André Yamasaki Martins Vieira

Daniel Abreu Vasconcelos de Paula

Priscilla Barreira Avegliano

Rogério Kakehashi

SARC – SISTEMA DE AUTOMAÇÃO DOS RESTAURANTES COSEAS/USP

Monografia apresentada à Escola Politécnica da Universidade de São Paulo, como requisito parcial para a graduação no curso de Engenharia de Computação.

São Paulo

2005

André Yamasaki Martins Vieira

Daniel Abreu Vasconcelos de Paula

Priscilla Barreira Avegliano

Rogério Kakehashi

SARC – SISTEMA DE AUTOMAÇÃO DOS RESTAURANTES COSEAS/USP

Monografia apresentada à Escola Politécnica da Universidade de São Paulo, como requisito parcial para a graduação no curso de Engenharia de Computação.

Orientadora:
Profa. Dra. Selma Shin Shimizu Melnikoff

São Paulo
2005

"Nosso cuidado não deve ser o de viver muito, e sim o de viver bem, porque o primeiro depende do destino, e o segundo de nossa conduta". Sêneca (4 a.C. – 65 d.C.)

"Há pessoas que desejam saber só por saber, e isso é curiosidade: outras, para alcançarem fama, e isso é vaidade; outras, para enriquecerem com a sua ciência, e isso é um negócio torpe; outras, para serem edificadas, e isso é prudência; outras, para edificarem os outros, e isso é caridade"

São Tomás de Aquino

AGRADECIMENTOS

Dedicamos os nossos mais sinceros agradecimentos à Profª. Dra. Selma Shin Shimizu Melnikoff pela amizade, comprometimento, conhecimento compartilhado e por sua orientação precisa e objetiva em diversos momentos de dúvidas; aos demais professores da Escola Politécnica da Universidade de São Paulo, por dividirem parte valiosa de seus conhecimentos, os quais foram fundamentais em cada parte desta caminhada; aos nossos pais, por estarem sempre presentes; e a todos os nossos colegas, os quais tivemos o prazer de vivenciar estes cinco anos de nossas vidas e que se mostraram verdadeiros amigos na busca pelo mesmo objetivo. A todos vocês, nosso muito obrigado.

RESUMO

O objetivo deste trabalho é desenvolver um sistema computacional que auxilie o gerenciamento dos diversos departamentos responsáveis pelo funcionamento dos restaurantes COSEAS/USP, controlando questões nutricionais, estoque dos produtos utilizados no preparo das refeições, transações financeiras e o acesso dos clientes aos restaurantes por meio de catracas eletrônicas e Smart Cards. Surgiu da necessidade da modernização do sistema utilizado atualmente, de capacidade limitada de armazenamento, com poucas funcionalidades e de interface pouco amigável.

Palavras-chave: J2EE, *Design Patterns*, MVC, *Smart Card*, Servidores de Aplicação, *Frameworks* de desenvolvimento.

ABSTRACT

The objective of this project is to develop a computational system to aid the management of the several departments responsible for the COSEAS/USP restaurants operation, controlling nutritional points, stock levels, financial transactions and access to the restaurants using Smart Cards through electronic ratchets. It has emerged from the modernization needed from current system, which has a limited storage capability, few functionalities and a non-friendly interface.

Keywords: J2EE, Design Patterns, MVC, Smart Card, Application Servers, Development Frameworks.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

APDU – *Application Processing Data Unit*

API – *Application Programming Interface*

B2B – *Business to Business*

BPD – Business Process Diagram

BPEL4WS – *Business Process Execution Language for Web Services*

BPM – *Business Process Management*

BPMI – Business Process Management Initiative

BPMN – Business Process Modeling Notation

CPU – Unidade Central de Processamento

CVS – Concurrent Versions System

EEPROM – *Electrically Erasable Programmable Read-Only Memory*

EJB – *Enterprise JavaBeans*

GPL – *General Public License*

HTTP – Hyper Text Transfer Protocol

ISO – *International Organization for Standardization*

J2EE – *Java 2 Plataform, Entreprise Edition*

J2SE – *Java 2 Plataform, Standard Edition*

JDBC – *Java Database Connectivity*

JMS – *Java Message Service*

JSP – *Java Server Pages*

MIP – Milhão de Instruções por Segundo

MVC – *Model View Controller*

ODBC – *Open DataBase Connectivity*

POJO – *Plain Old Java Object*

RAM – *Random Access Memory*

ROM – *Read Only Memory*

SARC – Sistema de Automação dos Restaurantes COSEAS/USP

SQL – *Structured Query Language*

UML – *Unified Modeling Language*

URL – Uniform Resource Locator

USP – Universidade de São Paulo

XML – *Extensible Markup Language*

1 INTRODUÇÃO

O objetivo deste trabalho é projetar um sistema capaz de modernizar os processos envolvidos nos restaurantes COSEAS/USP e desenvolver um protótipo que mostra a viabilidade da sua implementação. O sistema fornece apoio para a venda de tickets refeição e o monitoramento da entrada de clientes nos restaurantes, além de automatizar alguns dos principais processos de seu gerenciamento, controlando o estoque de insumos, movimentações financeiras e informações nutricionais referentes às refeições servidas diariamente nos restaurantes localizados em diversas unidades da Universidade de São Paulo.

As principais funcionalidades do sistema englobam:

- A venda de *tickets* refeição pela internet;
- O uso de *Smart Cards* visando uma maior praticidade em operações de crédito e débito;
- O aprimoramento do controle de acesso aos restaurantes providos de catracas eletrônicas;
- A elaboração de relatórios gerenciais;
- O monitoramento da quantidade e da validade de produtos armazenados nos estoques, verificando a necessidade de novas compras de acordo com o nível atual, e a quantidade demandada em um certo período;
- A produção automática do cardápio semanal, procurando atender todos os requisitos de custo, valor calórico e diversificação de itens.

Para alcançar o objetivo proposto, necessitou-se primeiramente compreender a relação entre os processos de todos os módulos responsáveis pelo correto funcionamento dos restaurantes, desde procedimentos básicos de compra de insumos até a entrega do produto final ao consumidor e a venda de tickets refeição aos clientes. Em uma primeira etapa, foram realizadas diversas reuniões com os gerentes destes módulos a fim de coletar todas as informações necessárias para compreender os requisitos funcionais que o sistema deveria apresentar no final do projeto. A cada encontro foi gerado um documento de especificação relativo ao módulo em questão, que serviu de base para a definição dos casos de uso envolvidos.

Todo este trabalho surgiu da necessidade da reforma do sistema utilizado atualmente, de capacidade bastante limitada de armazenamento, com funcionalidades restritas e de baixa influência no escopo do processo geral, além de possuir uma interface homem-máquina pouco amigável e intuitiva. Pretendeu-se também eliminar a possibilidade de falsificação dos tickets refeições, além do gasto com papel e tinta de impressão. Há um controle manual de gastos, que favorece a ocorrência de erros e a perda desnecessária de tempo com os cálculos envolvidos. Além do que foi mencionado, aliamos a este cenário o nosso desejo de criar um projeto em benefício à universidade, que tanto contribuiu para a nossa formação pessoal e profissional.

O Capítulo 2 trata da modelagem dos processos envolvidos em BPMN (*Business Process Modeling Notation*), um padrão de notação desenvolvido pela BPMI (*Business Process Management Initiative*) para representar processos de negócio por meio de diagramas. Esta notação foi utilizada para descrever os principais processos levantados junto ao cliente. O Capítulo 3 descreve a arquitetura do sistema, bem como o *hardware* e o *software* utilizados no protótipo para testes do mesmo. O Capítulo 4 cita o ambiente de desenvolvimento e o domínio da solução. Os resultados dos testes e as dificuldades encontradas são mostrados no Capítulo 5. No Capítulo 6 é apresentada a conclusão do trabalho e, em seguida, encontram-se os anexos e as referências bibliográficas que serviram como base para a realização deste projeto.

2 MODELAGEM DOS PROCESSOS DE NEGÓCIO

2.1 Introdução

Na fase inicial do projeto procurou-se compreender os principais processos de negócio nos diferentes módulos que formam a administração dos restaurantes COSEAS/USP. A modelagem destes processos foi elaborada seguindo a especificação BPMN (*Business Process Modeling Notation*), desenvolvida pela BPMI (*Business Process Management Initiative*). A seguir, são descritos brevemente os seus principais conceitos e objetivos.

2.2 BPMN – Business Process Modeling Notation

Desenvolvido pelo BPMI (*Business Process Management Initiative*), uma organização independente e sem fins lucrativos que desenvolve especificações abertas para o gerenciamento de processos de negócio, o BPMN é um padrão de notação gráfica que se apóia em diagramas para descrever processos de negócio [1]. Esta notação foi projetada para representar a seqüência de processos e mensagens que fluem em um conjunto relacionado de atividades. Um dos seus principais objetivos é ser facilmente interpretado por todas as pessoas envolvidas nos processos de negócios, desde analistas que criam os primeiros esboços dos processos, até desenvolvedores que implementam a tecnologia necessária, além dos participantes que irão monitorar e gerenciar tais processos.

A meta é padronizar a modelagem de processos de negócio e alcançar o entendimento por parte de todas as pessoas envolvidas no projeto, independentemente de sua posição, através de uma linguagem simples e intuitiva. O benefício para os seus usuários acontece de forma similar ao provocado pela UML (*Unified Modeling Language*) quando esta padronizou a representação de software orientado a objetos.

Um segundo objetivo do BPMN, embora não aplicado diretamente neste projeto, é assegurar que linguagens projetadas para a execução dos processos de negócio como a XML (*Extensible Markup Language*) e a BPEL4WS (*Business*

Process Execution Language for Web Services) possam ser representadas com esta notação. Estas linguagens estão sendo utilizadas em larga escala nos sistemas de BPM (*Business Process Management*) que operam com *web services*. Elas são otimizadas para a operação e interoperação entre sistemas de BPM, mas são menos adequadas (legíveis) para o uso direto daqueles que projetam, gerenciam e monitoram os processos envolvidos. Apesar de os sistemas de *software* conseguirem tratar, com estas linguagens, processos relativamente complexos, desarticulados e organizados de uma forma não intuitiva, o entendimento humano é prejudicado, uma vez que usualmente é mais confortável visualizar processos de negócio em um formato gráfico. Este contexto criou a necessidade de um mecanismo formal que pudesse mapear a visualização apropriada dos processos de negócio (a notação) em um respectivo formato apropriado da linguagem de execução de sistemas BPM (BPEL4WS). A interoperação entre os processos de negócio pode ser expressa com a padronização do BPMN, que provê um BPD (*Business Process Diagram*) para o uso de pessoas que projetam e gerenciam os processos de negócio. Futuramente, a capacidade desta notação será expandida com a incorporação da representação de conceitos como processos do tipo *public* e *private*, tratamento de exceções, *transactions*, entre outros.

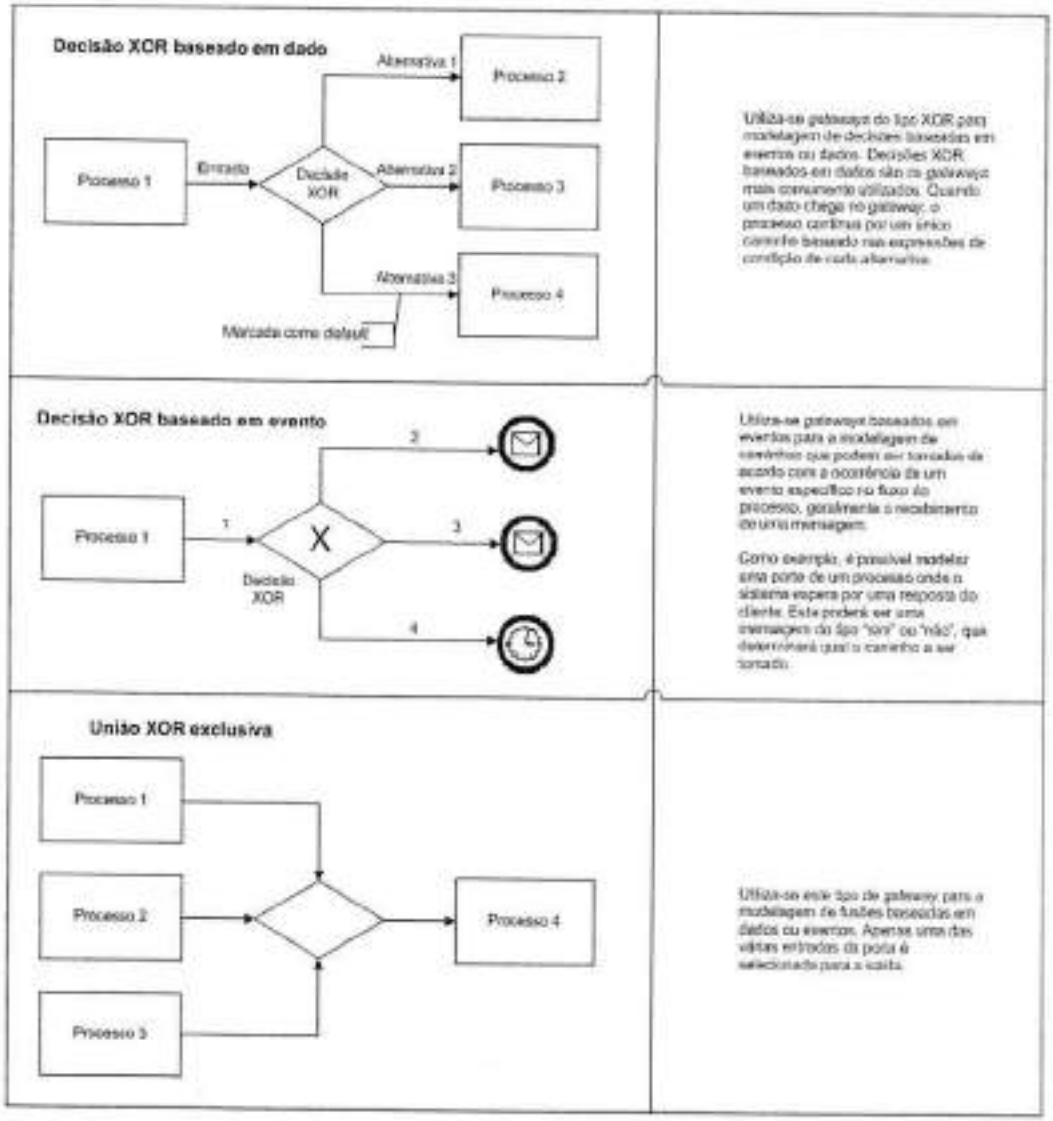
2.3 Modelagem – princípios e notações básicas

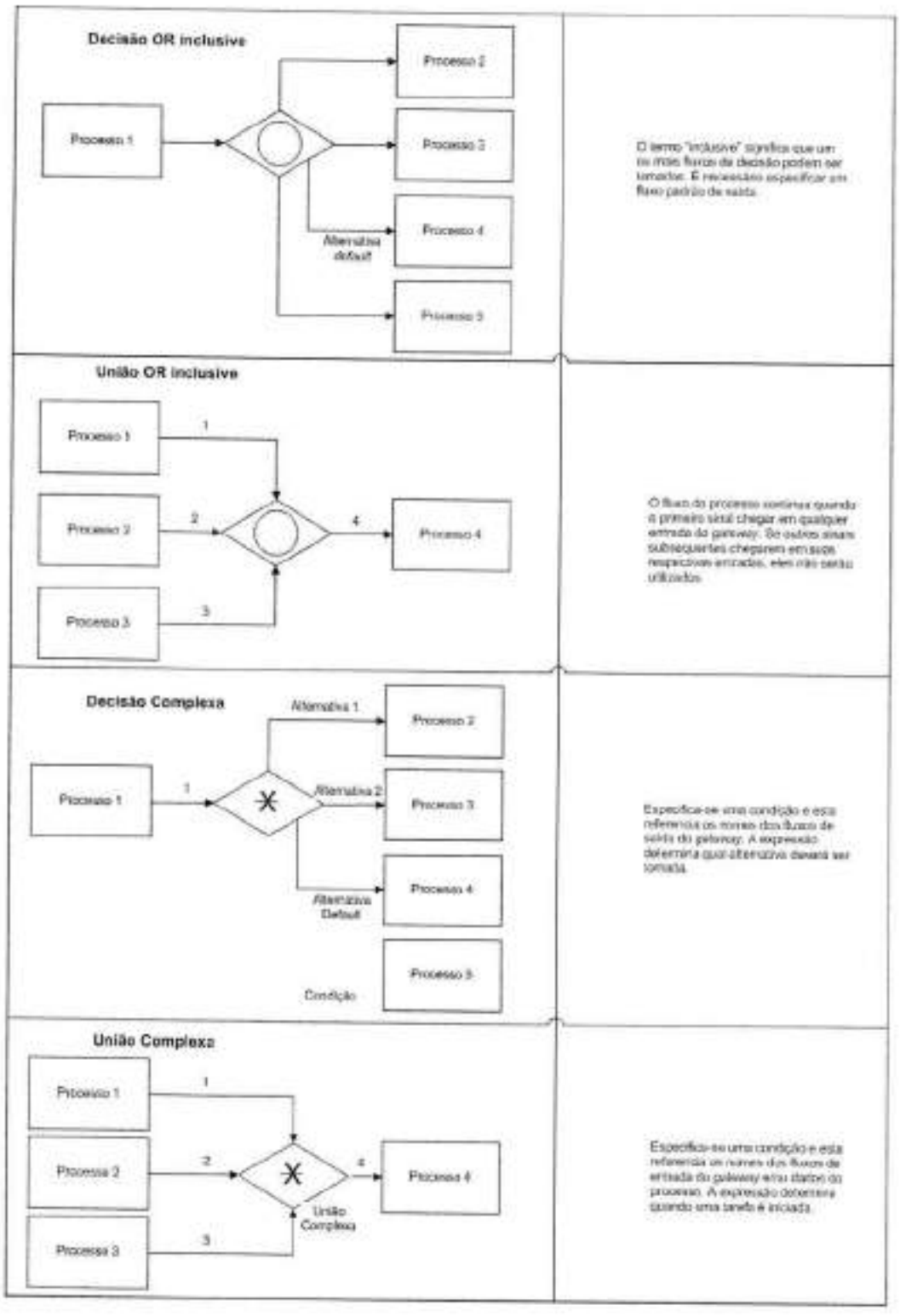
Conforme mencionado na seção anterior, o padrão BPMN especifica a elaboração de um diagrama chamado BPD (*Business Process Diagram*) para representar os processos de negócio de interesse. Este diagrama foi projetado para realizar eficientemente duas tarefas: em primeiro lugar, deve ser de fácil utilização e entendimento por parte de todos os seus usuários, incluindo pessoas de áreas não técnicas (usualmente a gerência). Em segundo lugar, deve oferecer toda a expressividade necessária para a modelagem de processos de negócio relativamente complexos, além de permitir o mapeamento natural para linguagens de execução [1].

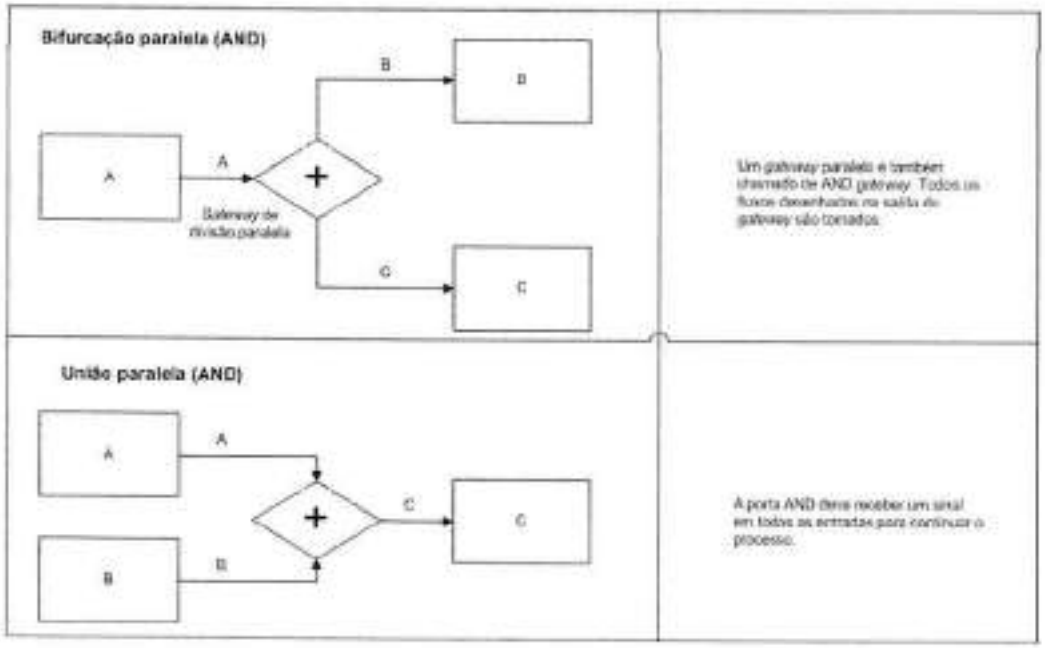
Para que seja feita a modelagem de todo o fluxo do processo, devem-se modelar os eventos responsáveis pela ativação do processo, os que são efetivamente executados e os resultados finais de todo o fluxo.

Etapas de decisão e ramificações de fluxos são modeladas utilizando-se elementos denominados *gateways*, cujos símbolos são similares aos de decisão encontrados em fluxogramas comuns (representada pela figura geométrica de um losango). Um *gateway* pode ser visto como uma questão que é lançada em um determinado ponto do fluxo do processo, apresentando um conjunto definido de alternativas a serem seguidas em função da resposta dada. Na Tabela 1 estão apresentados os tipos de representação envolvendo *gateways*:

Tabela 1 - Tipos de gateways e símbolos associados




Decisão OR inclusive
Processo 1
Processo 2
Processo 3
Processo 4
Alternativa default
Processo 5
O termo "inclusive" significa que um ou mais fluxos de decisão podem ser tomados. É necessário especificar um fluxo padrão de saída.
União OR inclusive
Processo 1
Processo 2
Processo 3
1
2
3
4
Processo 4
O fluxo do processo continua quando o primeiro sinal chegar em qualquer entrada do gateway. Os outros sinais subsequentes chegarem em suas respectivas entradas, eles não serão utilizados.
Decisão Complexa
Processo 1
1
Alternativa 1
Alternativa 2
Alternativa Default
Processo 2
Processo 3
Processo 4
Processo 5
Condição
Especifica-se uma condição e esta referencia os nomes dos fluxos de saída do gateway. A expressão determina qual alternativa deverá ser tomada.
União Complexa
Processo 1
Processo 2
Processo 3
1
2
3
4
União Complexa
Processo 4
Especifica-se uma condição e esta referencia os nomes dos fluxos de entrada do gateway e/ou default do processo. A expressão determina quando uma tarefa é iniciada.



Um processo pode englobar sub-processos, que podem ser graficamente mostrados por meio de outro BPD conectado via um *hyperlink* para um símbolo de processo. Se ele não puder ser decomposto, ele é considerado uma tarefa – o nível mais baixo de um processo. O uso destes diferentes substantivos simplesmente reflete a relação hierárquica entre os mesmos. Um sinal de '+' contido no símbolo do processo indica que o mesmo é composto por sub-processos (possui ao menos um diagrama-filho). Caso contrário, pode-se deduzir que ele se trata de uma simples tarefa.

Para representar a ordem de execução dos processos dentro de uma organização ou em um departamento, os processos são conectados através de elementos denominados *Sequence Flow*, cuja representação gráfica é uma linha terminada por uma seta preenchida. Um segundo tipo de linha de fluxo é conhecido como *Message Flow*, que serve para a comunicação entre diferentes organizações ou departamentos.

A Figura 1 apresenta, apenas com o intuito de ilustrar, um BPD de um sistema de leilão *online* com alguns dos itens básicos já descritos.

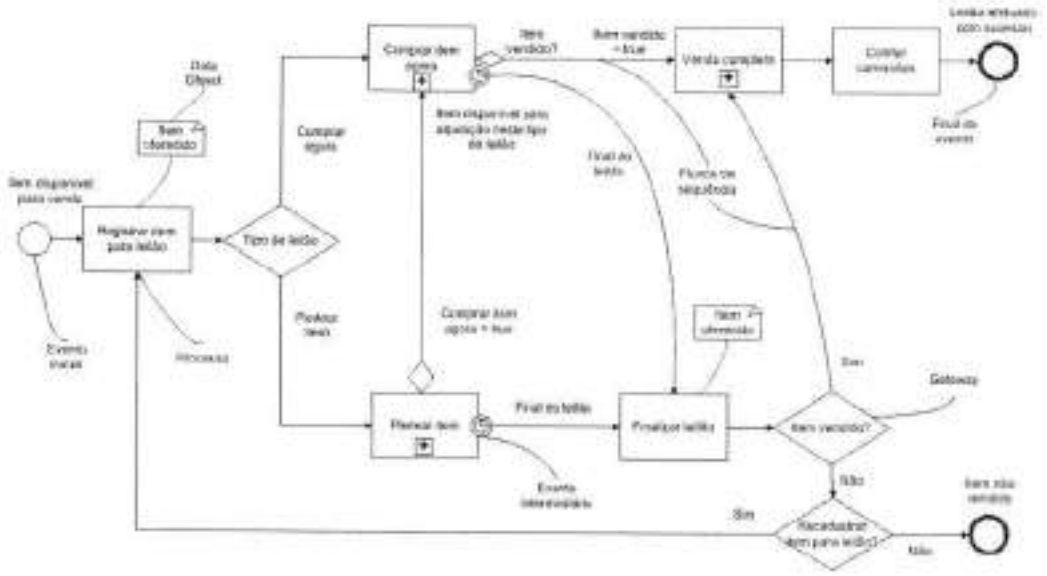


Figura 1 - Exemplo de um BPD para um sistema de leilão on-line

Pode-se especificar também o escopo das atividades, ou seja, dizer quem toma as decisões para executar um determinado conjunto de ações ou indicar onde elas ocorrem, delimitando seus processos e eventos dentro de uma área retangular sombreada denominada *pool*. Uma *pool* é utilizada tipicamente para representar uma organização, enquanto que uma *lane*, um departamento, conforme ilustra a Figura 2.

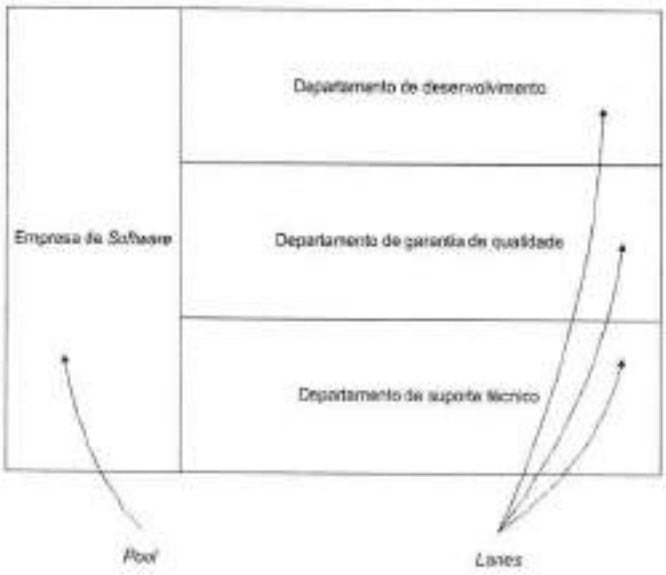


Figura 2 - Pools e Lanes

Podem-se ainda empregar as *lanes* para representar outros itens, como funções (algo que a organização executa, como *marketing*, vendas ou treinamento), aplicações (um *software*), locais (uma região física da companhia), classes (módulos de *software* em um programa orientado a objetos) e entidades (representando tabelas lógicas em um banco de dados). Existem ocasiões em que os processos necessitam seguir a execução em outras *pools*, porque são nelas que se encontram os demais recursos necessários para a continuação ou a finalização de uma atividade. Este elemento possui um especial destaque na descrição de processos B2B, onde diferentes organizações trocam mensagens entre si para completarem uma determinada atividade. A Figura 3 ilustra alguns dos itens descritos até o momento para o mesmo cenário da Figura 1 (um processo de leilão *online*):

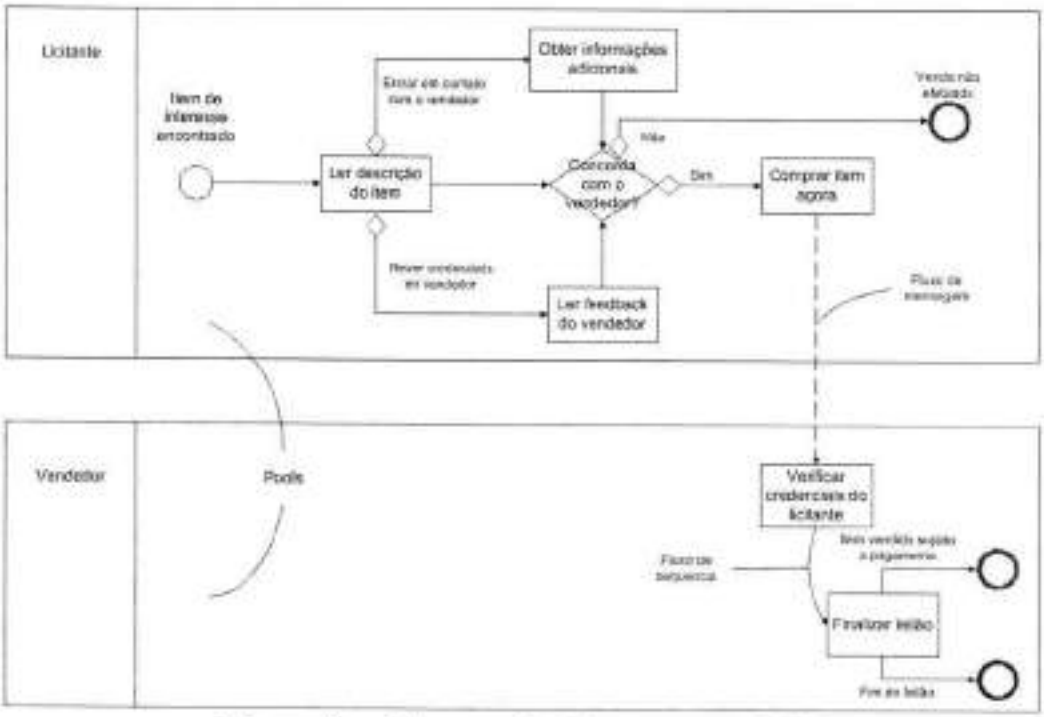


Figura 3 - Diagrama de Processos em BPMN

Durante a modelagem de um processo, é indispensável a representação dos eventos que disparam os fluxos de atividades, bem como aqueles que ocorrem no decorrer das etapas ou os que terminam um determinado processo. A simbologia que a notação BPMN reserva para estes elementos é ilustrada na Tabela 1.

Tabela 1 - Tipos básicos de eventos e suas notações

Evento inicial	Evento intermediário	Evento final
 Inicia o fluxo de um processo	 Ocorre durante um processo em andamento	 Finaliza o fluxo de um processo

Quando processos mais complexos, como aqueles envolvendo *B2B Web Services*, são modelados, é preciso representar eventos mais complexos, como mensagens, *timers* e condições de erro. Da mesma maneira, o padrão BPMN também disponibiliza símbolos correspondentes a cada tipo particular de evento, como exemplificados na Tabela 2.

Freqüentemente um evento ocorre durante um processo em execução, causando a sua interrupção e o disparo de um novo processo. É comum também que um evento aconteça ao final de um processo, provocando o início da execução de um novo. A modelagem destes eventos intermediários se dá colocando-se diretamente o símbolo adequado junto ao processo ao qual ele está associado.

Por tudo o que foi exposto neste capítulo, percebe-se que o padrão BPMN possui diversas características que o tornam um padrão de modelagem de processos de negócio e *web services* bastante interessante [1].

Foram modelados os principais processos de negócio da COSEAS/USP, relevantes ao Projeto de Formatura, segundo os critérios estabelecidos pela BPMN, representando as suas atividades e os fluxos de controle que definem a ordem em que elas são executadas. Os exemplos se encontram no Anexo A desta monografia. Vale ressaltar que foi elaborada apenas uma parte de sua especificação, objetivando a obtenção de uma familiaridade mínima para a compreensão do negócio, a qual permitiu a especificação do sistema desenvolvido.

Tabela 2 - Tipos particulares de eventos e suas notações

Eventos iniciais	Eventos intermediários	Eventos finais	Descrição
Mensagem inicial 	Mensagem 	Mensagem final 	Uma mensagem inicial desencadeia o início de um processo, ou continua um determinado processo no caso de um evento intermediário. Uma mensagem final denota uma mensagem gerada no final de um processo.
Timer inicial 	Timer 	Um timer não pode ser um evento final	Um tempo específico ou um ciclo (toda segunda-feira às 9 h por exemplo) pode ser configurado para desencadear o início de um processo ou continuar um outro, no caso de um evento intermediário.
Regra inicial 	Regra 	Uma regra não pode ser um evento final	Dispara um processo quando as condições de uma determinada regra se tornaram verdadeiras.
Link inicial 	Link 	Link final 	Um link é um mecanismo utilizado para conectar o evento final de um processo a um evento inicial de outro processo.
Início múltiplo 	Múltiplo 	Múltiplo final 	Para múltiplos eventos de início, há várias maneiras de disparar um processo ou continuar um determinado processo no caso de um evento intermediário. Apenas um deles é necessário. Para um final múltiplo, há diversas consequências ao término de um processo (todas ocorrerão).
Uma exception não pode ser um evento inicial	Exception 	Exception final 	A ocorrência de uma exception final informa o processo que um erro deve ser gerado. Este erro será tratado por uma exception intermediária.
Uma compensação não pode ser um evento inicial	Compensação 	Compensação final 	Um evento de compensação final informa o processo que uma compensação é necessária.
Um evento final não pode iniciar um processo	Um evento final não pode ser um evento intermediário.	Cancelamento final 	Um evento final significa que o usuário decidiu cancelar a execução do processo.
Um evento fatal não pode ser um evento inicial	Um evento fatal não pode ser um evento intermediário	Final fatal 	Este evento significa que ocorreu um erro fatal e que todas as atividades do processo devem ser finalizadas imediatamente, sem qualquer tratamento ou compensação.

3 ARQUITETURAS DE HARDWARE E SOFTWARE

Este capítulo descreve as arquiteturas de hardware e de software definidas para o sistema. Em seguida, serão descritas as tecnologias envolvidas no seu desenvolvimento e na obtenção da infra-estrutura necessária para a sua execução.

3.1 Arquitetura de hardware do sistema

A Figura 4 ilustra a arquitetura de hardware final do sistema:

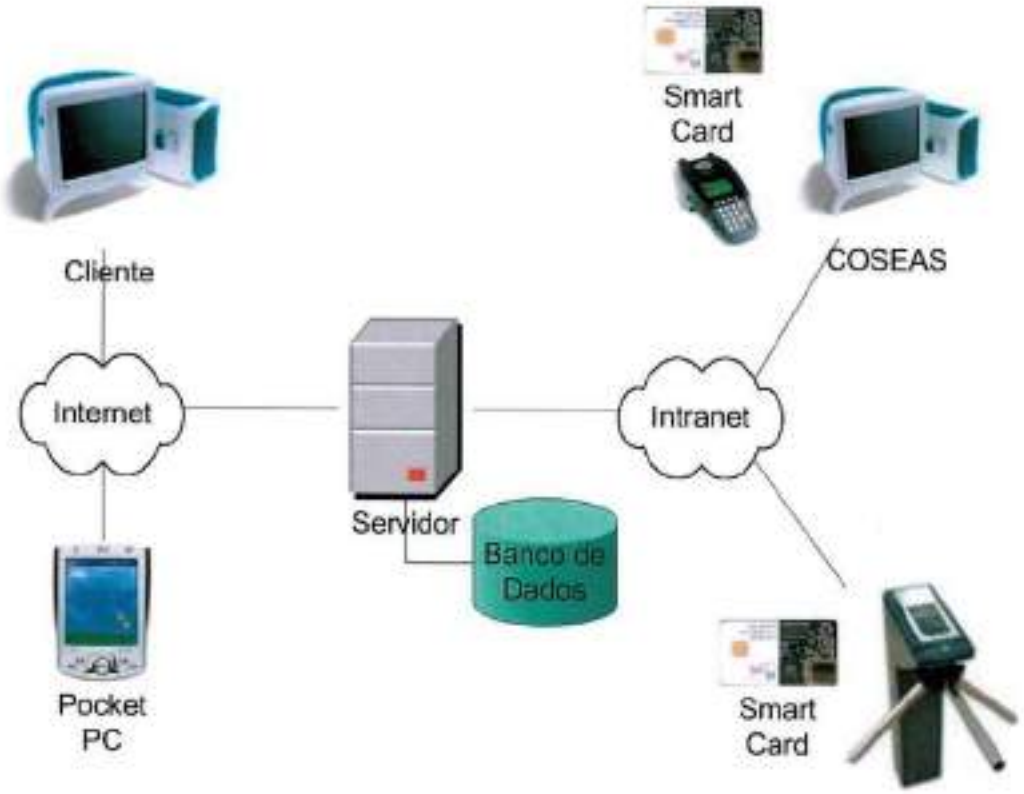


Figura 4- Arquitetura final de hardware do sistema

Nesta figura, o equipamento cliente permite o cliente do restaurante adquirir *tickets* refeição através da internet. O *pocket PC* permite fazer a atualização de informações de níveis estoque, manipulada por um funcionário do COSEAS. O servidor é encarregado de hospedar os módulos do sistema e o banco de dados associado é responsável por armazenar todas as informações necessárias ao

gerenciamento dos restaurantes, bem como alguns dados relativos aos clientes, incluindo os valores de saldo em seus cartões.

O processo de crédito nos *Smart Cards* é realizado por sua leitora. O cliente deve se dirigir a um guichê próprio do COSEAS (onde está localizado a leitora do cartão) e o atendente credita o valor pago no cartão, além de registrá-lo no sistema por meio de um portal web (intranet).

As catracas eletrônicas estão localizadas nas entradas dos restaurantes e são responsáveis pelo controle de acesso aos estabelecimentos. A leitura do cartão é realizada e, havendo crédito suficiente, a passagem é liberada para o cliente após a operação de débito.O saldo do cartão é atualizado por meio da comunicação da catraca com o sistema (intranet).

3.2 Arquitetura de software do sistema

A arquitetura de software do sistema é baseada no *design pattern Model-View-Controller* (MVC). Este *pattern* é implementado na arquitetura chamada *Model 2*, que é a utilizada no SARC.

Inicialmente é apresentado o *design pattern* MVC. Em seguida, é feita a descrição dos modelos de arquitetura *Model 1* e *Model 2*, bem como as suas vantagens de uso em um sistema.

3.2.1 Design Pattern MVC

O sistema, que usa o *design pattern* MVC (*Model View Controller*), é organizado em três componentes: o modelo de aplicação, com a representação de dados e a lógica de negócio, o da interface de usuários e o controlador das requisições e das respostas. Desta forma, uma modificação em um dos componentes pode ser feita sem necessitar alterar nenhum outro, tornando a estrutura mais adequada para alterações [5].

O MVC possui raízes antigas em publicações anteriores a 1973, quando a programação orientada a objetos ainda estava sendo concebida e os conceitos de componentes distribuídos ainda estavam em estudo. Foi formalmente definido em

1978 e tem o propósito de reduzir a complexidade de interfaces para sistemas robustos e complexos [4].

Este *design pattern*, é constituído pelos seguintes componentes:

- *Model*: é a representação das informações sobre a qual a aplicação irá operar e a lógica de negócio.
- *View*: é geralmente um elemento de interface, que renderiza o modelo em um formato pertinente à interação com o usuário.
- *Controller*: é o responsável pela resposta a eventos (tipicamente de usuários) e solicita alterações no *Model* ou *View* quando apropriado.

No padrão MVC, o fluxo da requisição é mediado por um *Controller* central. É ele quem envia as requisições – no caso, HTTP *requests* – ao *handler* apropriado. Os *handlers* estão associados ao *Model* e cada *handler* age como um adaptador entre as requisições e o *Model*. O *Model* representa, ou encapsula, a lógica de negócio ou o estado da aplicação, realizando a requisição recebida. O controle é então novamente repassado ao *Controller*, que repassa ao *View* apropriado. Esta transferência pode ser determinada a partir da consulta de um conjunto de *mappings*, geralmente carregados em um arquivo de configuração. Isto provê uma ligação fraca entre o *View* e o *Model*, o que torna as aplicações significativamente fáceis de se criar e manter.

A Figura a 5 ilustra o fluxo em um *design pattern* MVC:

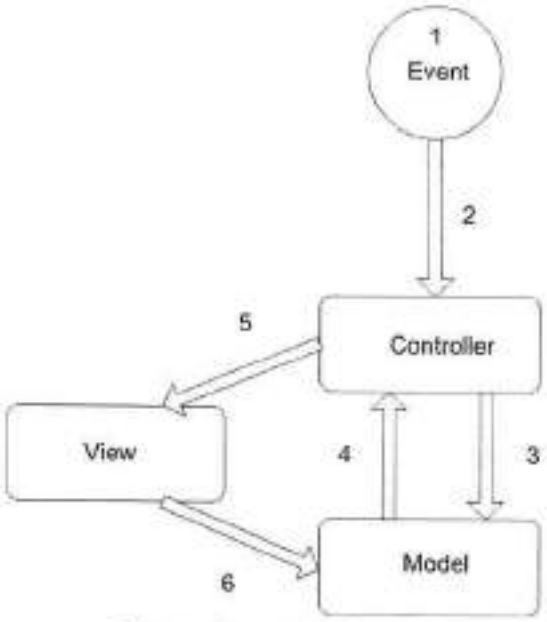


Figura 5 – Modelo MVC

1. Usuário interage com a interface (por exemplo, aperta um botão).
2. *Controller* recebe a notificação da ação.
3. *Controller* acessa o *Model*, que realiza as operações, de acordo com a ação do usuário.
4. *Model* notifica o *Controller* sobre suas operações.
5. *Controller* consulta o arquivo de configurações e repassa o controle ao *View* apropriado.
6. *View* acessa o *Model* para gerar a interface para o usuário.

A interface está, então, pronta para receber novas interações por parte do usuário.

O componente *Model* do *design pattern* do MVC ainda pode ser dividido em duas partes: uma que representa o estado interno e outra que realiza as ações que alteram este estado. Pode-se dizer que uma parte é o substantivo, que armazena o estado, e a outra é o verbo, que contém a lógica de negócio (algoritmos) e realiza as alterações neste estado, ou no substantivo.

No *design pattern* MVC ocorre uma dissociação dos componentes da aplicação, provendo uma estabilidade à lógica de negócio, apesar das recorrentes alterações na interface. Este fato é de suma importância no desenvolvimento e na manutenção de aplicações de grandes porte e complexidade.

3.2.2 Arquitetura Model 1 e Model 2

Em aplicações para *Web*, geralmente são utilizados dois tipos de modelos de arquitetura: *Model 1* e *Model 2*. O primeiro é mais simples e é recomendado apenas para aplicações de pequeno porte. Já o segundo é indicado para aplicações de médio e grande porte e é baseado no *design pattern* MVC [4].

Os modelos diferem entre si basicamente pela localização da carga de processamento de requisições, conforme pode ser visto nas Figuras 6 e 7.

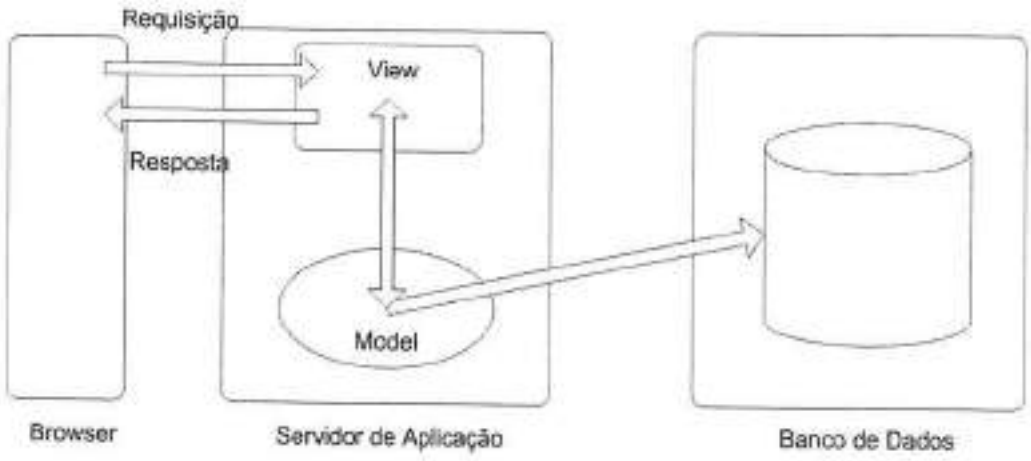


Figura 6 - Arquitetura Model 1

No *Model 1*, a interface (*View*) é responsável por processar sozinha as requisições e responder de volta ao cliente.

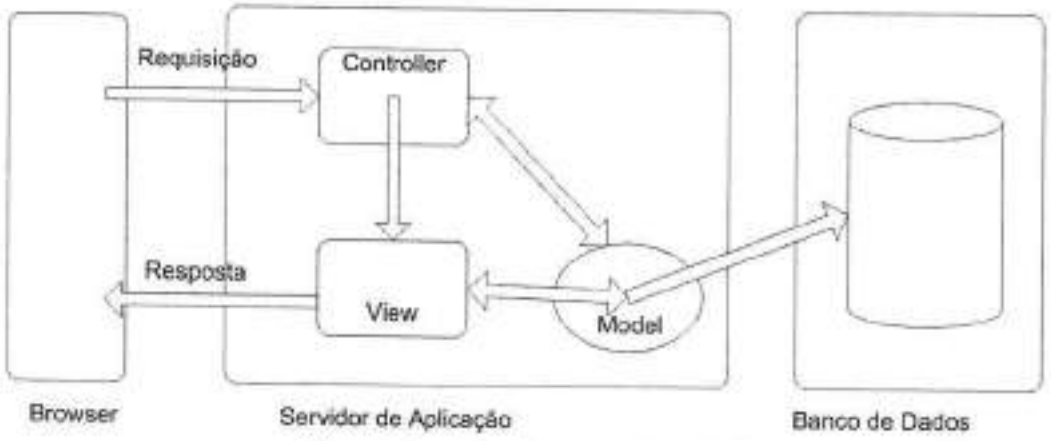


Figura 7 - Arquitetura Model 2

No *Model 2*, há uma divisão de tarefas: o *Controller* processa as requisições que chegam e decide qual *View* será encaminhada como resposta ao cliente, a qual é responsável apenas por devolver os objetos instanciados pelo *Controller* [4].

A aplicação SARC terá uma carga considerável de acessos, visto que atenderá a toda a comunidade USP, além de ser utilizado pelos funcionários COSEAS para o gerenciamento de estoques.

Portanto pode ser considerada uma aplicação de médio porte, tendo o *Model 2* como arquitetura mais indicada.

3.2.3 Conclusões

A principal vantagem do emprego do *design pattern* MVC é a separação entre a lógica de negócio do sistema, que geralmente se estabiliza antes da interface, que sofre recorrentes alterações para adequar-se aos padrões estéticos vigentes e otimizar a sua usabilidade.

Outro aspecto positivo dessa dissociação é que o tipo de codificação da interface permite que ela seja desenvolvida por *web designers*, que geralmente não possuem conhecimentos profundos de programação Java ou outra linguagem. Além disso, este *pattern* pode (deve) ser empregado em sistemas de grande porte, pois permite que o desenvolvimento do mesmo seja feito por diversos grupos, que podem atuar em paralelo, cada qual em uma camada do sistema.

4 TECNOLOGIAS ENVOLVIDAS

Esta seção faz um estudo sobre as tecnologias empregadas no SARC. Ela está dividida em três partes: a primeira trata de assuntos relacionados a *Smart Cards*; em seguida é descrita a plataforma J2EE (*Java 2 Plataform, Enterprise Edition*) e os *frameworks* empregados que possibilitaram a implementação da aplicação, além do modelo de arquitetura escolhido; por fim são abordados os recursos de software utilizados no ambiente de execução, necessários para o funcionamento do sistema.

4.1 Smart Cards

4.1.1 Introdução

Embora o dispositivo tenha sido inventado no final da década de 60 por dois engenheiros alemães, Jurgen Dethloff e Helmut Grottrupp, o termo *Smart Card* foi primeiramente utilizado pelo francês Roy Bright em1980. Eles solicitaram a patente em fevereiro de 1969, mas foi concedida apenas em 1982. O jornalista francês Roland Moreno arquivou 47 patentes relacionadas a Smart Cards em 11 países entre 1974 e 1979 [22].

Nos últimos anos, a capacidade de armazenamento e de processamento destes cartões tem avançado substancialmente. As áreas que mais se destacam no desenvolvimento de aplicações para *Smart Cards* são as áreas bancária, médica, de identificação pessoal e de telefonia celular. Conhecimentos a respeito de sistemas para *Smart Cards* estão cada vez mais presentes nos trabalhos de profissionais e pesquisadores especializados em computação, identificando novas áreas de atuação.

Um *Smart Card* tem o mesmo formato e tamanho de um cartão de crédito comum, mas pode armazenar tipicamente de 4KB a 64 KB de informação, além de realizar processamento de dados. Este tipo de cartão é especialmente útil em aplicações que requerem tanto a segurança quanto a integridade dos dados neles armazenados.

4.1.2 Tipos de Smart Cards

Atualmente os *Smart Cards* são classificados segundo três categorias principais [22]:

- *Memory Cards* – Cartões de memória (não realizam processamento dos dados) providos de um *chip* de circuito integrado. A comunicação com o leitor exige o contato físico.
- *Microprocessor Cards* – Idênticos ao anterior, mas possuem um microprocessador embutido que permite o processamento de dados.
- *Contactless Cards* – Estes cartões também possuem um microprocessador embutido, mas a comunicação é realizada de maneira eletromagnética, não necessitando do contato físico com o leitor.

4.1.2.1 Memory Cards

Este tipo de cartão foi o primeiro a ser desenvolvido. Eles possuem um *chip* de circuito integrado contendo apenas uma memória não volátil e o circuito necessário para a leitura e escrita nesta memória. São dependentes do computador ou do leitor para o processamento e correspondem à maioria dos *Smart Cards* utilizados atualmente.

Memory Cards não são excessivamente caros e fornecem um nível modesto de segurança. São adequados para aplicações que realizam operações determinadas, como subtração em cartões de telefones pré-pagos, por exemplo.

4.1.2.2 Microprocessor Cards

São também conhecidos como *chip cards*, uma vez que possuem um microprocessador embutido no corpo do cartão e podem processar dados. A atual geração deste tipo de cartão possui um poder de processamento equivalente ao computador IBM-XT, com menor capacidade de armazenamento. Como exemplos de aplicações, podem-se citar as que necessitam armazenar valores monetários e também as que permitem o acesso a determinadas redes de computadores.

4.1.2.3 Contactless Cards

Estes cartões se comunicam com o leitor através de sinais eletromagnéticos. A energia necessária para executar as operações do *chip* é fornecida por meio de microondas de freqüência determinada pelo leitor. Oferecem maior facilidade de uso em certas aplicações nas quais a posse do cartão é suficiente para o seu uso, como por exemplo a identificação em certos ambientes.

4.1.3 Elementos de Smart Cards

A CPU (Unidade Central de Processamento), memória e trilhas de entrada e saída são montadas em um único chip de circuito integrado. A Figura 8 ilustra a organização destes componentes no cartão [22].

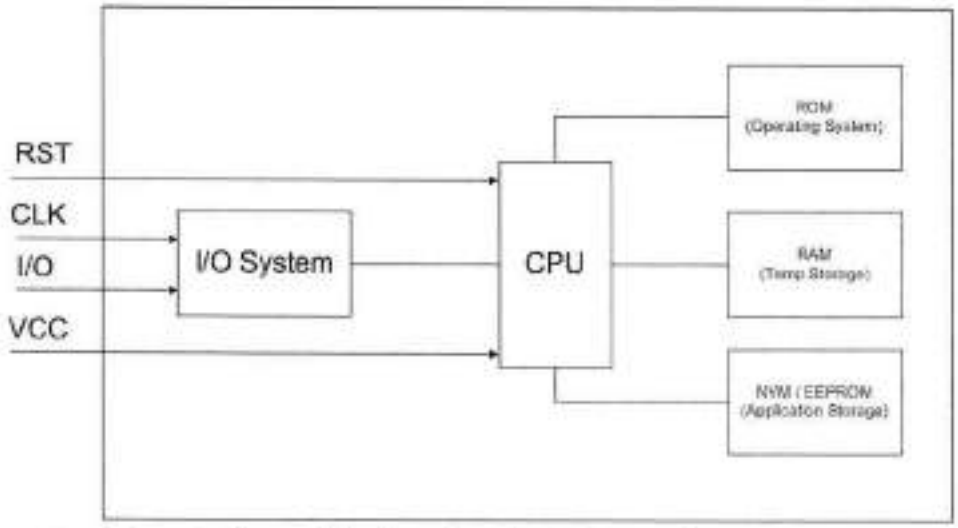


Figura 8 - Layout de um chip de circuito integrado em um *Smart Card*

4.1.3.1 A Unidade de Processamento Central

A CPU de um *Smart Card* consiste de um microcontrolador de 8 bits que tipicamente utiliza o conjunto de instruções do Motorola 6805 ou do Intel 8051. Executam instruções de máquina na velocidade de aproximadamente 400.000 instruções por segundo. Os *chips* mais atuais podem executar mais de 1 milhão de

instruções por segundo (1 MIP). O tempo que um *Smart Card* leva para realizar uma transação é de 1 a 3 segundos. Alguns *chips* incluem um co-processador para acelerar a codificação de dados [22].

4.1.3.2 O Sistema de Memória em Smart Cards

Os *Smart Cards* possuem uma quantidade relativamente pequena de memória RAM (cerca de 256 a 1.000 bytes). Eles também contêm uma memória de apenas leitura (ROM) e uma não-volátil (EEPROM).

Os dados armazenados na memória RAM não são preservados quando a energia é cortada. Entretanto, o seu papel é essencial em algumas aplicações, visto que o tempo que a CPU gasta para ler ou escrever neste tipo de memória é relativamente muito menor.

Os *Smart Cards* de propósito geral contém entre 8KB e 32 KB de memória ROM. É nela que o sistema operacional e as rotinas específicas de comunicação e aritméticas são armazenados. As informações inseridas neste tipo de memória não podem ser modificadas.

Os dados variáveis são armazenados na memória não-volátil do cartão (EEPROM), cujo conteúdo pode ser modificado conforme as necessidades da aplicação. Esta memória pode ser lida e escrita e possibilita a retenção do seu conteúdo mesmo quando a energia é retirada. Os *Smart Cards* convencionais possuem geralmente entre 1 KB e 16 KB de EEPROM. Os dados nela armazenados podem ser preservados por cerca de 10 anos e o tempo de escrita leva cerca da 3 a 10 milisegundos [22].

4.1.3.3 Input/Output em Smart Cards

A comunicação com o mundo externo é feita por meio de uma via simples que fica aos cuidados do processador, que utiliza protocolos de comunicação para filtrar informações que são passadas para e de outros componentes do *chip*. Estes protocolos também podem ser usados para autenticação, cujos detalhes serão descritos posteriormente.

4.1.3.4 Dispositivos de interface

O *Smart Card* não contém qualquer fonte de energia ou sinal de relógio independente, que são necessários para o processador inserido no cartão. Desta maneira, o cartão deve ser conectado a um dispositivo que forneça estes itens. Este dispositivo é conhecido como dispositivo de interface, terminal ou leitor.

Além destas funcionalidades, o leitor é responsável por estabelecer um canal de comunicação entre a aplicação residente no computador e o sistema operacional do cartão. Atualmente quase todos os leitores permitem tanto a leitura quanto a escrita no cartão.

4.1.3.5 O Sistema Operacional

O sistema operacional encontrado na maioria dos cartões implementa um conjunto padrão de comandos (usualmente na ordem de 20 a 30), aos quais o *Smart Card* responde. O padrão mais comum é o ISO 7816, que descreve uma faixa de comandos que podem ser implementados. Alguns fabricantes podem possibilitar extensões deste padrão ou até mesmo adições. A relação entre a leitora e o cartão é do tipo mestre/escravo: o leitor envia um comando ao cartão e este o executa, retornando um resultado correspondente, se houver [22].

A maior parte dos sistemas operacionais dos *Smart Cards* suportam um sistema de arquivos baseados no padrão ISO 7816. Um arquivo é tratado como um bloco contínuo na memória do cartão. Uma vez que um arquivo é alocado, ele não pode ser estendido, o que significa que ele deve ser criado com o seu tamanho máximo esperado. O sistema de arquivos não suporta a função de *garbage collection* ou qualquer tipo de compactação. Se um arquivo A, seguido por um arquivo B, for apagado, o espaço ocupado pelo primeiro é perdido até que o segundo também seja apagado.

Os sistemas operacionais suportam o conjunto usual de operações relacionadas a arquivos, tais como *create*, *delete*, *read*, *write* e *update*. Outras operações não usuais são suportadas em tipos particulares de arquivos. Arquivos lineares, por exemplo, consiste de uma série de registros de tamanho fixo que podem ser

acessados pelo número do registro ou lidos seqüencialmente usando operações como *read next* e *read previous* [22].

4.1.3.6 Estrutura de diretórios no Smart Card

A maioria dos *Smart Cards* possui um sistema de arquivos baseado na estrutura de diretórios do UNIX. Trata-se de um sistema de arquivos hierárquico cuja raiz, chamada de *Master File*, possui o endereço 3f.00. Os nomes de arquivos possuem tamanho de 2 bytes e podem ser divididos em quatro categorias:

- Arquivos transparentes, que são vistos como uma seqüência de bytes;
- Arquivos linearmente fixos, que são vistos como uma seqüência de registros de tamanho fixo;
- Arquivos linearmente variáveis, que são vistos como uma seqüência de registros de tamanho variável;
- Arquivos cíclicos, que são vistos como uma seqüência infinita de registros de tamanho fixo.

Um exemplo da estrutura de diretórios de um Smart Card pode ser visto na Figura 9.

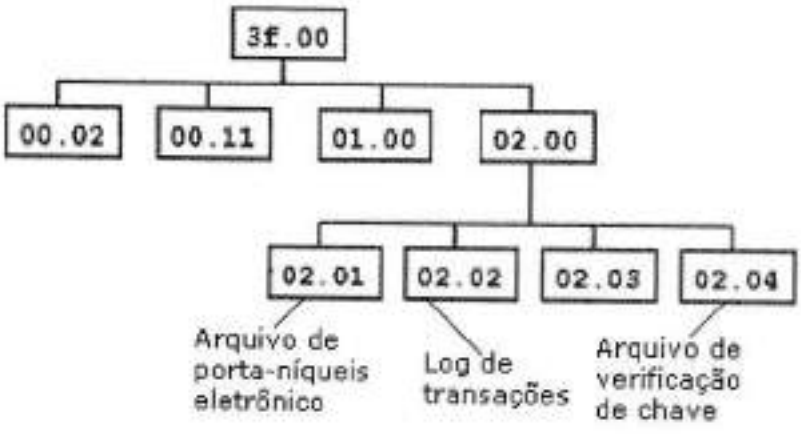


Figura 9 - Estrutura de diretórios em um Smart Card

4.1.3.7 Application Protocol Data Units (APDU´s)

Smart Cards se comunicam com dispositivos de interface por meio de pacotes de dados que são construídos seguindo um protocolo definido. Estes pacotes são chamados APDU´s (*Application Protocol Data Units*), que podem representar um comando ou uma resposta. Um *Smart Card* recebe uma APDU do terminal e executa a ação especificada nele (em um campo do seu protocolo), retornando uma respectiva APDU de resposta.

A Tabela 3 ilustra o formato da APDU de comando, especificada na quarta parte do padrão ISO 7816 [22].

Tabela 3 - APDU de comando

APDU de Comando						
Cabeçalho				Parte Condicional		
CLA	INS	P1	P2	Lc	Campo de Dados	Le

O cabeçalho especifica o comando a ser executado pelo Smart Card. Ele é composto de quatro campos, cada um formado por um byte:

- CLA (Class byte): Identifica a aplicação;
- INS (Instruction byte): Identifica o código da instrução;
- P1 e P2 (Parameter bytes): Carregam informações extras. Geralmente especificam o endereçamento utilizado pela instrução definida pelos campos CLA e INS;
- Lc: Número de bytes do campo de dados;
- Le: Número máximo de bytes esperado no campo de dados da APDU de resposta correspondente;

A Tabela 4 ilustra o formato da APDU de resposta, também especificada na quarta parte do padrão ISO 7816 [22].

Tabela 4 - APDU de resposta

APDU de resposta		
Parte Condicional	Bytes de Status	
Campo de Dados	SW1	SW2

SW1 e SW2 (*Status bytes*): Informam o status do processamento da APDU de comando enviada ao cartão. Um deles é utilizado para especificar uma categoria de erro e o outro é utilizado para especificar o *status* de um comando ou indicar a ocorrência de um erro.

Todos os detalhes com relação ao protocolo de comunicação do Smart Card com a leitora e o padrão ISO 7816 podem ser encontrados em [22].

4.1.3.8 Comunicação

O canal de comunicação do *Smart Card* é do tipo *half-duplex*. Isto significa que um dado pode fluir da leitora para o cartão ou vice-versa, mas não em ambas as direções ao mesmo tempo. A leitora amostra o sinal da linha serial na mesma taxa com que o transmissor envia os dados, a fim de se obter uma correta recepção. Esta taxa é conhecida como *bit rate* ou *baud rate*. Os dados recebidos e transmitidos pelo cartão são armazenados em um *buffer* na mamória RAM do cartão. Conseqüentemente, pacotes de dados relativamente pequenos (10 a 100 bytes) são movidos em cada mensagem [22].

4.1.3.8.1 Software de Smart Cards

Os *softwares* de *Smart Cards* podem ser divididos em dois tipos: *Card Software* e *Host Software*. O primeiro é conhecido também como *card-side software* e, como o próprio nome sugere, é executado no próprio cartão. Basicamente, ele provê serviços computacionais para as aplicações que acessam os dados contidos no

cartão. Além disso, é capaz de protegê-los de aplicações que, por algum motivo, possam vir a acessá-los incorretamente, contribuindo para a manutenção da integridade dos dados e de propriedades de segurança em cartões particulares.

O segundo tipo de software, o *Host Software*, é executado no computador que é conectado à leitora do cartão. É conhecido como *reader-side software* e inclui aplicativos de usuário final. Ao contrário do primeiro, que é escrito em linguagem *assembly*, *host softwares* são geralmente escritos em uma linguagem de programação de alto nível, como C, C++, Java, etc. Além disso, geralmente se relacionam com bibliotecas comerciais e *drivers* para a comunicação de cartões e leitores [22].

4.1.3.8.2 Segurança em Smart Cards

Uma das principais razões da existência de *Smart Cards* está relacionado com a segurança. O cartão provê uma plataforma computacional onde transações e informações podem ser tratadas de maneira segura. A partir destas características, os *Smart Cards* são adequados para reforçar a segurança em diversos sistemas. Os principais exemplos destas aplicações incluem sistemas de acesso físico a ambientes e sistemas financeiros, incluindo aplicações de comércio eletrônico. Neste projeto, a utilização de Smart Cards visa justamente atender requisitos de segurança, tanto no que diz respeito ao controle do acesso de clientes aos restaurantes COSEAS, quanto em operações de crédito e débito.

O acesso à informação armazenada no *Smart Card* pode ser configurado para que seja controlado estritamente pelo portador do cartão, o seu emissor ou o provedor de qualquer aplicação específica no cartão. O controle do acesso é geralmente implementado por meio de uma requisição de uma chave, a fim de se obter o acesso a certos arquivos. Estas chaves são armazenadas em arquivos específicos no cartão e apenas o *Smart Card* pode acessá-los para comparar com a chave obtida do leitor ou usuário.

Se o processo de autenticação não for realizado com sucesso, a comunicação é bloqueada. Um registro da quantidade de tentativas frustradas pode ser armazenado no cartão e, uma vez que um certo número de falhas consecutivas for detectado, o cartão pode destruir completamente o seu conteúdo.

Com relação à criptografia, alguns cartões realizam este processo em grande volume, o que exige um maior poder computacional. Em particular, existem alguns cartões que são fabricados para este propósito e são equipados com um processador dedicado exclusivamente a esta tarefa (conhecidos como *crytoprocessors*). A criptografia pode ser aplicada a todas as mensagens para o cartão e provenientes dele, ou alternativamente para apenas algumas mensagens particulares. Geralmente os programadores de *Smart Cards* não necessitam projetar algoritmos de autenticação ou de criptografia para as suas aplicações. Ao invés disso, eles usam as facilidades que são embutidas no próprio cartão, que já garantem um certo nível de segurança [22].

4.1.3.8.3 Equipamentos e ferramentas utilizadas

Para o desenvolvimento do projeto, foi utilizados a leitora Schlumberger Reflex 72 e o cartão de memória Schlumberger Payflex de 1K. A leitora possui interface serial RS-232 e um cabo auxiliar para a alimentação do tipo PS2.

A comunicação com o *Smart Card* será feita por meio de um software próprio, sob licença GPL (*General Public License*), que fará uso de APDUs capazes de selecionar arquivos, ler registros e receber respostas correspondentes. Os comandos são transmitidos à leitora por meio da interface serial (não será necessário nenhum *driver*) e estes são, por sua vez, encaminhados ao *Smart Card*.

O desenvolvimento da aplicação foi feito na plataforma Linux (*kernel* 2.4.29), mas pode facilmente ser alterada para ser executada em plataforma Windows. Não foi utilizado nenhum kit de desenvolvimento de aplicações para *Smart Cards*.

4.2 Plataforma J2EE e frameworks

Nesta seção são descritos os principais recursos utilizados neste trabalho, que são os seguintes:

- Padrão J2EE (*Java 2 Plataform, Enterprise Edition*) é descrito, bem como os componentes definidos por esta especificação e que são utilizados na aplicação [6];

- *JavaBeans* que são definidos na especificação do J2SE, já que são componentes essenciais da camada de negócio do sistema [8];
- *Framework* Struts, que faz uso de componentes definidos no padrão J2EE para permitir a implementação do modelo MVC em Java [17];
- *Framework* Hibernate, que é responsável pela persistência dos dados do sistema no banco de dados [9].

Este conjunto de padrões e *frameworks* formam a base para o desenvolvimento de aplicações de grande porte e são extremamente úteis, pois tornam este processo muito mais simples e rápido, sendo responsáveis pela implementação de tarefas de mais baixo nível. O programador pode concentrar-se apenas na lógica de negócio, que é o núcleo principal da aplicação.

4.2.1 J2EE - Java 2 Plataform, Entreprise Edition

A plataforma J2EE (*Java 2 Plataform, Enterprise Edition*) é uma colaboração entre líderes da área de softwares corporativos, incluindo fornecedores de sistemas operacionais e sistemas de gerenciamento de bancos de dados, vendedores de *middleware* e ferramentas e desenvolvedores de componentes e aplicações verticais de mercado.

A J2EE é uma plataforma de desenvolvimento para aplicações com arquiteturas de múltiplas camadas, definido por uma especificação e é considerado informalmente um padrão.

A plataforma J2EE simplifica a arquitetura de aplicações de negócio, baseando-as em componentes modularizados e padronizados, além de prover um conjunto completo de serviços para estes componentes, o que diminui a complexidade da programação. Além disso, permite a integração com diversas tecnologias legadas.

A plataforma J2EE utiliza vários atributos do J2SE (*Java 2 Plataform, Standard Edition*) como a portabilidade, API JDBC para acesso ao banco de dados, tecnologia CORBA para a interação com recursos já existentes e um modelo de segurança que protege os dados até mesmo em aplicações na internet. Ainda possui

suporte total aos componentes *Enterprise JavaBeans*, *Java Servlet API*, *JavaServer Pages* e a tecnologia XML [2].

A plataforma J2EE possui testes de compilação e especificação que garantem a portabilidade das aplicações através da diversidade de sistemas capazes de suportar a plataforma J2EE e ainda assegura serviços *Web* de interoperabilidade por meio do suporte ao *WS-I Basic Profile*.

A portabilidade e a escalabilidade são palavras-chave na análise de viabilidade de uma aplicação e estas características são inerentes à plataforma J2EE. Além disso, esta plataforma engloba fontes já existentes em aplicações multi-camada em um modelo de aplicação baseado em componentes. A plataforma J2EE abrange diversos sistemas, dentre eles os de administração de banco de dados, monitores de transação etc.

A plataforma J2EE é composta de quatro grandes elementos:

- *J2EE Application Programming Model*: é o modelo padrão de programação usado para facilitar o desenvolvimento de aplicações multi-camada.
- *J2EE Plataform*: inclui políticas necessárias e APIs como os *Java Servlets* e *Java Message Service* (JMS).
- *J2EE Compatibility Test Suite*: garante que os produtos J2EE são compatíveis com os padrões da plataforma.
- *J2EE Reference Implementation*: explica as capacidades e provê definições operacionais.

Atualmente, com a inclusão do *WS-I Basic Profile*, é possível construir aplicações na plataforma J2EE, como *Web Services*, capazes de agir em conjunto com *Web Services* em ambientes não J2EE.

A plataforma J2EE inclui especificações únicas de componentes como *Enterprise JavaBeans* (EJB), *Servlets* e *Java Server Pages* (JSP), entre outros. Os componentes J2EE são escritos em linguagem Java e compilados da mesma forma que qualquer programa nessa linguagem. A diferença entre os componentes J2EE e classes Java padrão é que os componentes J2EE são incorporados em uma aplicação J2EE e passam por uma verificação de sua formação e adequação à especificação J2EE. Estes componentes são montados em uma aplicação J2EE sendo executados e gerenciados por um servidor J2EE.

4.2.1.1 Java Servlet

Os *servlets* representam os programas Java que são executados no servidor *web*. Um *servlet* é uma classe em Java que geralmente estende as capacidades de servidores que hospedam o acesso a aplicações via um modelo de requisição-resposta.

Com os *servlets* é possível gerar *sites* dinâmicos em Java e sua principal tarefa é receber uma requisição e gerar a resposta, baseada na mesma.

A *Servlet* API define as interações entre um *servlet* e o *container web*, que é responsável por mapear uma determinada URL para o *servlet* correspondente e verificar permissões de acesso.

A API também define subclasses HTTP de requisições genéricas e suas respostas, bem como um objeto HTTP *session*, que rastreia múltiplas requisições e respostas entre o *web server* e o cliente.

Os *servlets* devem ser empacotados como uma *Web Application* e todos os *servlets* devem implementar a interface *servlet*, que define os métodos do ciclo de vida: inicialização do *servlet*, serviços de requisição e remoção do *servlet* do servidor:

- O *servlet* é construído e inicializado com o método *init*.
- Qualquer chamada de clientes ao método *service* são tratadas.
- O *servlet* pára de trabalhar e é então destruído com o método *destroy*. Em seguida, é recolhido pelo *garbage collector* e finalizado.

A *Interface Servlet* provê os métodos *getServletConfig*, que fornece qualquer informação de inicialização do *servlet*, e *getServletInfo*, que permite que o *servlet* forneça qualquer informação básica sobre si mesmo.

Os pacotes *javax.servlet* e *javax.servlet.http* fornecem interfaces e classes para a programação de *servlets*.

Para a implementação de um serviço genérico, independente de protocolo, a classe *GenericServlet*, que é fornecida na API *Java Servlet*, pode ser usada ou extendida.

Para a criação de um *servlet* HTTP, a classe *HttpServlet* deve ser utilizada. Ela provê métodos como *doGet* e *doPost*, para lidar com serviços específicos HTTP.

4.2.1.2 Java Server Pages (JSP)

É a tecnologia Java que permite a geração dinâmica de HTML, XML ou qualquer outro tipo de página *web*. Permite a inserção de fragmentos do código e de ações pré-definidas em um documento de texto puro.

Uma página JSP possui dois tipos de texto:

- Dados estáticos, que podem ser expressos em qualquer formato baseado em texto, como HTML, XML etc.
- Elementos JSP, que determinam como o conteúdo dinâmico da página deve ser construído, chamando funcionalidades previamente definidas.

Esta tecnologia ainda permite a criação de uma biblioteca de *tags* JSP, que agem como extensões das *tags* padrão HTML ou XML.

JSPs são compiladas em *servlets* por meio de um compilador JSP. Este compilador pode tanto gerar um código do *servlet* em Java que é, então, compilado em um compilador Java, ou gerar diretamente o *byte code* do *servlet*. Em ambos os casos, é o compilador JSP que transforma a página em um *servlet* Java.

De maneira geral, o JSP pode ser considerado como a abstração de alto nível dos *servlets*.

4.2.2 JavaBeans

É um objeto que se adequa aos protocolos de comunicação e configuração definidos pela especificação *JavaBeans*. É um componente integrante da arquitetura da *Java 2 Plataform, Standard Edition* (J2SE) e são programas/componentes reutilizáveis.

Certas convenções devem ser respeitadas a fim de que a classe seja um *JavaBean*. São elas:

A classe deve ser capaz de ser persistida e de restaurar seu estado (serializáveis).

- Deve ter um construtor sem argumentos.

- Suas propriedades devem estar acessíveis somente através dos métodos *get* e *set*, que seguem o padrão de nomenclatura.
- Deve conter os métodos add/*remove* para eventos.

A API completa do *JavaBeans* está empacotada no *java.beans*, uma das APIs centrais Java.

JavaBeans são objetos simples em Java, ou POJOs (*Plain Old Java Objects*), que seguem padrões rígidos de nomenclatura e são serializáveis. Não devem ser confundidos com *Enterprise JavaBeans*, que representam um modelo completo de um componente, não uma única classe apenas. A complexidade dos EJBs é alta e eles não são necessários em todas as aplicações.

4.2.3 Struts

É um *framework open-source* para desenvolvimento de aplicações J2EE, mantido pelo projeto *Jakarta*, que está sob a administração da *Apache Software Foundation*.

Este *framework* permite que o projeto e implementação de aplicações *web* de grande porte seja feito por diferentes grupos de pessoas (*designers* de páginas, desenvolvedores de componentes etc).

Pode-se dizer que o Struts é a implementação em Java do *design pattern* MVC e, sendo assim, resulta em:

- *Model*: representado por *JavaBeans*
- *View*: representado por JSPs
- *Controller*: representado por *Servlets*

4.2.3.1 Model

Aplicações de grande porte geralmente representam um conjunto de operações de negócio possíveis como métodos que podem ser acionados por *bean* ou *beans* que armazenam o estado da informação. Outros sistemas representam as operações disponíveis separadamente, com o uso, por exemplo, de *Session EJBs*.

Em aplicações de menor porte, por outro lado, as operações disponíveis podem estar inseridas em classes *Action*, que fazem parte da camada de controle do Struts. Isto é muito útil quando a lógica é bastante simples ou a reutilização da lógica de negócio em outros ambientes não é contemplada.

O estado real de um sistema é geralmente representado como um conjunto de uma ou mais classes *JavaBeans* (*System State Beans*), cujas propriedades definem o estado atual. *Business Logic Beans* são *JavaBeans* que tem como função encapsular a lógica funcional da aplicação em métodos. Estes métodos podem fazer parte da mesma classe do *System State Beans* ou podem estar em classes separadas.

No sistema SARC ocorre esta divisão de *Business Logic Beans* e *System State Beans*. Sendo assim, os *System State Beans* são passados como parâmetros nas chamadas de métodos dos *Business Logic Beans*.

As classes *Business Logic Beans* não devem ter o conhecimento de que elas estão sendo executadas em um ambiente de aplicação *web* (não devem importar *javax.servlet.**, por exemplo). As classes *Action* (que fazem parte da camada *Controller*) devem traduzir todas as requisições HTTP em chamadas de métodos como *get* ou *set*. Isto permite que a classe da lógica de negócio possa ser reutilizada em ambientes diferentes de uma aplicação *web*, para a qual foi inicialmente criada.

É a classe *Business Logic Bean* que fará a chamada da classe de acesso ao Banco de Dados.

4.2.3.2 View

A camada de *View* do Struts é baseada na tecnologia *JavaServer Pages* (JSP). Um conjunto de *tags* personalizadas, que são permitidas em JSP, estão inclusas no Struts. Isto facilita a criação de interfaces com o usuário, visto que estas *tags* interagem perfeitamente com os *ActionForm Beans*. *ActionForms* recebem e validam qualquer entrada que é requisitada pela aplicação.

O Struts permite o desenvolvimento de aplicações internacionalizadas e localizadas (adaptadas de acordo com opções regionais e de idioma). As classes padrão que tornam isso possível são:

- *Locale*: a classe Java fundamental que permite a internacionalização. Cada *Locale* representa uma opção de país e idioma (mais um idioma variante opcional) e um conjunto de suposições de padrão para números, datas etc.
- *ResourceBundle*: A classe *java.util.ResourceBundle* provê as ferramentas básicas para o suporte à múltiplas línguas.
- *PropertyResourceBundle*: uma das implementações padrão da classe *ResourceBundle*.
- *MessageFormat*: a classe *java.text.MessageFormat* permite que partes do *string* da mensagem sejam substituídas, baseados em argumentos definidos no momento da execução. Isto é útil em casos que as palavras aparecem em uma ordem diferente em idiomas distintos.
- *MessageResources*: A classe *org.apache.struts.util.MessageResources* permite a requisição de uma *string* em particular para determinado *Locale* (associado ao usuário).

O Struts fornece uma maneira simples de construir *forms*, baseado em *Custom Tag Library* do JSP 1.1. Não há necessidade de referenciar o *JavaBean* que contém os valores iniciais a serem retornados. Isto é feito automaticamente pela *tag* JSP, que faz uso das vantagens deste *framework*.

O Struts define *tags* para todos os tipos de campos de entrada listado abaixo:

- *Checkboxes*
- Campos ocultos
- Campos para inserção de senha
- Botões do tipo *radio*
- Botões do tipo *reset*
- Listas de selção
- *option*
- Botões do tipo *submit*
- Campos de entrada de texto
- Caixas de texto

Nestes casos, a *tag* do campo deve estar aninhada em uma *tag form*, para que o campo saiba qual *bean* deve ser usado para a exibição de valores iniciais. Podem-se ainda ter outras *tags* de exibição:

- [*logic*] *iterate*: repete o corpo da *tag* uma vez para cada elemento de uma coleção especificada (que pode ser um *Hashtable*, *Enumeration*, *Vector* ou um *array* de objetos).
- [*logic*] *present*: depende do atributo especificado, que podem ser: *cookie*, *header*, *name*, *scope*, *user* etc.
- [html] *link*: gera um elemento HTML do tipo <a> e automaticamente mantém a sessão para a URL especificada.
- [*bean*] *parameter*: busca o valor especificado no parâmetro de requisição e mostra-o como uma *String* na página.

Além da interação do *bean* e do *form* já descrita, o Struts oferece mais uma facilidade para a validação de campos de entrada de dados.

O método *validate* é chamado pelo *servlet* de controle depois que as propriedades do *bean* tenham sido populadas, mas antes do método execute da classe *action* correspondente.

Se o método *validade* encontrar problemas, uma instância de *ActionErrors* é retornada, contendo a mensagem de erro (dentro do domínio da aplicação definido por *MessageResources*) que deve ser exibida. O *servlet* de controle armazena este *array* para ele ser utilizado na tag *<html:errors>*. Caso contrário, ou seja, se nenhum erro de validação foi encontrado, o método retorna *null* e o *servlet* de controle irá invocar o método *perform* da classe *Action* pertinente.

Esta validação é opcional, mas foi implementada na aplicação por agilizar o processo de validação de dados na entrada. O método *validate* faz apenas uma primeira validação dos dados mais superficial. A lógica de negócio fica responsável por validações mais complexas.

4.2.3.3 Controller

É a parte da aplicação responsável por receber as requisições do cliente, decidindo qual função da lógica de negócio deve ser acionada (baseado na requisição

recebida), resgatar os dados da camada *Model* (se necessário) para enviá-los ao *View* e selecionar o *View* adequado para responder ao usuário.

No Struts, o componente básico do *Controller* é um *servlet* da classe *ActionServlet*. Este *servlet* é configurado a partir de um conjunto de *ActionMappings*. Um *ActionMapping* define o caminho entre a URI requisitada e a classe *Action* correspondente. Todas as *Actions* são subclasses de *org.apache.struts.action.Action*. *Actions* encapsulam as chamadas das classes da lógica de negócio, interpretam sua saída e delegam o controle para o componente do *View* apropriado para gerar a resposta.

A instância do *ActionServlet* é também responsável pela inicialização dos recursos. Quando o *Controller* é inicializado, ele primeiro carrega o *config* da aplicação que contém todos os parâmetros de inicialização.

Para cada requisição feita ao *Controller*, o método *process* será invocado. Este método apenas determina qual módulo deve servir à requisição e, então, chama o método *process* do *RequestProcessor* do módulo.

O *RequestProcessor* é onde ocorre a maior carga de processamento das requisições. O método *process* invoca:

- *ProcessPath*: determina o caminho requistado.
- *ProcessLocale*: seleciona o *locale* da requisição, se nenhuma tiver sido selecionada ainda.
- *Processmapping*: seleciona o *ActionMapping* associado a este caminho.
- *ProcessActionForm*: instancia (se necessário) um *ActionForm* associado a este mapeamento e coloca-o no escopo apropriado.
- *ProcessPopulate*: popula o *ActionForm* associado a requisição, se houver.
- *ProcessValidate*: realiza a validação do *ActionForm* associado a essa requisição.
- *ProcessForward*: se este mapeamento representa um *foward*, encaminha para o caminho especificado pelo mapeamento.
- *ProcessActionPerform*: É o ponto onde o método *perform* ou execute da *action* é invocado.

ActionForm beans são componentes da camada *Controller* que realizam o transporte das informações entre a camada *Model* e *View*. Possuem propriedades que correspondem às propriedades do *bean* da camada *Model*.

Um *ActionForm bean* é uma classe em Java que estende a classe *ActionForm* e é responsável pelos *forms* de entrada da aplicação. Ele possui os métodos *get/set*, mas nenhuma lógica de negócio.

Um *ActionForm* é um *JavaBean* associado com um ou mais *ActionMappings*. Ele tem suas propriedades inicializadas a partir dos parâmetros da requisição antes que o método execute da *Action* seja chamado.

Quando as propriedades deste *bean* forem populadas, mas antes da chamada do método *execute* da *Action*, o método *validate* do *bean* é executado, verificando se os valores submetidos pelo usuário são válidos. Caso haja algum problema na validação, uma mensagem de erro é retornada e o *servlet* de controle irá tratar este erro. Caso contrário, o método *validate* retornará *null*, indicando que tudo está correto e o método *execute* da *Action* deve ser chamado.

Pode existir apenas um *bean* para cada *form* (*finely-grained objects*) ou um *bean* que age em diversos *forms* ou até mesmo toda a aplicação (*coarsely-grained objects*).

Beans devem ser declarados no arquivo de configuração do Struts para que o *servlet* de controle faça as seguintes operações automaticamente antes de invocar o método apropriado da Action:

- Procurar por uma instância do *bean* da classe apropriada, com escopo apropriado.
- Se tal instância do *bean* não estiver disponível, uma nova é automaticamente criada e adicionada ao escopo.
- Para cada parâmetro da requisição cujo nome corresponde ao nome *property* do *bean*, o método *set* correspondente será chamado.
- O *ActionForm bean* atualizado será enviado ao método execute da *Action*, fazendo com que os valores nele contidos se tornem disponíveis aos *beans* que armazenam o estado do sistema e os de lógica de negócio.

É interessante que seja realizada a validação do *ActionForm bean*, visto que eles constituem o *firewall* entre o HTTP e o *Action*. O método *validate* deve ser

chamado, o que impede que *beans* que falharam na sua validação sejam enviados à *Action* correspondente. Uma *Action* é um adaptador entre o conteúdo de uma requisição HTTP que chega e a lógica de negócio correspondente que deve ser executada para processar essa requisição. O *Controller* selecionará a *Action* apropriada para cada requisição, irá criar uma instância dela (se necessário) e invocará o método *execute*.

As *Actions* devem ser programadas de maneira *thread-safe*, pois o *Controller* irá compartilhar a mesma instância para múltiplas requisições que poderão ocorrer simultaneamente.

Quando uma instância da *Action* é criada, o *Controller* irá chamar o método *setServlet* com um argumento não nulo para identificar a instância do *servlet* que será vinculada à *Action*. Quando o *servlet* é fechado (ou reiniciado), o método *setServlet* será invocado com um argumento *null*.

O método *execute* da *Action* processa a requisição e retorna um objeto do tipo *ActionFoward*, que identifica para onde o controle deve ser encaminhado (por exemplo, um JSP).

O método *execute* realiza as seguintes tarefas:

- Valida o estado atual da sessão do usuário (verifica, por exemplo, se o usuário efetuou o *login*. Caso não tenha feito, redireciona-o para a página inicial).
- Atualiza os objetos do lado do servidor que serão utilizados para a criação da próxima página de interface com o usuário.
- Retorna o *ActionFoward* apropriado, já com os *beans* atualizados.

O *ActionMapping* é a classe Java responsável pelo mapeamento da URI de requisição e a classe *Action* correspondente.

As principais propriedades desta classe são:

- *type* – nome da classe Java da implementação da classe *Action* usada no mapeamento.
- *name* – nome do *Form bean* definido no arquivo *config* que será usado pela *Action*.
- *path* – a URI de requisição que determina o mapeamento a ser utilizado.

- *validate* – deve ser configurado como *true* caso o método *validate* da *Action* associada deva ser invocado.
- *forward* – caminho da URI para qual o controle deva ser encaminhado quando este mapeamento for invocado.

4.2.4 SWIG

Swig é uma ferramenta que simplifica a integração de programas escritos em C e C++ com linguagens de *script*, como Perl, Python, Ruby e Tcl (nestas linguagens é assumida a existência de um conjunto de componentes já desenvolvidos em outras linguagens, de forma que o objetivo passa a ser o de combinar estes componentes e não o de desenvolver programas a partir de estruturas de dados elementares). Linguagens de *script* são também conhecidas como de linguagens de colagem ou de integração de sistemas.

O Swig utiliza as declarações contidas nos cabeçalhos dos arquivos escritos em C/C++ para gerar o código intermediário que as linguagens de *script* precisam para acessar o código C/C++ subjacente.

Além das linguagens de *script* citadas anteriormente, a partir da versão 1.3.6 do Swig a linguagem Java (Java JDK 1.1 ou superior) também passou a ser suportada. O Swig será útil para integrar o módulo do SARC relacionado ao Smart Card (implementado em Java) com a aplicação responsável pela comunicação da leitora com o computador (implementado em C).

4.2.5 Hibernate

O Hibernate provê uma solução para o mapeamento das classes para as tabelas do banco de dados, sendo a camada de persistência da aplicação. É um *software* livre, *open source* e distribuído sobre a licença LGPL [9].

O Hibernate fornece persistência transparente para *Plain Old Java Objects* (no caso do SARC, são os *beans* da camada *Model*) que, para poderem ser persistidas, devem apenas ter um construtor sem argumentos. Além disso, o Hibernate trata de maneira transparente as chaves primárias, estrangeiras e tabelas n:m associativas.

O Hibernate provê uma camada de abstração entre a lógica e os mecanismos de persistência, fazendo com que a aplicação seja facilmente alterada (uma troca no banco de dados não requer que nenhuma linha de código seja alterada). Apenas o arquivo *hibernate.cfg.xml* deve ser atualizado. Com o uso deste *framework*, o desenvolvedor deve preocupar-se apenas com a lógica de negócio, visto que toda a persistência dos dados é feita pelo Hibernate.

Este *framework* possui uma linguagem própria, similar à SQL, intitulada HQL (*Hibernate Query Language*). Esta linguagem faz com que o conceito de linhas no banco de dados seja abstraído. O programador deve pensar apenas em objetos que devem ser persistidos na base de dados ou retornados em uma consulta.

O mapeamento dos campos da tabela do banco de dados com os campos da classe POJO é feito no arquivo de configuração *.hbm.xml*, que deve ser preferencialmente único para cada classe Java (uma boa prática recomendada na documentação do Hibernate. Algumas ferramentas que utilizam este *framework* requerem que o mapeamento seja feito individualmente). Uma mudança nas tabelas do banco de dados ou nos atributos da classe Java mapeada faz com que seja necessária apenas uma alteração no arquivo xml e uma nova montagem dos componentes no servidor de aplicação.

O arquivo *.hbm.xml* deve possuir as seguintes *tags* que descrevem a relação dos atributos da classe Java com os campos na tabela do banco de dados:

- <class>: declaração de uma classe persistente. Pode conter diversos parâmetros, dentre eles *name*, que indica o nome da classe POJO a ser persistida e *table*, que indica o nome da tabela no banco de dados para a qual a classe deve ser mapeada. Caso o parâmetro *name* não esteja presente, o Hibernate assume que a classe a ser mapeada não é do tipo POJO.
- <id>: deve estar aninhada na *tag* *<class>* e é uma informação obrigatória no mapeamento. Indica qual atributo na classe Java é seu identificador e relaciona-o à coluna na tabela que é a chave primária. Pode conter parâmetros como *name*, que indica o nome do atributo da classe mapeada que é seu identificador, *type*, que indica o tipo do identificador e *column*, que indica o nome da coluna na tabela que é a chave primária.

- <property>: deve estar aninhada na *tag* *<class>* e é a declaração de um atributo da classe Java que deve ser persistida. Pode possuir diversos parâmetros, dentre eles *name*, que é o nome do atributo da classe Java e *column*, que indica o nome da coluna da tabela à qual o atributo deve ser mapeado.

O Hibernate também possui um arquivo *.cfg.xml*, que contém as informações do Banco de Dados em uso e dados necessários para o acesso ao mesmo (como nome de usuário e senha), a lista de arquivos *.hbm.xml* das classes a serem persistidas.

Todos os arquivos xml necessários podem ser facilmente gerados com ferramentas desenvolvidas. O arquivo de configuração (*.cfg.xml*) pode ser gerado pelo JBoss-IDE [20] e os xmls de mapeamento (*.hbm.xml*) com a ferramenta Hibernator, ambas que podem ser instaladas como *plugins* para o *software* Eclipse [21], que está sendo utilizado no ambiente de desenvolvimento.

4.3 Ambiente de execução

Nesta seção é feita uma descrição dos programas utilizados no ambiente de execução do SARC. São eles: o servidor de aplicação JBoss, o servidor *Web* Tomcat (contido no JBoss) e o banco de dados *MySQL*.

Foram preferencialmente escolhidos *softwares* livres e *open-source*, fazendo com que a aplicação seja independente de fornecedores e marcas. Além disso, a aplicação pode ser executada em plataforma de sistema operacional Windows ou Linux, pois foi desenvolvida em Java e possui um ambiente de execução que segue a filosofia do *software* livre.

É apresentada, a seguir, uma descrição resumida de cada um deles.

4.3.1 JBoss

É um servidor de aplicação *open-source* totalmente desenvolvido em Java que implementa os serviços J2EE.

O JBoss lida com toda a lógica da aplicação e conectividade do modelo cliente-servidor. Ele pode ser encarado como o *middleware* da aplicação, visto que fornece uma camada de abstração aos programadores que não precisam preocupar-se com o

sistema operacional ou com as inúmeras interfaces que se fazem necessárias em uma aplicação voltada para a *Web*.

O JBoss possui suporte ao *framework* Hibernate (ambos os projetos encontram-se sob a supervisão do mesmo grupo, a JBoss Inc), sendo o responsável pelo gerenciamento do *pool* de conexões ao banco de dados [11].

4.3.2 Tomcat

O Tomcat é o servidor *web* voltado para aplicações em Java, desenvolvido pela *Apache Software Foundation*. Ele já está incluso no servidor de aplicação JBoss e é responsável pelo diálogo com o *servlet* de controle da aplicação (por isso também é denominado um *servlet container*) [16].

O Tomcat implementa as especificações de *servlets* e *JavaServer Pages* (JSP) da *Sun Microsystems* e contém um compilador, o Jasper, que compila JSPs em *servlets*.

Ele age como um *daemon*, servindo documentos *web*. O Tomcat aceita as requisições HTTP da rede, fornecendo também uma resposta HTTP.

4.3.3 MySQL

É um sistema de gerenciamento de banco de dados que utiliza a linguagem SQL (*Structured Query Language*) como interface. É atualmente um dos bancos de dados mais utilizados, com mais de 4 milhões de instalações no mundo, sendo distribuído sob a *GNU General Public Licence* (GPL).Diferentemente dos projetos como o Apache, em que o desenvolvimento de *software* é feito pela comunidade, o MySQL pertence e tem suporte de uma empresa sueca, a MySQL AB [13].

O MySQL é escrito em C e C++ e é reconhecido pelo seu desempenho e robustez. É multi-tarefa e multi-usuário e possui inúmeras características positivas, tais como portabilidade (suporta diversas plataformas como Windows, Linux, MacOS, FreeBSD, Solaris, OpenBSD entre outros), compatibilidade com diversas linguagens (possui *drivers* ODBC, JDBC e .NET), possui uma ótima estabilidade e requer poucos recursos de *hardware*.

4.3.4 Versões dos aplicativos

Para o desenvolvimento do sistema, foram empregadas as seguintes versões de softwares:

- Java JDK 1.4.2_08;
- Eclipse 3.0.1;

Os frameworks utilizados foram:

- Hibernate 2.1.8
- SWIG
- Struts 1.2.7

No ambiente de execução, foram utilizados:

- MySQL 4.1.12 (com o *driver* MySQL *Conector* 3.1)
- JBoss 3.2.5 (a versão 4.x do JBoss não foi empregada por ser compatível apenas com o Java JDK 1.5)

4.4 Conclusões

Atualmente, encontramos diversos *frameworks* e especificações que tornam o desenvolvimento de aplicações de grande e médio porte muito mais simples. Seguindo os padrões, as aplicações tornam-se escalonáveis e modularizadas e o foco do desenvolvimento passa a ser a lógica de negócio.

Vale ressaltar que todos os *frameworks* e softwares utilizados são livres, o que não acarreta em custos com o seu uso, além de proporcionar um grau de liberdade muito maior, independente de plataformas e marcas.

A configuração do ambiente para o desenvolvimento de uma aplicação de grande porte empregando todas as tecnologias mencionadas no capítulo é uma tarefa dispendiosa, mas que se torna compensadora quando analisada em longo prazo. Para o desenvolvimento de uma única aplicação, grande parte do tempo será gasto nesta configuração. Entretanto, os *frameworks*, padrões e softwares formam uma base para o desenvolvimento de outras aplicações que podem utilizar a mesma infra-estrutura, por exemplo.

5 DESCRIÇÃO DA ORGANIZAÇÃO DO SARC

O objetivo deste capítulo é descrever o agrupamento das classes (pacotes) que formam o sistema. Existem dois diretórios principais: src e web.

O diretório src contém os códigos-fonte sob a licença GPL. O diretório *web* possui arquivos de configuração necessários para a aplicação e as telas JSPs, bem como os arquivos de extensão .css, que definem os aspectos visuais das telas de interface com o usuário. A seguir, é descrito o conteúdo destes dois diretórios com maior nível de detalhamento.

5.1 Diretório src

Este diretório contém o código-fonte do sistema, dividido em pacotes. Cada pacote contém os arquivos de extensão .java que fazem parte de uma mesma camada do *software* e que possuam funcionalidades relacionadas.

No sistema existem dois tipos de pacotes:

- subcadeia .infra: são pacotes que contém as classes e arquivos de infra-estrutura, que são utilizados por todas as demais classes do sistema e que são independentes das particularidades da aplicação. Estas classes podem ser reaproveitadas para outras aplicações e podem ser consideradas como um *framework*.
- subcadeia .sarc: são pacotes diretamente relacionados à aplicação e que contém todas as classes que implementam as regras de negócio e as entidades do sistema.

Classificando os pacotes segundo o *design pattern* MVC, tem-se, para a camada *Model*:

- *br.usp.poli.infra.core.files*: contém o arquivo de configuração XML utilizado pelo *framework* Hibernate. Neste arquivo estão contidas as informações necessárias para o acesso ao Banco de Dados (tais como o nome do usuário, a senha, etc.) e os arquivos que devem ser mapeados

(arquivos terminados em *.hbm.xml* que fazem o mapeamento entre as classes da camada *Model* e as tabelas do Banco de Dados).

- *br.usp.poli.infra.model*: contém os componentes básicos das entidades da camada *Model* que representam o estado atual da aplicação (ou, conforme o Capítulo 3, são os substantivos da camada *Model*): a classe *EntityObject*. Esta classe é uma das mais importantes da infra-estrutura, pois é responsável por unir as entidades da camada *Model* com a camada DAO do Hibernate. Todas as classes da camada *Model* da aplicação (substantivos) estendem esta classe, que é a implementação básica de uma entidade do sistema.
- *br.usp.poli.infra.model.business*: Contém o componente básico das entidades da camada *Model* responsáveis pela lógica de negócio (ou, conforme o Capítulo 3, são os verbos da camada *Model*): a classe *BusinessObject*. Todas as classes desta camada responsáveis pelas regras de negócio estendem esta classe.
- *br.usp.poli.infra.model.business.exception*: Contém as classes responsáveis pelas exceções lançadas caso ocorra algum erro na camada de negócio da aplicação, ou se alguma regra do negócio for violada. São elas, respectivamente: *BusinessException* e *BusinessValidationException*.
- *br.usp.poli.infra.model.dao*: Contém as classes fundamentais para a utilização do *framework* Hibernate pela aplicação (mais especificamente, pela camada *Model*). A classe *EntityDAO* é a implementação mínima de um objeto DAO (*GoF design pattern*) e fornece os métodos que funcionam como ponte entre a aplicação e a camada de persistência. Esta classe permite a existência de apenas uma classe DAO, que é utilizada por toda a aplicação, fazendo com que as classes DAOs relativas a cada classe da camada *Model* sejam desnecessárias. Já a classe *EntityDAOFactory* fornece objetos DAO para toda a aplicação. Ela implementa os *patterns* DAO (definidos no padrão J2EE) e *Factory* (*GoF design pattern*).
- *br.usp.poli.infra.model.dao.exception*: Contém as classes responsáveis pelas exceções lançadas, caso seja encontrado algum erro na inicialização do Hibernate: *HibernateInitializerException* e *HibernateAlreadyConfigured*

Exception. Contém ainda as classes responsáveis pelas exceções lançadas, caso haja algum erro na camada DAO (classe *DAOException*) e caso alguma entidade recebida pelo sistema seja inválida ou não correspondente ao seu tipo esperado (classe *InvalidEntityException*).

- *br.usp.poli.infra.model.dao.hibernate*: Contém as classes referentes ao *framework* Hibernate. A classe *HibernateCore* implementa os métodos para a obtenção de *sessions*, *transactions* e outros elementos inerentes à camada DAO. A classe *HibernateInitializer* é responsável pela inicialização do Hibernate e é chamada pela classe *InitializerServlet*.
- *br.usp.poli.infra.security*: Contém a classe *DigestGenerator*, que é responsável pela utilização do algoritmo MD5, que gera o *hash* de uma senha. Esta classe é utilizada para o armazenamento de senhas dos usuários no banco de dados, garantindo a segurança da aplicação.
- *br.usp.poli.sarc.model*: Contém as classes referentes à camada *Model* da aplicação e que descrevem seu estado atual. Também possui os arquivos de extensão .hbm.xml, que fazem o mapeamento destas classes para as tabelas no Banco de Dados.
- *br.usp.poli.sarc.model.business*: Contém as classes referentes à camada *Model* da aplicação, mais especificamente, de sua lógica de negócio. Este pacote é o coração da aplicação, pois aqui são implementadas as funcionalidades oferecidas ao usuário pelo sistema.

Classificando os pacotes segundo o *design pattern* MVC, tem-se, para a camada *View*:

- *br.usp.poli.sarc.resources*: Contém o arquivo *ApplicationResources. properties* que permite que a aplicação seja facilmente transportada para um outro idioma. Todos os arquivos do tipo JSPs da aplicação, que compõem a camada *View*, encontram-se no diretório *web*, bem como os arquivos de extensão .css, que definem a aparência da interface do sistema como o usuário.

Finalmente, classificando os pacotes segundo o *design pattern* MVC, tem-se, para a camada *Controller*:

- *br.usp.poli.infra.controller.action*: Contém o componente básico da camada *Controller*, implementada pelo Struts: a classe *StrutsDefaultAction*. Ela é a implementação mínima de uma *Action* padrão do Struts. Há ainda uma outra classe, a *StrutsDispatchAction*, que estende a classe *DispatchAction*, também definida pelo Struts. Esta *Action* é útil para casos onde a mesma classe deve ser reaproveitada para diversas tarefas, como no caso de remoção/inserção/atualização no Banco de Dados. O método a ser utilizado pela *Action* é definido pela propriedade *parameter*, mapeada no *ActionMappings*. Este pacote possui as funcionalidades centrais e básicas da camada *Controller* da aplicação.
- *br.usp.poli.infra.core*: Contém as classes de base para a aplicação. A classe *Constants* define constantes que serão utilizadas em toda a aplicação. Já a classe *InitializerServlet* é responsável pela inicialização do *servlet* principal de controle e de todos os recursos que devem ser configurados ou preparados para a inicialização da aplicação. No caso, o recurso Hibernate é inicializado: sua *SessionFactory* é configurada e inicializada a partir do arquivo de configurações XML, contido no pacote *br.usp.poli.infra.core.files*.
- *br.usp.poli.infra.exception*: Contém a classe *InfraException*, responsável por lançar as exceções, caso ocorra alguma falha nas classes de infra-estrutura do sistema.
- *br.usp.poli.sarc.controller.action*: Contém as classes da camada *Controller* da aplicação, mapeadas no *ActionMappings*.

5.2 Diretório web

O segundo diretório principal, definido com o nome *web*, contém todos os arquivos JSPs das telas do sistema, que compõe a camada *View* da aplicação. Estes arquivos encontram-se no diretório sarc.

O diretório css contém os arquivos de extensão .css que definem o padrão da interface do sistema.

Outro diretório importante é o WEB-INF que contém os arquivos XML de configuração da aplicação, que são:

- *struts-config.xml*: este é o *ActionMapping* da aplicação, que faz o mapeamento da URL recebida e a classe *Action* correspondente, bem como o mapeamento entre os *FormBeans* e suas classes correspondentes na camada *Model*.
- *validation.xml*: arquivo que descreve as regras de validação dos *FormBeans* da aplicação. Ele indica quais campos têm seu preenchimento obrigatório e as regras de validação dos mesmos (número e tipos de caracteres).

6 CONSIDERAÇÕES FINAIS

6.1 Resultados obtidos

Até o momento o sistema apresenta resultados satisfatórios. No entanto, são necessários mais testes com uma quantidade maior de pessoas para averiguar sua viabilidade.

Testes unitários (testes de caixa branca) foram feitos ao longo do processo de desenvolvimento. Posteriormente, foram realizados testes de caixa preta que analisavam a integração dos diversos módulos do sistema

No entanto, para assegurar a validação do sistema, usuários com o perfil de funcionários do COSEAS testariam funcionalidades envolvendo os controladores de estoque, nutricionista, alunos, funcionários da USP, compradores e administradores do departamento financeiro.

Seria interessante ainda avaliar com mais precisão o funcionamento correto de todas as consultas a banco de dados e sua persistência, as interfaces de comunicação, *hardware*, *software* e comunicação além das páginas *web* do sistema.

Aparentemente, o código desenvolvido não apresenta grandes problemas, com as funcionalidades essenciais funcionando corretamente. Entre os principais módulos do *software* em funcionamento, temos o de cadastro, compras, vendas, estoque e crédito.

Os atores atendidos pela versão inicial do sistema seriam os clientes, o caixa, os nutricionista, o supervisor de estoque e o funcionário da administração.

6.2 Dificuldades enfrentadas e soluções

Após a instalação do Hibernate, era necessário alterar arquivos de configuração para que a base de dados fosse reconhecida e corretamente persistida no ambiente de desenvolvimento. Como os tutoriais para criar as configurações eram escassos e pouco claros, foi necessário um tempo adicional para atribuir as configurações corretas.

Outro ponto importante a ser considerado é que havia muitos *plugins* a serem instalados e cada um possuía o seu próprio padrão. Também houve problemas com arquivos do tipo .jar que não eram encontrados em determinados locais. Mesmo alterando as variáveis de ambiente, o problema persistia. Sendo assim, a solução foi copiar os arquivos para os diretórios necessários.

O JBoss apresentou problemas com relação ao diretório em que poderia ser instalado (não era permitido o caractere de espaço). Após essa correção, surgiram outros erros com o *plugin* de interface com o Eclipse. Após várias pesquisas na internet, descobriu-se que a versão era incompatível com a do Eclipse. Assim, foi necessário instalar uma outra versão do *plugin* para que o JBoss funcionasse corretamente.

Após a criação de uma página de algum módulo do sistema, esta deveria ser configurada em diversos módulos da arquitetura do *design pattern* MVC para que esta página fosse incorporada e acessada como parte integrante do sistema.

Existiram dificuldades em gerar alguns scripts no MySQL para fazer com que as classes de lógica de negócio pudessem acessar corretamente a base de dados relacional do sistema.

Houve uma certa dificuldade em cadastrar o depósito de arquivos do sistema e acessá-lo remotamente. O problema foi resolvido pesquisando-se outros servidores para o serviço de CVS, até o encontro de um servidor satisfatório denominado *FreeRepository*.

Por fim, com relação aos *Smart Cards* a maior dificuldade foi encontrar informações completas a respeito da arquitetura e preços de equipamentos à venda em empresas localizadas em território nacional. As restrições inerentes ao processo de aquisição de equipamentos por conta da universidade atrasaram excessivamente o prazo de entrega do produto e então a compra não foi concretizada. Passou-se a utilizar alguns equipamentos do próprio departamento para a realização de testes, embora não possuíam nenhuma documentação. Até o momento da elaboração deste relatório, não tivemos sucesso na compreensão total dos dados já trocados entre a leitora e o cartão.

6.3 Contribuições

O principal legado do SARC foi o desenvolvimento de uma infra-estrutura para aplicações de médio/grande porte. Esta etapa demandou grande parte da fase de desenvolvimento do projeto e teve como produto um conjunto de classes Java independente das particularidades da camada de negócio da aplicação. Estas classes podem ser reaproveitadas em outros projetos sem sofrer qualquer alteração, podendo ser consideradas um *framework* de integração entre o Hibernate e o Struts.

Outra contribuição importante da aplicação é o módulo responsável pela comunicação com o *Smart Card*. A solução implementada dispensa o uso de *kits* de desenvolvimento extremamente caros e de *drivers* proprietários (a comunicação é feita diretamente com a porta serial), sendo uma solução eficiente e barata. Além disso, pode ser reutilizada para qualquer leitora de *Smart Card* com interface serial RS-232 que se adeque a especificação ISO 7816.

Vale ressaltar que o projeto é um exemplo de aplicação desenvolvido exclusivamente com ferramentas abertas e livres, comprovando a eficiência de *softwares* que seguem tal filosofia, que, por não necessitarem do pagamento de *royalties*, tornam a solução mais barata.

6.4 Versões futuras

Como a segurança não foi o foco, novos métodos poderiam ser desenvolvidos para tornar o sistema mais seguro, mas sem perder em demasia a praticidade. Nesse caminho, há a possibilidade de implementar novas formas de pagamentos com a parceria dos bancos.

As versões futuras do software teriam novas funcionalidades que com certeza surgiriam com a utilização do sistema. Provavelmente incluiriam novos tipos relatórios e acessos mais eficazes a determinadas funcionalidades.

Seria interessante a disponibilização do site em diversas línguas, visto que USP recebe estudantes de inúmeras nacionalidades. Essa funcionalidade seria de fácil implementação, uma vez que o sistema foi projetado com esta preocupação.

6.5 Viabilidade e implantação

O sistema SARC mostra-se muito eficiente. No entanto, para que sua implantação seja viável, é necessário um bom planejamento e que seja aumentado sua segurança.

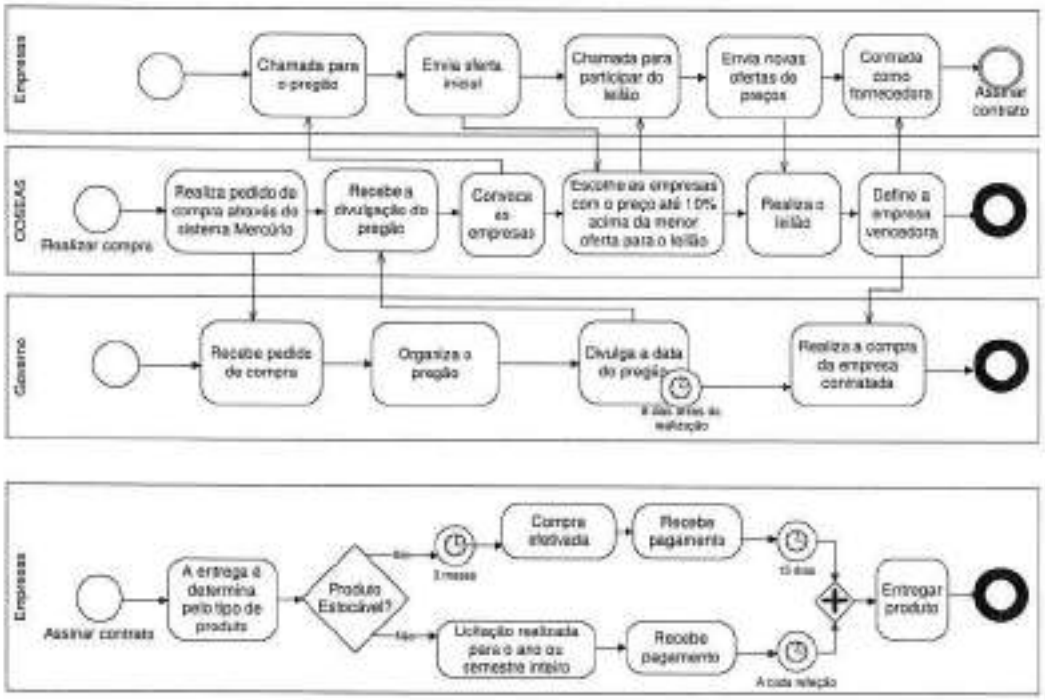
Em relação à parte que envolve o *Smart Card*, inicialmente, este pode ser implantado em restaurantes de menor porte em relação ao COSEAS, no qual o SARC trabalharia em paralelo com o sistema atual. Um grupo pequeno de pessoas tentaria encontrar problemas no sistema e fazer novas sugestões, as quais seriam analisadas de acordo com as facilidades que elas proporcionariam e o grau de dificuldade de implementá-las.

Após a consolidação do sistema, iniciaria a sua implantação nos demais restaurantes, ainda mantendo os dois sistemas. Caso o SARC suporte a imensa demanda, o sistema de pagamento via tickets seria suspenso, e o SARC atuaria sozinho, podendo-se efetuar a troca dos *tickets* durante um período. Estima-se que esse processo levaria cerca de seis meses.

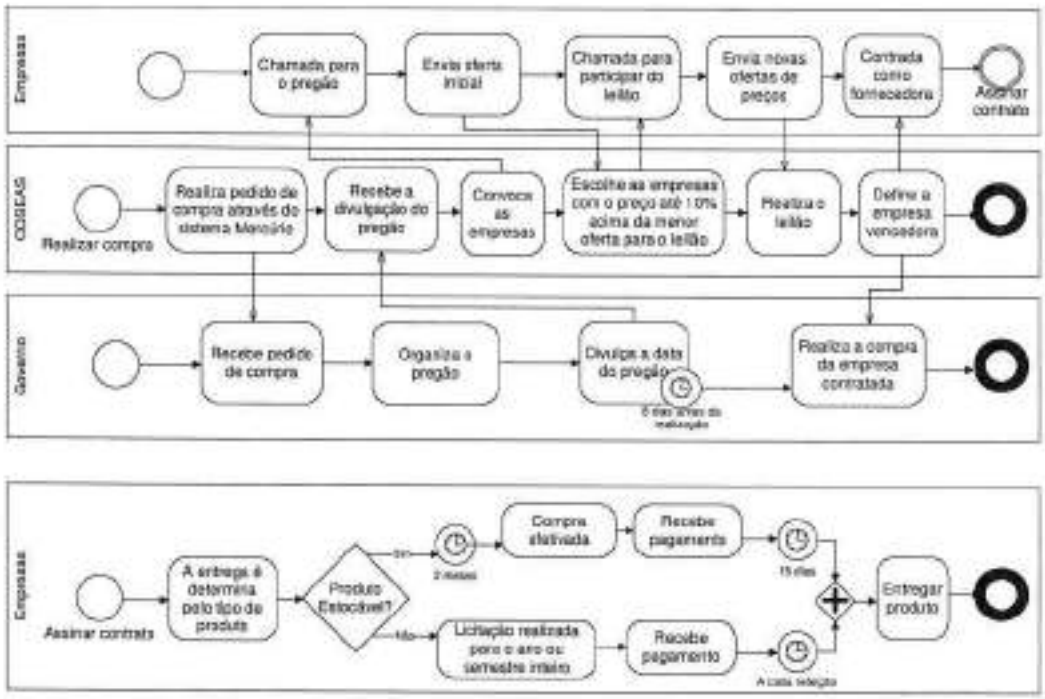
As demais partes do sistema seguiriam um processo semelhante, com sistemas em paralelos e com um volume menor de dados no início. Com a consolidação, a carga de dados seria aumentada, até que o sistema torne-se auto-suficiente.

7 ANEXO A – DIAGRAMAS BPMN

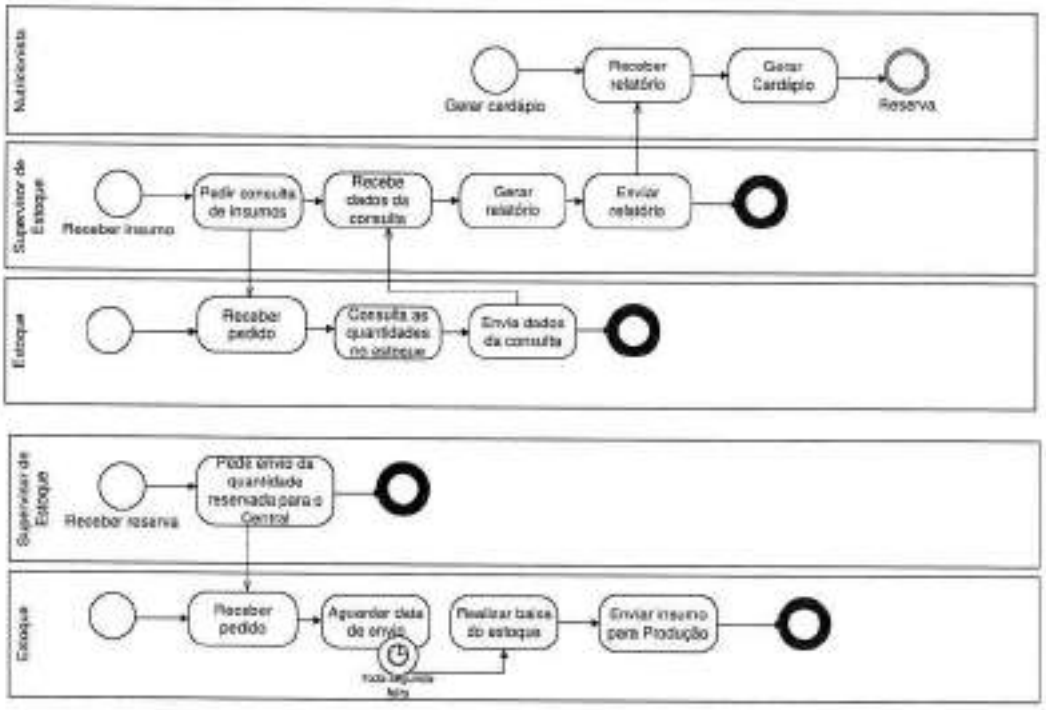
7.1 Compra de insumos



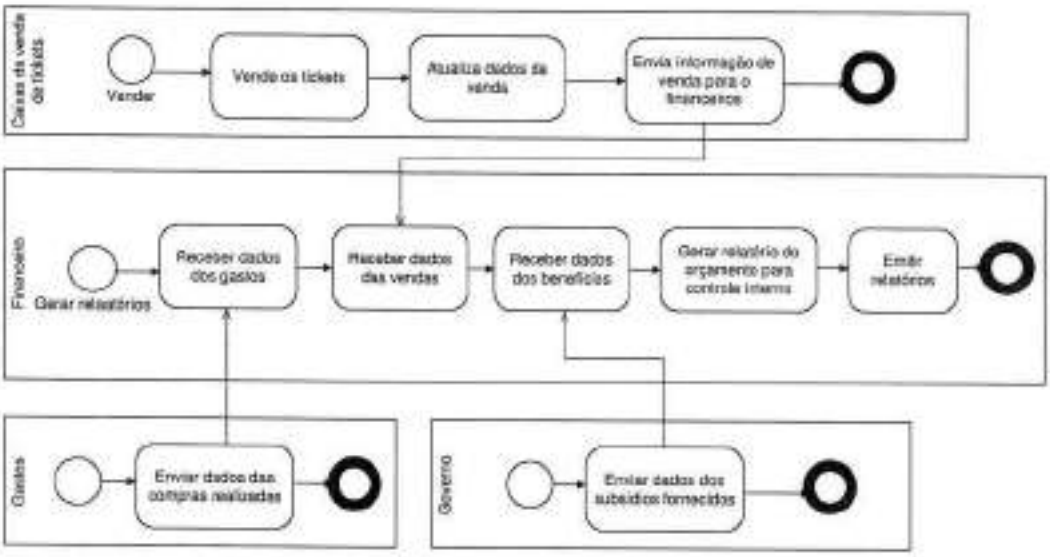
7.2 Recebimento de insumos



7.3 Geração de cardápio



7.4 Controle financeiro



8 ANEXO B – MODELO DE CASOS DE USO

1. Descrição dos Atores

- **Cliente:** Tem suas informações cadastrais e de freqüência armazenadas pelo sistema. Pode acessá-lo via *Internet* para efetuar a compra de *tickets* refeição.
- **Caixa:** Responsável pela venda de *tickets* refeição nos postos espalhados pelas faculdades. Acessa o sistema via *Intranet*.
- **Nutricionista:** Acessa o sistema via *Intranet* utilizando uma senha padrão para nutricionistas do COSEAS. Aprova o cardápio elaborado pelo sistema ou faz alterações no mesmo. Para tanto, tem acesso aos relatórios de estoque, que lista os produtos armazenados, bem como seus prazos de validade. Também tem a função de cadastrar os pratos que serão utilizados na elaboração do cardápio.
- **Supervisor de estoque:** Acessa o sistema via *Intranet*, com a senha padrão de supervisor de estoque. É responsável também pelo cadastro de produtos no Banco de Dados e pela entrada e saída de insumos nos almoxarifados.
- **Funcionário da administração:** Acessa o sistema via *Intranet*, usando uma senha padrão de funcionário de administração. Tem acesso a todos os relatórios gerados pelo sistema e é responsável pelo cadastro dos restaurantes na base de dados do sistema.

2. Descrição dos casos de uso

2.1 Módulo de Cadastro

2.1.1 Inserir Produto

Descrição: Este caso de uso descreve o processo de cadastro de um novo produto.

Evento iniciador: Supervisor de estoque seleciona a área de produtos cadastrados.

Atores: Supervisor de estoque.

Pré-condição: Supervisor *logado* no sistema.

Seqüência de eventos:

1. Supervisor de estoque seleciona a área de produtos cadastrados.
2. Sistema exibe uma lista de todos os produtos cadastros.
3. Supervisor seleciona a opção de cadastro de novos produtos.
4. Sistema exibe os campos necessários ao cadastro.
5. Supervisor insere os dados solicitados e confirma a operação.
6. Sistema registra no banco de dados as informações inseridas da etapa anterior e envia mensagem de sucesso ao usuário.
7. Supervisor finaliza a sessão.

Pós-condição: Novo produto cadastrado no sistema.

Cenário Secundário:

1. Supervisor insere um dado inconsistente ou fora de algum padrão: sistema emite uma mensagem de erro e retorna ao Passo 4.
2. Supervisor desiste de cadastrar um novo produto: sistema volta à sua tela inicial.
3. Supervisor tenta cadastrar um produto já existente na base de dados: sistema emite uma mensagem de erro e retorna ao Passo 4.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.2 Alterar Produto

Descrição: Este caso de uso descreve o processo de modificação de um campo de um produto cadastrado.

Evento iniciador: Supervisor de estoque seleciona a área de produtos cadastrados.

Atores: Supervisor de estoque.

Pré-condição: Supervisor *logado* no sistema, produto cadastrado.

Seqüência de eventos:

1. Supervisor de estoque seleciona a área de produtos cadastrados.
2. Sistema exibe uma lista de todos os produtos cadastros.
3. Supervisor seleciona o produto que ele deseja alterar.
4. Sistema exibe os campos do produto escolhido.
5. Supervisor seleciona a opção "alterar produto".
6. Sistema exibe os campos do produto, habilitados para edição.

7. Supervisor insere o(s) novo(s) dado(s) e confirma a operação.
8. Sistema registra no banco de dados as novas informações e envia mensagem de sucesso ao usuário.
9. Supervisor finaliza a sessão.

Pós-condição: Produto recadastrado no banco de dados.

Cenário Secundário:

1. Supervisor desiste de alterar os dados do produto: sistema volta à sua tela inicial.
2. Supervisor insere dados inconsistentes (Passo 7): sistema emite uma mensagem de erro e retorna ao Passo 6.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.3 Excluir Produto

Descrição: Este caso de uso descreve o processo de remoção de um produto cadastrado no sistema.

Evento iniciador: Supervisor de estoque seleciona a área de produtos cadastrados.

Atores: Supervisor de estoque.

Pré-condição: Supervisor *logado* no sistema.

Seqüência de eventos:

1. Supervisor de estoque seleciona a área de produtos cadastrados.
2. Sistema exibe uma lista de todos os produtos cadastros.
3. Supervisor seleciona a opção de exclusão de um determinado produto.
4. Sistema exibe uma mensagem pedindo a confirmação da operação.
5. Supervisor confirma a operação.
6. Sistema remove do banco de dados as informações referentes ao produto em questão e envia mensagem de sucesso ao usuário.
7. Supervisor finaliza a sessão.

Pós-condição: Produto excluído do banco de dados.

Cenário Secundário:

1. Supervisor desiste de excluir o produto: sistema volta à sua tela inicial.
2. Supervisor tenta excluir produto que possui pendência (compra já efetuada): ssitema exibe mensagem de erro e retorna ao Passo 2.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.4 Cadastrar Fornecedor

Descrição: Este caso de uso descreve o processo de cadastro de um novo fornecedor.

Evento iniciador: Supervisor de estoque seleciona a área de fornecedores cadastrados.

Atores: Supervisor de estoque.

Pré-condição: Supervisor *logado* no sistema.

Seqüência de eventos:

1. Supervisor de estoque seleciona a área de fornecedores cadastrados.
2. Sistema exibe uma lista de todos os fornecedores cadastros.
3. Supervisor seleciona a opção de cadastro de novos fornecedores.
4. Sistema exibe os campos necessários ao cadastro.
5. Supervisor insere os dados solicitados e confirma a operação.
6. Sistema registra no banco de dados as informações inseridas da etapa anterior e envia mensagem de sucesso ao usuário.
7. Supervisor finaliza a sessão.

Pós-condição: Novo fornecedor cadastrado no sistema.

Cenário Secundário:

1. Supervisor insere um dado inconsistente ou fora de algum padrão: sistema emite uma mensagem de erro.
2. Supervisor desiste de cadastrar um novo fornecedor: sistema volta à sua tela inicial.
3. Supervisor tenta cadastrar um fornecedor já existente na base de dados: sistema emite uma mensagem de erro e retorna ao Passo 4.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.5 Alterar Fornecedor

Descrição: Este caso de uso descreve o processo de modificação de um campo de um fornecedor cadastrado.

Evento iniciador: Supervisor de estoque seleciona a área de fornecedores cadastrados.

Atores: Supervisor de estoque.

Pré-condição: Supervisor *logado* no sistema, fornecedor cadastrado.

Seqüência de eventos:

1. Supervisor de estoque seleciona a área de fornecedores cadastrados.
2. Sistema exibe uma lista de todos os fornecedores cadastros.
3. Supervisor seleciona o fornecedor que ele deseja alterar.
4. Sistema exibe os campos do fornecedor escolhido
5. Supervisor seleciona a opção "alterar fornecedor"
6. Sistema exibe os campos do fornecedor, habilitados para edição.
7. Supervisor insere o(s) novo(s) dado(s) e confirma a operação.
8. Sistema registra no banco de dados as novas informações e envia mensagem de sucesso ao usuário.
9. Supervisor finaliza a sessão.

Pós-condição: Fornecedor recadastrado no banco de dados.

Cenário Secundário:

1. Supervisor desiste de alterar os dados do fornecedor: sistema volta à sua tela inicial.
2. Supervisor fornece dados inconsistentes (Passo 7): sistema exibe mensagem de erro e retorna ao Passo 6.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.6 Excluir Fornecedor

Descrição: Este caso de uso descreve o processo de remoção de um fornecedor cadastrado no sistema.

Evento iniciador: Supervisor de estoque seleciona a área de fornecedores cadastrados.

Atores: Supervisor de estoque.

Pré-condição: Supervisor *logado* no sistema.

Seqüência de eventos:

1. Supervisor de estoque seleciona a área de fornecedores cadastrados.
2. Sistema exibe uma lista de todos os fornecedores cadastros.
3. Supervisor seleciona a opção de exclusão de um determinado fornecedor.

4. Sistema exibe uma mensagem pedindo a confirmação da operação.
5. Supervisor confirma a operação.
6. Sistema remove do banco de dados as informações referentes ao fornecedor em questão e envia mensagem de sucesso ao usuário.
7. Supervisor finaliza a sessão.

Pós-condição: Fornecedor excluído do banco de dados.

Cenário Secundário:

1. Supervisor desiste de excluir o fornecedor: sistema volta à sua tela inicial.
2. Supervisor tenta excluir for4necedor com pendência (compra em andamento): sistema exibe mensagem de erro e retorna ao Passo 2.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.7 Inserir Restaurante

Descrição: Este caso de uso descreve o processo de cadastro de um novo restaurante.

Evento iniciador: Supervisor de estoque seleciona a área de restaurantes cadastrados.

Atores: Supervisor de estoque.

Pré-condição: Supervisor *logado* no sistema.

Seqüência de eventos:

1. Supervisor de estoque seleciona a área de restaurantes cadastrados.
2. Sistema exibe uma lista de todos os restaurantes cadastros.
3. Supervisor seleciona a opção de cadastro de novos restaurantes.
4. Sistema exibe os campos necessários ao cadastro.
5. Supervisor insere os dados solicitados e confirma a operação.
6. Sistema registra no banco de dados as informações inseridas da etapa anterior e envia mensagem de sucesso ao usuário.
7. Supervisor finaliza a sessão.

Pós-condição: Novo restaurante cadastrado no sistema.

Cenário Secundário:

1. Supervisor insere um dado inconsistente ou fora de algum padrão: sistema emite uma mensagem de erro.

2. Supervisor desiste de cadastrar um novo restaurante: sistema volta à sua tela inicial.
3. Supervisor tenta cadastrar um produto já existente na base de dados: sistema emite uma mensagem de erro e retorna ao Passo 4

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.8 Alterar Restaurante

Descrição: Este caso de uso descreve o processo de modificação de um campo de um restaurante cadastrado.

Evento iniciador: Supervisor de estoque seleciona a área de restaurantes cadastrados.

Atores: Supervisor de estoque.

Pré-condição: Supervisor *logado* no sistema, restaurante cadastrado.

Seqüência de eventos:

1. Supervisor de estoque seleciona a área de restaurantes cadastrados.
2. Sistema exibe uma lista de todos os restaurantes cadastros.
3. Supervisor seleciona o restaurante que ele deseja alterar.
4. Sistema exibe os campos do restaurante escolhido.
5. Supervisor seleciona a opção "alterar restaurante".
6. Sistema exibe os campos do restaurante, habilitados para edição.
7. Supervisor insere o(s) novo(s) dado(s) e confirma a operação.
8. Sistema registra no banco de dados as novas informações e envia mensagem de sucesso ao usuário.
9. Supervisor finaliza a sessão.

Pós-condição: Restaurante recadastrado no sistema.

Cenário Secundário:

1. Supervisor desiste de cadastrar um novo restaurante: sistema volta à sua tela inicial.
2. Supervisor fornece dados inconsistentes (Passo 7): sistema exibe mensagem de erro e retorna ao Passo 6.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.9 Excluir Restaurante

Descrição: Este caso de uso descreve o processo de remoção de um restaurante cadastrado no sistema.

Evento iniciador: Supervisor de estoque seleciona a área de restaurantes cadastrados.

Atores: Supervisor de estoque.

Pré-condição: Supervisor *logado* no sistema.

Seqüência de eventos:

1. Supervisor de estoque seleciona a área de restaurantes cadastrados.
2. Sistema exibe uma lista de todos os restaurantes cadastros.
3. Supervisor seleciona a opção de exclusão de um determinado restaurante.
4. Sistema exibe uma mensagem pedindo a confirmação da operação.
5. Supervisor confirma a operação.
6. Sistema remove do banco de dados as informações referentes ao restaurante em questão e envia mensagem de sucesso ao usuário.
7. Supervisor finaliza a sessão.

Pós-condição: Restaurante excluído do banco de dados.

Cenário Secundário:

1. Supervisor desiste de excluir o restaurante: sistema volta à sua tela inicial.
2. Supervisor tenta excluir restaurante com pendência (produtos em estoque): sistema emite mensagem de erro e retorna ao Passo 2.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.10 Inserir Prato

Descrição: Este caso de uso descreve o processo de cadastro de um novo prato.

Evento iniciador: Nutricionista seleciona a área de pratos cadastrados.

Atores: Nutricionista.

Pré-condição: Nutricionista *logado* no sistema.

Seqüência de eventos:

1. Nutricionista seleciona a área de pratos cadastrados.
2. Sistema exibe uma lista de todos os pratos cadastros.
3. Nutricionista seleciona a opção de cadastro de novos pratos.

4. Sistema exibe os campos necessários ao cadastro.
5. Nutricionista insere os dados solicitados e confirma a operação.
6. Sistema registra no banco de dados as informações inseridas da etapa anterior e envia mensagem de sucesso ao usuário.
7. Nutricionista finaliza a sessão.

Pós-condição: Novo prato cadastrado no sistema.

Cenário Secundário:

1. Nutricionista insere um dado inconsistente ou fora de algum padrão: sistema emite uma mensagem de erro.
2. Nutricionista desiste de cadastrar um novo prato: sistema volta à sua tela inicial.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.11 Alterar Prato

Descrição: Este caso de uso descreve o processo de modificação de um campo de um prato cadastrado.

Evento iniciador: Nutricionista seleciona a área de pratos cadastrados.

Atores: Nutricionista.

Pré-condição: Nutricionista *logado* no sistema, prato cadastrado.

Seqüência de eventos:

1. Nutricionista seleciona a área de pratos cadastrados.
2. Sistema exibe uma lista de todos os pratos cadastros.
3. Nutricionista seleciona o prato que ele deseja alterar.
4. Sistema exibe os campos do prato escolhido, habilitados para edição.
5. Nutricionista insere o(s) novo(s) dado(s) e confirma a operação.
6. Sistema registra no banco de dados as novas informações e envia mensagem de sucesso ao usuário.
7. Nutricionista finaliza a sessão.

Pós-condição: Prato recadastrado no banco de dados.

Cenário Secundário:

1. Nutricionista desiste de cadastrar um novo prato: sistema volta à sua tela inicial.

2. Nutricionista fornece dados inconsistentes (Passo 5): sistema envia mensagem de erro e retorna ao Passo 4.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.12 Excluir Prato

Descrição: Este caso de uso descreve o processo de remoção de um prato cadastrado no sistema.

Evento iniciador: Nutricionista seleciona a área de pratos cadastrados.

Atores: Nutricionista.

Pré-condição: Nutricionista *logado* no sistema.

Seqüência de eventos:

1. Nutricionista seleciona a área de pratos cadastrados.
2. Sistema exibe uma lista de todos os pratos cadastros.
3. Nutricionista seleciona a opção de exclusão de um determinado prato.
4. Sistema exibe uma mensagem pedindo a confirmação da operação.
5. Nutricionista confirma a operação.
6. Sistema remove do banco de dados as informações referentes ao prato em questão e envia mensagem de sucesso ao usuário.
7. Nutricionista finaliza a sessão.

Pós-condição: Prato excluído do sistema.

Cenário Secundário:

1. Nutricionista desiste de excluir o prato: sistema volta à sua tela inicial.
2. Nutricionista tenta remover prato com pendência (em uso no cardápio): sistema envia mensagem de erro e retorna ao Passo 2.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.13 Inserir Cardápio Servido

Descrição: Este caso de uso descreve o processo de cadastro de um novo cardápio servido.

Evento iniciador: Funcionário seleciona a área de cardápio servidos cadastrados.

Atores: Funcionário.

Pré-condição: Funcionário *logado* no sistema.

Seqüência de eventos:

1. Funcionário seleciona a área de cardápios servidos cadastrados.
2. Sistema exibe uma lista de todos os cardápios servidos cadastrados.
3. Funcionário seleciona a opção de cadastro de novos cardápios servidos.
4. Sistema exibe os campos necessários ao cadastro.
5. Funcionário insere os dados solicitados e confirma a operação.
6. Sistema registra no banco de dados as informações inseridas da etapa anterior e envia mensagem de sucesso ao usuário.
7. Funcionário finaliza a sessão.

Pós-condição: Novo cardápio servido cadastrado no sistema.

Cenário Secundário:

1. Funcionário insere um dado inconsistente ou fora de algum padrão: sistema emite uma mensagem de erro.
2. Funcionário desiste de cadastrar um novo cardápio servido: sistema volta à sua tela inicial.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.14 Alterar Cardápio Servido

Descrição: Este caso de uso descreve o processo de modificação de um campo de um cardápio servido cadastrado.

Evento iniciador: Funcionário seleciona a área de cardápios servidos cadastrados.

Atores: Funcionário.

Pré-condição: Funcionário *logado* no sistema, cardápio servido cadastrado.

Seqüência de eventos:

1. Funcionário seleciona a área de cardápios servidos cadastrados.
2. Sistema exibe uma lista de todos os cardápios servidos cadastros.
3. Funcionário seleciona o cardápio servido que ele deseja alterar.
4. Sistema exibe os campos do cardápio servido escolhido, habilitados para edição.
5. Funcionário insere o(s) novo(s) dado(s) e confirma a operação.
6. Sistema registra no banco de dados as novas informações e envia mensagem de sucesso ao usuário.

7. Funcionário finaliza a sessão.

Pós-condição: Cardápio servido recadastrado no banco de dados.

Cenário Secundário:

1. Funcionário desiste de cadastrar um novo cardápio servido: sistema volta à sua tela inicial.
2. Funcionário fornece dados inconsistentes (Passo 5): sistema envia mensagem de erro e retorna ao Passo 4.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.15 Excluir Cardápio Servido

Descrição: Este caso de uso descreve o processo de remoção de um cardápio servido cadastrado no sistema.

Evento iniciador: Funcionário selecciona a área de cardápios servidos cadastrados.

Atores: Funcionário.

Pré-condição: Funcionário *logado* no sistema.

Seqüência de eventos:

1. Funcionário seleciona a área de cardápios servidos cadastrados.
2. Sistema exibe uma lista de todos os cardápios servidos cadastros.
3. Funcionário seleciona a opção de exclusão de um determinado cardápio servido.
4. Sistema exibe uma mensagem pedindo a confirmação da operação.
5. Funcionário confirma a operação.
6. Sistema remove do banco de dados as informações referentes ao cardápio servido em questão e envia mensagem de sucesso ao usuário.
7. Funcionário finaliza a sessão.

Pós-condição: Cardápio servido excluído do sistema.

Cenário Secundário:

1. Funcionário desiste de excluir o cardápio servido: sistema volta à sua tela inicial.
2. Funcionário tenta remover cardápio servido com pendência (em uso no cardápio): sistema envia mensagem de erro e retorna ao Passo 2.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.16 Inserir Aluno

Descrição: Este caso de uso descreve o processo de cadastro de um novo aluno.

Evento iniciador: Funcionário seleciona a área de alunos cadastrados.

Atores: Funcionário.

Pré-condição: Funcionário *logado* no sistema.

Seqüência de eventos:

1. Funcionário seleciona a área de alunos cadastrados.
2. Sistema exibe uma lista de todos os alunos cadastrados.
3. Funcionário seleciona a opção de cadastro de novos alunos.
4. Sistema exibe os campos necessários ao cadastro.
5. Funcionário insere os dados solicitados e confirma a operação.
6. Sistema registra no banco de dados as informações inseridas da etapa anterior e envia mensagem de sucesso ao usuário.
7. Funcionário finaliza a sessão.

Pós-condição: Novo aluno cadastrado no sistema.

Cenário Secundário:

1. Funcionário insere um dado inconsistente ou fora de algum padrão: sistema emite uma mensagem de erro.
2. Funcionário desiste de cadastrar um novo aluno: sistema volta à sua tela inicial.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.17 Alterar Aluno

Descrição: Este caso de uso descreve o processo de modificação de um campo de um cardápio servido cadastrado.

Evento iniciador: Funcionário seleciona a área de alunos cadastrados.

Atores: Funcionário.

Pré-condição: Funcionário *logado* no sistema, aluno cadastrado.

Seqüência de eventos:

1. Funcionário seleciona a área de alunos cadastrados.
2. Sistema exibe uma lista de todos os alunos cadastros.
3. Funcionário seleciona o aluno que ele deseja alterar.

4. Sistema exibe os campos do aluno escolhido, habilitados para edição.
5. Funcionário insere o(s) novo(s) dado(s) e confirma a operação.
6. Sistema registra no banco de dados as novas informações e envia mensagem de sucesso ao usuário.
7. Funcionário finaliza a sessão.

Pós-condição: Aluno recadastrado no banco de dados.

Cenário Secundário:

1. Funcionário desiste de cadastrar um novo aluno: sistema volta à sua tela inicial.
2. Funcionário fornece dados inconsistentes (Passo 5): sistema envia mensagem de erro e retorna ao Passo 4.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.1.18 Excluir Aluno

Descrição: Este caso de uso descreve o processo de remoção de um aluno cadastrado no sistema.

Evento iniciador: Funcionário seleciona a área de alunos cadastrados.

Atores: Funcionário.

Pré-condição: Funcionário *logado* no sistema.

Seqüência de eventos:

1. Funcionário seleciona a área de alunos cadastrados.
2. Sistema exibe uma lista de todos os alunos cadastros.
3. Funcionário seleciona a opção de exclusão de um determinado aluno.
4. Sistema exibe uma mensagem pedindo a confirmação da operação.
5. Funcionário confirma a operação.
6. Sistema remove do banco de dados as informações referentes ao aluno em questão e envia mensagem de sucesso ao usuário.
7. Funcionário finaliza a sessão.

Pós-condição: Aluno excluído do sistema.

Cenário Secundário:

1. Funcionário desiste de excluir o aluno: sistema volta à sua tela inicial.

2. Funcionário tenta remover aluno com pendência (em uso no cardápio): sistema envia mensagem de erro e retorna ao Passo 2.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.2 Módulo de Crédito

2.2.1 Debitar créditos ao passar pela catraca

Descrição: Este caso de uso descreve o processo de débito da conta do usuário ao passar pela catraca eletrônica.

Evento iniciador: Cliente passa o cartão no leitor da catraca.

Atores: Cliente.

Pré-condição: Leitor do cartão funcionando e aguardando leitura.

Seqüência de eventos:

1. Sistema realiza leitura no cartão;
2. Sistema verifica se usuário já realizou a refeição;
3. Sistema verifica o crédito contido no cartão;
4. Sistema registra a hora da entrada do cliente no restaurante;
5. Sistema destrava catraca, liberando o acesso ao restaurante;

Pós-condição: Entrada registrada, número de créditos no cartão decrementado e catraca liberada.

Cenário secundário:

1. Falha na leitura (Passo 1) – Informa erro na leitura. Sistema barra entrada e pede nova leitura;
2. Usuário travado (Passo 2) – Informa que usuário já realizou a refeição. Sistema barra entrada e pede nova leitura;
3. Aluno sem crédito (Passo 3) – Informa que não há mais crédito. Sistema barra entrada e pede nova leitura;

Inclusão: Nenhuma.

2.3 Módulo de Compra

2.3.1 Sugerir Compra

Descrição: Este caso de uso descreve o processo de sugestão de compra de insumos.

Evento iniciador: Supervisor de estoque requisita a função de sugestão de compra

Atores: Supervisor de estoque.

Pré-condição: Supervisor de estoque *logado* no sistema.

Seqüência de eventos:

1. Supervisor de estoque requisita a função sugestão de compra.
2. Sistema exibe a lista de insumos cadastrados.
3. Supervisor de estoque seleciona os produtos.
4. Sistema consulta sua base de dados do estoque e executa algoritmos para gerar a sugestão de compra e exibe o resultado na tela.
5. Supervisor de estoque finaliza a operação.

Pós-condição: Sugestão de compra exibida ao Supervisor de estoque.

Cenário Secundário:

1. Supervisor de estoque desiste da operação (Passos 2 e 3) – sistema retorna à tela inicial.

Inclusão: Caso de uso *logar* no sistema.(Passo 1).

2.4 Módulo de Consumo

2.4.1 Gerar Cardápio

Descrição: Este caso de uso descreve o processo de geração do cardápio semanal utilizado nos restaurantes do COSEAS - USP.

Evento iniciador: Nutricionista requisita a função gerar cardápio.

Atores: Nutricionista.

Pré-condição: Nutricionista *logado* no sistema.

Seqüência de eventos:

2. Nutricionista requisita a função gerar cardápio.
3. Sistema consulta sua base de dados do estoque, gera o cardápio e exibe o resultado na tela.

4. Nutricionista aprova o cardápio sugerido.
5. Sistema registra o cardápio na sua base de dados.
6. Nutricionista finaliza a operação.

Pós-condição: Cardápio cadastrado e disponibilizado para visualização na *internet*.

Extensão:

1. Nutricionista não aprova o cardápio sugerido e faz as modificações necessárias manualmente, utilizando o botão alterar. (Passo 3) – Caso de uso Alterar Cardápio.

Inclusão:

1 Caso de uso *logar* no sistema(Passo 1).

2.4.2 Alterar Cardápio

Descrição: Este caso de uso descreve o processo de alteração do cardápio semanal gerado pelo sistema.

Evento iniciador: Nutricionista seleciona função alterar cardápio.

Atores: Nutricionista.

Pré-condição: Nutricionista *logado* no sistema e cardápio gerado automaticamente.

Seqüência de eventos:

2 Nutricionista seleciona função alterar cardápio.
3 Sistema exibe os campos dos pratos constituintes de cada refeição, habilitados para edição.
4 Nutricionista altera os dados convenientes e confirma a operação.
5 Sistema registra o cardápio na sua base de dados.
6 Nutricionista finaliza a operação.

Pós-condição: Cardápio é alterado sendo disponibilizado para visualização na Internet.

Cenário Secundário:

1 Nutricionista desiste de efetuar a alteração no cardápio (Passos 2 e 3). Sistema retorna à tela de exibição do cardápio gerado automaticamente.
2 Nutricionista fornece dados inconsistentes (Passo 3): sistema envia mensagem de erro e retorna ao Passo 2.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.4.3 Gerar Relatório de Demanda

Descrição: Este caso de uso descreve o processo de geração de relatórios para averiguar a demanda de um dia. Serão analisadas as demandas com base no cardápio da refeição, condições climáticas, dia da semana e no horário em que foi servida (almoço ou jantar).

Evento iniciador: Usuário solicita relatório.

Atores: Funcionário da administração e Nutricionista.

Pré-condição: dados disponíveis para análise e usuário *logado* no sistema.

Seqüência de eventos:

1 Usuário solicita a função de geração de relatório de demanda.

2 Sistema solicita ao usuário o preenchimento opcional da data de início, data de fim, dia da semana, condições climáticas, horário da refeição e cardápio;

3 Usuário preenche os dados solicitados e confirma a operação;

4 Sistema faz uma busca baseada nos dados preenchidos pelo usuário e retorna um relatório padronizado com os dados preenchidos pelo usuário, o número de maior movimento, o número de menor movimento, o número médio do movimento bem como o desvio padrão e o número de dias analisados.

Pós-condição: Relatório exibido.

Cenário secundário:

1 Usuário fornece dados inconsistentes (Passo 3): sistema envia mensagem de erro e retorna ao Passo 2

2 Falha na busca (Passo 4) – Informa a causa da falha. Volta ao Passo 2

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.5 Módulo de Venda

2.5.1 Vender ticket refeição nos caixas

Descrição: Este caso de uso descreve o processo de venda de ticket utilizando caixas.

Evento iniciador: Caixa requisita a função venda de tickets.

Atores: Caixa.

Pré-condição: Caixa *logado* no sistema.

Seqüência de eventos:

1 Caixa requisita a função venda de tickets.

2 Sistema pede a quantidade de tickets.

3 Caixa informa o dado solicitado.

4 Sistema informa o valor da compra e pede a confirmação da operação.

5 Caixa confirma a operação.

6 Sistema requisita o cartão para gravar os dados.

7 Caixa passa o cartão no leitor.

8 Sistema credita o número de refeições ao cliente.

9 Caixa finaliza a operação.

Pós-condição: Aluno recebe créditos para utilizar nos restaurantes da USP.

Cenário Secundário:

1. Aluno desiste de fazer a compra e o caixa clica no botão cancelar. O sistema volta a sua tela inicial (passos 2,3,4 e 5).

Inclusão: Caso de *logar* no sistema (Passo1).

2.5.2 Vender ticket refeição pela Internet

Descrição: Este caso de uso descreve o processo de venda de ticket utilizando acesso remoto via internet.

Evento iniciador: Usuário habilitado pelo sistema acessa página do sistema de venda de tickets.

Atores: Cliente (Aluno ou Funcionário).

Pré-condição: Cliente *logado* no sistema.

Seqüência de eventos:

1 Cliente requisita a função venda de tickets.

2. Sistema pede a quantidade de tickets

3. Cliente informa o número de tickets a serem adquiridos.

4. Sistema informa o valor da compra e gera uma lista de bancos conveniados.

5. Cliente seleciona o banco de preferência para efetuar a transação.

6. Sistema pede a identificação da transação.

7. Cliente informa o número de identificação.

8. Sistema gera o comprovante.
9. Cliente imprime o comprovante para apresentá-lo nos caixas..
10. Cliente finaliza a operação.

Pós-condição: Depósito feito na conta corrente do COSEAS e comprovante de transação impresso.

Cenário Secundário:

1. Aluno desiste de fazer a compra e clica no botão cancelar. O sistema volta a sua tela inicial (passos 2,3,4, 5 e 6).

Inclusão: Caso de uso de *logar* no sistema (Passo 1).

2.6 Módulo de Estoque

2.6.1 Entrar Insumos

Descrição: Este caso de uso descreve o processo de registro da entrada de insumos no estoque do COSEAS (que pode ser local ou remoto), procedimento necessário à supervisão geral da quantidade de insumos disponíveis para o preparo das refeições.

Evento iniciador: Supervisor de estoque seleciona a área de entradas de insumos.

Atores: Supervisor de estoque.

Pré-condição: Supervisor *logado* e insumo pré-cadastrado no sistema.

Seqüência de eventos:

1. Supervisor de estoque seleciona a área de entradas de insumos.
2. Sistema exibe todos os campos necessários ao registro da entrada de um insumo qualquer.
3. Supervisor insere os dados solicitados e confirma a operação.
4. Sistema registra no banco de dados as informações inseridas na etapa anterior e emite uma mensagem notificando o sucesso da operação.
5. Supervisor finaliza a sessão.

Pós-condição: Registro da entrada de um insumo no estoque.

Cenário Secundário:

1. Supervisor insere um código de insumo inválido (não cadastrado previamente): sistema emite uma mensagem de erro.

2. Supervisor fornece dados inconsistentes (Passo 2): sistema emite uma mensagem de erro e retorna ao Passo 1.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.6.2 Sair Insumos

Descrição: Este caso de uso descreve o processo de baixa de insumos do estoque do COSEAS (local ou remoto).

Evento iniciador: Supervisor de estoque seleciona a área de baixa de insumos.

Atores: Supervisor de estoque.

Pré-condição: Supervisor *logado* e insumos armazenados no estoque do COSEAS.

Seqüência de eventos:

1. Supervisor de estoque seleciona a área de baixa de insumos.
2. Sistema pede a identificação do almoxarifado do qual será feita a baixa do insumo.
3. Usuário preenche os dados solicitados e confirma a operação;
4. Sistema exibe uma lista de todos os insumos armazenados naquele estoque do COSEAS.
5. Supervisor seleciona o tipo de insumo que ele deseja dar baixa no estoque.
6. Sistema apresenta a quantidade total disponível para retirada, além de um campo adicional onde o supervisor definirá o valor a ser contabilizado na baixa do estoque e o local de destino do insumo.
7. Supervisor informa a quantidade de insumo a ser retirado do estoque e confirma a operação.
8. Sistema atualiza o valor da quantidade de insumos disponíveis no estoque e envia uma mensagem ao usuário notificando o sucesso da operação
9. Supervisor finaliza a sessão.

Pós-condição: Estoque local quantitativamente atualizado.

Cenário Secundário:

1. Usuário fornece dados inconsistentes (Passo 3): sistema envia mensagem de erro e retorna ao Passo 2
2. Supervisor insere um valor acima do total armazenado no estoque (Passo 7): sistema emite uma mensagem de erro e retorna ao Passo 6.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.6.3 Gerar relatórios sobre prazos de validade

Descrição: Este caso de uso descreve o processo de geração de um relatório sobre os alimentos estocados que terão seu prazo de validade vencido até determinada data.

Evento iniciador: Usuário seleciona a área de relatórios.

Atores: Nutricionista ou Supervisor de estoque.

Pré-condição: Usuário *logado* e insumos armazenados no estoque do COSEAS.

Seqüência de eventos:

1. Usuário seleciona a área de relatórios.
2. Sistema exibe os tipos de relatórios disponíveis para o usuário.
3. Usuário seleciona o tipo de relatório de prazos de validade.
4. Sistema pede a data ao usuário.
5. Usuário entra com a data desejada.
6. Sistema exibe todos os produtos armazenados que terão seu prazo de validade vencido até a data fornecida pelo nutricionista.
7. Usuário finaliza a sessão.

Pós-condição: Relatório gerado.

Cenário Secundário:

1. Usuário fornece uma data inconsistente (Passo 5): sistema emite uma mensagem de erro e retorna ao Passo 4.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.6.4 Gerar relatórios sobre produtos em estoque

Descrição: Este caso de uso descreve o processo de geração de um relatório sobre os insumos armazenados nos estoques do COSEAS.

Evento iniciador: Usuário seleciona a área de relatórios.

Atores: Nutricionista ou Supervisor de estoque.

Pré-condição: Nutricionista *logado* e insumos armazenados no estoque do COSEAS.

Seqüência de eventos:

1. Usuário seleciona a área de relatórios.

2. Sistema exibe os tipos de relatórios disponíveis para o usuário.
3. Usuário seleciona o tipo de relatório de produtos em estoque.
4. Sistema exibe todos os produtos armazenados, bem como sua quantidade, localização e prazos de validade.
5. Usuário finaliza a sessão.

Pós-condição: Relatório gerado.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.7 Módulo Financeiro

2.7.1 Gerar relatório financeiro

Descrição: Este caso de uso descreve o processo de geração de relatórios de finanças. Serão mostrados os custos e a receita adquirida com base nos dados de entrada.

Evento iniciador: Funcionário da administração solicita relatório.

Atores: Funcionário da administração.

Pré-condição: Funcionário *logado* e dados disponíveis para análise.

Seqüência de eventos:

1. Funcionário da administração solicita relatório;
2. Sistema solicita ao Funcionário da administração o preenchimento opcional da data de início, data de fim;
3. Sistema faz uma busca baseada nos dados preenchidos pelo usuário e retorna um relatório contendo as despesas e receitas do período.

Pós-condição: Relatório exibido.

Cenário secundário:

1. Falha na busca (Passo 2) – Informa a causa da falha. Volta ao Passo 1.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

2.7.2 Incluir Receita

Descrição: Este caso de uso descreve o processo de inclusão de receitas no total de receitas.

Evento iniciador: Funcionário da administração solicita inclusão de receita.

Atores: Funcionário da administração.

Pré-condição: Funcionário *logado*.

Seqüência de eventos:

1. Funcionário da administração solicita inclusão de receita;
2. Sistema solicita ao Funcionário da administração os dados necessários para a inclusão;
3. Funcionário da administração entra com os dados;
4. Sistema pede a confirmação da operação;
5. Funcionário da administração confirma operação.

Pós-condição: Receita cadastrada.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

Extensão: Usuário cancela operação (Passo 4).

2.7.3 Incluir Despesa

Descrição: Este caso de uso descreve o processo de inclusão de despesas no total de despesas.

Evento iniciador: Funcionário da administração ou supervisor de estoque solicita inclusão de despesa.

Atores: Funcionário da administração e Supervisor de estoque.

Pré-condição: Nenhuma.

Seqüência de eventos:

1. Funcionário da administração ou supervisor de estoque solicita inclusão de despesa.
2. Sistema solicita ao Funcionário da administração ou supervisor os dados necessários para a inclusão;
3. Funcionário da administração ou supervisor de estoque entra com os dados;
4. Sistema pede a confirmação da operação;
5. Funcionário da administração ou supervisor de estoque confirma operação.

Pós-condição: Despesa cadastrada.

Inclusão: Caso de uso *logar* no sistema (Passo 1).

Extensão: Usuário cancela operação(Passo 4).

2.8 Módulo Geral

2.8.1 Logar no sistema

Descrição: Este caso de uso descreve o processo de autenticação de usuário no sistema

Evento iniciador:Usuário acessa a página de home do sistema.

Atores: Clientes (Alunos ou Funcionários), Supervisor de estoque, Nutricionista, Caixa, Funcionário da Administração.

Pré-condição: Usuário cadastrado no sistema.

Seqüência de eventos:

1. Sistema pede dados de identificação do usuário;
2. Cliente fornece seu *login* e senha;
3. Sistema verifica a validade dos dados fornecidos, identifica a classe do usuário e redireciona-o para a sua página inicial.

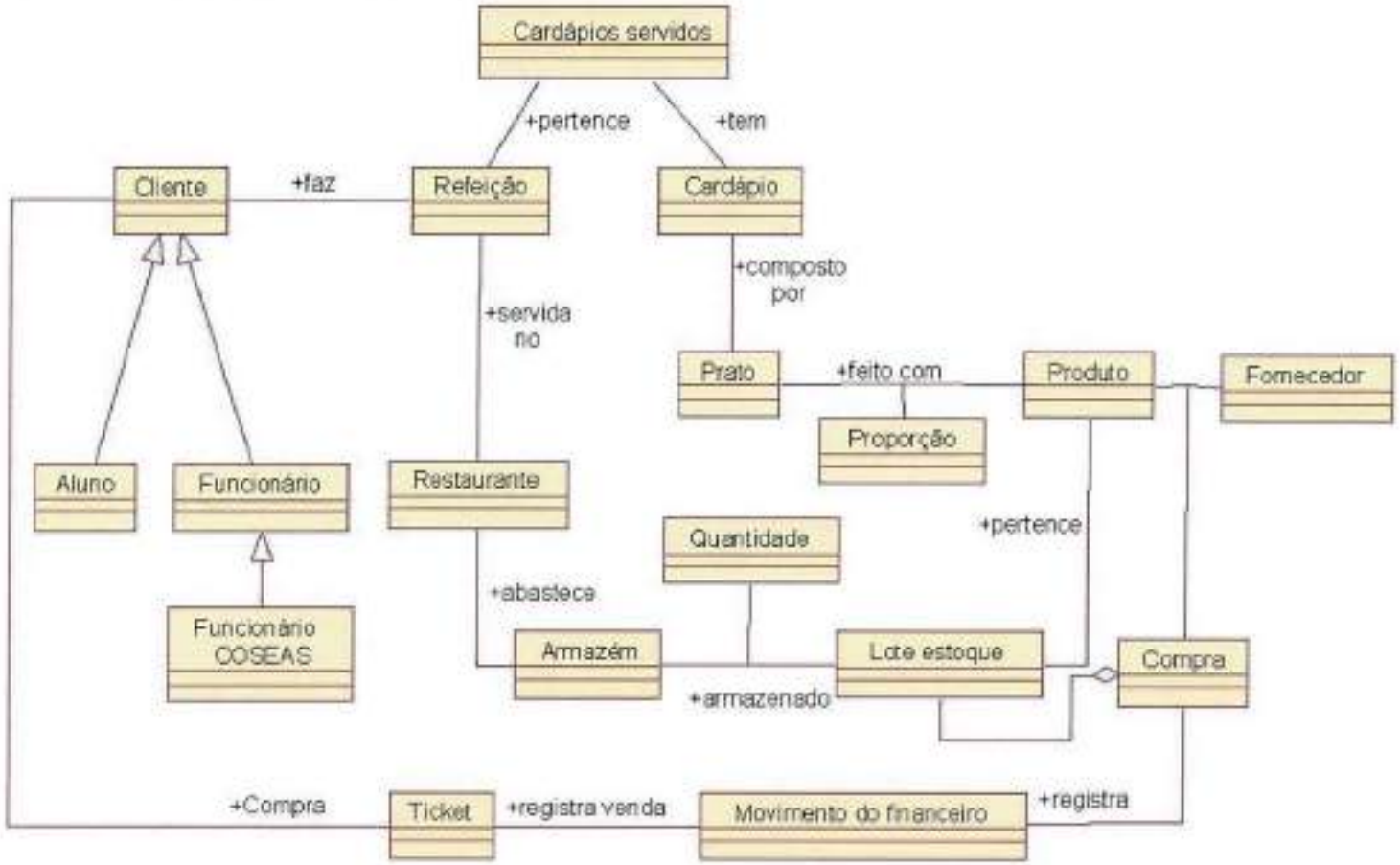
Pós-condição: usuário *logado*.

Cenário Secundário:

1. Usuário fornece dados inconsistentes (Passo 2): sistema exibe mensagem de erro e retorna ao Passo 1.

Inclusão: Nenhuma.

9 ANEXO C– DIAGRAMA DE CLASSES



10 REFERÊNCIAS BIBLIOGRÁFICAS

[1] BPMI, Business Process Management Initiative. *Business Process Modeling Notation (BPMN)*. Disponível em: <http://www.bpmn.org/Documents/BPMN% 20V1-0%20May%203%202004.pdf>. Acesso em: 15 de mar. 2005

[2] BOND, Martin et al. *Aprenda J2EE em 21 dias*. São Paulo: Editora Makron Books, 2003

[3] CRAWFORD, William. *Essential Multitier J2EE Design Patterns (2004)*. Disponível em <http://www.oracle.com/technology/oramag/oracle/03-jan/ o13j2ee.html>. Acesso em: 07 de abr. 2005

[4] KURNIAWAN, Budi. *Almost All Java Web Apps Need Model 2*. Disponível em: <http://www.fawcette.com/javapro/2002_06/online/servlets_06_11_02>. Acesso em: 17 de abr. 2005

[5] MILLS, Duncan. *J2EE Design Patterns: Understanding MVC*. Disponível em: <http://otn.oracle.com/oramag/webcolumns/2003/techarticles/mills_mvc.html>. Acesso em: 18 de abr. 2005

[6] SUN MICROSYSTEMS, *Designing Enterprise Applications with the J2EE™ Platform, Second Edition (2002)*. Disponível em: <http://java.sun.com/blueprints/guidelines/ designing_enterprise_applications_2e>. Acesso em: 18 de abr. 2005

[7] ARMSTRONG, Eric and BALL, Jennifer. *The J2EE (TM) 1.4 Tutorial*. Disponível em: <http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html>. Acesso em: 18 abr. 2005

[8] SUN MICROSYSTEMS, *JavaBeans*. Disponível em: <http://java.sun.com/ products/javabeans/index.jsp>. Acesso em 20 abr. 2005

[9] HIBERNATE. *Site Oficial do Projeto Hibernate*. Disponível em: <http://www.hibernate.org>. Acesso em: 20 de abr. 2005.

[10] APACHE SOFTWARE FOUNDATION. *Site Oficial do Jakarta Struts Framework*. Disponível em: <http://jakarta.apache.org/struts/>. Acesso em: 21 de abr. 2005

[11] MEMBERS JBOSS. *Site Oficial do Projeto Container JBoss*. Disponível em: <http://www.jboss.org/index.html>. Acesso em: 22 de abr. 2005.

[12] EVERETT, David B. *Smart Card Technology: Introduction To Smart Cards*. Disponível em <http://www.smartcard.co.uk/tech1.html>. Acesso em: 16 de mai. 2005.

[13] MYSQL AB. Disponível em: <http://www.mysql.com>. Acesso em: 17 de mai. 2005

[14] MYSQL REFERENCE MANUAL. Disponível em: <http://dev.mysql.com/doc/>. Acesso em: 17 de mai. 2005

[15] SOARES, W. *MySQL: Conceitos e Aplicações*. São Paulo: Érica, 2001

[16] APACHE SOFTWARE FOUNDATION. *Site Oficial do Projeto Tomcat*. Disponível em: <http://jakarta.apache.org/tomcat/>. Acesso em 20 de mai. 2005

[17] SOUZA, Welington. *Struts Tutorial* . Disponível em: <http://www.portaljava.com.br/home/modules.php?name=Content&pa=showpage&pid=63>. Acesso em: 25 de mai. 2005

[18] DEITEL, H. M.; DEITEL, P. J. *Java Como Programar*. 4 Ed., São Paulo: Bookman, 2002

[19] PRESSMAN, R. S. Engenharia de Software. 5. Ed., Rio de Janeiro: McGrawHill, 2002.

[20] CULPEPPER, M. *JBoss Eclipse IDE*. Disponível em: <http://www.jboss.org/products/Jbosside>. Acesso em: 10 de jul. 2005

[21] ECLIPSE FOUNDATION. *Site Oficial do Projeto Eclipse*. Disponível em: <http://www.eclipse.org/>. Acesso em 10 de mar. 2005

[22] SHILLINGTON, N; WAKER, T. *The design of a Smart Card Interface Device*. Disponível em: < http://www.cs.uct.ac.za/Research/DNA/SOCS/rchap1.html>. Acesso em 20 de jun. 2005