mesmo número. Como o código de barras é diferente, pode-se considerar então que eles são números diferentes. Para isso, observa-se o número de dígitos presentes no código de barras. Para seis dígitos, considera-se o intervalo [0;999.999]. Para oito dígitos, soma-se 100.000, obtendo o intervalo [100.000;100.099.999]. Assim, o código de barras 000000 representaria zero e o código de barras 00000000 representaria 100.000. Este tipo de codificação permitiu, neste exemplo, o aproveitamento de 99.999 códigos a mais. Repetindo-se este raciocínio para mais pares de dígitos, o aproveitamento será ainda maior.

Para as mercadorias da IMJ, entretanto, será necessário identificar a sua referência (11 dígitos + 1 dígito verificador) e o seu peso, que pode variar em uma mesma referência. As etiquetas das mercadorias são impressas com três dígitos, sendo um para uma casa decimal. Ao todo, desconsiderando o digito verificador, são 14 dígitos que devem ser codificados, entretanto, para se ter uma leitura adequada, o máximo conseguido foram 8 dígitos na etiqueta.

Como os dígitos na referência representam informação acerca das características da mercadoria para compreensão humana, é possível representa-los utilizando um código seqüencial com menos dígitos. No início de 2004, constavam no banco de dados 1.844 referências diferentes. Este número pode ainda aumentar, devido ao desejo da empresa em acabar com as referências genéricas[2], transformando-as em referências que identificam uma mercadoria conforme sua "cara", ou aparência.

As mercadorias cuja etiqueta é impressa com peso são a minoria. No início de 2004, constam no banco de dados 186 referências cujo peso varia em uma mesma referência e 1.658 referências cujo peso médio é obtido do banco de dados, não necessitando a impressão do peso.
O peso não será codificado, já que pode variar sequencialmente no intervalo [0,0;40,0] gramas, aproximadamente. No banco de dados, o peso máximo cadastrado é 25,2g. Por segurança, será considerado um intervalo de [0,0;99,9].

Mercadorias com peso

As etiquetas impressas na IMJ utilizam o formato "9PP,P9", onde "9" é um dígito fixado antes e depois do peso "PP,P", apenas para não tornar óbvio o peso para o cliente. Desta forma, três dígitos bastam para identificar o peso no código de barras, conforme formato a seguir:

Pares	*Formato do código de barras*	*Intervalo para pesos*	*Intervalo para código de mapeamento*	*Quantidade de referências*
3	PP|PR|RR	[0,0;99,9]	[0;999]	1.000
4	PP|PR|RR|RR	[0,0;89,9]	[1.000;100.999]	100.000
4	9P|PP|PR|RR	[90,0;1089,9]	[0;999]	

Total: 101.000

Não é viável a utilização de 2 pares com peso, visto que o formato "PP|PR" possibilita apenas 10 referências diferentes. Para dois pares de dígitos, serão considerados todos dígitos como referência, sem peso, conforme tópico a seguir.

Afim de possibilitar a inserção de pesos raros, acima de 100g, há um caso específico em que o código de barras com 4 pares de dígitos prefixado com um 9, os próximos quatro dígitos representam o peso acrescido de 90g. Assim, este formato abrange o intervalo de 90g a 1089,9g, mapeando 1.000 referências distintas para mercadorias nesta faixa de peso.

2 Existem referências que não identificam unicamente uma mercadoria, mas um grupo de mercadoria cujos componentes de custos são os mesmos.

Ao todo, pode-se garantir 102.000 referências diferentes para mercadorias a serem impressas com o peso. Se a chave primária, entretanto, da tabela de mapeamento de referência e peso para código de barras for composta pelos atributos “peso” e “código de mapeamento”, será possível registrar 102.000 referências para cada peso. Certamente não serão utilizadas todas as combinações, visto que as mercadorias possuem pesos semelhantes. No banco de dados citado anteriormente, das 1.844 referências cadastradas, constam somente 135 pesos diferentes, considerando apenas uma casa decimal.

Mercadorias sem peso

As mercadorias a serem impressas sem peso são mais simples. Todos os dígitos se referem ao código de mapeamento da referência, conforme formato na tabela a seguir:

Pares	*Formato do código de barras*	*Intervalo para pesos*	*Intervalo para mapeamento*	*Quantidade de referências*
3	00|0R|RR	-	[0;999]	1.000
4	00|0R|RR|RR	-	[1.000;100.999]	100.000

Total: 101.000

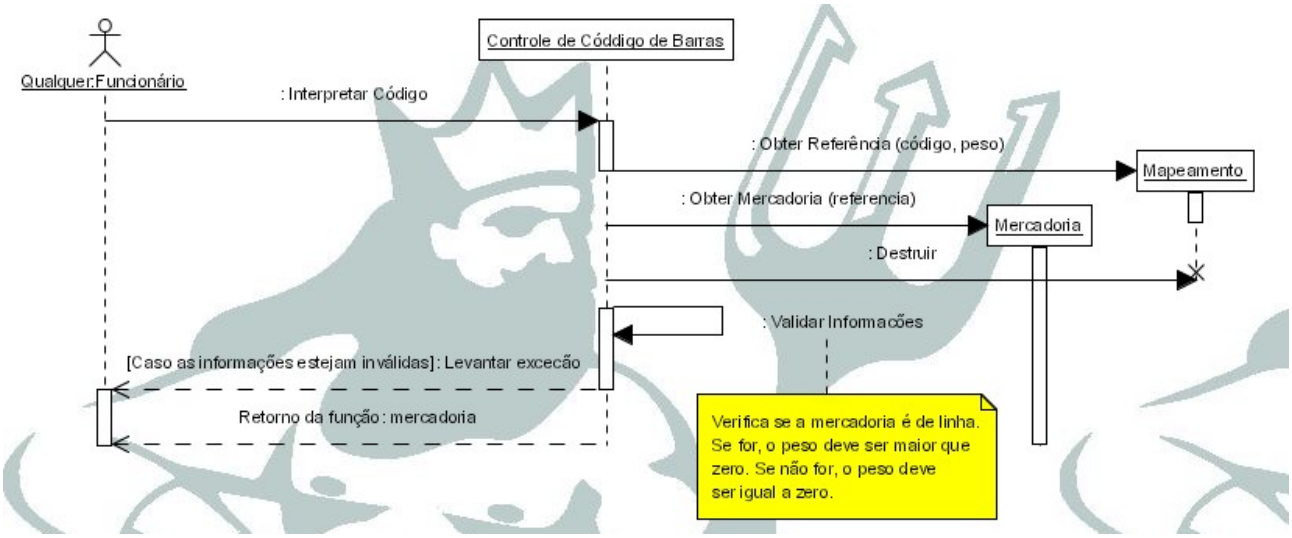
Para os formatos de três e quatro pares, os três primeiros dígitos são normalmente utilizados para pesos. Como todos os dígitos são nulos, não existe peso na etiqueta.

Qualidade do código de barras

Como o número de dígitos no código de barras influencia a facilidade de leitura com a maior tolerância a distanciamento do leitor em relação à etiqueta, aquelas com menor quantidade de pares são preferenciais. A tabela a seguir avalia a qualidade da leitura conforme quantidade de pares para as etiquetas utilizadas na IMJ em 2004, que possui largura de 2cm, utilizando a impressora HP PSC 750 na resolução máxima (600x600 dpi).

Pares	*Qualidade*
3	Muito bom
4	Bom

Para mapear as 1.658 referências impressas sem peso em 2004 na IMJ, serão utilizadas referências de 3 e 4 pares. Para mapear as 186 referências impressas com peso em 2004 na IMJ serão utilizadas referências de 3 pares, cuja qualidade é muito boa, restando ainda 814 referências com 3 pares, considerando todas as referências com o mesmo peso.



Capítulo 8

Sistema Para Recepção

A recepcionista é responsável pelo registro de entrada e saída de pessoas da empresa, registro de ligações realizadas para telefones celulares, interurbanos ou ligações recebidas a cobrar, além de outras tarefas que não necessitam registro, como realizar ligações locais, consultando uma agenda telefônica. Outra tarefa é o controle do rodízio de atendimento no setor de vendas.

Não é apropriado que ela tenha acesso aos dados cadastrados dos clientes da empresa, exceto dos números de telefone, por segurança de informação. (Confirmar se realmente pode mostrar telefone, cidade e estado de todos os clientes)

Anteriormente ao sistema, para o controle de entrada e saída, a recepcionista registrava em papel o nome da pessoa (normalmente apenas o primeiro nome), o setor de atendimento, o turno do dia, o código do vendedor, a data e hora de entrada e saída e a cidade de origem do cliente. O registro de ligações também era feito em papel, contendo a data, o ddd, número do telefone, a pessoa que originou a ligação, o destinatário (se cliente ou particular), o nome do vendedor (caso seja recepção de ligação a cobrar), a hora e a cidade.

Fora requisitado por Napoleão um sistema para o registro computacional dos dados, permitindo avaliar o movimento na empresa por horário, consulta de agenda telefônica, um mecanismo de auxílio à verificação por parte da recepcionista de qual sócio estaria presente na empresa e o controle automático de rodízio de atendimento.

Com o sistema desenvolvido, torna-se possível facilmente registrar essas informações no computador, permitindo ainda que seja analisado a qualquer momento os dados estatísticos sobre os visitantes, tais como motivo das visitas, tempo de espera por atendimento na repceção (informações por setor) e crescimento de visitantes por dia e/ou setor. (Figura)

Ao invés de apenas armazenar as informações no banco de dados, foi construído um servidor que mantém informações na memória sobre as pessoas que estão na empresa. Assim, é possível construir programas que visualizem os clientes que estão na empresa e por qual funcionário/setor ele está sendo atendido.

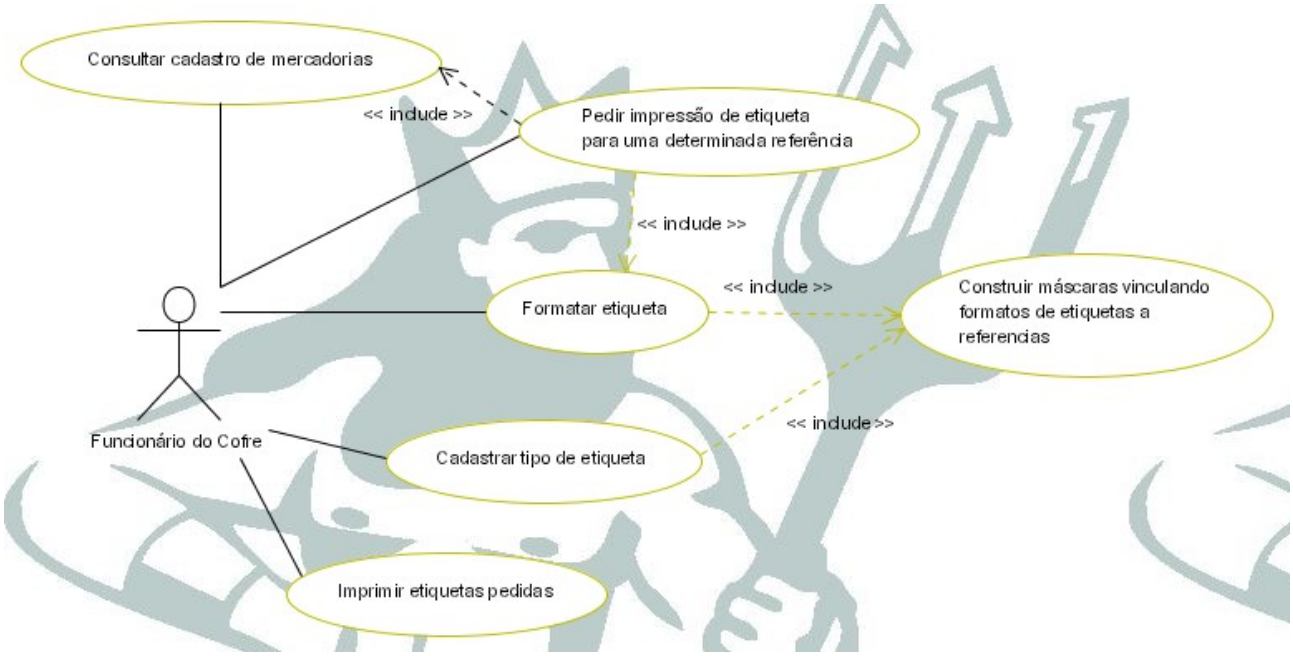
Ao efetuar o registro de entrada de visitante/cliente, é possível alocá-lo a um rodízio de setor ou a uma fila destinada ao atendimento específico de um funcionário. O funcionário só atenderá o cliente inserido em sua fila única, se este cliente for o primeiro da fila e sua hora de entrada for anterior ao primeiro da fila do rodízio do setor. A prioridade é dada pelo momento de chegada na empresa. Logo que um funcionário estiver disponível para atendimento, a recepcionista será alertada para encaminhar o cliente ao funcionário em questão. Um alarme sonoro é tocado caso ela não confirme o alerta em 5 segundos.

O registro de telefonema foi implementado de forma simples. A única tarefa da recepcionista é registrar os dados, não podendo, no entanto, consultá-los ou mesmo alterá-los.

Foi construída uma agenda telefônica contendo todos os telefones dos clientes cadastrados da empresa. A recepcionista pode cadastrar telefones, porém estes dados não são inseridos no banco de dados do servidor, mas em um arquivo no computador local. Somente estes dados podem ser alterados ou removidos.

Capítulo 9

Sistema Para O Cofre



Capítulo 10

Sistema Para O Setor De Varejo

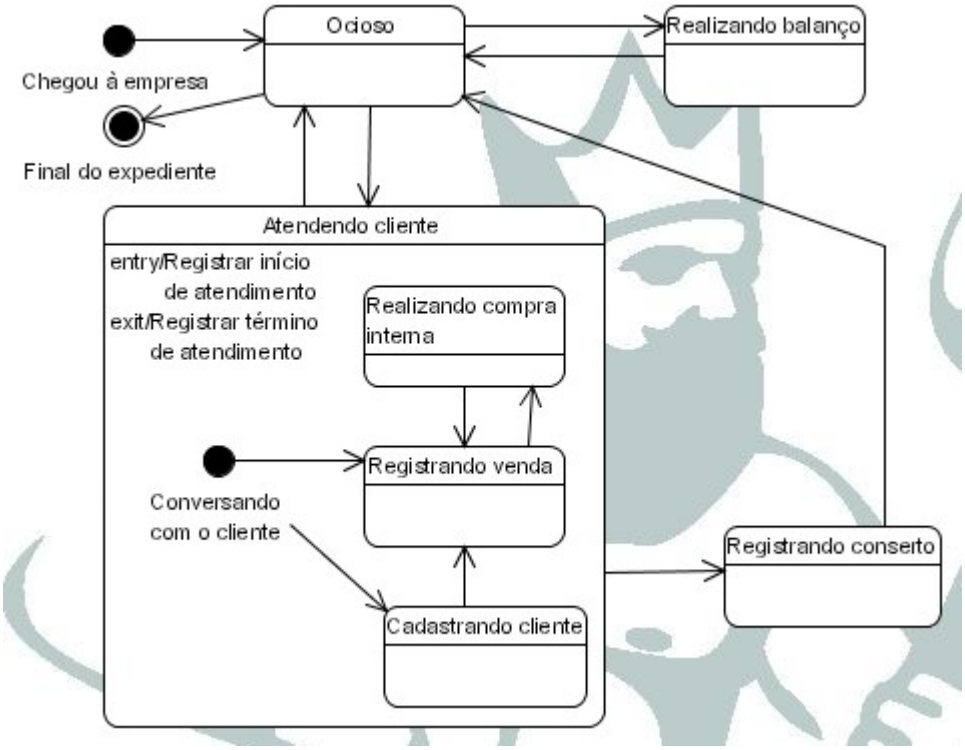


Figura 1- Estados de um vendedor do varejo.

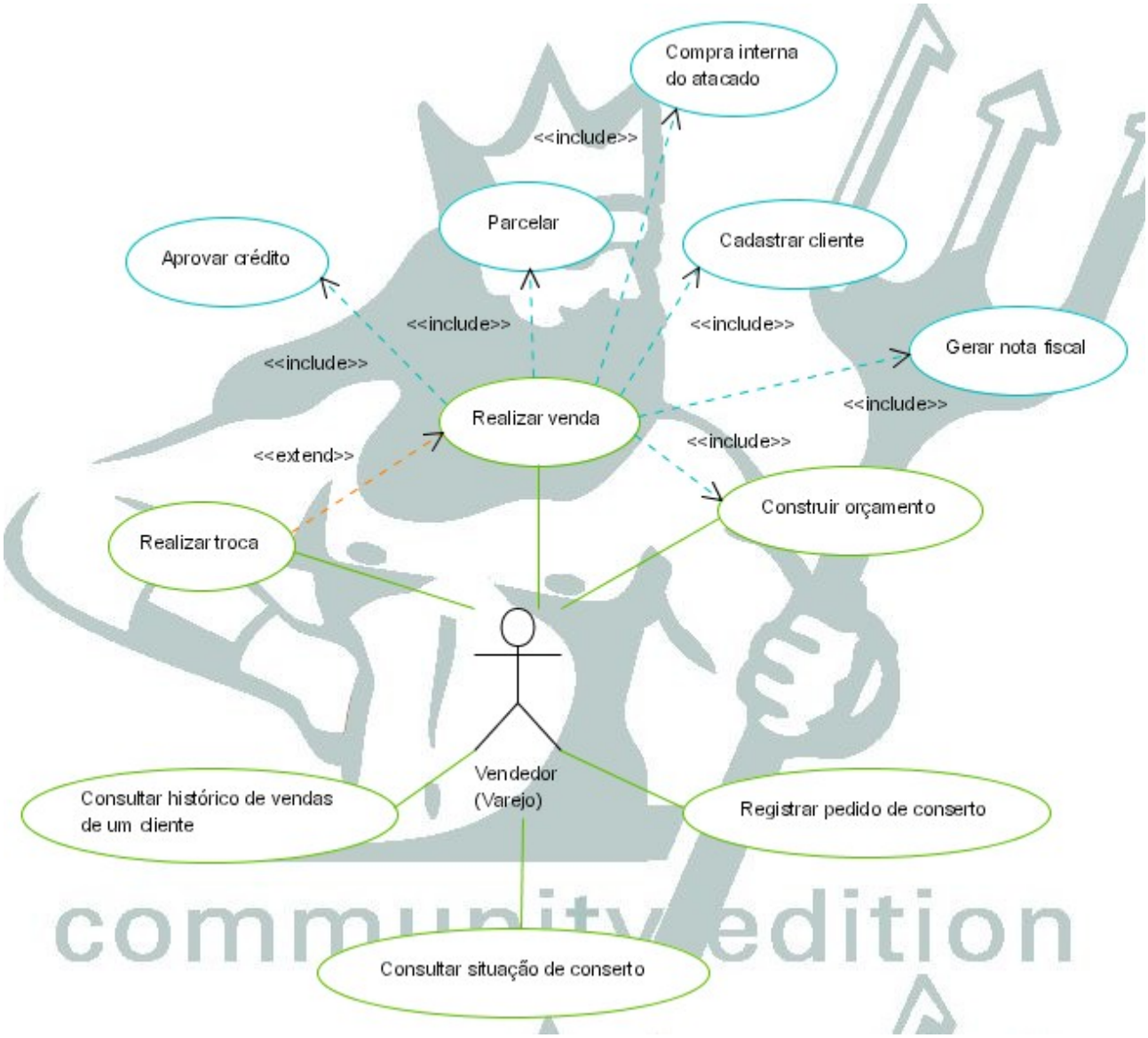


Figura 2- Caso de uso do sistema para o setor de varejo por parte do vendedor.

Capítulo 11

Questões Pendentes

Classe Entidades.VisitaAtendimento

O modelo de classe está ruim. Poderia ser desfeito, transferindo suas funcionalidades à classe Visita.

Enumeração Entidades.MotivoContato

Melhor posicionada na camada de negócio.

Diretório do projeto Negócio – Controle.Compartilhado

O projeto *Negócio – Controle.Compartilhado* está erroneamente nomeado de "Interface – Negócio.Contexto".

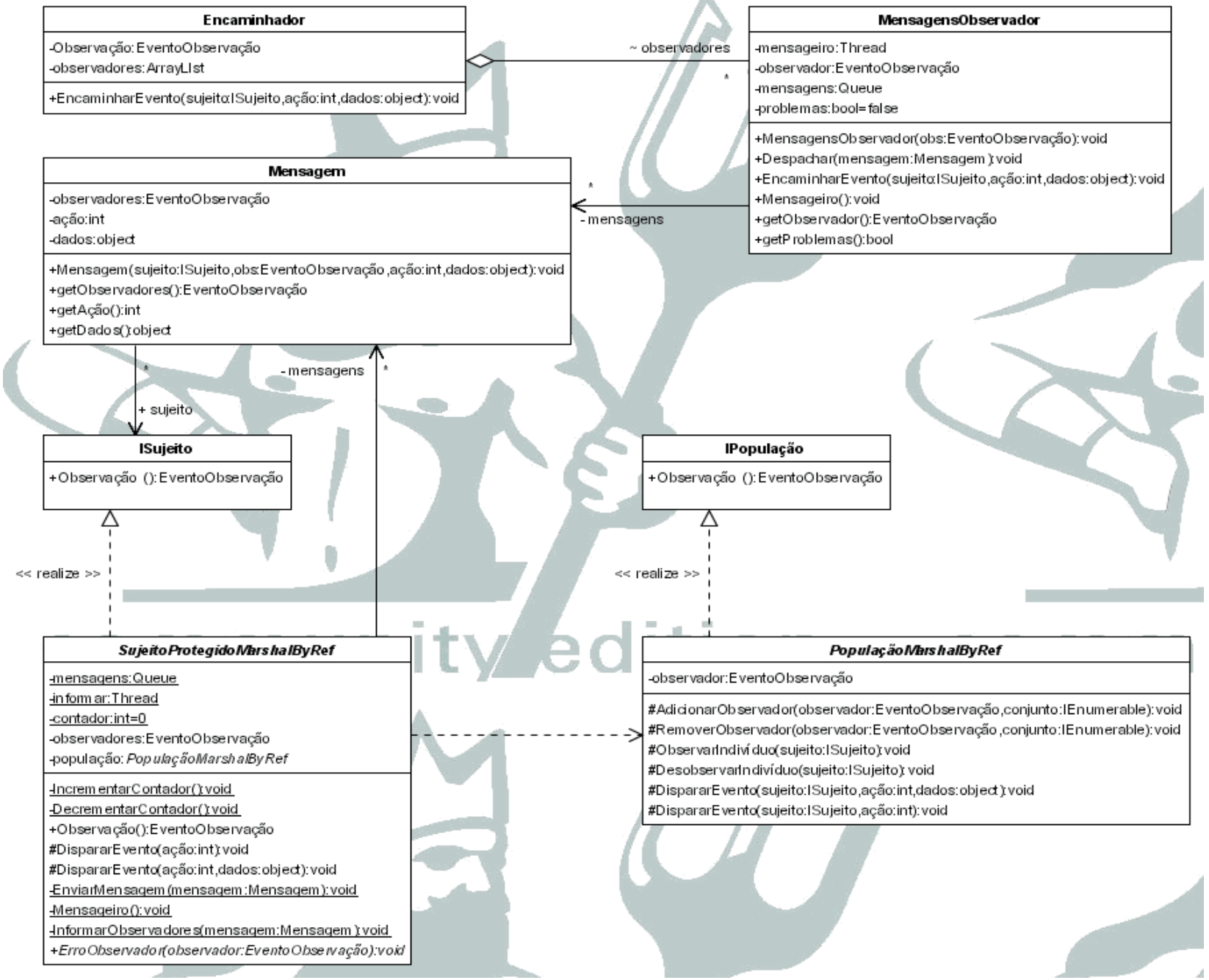
Observador.SujeitoProtegido está obsoleto

A classe *Observador.SujeitoProtegido* é obsoleta e deverá ser substituída por *Observador.SujeitoProtegidoMarshalByRef*, uma vez que os sujeitos devem ser controlados/manipulados pelo servidor.

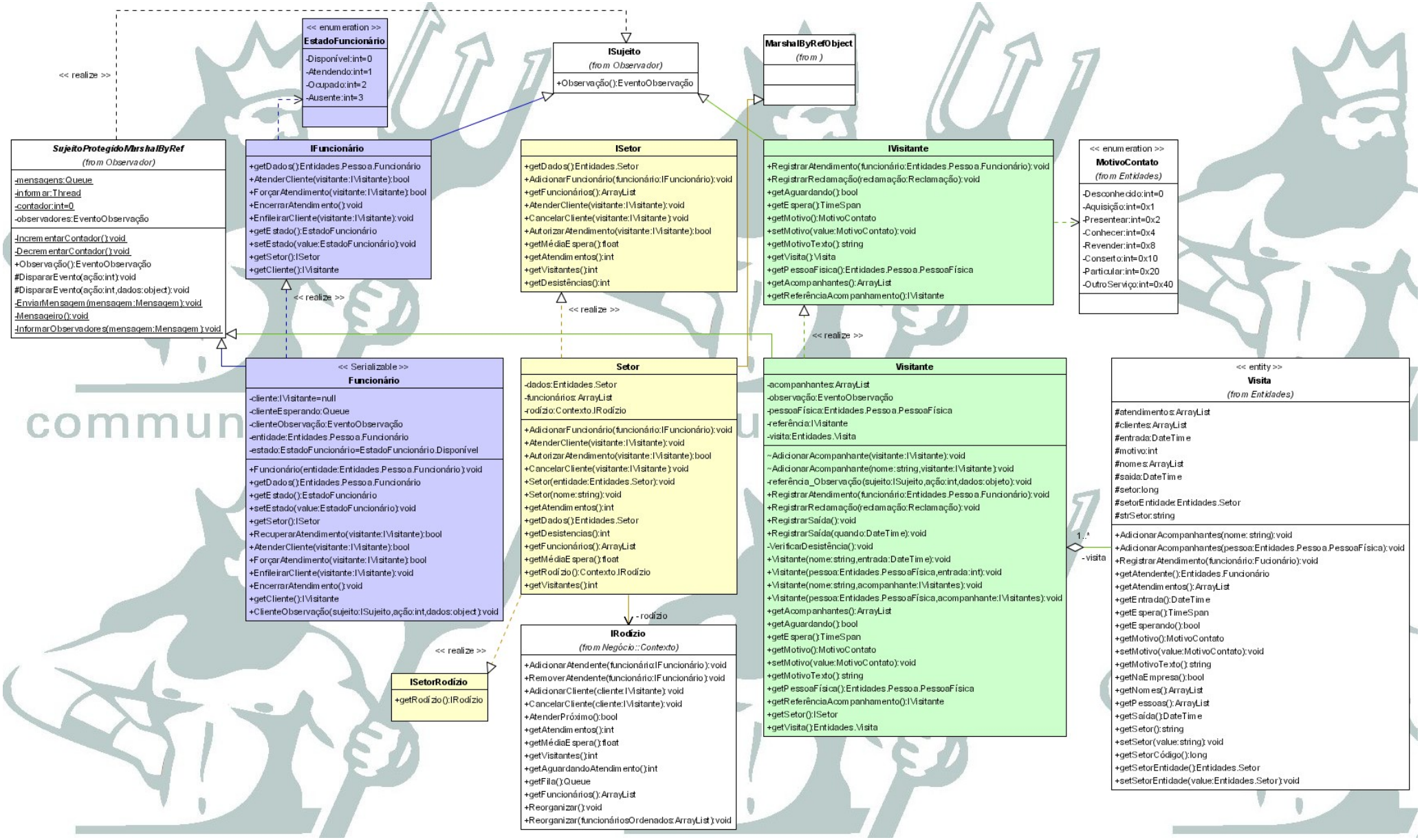
Anexo A

Diagrama De Classes

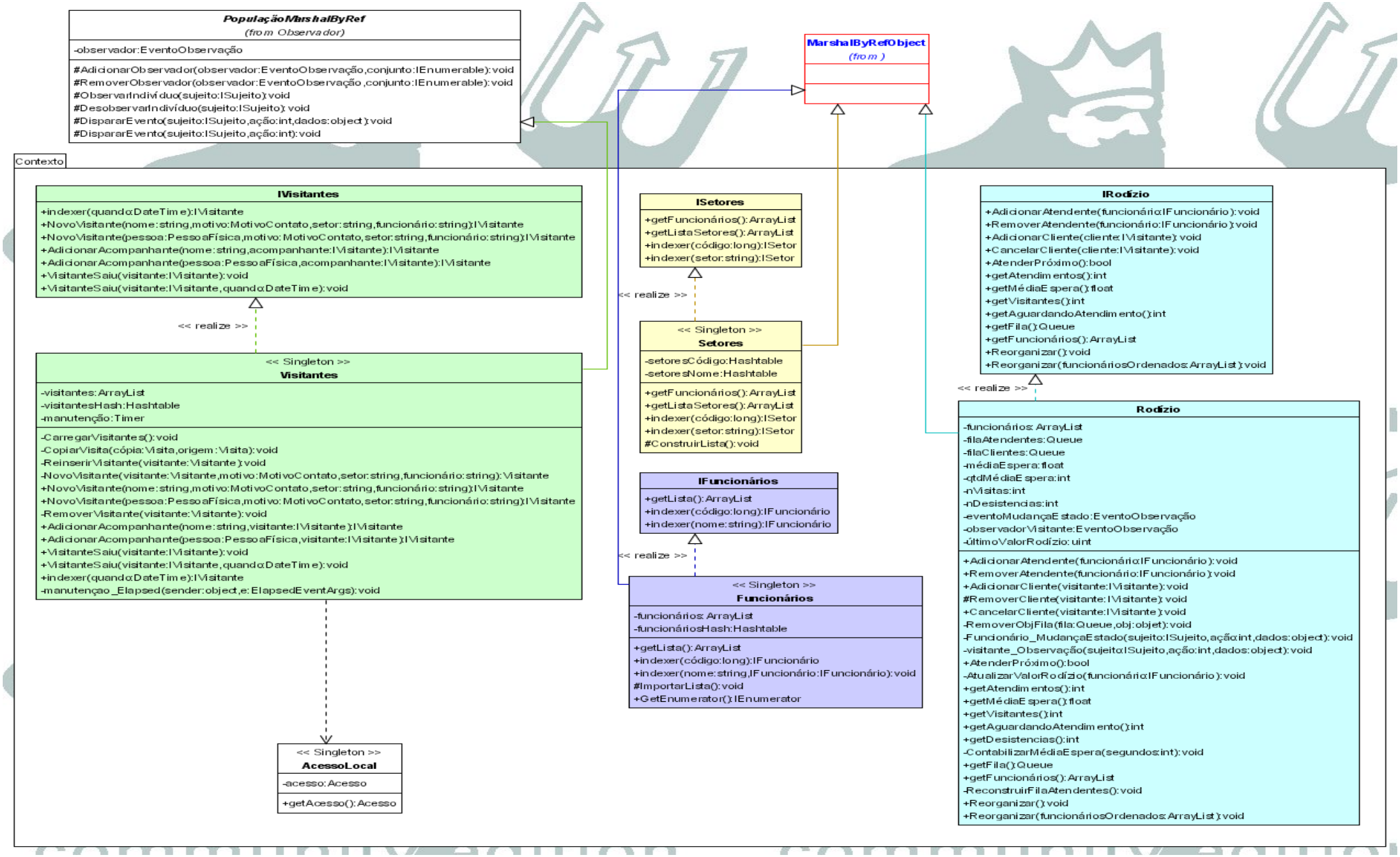
Classes - Namespace: Observador



Classes – Namespace: Negócio



Classes – Namespace: Negócio.Contexto



Anexo B

Erros Desenvolvendo

“Client does not support authentication protocol requested by server; consider upgrading MySQL client”

A partir da versão 4.0, o SGDB MySQL utiliza um protocolo de autenticação diferente das versões anteriores. A versão 0.7.2.13867 da biblioteca ByteFX.Data.dll não implementa este novo protocolo e, portanto, não consegue autenticar usuários com senha normalmente. Para solucionar este problema, dois procedimentos devem ser realizados:

1. Reiniciar o servidor MySQL com a opção “–old-passwords”;
2. Alterar a senha do usuário para que esta seja então armazenada no formato antigo pelo MySQL.

“Unknown transmission status: sequence out of order”

Ocorre quando várias *querys* são feitas simultaneamente em uma mesma conexão ao banco de dados. Isso pode ocorrer se a camada de acesso é compartilhada, como o objeto da classe *singleton AcessoLocal*, na camada de negócio.

Para solucionar este problema, deve ser usado o comando “lock (this)” na camada de acesso, para garantir que apenas uma operação no banco de dados é realizada em um determinado momento.

Sistema trava ao manipular dados no banco de dados

Parece existir um bug na biblioteca da ByteFX que acarreta em travamento ao fechar duas vezes um objeto do tipo *MySqlDataReader*.

Após um tempo funcionando, o sistema perde referência de objeto remoto

Os objetos remotos possuem tempo de vida curto. Para tornar o tempo de vida infinito, deve-se substituir o método “InitializeLifetimeService”, conforme exemplo a seguir:

```
public override object InitializeLifetimeService()
{
      return null;
}
```

Anexo C

Dead-lock

O dead-lock ocorre em dois casos: (i) quando duas *threads* concorrentes dependem de mesmas travas; (ii) quando duas *threads* dependem da pilha de chamadas da outra.

O primeiro caso pode ser facilmente enxergado, conforme exemplo a seguir:

Programa A	Programa B
lock (a)	lock (b)
lock (b)	lock (a)
(...)	(...)

Caso os programas A e B executem concorrentemente cada instrução, o programa A irá esperar pelo recurso b enquanto o programa B irá esperar pelo recurso A, infinitamente.

Infelizmente, os dead-locks que ocorrem na programação raramente são do tipo ilustrado acima. Observe o código[3] a seguir:

```
/// <summary>
/// Carrega lista de pessoas provenientes de setores específicos
/// em um período específico.
/// </summary>
/// <param name="setores">Lista de setores.</param>
/// <param name="início">Período inicial.</param>
/// <param name="final">Períofo final.</param>
void CarregarDeSetores(Setor [] setores, DateTime início, DateTime final)
{
      ArrayList      itens          = new ArrayList();
      DCriEntPesIte  métodoCriarEntidadePessoaItem;

      lock (this)
      {
            SinalizarCarga();

            métodoCriarEntidadePessoaItem =
                  new DCriEntPesIte(CriarEntidadePessoaItem);

            lock (listaPessoas.Itens.SyncRoot)
            {
                  foreach (Setor setor in setores)
                  {
                        Pessoa [] pessoas;

                        pessoas = Pessoa.ObterPessoas(setor, início, final);
```

3 Este código foi extraído da classe *Apresentação.Atendimento.Clientes.BaseSeleçãoCliente*. Para facilitar a visualização, parte do código foi removido.

```
                        /* O framework dotNet exige que controles
                         * vinculados sejam construídos em um mesmo
                         * contexto. Como este método é chamado de forma
                         * assíncrona, o contexto do método e do controle
                         * são diferentes, exigindo construir os itens no
                         * contexto de ListaPedidos.
                         */
                        object resultado;

                        resultado = listaPessoas.Invoke(
                              métodoCriarEntidadePessoaItem,
                              new object [] { pessoas });

                        itens.AddRange((ICollection) resultado);
                  }
            }

            DAdicionarItens método = new DAdicionarItens(AdicionarItens);
            listaPessoas.Invoke(método , new object [] { itens });
      }
}


/// <summary>
/// Sinaliza início de carga de dados.
/// </summary>
private void SinalizarCarga()
{
      DSinalizarCarga método;

      método = new DSinalizarCarga(listaPessoas.SinalizarCarga);
      listaPessoas.Invoke(método, new object [] { strCarregando });
}

/// <summary>
/// Adiciona itens à ListaPessoa.
/// </summary>
/// <param name="itens">Itens a serem adicionados.</param>
private void AdicionarItens(ICollection itens)
{
      listaPessoas.Dessinalizar();
      listaPessoas.Itens.AddRange(itens);

      foreach (ListaPessoasItem item in itens)
            item.Visible = true;
}


--- ListaPessoas ------------------------------------------------------------

/// <summary>
/// Sinaliza que a lista está carregando.
/// </summary>
/// <param name="descrição">Descrição personalizada.</param>
public void SinalizarCarga(string descrição)
{
      itens.Clear();
      AdicionarSinalização(new SinalizaçãoCarga(descrição));
}


--- Itens -------------------------------------------------------------------

/// <summary>
/// Limpa a lista.
```

```
/// </summary>
public override void Clear()
{
	lock (listaPessoas.Itens.SyncRoot)
	{
		foreach (ListaPessoasItem item in this)
		{
			listaPessoas.Controls.Remove(item);
			((ListaPessoasItem) item).Fechar -= itemFechar;
		}
		base.Clear();
	}
}
```

Anexo D

Glosário

GPL General Public License

IMJ Indústria Mineira de Jóias

LGPL Lesser General Public License, disponível em http://www.gnu.org/copyleft/lesser.html

Singleton Padrão de desenvolvimento, em que as classes possuem instância única.

UI Interface do Usuário, do inglês *User Interface*. Exemplo: janelas, web sites, console, LEDs.