UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

SISTEMA DE AUXÍLIO À MATRÍCULA DE ALUNOS UTILIZANDO *JAVA 2 ENTERPRISE EDITION*

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

ROGÉRIO SORROCHE

BLUMENAU, NOVEMBRO/2002

2002/2-55

SISTEMA DE AUXÍLIO À MATRÍCULA DE ALUNOS UTILIZANDO *JAVA 2 ENTERPRISE EDITION*

ROGÉRIO SORROCHE

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Maurício Capobianco Lopes — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Maurício Capobianco Lopes

_____

Prof. Marcel Hugo

_____

Prof. Paulo César Rodacki Gomes

AGRADECIMENTOS

Agradeço, principalmente, a Deus por todo seu amor, pela vida e por todas as suas bênçãos e dádivas.

Agradecimentos aos meus amigos que incentivaram na caminhada em busca deste título.

Ao Núcleo de Informática, dirigido pelo professor Marcel Hugo, pelo incentivo, apoio e cooperação, provendo a estrutura, o treinamento e todo o necessário para que o desenvolvimento deste trabalho fosse realizado com sucesso.

À Universidade Regional de Blumenau, instituição na qual sou funcionário, espero que este trabalho possa ser útil no auxílio de seus alunos.

A todos meus professores, principalmente, ao orientador professor Maurício Capobianco Lopes pelo seu apoio, incentivo e paciência.

Aos amigos Mauro Schramm e Emerson de Pinho Adam, companheiros de trabalho que prestaram valiosa ajuda no desenvolvimento deste trabalho.

A todos que direta ou indiretamente me ajudaram neste trabalho.

RESUMO

Este trabalho apresenta o desenvolvimento e a implementação de um software de auxílio à matrícula de alunos na Universidade Regional de Blumenau. No processo de matrícula, os alunos que pretendem cursar disciplinas em diversas fases, não seguindo a grade curricular normal, deverão ser auxiliados por este software para escolher as disciplinas de acordo com o seu quadro de horários e a grade curricular de seu curso. No modelo desenvolvido, o aluno pode optar por tentar alocar o máximo de disciplinas ou atingir o mínimo de créditos que terá que pagar. Para a implementação foi utilizada uma técnica de alocação baseada na Teoria dos Grafos. A aplicação foi implementada utilizando a tecnologia *Java 2 Enterprise Edition*.

ABSTRACT

This work presents the development and the implementation of a help software to register students in the Universidade Regional de Blumenau. On register process, the students who intend to study subjects in several levels, not following the normal subjects table, will be helped by this software to choose the subjects in accordance with yours time table and the course subjects table. In the developed model, the student can choose between taking the maximum number of subjects or reach the minimum number of credits to be paid. For the implementation it was used an allocation technique based on the Graph Theory. The application was implemented using *Java 2 Enterprise Edition*.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

1 INTRODUÇÃO

Semestralmente a Universidade Regional de Blumenau (FURB) disponibiliza aos seus alunos o procedimento de reserva de vaga. Este procedimento serve como uma pré-matrícula, onde o aluno escolhe as disciplinas que pretende cursar no próximo semestre. A reserva de vaga pode ser feita através da internet, pois já existe um sistema automatizado feito em *Java Applets*. Este *applet* mostra uma lista de todas as disciplinas que são oferecidas ao curso em que o aluno está matriculado, agrupadas por semestre. No sistema, o aluno escolhe a disciplina desejada e automaticamente é mostrada uma grade com os horários semanais da disciplina. Se houver coincidência de horário o mesmo é mostrado com uma cor diferente e uma mensagem ao aluno é emitida, caso contrário, a disciplina pode ser incluída pelo aluno em sua grade.

Para alunos considerados regulares, isto é, que seguiram o currículo do curso sem reprovações, o procedimento funciona muito bem. Entretanto, esta não é a regra e muitos alunos têm pelo menos uma ou duas disciplinas nas quais não obtiveram aprovação. Para agravar esta situação, muitas disciplinas são pré-requisitos de outras, ou seja, o aluno tem que obter aprovação nestas disciplinas antes de cursar as outras. Por isso torna-se difícil para o aluno reservar as disciplinas que se encaixem no horário de maneira satisfatória, pois ele tem que passar semestre por semestre, disciplina por disciplina, para verificar se alguma disciplina se encaixa no horário que ele tem disponível. Na grande maioria das vezes ele acaba não fazendo isso.

Neste contexto, surge a necessidade de um sistema automatizado que elabore sugestões de horário na reserva de vaga. Isto envolve selecionar as disciplinas que o aluno está apto a cursar, verificar a disponibilidade de horários do aluno, e, então, elaborar sugestões de horário. Deve-se levar em conta algumas restrições que o próprio aluno pode informar como, por exemplo, dar mais prioridade a uma disciplina do que a outra, ou ainda o fato de que o aluno tem um mínimo de créditos a pagar todo semestre, e muitas vezes não consegue reservar as disciplinas de modo que este mínimo seja atendido. Assim, o sistema poderia beneficiar tanto o aluno, permitindo a ele um melhor aproveitamento de sua grade de horários, quanto à instituição, que teria um maior retorno financeiro, uma vez que o aluno estaria fazendo um maior número de disciplinas e, conseqüentemente, reduzindo o seu tempo de curso.

Problemas de arranjo e otimização como o citado acima, recebem a designação genérica de problemas de *timetabling*, que constituem em construir quadros de horários para uma série de atividades atendendo a um determinado conjunto de restrições. Os problemas de *timetabling* são, em geral, classificados quanto a sua complexidade como problemas pertencentes à classe *NP-Difícil*, ou seja, nem sempre possuem algoritmos eficientes para sua execução em tempos de processamento polinomiais. Em outras palavras, a computação do algoritmo, para um determinado problema, cresce exponencialmente em função do tamanho da instância (Braz, 2000).

Para resolver problemas como estes, surgiram os métodos heurísticos ou aproximativos, definidos, segundo Goldbarg (2000), como algoritmos de busca de soluções em que não existe qualquer garantia de sucesso. Muitos métodos heurísticos já foram aplicados para problemas de *timetabling*, dentre os quais pode-se citar: *Algoritmos Genéticos* (Braz, 2000) e (Lucas, 2000), *Tabu Search* (Hertz, 1992), *Simulated Anneling* (Thompson, 1996) e *Teoria dos Grafos* (Schwarz, 1990).

Neste trabalho foi utilizado um modelo da Teoria dos Grafos, que utiliza o conceito de conjuntos independentes maximais, baseado no trabalho de Schwarz (1990), que aplicou o método para a elaboração de grade de horários em instituições de ensino.

Para a implementação desta aplicação foram utilizadas as tecnologias definidas pela plataforma *Java 2 Enterprise Edition* (*J2EE*) que segundo Sun (2002b), define um padrão para o desenvolvimento de aplicações multicamadas. Nesta arquitetura multicamadas, a camada que contém as regras de negócio, implementada utilizando *Enterprise JavaBeans*, pode ficar concentrada no servidor de aplicações, sendo compartilhada com diversas aplicações clientes. As aplicações clientes não contêm a lógica do negócio, atendo-se somente à camada de apresentação. Na camada de apresentação foram utilizadas as tecnologias de *Servlets* e *JavaServer Pages*.

Segundo Marinescu (2002), sem um conjunto de boas práticas de modelagem, o desenvolvimento utilizando a arquitetura multicamadas *J2EE*, pode se tornar muito difícil. Os desenvolvedores acabam reinventando a roda, ou simplesmente, cometendo custosos erros de projeto. Segundo Alur (2002), uma boa maneira de adquirir experiência em projeto é pela utilização de padrões de projeto. Os padrões de projetos constituem um novo mecanismo para

expressar experiências na elaboração de projetos orientados a objetos. Marinescu (2002) caracteriza um padrão de projeto como uma solução utilizando boas práticas para um problema comum recorrente. Os padrões de projeto descrevem um problema e a sua solução formando assim, uma linguagem comum, que permite a troca de experiências.

Sendo assim, neste trabalho é apresentado um sistema de auxílio à matrícula de alunos, desenvolvido utilizando-se uma arquitetura multicamadas baseado no J2EE e seguindo alguns padrões de projeto específicos para esta arquitetura.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um protótipo de uma aplicação para a elaboração de sugestões de horários na reserva de vaga da Universidade Regional de Blumenau utilizando *Java 2 Enterprise Edition*.

Como objetivos específicos do trabalho destacam-se:

a) implementar um algoritmo para a elaboração de várias sugestões para a reserva de vaga de um aluno;
b) disponibilizar uma interface *Web*, para a aplicação;
c) verificar os pontos fortes e fracos da tecnologia *J2EE* no desenvolvimento de aplicações.

1.2 ESTRUTURA DO TRABALHO

O primeiro capítulo do trabalho apresenta sua introdução, destacando a contextualização e justificativas, bem como seus objetivos.

O segundo capítulo aborda a teoria dos grafos falando de seus conceitos e características. Também apresenta algoritmos de particionamento de grafos em conjuntos independentes.

No terceiro capítulo são apresentadas as tecnologias de *Enterprise JavaBeans*, *Servlets* e *JavaServer Pages*.

No quarto capítulo são apresentados os conceitos e fundamentos sobre padrões de projeto e também são demonstrados os padrões utilizados neste trabalho.

O quinto capítulo apresenta o desenvolvimento do trabalho, mostrando os diagramas de classe, casos de uso e diagramas de seqüência. É apresentado também o problema da geração de sugestões e a definição do grafo para o problema.

O sexto capítulo apresenta as conclusões, dificuldades encontradas no decorrer do desenvolvimento do trabalho e sugestões para futuras implementações e extensões deste trabalho.

2 GRAFOS

Neste capítulo são abordados os principais conceitos da teoria dos grafos utilizados na implementação do trabalho. Ainda são descritos os algoritmos que serviram como base para a solução implementada.

2.1 DEFINIÇÕES

Segundo Gross (1998), configurações de nós e conexões podem ocorrer em uma grande diversidade de aplicações. Elas podem representar redes físicas, como circuitos elétricos, rodovias ou moléculas orgânicas, e também podem ser usadas para representar interações menos tangíveis, como relacionamentos sociais, bases de dados ou no fluxo de controle em um programa de computador.

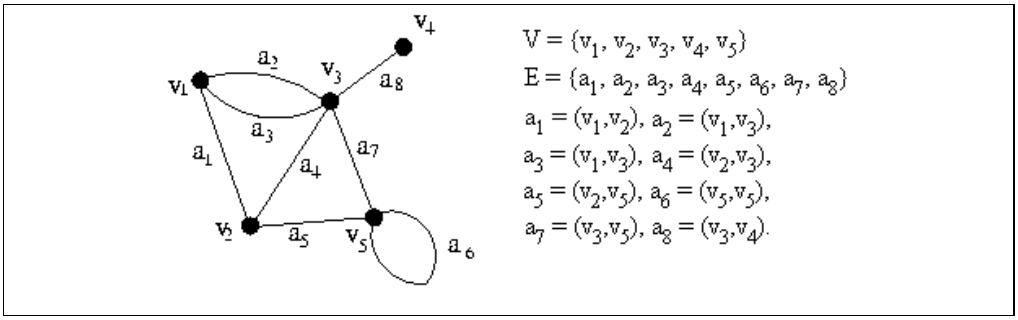
Formalmente, estas configurações são modeladas por estruturas combinatoriais chamadas grafos, consistindo de dois conjuntos chamados vértices e arestas e uma relação de incidência sobre elas. Grafos são modelos altamente versáteis para analisar uma ampla gama de problemas práticos em que pontos e conexões entre eles podem ter alguma interpretação física ou conceitual (Gross, 1998). No caso do problema em questão neste trabalho eles são especialmente úteis para representar conjuntos maximais independentes que possam vir a solucionar a alocação das disciplinas baseadas em seus horários.

Segundo Szwarcfiter (1984), um grafo *G(V, E)* é definido com um conjunto não vazio *V* e um conjunto *E* de pares não ordenados de elementos distintos de *V*. Os elementos de *V* são os *vértices* e os de *E* são as *arestas* de *G*. Cada aresta $e \in E$ é denotada pelo par de vértices $e = (v, w)$ que a forma. Nesse caso, os vértices *v* e *w* são os extremos da aresta *e*, sendo denominados adjacentes. A aresta *e* é incidente a ambos *v* e *w*.

A Figura 1 mostra uma *representação geométrica* de um grafo na qual seus vértices correspondem a pontos distintos do plano, enquanto que cada aresta (*v*, *w*) é representada por uma linha unindo os pontos *v* e *w*.

Formatado

Figura 1 – Grafo e sua representação geométrica



Fonte: Baseado em Gagnon (2001).

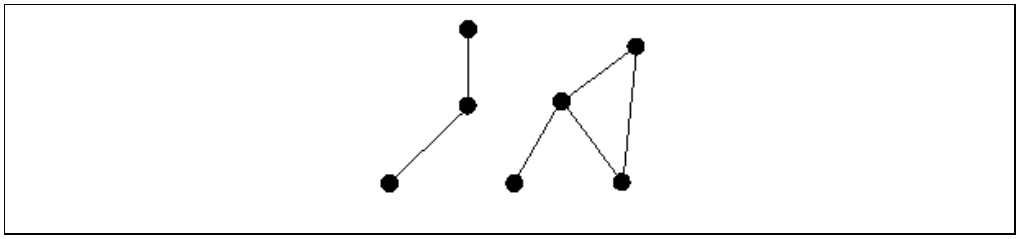
Seja dois vértices *v* e *w*, e uma aresta *e = (v, w)*. A aresta *e* é dita *incidente* a ambos vértices *v* e *w*. Duas arestas não paralelas que são incidentes a um mesmo vértice são ditas *adjacentes*. Dois vértices que são ligados por uma mesma aresta, também são ditos adjacentes. O número de arestas incidentes a um vértice *v* é chamado o *grau* do vértice *v*, denotado por *gr(v)* (Gagnon, 2001).

Uma aresta do tipo *e = (v, v)*, isto é, formada por um par de vértices idênticos é chamada *laço*. Duas arestas que são incidentes ao mesmo par de vértices, são chamadas *paralelas*. Um grafo que possua arestas paralelas é chamado *multigrafo*. Caso contrário, será um grafo simples (Szwarcfiter, 1984).

Existem vários tipos de grafos, como por exemplo, os dígrafos ou dirigidos e os valorados. Neste trabalho, entretanto, será dada especial atenção aos completos, conexos e complementares.

Um grafo *G(V, E)*, é dito *conexo* quando existe pelo menos um caminho entre cada par de vértices de *G*. Caso contrário *G* é desconexo (Figura 2). Um grafo será *totalmente desconexo* quando não possuir arestas (Szwarcfiter, 1984).

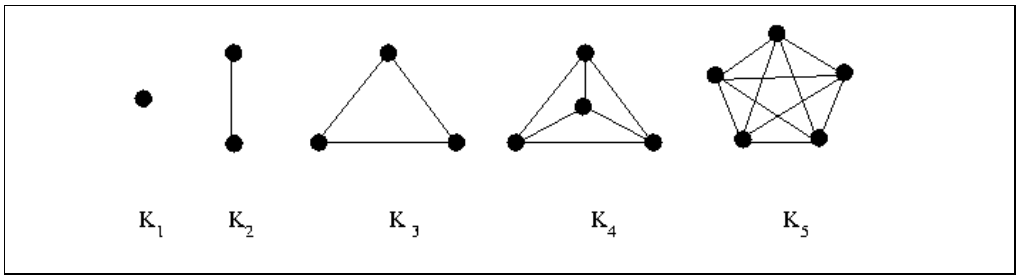
Figura 2- Grafo desconexo



Fonte: Gagnon (2001).

Segundo Szwarcfiter (1984), um grafo é *completo* quando existe uma aresta entre cada par de seus vértices (Figura 3). Utiliza-se a notação K_n, para designar um grafo completo com n vértices. O grafo simples K_n possui o número máximo possível de arestas para um dado n.

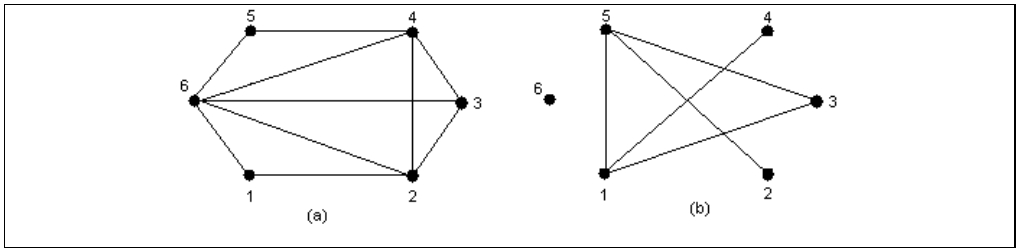
Figura 3 – Grafos completos



Fonte: Gagnon (2001).

Um grafo *G'* é dito *complementar* de *G* se possui o mesmo conjunto de vértices de *G* e tal que para todo par de vértices distintos $v, w \in V$, tem-se que *(v, w)* é uma aresta de *G'* se e somente se não for de *G* (Rabuske, 1992). A Figura 4 mostra um grafo e seu complemento.

Figura 4 – Um grafo e seu complemento



Fonte: (Szwarcfiter, 1984).

2.2 SUBGRAFOS

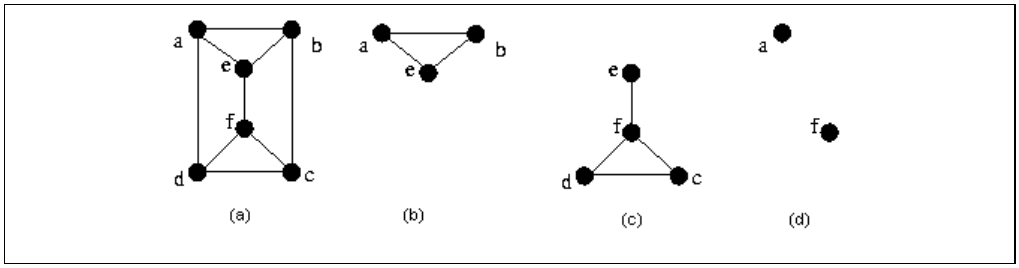
Segundo Szwarcfiter (1984), um *subgrafo* $G_2(V_2, E_2)$ de um grafo $G_1(V_1, E_1)$ é um grafo tal que $V_2 \subseteq V_1$ e $E_2 \subseteq E_1$. Se além disso, G_2 possuir toda aresta (v, w) de G_1 tal que ambos v e w estejam em V_2, então G_2 é o *subgrafo induzido pelo subconjunto de vértices* V_2.

Existem dois tipos de subgrafos que apresentam propriedades interessantes. Denomina-se *clique* de um grafo G a um subgrafo de G que seja completo. Chama-se *conjunto independente de vértices* a um subgrafo induzido de G, que seja totalmente desconexo. Ou seja, numa clique existe uma aresta entre qualquer par de vértices. Num conjunto independente de vértices não há aresta entre qualquer par de vértices. O tamanho de uma clique ou conjunto independente de vértices é igual à cardinalidade de seu subconjunto de vértices.

Considerando-se o grafo apresentado na Figura 5(a), o subgrafo ilustrado na Figura 5(b) é uma clique, na Figura 5(c) o subgrafo apresentado é um grafo induzido pelos vértices *c*, *d*, *e* e *f*, e o grafo da Figura 5(d) é um conjunto independente de vértices (Szwarcfiter, 1984).

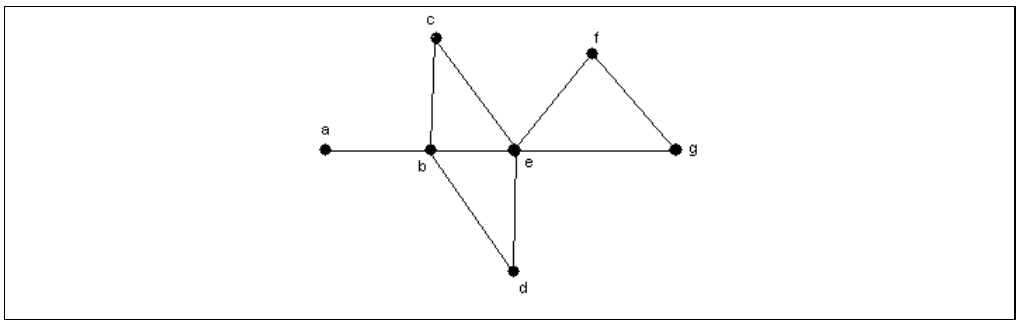
Formatado

Figura 5 – Subgrafos



De acordo com Rabuske (1992), um *conjunto independente maximal* é um conjunto independente para o qual nenhum outro vértice pode ser adicionado sem que destrua a propriedade de independência. Os conjuntos *{a, c, d, f}* e *{b, g}* na Figura 6, são conjuntos independentes maximais. Um grafo qualquer pode conter muitos conjuntos independentes maximais e eles podem ser de diferentes tamanhos. Dentre os conjuntos independentes maximais, normalmente, o que possui maior número de vértices é o de maior interesse.

Figura 6 – Conjuntos independentes maximais



O número de vértices do maior conjunto independente de um grafo *G* é chamado *número independente* ou *coeficiente de estabilidade interna*, *NI(G)*.

2.2.1 MÉTODOS PARA ACHAR CONJUNTOS INDEPENDENTES

Segundo Rabuske (1992), existem diversos métodos para encontrar conjuntos independentes. Muitas vezes porém o problema é apresentado como sendo o de determinação das cliques maximais de um grafo, pois a cada conjunto independente de um grafo *G* corresponde a uma clique maximal de um grafo complementar *G'*.

A seguir estão descritos dois algoritmos que podem ser utilizados para encontrar conjuntos independentes em um grafo. A técnica do *algoritmo guloso* possui como característica importante a simplicidade e pode se utilizada para uma infinidade de problemas. O algoritmo de Bron e Kerbosh foi proposto por Bron (1973), como um método para encontrar todas as cliques de um grafo e adaptado por Schwarz (1990) para encontrar os conjuntos independentes maximais.

2.2.1.1 ALGORITMO GULOSO

Segundo Szwarcfiter (1984), a denominação algoritmo guloso provém do fato de que a cada passo procura-se incorporar à solução até então construída, a melhor porção possível, compatível com algum critério especificado. O algoritmo é descrito a seguir.

Dado um conjunto S, deseja-se determinar um subconjunto $S' \subseteq S$ tal que:

a) S' satisfaz uma propriedade P;
b) S' é máximo ou mínimo em relação a algum critério α;

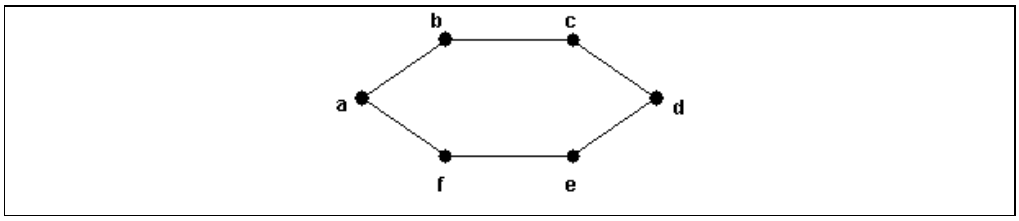
O algoritmo guloso para resolver este problema consiste num processo iterativo em que S' é construído, adicionando-se ao mesmo elementos de S, um a um. Isto é, a cada passo, determina-se o elemento $s \in S$ que adicionado a S' maximiza ou minimiza α.

Este processo garante que o subconjunto S' obtido satisfaz P, visto que esta condição é verificada passo a passo, no algoritmo. Contudo, há aplicações em que não é possível garantir a maximalidade ou minimalidade do S' obtido. Assim sendo, para a aplicação desse método, é necessária uma prova de que o subconjunto obtido seria de fato máximo ou mínimo (Szwarcfiter, 1984).

Rabuske (1992) dá a definição de um algoritmo guloso para encontrar conjuntos independentes em um grafo $G(V, E)$, em que a cada passo seleciona-se um vértice de V que não seja adjacente a nenhum dos vértices do conjunto S' sendo formado. Neste processo, para determinar se o conjunto independente S' formado é maximal, é necessário verificar se S' não está contido em nenhum outro conjunto independente formado anteriormente.

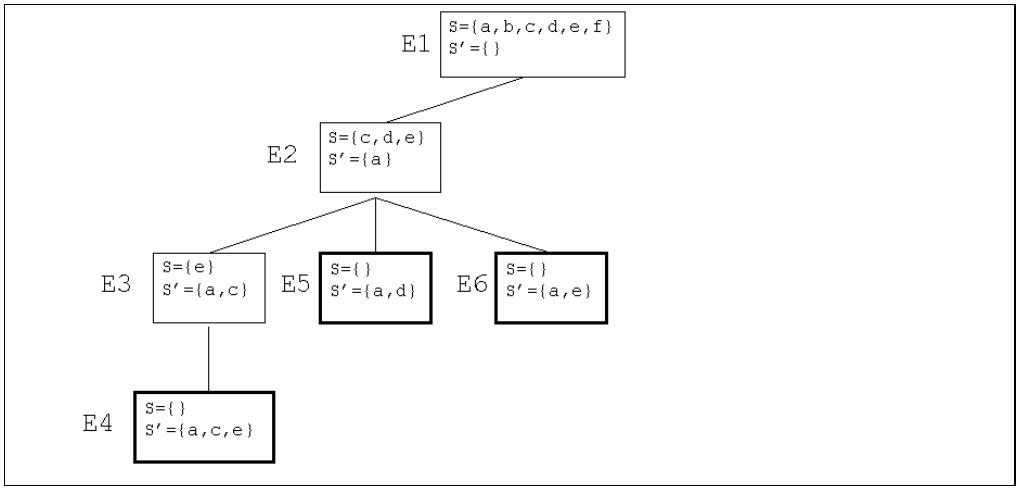
Segue abaixo o desenvolvimento passo a passo do algoritmo guloso considerando-se o grafo apresentado na Figura 7. Deseja-se encontrar os conjuntos independentes que podem ser formados a partir deste grafo, escolhendo-se os vértices na ordem alfabética.

Figura 7 – Grafo exemplo



A Figura 8 apresenta os estados do algoritmo a cada passo executado.

Figura 8 – Desenvolvimento do Algoritmo Guloso



Inicialmente cria-se o estado inicial *E1*, onde o conjunto *S* contém todos os vértices do grafo que podem ser adicionados à solução. O conjunto solução *S'* inicialmente está vazio. Escolhendo-se o vértice *a* e adicionando-o ao conjunto *S'* elimina-se de *S* todos os vértices adjacentes ao vértice *a* (estado *E2*). Este passo é executado até que o conjunto *S* esteja vazio, quando uma nova solução estará formada (estado *E4*). Para gerar os próximos conjuntos independentes, o algoritmo tem que recuperar os estados anteriores da busca (*backtracking*) e tentar alocar outro vértice chegando-se assim aos estados *E5* e *E6* que contém novas soluções.

Pode-se notar que o conjunto independente *S'={a, e}* formado no estado *E6*, não é maximal pois está contido no conjunto *S'={a, c, e}* formado no estado *E4*.

2.2.1.2 ALGORITMO DE BRON E KERBOSH

Este algoritmo é baseado no modelo proposto por Bron e Kerbosh em Bron (1973) que consiste em gerar sistematicamente todas as cliques de um grafo completo formando uma árvore. Schwarz (1990) propôs uma variação que foi utilizada no problema da geração de horários, para encontrar conjuntos independentes maximais. O algoritmo em essência é o mesmo, mas é aplicado sobre o grafo complementar em relação ao grafo completo. Segue a descrição do algoritmo (Schwarz, 1990) e (Furtado, 1973).

Neste algoritmo são considerados três conjuntos:

a) o conjunto *VI* de vértices independentes;
b) o conjunto *CAND* dos vértices candidatos a exame para possível inclusão em *VI*;
c) o conjunto *ANT* dos vértices que já entraram em alguma configuração anterior de *VI*.

O algoritmo usa um operador de extensão cuja finalidade é gerar todas as extensões possíveis de uma dada configuração de *VI* a partir de vértices em *CAND*, mas sem incluir os vértices de *ANT*, pois se considera que configurações de *VI* contendo estes vértices já terão sido geradas anteriormente.

O operador de extensão é recursivo e consiste em:

a) selecionar um candidato por algum critério;
b) adicioná-lo a *VI*;
c) criar novos conjuntos *CAND* e *ANT* a partir dos antigos, removendo todos os pontos adjacentes ao candidato selecionado e o próprio candidato. Os conjuntos antigos são salvos em uma pilha;
d) chamar o operador de extensão sobre estes novos conjuntos;
e) regressando da chamada do operador, restaurar os antigos conjuntos *CAND* e *ANT*, remover o candidato de *CAND* e colocá-lo em *ANT*.

Um novo conjunto independente maximal está formado quando os dois conjuntos *CAND* e *ANT* estiverem vazios. Se *CAND* está vazio significa que não há mais vértices

candidatos para estenderem esta solução. A condição de *ANT* vazio também é necessária, pois se algum vértice ainda foi não retirado de *ANT*, isso indica que nenhum vértice em *VI* é adjacente a ele, ou seja, este vértice faz parte do conjunto independente e como está em *ANT* significa que já foi gerado anteriormente. Portanto, equivale a se dizer que o conjunto *VI* está contido em outro conjunto que já foi gerado e por isso não pode ser maximal.

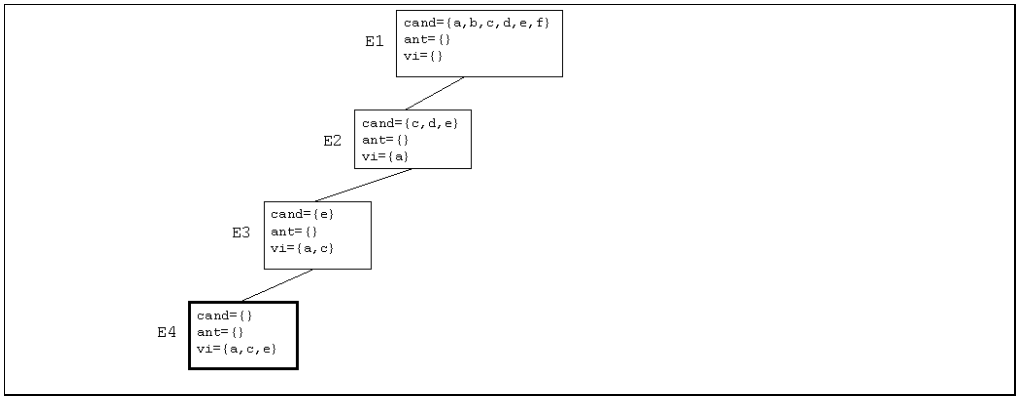
É possível determinar então casos em que uma configuração de *VI* não pode conduzir a um conjunto maximal, verificando que não é possível fazer com que o conjunto *ANT* fique vazio. Para isso basta haver um vértice em *ANT* que não seja adjacente a nenhum dos outros vértices em *CAND*, pois assim nenhum dos candidatos que se possa selecionar removeria esse vértice de *ANT*.

Este procedimento caracteriza o algoritmo como do tipo *Branch and Bound*, em que se tenta várias alternativas abandonando-as a partir do ponto em que se pode prever um insucesso.

A seguir está demonstrada a execução do algoritmo de Bron e Kerbosh para encontrar os conjuntos independentes maximais para o grafo da Figura 7, utilizando como critério de escolha dos vértices a ordem alfabética das letras associadas a cada vértice.

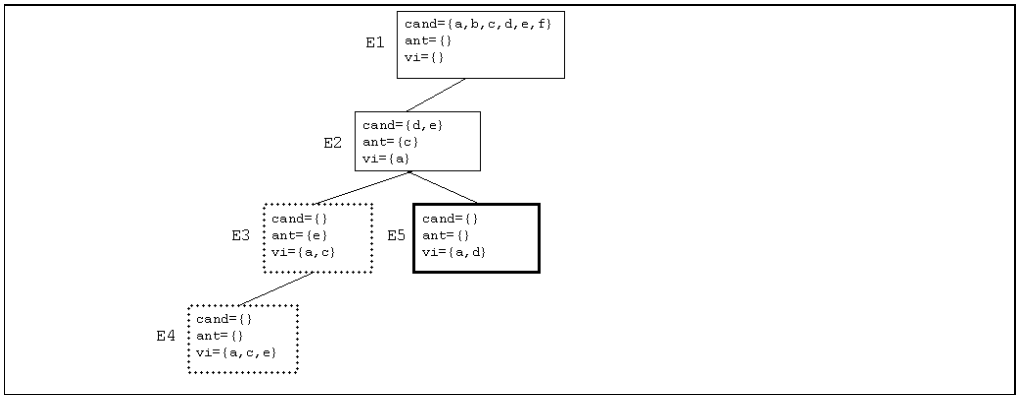
A Figura 9 apresenta os passos do algoritmo até a geração da primeira sugestão. Inicialmente tem-se o estado inicial *E1*, onde *CAND* contém todos os vértices do grafo e *ANT* e *VI* estão vazios. Chama-se então o operador de extensão, que seleciona o vértice *a* de *CAND* e adiciona-o a *VI*. Os conjuntos *CAND* e *ANT* são duplicados e é removido de *CAND* o próprio *a* e os vértices *b* e *c*, que são adjacentes ao *a*. Os conjuntos *CAND* e *ANT* antigos são salvos em uma pilha. O operador de extensão é chamado então recursivamente até que *CAND* esteja vazio. Como *ANT* também estará vazio indica que uma nova solução viável estará formada (estado *E4*).

Figura 9 – Geração da primeira solução do algoritmo de Bron e Kerbosh



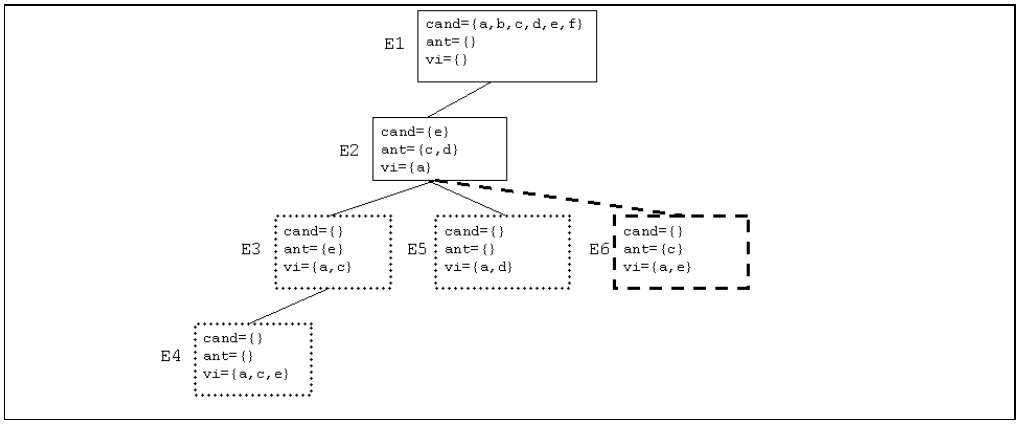
Após gerada a primeira solução o operador de extensão retrocede do estado *E4* para o estado *E3*. O vértice *e* será removido de *CAND* e adicionado a *ANT* conforme apresentado na Figura 10. Neste ponto *CAND* estará vazio, o que indica que não é mais possível estender esta solução. Mas como *ANT* não está mais vazio indica que não é uma solução viável, obrigando o operador de extensão a retroceder para o estado *E2*. Restaurados os conjuntos *CAND* e *ANT*, o vértice *c* é removido de *CAND* e adicionado a *ANT*. Neste momento *CAND* conterá os vértice *d* e *e*, podendo ser novamente estendido. Chamando-se o operador de extensão, será selecionado o vértice *d* e adicionado a *VI* e remove-se de *CAND* e de *ANT* os vértices adjacentes ao *d* tornando estes conjuntos vazios. Chega-se assim ao novo conjunto independente {*a*, *d*} (estado *E5*).

Figura 10 – Retorno do operador de extensão e geração de nova sugestão



Do estado *E5* o operador de extensão retrocede novamente para o estado *E2*, retirando o vértice *d* de *CAND* e adicionando-o a *ANT*, restando somente o vértice *e* em *CAND* (Figura 11). Deste ponto verifica-se que não é possível gerar um novo conjunto independente maximal, pois se for chamado o operador de extensão para adicionar o vértice *e* em *VI*, este não irá remover de *ANT* o vértice *c*, pois *c* não é adjacente a *e*.

Figura 11 – Condição limite do algoritmo de Bron e Kerbosh



A partir daí o algoritmo retorna ao estado *E1* e continua sua execução, retirando o vértice *a* de *CAND* e colocando-o em *ANT*, selecionando o vértice *b* de CAND para gerar novas soluções.

3 JAVA 2 ENTERPRISE EDITION (J2EE)

Neste capítulo será abordada a tecnologia utilizada para o desenvolvimento do trabalho, os quais são *Enterprise JavaBeans*, que são componentes que implementam regras de negócio, *Servlets* e *JavaServer Pages*, que são classificados como componentes *Web*. Todas estas tecnologias fazem parte da plataforma de desenvolvimento *Java 2 Enterprise Edition* (*J2EE*) que segundo (Sun, 2002a) usa um modelo de aplicações multicamadas distribuído.

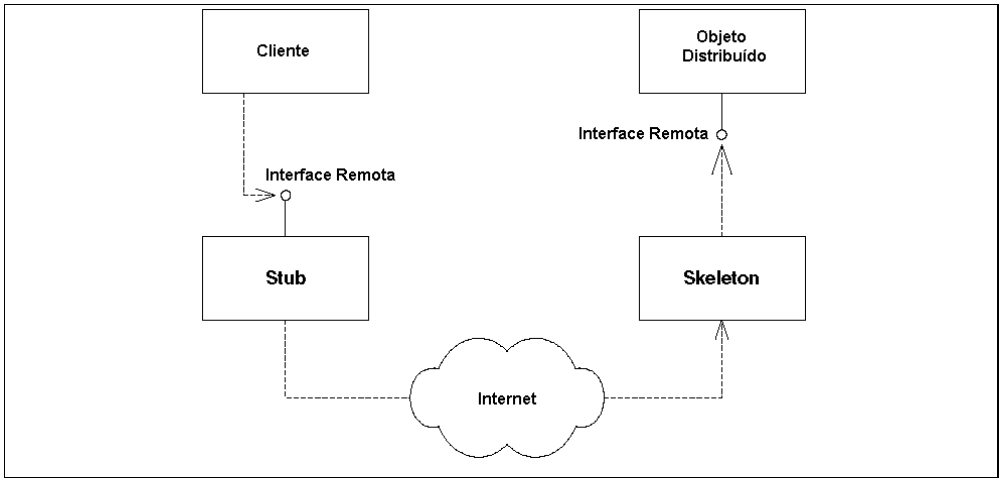
3.1 ENTERPRISE JAVABEANS

De acordo com Roman (2002), *Enterprise JavaBeans* (*EJB*) são especificamente usados para resolver problemas de negócio. O padrão *EJB* é uma arquitetura de componentes que podem ser implantados em um ambiente distribuído. Ele especifica um contrato entre os componentes e os servidores de aplicação, de modo que um componente *EJB* pode ser implantado em qualquer servidor de aplicações que suporte *EJB*. Os *EJB* também referidos como *enterprise beans*, segundo Sun (2002b), são escaláveis, transacionais e multi-usuários.

3.1.1 FUNCIONAMENTO

Segundo Haefel (2001), EJB é um modelo de componentes baseado em objetos distribuídos. As arquiteturas de objetos distribuídos são baseadas em uma camada de comunicação de redes. A Figura 12 mostra como um cliente interage com um objeto distribuído:

Figura 12 – Objetos Distribuídos



Fonte: Roman (2002).

Essencialmente existem três partes nesta arquitetura: o objeto distribuído ou de negócio, o objeto *skeleton* e o objeto *stub*. Neste modelo o cliente chama o *stub*, que é um objeto que está no cliente, e que mascara a comunicação da rede para o cliente. O *stub* sabe como se comunicar sobre a rede, convertendo os objetos e parâmetros se necessário. O *stub* se comunica através da rede com o *skeleton*, que é um objeto que reside no servidor. O *skeleton* tem a responsabilidade de mascarar a comunicação da rede para o objeto distribuído. Ele conhece o protocolo de comunicação entre o *stub* e ele próprio, e sabe como converter os parâmetros passados através da rede em objetos. O *skeleton* então delega a chamada para o objeto distribuído, que faz o trabalho e retorna para o *skeleton*, que por sua vez retorna para o *stub*, que devolve o controle para o cliente (Roman, 2002).

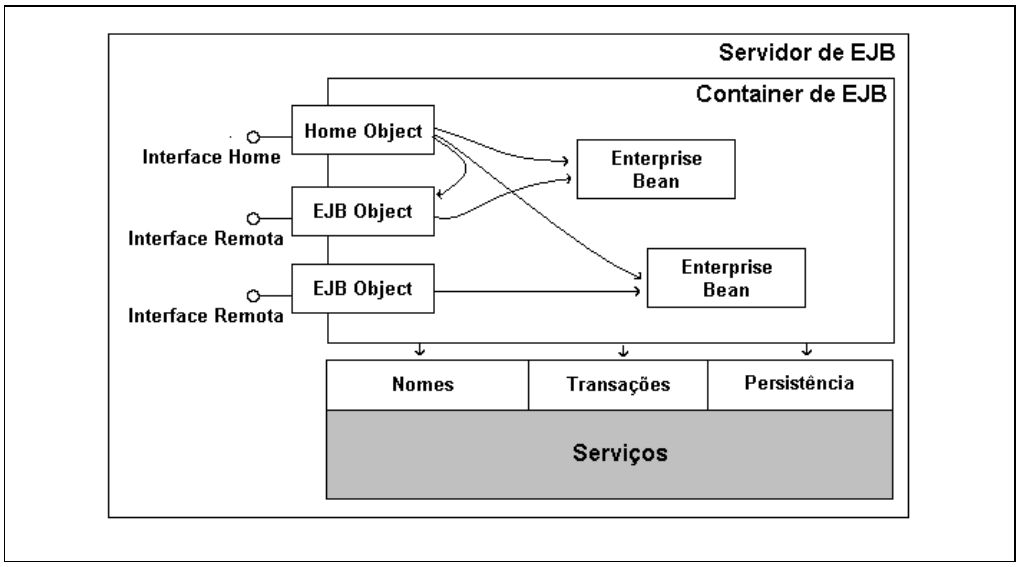
Um ponto chave é que o *stub* e o objeto distribuído implementam a mesma interface. Isto significa que o *stub* tem os mesmos métodos que o objeto distribuído. O cliente, chamando um método no *stub*, tem a impressão de estar acessando o objeto distribuído diretamente, mas na realidade, o cliente está chamando um objeto que representa o objeto distribuído e que sabe se comunicar através da rede com este objeto (Roman, 2002).

3.1.2 ARQUITETURA

Segundo Seshadri (1999), os elementos definidos pela arquitetura de *EJB* são ilustrados na Figura 13 e consistem de:

a) um servidor de *EJB*;
b) *containers* de *EJB* que executam dentro do servidor;
c) *home objects*, *EJB objects* e *enterprise beans* que executam dentro de *containers*;
d) sistemas auxiliares como serviço de nomes, transações e persistência.

Figura 13 – Principais elementos da arquitetura de EJB



Fonte: Baseado em Jubin (1999) e Roman (2002).

Os elementos desta arquitetura serão detalhados a seguir.

3.1.2.1 SERVIDOR DE EJB

O servidor de *EJB* é a entidade mais externa dos vários elementos que compõem um ambiente de *EJB*. O servidor gerencia um ou mais *containers* de *EJB* e provê os serviços de suporte, como gerenciamento de transações, persistência e acesso aos clientes. O servidor também disponibiliza um serviço de nomes para os clientes localizarem os *enterprise beans* distribuídos através da rede (Jubin, 1999).

3.1.2.2 O CONTAINER DE EJB

O *container* atua como um intermediário entre o *bean* e o servidor de *EJB*. Segundo Roman (2002), a principal responsabilidade do *container* é prover um ambiente em que o *EJB* possa ser executado e chamado remotamente pelos clientes. O *container* irá implicitamente lidar com os problemas advindos da arquitetura de componentes distribuídos. Ele irá injetar os serviços de baixo nível, necessários para que o *EJB* execute, como gerenciamento do ciclo de vida, comunicação de rede, gerenciamento de transações e persistência.

Segundo Jubin (1999), os *EJB* ganham acesso aos serviços do *container* por implementar os métodos de gerenciamento do *bean*. Um *bean* nunca chama o *container* para executar um serviço específico. É sempre o *container* que chama os métodos de gerenciamento do *bean* para notificar sobre certas circunstâncias ou eventos (Haefel, 2002).

3.1.2.3 O HOME OBJECT E A REMOTE HOME INTERFACE

Um problema que surge com a arquitetura de objetos distribuídos é como adquirir referências para um objeto remoto. Em um ambiente distribuído o cliente não pode simplesmente chamar o construtor do objeto, pois o mesmo pode estar executando em uma máquina diferente. Por isso, para adquirir uma referência ao objeto o cliente deve chamar uma *fábrica* de objetos. A especificação chama esta fábrica de *home object*. O *home object* é criado pelo container. O *home object* é uma implementação da *remote home interface* que expõe para os clientes os métodos disponíveis para criar, remover e localizar os *EJB* (Roman, 2002).

3.1.2.4 O EJB OBJECT E A REMOTE INTERFACE

Quando um cliente invoca um método em um *EJB*, ele nunca acessa a instância do *bean* diretamente. Ao invés disso, a chamada do cliente é interceptada pelo *container* e, então, delegada para a instância do *bean*. Isto faz parte da arquitetura de objetos distribuídos, e é a maneira pela qual o *container* pode prover implicitamente os serviços necessários à execução do *bean* (Roman, 2002).

O *container* intercepta a chamada do cliente através de um objeto chamado *EJB object*. O *EJB object* replica todos os métodos de negócio que o *EJB* expõe na *remote interface*.

Quando o cliente faz uma chamada a um método de negócio do *bean*, ele na verdade estará chamando o método do *EJB object*, que irá chamar os serviços do *container* e então delegar a chamada do método para a instância do *bean*. A criação do *EJB object* é feita automaticamente pelo servidor de *EJB* (Roman, 2002).

A *remote interface* descreve os métodos de negócio, que podem ser chamados remotamente, disponíveis no *bean*. Esta interface deve ser disponibilizada pelo desenvolvedor do *bean*.

3.1.2.5 A CLASSE DE IMPLEMENTAÇÃO DO BEAN

A parte principal do *EJB* é a classe que contém a implementação. Esta classe tem uma interface bem definida e obedece a certas regras. Essas regras são diferentes para cada tipo de *bean*. Os métodos de negócio expostos nas interfaces do *EJB* e os métodos de gerenciamento do ciclo de vida do *EJB* devem ser implementados nesta classe. Entretanto, esta classe não é uma implementação das interfaces diretamente (Haefel, 2002).

3.1.2.6 AS INTERFACES LOCAIS

Um problema que existe na utilização da *home interface* e da *remote interface* é que sempre envolve uma comunicação pela rede, mesmo que o *bean* esteja no mesmo processo que o cliente. Para prover acesso local aos *beans*, a especificação de *EJB* definiu as interfaces locais. Estas interfaces são opcionais e os métodos definidos por elas só podem ser invocados de uma aplicação que esteja executando no mesmo *container* que os *EJB*. Para criar, localizar e remover *beans* a especificação define a interface *local home*, que o container implementa através do *local object*, ao invés do *home object*. Para acessar os métodos de negócio dos *beans*, a especificação define a *local interface*, que é semelhante à *remote interface*, mas que é implementada pelo *local object* (Roman, 2002).

3.1.3 TIPOS DE ENTERPRISE BEANS

Existem três tipos de *enterprise beans*:

a) *entity beans*: modelam objetos do mundo real;
b) *session beans*: modelam processos de negócio;

c) *message-driven beans*: são similares a *session beans*, mas são ativados por mensagens de forma assíncrona.

3.1.3.1 ENTITY BEANS

Entity beans descrevem o estado e o comportamento de objetos do mundo real e permitem aos desenvolvedores encapsular os dados e as regras de negócio associados com conceitos específicos, tais como uma pessoa, uma conta bancária ou um item de estoque.

Entity beans usualmente representam registros de algum tipo de base de dados e são identificados por uma chave primária. Um *entity bean* existe enquanto o dado que ele representa na base de dados existir. A persistência dos *beans* pode ser gerenciada pelo próprio *bean* (*bean managed persistense*) ou pode ser delegada para o container (*container managed persistense*). Quando o *bean* utiliza *container managed persistence*, o *container* gerencia automaticamente a persistência mapeando cada campo do *bean* para a base de dados e automaticamente lê, insere, remove ou altera os dados do banco de dados que o *bean* representa. *Beans* que utilizam *bean managed persistence* fazem todo este trabalho explicitamente: o desenvolvedor do *bean* tem que escrever o código para manipular a base de dados, entretanto, é o *container* que determina quando é seguro persistir os dados (Haefel, 2001).

Os *entity beans* podem ser acessados concorrentemente por vários clientes — a concorrência é gerenciada pelo *container*. Para ler ou alterar os dados de um *entity bean* os clientes chamam os métodos do *bean*, como o método *create* para criar um novo *bean*, o método *remove* para remover o *bean*, os métodos *get/set* dos campos do *bean* e os métodos *find* que são utilizados para localizar um único *bean* ou uma coleção de *beans* (Roman, 2002).

3.1.3.2 SESSION BEANS

Enquanto um *entity bean* possui métodos que atuam sobre o objeto de negócio que ele representa, um *session bean* pode atuar em vários *entity beans* ou fazer várias operações diferentes. Para exemplificar, poderia se representar um processo de venda ao cliente como um *session bean*, a venda em si e os itens vendidos poderiam ser modelados como *entity beans* (Roman, 2002).

Um *session bean*, é criado para prover algum serviço para o cliente, e só existe durante o contexto de uma sessão cliente-servidor (Sun, 2002b). Embora *session beans* possam ser transacionais, eles não são recuperáveis com um *crash* do sistema. E, ao contrário de *entity beans*, *session beans* não são persistentes (Seshadri, 1999).

Um *session bean* é considerado privado para um cliente. Isto habilita o *bean* a manter informação de um cliente específico durante uma sessão. Isto é chamado estado conversacional. Um *session bean* que pode manter um estado conversacional durante várias chamadas de métodos é chamado *stateful*, o oposto deste é chamado *stateless*. Um *stateless session bean*, não pode conter variáveis de instância no contexto do objeto, pois entre a chamada de um método e outro, não há garantia que seu estado será mantido (Jubin, 1999).

3.1.3.3 MESSAGE-DRIVEN BEANS

Um *message-driven bean* possibilita aos clientes acessarem os serviços de negócio na camada de *EJB*, de forma assíncrona. Os *message-driven beans* são ativados somente por mensagens assíncronas recebidas de uma fila de mensagens *JMS*, para os quais eles estão assinados. O serviço de mensagens *JMS* é um padrão de serviço de mensagens, projetado para facilitar a utilização de *message-oriented middlewares* (*MOM*), que são servidores que suportam mensagens, os quais há uma grande variedade disponível no mercado (Roman, 2002).

O cliente não acessa diretamente o *bean*, ao invés disso, o cliente envia mensagens para a fila de mensagens. Dentre as vantagens de se utilizar *message-driven beans* pode-se citar (Roman, 2002):

a) performance: pois o cliente não precisa ficar bloqueado esperando o processo terminar;
b) confiabilidade: mesmo que o servidor esteja momentaneamente parado, as mensagens enviadas para o *bean* ficarão armazenadas no serviço de mensagens até que o servidor volte a funcionar e estas mensagens possam ser encaminhadas para o *bean*;
c) suporte a múltiplos remetentes e receptores de mensagens: o serviço de mensagens pode encaminhar a mensagem para vários *beans* em vários servidores.

3.1.4 PERSISTÊNCIA

Uma das características de *EJB* é a persistência embutida, que é obtida através de *entity beans*.

Em *entity beans* com *bean managed persistense* (*BMP*), o desenvolvedor reescreve os métodos `ejbLoad()` (onde o desenvolvedor escreve o código para ler os dados da base de dados para o *bean*) e `ejbStore()` (que é responsável por escrever os dados do *bean* para a base de dados). Para isso, o desenvolvedor pode utilizar a API de acesso a banco de dados relacionais JDBC e executar comandos SQL. Quando a persistência é gerenciada pelo *bean* o desenvolvedor ganha muita flexibilidade para persistir os dados da maneira desejada, mas amarra o *bean* a um método de persistência e exige que o desenvolvedor reescreva e mantenha o código de persistência. *BMP* é útil, entretanto, quando o *bean* precisa acessar várias tabelas ou utilizar dados de sistemas legados ou integrados (Seshadri, 1999).

Em *container managed persistense* (CMP), é possível associar os campos do *entity bean* com os campos dos dados persistentes. O *container* gera o código automaticamente para persistir os dados em bancos de dados relacionais, isto é, um mapeamento objeto-relacional automático. Adicionalmente, o *entity bean* simplesmente pode ser persistido em uma fonte de dados diferente usando um *container* diferente e re-mapeando os campos (Seshadri, 1999).

3.1.5 TRANSAÇÕES

Nos EJB o desenvolvedor não é exposto à complexidade de transações que poderiam ser distribuídas em várias bases de dados diferentes e múltiplas plataformas. Esta responsabilidade fica a cargo do servidor de aplicações (*middleware*).

As transações podem ser configuradas em um ambiente *EJB* de duas formas diferentes: o desenvolvedor pode criar uma transação explicitamente usando a API *JTS*, de transações para Java, ou, ele pode demarcar os limites de uma transação de forma declarativa no *deployment descriptor* do *bean* (tópico 3.1.7), podendo ser no *bean* inteiro ou em métodos específicos. Os valores válidos para os atributos de transação de um *bean* são (Roman, 2002):

a) *Required*: o *bean* sempre executa dentro de uma transação; se já existe uma transação aberta, o *bean* executa dentro desta transação, se não, o *container* inicia uma nova transação;

b) *RequiresNew*: o *bean* sempre inicia uma nova transação; se já existe uma transação aberta, ela é suspensa durante a invocação do *bean*;
c) *Supports*: o *bean* executa dentro de uma transação somente se o cliente já tiver uma transação executando, se o cliente não tem uma transação o *bean* executa sem transação;
d) *Mandatory*: obriga que uma transação já esteja aberta quando o método do *bean* for invocado. Se não houver uma transação executando a exceção `javax.ejb.TransactionRequiredException` será gerada;
e) *NotSupported*: o *bean* nunca executa no contexto de uma transação. Se existe uma transação executando ela é suspensa até que o *bean* retorne da invocação; todas as operações feitas pelo *bean*, como leitura e gravação para base de dados, não são transacionais;
f) *Never*: o *bean* nunca executa no contexto de uma transação. Se o cliente chamar o *bean* no contexto de uma transação o *container* gera uma exceção.

Segundo Haefel (2001), Jubin (1999) e Roman (2002), *EJB* suporta isolamento de transações, um conceito que permite que mudanças feitas por uma transação sejam visíveis às outras. Os valores válidos são:

a) *READ UNCOMMITTED*: a transação pode ler dados que foram modificados por outra transação que ainda está executando;
b) *READ COMMITTED*: a transação lê somente dados que foram salvos;
c) *REPEATABLE READ*: a transação não pode alterar dados que estão sendo lidos por uma transação diferente;
d) *SERIALIZABLE*: a transação executa em série com respeito às outras tendo acesso exclusivo aos dados. Outras transações que tentem acessar estes dados serão suspensas até que a transação termine.

3.1.6 SERVIÇO DE NOMES

A *Java Naming and Directory Interface* (*JNDI*) é uma interface padrão para localizar usuários, máquinas, redes, objetos e serviços. *JNDI* é usado em *EJB*, *RMI-IIOP (Remote Method Invocation – Internet Inter-Orb Protocol)*, *JDBC* e é o meio padrão de localizar coisas pela rede. Para os clientes localizarem um *bean* disponível na rede, eles devem

pesquisar estes *beans* no servidor de *JNDI*. O nome dos *beans* que são registrados no servidor JNDI é configurado nos *deployment descriptors*. É responsabilidade do *container* deixar os *beans* disponíveis para os clientes por *JNDI* (Roman, 2002).

3.1.7 DEPLOYMENT DESCRIPTORS

Segundo Thomas (1998), os descritores de implantação (*deployment descriptor*), são usados para definir a configuração dos serviços providos pelo servidor para o *EJB*. Estas configurações informam o *container* como gerenciar e controlar o *EJB*. Os descritores de implantação podem ser criados somente no momento de implantar o *EJB* no servidor.

Entre outras coisas o descritor de implantação define o nome da classe que contém a implementação do *EJB*, o nome da *home interface*, o nome da *remote* interface, o nome JNDI para que o *bean* possa ser encontrado através da rede e também as propriedades de ambiente.

3.2 COMPONENTES WEB

Sun (2002b) define um componente *web* como uma unidade de software que responde a requisições. Um componente *web* processa a interação com o usuário em uma aplicação baseada na web. A especificação da plataforma *J2EE* especifica dois tipos de componentes *web*: *Servlets* e *JavaServer Pages (JSP).*

3.2.1 SERVLETS

Segundo Hunter (2001) um *servlet* é uma extensão genérica de um servidor, isto é, uma classe Java que pode ser carregada dinamicamente para expandir a funcionalidade de um servidor. Geralmente os *servlets* são usados em servidores *web*, onde eles funcionam como aplicações CGI (*Common Gateway Interface*), recebendo requisições e gerando respostas.

A classe do *servlet* executa no servidor *web* por meio de um *container* de *servlets*. O servidor *web* mapeia um conjunto de URL (*Universal Resource Locator*) para um *servlet.* Quando o servidor recebe uma requisição para uma destas URL, ele repassa a requisição para o *servlet*, que gera uma resposta para o cliente. Geralmente a resposta é um documento HTML ou XML. A grande vantagem do *servlet* em relação ao *applet* é que o *servlet* executa

no servidor, por isso, ele não depende da compatibilidade entre os vários navegadores de Internet (Goodwill, 2001).

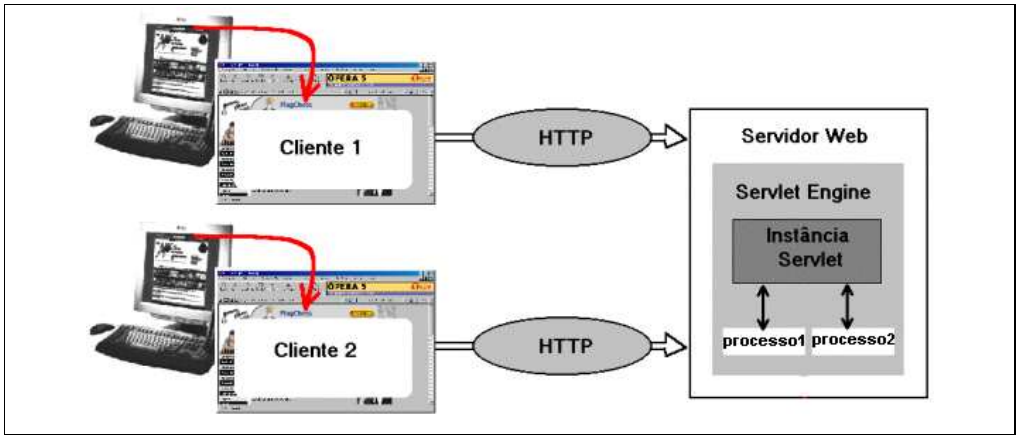
3.2.1.1 ARQUITETURA

O *container* de servlets fornece mecanismos que gerenciam a comunicação entre o cliente e o servidor. Quando é feita uma solicitação de um cliente para um *servlet*, ele recebe dois objetos: um objeto `request` que encapsula a comunicação entre o cliente e o servidor e um objeto de `response` que encapsula a comunicação do servidor com o cliente (Hunter, 2001).

O objeto `request` permite ao *servlet* acessar informações tais como os parâmetros passados pelo cliente, o protocolo que está sendo usado pelo cliente, o nome da máquina remota que fez a solicitação e o servidor que recebeu esta solicitação. O objeto `response` fornece ao *servlet* um meio de enviar uma resposta ao cliente (Hunter, 2001).

O *servlet* é carregado e executado pelo servidor *Web*. Uma vez inicializado, o *servlet* estará apto a lidar com centenas de acessos simultaneamente. Geralmente o servidor cria somente uma instância do *servlet* e, a cada acesso, dispara um novo processo (Adam, 2001). A Figura 14 apresenta este modelo.

Figura 14 – Vários processos utilizando a mesma instância



Fonte: Baseado em Adam (2001).

O *servlet* possui alguns métodos padrões para atender ao protocolo HTTP. A cada vez que o *servlet* recebe uma solicitação o servidor inicia um novo processo e chama o método `service()`. Este método verifica o tipo de solicitação feita (POST, GET, HEAD, DELETE) e chama os métodos que tratam estes tipos de solicitação que podem ser `doGet()`, `doPost()`, `doHead()` e `doDelete()` (Hall, 2000).

3.2.1.2 SESSÕES

O protocolo HTTP é um protocolo não orientado a sessão. Cada vez que o cliente faz uma solicitação ele abre uma conexão separada com o servidor e o servidor não mantém automaticamente informações sobre o cliente. A API de *servlets* implementa de maneira proprietária o mecanismo de sessões. Uma sessão é inicializada quando o cliente faz a primeira chamada ao servidor, gerando um número de identificação. Este identificador é mantido durante as várias conexões do cliente, até que ele feche o navegador ou até que o seu tempo expire (*time out*) (Hall, 2000).

Uma sessão é mantida em memória no lado do servidor; os *servlets* permitem que sejam mantidas informações sobre a sessão através da classe `javax.servlet.HttpSession`. A classe `javax.servlet.HttpSession` fornece uma coleção de métodos para criar, manipular e destruir sessões (Adam, 2001).

3.2.1.3 FILTROS

Segundo Hall (2000), um filtro executa no servidor antes do *servlet* ou da *página* aos quais está associado. Um filtro pode ser mapeado para uma ou mais URL e pode examinar as informações que foram encaminhadas junto com a solicitação. Após isso, o filtro pode tomar uma das seguintes ações:

a) invocar o recurso solicitado normalmente;
b) invocar o recurso solicitado com as informações modificadas;
c) invocar o recurso, mas, modificar a resposta antes de enviá-la para o cliente;
d) prevenir o recurso de ser invocado e redirecionar para um recurso diferente ou retornar um código de erro.

Para criar um filtro é necessário implementar a interface `Filter`. Esta interface define três métodos: `doFilter`, `init` e `destroy`. O método `doFilter` contém o código de filtragem,

o método `init` faz operações de inicialização e o método `destroy` faz a liberação dos recursos utilizados. Os dois primeiros argumentos do método `doFilter` são os objeto `request` e `response`, respectivamente, que fornecem acesso às informações transmitidas entre o cliente e o servidor. O último argumento é um objeto do tipo `FilterChain`, que é usado para invocar o próximo filtro, ou se não houver mais filtros, o recurso solicitado.

3.2.2 JAVASERVER PAGES

Muitas aplicações *Web* produzem primariamente páginas HTML dinâmicas que, quando acessadas, mudam somente os dados, e não a sua estrutura básica. Os *servlets* não fazem distinção entre o que é código de formatação HTML e o que são dados. A tecnologia de *JavaServer Pages* (*JSP*), entretanto, difere deste modelo de programação. Uma página JSP é primariamente um documento que especifica conteúdo dinâmico, ao invés de um programa que produz conteúdo. As páginas JSP fornecem uma alternativa "centrada em documentos", ao contrário dos *servlets* que são classes para criar conteúdo dinâmico.

Sun (2002b) define uma página *JSP* como um documento contendo HTML estático, com algumas marcações para incluir dados ou para executar alguma lógica embutida na própria página *JSP*. Goodwill (2001) explica que as páginas *JSP* são uma extensão de *servlets*. De fato, o servidor irá transformar a página *JSP* em um *servlet* que irá gerar o conteúdo da página (Hunter, 2001).

Em uma página *JSP* o HTML estático é enviado para o cliente da maneira em que aparece na página. As marcações especiais podem ser de três tipos:

a) *diretivas*: são instruções para o compilador de páginas *JSP* e são avaliadas em tempo de compilação;
b) elementos de *script*: são blocos de código Java embutidos na página *JSP;*
c) *custom tags*: são marcações definidas pelo usuário.

3.2.2.1 DIRETIVAS

As diretivas *JSP* afetam toda a estrutura do *servlet* que resulta da página *JSP*. Em páginas *JSP* existem três tipos de diretivas: *`page`*, *`include`* e *`taglib`*. A diretiva *`page`* habilita o desenvolvedor a controlar a estrutura do *servlet* por importar classes, customizar a superclasse do *servlet* e alterar o tipo de conteúdo da página. A diretiva *`include`* serve para

inserir um arquivo na classe do *servlet* quando a página é transformada em um *servlet*, sendo colocada no documento no ponto em que o arquivo deve ser inserido. A diretiva `taglib` é usada para definir *custom tags* e importar bibliotecas de *tags* chamadas *tag libraries*. (Hall, 2000)

3.2.2.2 ELEMENTOS DE SCRIPT

Elementos de *script* permitem inserir código Java no *servlet* que é gerado a partir da página *JSP*. Estes elementos podem ser de três formas (Hall, 2000):

a) expressões: são da forma `<%= expressão %>`. O seu conteúdo é avaliado, convertido em um *string* e inserido diretamente na saída gerada para o cliente;
b) declarações: são da forma `<%! código %>`. É usado para definir métodos ou variáveis que são inseridas no corpo da classe do *servlet* gerado;
c) *scriptlets*: são da forma `<% código %>`. Permitem inserir qualquer código Java. O seu conteúdo é inserido no método `_jspService()`, que é chamado pelo método `service()` do *servlet* gerado, para responder a uma solicitação de um cliente. A geração de saída para o cliente é da mesma forma dos *servlets*.

3.2.2.3 CUSTOM TAGS

Segundo Hall (2000), *custom tags* são marcações definidas pelo desenvolvedor, que são convertidas por conteúdo dinâmico quando a página é servida. O conteúdo dinâmico é criado por uma classe que o programador cria e empacota em uma biblioteca de *tags* chamada *tag library*. O programador define a sintaxe para uma *tag* e implementa o comportamento para esta *tag* na classe. Os desenvolvedores de páginas *JSP* importam estas *tags* na página e as utilizam como qualquer marcação HTML.

3.2.2.4 VARIÁVEIS PREDEFINIDAS

Uma página JSP possui quatro variáveis predefinidas, que segundo Hall (2000) são:

a) `request`: encapsula a comunicação entre o cliente e o servidor;
b) `response`: encapsula a comunicação entre o servidor e o cliente;
c) `session`: a sessão associada com o cliente que fez a solicitação;
d) `out`: usado para enviar a saída para o cliente.

4 PADRÕES DE PROJETO

Os padrões de projeto, também conhecidos como *Design Patterns*, visam uma melhor reutilização de software. Estes padrões tornam mais fácil reutilizar projetos e arquiteturas bem sucedidas. Eles ajudam a escolher alternativas de projeto que tornam um sistema reutilizável e a evitar alternativas que comprometam a reutilização.

Segundo Marinescu (2002), um padrão de projeto é definido como uma solução desenvolvida utilizando boas práticas para um problema comum que ocorre várias vezes. Um padrão de projeto documenta e explana um problema importante que pode ocorrer no projeto ou implementação de uma aplicação e então discute a melhor solução prática para o problema.

Schneide (1999) explica que a meta é a criação de uma linguagem comum, que permita uma comunicação efetiva no que se refere à troca de experiências sobre problemas e suas soluções. Desta forma, soluções que se aplicaram a situações particulares, podem ser novamente aplicadas em situações semelhantes por outros desenvolvedores.

Em termos de orientação a objetos, padrões de projeto identificam classes, instâncias, seus papéis, colaborações e a distribuição de responsabilidades. Seriam, então, descrições de classes e objetos que se comunicam, que são implementados a fim de solucionar um problema comum em um contexto específico (Schneide, 1999). As vantagens de se utilizar padrões em um projeto são listadas por Alur (2002):

a) foram testados: refletem a experiência e conhecimento dos desenvolvedores que utilizaram estes padrões com sucesso em seu trabalho;
b) são reutilizáveis: fornecem uma solução pronta que pode ser adaptada para diferentes problemas quando necessário;
c) são expressivos: formam um vocabulário comum para expressar grandes soluções sucintamente. Os desenvolvedores podem dizer que uma aplicação foi construída usando um padrão em particular, ao invés de explicar toda a semântica envolvida;
d) facilitam o aprendizado: reduzem o tempo de aprendizado de uma determinada biblioteca de classes. Isto é fundamental para o aprendizado dos desenvolvedores novatos;
e) diminuem re-trabalho: quanto mais cedo são usados, menor será o re-trabalho em etapas mais avançadas do projeto.

4.1 ESTRUTURA DE UM PADRÃO DE PROJETO

Um fator que torna os padrões bem adaptados para catalogar conhecimento é a sua estrutura. Bons padrões apresentam meios de resolver um problema e são estruturados de maneira a explanar os aspectos deste problema e a sua solução. O modelo apresentado neste trabalho foi baseado no modelo proposto por Alur (2002) e consiste nas seguintes seções:

a) contexto: define o ambiente sob o qual o padrão existe;
b) problema: descreve as questões de projeto enfrentadas pelo desenvolvedor;
c) forças: lista as razões e motivações que afetam o problema e a solução. A lista de forças destaca as razões pelas quais alguém poderia optar por utilizar o padrão;
d) solução: descreve resumidamente o enfoque da solução;
e) conseqüências: lista as vantagens e desvantagens de se utilizar o padrão;
f) estrutura: utiliza diagramas de classes para mostrar a estrutura básica da solução e diagramas de seqüência para apresentar os mecanismos da solução. Há uma explicação dos participantes e colaborações e pode conter exemplos de código fonte.

Há uma grande quantidade de documentação de padrões de software disponíveis atualmente. Esses padrões estão organizados em vários níveis de conceito: padrões de arquitetura, padrões de projetos, padrões de análises e padrões de programação. Neste trabalho serão apresentados padrões de projeto que descrevem a estrutura de uma aplicação e outros que descrevem os elementos do projeto. O ponto em comum entre eles é que foram desenvolvidos e aplicados especificamente à plataforma de desenvolvimento *J2EE*.

4.2 O CATÁLOGO DE PADRÕES DE PROJETO

O catálogo de padrões de projeto aqui apresentados foi dividido de acordo com as camadas a que eles pertencem. A camada de apresentação contém os padrões relacionados aos *Servlets* e páginas *JSP*. Os padrões da camada de negócios são relacionados à tecnologia de *EJB* e os padrões da camada de integração estão relacionados à comunicação com recursos e sistemas externos.

Os padrões da camada de apresentação conforme apresentados por Alur (2002):

a) *intercepting filter*: intercepta solicitações e respostas, e aplica um filtro;

b) *front controller*: fornece um controlador centralizado para manter a lógica de processamento que ocorre na camada de apresentação e que, erroneamente é colocado nas visões (*view*);
c) *view helper*: encapsula a lógica que não esteja relacionada à formatação da apresentação em componentes auxiliares;
d) *composite view*: cria uma visão através da composição de outras visões;
e) *service to worker*: combina um componente distribuidor (*dispatcher*) com os padrões *Front Controller* e *View Helper*. Neste padrão a responsabilidade de recuperar o conteúdo para apresentar é do controlador;
f) *dispatcher view*: semelhante ao padrão *Service to Worker*, mas, neste padrão a recuperação do conteúdo é responsabilidade das visões.

Os padrões da camada de negócios são (Alur, 2002).

a) *business delegate*: reduz o acoplamento entre o cliente e a camada de negócios;
b) *data transfer object*: facilita o intercâmbio de dados entre as camadas;
c) *data transfer object factory*: facilita e centraliza a criação de *data transfer objects*;
d) *session facade*: fornece uma interface unificada para um sistema distribuído;
e) *composite entity*: representa uma prática mais recomendada para criar *entity beans* de granulação grossa, agrupando os objetos dependentes em um único *entity bean*;
f) *value list handler*: gerencia o processamento e armazenamento em *cache* de consultas que retornam uma grande quantidade de informações;
g) *service locator*: encapsula a complexidade de localização e criação de serviços de negócio e localiza os objetos *Home* dos *EJB*.

Os padrões da camada de integração descritos por Alur (2002) são:

a) *data access object*: abstrai e encapsula o acesso aos dados;
b) *service activator*: facilita o processamento assíncrono para *Message-driven beans*.

A seguir serão descritos os padrões que foram utilizados na implementação deste trabalho.

4.3 PADRÕES DA CAMADA DE APRESENTAÇÃO

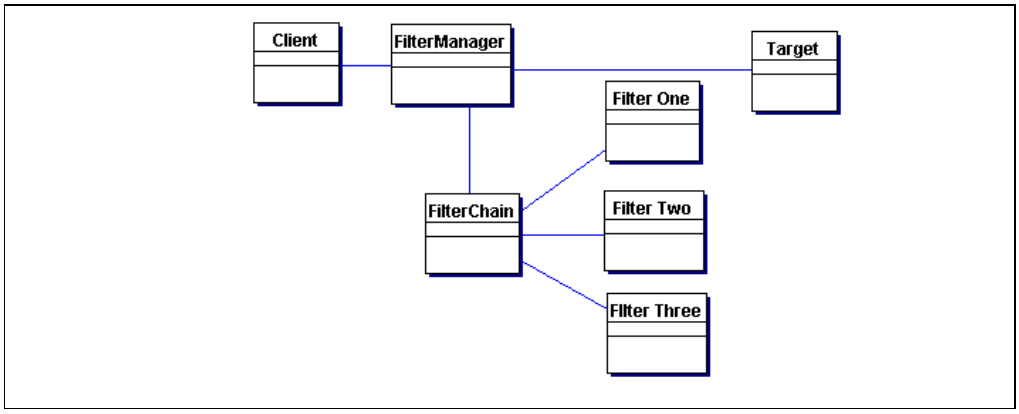
4.3.1 INTERCEPTING FILTER

Este padrão foi proposto por Alur (2002) e visa aplicar um filtro nas solicitações e envio de respostas aos clientes. O padrão é descrito a seguir:

a) contexto: as requisições da camada de apresentação devem ser modificadas, examinadas ou descompactadas antes de serem processadas mais adiante;
b) problema: quando uma aplicação *WEB* recebe uma requisição, alguns testes devem ser feitos antes do estágio de processamento principal, os quais podem ser:
   - testar se o cliente foi autenticado;
   - testar se o cliente tem uma sessão válida;
   - verificar se o endereço IP do cliente é de uma rede confiável;
   - verificar se o caminho informado viola alguma restrição;
   - mudar a codificação dos dados se necessário;
   - verificar o navegador do cliente.
c) forças: as forças para se utilizar este padrão de projeto são as seguintes:
   - é desejável a centralização da lógica comum;
   - os serviços de filtragem devem ser fáceis de adicionar e remover;
d) solução: criar um mecanismo para adicionar e remover filtros de forma declarativa, sem alterar o código da aplicação principal. A especificação de *servlets* inclui um mecanismo padrão para criar cadeias de filtros. Estes filtros podem ser configurados no *deployment descriptor* da aplicação;
e) conseqüências: as conseqüências da utilização deste padrão são:
   - centraliza o controle e processamento das requisições;
   - melhora a reutilização, pois são conectáveis, podendo ser removidos ou adicionados de forma transparente para o código existente;
   - configuração declarativa e flexível: vários filtros podem ser combinados em permutações variadas sem necessidade de recompilação;
f) estrutura: a Figura 15 apresenta o diagrama de classe do padrão *Intercepting Filter*, onde uma requisição de um objeto *Client* é interceptada por um objeto *FilterManager*, que cria a cadeia de filtros *FilterChain* que por sua vez, aplica todos os filtros necessários e devolve o controle para o *FilterManager*. Se a

aplicação dos filtros foi efetuada com sucesso o *FilterManager* encaminha então a requisição para o objeto *Target* que representa o objeto que o *Client* requisitou;

Figura 15 – Padrão de Projeto Intercepting Filter



Fonte: Alur (2002).

4.3.2 FRONT CONTROLLER

Este padrão tem o objetivo de colocar um controlador central para atender a todas as solicitações dos clientes, retirando esta lógica das visões (*view*). Foi descrito por Alur (2002), conforme segue:

a) contexto: na camada de apresentação de uma aplicação *web* ocorre processamento que deve ser controlado para cada usuário e por múltiplas requisições. Este processamento pode ser gerenciado de maneira centralizada ou descentralizada;
b) problema: quando o usuário acessa diretamente as visões sem operar através de um mecanismo centralizado, três problemas podem ocorrer:
   - o código para tratar as solicitações do usuário é colocado em cada visão tornando a manutenção mais difícil;
   - cada visão é obrigada a chamar os serviços do sistema resultando em código duplicado;
   - a navegação de uma visão para outra (*links*), é codificada em cada visão;
c) forças: as forças para a utilização do *Front Controller* são as seguintes:
   - a lógica que é mais bem tratada em um local central, é replicada em inúmeras visões;

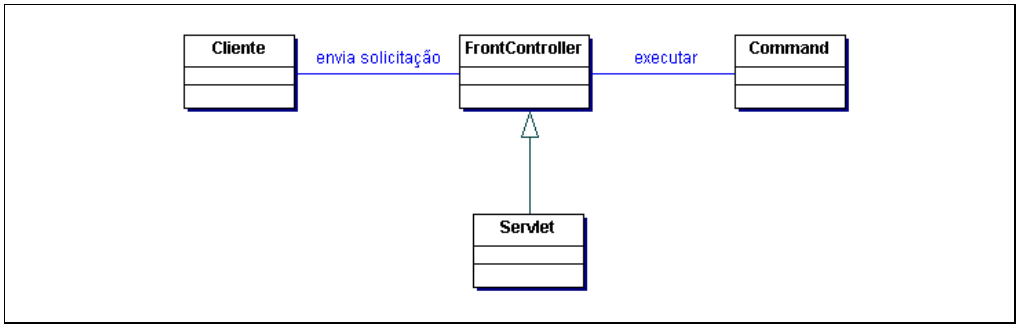
- a recuperação e o tratamento de dados para apresentar, são complicados;
- visões diferentes são utilizadas para responder à solicitações de negócio semelhantes;
- nos caso de uso do sistema o fluxo de trabalho é dividido entre várias visões;

d) solução: utilizar um controlador como o ponto inicial de contato para o tratamento das solicitações. O controlador gerencia o tratamento da solicitação por chamar os serviços necessários, delegar o processamento da lógica de negócios, selecionar a visão apropriada, tratar erros e gerenciar a seleção de estratégia de criação de conteúdo;

e) conseqüências: as conseqüências alistadas por Alur (2002) são:
   - centraliza o controle: facilita o tratamento da solicitação, principalmente quando a lógica de negócios se estende por múltiplas solicitações;
   - melhora o gerenciamento da segurança: é possível restringir tentativas de acesso ilícito na aplicação;
   - melhora a reutilização: facilita o particionamento da aplicação e a separação da lógica de criação de conteúdo e de negócios da lógica de formatação dos dados;

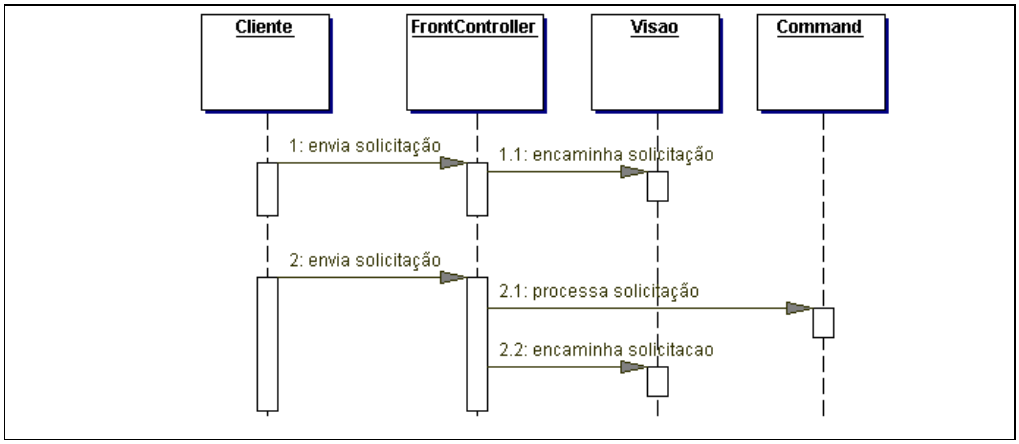
f) estrutura: a Figura 16 apresenta o diagrama de classe do padrão *Front Controller*. O diagrama de seqüência apresentado na Figura 17 mostra como o controlador trata uma solicitação onde, uma requisição de um objeto *Client* é feita para um objeto *FrontController*, que determina o processamento a ser efetuado para esta requisição e encaminha para as visões. O *FrontController* processa as requisições por meio de objetos auxiliares, geralmente, estes objetos são derivados do padrão *Command* descrito por Gamma (2000), que representam uma única operação como um objeto.

Figura 16 – Diagrama de classes do *Front Controller*



Fonte: Baseado em Alur (2002).

Figura 17 – Diagrama de seqüência do *Front Controller*



Fonte: Baseado em Alur (2002).

4.3.3 COMPOSITE VIEW

Este padrão descreve uma maneira de criar visões (geralmente páginas *Web*) que possam ser compostas de outras visões. Este padrão foi descrito por Alur (2002), conforme segue:

a) contexto: as visões geralmente têm algum conteúdo semelhante, que pode ser reutilizado em outras visões. Estas visões também possuem um layout padronizado, onde podem ser utilizados templates;

b) problema: entre as várias visões de uma aplicação, os dados e o conteúdo podem variar, mas muitos elementos como o cabeçalho ou a barra de opções são os mesmos. Além disso, a estrutura e o layout podem ser os mesmos em todas as visões, e alguns elementos podem parecer em muitas visões diferentes. Se estes elementos forem codificados em cada visão, torna-se difícil modificar a sua estrutura e manter uma aparência e comportamento consistentes para toda a aplicação.
c) forças: os forças para se utilizar este padrão são:
   - pequenas parcelas do conteúdo da visões se alteram freqüentemente;
   - as alterações no layout são mais difíceis de gerencias quando os elementos que se repetem são codificados diretamente na visão;
   - diversas visões compostas utilizam visões secundárias similares, como uma tabela de inventário do cliente, que contém vários grupos de informações do cliente;
d) solução: utilizar visões que sejam compostas de outras visões. Cada visão pode ser incluída dinamicamente no todo e o layout da página pode ser gerenciado independentemente do conteúdo;
e) conseqüências: como conseqüências da utilização deste padrão pode-se alistar:
   - melhora a modularidade e a reutilização: é possível reutilizar as visões simplesmente incluindo-as em uma outras visões. Este tipo de layout e composição dinâmicos reduz a duplicação;
   - aumenta a flexibilidade: o layout pode ser mudado em tempo de execução;
   - aumenta a capacidade de manutenção: é muito mais eficiente modificar o layout, por este estar centralizado e não misturado com o código das visões. Também, é mais fácil manter as visões por estas não conterem código de formatação;
   - impacto no desempenho: diminui o desempenho pois cada visão da composição tem de ser interpretada;
f) estrutura: uma estratégia comum para implementar o padrão *Composite View* em uma aplicação utilizando páginas *JSP* é utilizando *custom tags*. Neste trabalho foi utilizada uma biblioteca de templates chamada *Struts* (Burns, 2002). O Quadro 1 mostra o código fonte de uma *Composite View* utilizando *Struts*. Neste caso, o

template para a página é *template.jsp*. A página *template.jsp* define o layout e a posição para os quatro elementos ou seções: cabeçalho, barra-lateral, conteúdo e rodapé, conforme mostrado no Quadro 2. Esta template irá gerar uma página com a estrutura semelhante à da Figura 18.

Quadro 1 – Página *JSP* utilizando o padrão *Composite View*

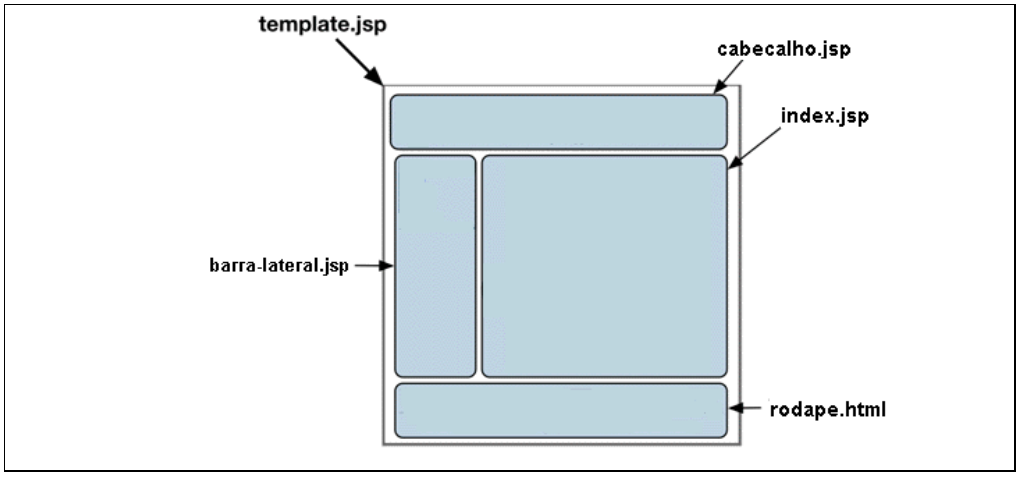
```
<%@ taglib uri='/WEB-INF/struts-template.tld' prefix='template' %>
<template:insert template='/template.jsp'>
  <template:put name='cabecalho'     content='/cabecalho.jsp' />
  <template:put name='barra-lateral' content='/barra-lateral.jsp' />
  <template:put name='conteudo'      content='/index.jsp'/>
  <template:put name='rodape'        content='/rodape.html' />
</template:insert>
```

Quadro 2 – Definição da *template* template.jsp

```
<%@ taglib uri='/WEB-INF/struts-template.tld' prefix='template' %>
<html><head>
<table>
   <tr valign='top'>
      <td>  <template:get name='barra-lateral'/>  </td>
      <td><table>
          <tr><td> <template:get name='cabecalho'/> </td></tr>
          <tr><td> <template:get name='conteudo'/>  </td></tr>
          <tr><td> <template:get name='rodape'/> </td></tr>
      </table></td>
   </tr>
</table>
</body></html>
```

Formatado

Figura 18 – Múltiplas páginas JSP formando um Composite View



4.4 PADRÕES DA CAMADA DE NEGÓCIOS

4.4.1 SESSION FAÇADE

Este padrão representa uma maneira de isolar a complexidade do modelo de negócios do cliente e diminuir o tráfego da rede, ocasionado por múltiplas chamadas remotas aos *EJB*. Este padrão é um dos padrões mais fundamentais para aplicações utilizando *EJB*, tendo a seguinte descrição (Marinescu, 2002):

a) contexto: às vezes, a realização de um caso de uso do sistema envolve a invocação de diversos *EJB*, expondo assim a complexidade dos serviços para os clientes e ocasionando ruído na rede pois as invocações são remotas;
b) problema: a execução de um caso de uso do sistema gerenciada pelos clientes apresenta uma série de problemas:
   - o alto índice de chamadas remotas traz prejuízos à performance da rede;
   - diminui a manutenibilidade do código, visto que o fluxo é implementado pelo cliente;
   - código fortemente acoplado, pois o cliente é escrito usando diretamente os *EJB*. Se a camada de *EJB* mudar, é necessário mudar o cliente também;
   - pouca reusabilidade, pois outros tipos de clientes não poderão utilizar esta mesma lógica;

- pouca separação de papéis entre os desenvolvedores da camada de negócio e de apresentação;

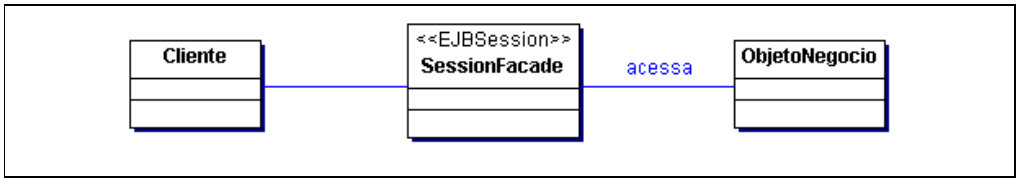
c) forças: as forças que motivam a utilizar o padrão *Session Facade* são:
   - fornecer uma interface mais simples para os clientes ocultando todas as interações entre os componentes de negócio;
   - reduzir o número de objetos de negócio expostos ao cliente;
   - ocultar do cliente as interdependências entre os objetos de negócio;
   - fornecer uma camada de serviço uniforme;
   - diminuir o acoplamento entre as camadas por ocultar os objetos de negócio;
   - diminuir o número de invocações de métodos feitos através da rede;

d) solução: ocultar a camada de objetos de negócio do cliente por adicionar uma camada de *session beans* chamada *Session Facade*. Os clientes devem acessar somente os *Session Façade*, não devem ter acesso ao modelo de objetos do servidor;

e) conseqüências: as principais conseqüências da utilização do padrão são:
   - baixa perda de performance com a rede, visto que o cliente pode executar a operação (caso de uso) em apenas uma chamada remota;
   - separação da lógica entre a camada de negócios e de apresentação, visto que a lógica de negócios é completamente ocultada pelo *Session Facade*, os clientes só se preocupam com a apresentação dos dados;
   - integridade transacional, devido ao tempo da transação ser reduzido, pois as operações são locais;
   - baixo acoplamento: se o modelo de objetos distribuídos mudar não será necessário mudar o cliente;
   - boa reusabilidade: a lógica de negócios é encapsulada no servidor podendo ser acessada de qualquer tipo de cliente;
   - centraliza o gerenciamento de segurança e transação: as configurações de segurança e transação podem ser gerenciadas no nível do *Session Facade* e não para cada objeto de negócio;
   - separação entre os objetos de negócio e as ações: a camada de *session beans* modela os casos de uso do sistema, enquanto os *entity beans* modelam os objetos de negócio;

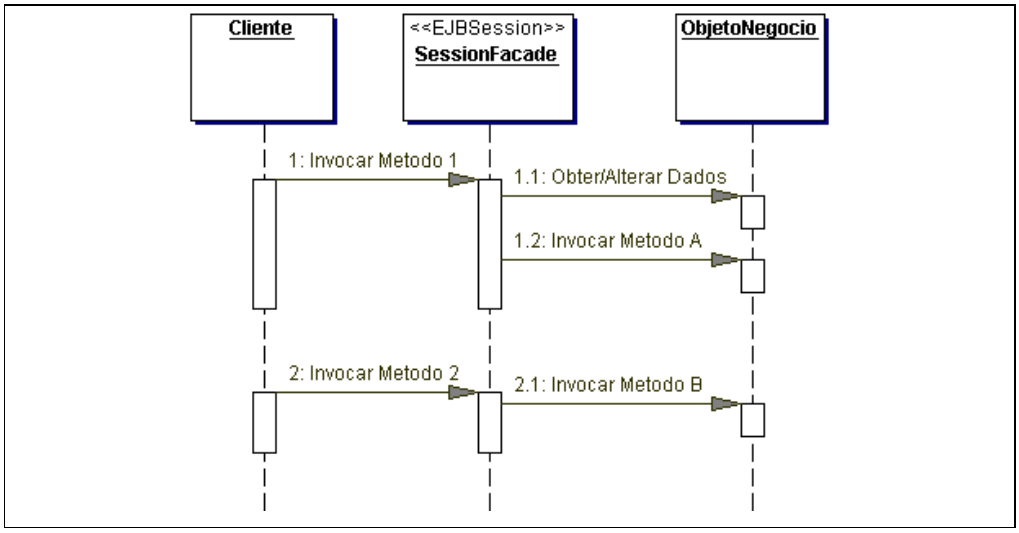
f) estrutura: a Figura 19 apresenta o diagrama de classe do padrão *Session Facade*. Participam deste padrão o objeto *Cliente*, que acessa o serviço de negócios, o objeto *SessionFacade*, é um *session bean* que proporciona uma abstração para o cliente e o objeto *ObjetoNegocio* que representa qualquer objeto de negócio. A Figura 20 mostra o diagrama de seqüência para este padrão.

Figura 19 – Diagrama de classe do padrão Session Facade



Fonte: Baseado em Alur (2002).

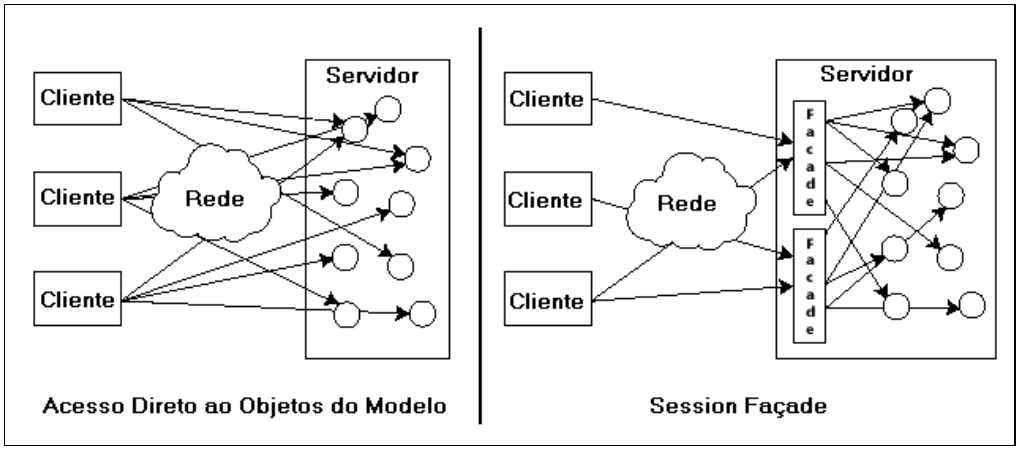
Figura 20 – Diagrama de seqüência do padrão Session Facade



Fonte: Baseado em Marinescu (2002).

A Figura 21 mostra os benefícios na arquitetura do sistema usando um *Session Facade*:

Figura 21 - Benefícios arquiteturais da utilização do padrão *Session Facade*



Fonte: Baseado Marinescu (2002).

4.4.2 BUSINESS DELEGATE

Este padrão apresenta uma forma de isolar o cliente da manipulação de objetos de negócio remotos, no caso os *EJB*. Segue a definição do padrão conforme Marinescu (2002):

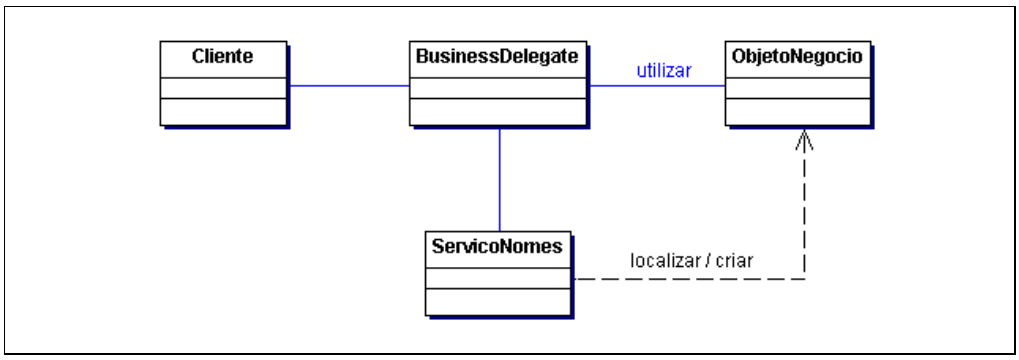
a) contexto: a aplicação cliente necessita acessar serviços de um objeto de negócios remoto;
b) problema: vários passos são necessários para acessar os serviços de um objeto remoto, como por exemplo, localizar o objeto remoto e instanciá-lo, tratando os erros que podem ser geradas por estes processos. Colocar esta lógica nos clientes diminui a manutenibilidade e acopla os clientes aos objetos de negócio. Também, não é possível fazer *cache* nos clientes para minimizar o tráfego de rede;
c) forças: as forças que motivam a utilizar o padrão *Business Delegate* são:
   - os clientes precisam acessar a camada de negócios;
   - as interfaces dos serviços de negócio podem mudar freqüentemente;
   - é desejável ocultar dos clientes os detalhes de implementação dos serviços;
   - os clientes precisam implementar mecanismos de *cache* para informações de serviços de negócio;

d) solução: ocultar os detalhes de implementação por traz dos serviços de negócio por adicionar uma camada de objetos *Business Delegate* no lado cliente. O cliente faz as solicitações dos serviços de negócio para o *Business Delegate* que delega estas solicitações para os serviços de negócio, que são geralmente objetos *Session Facade*;

e) conseqüências: as principais conseqüências da utilização do padrão são:
   - reduz o acoplamento ocultando os detalhes de implementação dos serviços distribuídos;
   - manipula exceções inerentes a objetos distribuídos;
   - implementa recuperação de falhas automática sem expor os problemas ao cliente;
   - expõe uma interface mais simples para a camada de negócios;
   - aumenta o desempenho pois pode oferecer serviços de *cache*;
   - introduz uma camada adicional, podendo aumentar a complexidade e diminuir a flexibilidade;
   - oculta a localização dos serviços remotos;

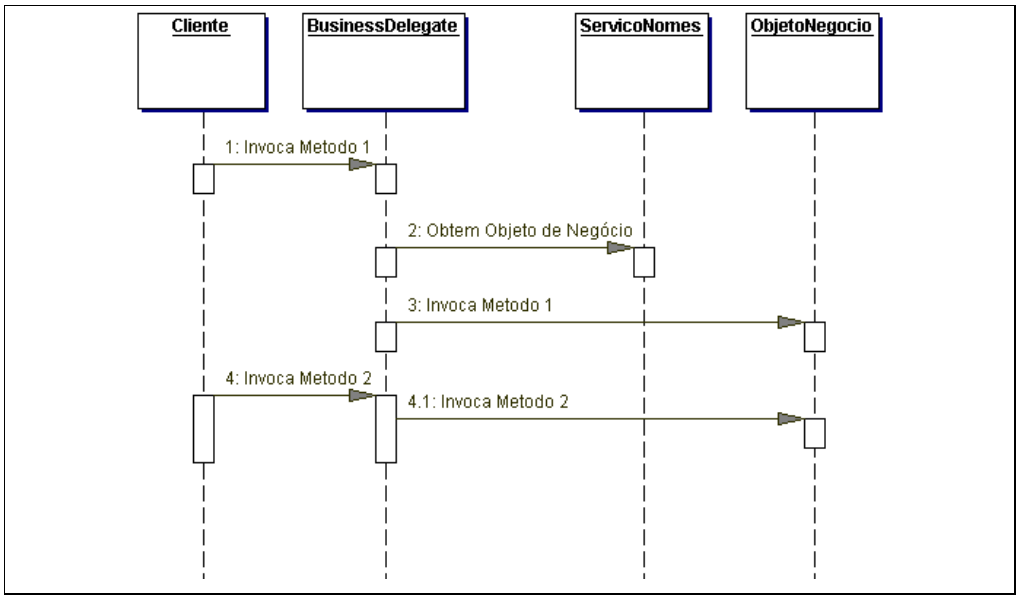
f) estrutura: a Figura 22 apresenta o diagrama de classe do padrão *Business Delegate*. Neste padrão o objeto *Cliente* acessa o *BusinessDelegate*, que localiza os objetos de negócio distribuídos e delega as requisições dos clientes. A Figura 23 mostra o diagrama de seqüência para este padrão.

Figura 22 – Diagrama de classe do padrão *Business Delegate*



Fonte: Baseado em Alur (2002).

Figura 23 – Diagrama de seqüência do padrão Business Delegate



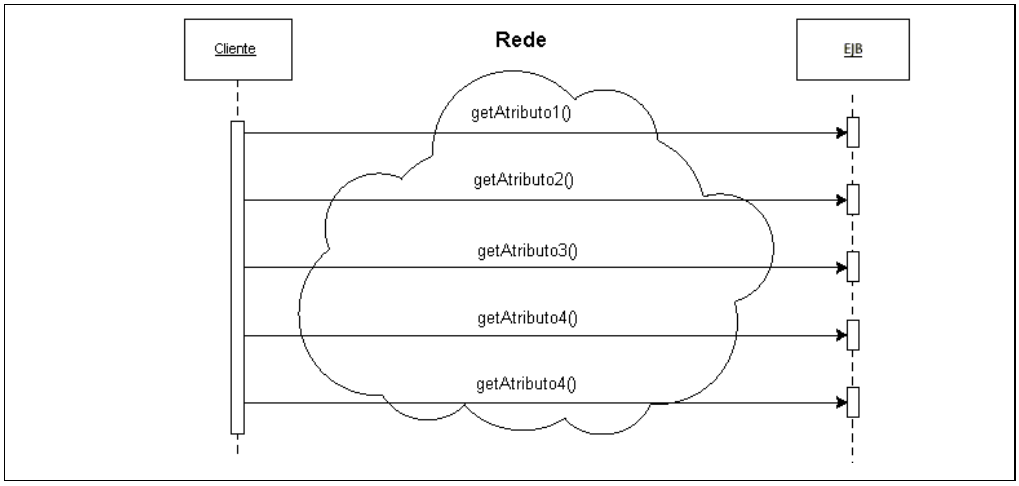
Fonte: Baseado em Alur (2002).

4.4.3 DATA TRANSFER OBJECT

Este padrão descreve uma maneira eficiente de trocar dados entre as camadas, tendo a seguinte descrição (Marinescu, 2002):

a) contexto: os clientes precisam trocar dados com o servidor;
b) problema: freqüentemente o cliente necessita obter ou alterar o valor de vários atributos de um objeto de negócio, em um ambiente utilizando *EJB*, qualquer chamada de método feita para um objeto de negócio é potencialmente remota. Dessa maneira, tais chamadas remotas irão utilizar a camada de rede, independente da proximidade entre os objetos de negócio e o cliente, causando uma sobrecarga na rede (Alur, 2002). A Figura 24 ilustra este problema:

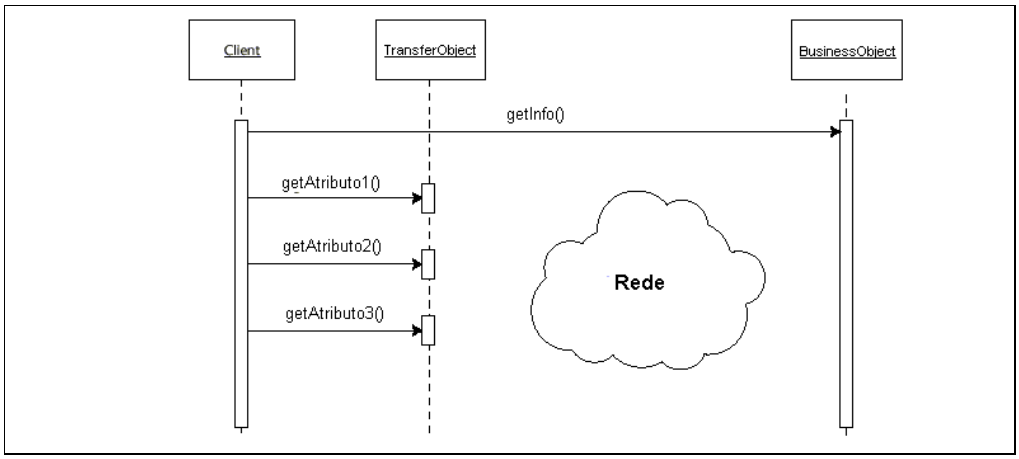
Figura 24 – Um meio ineficiente de obter dados do servidor



Fonte: Baseado em Marinescu (2002).

c) forças: as forças que motivam a utilização deste padrão acordo com Alur (2002) são:
   - as aplicações de negócio têm uma freqüência maior de transações de leitura do que de atualização;
   - o cliente normalmente requer, dos objetos de negócio, valores para mais de um atributo;
   - o número de chamadas para os objetos de negócio afeta o desempenho da rede;
d) solução: criar um objeto chamado *Data Transfer Object* para encapsular os dados dos objetos de negócio. Este objeto é retornado em uma única chamada de método com uma cópia dos dados que estão no objeto de negócio;
e) conseqüências: as conseqüências são (Alur, 2002):
   - transfere mais dados em menos chamadas remotas;
   - duplicação de código, visto que é necessário criar uma nova classe que represente o objeto de negócio;
f) estrutura: a Figura 25 mostra a interação entre os objetos utilizando o padrão *Data Transfer Object*. Neste padrão o C*liente* solicita para o *ObjetoNegocio* os dados de que ele necessita que são devolvidos encapsulados no objeto *DataTransferObject*.

Figura 25 - Diagrama de seqüência *Data Transfer Object*



Fonte: Baseado em Marinescu (2002).

4.5 PADRÕES DA CAMADA DE INTEGRAÇÃO

4.5.1 DATA ACCESS OBJECT

O padrão *Data Access Object* (*DAO*) apresenta uma maneira de abstrair e encapsular o acesso aos dados. Este padrão *DAO* é descrito a seguir (Alur, 2002):

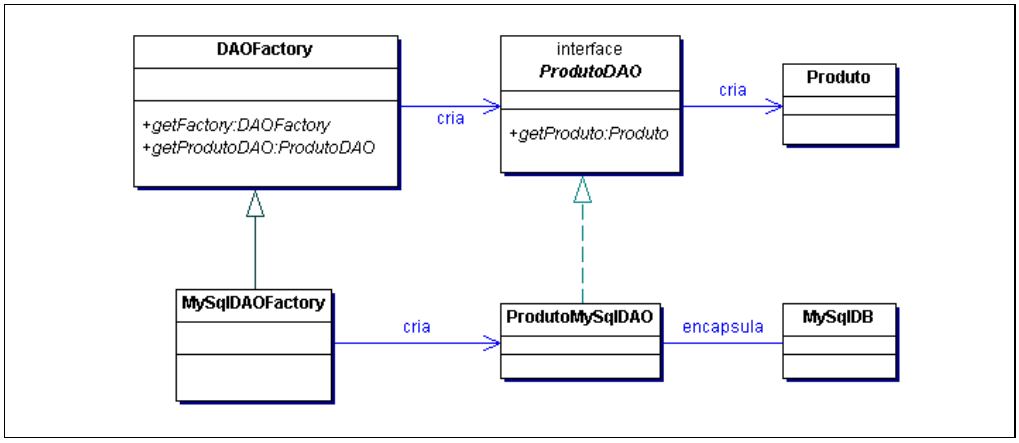
a) contexto: o acesso a dados varia dependendo da fonte de dados;
b) problema: para muitas aplicações o armazenamento persistente é implementado com mecanismos diferentes. Outros sistemas talvez precisem acessar dados em sistemas separados;
c) forças: as forças que motivam a utilização deste padrão acordo com Alur (2002) são:
   - o sistema precisa acessar informações de fontes de dados como bancos de dados, aplicações legadas e sistemas integrados;
   - as *API* de acesso aos dados variam dependendo do fornecedor do produto;
   - a portabilidade dos componentes do sistema é afetada quando mecanismos de acesso a dados específicos são incluídos nos componentes;
   - os componentes do sistema precisam acessar aos dados de forma transparente, independente do tipo de fonte de dados utilizado;

d) solução resumida: utilizar um *Data Access Object* para extrair e encapsular todos os acessos às fontes de dados;

e) conseqüências: as conseqüências segundo Alur (2002) são:
   - permite a transparência, pois os objetos de negócio podem utilizar a fonte de dados sem conhecer os detalhes da sua implementação;
   - permite a migração mais fácil para um banco de dados diferente, pois as alterações serão feitas somente na camada de *DAO*;
   - reduz a complexidade dos objetos que utilizam o *DAO*, visto que estes não conterão código de acesso a dados, como instruções SQL por exemplo;
   - centraliza o acesso aos dados em uma camada separada que isola o resto da aplicação do acesso aos dados;

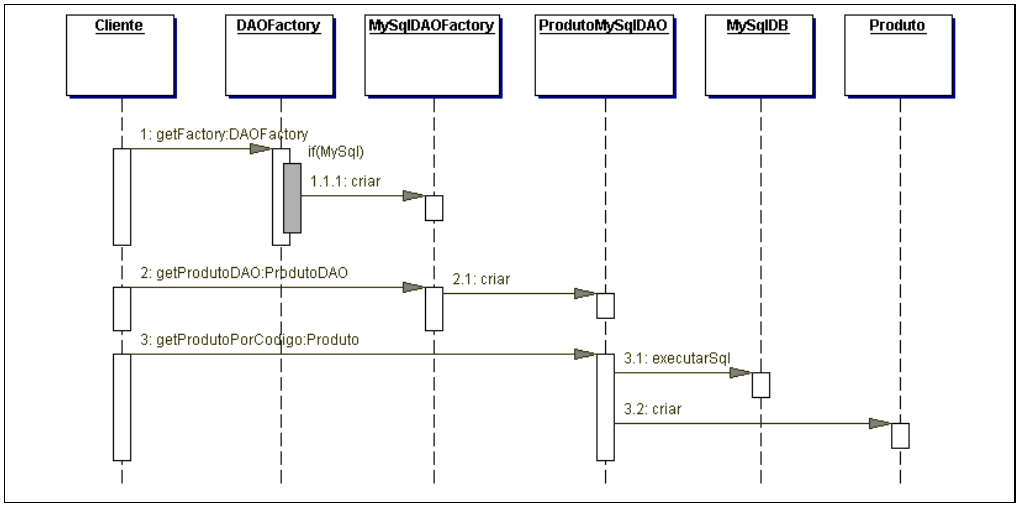
f) estrutura: A Figura 26 mostra o diagrama de classe para o padrão *DAO*. Neste padrão de projeto há uma interface padrão para cada *DAO*, como *ProdutoDAO*. Para cada tipo de fonte de dados há uma implementação desta interface. *ProdutoMySqlDAO*, é uma implementação especifica para um bando de dados chamado *MySql*. Para abstrair a criação destes objetos existe o objeto *DAOFactory* que são fábricas de objetos. *MySqlDAOFactory*, é uma implementação de *DAOFactory* que cria objetos *DAO* para o banco de dados *MySql*. O objeto *DAOFactory* possui um método chamado *getFactory* que retorna uma implementação específica de *DAOFactory*, dependendo de algum parâmetro que pode ser configurado em um arquivo de propriedades. A Figura 27 apresenta o diagrama de seqüência do padrão *DAO*.

Figura 26 - Diagrama de classe *Data Access Object*



Fonte: Baseado em Alur (2002).

Figura 27 - Diagrama de seqüência *Data Access Object*



Fonte: Baseado em Alur (2002).

5 DESENVOLVIMENTO DO SISTEMA

Neste capítulo estão apresentadas as várias etapas do desenvolvimento do sistema, destacando os requisitos, aspectos de especificação e implementação e uma análise dos resultados.

5.1 REQUISITOS DO PROBLEMA

O sistema em questão tem por finalidade apoiar o acadêmico em seu processo de matrícula, gerando sugestões de horários. Para a geração destas sugestões de matrícula, o sistema deve levar em conta vários fatores, entre os quais destacam-se:

a) a necessidade de otimizar o número de disciplinas ou créditos financeiros que o aluno irá cursar, em função da restrição quanto ao número mínimo de créditos que os alunos devem pagar semestralmente. Assim, é desejável que o aluno possa se matricular no maior número possível de disciplinas ou pelo menos em um número que satisfaça este mínimo de créditos;

b) a disponibilidade de horários do aluno para cursar as disciplinas, e sua prioridade em relação às disciplinas que deseja ou precisa cursar em uma determinada fase;

c) o conjunto de pré-requisitos, ou seja, disciplinas que o aluno não pode cursar em função de não ter sido aprovado em determinadas disciplinas anteriores, além de desconsiderar na alocação as disciplinas de fases anteriores do curso em que o aluno já foi aprovado;

d) a possibilidade de o aluno matricular-se em turmas oferecidas a outros cursos, uma vez que em muitos casos a mesma disciplina pode ser oferecida a diversos cursos sendo que o aluno poderá cursá-la desde que encaixe em sua grade de horários.

Sendo assim, o gerador de sugestões de matrícula, deve procurar alocar um conjunto de disciplinas em uma tabela de horários, levando em conta as restrições apresentadas acima. Além disso, o aluno poderá optar por otimizar a resposta do sistema para obter o máximo de créditos possíveis ou o máximo de disciplinas possíveis.

É importante ressaltar que as informações consultadas neste sistema já existem e são cadastrados através do sistema de Registros Acadêmicos da Graduação (RGRA), também referido como sistema acadêmico, desenvolvido e mantido pelo Núcleo de Informática da FURB. No anexo 1 é apresentado o modelo de dados deste sistema.

5.2 ESPECIFICAÇÃO

Neste tópico serão apresentadas a especificação do algoritmo de geração das sugestões baseado na teoria dos grafos e a especificação do problema segundo a notação da UML (*Unified Modeling Language*) (Booch, 2000).

Excluído: a especificação do problema segundo a notação da UML (*Unified Modeling Language*) (Booch, 2000) e

Formatados: Marcadores e numeração

5.2.1 ESPECIFICAÇÃO DO ALGORITMO PARA SUGESTÃO

O problema de geração das sugestões de matrícula pode ser caracterizado como uma busca em grafos. Assim, foi utilizada a busca em árvore, recorrendo-se a técnicas heurísticas para encontrar soluções em tempo razoável. Este modelo foi baseado no modelo proposto por Schwarz (1990), para a resolução de problemas de geração de horários em instituições de ensino.

Formatados: Marcadores e numeração

5.2.1.1 DEFINIÇÃO DO GRAFO

O grafo *G(V, E)* formado para a solução do problema foi definido como segue:

Formatados: Marcadores e numeração

a) *V* é um conjunto de vértices que representam as opções de disciplinas que podem ser alocadas. Cada vértice $v \in V$ é uma opção de disciplina que pode ser alocada para o aluno;
b) *E* é um conjunto de arestas que representam as restrições. Cada aresta $e = (v, w)$ representa uma restrição de incompatibilidade de alocação mútua das opções *v* e *w*.

O conjunto *E* de restrições pode conter dois tipos de restrições:

Formatados: Marcadores e numeração

a) coincidência de horários: as disciplinas alocadas não podem ter aulas no mesmo horário;
b) coincidência de disciplina: uma vez que uma disciplina pode ser oferecida a várias turmas e o aluno pode se matricular em qualquer uma delas, uma sugestão de matrícula não pode conter mais de uma vez a mesma disciplina, mesmo que seus horários não sejam coincidentes.

5.2.1.2 SOLUÇÃO VIÁVEL E SOLUÇÃO ÓTIMA

Como solução viável para o problema estabeleceu-se que dado o grafo *G(V, E)*, conforme definido previamente, deve-se assegurar que duas opções de alocação incompatíveis não façam parte da solução ao mesmo tempo. Para tanto, a condição de que quaisquer dois vértices do subconjunto *S* de opções de alocação nunca sejam adjacentes, deve ser satisfeita. Em outras palavras, *S* deve ser um conjunto independente.

Para se garantir que o maior número possível de disciplinas seja alocado, *S* também deve ser um conjunto independente maximal, ou seja, não pode haver uma solução que contenha um subconjunto das disciplinas de uma outra solução gerada anteriormente.

Uma das condições a serem satisfeitas na busca de uma solução ótima para o problema é procurar alocar as disciplinas de acordo com a disponibilidade e a preferência do aluno. Outra condição é a de que o aluno pode optar por otimizar o número de disciplinas ou o número de créditos financeiros alocados. Para satisfazer estas condições foram utilizados coeficientes de custo nos vértices ou disciplinas do grafo. O curso c_{ik} reflete o desejo do aluno em cursar a disciplina *i* no horário *k*. Há também o coeficiente de custo c_i que indica que a disciplina i_1, dependendo do seu número de horas/aula ou créditos financeiros, é mais promissora do que a disciplina i_2. A Tabela 1 apresenta a definição do cálculo destes coeficientes de custo.

Tabela 1 – Cálculo dos coeficientes de custo

Item	Peso	Opções	Custo
Prioridade atribuída pelo aluno a cada disciplina (c_{ik})	10000	Obrigatória	1
		Desejável	2
		Normal	3
		Aceitável	4
Disponibilidade de horários do aluno (c_{ik})	100	Disponível	0
		Aceitável	nº horas da disciplina aceitáveis para o aluno / total de horas de todas as disciplinas
Tipo de Otimização (c_i)	1	Número de Créditos Financeiros	nº de créditos máximo – nº de créditos da disciplina
		Número de Disciplinas	nº de horas da disciplina

Excluído: ¶
[MELHORAR: FAZER UMA TABELA]

Formatado

O custo c_{ik} determina a prioridade que o aluno irá atribuir a cada disciplina, onde quanto maior a prioridade menor será o custo. Neste custo está somado também um valor

correspondente à disponibilidade de horários do aluno. O custo c_j está associado ao tipo de otimização que o aluno deseja fazer na geração das sugestões. Quando o aluno optar por otimizar para obter o maior número de créditos financeiros, as disciplinas que tiverem o maior número de créditos terão o menor custo. Caso o aluno opte por otimizar para obter o maior número de disciplinas, as disciplinas que tiverem o menor número de horas terão um custo menor.

Estes custos representam uma penalidade associada a cada disciplina, ou seja, quanto maior o custo menos desejável será a alocação desta disciplina.

Cada vértice *v* do grafo tem então um custo associado. Por extensão, a todas as soluções viáveis pode-se associar um custo total. A solução viável de menor custo é a solução ótima do problema.

Formatados: Marcadores e numeração

5.2.1.3 PROCESSO DE ALOCAÇÃO

O processo de busca da solução é um procedimento de geração de subgrafos, visando encontrar conjuntos independentes. Também os conjuntos gerados serão maximais, o que quer dizer que, nenhuma sugestão gerada estará contida em uma solução gerada anteriormente.

A idéia fundamental é explorar a árvore formando um conjunto independente que a cada nível recebe uma nova opção de disciplina, satisfazendo a todas as restrições do problema. O processo continua até alcançar um determinado nível onde:

Formatados: Marcadores e numeração

a) não existem opções restantes;
b) nenhuma das opções restantes apresenta condições de ser agregado ao conjunto em formação, sem vir a repetir conjuntos anteriormente formados.

Para determinar novos conjuntos, retrocede-se a níveis anteriores, a partir dos quais a busca tem prosseguimento. Este processo se repete até que se retroceda ao nível inicial, quando a busca estará terminada.

Para o procedimento de busca, tanto o algoritmo guloso, quanto o algoritmo proposto por Bron e Kerbosh (Bron, 1973) e modificado por Schwarz (1990), foram implementados. No entanto, o algoritmo de Bron e Kerbosh apresentou um tempo de resposta menor (ver

resultados e discussão). No anexo 2 é apresentada a implementação do algoritmo guloso e no anexo 3 é apresentada a implementação do algoritmo de Bron e Kerbosh.

Formatados: Marcadores e numeração

5.2.1.4 ESTRATÉGIAS PARA APERFEIÇOAR A BUSCA

De acordo com Schwarz (1990), algumas estratégias podem ser adotadas para melhorar a performance do processo de busca. As que foram adotadas neste trabalho são:

Formatados: Marcadores e numeração

a) ordenação dos vértices: as opções de alocação de disciplinas são ordenadas em ordem crescente de acordo com o seu custo associado;
b) escolha dos vértices: adotou-se o critério de escolha dos vértices de menor custo em cada nível.

5.2.2 ANÁLISE DO PROBLEMA

Para a especificação do protótipo foram utilizados os seguintes diagramas:

a) diagrama de caso de uso: utilizado para mostrar o caso de uso do sistema e seus atores e relacionamentos;
b) diagrama de classes: adotado para mostrar o conjunto de classes e seus relacionamentos;
c) diagrama de estados: utilizado para mostrar a transição entre as várias telas do sistema.
d) diagrama de seqüência: adotado para mostrar a seqüência da troca de mensagens entre os objetos;

A ferramenta utilizada para fazer a especificação segundo a notação da UML foi o *Together* da *TogetherSoft*.

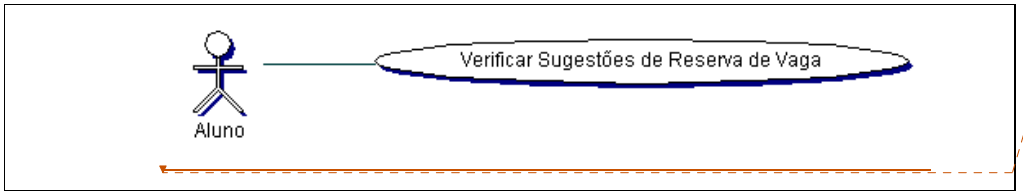
Formatados: Marcadores e numeração

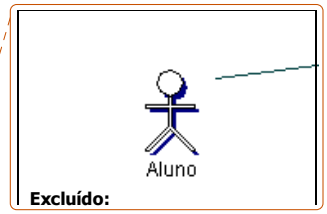
5.2.2.1 DIAGRAMA DE CASOS DE USO

No diagrama da Figura 28 é apresentado o único caso de uso do sistema, onde o aluno verifica as sugestões de reserva de vaga geradas pelo sistema.

Excluído: ¶

Figura 28 – Diagrama de caso de uso





Excluído:

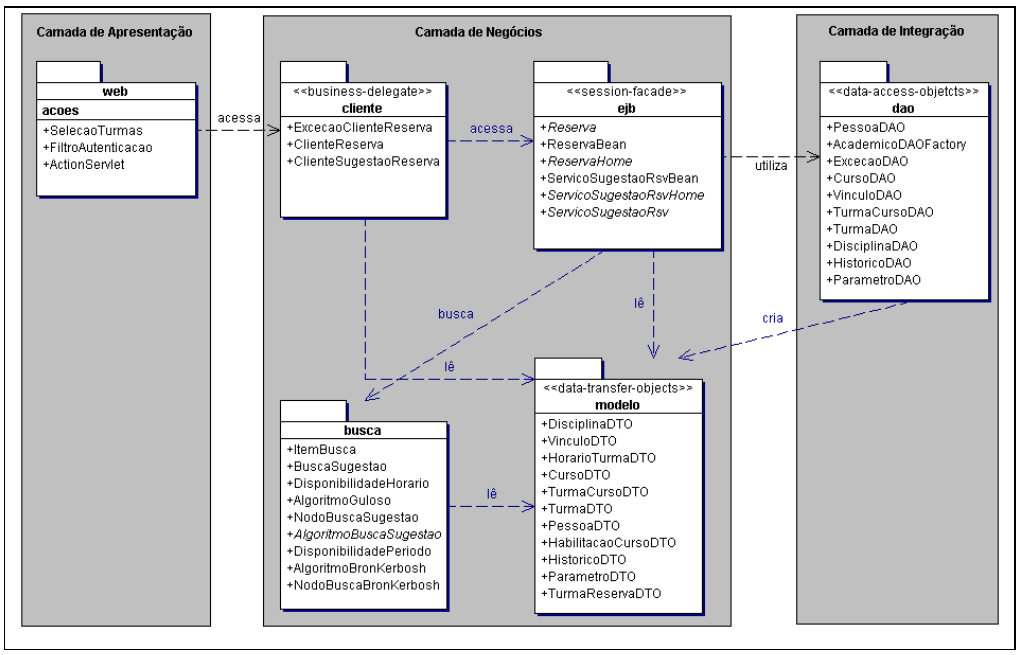
Formatados: Marcadores e numeração

5.2.2.2 DIAGRAMA DE CLASSES

As classes definidas para este sistema estão separadas em pacotes, sendo eles: *ejb*, *cliente*, *dao*, *modelo*, *busca* e *web* (Figura 29).

Excluído: ·····Quebra de página·····

Figura 29 – Estrutura de pacotes



Cada um destes pacotes pertence a uma camada do sistema e contém os padrões de projeto para estas camadas. Segue a descrição da função de cada pacote:

a) pacote *web*: contém a implementação de um *Intercepting Filter*, que verifica a cada solicitação se o usuário efetuou a autenticação e, um *Front Controller* que age como ponto inicial de contato para efetuar o processamento da lógica na camada *web*;

b) pacote *cliente*: neste pacote estão definidos os *Business Delegate*, que são classes que representam os objetos de negócio distribuídos. Estas classes ficam do lado do cliente e simplificam o acesso aos *EJB*;
c) pacote *ejb*: contém a definição dos *EJB* que formam a camada de *Session Facade* do sistema. Os *EJB* definidos neste pacote são uma fachada para a reserva de vaga e para o serviço de elaboração de sugestões de matrícula;
d) pacote *busca*: contém a implementação dos algoritmos de busca e elaboração de sugestões;
e) pacote *dao*: contém os objetos de acesso a dados (padrão *Data Access Objects*), que extraem e encapsulam o acesso para as tabelas do sistema acadêmico;
f) pacote *modelo*: os objetos definidos neste pacote encapsulam os dados das tabelas do sistema acadêmico (padrão *Data Transfer Object*), que são passados da camada de negócios para a camada de apresentação.

A seguir serão descritos os pacotes mais detalhadamente e serão apresentados os diagramas de classes para cada pacote.

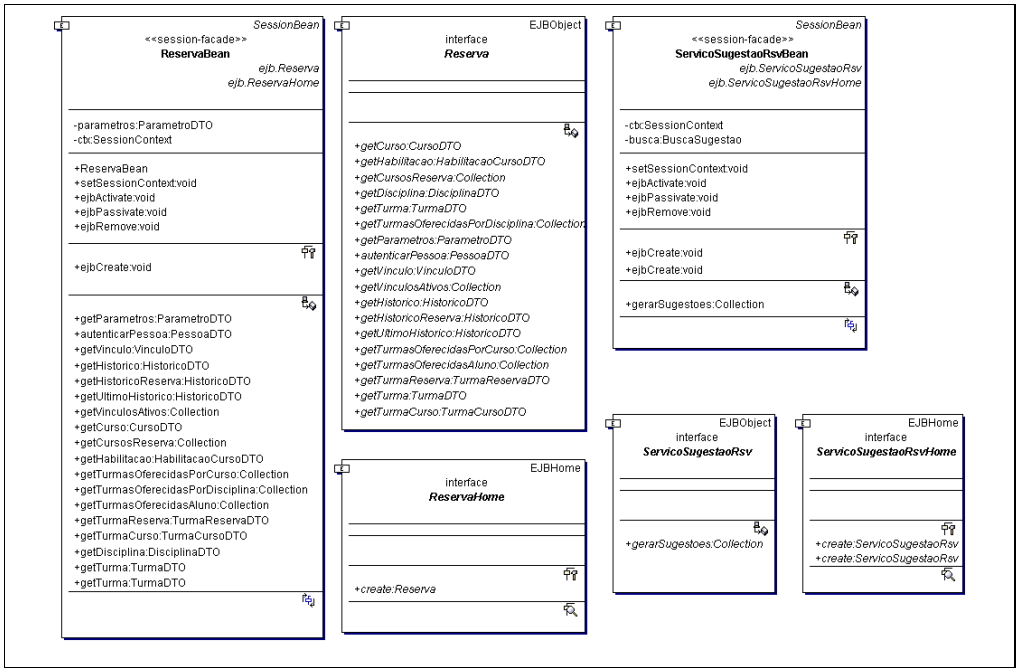
5.2.2.2.1 PACOTE EJB

Formatados: Marcadores e numeração

Este pacote contém os *session beans* que formam a camada de *Session Facade* do sistema. Seu diagrama de classes é apresentado na Figura 30.

Excluído: ¶

Figura 30 – Diagrama de classes do pacote *ejb*



As classes possuem as seguintes finalidades:

a) *ReservaBean*: é a classe de implementação do *EJB* que provê os serviços para a reserva de vaga. Este *EJB* contém os métodos para obter a lista de disciplinas oferecidas, autenticar um aluno e outras funções utilitárias;

b) *Reserva*: é a *remote interface* do *ReservaBean*;

c) *ReservaHome*: é a *home interface* do *ReservaBean*;

d) *ServicoSugestaoRsvBean*: é a classe de implementação do *EJB* que provê o serviço de geração das sugestões de reserva de vaga;

e) *ServicoSugestaoRsv*: é a *remote interface* do *ServicoSugestaoRsvBean*;

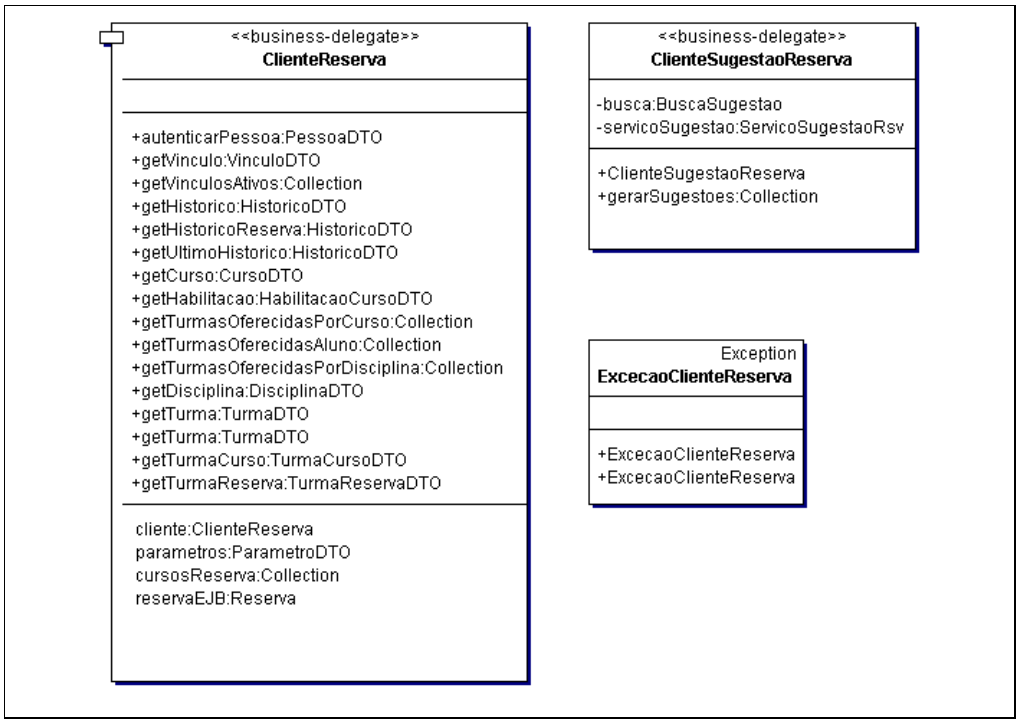
f) *ServicoSugestaoRsvHome*: é a *home interface* do *ServicoSugestaoRsvBean*.

Formatados: Marcadores e numeração

5.2.2.2.2 PACOTE CLIENTE

Esta pacote contém os *business delegate* que abstraem o acesso aos dois *EJB* definidos anteriormente (*ReservaBean* e *ServicoSugestaoRsvBean)*, para os clientes (Figura 31).

Figura 31 – Diagrama de classes do pacote *cliente*



Estas classes têm a seguinte função:

a) *ClienteReserva*: é o *business delegate* do *EJB ReservaBean*. Os clientes acessam esta classe que por sua vez delega todas as chamadas de método para o *EJB*;

b) *ClienteSugestaoReserva*: fornece acesso ao *EJB ServicoSugestaoRsvBean*;

c) *ExcecaoClienteReserva*: exceção que pode ser gerada pelos métodos dos *business delegate*.

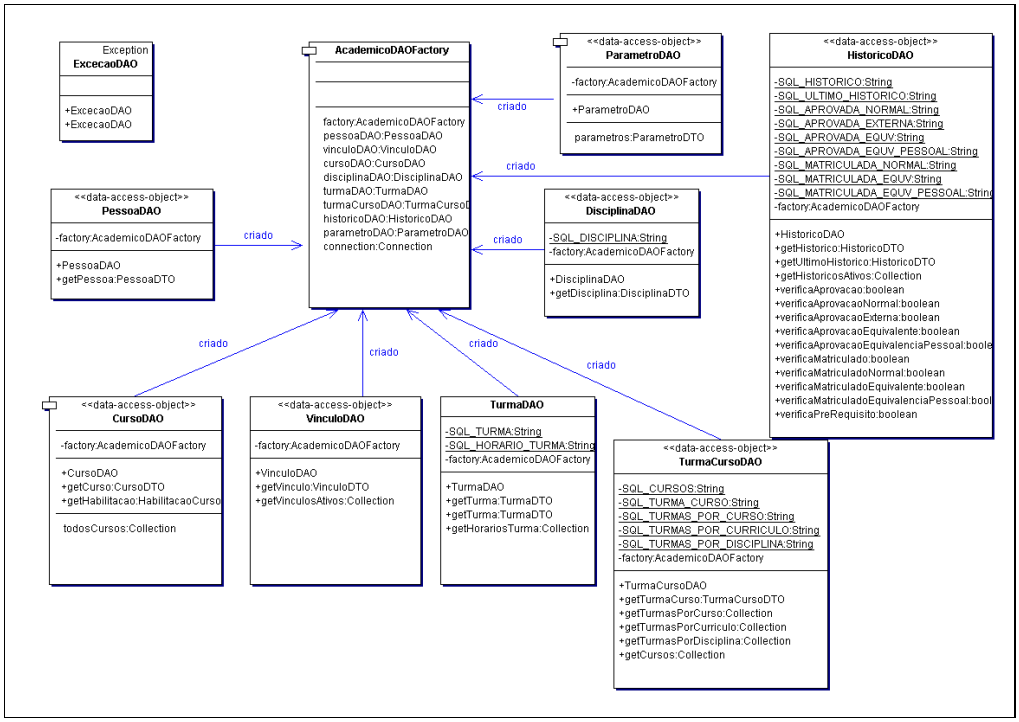
5.2.2.2.3 PACOTE DAO

Formatados: Marcadores e numeração

Neste pacote estão definidos todos os *data access objects.* Estes objetos abstraem o acesso às tabelas do banco de dados acadêmico. O diagrama de classes deste pacote é apresentado na Figura 32.

Excluído: ¶

Figura 32 – Diagrama de classes do pacote *dao*



Segue a descrição de cada classe:

a) *PessoaDAO*: é a classe de acesso a dados para a tabela `Pessoa`, que contém a identificação dos alunos na universidade;

b) *CursoDAO*: é a classe de acesso a dados para as tabelas `Curso_Graduacao` e `Habilitacao_Curso_Graduacao`, onde estão cadastrados os cursos da graduação e as várias habilitações que o aluno pode obter em cada curso;

c) *VinculoDAO*: é a classe de acesso a dados para a tabela `Vinculo`. Um vínculo representa a matrícula de um aluno em um curso;

d) *TurmaDAO*: esta classe fornece acesso aos dados das tabelas `Turma_Graduacao` e `Turma_Horario_Graduacao`. Uma turma representa a oferta de uma disciplina, num determinado semestre[1];

e) *TurmaCursoDAO*: esta classe fornece acesso à tabela `Turma_Curso_Graduacao`. Nesta tabela estão informados os cursos para os quais uma "turma" é oferecida;

f) *DisciplinaDAO*: é a classe de acesso à tabela `Disciplina_Graduacao`, que contém o cadastro de todas as disciplinas;

g) *ParametroDAO*: fornece acesso aos parâmetros cadastrados na tabela `Parametro_Graduacao`, como o semestre atual e semestre para reserva de vaga;

h) *HistoricoDAO*: classe que fornece acesso à tabela `Historico_Graduacao` e `Historico_Disciplina_Graduacao`. Nestas tabelas está armazenado o histórico dos alunos por semestre;

i) *AcademicoDAOFactory*: classe que é utilizada para abstrair a criação dos objetos *DAO*, fornecendo um meio de trocar a implementação dos *DAO* de maneira transparente;

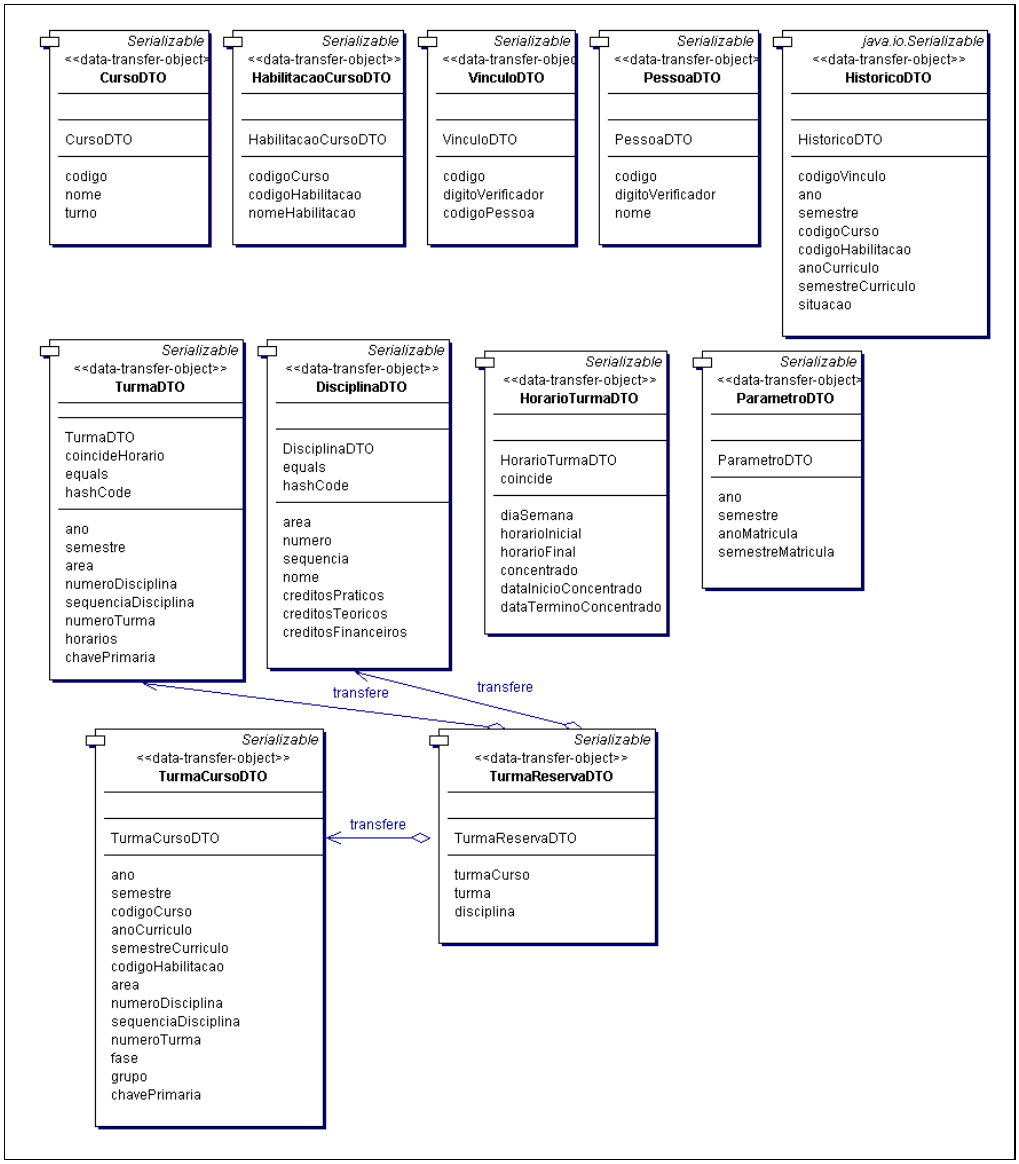
5.2.2.2.4 PACOTE MODELO

Formatados: Marcadores e numeração

Neste pacote estão os *data transfer objects*, estes objetos servem para trocar informações entre a camada de negócios e os clientes. A Figura 33 apresenta o diagrama de classes deste pacote.

[1] No sistema acadêmico, uma "turma" designa a oferta de uma disciplina, pois uma disciplina pode ser oferecida a vários cursos, ou dividida em várias turmas em um mesmo curso. Entretanto, do ponto de vista de um aluno, uma turma representa apenas uma disciplina. Por isso, neste trabalho o termo disciplina será utilizado muitas vezes para se referir a uma turma.

Figura 33 – Diagrama de classes do pacote *modelo*



Estas classes têm as seguintes finalidades:

a) *CursoDTO*: esta classe representa os dados referentes a um curso armazenados na tabela `Curso_Graduacao`;

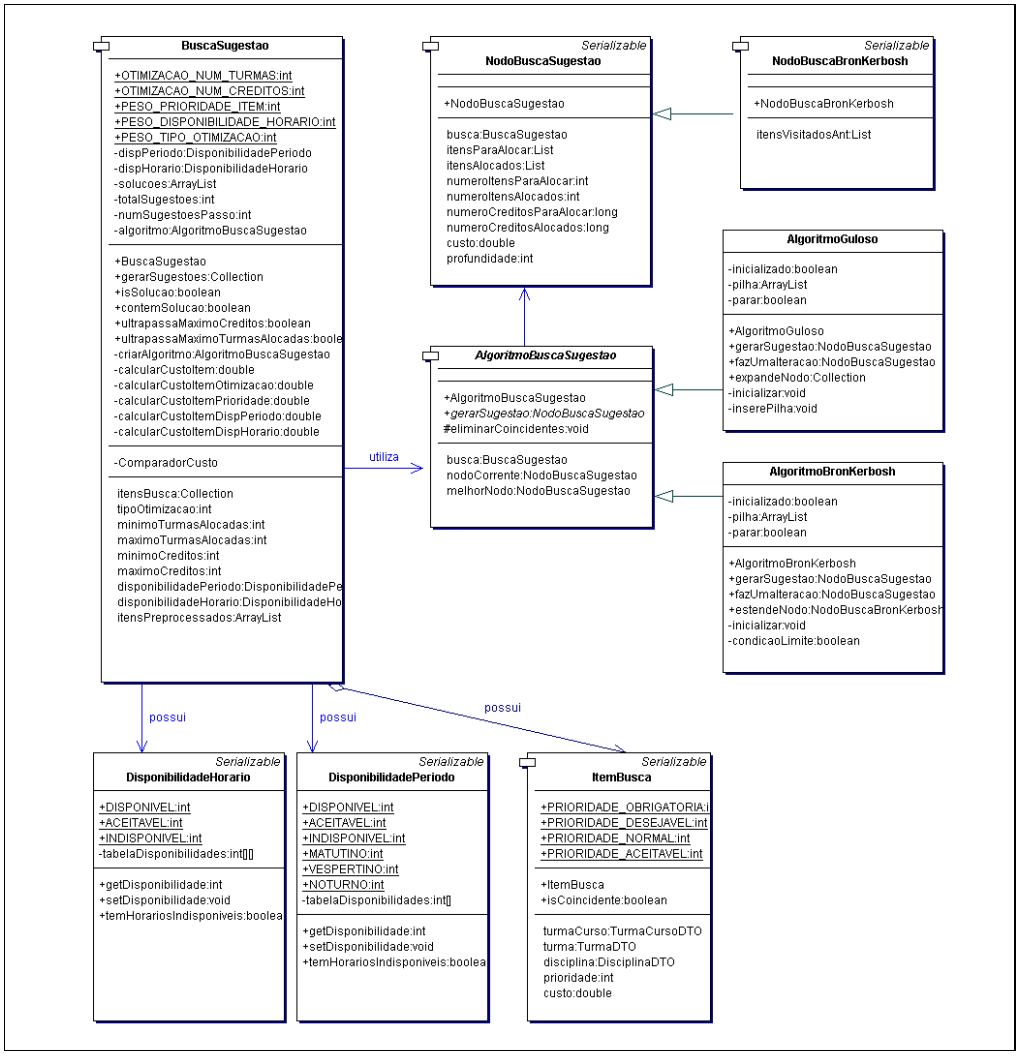
b) *HabilitacaoCursoDTO*: esta classe representa uma habilitação dentro de um curso. Estes dados estão armazenados na tabela `Habilitacao_Curso_Graduacao`;
c) *VinculoDTO*: esta classe representa os dados da tabela `Vinculo`;
d) *PessoaDTO*: esta classe representa as informações de identificação de uma pessoa, armazenadas na tabela `Pessoa`;
e) *HistoricoDTO*: esta classe contém os dados que estão armazenados na tabela `Historico_Graduacao`;
f) *TurmaDTO*: esta classe representa os dados da tabela `Turma_Graduacao`;
g) *DisciplinaDTO*: esta classe representa uma disciplina e contém as informações armazenadas na tabela `Disciplina_Graduacao`;
h) *HorarioTurmaDTO*: esta classe contém os dados referentes à tabela `Turma_Horario_Graduacao`, que contém os horários em que uma turma é oferecida;
i) *ParametroDTO*: esta classe contém os parâmetros cadastrados para a reserva de vaga, armazenados na tabela `Parametro_Graduacao`;
j) *TurmaCursoDTO*: esta classe contém os dados da tabela `Turma_Curso_Graduacao`;
k) *TurmaReservaDTO*: esta classe é um *data transfer object* que já possui agregados a ela os objetos *DisciplinaDTO*, *TurmaDTO* e *TurmaCursoDTO*, para a transmissão desses três objetos, que geralmente são utilizados em conjunto, para o cliente em uma única chamada;
l) *TurmaCursoPK*: representa a chave primária de um objeto *TurmaCursoDTO* para facilitar a manipulação por clientes *web*.

5.2.2.2.5 PACOTE BUSCA

Formatados: Marcadores e numeração

Neste pacote está a especificação do algoritmo de elaboração das sugestões. A Figura 34 apresenta o diagrama de classes deste pacote.

Figura 34 – Diagrama de classes do pacote *busca*



Estas classes têm as seguintes finalidades:

a) *BuscaSugestao*: esta classe representa uma busca por sugestões. Ela recebe um conjunto de objetos *ItemBusca*, as restrições, e a disponibilidade de horários do aluno e devolve um conjunto de sugestões;

b) *DisponibilidadeHorario*: esta classe representa a disponibilidade do aluno por horário;

c) *DisponibilidadePeriodo*: esta classe representa a disponibilidade do aluno por período (matutino, vespertino, noturno);

d) *ItemBusca*: esta classe representa uma disciplina ou turma com a prioridade atribuída pelo aluno e que será pesquisada na busca. Cada *ItemBusca* pode ser considerado como um vértice em um grafo que é pesquisado;

e) *AlgoritmoBuscaSugestao*: esta é uma superclasse abstrata de um algoritmo de busca de sugestões;

f) *AlgoritmoGuloso*: esta é a implementação do algoritmo guloso para a geração de sugestões;

g) *AlgoritmoBronKerbosh*: esta é a implementação do algoritmo de Bron e Kerbosh para a geração de sugestões;

h) *NodoBuscaSugestao*: esta classe representa o estado da busca a cada iteração, o algoritmo guloso utiliza esta implementação;

i) *NodoBuscaBronKerbosh*: esta classe é uma especialização de *NodoBuscaSugestao* para o algoritmo de Bron e Kerbosh, que contém um conjunto para armazenar os itens que já foram visitados.

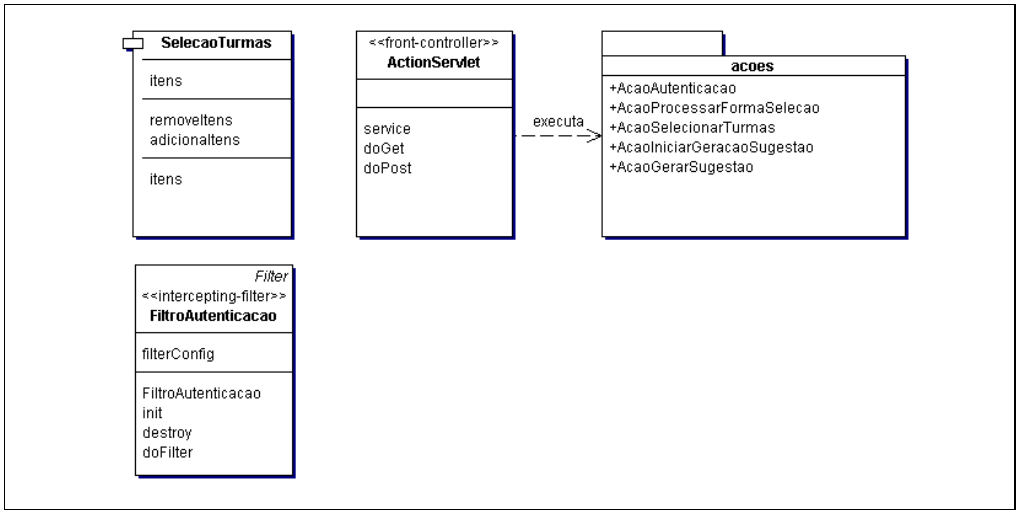
5.2.2.2.6 PACOTE WEB

Formatados: Marcadores e numeração

Neste pacote estão as classes utilizadas na camada web juntamente com as páginas *JSP* e os *servlets*. A Figura 35 apresenta o diagrama de classes deste pacote.

Excluído: ·····Quebra de página·····

Figura 35 – Diagrama de classes do pacote *web*



A função de cada classe é descrita a seguir:

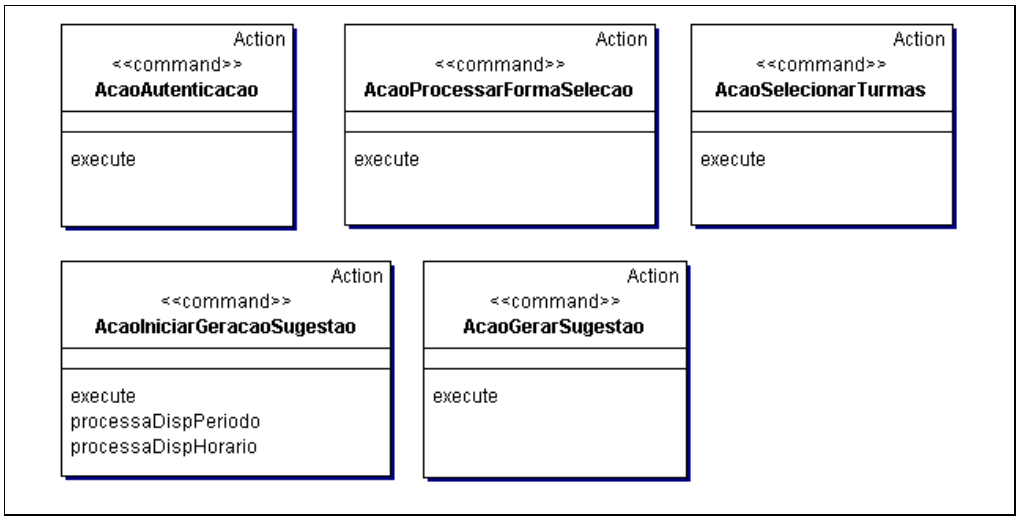
a) *SelecaoTurmas*: classe que contém as disciplinas ou turmas que foram selecionadas para gerar sugestões;

b) *FiltroAutenticação*: é o *Intercepting Filter* para a aplicação que verifica em todos os acessos se o usuário efetuou a autenticação;

c) *ActionServlet*: é o *servlet* que processa a lógica de controle da camada web. Este *servlet* se adequa ao padrão *Front Controller* e processa a sua lógica através das ações (padrão *Command*). Este servlet é definido na API *Struts* (Burns, 2002).

As ações ou comandos executados pelo *ActionServlet*, estão definidos no pacote *acoes*, que é um sub-pacote de *web*. A Figura 36 apresenta o diagrama de classes para este pacote.

Figura 36 – Diagrama de classes para o pacote *acoes*



As classes deste pacote são descritas a seguir:

a) *AcaoAutenticacao*: esta ação é executada para autenticar o aluno no sistema validando o seu código e senha, informados na tela de autenticação;

b) *AcaoProcessarFormaSelecao*: esta a ação irá selecionar as disciplinas que o aluno ainda não obteve aprovação e para as quais já possui os pré-requisitos;

c) *AcaoSelecionarTurmas*: esta ação seleciona as disciplinas que o aluno escolheu e armazena na sessão para uso posterior na geração das sugestões;

d) *AcaoIniciarGeracaoSugestoes*: esta ação inicia uma nova busca de sugestões com as disciplinas selecionadas;

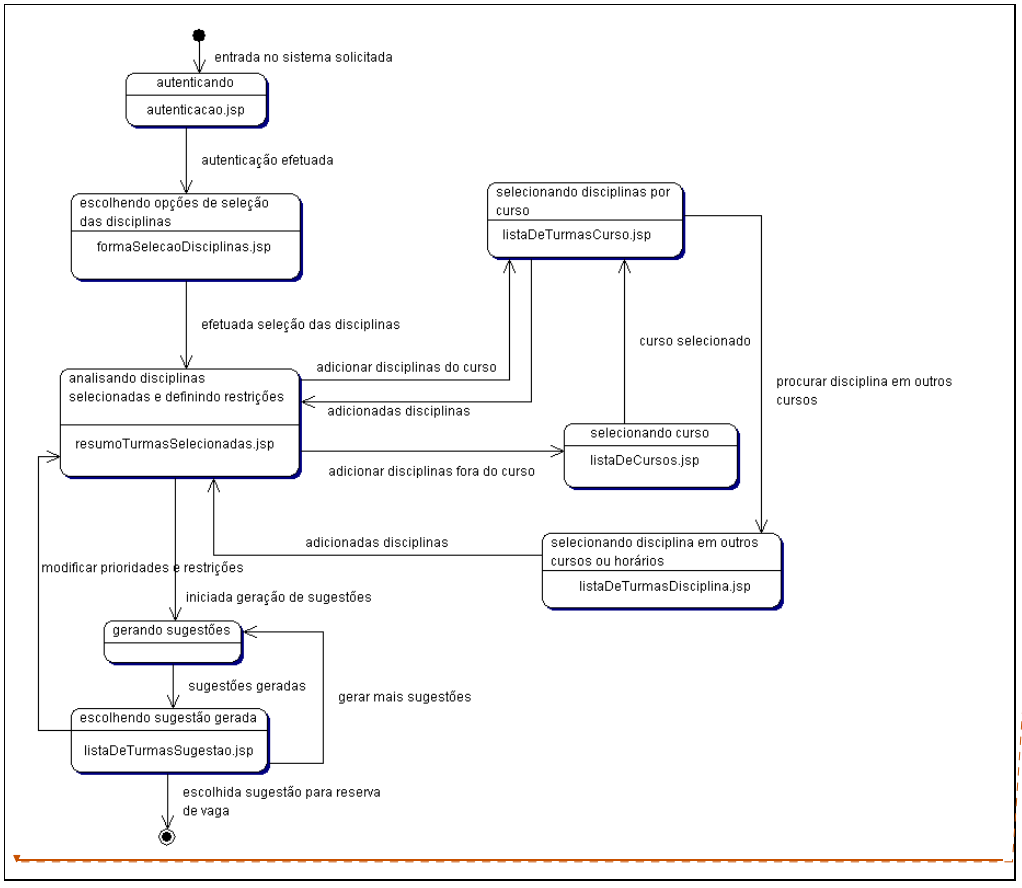
e) *AcaoGerarSugestoes*: após iniciada uma busca esta ação comanda a busca de sugestões para gerar as próximas sugestões.

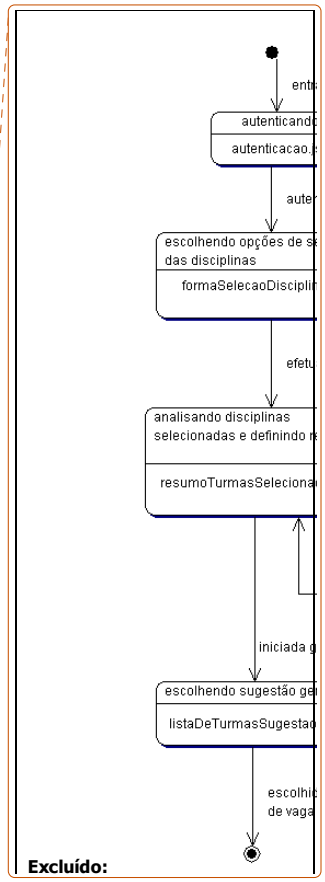
Formatados: Marcadores e numeração

5.2.2.3 DIAGRAMA DE ESTADOS

O diagrama de estados apresentado na Figura 37 mostra a seqüência de passos que o aluno irá seguir para gerar as sugestões. Neste diagrama as telas ou páginas JSP do sistema foram representadas como estados. Há também o estado "Gerando Sugestões", que é o estado em que o sistema encontra-se elaborando as sugestões de reserva de vaga através do algoritmo de busca.

Figura 37 – Diagrama de estados





Excluído:

Neste fluxo, o aluno irá inicialmente se autenticar no sistema e após isso irá para uma tela onde poderá informar as opções de seleção das disciplinas que deseja incluir na geração das sugestões. Estas opções visam incluir somente disciplinas da grade curricular do curso do aluno que ele ainda não obteve aprovação e para as quais já possua os pré-requisitos. Feita a seleção das disciplinas será apresentado um resumo para que o aluno possa definir prioridades para estas disciplinas e informar as restrições. Neste momento o aluno poderá iniciar a geração das sugestões ou adicionar mais disciplinas do seu curso ou de outros cursos. O aluno poderá ainda, incluir disciplinas que façam parte da grade curricular do seu curso, mas, que estão sendo oferecidas em horários diferentes. Depois de geradas as sugestões, o aluno escolhe uma delas e efetua a sua reserva de vaga, terminando assim o processo.

Formatados: Marcadores e numeração

5.2.2.4 DIAGRAMAS DE SEQÜÊNCIA

Os diagramas de seqüências demonstram como é feita a troca de mensagens entre as classes (objetos). Dentre transições definidas na Figura 37, algumas envolvem apenas a solicitação de uma página (tela), no entanto, em outras há um processamento envolvido. Para cada uma destas transições foi gerado um diagrama de seqüência correspondente, conforme apresentado a seguir.

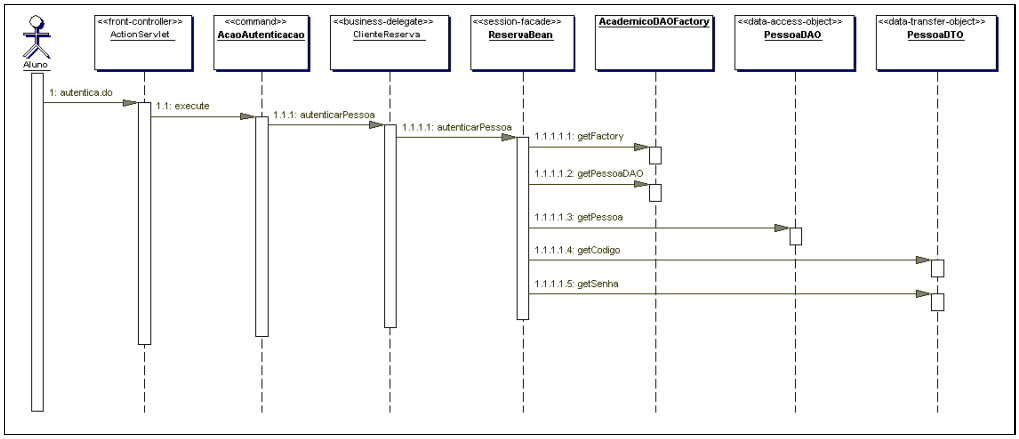
Formatados: Marcadores e numeração

5.2.2.4.1 AUTENTICAÇÃO

Este diagrama inicia quando o aluno informa seu código de identificação pessoal e senha em um formulário *HTML* e submete este formulário para validação. Este formulário é encaminhado para o *servlet ActionServlet* que recebe todas as solicitações do aluno e executa um comando *AcaoAutenticação* que por sua vez chama o método *autenticarPessoa* no objeto *ClienteReserva*. A lógica de validação está na classe *ReservaBean* que acessa o banco de dados através do objeto *PessoaDAO* e confere o código e senha informados (Figura 38).

Excluído: ¶

Figura 38 – Diagrama de seqüência da autenticação

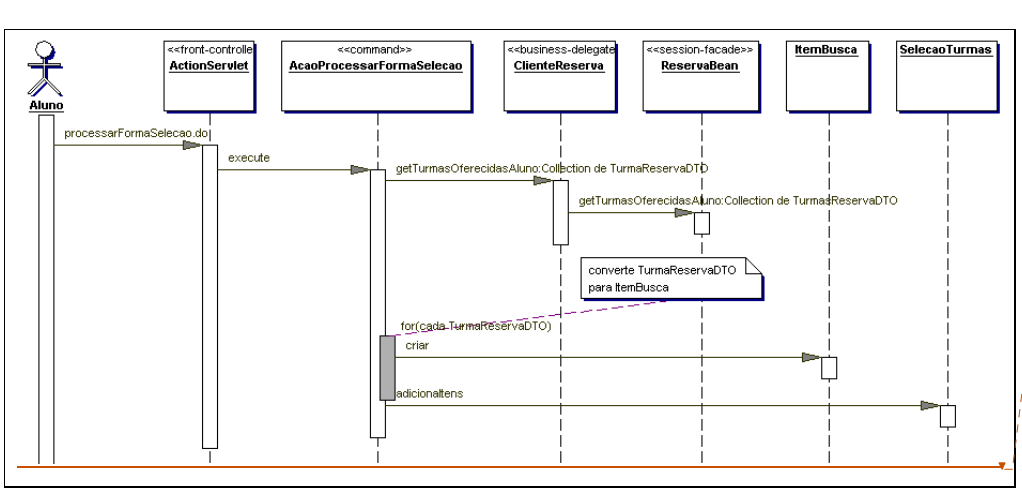


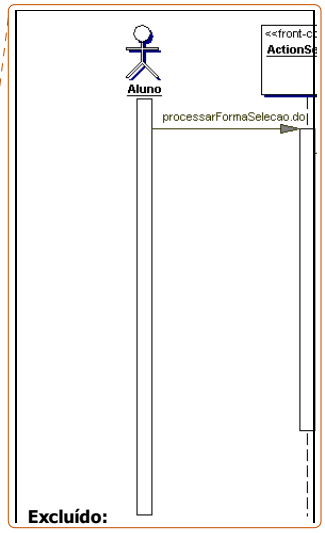
Formatados: Marcadores e numeração

5.2.2.4.2 SELEÇÃO DAS DISCIPLINAS

No momento em que o aluno escolhe as opções de seleção das disciplinas, é chamado o método *getTurmasOferecidasAluno* no *EJB ReservaBean*, que obtém todas as turmas ou disciplinas que são oferecidas para o aluno no seu curso e currículo. Este procedimento retorna um conjunto de objetos *TurmaReservaDTO*, que são então convertidos em objetos do tipo *ItemBusca* e adicionados para a seleção. A Figura 39 apresenta este diagrama de seqüência.

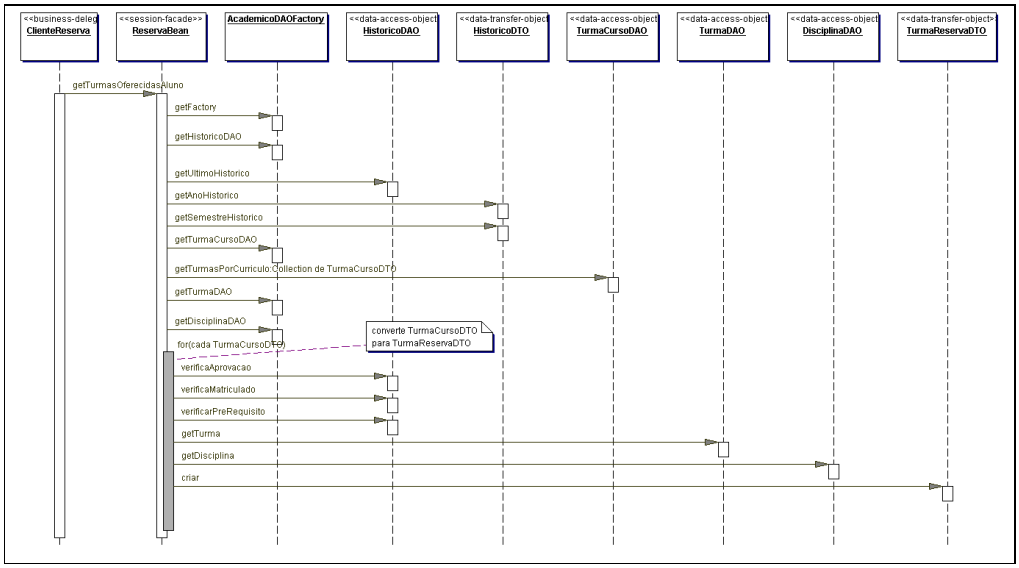
Figura 39 – Diagrama de seqüência seleção das disciplinas





O método *getTurmasOferecidasAluno* representa um serviço provido pelo s*ession facade ReservaBean*. O diagrama de seqüência para este serviço é mostrado na Figura 40.

Figura 40 – Diagrama de seqüência obtendo disciplinas por aluno



Neste diagrama o *EJB ReservaBean* obtém informações do histórico do aluno e por meio do *data access object TurmaCursoDAO* lê do banco de dados todas as disciplinas que são oferecidas de acordo com o curso e grade curricular que o aluno está seguindo. Este procedimento verifica se o aluno está aprovado, matriculado ou ainda não possui os pré-requisitos para cursar uma disciplina, descartando tais disciplinas. Após isso, são criados os objetos *TurmaReservaDTO* que serão retornados.

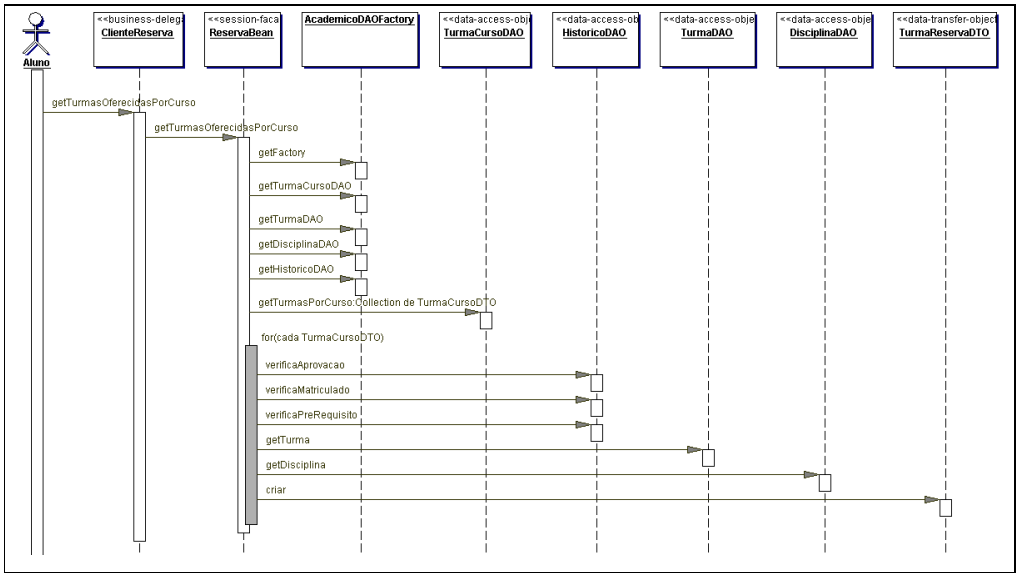
5.2.2.4.3 ADICIONANDO DISCIPLINAS

Formatados: Marcadores e numeração

Depois de efetuada a seleção das disciplinas da grade curricular do aluno, ele ainda poderá optar por adicionar mais disciplinas do seu curso ou de outros cursos. O diagrama da Figura 41 apresenta a seqüência de mensagens para obter as disciplinas oferecidas a um curso específico.

Excluído: ¶

Figura 41 - Diagrama de seqüência obtendo disciplinas por curso

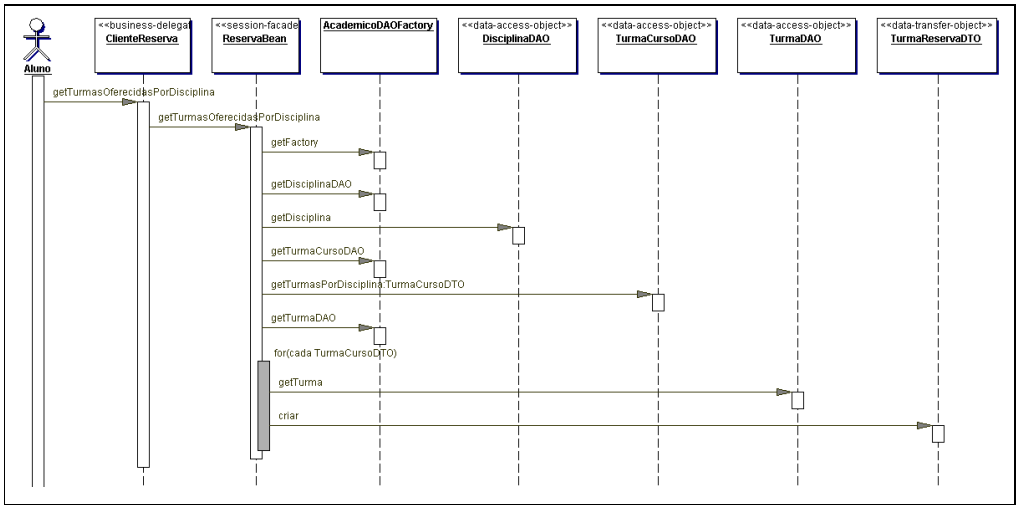


Neste diagrama, o método *getTurmasOferecidasPorCurso* do *session facade ReservaBean*, retorna um conjunto de objetos *TurmaReservaDTO*. O processo envolve acessar o objeto *TurmaCursoDAO* para obter a relação de disciplinas que é oferecida a um determinado curso (objetos *TurmaCursoDTO*), opcionalmente verificar se o aluno pode cursar esta disciplina (no *HistoricoDAO*) e, obter os objetos *TurmaDTO* (em *TurmaDAO*) e *DisciplinaDTO* (em *DisciplinaDAO*) para construir o objeto *TurmaReservaDTO*.

A partir de uma disciplina, o aluno pode procurar todas as turmas que foram criadas para esta disciplina em outros cursos e horários. Para obter as turmas de uma determinada disciplina, o *ReservaBean* possui o método *getTurmasOferecidasPorDisciplina* que retorna um conjunto de objetos *TurmaReservaDTO* a partir do código de uma disciplina informada como parâmetro. A Figura 42 apresenta o diagrama de seqüência para este procedimento.

Excluído: ¶

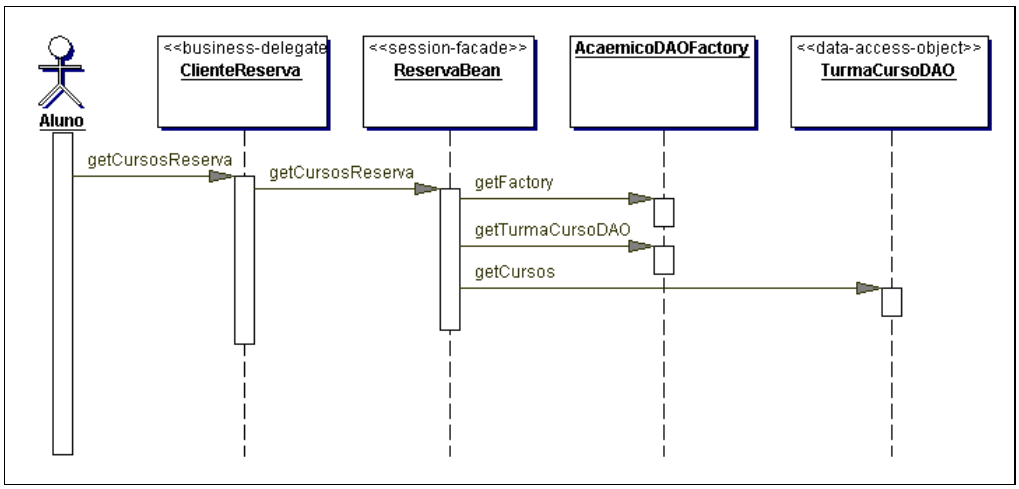
Figura 42 - Diagrama de seqüência obtendo turmas de uma disciplina



Quando o aluno optar por selecionar disciplinas de fora do seu curso, deve ser apresentada a ele uma relação dos cursos que estão disponíveis para a reserva de vaga. A Figura 43 apresenta o diagrama de seqüência do procedimento que retorna esta relação. Neste diagrama, o método *getCursos* de *TurmaCursoDAO* retorna um conjunto de objetos *CursoDTO*, que contém apenas os cursos que possuem turmas ou disciplinas oferecidas para eles.

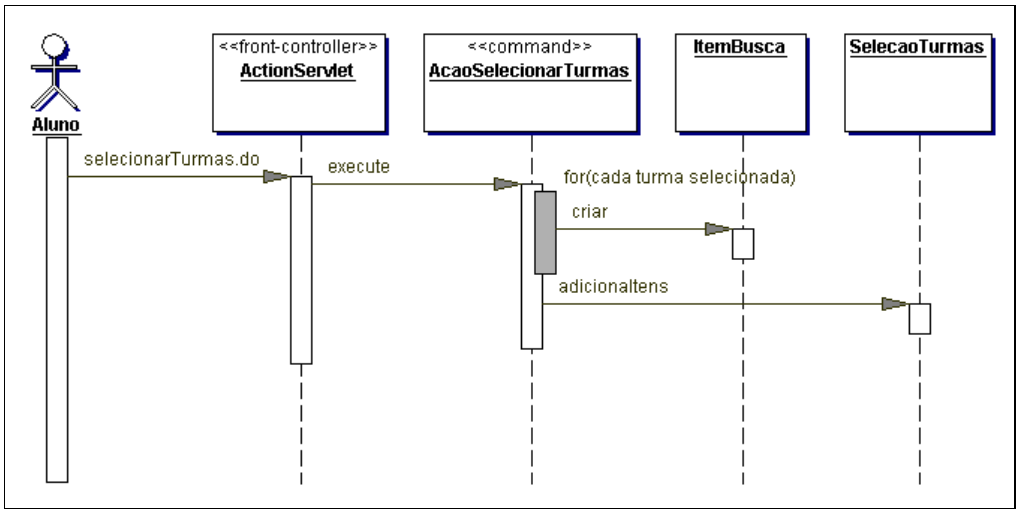
Excluído: Quebra de página

Figura 43 - Diagrama de seqüência obtendo cursos disponíveis para reserva de vaga



Depois de o aluno escolher um conjunto de disciplinas para adicionar à seleção, será executado o procedimento da Figura 44, que pega os parâmetros passados para o *ActionServlet*, converte-os em objetos *ItemBusca* e adiciona no objeto *SelecaoTurmas*. O fluxo de execução deste procedimento está implementado no objeto *AcaoSelecionarTurmas* que é executado pelo *ActionServlet*.

Figura 44 - Diagrama de seqüência selecionando disciplinas



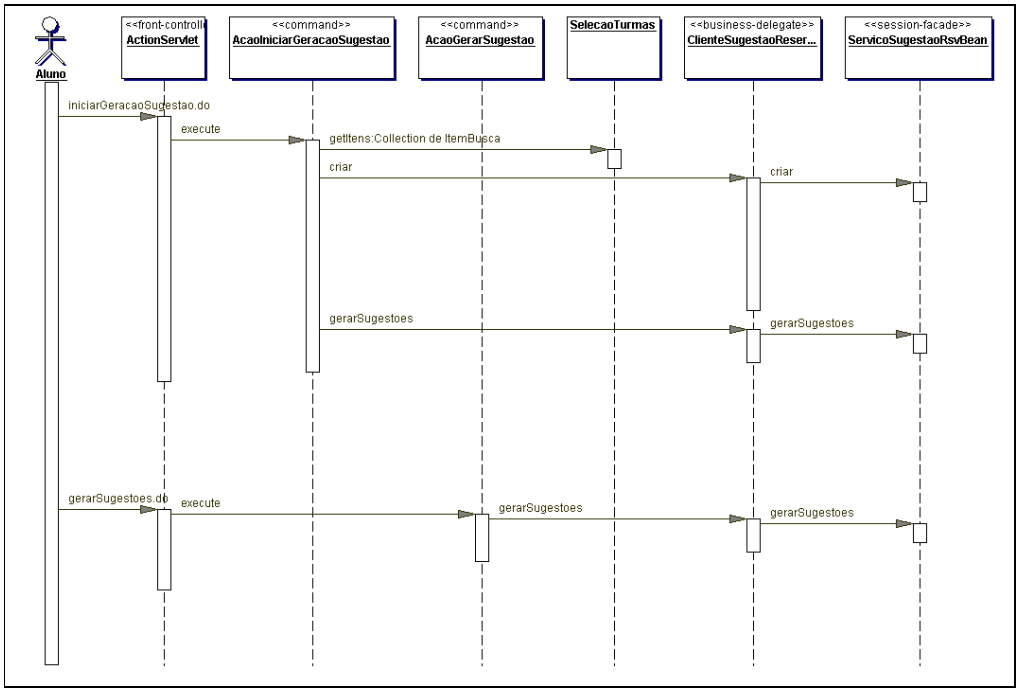
5.2.2.4.4 GERAÇÃO DAS SUGESTÕES

Depois de selecionado um conjunto de disciplinas o aluno informa as restrições e preferências e inicia a geração das sugestões. Neste momento o *ActionServlet* irá executar o comando *AcaoIniciarGeracaoSugestao* que obtém todas as disciplinas selecionadas do objeto *SelecaoTurmas* e cria uma nova busca de sugestões. Após criada a busca serão geradas inicialmente dez sugestões e apresentadas ao aluno. O aluno poderá então escolher uma das sugestões para a reserva de vaga ou poderá gerar mais sugestões. Para gerar mais sugestões o *ActionServlet* irá executar o comando *AcaoGerarSugestao* que acessa a busca criada anteriormente e gera as próximas sugestões. O acesso dos clientes ao processo de geração de sugestões é feito através do *business delegate ClienteSugestaoReserva* e do *session facade ServicoSugestaoRsvBean*. A Figura 45 apresenta este diagrama de seqüência.

Formatados: Marcadores e numeração

Excluído: ¶

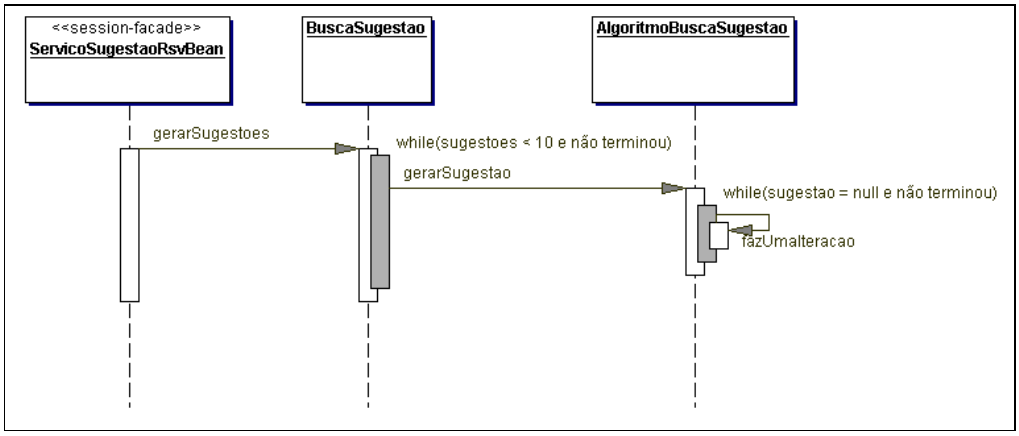
Figura 45 – Diagrama de seqüência geração das sugestões



O processo de busca não está implementado diretamente no *EJB ServicoSugestaoRsvBean* mas, em uma camada inferior. A Figura 46 apresenta o diagrama de seqüência para o processo de busca. Neste diagrama, o *session facade* acessa o objeto *BuscaSugestao* que contém todas as definições para uma busca de sugestões de matrícula. O método *gerarSugestoes* de *BuscaSugestao* gera até dez sugestões, se for possível, e retorna para o cliente. A técnica de elaboração das sugestões está implementada em uma classe do tipo *AlgoritmoBuscaSugestao*, que pode ser *AlgoritmoGuloso* ou *AlgoritmoBronKerbosh*.

Excluído: ¶

Figura 46 – Diagrama de seqüência processo de busca

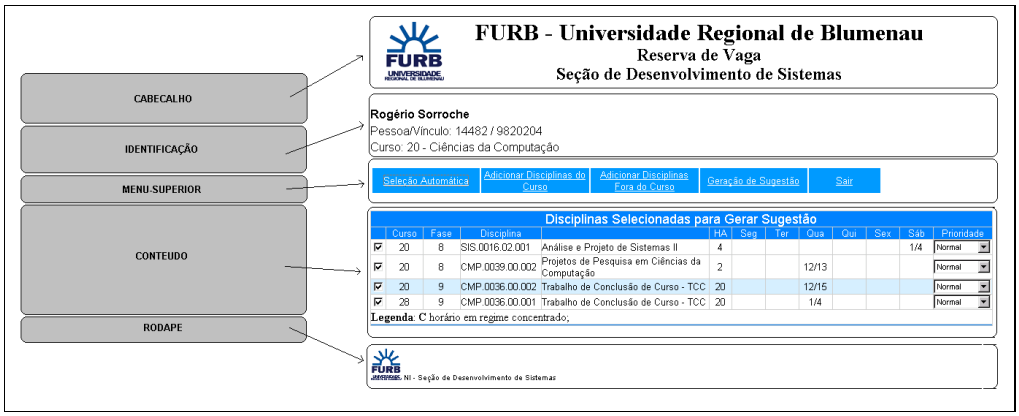


5.2.3 DEFINIÇÃO DA INTERFACE DO USUÁRIO

Para a criação das telas do sistema, foi definido de acordo com o padrão de projeto *Composite View*, uma *template* para as páginas *JSP*. Esta *template* é uma página *JSP* que especifica o layout de todas as páginas do sistema. A *template* define cinco seções em uma página e utiliza *custom tags* para incluir outras páginas JSP nestas seções. A Figura 47 apresenta o layout produzido pela *template*.

Excluído: ¶

Figura 47 – Definição de uma template para páginas do sistema



Utilizando esta *template*, o conteúdo das seções cabeçalho, identificação, menu-superior e rodapé, foram definidos somente uma vez para todo o sistema, como páginas *JSP*,

sendo reutilizadas várias vezes. Isto torna necessário definir novamente somente a seção conteúdo, que se modifica para cada tela do sistema.

5.3 IMPLEMENTAÇÃO

A implementação do sistema foi dividida em três fases:

a) camada de negócios: responsável por prover os serviços para os clientes, faz o acesso aos dados e devolve para os clientes. Esta parte do sistema foi desenvolvida utilizando *EJB*;
b) camada de apresentação: páginas *JSP*, responsáveis pela interface com o usuário e um *servlet* responsável pela lógica de controle da aplicação;
c) algoritmo de busca: recebe um conjunto de disciplinas, opções e restrições para serem alocadas e fornece sugestões de matrícula com estas disciplinas.

5.3.1 TÉCNICAS E FERRAMENTAS UTILIZADAS

Para o desenvolvimento deste trabalho foram utilizadas as seguintes ferramentas:

a) *Together*: ferramenta case *UML*, que dá suporte a *EJB* e padrões de projeto;
b) servidor de aplicações *J2EE JBoss*: o servidor *JBoss* é um servidor que suporta a execução de *EJB* e *servlets*;
c) *JDeveloper*: ferramenta de desenvolvimento para Java;
d) *Oracle*: é o banco onde estão todas as tabelas do sistema acadêmico de graduação.

5.3.2 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Neste tópico serão apresentadas as telas do protótipo, bem como os detalhes da operabilidade do protótipo implementado.

Basicamente, o sistema envolve o fato do aluno elencar um conjunto de disciplinas que deseja incluir nas sugestões, informar as opções que deseja incluir na elaboração (tipo de otimização, restrições e disponibilidade de horários) e então gerar as sugestões. A Figura 48 apresenta a tela inicial onde o aluno faz a autenticação no sistema.

Excluído: ¶

Figura 48 – Tela inicial do sistema

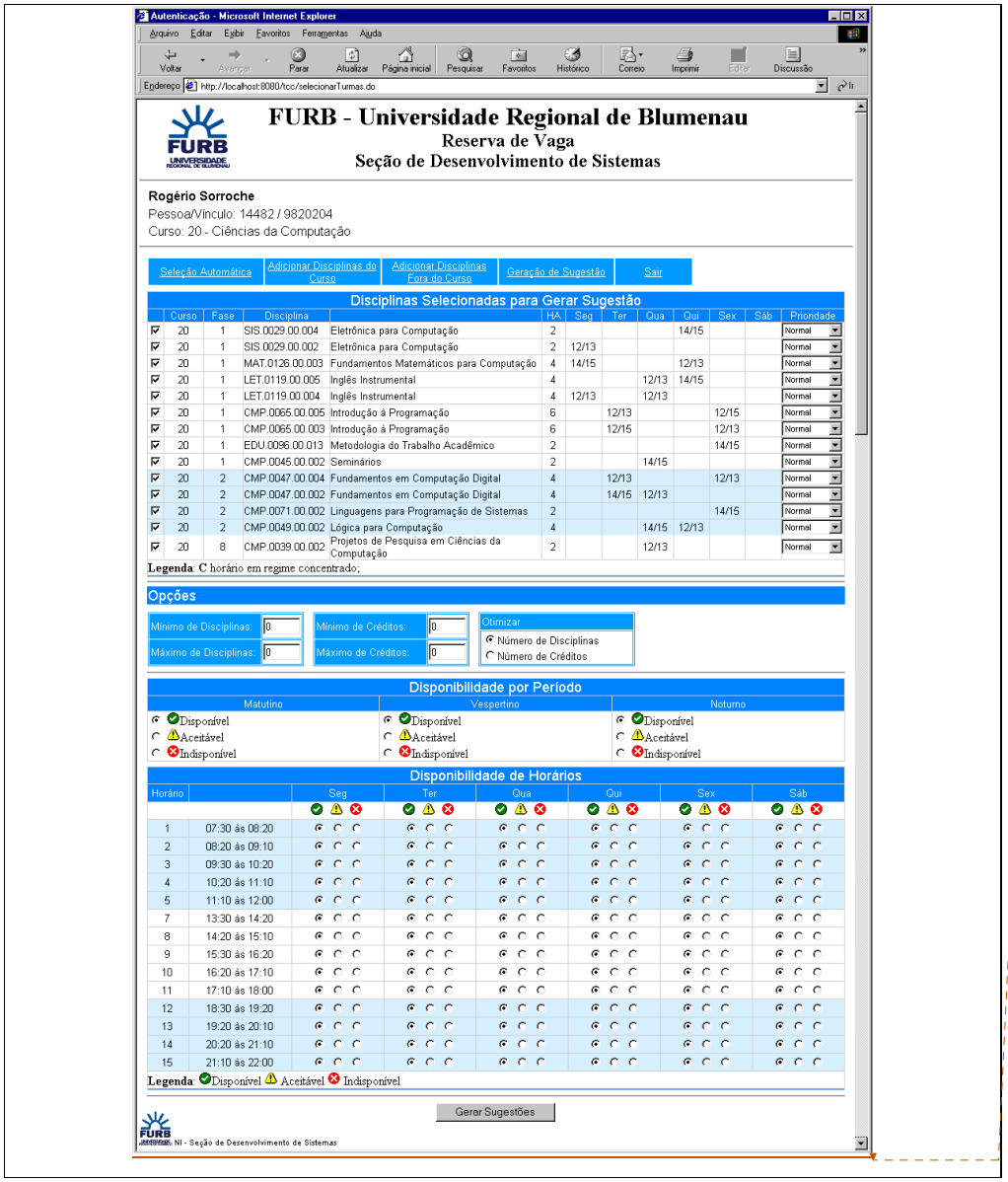
Após a autenticação é apresentada uma tela para o aluno com as opções de seleção das disciplinas para gerar sugestões. A Figura 49 apresenta esta tela.

Excluído: ------Quebra de página------

Figura 49 – Tela de opções de seleção das disciplinas

Nesta tela o aluno pode optar por descartar as disciplinas em que já obteve aprovação, as disciplinas matriculadas no semestre atual ou as disciplinas em que faltam cursar os pré-requisitos. O aluno também pode optar por deixar o sistema procurar em outros cursos uma disciplina que é ofertada ao seu curso mas em outros horários. Após selecionadas as disciplinas é apresentada a tela da Figura 50.

Figura 50 – Tela resumo disciplinas selecionadas e opções

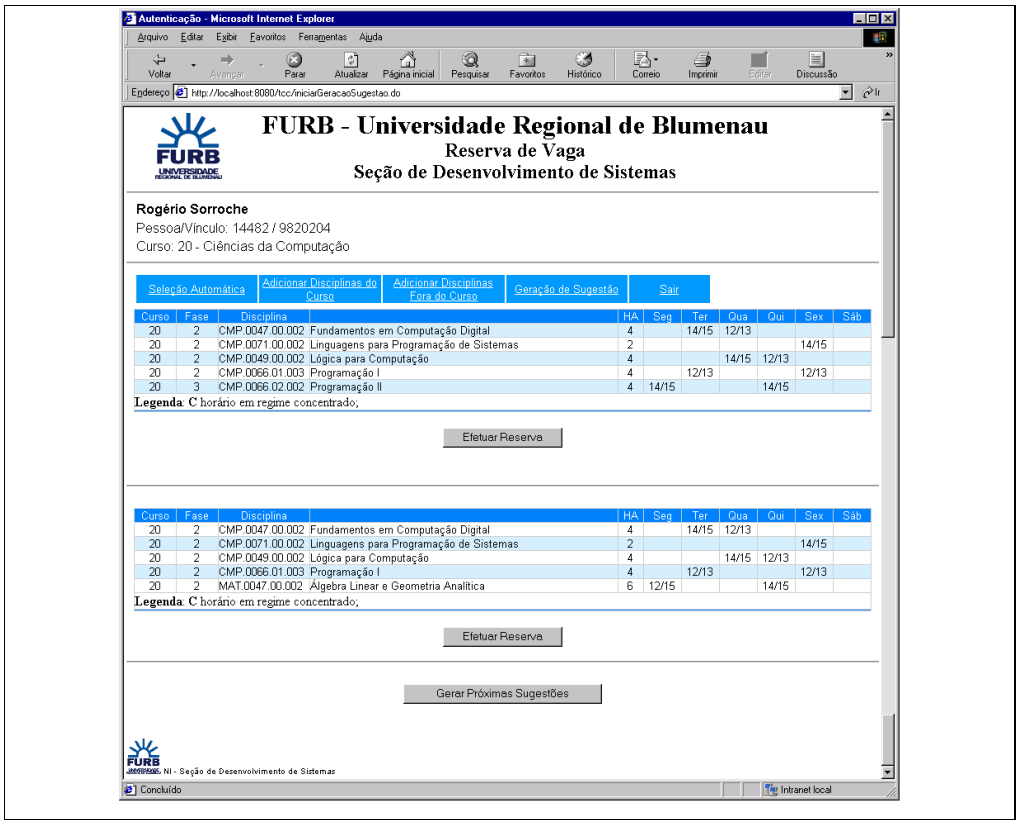


Nesta tela é apresentado para o aluno o conjunto de disciplinas que foram selecionadas, permitindo a ele definir uma prioridade para cada disciplina. As prioridades que o aluno pode escolher são em ordem crescente: “Aceitável”, “Normal”, “Desejável” e “Obrigatória”.

Também, é oferecido ao aluno a opção de restringir os números mínimo e máximo de disciplinas ou créditos, nas sugestões geradas. O aluno pode optar por gerar sugestões que maximizem o número de disciplinas ou o número de créditos. Nos itens “Disponibilidade por Período” e “Disponibilidade de Horários”, o aluno pode informar a sua disponibilidade em determinados horários ou períodos.

Quando o aluno pressionar o botão “Gerar Sugestões”, o sistema irá tentar gerar até dez sugestões e apresentá-las na tela da Figura 51.

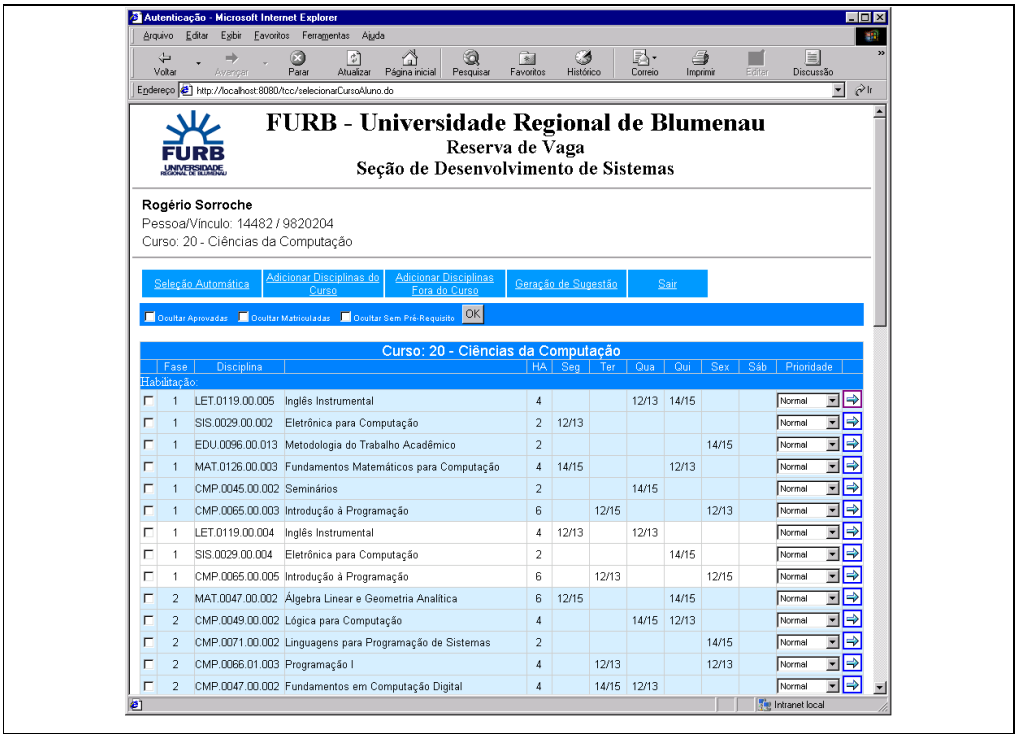
Figura 51 – Sugestões geradas



Esta tela apresenta então, as sugestões geradas onde o aluno pode efetuar a reserva de acordo com a sugestão ou pode tentar gerar mais sugestões se for possível, através do botão “Gerar Próximas Sugestões”.

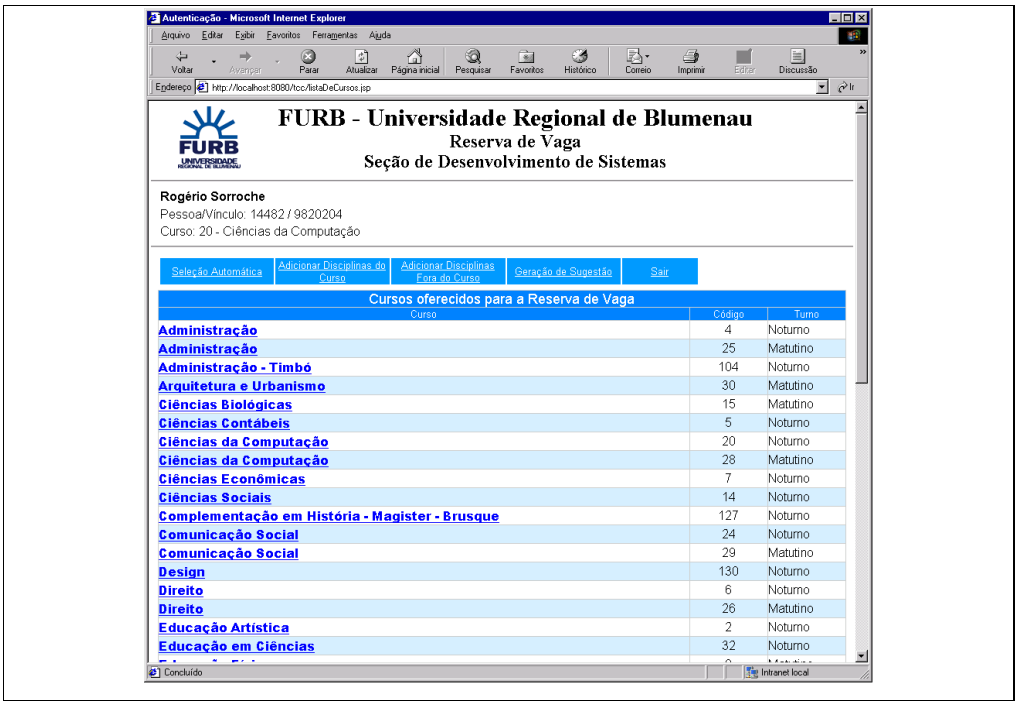
Na tela da Figura 50, através dos *links* “Adicionar Disciplinas do Curso” e “Adicionar Disciplinas Fora do Curso”, o aluno pode optar por adicionar mais disciplinas tanto do seu curso como de fora do curso. Se o aluno optar por adicionar disciplinas do curso será apresentada a tela da Figura 52, onde ele poderá escolher as disciplinas do seu curso. Nesta tela o aluno tem também a opção de ocultar as disciplinas já aprovadas, as matriculadas ou as disciplinas sem aprovação nos pré-requisitos.

Figura 52 - Tela de seleção de disciplinas por curso



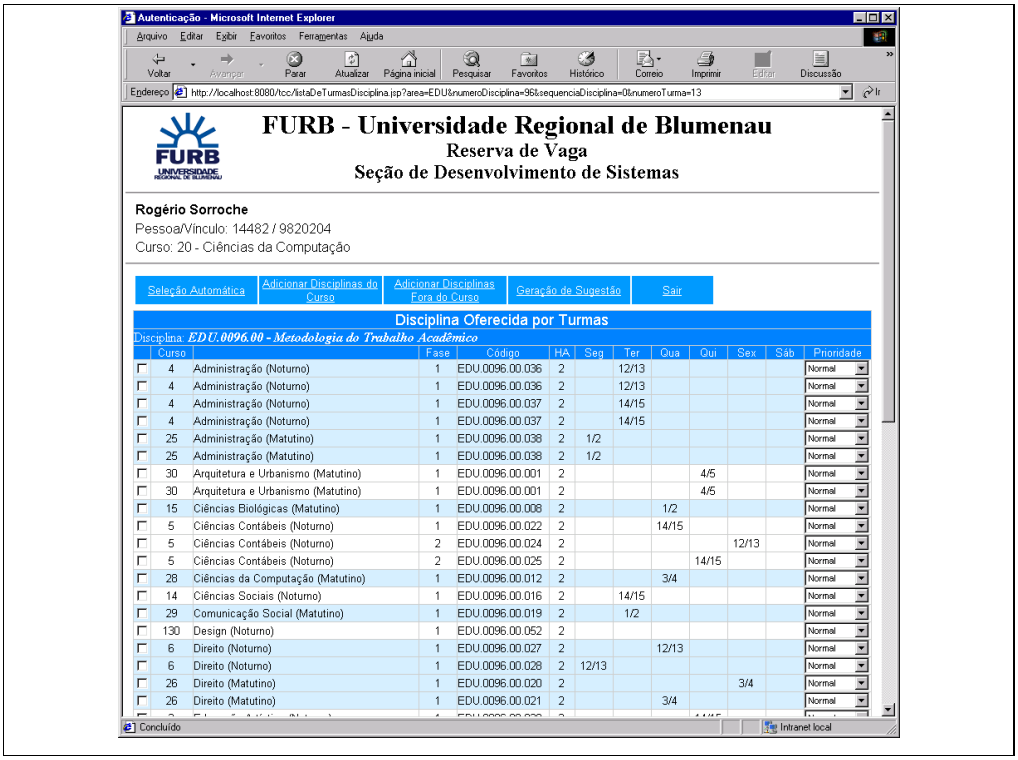
Ao selecionar o *link* “Adicionar Disciplinas Fora do Curso”, é apresentada a tela mostrada na Figura 53 para o aluno, onde ele pode optar pelo curso em que deseja selecionar disciplinas. Ao selecionar o curso desejado, a tela da Figura 52 é apresentada novamente, agora com as disciplinas do curso selecionado.

Figura 53 – Tela de cursos disponíveis



Na tela apresentada na Figura 52, no canto direito ao lado de cada disciplina há um *link*, onde o aluno pode optar por procurar a disciplina em outros cursos ou horários, ou seja, procurar todas as turmas que foram criadas a partir de um disciplina. É apresentada então a tela mostrada na Figura 54.

Figura 54 – Tela disciplina em outros cursos



Curso		Fase	Código	HA	Seg	Ter	Qua	Qui	Sex	Sáb	Prioridade
4	Administração (Noturno)	1	EDU.0096.00.036	2		12/13					Normal
4	Administração (Noturno)	1	EDU.0096.00.036	2		12/13					Normal
4	Administração (Noturno)	1	EDU.0096.00.037	2		14/15					Normal
4	Administração (Noturno)	1	EDU.0096.00.037	2		14/15					Normal
25	Administração (Matutino)	1	EDU.0096.00.038	2	1/2						Normal
25	Administração (Matutino)	1	EDU.0096.00.038	2	1/2						Normal
30	Arquitetura e Urbanismo (Matutino)	1	EDU.0096.00.001	2				4/5			Normal
30	Arquitetura e Urbanismo (Matutino)	1	EDU.0096.00.001	2				4/5			Normal
15	Ciências Biológicas (Matutino)	1	EDU.0096.00.008	2			1/2				Normal
5	Ciências Contábeis (Noturno)	1	EDU.0096.00.022	2			14/15				Normal
5	Ciências Contábeis (Noturno)	2	EDU.0096.00.024	2					12/13		Normal
5	Ciências Contábeis (Noturno)	2	EDU.0096.00.025	2				14/15			Normal
28	Ciências da Computação (Matutino)	1	EDU.0096.00.012	2			3/4				Normal
14	Ciências Sociais (Noturno)	1	EDU.0096.00.016	2		14/15					Normal
29	Comunicação Social (Matutino)	1	EDU.0096.00.019	2		1/2					Normal
130	Design (Noturno)	1	EDU.0096.00.052	2							Normal
6	Direito (Noturno)	1	EDU.0096.00.027	2			12/13				Normal
6	Direito (Noturno)	1	EDU.0096.00.028	2	12/13						Normal
26	Direito (Matutino)	1	EDU.0096.00.020	2					3/4		Normal
26	Direito (Matutino)	1	EDU.0096.00.021	2			3/4				Normal

Esta tela mostra todas as turmas que estão sendo oferecidas para uma determinada disciplina. Assim o aluno pode optar por cursar uma disciplina que faz parte da sua grade curricular em outro curso ou horário. Após selecionadas as disciplinas é apresentada novamente a tela da Figura 50.

5.4 RESULTADOS E DISCUSSÃO

Para avaliar os algoritmos de geração das sugestões, foram realizados testes utilizando um grafo composto pelo conjunto de disciplinas oferecidas para um determinado curso. Inicialmente, construiu-se o grafo com um total de 52 vértices e foram geradas todas as sugestões possíveis para este grafo. Após o término do processo foram eliminados cinco vértices aleatoriamente do grafo, iniciando-se o processo de geração das sugestões novamente, e assim sucessivamente até não haverem mais vértices no grafo.

Os testes foram realizados em um microcomputador PC/AT equipado com um microprocessador *AMD Athlon* de 1,3 Ghz e 512 Mb de memória RAM. O sistema operacional utilizado foi o *Microsoft Windows NT Workstation* versão 4.0. A máquina virtual *Java* utilizada para a execução dos testes foi a versão 1.3 fornecida pela *Sun Microsystems*. A execução dos testes foi realizada localmente. Neste ambiente, somente os serviços necessários ao sistema operacional estavam sendo executados e a máquina utilizada não estava sendo acessada por nenhum cliente.

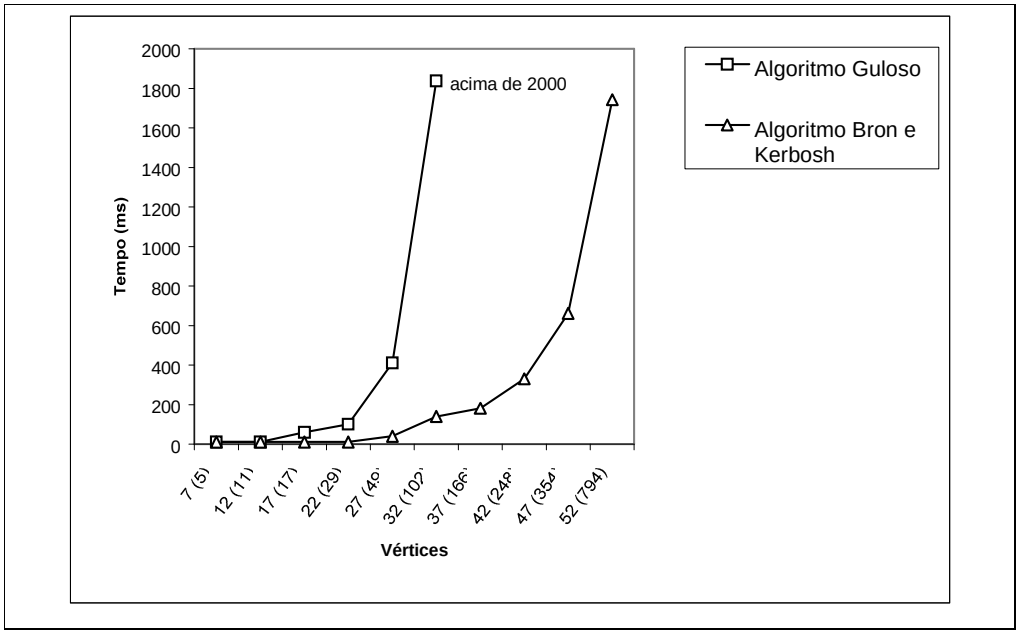
Para cada teste realizado, foram cronometrados o tempo das primeiras dez sugestões e o tempo total para gerar todas as sugestões. A Tabela 2 apresenta os resultados obtidos.

Tabela 2 – Teste comparativo Algoritmo Guloso X Algoritmo Bron e Kerbosh

		Algoritmo Guloso			**Algoritmo de Bron e Kerbosh**		
		Tempo (ms)			Tempo (ms)		
Vértices	Sugestões Geradas	Primeira Sugestão	Total	Média	Primeira Sugestão	Total	Média
7	5	1	1	2	1	1	0,20
12	11	1	10	0,91	1	1	0,20
17	17	1	60	3,53	1	10	0,59
22	29	10	100	3,49	1	10	0,34
27	49	10	411	8,39	1	40	0,82
32	102	10	1.837	18,01	1	140	1,37
37	166	10	5.388	32,46	1	181	1,09
42	248	10	17.115	61,02	1	330	1,33
47	354	60	40.668	114,88	10	661	1,87
52	794	70	206.977	260,68	60	1.742	2,19

O Quadro 3 apresenta graficamente o teste comparativo entre os dois algoritmos.

Quadro 3 – Gráfico comparativo Algoritmo Guloso X Algoritmo Bron e Kerbosh



Conforme se pode observar, o algoritmo de Bron e Kerbosh apresenta um tempo menor de resposta, especialmente para grafos maiores. No algoritmo Guloso constatou-se que a verificação feita após a geração de uma sugestão, para determinar se o conjunto formado é maximal, consome um tempo razoável.

Deve-se ressaltar, no entanto, que ambos os algoritmos se mostram aceitáveis para a geração das sugestões de matrícula, uma vez que geralmente os alunos irão selecionar poucas disciplinas e gerar poucas sugestões. Cabe-se dizer também que os dois algoritmos, obtiveram um tempo médio por sugestão de menos de um segundo, o que considerando se tratar de uma aplicação *web* interativa, é um tempo insignificante.

Com o desenvolvimento deste trabalho foi possível observar as vantagens e desvantagens da utilização da plataforma *J2EE* para o desenvolvimento de sistemas corporativos. Esta plataforma fornece a infra-estrutura e os serviços para habilitar o desenvolvimento de sistemas seguros, escaláveis e interoperáveis para aplicações empresariais. Dentre as vantagens desta plataforma se pode alistar:

a) portabilidade: por ser baseada na linguagem *Java* apresenta independência de sistemas operacionais e máquinas;
b) escalabilidade: devido à sua portabilidade e por ser centrada nos servidores é possível escalar os sistemas para atender demandas maiores em computadores de grande porte;
c) independência de fornecedor: a plataforma *J2EE* é baseada numa família de especificações que pode ser implementada por qualquer fornecedor;
d) baseada em componentes: cria um padrão para componentes de aplicações que podem ser distribuídos e reutilizados;
e) integração com sistemas legados: a especificação provê padrões para a integração com sistemas legados;
f) provê um *framework*: o desenvolvedor se preocupa somente com a lógica de negócios, sendo que a plataforma provê os serviços necessários como: transações, segurança e persistência.

Enfatiza-se ainda que a tecnologia de *EJB* permite facilmente a criação de uma camada de objetos de negócio que podem ser acessados por vários tipos de clientes, de pontos remotos. Devido a isso é possível criar um conjunto de serviços que podem ser oferecidos para os clientes e podem ser reutilizados muitas vezes. Um dos pontos fracos desta tecnologia é que devido aos *EJB* não serem modelados como objetos de um domínio de aplicação, mas como componentes, os conceitos de orientação a objetos como herança e polimorfismo não são possíveis. Também, a utilização de uma chave primária para a identificação de *entity beans* não faz parte dos conceitos de orientação a objetos. Neste trabalho não foi possível a utilização dos *entity beans* para modelar os dados do sistema acadêmico, pois para a leitura de uma grande quantidade de registros, como é o caso do sistema de reserva de vaga, eles apresentam uma séria perda de performance.

As tecnologias de componentes *web*, com a utilização de alguns padrões de projeto, podem se tornar grandemente reutilizáveis e possibilitar a criação de interfaces ricas para os usuários. Uma aplicação construída utilizando estes componentes pode ser acessada apenas com a utilização de um navegador *web*, sem a necessidade da instalação de qualquer software no cliente.

6 CONCLUSÕES

O objetivo principal deste trabalho, que era o de implementar um protótipo para auxiliar no processo de matrícula dos alunos, foi totalmente cumprido.

Com o uso do sistema proposto, os alunos que não estejam regulares com a grade curricular do seu curso, tem um valioso e rápido auxílio para se matricularem em um número maior de disciplinas, atingindo assim o mínimo de créditos financeiros que terão que pagar, dando um melhor aproveitamento à sua grade de horários e reduzindo o seu tempo de curso. Além disso, o sistema beneficia também a universidade, que terá um maior retorno financeiro.

Foram implementados dois algoritmos de elaboração das sugestões, o algoritmo Guloso e o algoritmo proposto por Bron e Kerbosh. O algoritmo de Bron e Kerbosh apresentou melhor performance em testes realizados, apesar de que ambos se mostraram adequados para a geração das sugestões.

A ferramenta CASE *Together* se mostrou muito prática uma vez que suporta uma metodologia de desenvolvimento ágil utilizando padrões de projeto e a modelagem de EJB.

Quanto à plataforma de desenvolvimento *J2EE*, mostrou-se apropriada para o desenvolvimento de aplicações empresariais. Esta plataforma apresenta uma série de vantagens para o desenvolvimento de grandes aplicações que necessitam de escalabilidade, disponibilidade e performance. Também destaca-se a portabilidade, uma vez que o protótipo foi desenvolvido numa plataforma *Windows NT* e posto em produção numa plataforma *Linux* sem que qualquer alteração fosse feita.

A tecnologia *Enterprise JavaBeans* demonstrou ser adequada para o desenvolvimento de sistemas multicamadas, pois possibilita que os serviços desenvolvidos sejam disponibilizados a vários tipos de clientes. Esta tecnologia permite criar uma interface padrão de componentes de negócio, que são totalmente reutilizáveis. Estes componentes permitem a integração de sistemas *J2EE* com sistemas legados e bancos de dados relacionais.

Os *Servlets* e as páginas *JSP* se mostraram uma boa opção de desenvolvimento de interface, pois o protótipo pode ser acessado através da *internet* dispensando exigência de instalação de *plug-ins* e *drivers*, sendo necessário tão somente ao usuário, um navegador *web*.

Os *servlets* se mostraram eficazes para o processamento de lógica de controle e recuperação de conteúdo na camada *web*, conforme especificado pelo padrão *front controller*, enquanto que, as páginas *JSP* são apropriadas para a construção de telas para apresentação dos dados.

Sobre os padrões de projeto verificou-se que ajudam a reduzir a complexidade e tempo para o desenvolvimento das aplicações e promovem a reutilização dos esforços de desenvolvimento. Para uma tecnologia relativamente nova como a J2EE, a utilização de padrões aumenta a confiabilidade do sistema uma vez que estes padrões já foram testados.

Destaca-se finalmente, que a viabilização deste Trabalho de Conclusão de Curso foi possível graças ao apoio integral recebido do Núcleo de Informática da FURB, desde a oferta de treinamento e consultoria nas tecnologias envolvidas até a disponibilização de informações para o teste do sistema.

6.1 EXTENSÕES

Como extensão deste trabalho propõe-se estudar outros métodos de geração de soluções como, por exemplo, *Algoritmos Genéticos*, *Simulated Anneling*, *Tabu Search* ou qualquer dos vários métodos existentes para problemas de *timetabling*, para fazer uma comparação de tempo de execução dos algoritmos e da qualidade das sugestões geradas.

Com respeito à tecnologia *J2EE*, sugere-se a pesquisa dos novos padrões, como por exemplo *JDO* (*Java Data Objects*), que implementa um mecanismo de persistência para camadas de objetos. Sugere-se ainda a elaboração de uma metodologia de desenvolvimento de sistemas utilizando estas novas tecnologias, bem como, o desenvolvimento de um sistema utilizando mais efetivamente *entity bean* e *message-driven beans*.

REFERÊNCIAS BIBLIOGRÁFICAS

ADAM, Emerson de Pinho. **Protótipo de uma aplicação para consultas acadêmicas utilizando servlets**. 2001. 79 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

ALUR, Deepak; CRUPI, John; MALKS, Dan. **Core j2ee patterns**: as melhores práticas e estratégias de design. Tradução Altair Dias Caldas de Moraes, Cláudio Belleza Dias, Guilherme Dias Caldas Moraes. Rio de Janeiro: Campus, 2002.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML**: **guia do usuário**. Rio de Janeiro: Campus, 2000.

BRAZ, Osmar de Oliveira Jr. **Otimização de horários em instituições de ensino superior através de algoritmos genéticos**. 2000. 144 f. Trabalho de Conclusão de Curso (Mestrado em Engenharia de Produção) – Programa de Pós-Graduação em Engenharia de Produção, Universidade Federal de Santa Catarina, Florianópolis.

BRON, Coen; KERBOSCH, Joep. Finding all cliques of an undirected graph. **The Communications of the ACM**, Eindhoven, v. 16, n. 9, p. 575-577, set. 1973.

BURNS, Ed; HUSTED, Ted; MCCLANAHAN, Craig R. **Struts user guide**, [2002?]. Disponível em: < http://jakarta.apache.org/struts/userGuide/index.html>. Acesso em: 15 nov. 2002.

FURTADO, Antônio Luz. **Teoria dos grafos**: algoritmos. Rio de Janeiro: Editora da Universidade de São Paulo, 1973.

GAGNON, Michel. **Algoritmos e teoria dos grafos**. [S.l.], 2001. Disponível em: < http://www.inf.ufpr.br/~michel/Disciplinas/Bac/Grafos/index_grafos.html>. Acesso em: 17 nov. 2002.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Padrões de projeto**: soluções reutilizáveis de software orientado a objetos. Tradução Luiz A. Meirelles Salgado. Porto Alegre: Bookman, 2000.

GOLDBARG, Marco Cesar, LUNA, Henrique Pacca L. **Otimização combinatória e programação linear**: modelos e algoritmos. Rio de Janeiro: Campus, 2000.

GOODWILL, James. **Developing java servlets**. Indianápolis: Sams Publishing, 2001.

GROSS, Jonathan; YELLEN, Jay. **Graph theory and its applications**. Nova Iorque: CRC Press, 1998.

HAEFEL, Richard M. **Enterprise javabeans**. 3. ed. Beijing: O'Reilly, 2001.

HALL, Marty. **Core servlets and javaserver pages**. New Jersey: Pretice Hall, 2000.

HERTZ, A. Finding a feasible course schedule using tabu search. **Discrete Applied Mathematics**. v. 35, p. 255-270, 1992.

HUNTER, Jason; CRAWFORD, William. **Java servlet programming**. Cambridge: O'Reilly, 2001.

JUBIN, Henri; FRIEDRICHS, Jürgen; TEAM, Jalapeño. **Enterprise javabeans by example**. New Jersey: Prentice Hall PTR, 1999.

LUCAS, Diogo Correa. **Algoritmos genéticos**: um estudo de seus conceitos fundamentais e aplicação no problema de grade horária. 2000. 65 f. Trabalho de Conclusão de Curso (Bacharelado em Informática) – Instituto de Física e Matemática – Universidade Federal de Pelotas, Pelotas.

MARINESCU, Floyd. **EJB design patterns**: advanced patterns, processes, and idioms. New York: John Wiley & Sons, 2002.

RABUSKE, Márcia Aguiar. **Introdução à teoria dos grafos**. Florianópolis: Ed. Da UFSC, 1992.

ROMAN, Ed; AMBLER, Scott; JEWELL, Tyler. **Mastering enterprise javabeans**. New York: John Wiley & Sons, 2002.

SCHNEIDE, Ricardo Luiz. **Design patterns**. Rio de Janeiro, maio 1999. Disponível em: < http://www.dcc.ufrj.br/~schneide/PSI_981/gp_6/design_patterns.html>. Rio de Janeiro, nov. 1999. Acesso em: 03 out 2002.

SCHWARZ, Gaston Adair; BARCIA, Ricardo Miranda. **Geração de horário em instituições de ensino com otimização simultânea de tempo e espaço**. 1990. 187 f. Dissertação (Mestrado em Engenharia de Produção) – Universidade Federal de Santa Catarina.

SESHADRI, Govind. **Enterprise java computing**: applications and architeture. Cambridge: Sign Books, 1999.

SUN, Sun Microsystems. **The j2ee tutorial**. [S.l.], 2002a. Disponível em: < http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Overview.html>. Acesso em: 20 ago. 2002.

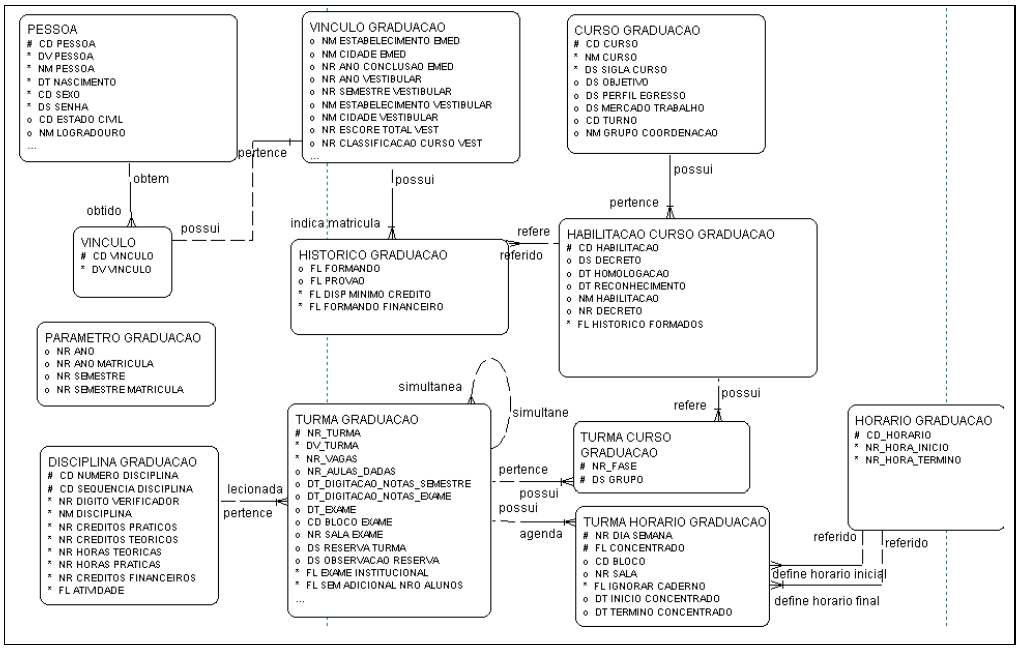
SUN, Sun Microsystems. **Designing enterprise applications with the j2ee platform, enterprise edition**. [S.l.], 2002b. Disponível em: < http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/index.html>. Acesso em: 26 ago. 2002.

SZWARCFITER, Jayme Luiz. **Grafos e algoritmos computacionais**. Rio de Janeiro: Campus, 1984.

THOMAS, Anne. Enterprise javabeans technology: server component model for the java platform. [S.l.], 1998. Disponível em: <http://www.ejbean.com/documents/downloads/white_paper.pdf >. Acesso em:01 out. 2002.

THOMPSON, J.; DOWSLAND, K. A., General cooling schedules for a Simulated Anneling based timetabling system. in: International Conference on the The Practice and Theory of Automated Timetabling, p. 345-363, 1996.

ANEXO 1 – MODELO DE DADOS DO SISTEMA ACADÊMICO



ANEXO 2 – ALGORITMO GULOSO

```
package busca;

import java.util.*;

public class AlgoritmoGuloso extends AlgoritmoBuscaSugestao {

    public AlgoritmoGuloso(BuscaSugestao busca) {
        super(busca);
    }

    // procedimento que controla a busca até encontrar uma sugestão
    public NodoBuscaSugestao gerarSugestao() {

        if ( !inicializado ) {
            inicializar();
            inicializado = true;
        }

        NodoBuscaSugestao nodo = null;
        while ( pilha.size() > 0 && nodo == null ) {
            try {
               nodo = fazUmaIteracao();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        return nodo;
    }

    public NodoBuscaSugestao fazUmaIteracao() throws Exception {

        // visita o próximo nodo da pilha
        NodoBuscaSugestao nodo = (NodoBuscaSugestao)pilha.remove(0);

        // se o nodo está nulo gera excecao
        if ( nodo == null ) {
            throw new Exception("Nodo de busca vazio");
        }

        // atribui como nodo  corrente
        setNodoCorrente(nodo);

        // verifica se é o melhor nodo
        if ( nodo.getCusto() > getMelhorNodo().getCusto() ) {
            setMelhorNodo(nodo);
        }

        // verifica se é uma condição de parada
        if ((nodo.getItensParaAlocar().size() == 0) && (nodo.getItensAlocados().size() != 0))
        {

             // se é uma solução e é uma nova solução
            if ( getBusca().isSolucao(nodo) && !getBusca().contemSolucao(nodo) ) {
                return nodo;
            }

        }

        // gera os sucessores deste nodo
        Collection sucessores = expandeNodo(nodo);

        // para cada operador gerado poe os nós na pilha
        // para cada sucessor adiciona na pilha
        for ( Iterator i = sucessores.iterator(); i.hasNext();  ) {

            // obtem nodo sucessor
```

```
            NodoBuscaSugestao sucessor = (NodoBuscaSugestao)i.next();

            // adiciona na pilha
            inserePilha(sucessor);

        }

        // se chegou aqui é porque não encontrou solução
        return null;

    }

    // procedimento que pega um nodo e gera todos os seus sucessores
    public Collection expandeNodo(NodoBuscaSugestao nodo) {

        // cria cópia das listas do nodo para não alterá-los
        Collection itensAlocar = (Collection)((ArrayList)nodo.getItensParaAlocar()).clone();

        // lista de sucessores que será retornada
        Collection sucessores = new ArrayList();

        for (Iterator i = itensAlocar.iterator(); i.hasNext(); ) {

            // pega próximo item de sugestao
            ItemBusca itm = (ItemBusca)i.next();

            // remove este item para que não seja mais pesquisado
            i.remove();

            // clona listas de itens atuais
            List itnsAlocarSucessor   = (List)((ArrayList)itensAlocar).clone();
            List itnsAlocadosSucessor = (List)((ArrayList)nodo.getItensAlocados()).clone();

            // elimina da lista de itens para alocar as que possuem coincidências
            eliminarCoincidentes(nodo, itm, itnsAlocarSucessor, true);

            // adiciona item na lista de itens alocados
            itnsAlocadosSucessor.add(itm);

            // cria nodo de busca
            sucessores.add( new NodoBuscaSugestao(getBusca(), itnsAlocarSucessor
                                , itnsAlocadosSucessor, nodo.getProfundidade()+1) );

        }

        return sucessores;
    }

    // procedimento de inicialização do algoritmo
    private void inicializar() {

        // cria pilha
        pilha  = new ArrayList();

        // cria nodo inicial
        NodoBuscaSugestao nodo = new NodoBuscaSugestao(getBusca()
                                                    , getBusca().getItensPreprocessados()
                                                    , new ArrayList(), 0);

        // atribui como nodo corrente e melhor
        setNodoCorrente(nodo);
        setMelhorNodo(nodo);

        // coloca nodo inicial na pilha
        pilha.add(nodo);

    }


    // insere nodo na pilha de acordo com a profundidade e custo
    private void inserePilha(NodoBuscaSugestao nodo) {

        boolean inserido = false;
```

```
        for (int i = 0; i < pilha.size() && !inserido ; i++) {

            NodoBuscaSugestao n = (NodoBuscaSugestao)pilha.get(i);

            if ( n.getProfundidade() < nodo.getProfundidade() ) {

                  pilha.add(i, nodo);
                  inserido = true;

             } else {

                   if ( nodo.getCusto() < n.getCusto() ) {
                        pilha.add(i, nodo);
                        inserido = true;
                   }

             }

        }

        if ( !inserido ) pilha.add(nodo);

    }

    private boolean inicializado = false;

    private ArrayList pilha;
    private boolean parar = true;
}
```

ANEXO 3 – ALGORITMO DE BRON E KERBOSH

```
package busca;

import java.util.*;

public class AlgoritmoBronKerbosh extends AlgoritmoBuscaSugestao {

    public AlgoritmoBronKerbosh(BuscaSugestao busca) {
        super(busca);
    }

    // procedimento que controla a busca até encontrar uma sugestão
    public NodoBuscaSugestao gerarSugestao() {

        if ( !inicializado ) {
            inicializar();
            inicializado = true;
        }

        NodoBuscaSugestao nodo = null;
        while ( pilha.size() > 0 && nodo == null ) {
            try {
               nodo = fazUmaIteracao();
            } catch (Exception e) {
            }
        }

        return nodo;
    }

    public NodoBuscaSugestao fazUmaIteracao() throws Exception {

        // visita o próximo nodo da pilha
        NodoBuscaBronKerbosh nodo = (NodoBuscaBronKerbosh)pilha.get(0);

        // atribui como nodo corrente
        setNodoCorrente(nodo);

        // verifica se é o melhor nodo
        if ( nodo.getCusto() > getMelhorNodo().getCusto() ) {
            setMelhorNodo(nodo);
        }

        boolean backtracking      = false; // indicador de backtracking
        boolean solucaoEncontrada = false; // indicador de solução encontrada

        // CAND vazio e ANT vazio, condicao de parada
        if ((nodo.getItensParaAlocar().size()==0) && (nodo.getItensVisitadosAnt().size()==0))
        {

            if ( getBusca().isSolucao(nodo) ) {
               solucaoEncontrada = true;
            }
            backtracking = true;

        // CAND vazio e ANT não vazio, impossível prosseguir
        } else if ( ( nodo.getItensParaAlocar().size() == 0 ) &&
                    ( nodo.getItensVisitadosAnt().size() > 0) ) {

            backtracking = true;

        // verifica se é viável prosseguir (bound), se não faz backtracking
        } else if ( condicaoLimite(nodo.getItensParaAlocar(),nodo.getItensVisitadosAnt() ) ) {

            backtracking = true;

        }
```

```
        // se for para fazer backtracking
        if ( backtracking ) {

            // desempilha
            pilha.remove(0);

        } else { // é para continuar estendendo nodo atual

            // estende o nodo atual gerando um novo nodo
            NodoBuscaSugestao novoNodo = estendeNodo(nodo);

            // empilha
            pilha.add(0, novoNodo);

        }

        // solução encontrada, retorna solução
        if ( solucaoEncontrada ) {
            return nodo;
        }

        // chegou aqui porque não encontrou solução
        return null;

    }

    public NodoBuscaBronKerbosh estendeNodo(NodoBuscaBronKerbosh nodo) {

        // pega proximo item CANDIDATO, removendo dos candidatos para não ser mais visitado
        ItemBusca itm = (ItemBusca)nodo.getItensParaAlocar().remove(0);

        // duplica as listas CAND e ANT
        List itnsAlocarSucessor  = (List)((ArrayList)nodo.getItensParaAlocar()).clone();
        List itnsVistAntSucessor = (List)((ArrayList)nodo.getItensVisitadosAnt()).clone();

        // colocar em ANT
        nodo.getItensVisitadosAnt().add(itm);

        // duplica a lista de vértices independentes VI, para facilitar backtracking
        List itnsAlocadosSucessor = (List)((ArrayList)nodo.getItensAlocados()).clone();

        // coloca o item como alocado
        itnsAlocadosSucessor.add(itm);

        // elimina as coincidências
        eliminarCoincidentes(nodo, itm, itnsAlocarSucessor , true);  // CAND
        eliminarCoincidentes(nodo, itm, itnsVistAntSucessor, false); // ANT

        // cria novo nodo de busca
        return new NodoBuscaBronKerbosh(getBusca(), itnsAlocarSucessor
                                        , itnsAlocadosSucessor, itnsVistAntSucessor
                                        , nodo.getProfundidade()+1);

    }

    // procedimento de inicialização
    private void inicializar() {

        // cria pilhas
        pilha = new ArrayList();

        // cria nodo inicial
        NodoBuscaBronKerbosh nodo = new NodoBuscaBronKerbosh(getBusca()
                                            , getBusca().getItensPreprocessados()
                                            , new ArrayList(), new ArrayList(), 0);

        // atribui como nodo corrente e melhor
        setNodoCorrente(nodo);
        setMelhorNodo(nodo);

        // coloca na pilha
```

```
        pilha.add(0, nodo);

    }

    // verifica condição bound ou limite
    private boolean condicaoLimite( List itParaAlocar, List itVistadosAnt ) {

        // se algum vértice de ANT não for adjacente a nenhum em CAND então, limite atingido
        for ( Iterator i = itVistadosAnt.iterator(); i.hasNext(); ) {

           ItemBusca ant = (ItemBusca)i.next();

           boolean adjacente = false;
           for ( Iterator a = itParaAlocar.iterator(); ( a.hasNext() && !adjacente ); ) {
              ItemBusca cand = (ItemBusca)a.next();
              adjacente = ant.isCoincidente(cand);
           }

           // não coincidiu com ninguém retorna true
           if ( !adjacente ) {
              return true;
           }

        }

        // se chegou aqui, não é condição limite
        return false;
    }

    private boolean inicializado = false;
    private ArrayList pilha;
    private boolean parar = true;

}
```