Makhles Reuter Lange

Atualização Tecnológica do Formulário de Inscrição do Processo Seletivo da Pós-Graduação da UFSC

Florianópolis

29 de junho de 2017

Makhles Reuter Lange

Atualização Tecnológica do Formulário de Inscrição do Processo Seletivo da Pós-Graduação da UFSC

Relatório Parcial do Trabalho de Conclusão do Curso do Curso de Ciências da Computação da Universidade Federal de Santa Catarina.

Universidade Federal de Santa Catarina - UFSC

Ciência da Computação

Orientador: Andréia Alves dos Santos Schwaab

Coorientador: Leandro José Komosinski

Florianópolis

29 de junho de 2017

SUMÁRIO

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

O acesso aos cursos de Pós-Graduação da UFSC é feito através de processos seletivos. Atualmente, existe um sistema[1] com poucas funcionalidades que está sendo utilizado por secretarias de alguns cursos. Outras utilizam o seu próprio formulário de inscrição online, enquanto que algumas não utilizam nenhum sistema de inscrição propriamente dito, restringindo-se ao uso de formulários de inscrição manuais.

O uso desses diversos sistemas de inscrição, com suas funcionalidades e interfaces distintas, acarreta em diversos problemas, tais como:

- Falta de padronização dos requisitos e tecnologias utilizadas.
- Autenticação de forma não-centralizada, o que além de ser uma vulnerabilidade em termos de segurança, implica em duplicação de código.
- Difícil manutenibilidade;
- Difícil obtenção de estatísticas relacionadas aos candidatos.

Não obstante, o sistema atual está carente de algumas funcionalidades consideradas primordiais, descritas nos itens a seguir. A implementação de tais funcionalidades no sistema atual é custosa, pois sua estrutura contempla os requisitos definidos na época da elaboração do sistema. Adicionalmente, as bibliotecas e tecnologias utilizadas no sistema atual são antigas[2], dificultando seu desenvolvimento e sua manutenibilidade. Justifica-se, portanto, a atualização tecnológica do sistema de inscrição.

---

1 Vide http://www.capg.ufsc.br/inscricao
2 Algumas estão, inclusive, descontinuadas.

1.1.1 Envio de documentos

A funcionalidade mais importante que será implementada neste novo sistema é a possibilidade do envio de documentos por parte dos candidatos, que atualmente têm o trabalho de comparecer às secretarias dos cursos com as cópias dos documentos. A praticidade dessa funcionalidade beneficia tanto o candidato quanto a universidade.

1.1.2 Salvamento das informações

Caso um candidato queira fazer a inscrição em algum dos programas disponibilizados pela UFSC através do sistema de inscrição atual, este deve, essencialmente, fornecer todas as informações em uma única sessão e, ao final, receberá um número de inscrição e a possibilidade de imprimir um comprovante de inscrição. Pretende-se oferecer ao candidato a opção de salvar o progresso do preenchimento do formulário. O candidato poderá, dessa forma, distribuir o preenchimento do formulário em várias sessões até que esteja seguro das informações fornecidas e decida-se por finalizar sua inscrição.

1.1.3 Ordem de preenchimento do formulário

No sistema atual, o preenchimento do formulário segue uma ordem predefinida. Muitas vezes, no entanto, o candidato não possui todas as informações necessárias e/ou relevantes ao iniciar o preenchimento do formulário. Pretende-se dar a possibilidade de preenchimento do formulário sem nenhuma ordem imposta neste processo[3].

1.2 SETIC

A Superintendência de Governança Eletrônica e Tecnologia da Informação e Comunicação (SeTIC), setor de informática da Universidade Federal de Santa Catarina (UFSC), é responsável pelo planejamento, pes-

[3] A não ser, é claro, em situações onde exista uma ordem implícita. O envio de documentos é um exemplo, pois estes dependem do programa e nível escolhidos.

quisa, aplicação e desenvolvimento de produtos e serviços de tecnologia da informação e comunicação da universidade[4].

Ressalta-se que o autor deste trabalho já foi estagiário na SeTIC e atualmente possui bolsa de extensão pela FEESC. Todas as atividades estão sendo desenvolvidas nas instalações da SeTIC, com equipamentos e tecnologias disponibilizados por esta. Assim, o último estágio do desenvolvimento do presente sistema, *i.e.*, a *operação* e a *manutenção*[5], serão feitas pela SeTIC.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

O objetivo principal deste Trabalho de Conclusão de Curso é desenvolver um sistema para a inscrição de candidatos nos processos seletivos dos diversos programas de Pós-Graduação da UFSC, utilizando o sistema de inscrição que está em uso atualmente como base.

1.3.2 Objetivos Específicos

- Implementar o Módulo de Inscrição - consiste na implementação das regras de negócio e das páginas que o candidato terá acesso ao realizar sua inscrição.

- Implementar o Módulo Administrativo - consiste na implementação das regras de negócio no sistema CAPG Secretaria (sistema em desenvolvimento para administração dos cursos por suas secretarias).

- Elaborar um artigo referente ao TCC.

[4] Mais informações em http://setic.ufsc.br/apresentacao/

[5] O sistema é liberado e implantado no ambiente de produção. Eventuais erros que não foram descobertos nos estágios anteriores e novos requisitos podem surgir ao longo do uso do sistema, justificando a sua constante manutenção.

1.4 RESULTADOS ESPERADOS

- Módulo do Formulário de Inscrição.
- Módulo Administrativo do Processo de Inscrição.
- Artigo referente à produção realizada anexo ao TCC.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 APLICAÇÕES WEB

2.2 PLATAFORMA JAVA, EDIÇÃO EMPRESARIAL

As **aplicações empresariais** têm o propósito de fornecer a *lógica do negócio* de uma empresa. O seu gerenciamento é feito de forma centralizada e é comum a sua interação com outros softwares empresariais. A plataforma Java, Edição Empresarial (Java EE), é um conjunto de tecnologias Java que são utilizadas para o desenvolvimento de aplicações empresariais. O objetivo da plataforma é fornecer aos desenvolvedores um conjunto de especificações / APIs que permitam a diminuição do tempo de desenvolvimento, a redução da complexidade e o aumento do desempenho de aplicações empresariais (JENDROCK et al., 2014). Tais especificações são *contratos* que são implementados por diversos fornecedores, *e.g.*, GlassFish, Oracle WebLogic, Apache TomEE, etc.

2.2.1 Aplicações Multi-Camadas

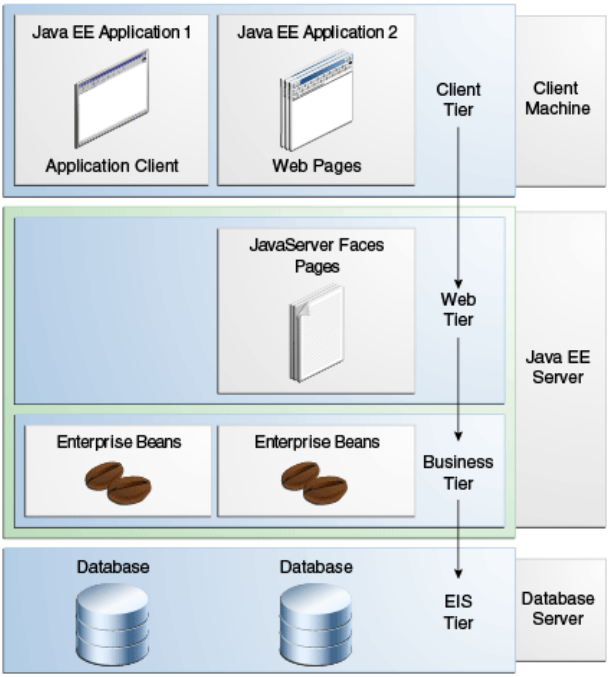
Segundo o modelo Java EE, as aplicações são distribuídas em camadas (*tiered design*)[1]. Define-se, desta forma, responsabilidades distintas para as diferentes partes do sistema, *i.e.*, para os diferentes componentes que compõem uma aplicação Java EE (vide Figura 1) e que são distribuídos nas camadas Cliente, Web, Negócio e EIS.

2.2.1.1 Camada Cliente

Esta camada é composta de clientes aplicativos que acessam o servidor Java EE e que geralmente se localizam em máquinas diferentes

[1] Os termos *tier* e *layer* são ambos traduzidos para o português como "camada". No entanto, uma *tier* representa uma unidade física, na qual um código ou processo é executado. Uma *layer*, por sua vez, representa uma unidade lógica, responsável pela organização lógica do código através da abstração dos dados. Diversas *layers* podem existir em computadores diferentes, ou em processos diferentes em um único computador, ou ainda em um único processo em um único computador (LHOTKA, 2005).

Figura 1 – Componentes Java EE distribuídos nas diversas camadas de duas aplicações Web.



Fonte: adaptado de JENDROCK, E. et al. *Java Platform, Enterprise Edition: The Java EE Tutorial*. 2014. [Acesso em 22/02/2017]. Disponível em: <https://docs.oracle.com/javaee/7/tutorial/>.

da máquina do servidor:

- Clientes Web - também chamados de *clientes magros*, são compostos de páginas dinâmicas (*e.g.*, XHTML) geradas na camada Web e por um navegador responsável por renderizá-las.

- Clientes Aplicativos - quando os usuários necessitam realizar atividades que requerem uma interface mais rica do que a disponibilizada por linguagens de marcação, utilizam-se clientes aplicativos

baseados nas APIs Swing ou AWT (*Abstract Window Toolkit*)[2]. Embora seja possível o estabelecimento de uma conexão HTTP com um *servlet* na camada Web, clientes aplicativos costumam acessar diretamente os *beans* gerenciais da camada de negócio.

- *Applets* - componentes embarcados que são incluídos nas páginas web. São escritos em Java e, portanto, necessitam da máquina virtual Java instalada no navegador web do cliente (e provavelmente do *plugin* Java e de um arquivo de política de segurança).

2.2.1.2 Camada Web

É a camada responsável pela interação entre os clientes e a camada de negócios. Suas principais funções são:

- Gerar, dinamicamente, conteúdo para o cliente em diversos formatos.
- Obter as entradas (dados e ações) da interface com o cliente e retornar os resultados dos componentes da camada de negócios.
- Controlar o fluxo das páginas no cliente.
- Manter o estado dos dados da sessão do usuário.
- Executar um pouco de lógica simples e armazenar dados em componentes JavaBeans temporariamente.

Servlets e páginas web criadas com a tecnologia *JavaServer Faces* (vide seção 2.2.2) são os componentes desta camada. *Servlets* são classes Java que possuem métodos adequados a receber requisições e construir respostas às tais requisições dinamicamente. O trecho de código a seguir (vide Algoritmo 1) representa um `HttpServlet` que atende a requisições HTTP do tipo GET através da sobrescrita do método `doGet()`. Os parâmetros `HttpServletRequest` e `HttpServletResponse` representam, respectivamente, a requisição feita pelo cliente e a resposta do servidor.

[2] Não obstante, pode-se utilizar uma interface em linha de comando.

Algoritmo 1 – Um *servlet* que imprime "*Hello World*".

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {
  public void doGet(HttpServletRequest req,
                    HttpServletResponse res)
      throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("<html>");
    out.println("<head><title>Hello World</title></head>");
    out.println("<body>");
    out.println("<big>Hello World</big>");
    out.println("</body></html>");
  }
}
```

Fonte: adaptado de Hunter e Crawford (1998)

Um *servlet* HTTP pode, além do método `doGet()`, sobreescrever outros métodos, tais como `doPut()`, `doDelete()`, etc, que representam requisições HTTP do tipo PUT, DELETE, etc, respectivamente.

2.2.1.3 Camada de Negócio

A camada de negócios possui componentes que provêm a lógica do negócio da aplicação, ou seja, código que provê funcionalidades para um determinado domínio do negócio. As tecnologias utilizadas nessa camada são as seguintes:

- Componentes Enterprise JavaBeans (EJB), que fornecem funcionalidades como gerenciamento de sessão, segurança, gerenciamento de transações, etc.
- *Web services* (JAX-RS, JAX-WS).
- Entidades de persistência - Java Persistence API (JPA).

2.2.1.4 Camada EIS

A camada EIS consiste de servidores de bancos de dados, sistemas de planejamento de recursos empresariais e outras fontes de dados legados. Esses recursos geralmente se localizam em suas próprias máquinas e são acessados pela camada de negócios. As tecnologias utilizadas nessa camada são:

- Java Database Connetivity API (JDBC) - uma interface que permite a conexão às bases de dados.
- Java Persistence API (JPA).
- Java Transaction API (JTA) - uma interface para a realização de transações.

2.2.2 JavaServer Faces

A tecnologia JavaServer Faces (JSF) é uma *especificação* de uma API Java utilizada para a criação de interfaces de usuário em aplicações web. Com essa API, é possível:

- Representar componentes[3] e o gerenciamento dos seus estados;
- Fazer o controle de eventos;
- Realizar a validação e a conversão de dados no lado do servidor;
- Definir regras para a navegação entre páginas;
- Prover o suporte à internacionalização e acessibilidade;

Existem duas implementações principais: Apache MyFaces e Oracle Mojarra. Ambas contêm pelo menos os componentes padrões, ou seja, os componentes responsáveis por gerar qualquer um dos elementos HTML básicos (tabelas, caixas de entrada de texto, botões, seletores, etc).

[3] Componentes, ou *widgets*, são entidades autônomas e reutilizáveis da interface de usuário.

2.2.2.1 Arquitetura do Framework

O *framework* JSF segue o padrão arquitetural[4] MVC baseado em componentes. A grande vantagem do padrão MVC é a separação entre apresentação e comportamento (lógica) da aplicação:

Visão - fornece a representação da informação para o usuário da aplicação. Pode-se ter diversas visões do mesmo modelo. A tecnologia padrão utilizada pelo JSF é a Facelets (vide Seção 2.2.3).

Modelo - representa o comportamento da aplicação em termos do domínio do problema, e independe da visão.

Controlador - converte entradas/eventos em comandos para a visão ou para o modelo.

Todas as requisições feitas à aplicação devem passar primeiramente pelo controlador através do `FacesServlet`, que é o *servlet* responsável por gerenciar o ciclo de vida do processamento de requisições. Ou seja, o próprio *framework* JSF assume o papel de controlador no padrão MVC (vide esquema A na Figura 2). Dessa forma, o desenvolvedor da aplicação não precisa se preocupar em escrever código *boilerplate* tal como o do Algoritmo 1. A implementação de controladores para atender a diferentes tipos de requisições, tais como o mostrado no Algoritmo 1, é utilizada em *action-based frameworks* (SpringMVC, Struts, ASP.NET, etc).

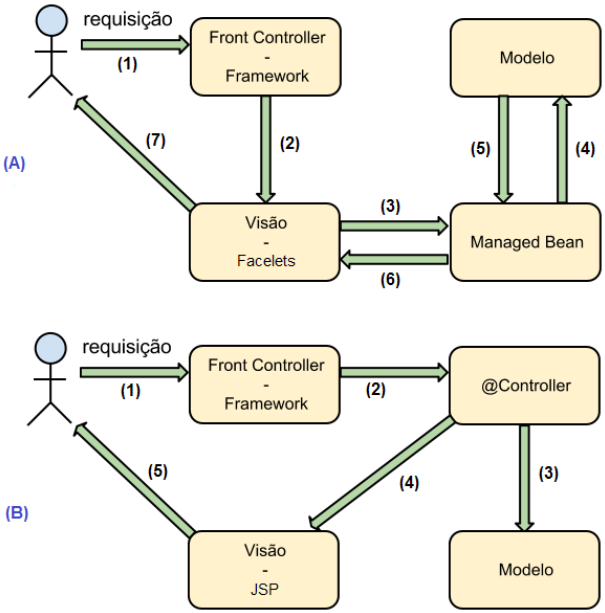
A Figura 2 mostra esquemas representativos destes dois modelos MVC. Em (A), tem-se o *framework* JSF, baseado em componentes. Resumidamente, a requisição feita pelo usuário da aplicação é recebida pelo `FacesServlet`, que delega o processamento para a visão responsável pela geração da página acessada. O componente acessado (por exemplo, um botão) invoca um método em um *managed bean*[5]. Este, por sua vez,

[4] Um padrão arquitetural expressa um esquema de organização estrutural para sistemas baseados em *software*. O padrão provê um conjunto de subsistemas predefinidos, especifica suas responsabilidades, e inclui regras e diretrizes para organizar a relação entre eles (BUSCHMANN et al., 1996).

[5] Uma classe Java que trabalha junto à visão.

pode buscar dados do modelo e retorná-los para a visão. Finalmente, a visão envia a resposta ao usuário. O ciclo de vida do processamento de requisições do JSF é bem mais complexo, pois envolve a criação de uma árvore de componentes, a conversão e validação dos dados obtidos destes componentes, o gerenciamento de eventos oriundos destes componentes e a propagação dos dados para os *beans*.

Figura 2 – Comparação entre os dois tipos de padrão MVC. Em (A), tem-se o *framework* baseado em componentes. Em (B), o baseado em ações. Os números ao lado das setas indicam a ordem do processamento de uma requisição.



Fonte: Caelum. Adaptado de ALMEIDA, A. *Entenda os MVCs e os frameworks Action e Component Based*. 2012. [Acesso em 25/04/2017]. Disponível em: <http://blog.caelum.com.br/entenda-os-mvcs-e-os-frameworks-action-e-component-based/>.

No esquema (B), o desenvolvedor da aplicação deve criar classes

controladoras[6] para tratar as diversas requisições, invocando o modelo e atualizando a visão.

2.2.3 Facelets

Facelets é uma tecnologia de apresentação utilizada em aplicações baseadas em JSF. Atualmente, é também a tecnologia definida na especificação do JSF, substituindo assim a descontinuada JSP (JavaServer Pages).

Com esta tecnologia, as páginas web são criadas em XHTML e os componentes são inseridos através de *tags* de diversas bibliotecas. No Algoritmo 2, por exemplo, utiliza-se as *tags* `<h:inputText>` e `<f:validateLongRange>`, das bibliotecas JSF HTML e Core, respectivamente, para a entrada de texto e validação desse texto (após sua conversão) entre um valor mínimo e um valor máximo.

Algoritmo 2 – Utilização de *tags* em um arquivo XHTML.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">

<h:body>
  <h:form>
    <h:inputText id="userNumber"
                 title="Insira um numero entre 0 e 10:"
                 value="#{aBean.userNumber}">
      <f:validateLongRange minimum="#{aBean.minimum}"
                           maximum="#{aBean.maximum}"/>
    </h:inputText>
    <h:commandButton id="submit"
                     value="Submit"
                     action="response"/>
  </h:form>
</h:body>
```

Fonte: Adaptado de Jendrock et al. (2014).

[6] Classes que estendem a classe `HttpServlet`.

Dentre suas vantagens, (JENDROCK et al., 2014) destacam:

- Reuso de código através de *templates* e da composição de componentes.
- Redução do tempo de compilação.
- Validação de expressões em *Expression Language* em tempo de compilação.

2.2.4 Primefaces

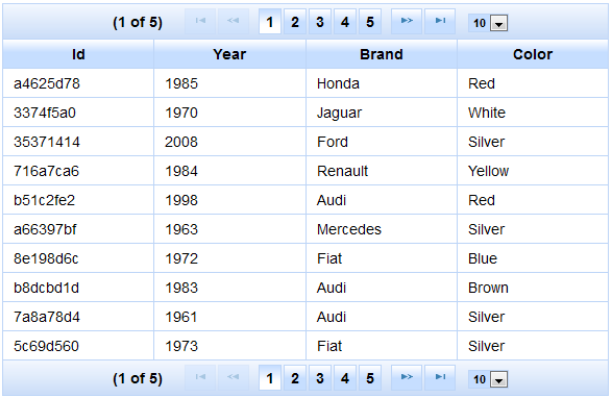
Bibliotecas de componentes oferecem funcionalidades previamente testadas e eventualmente úteis, tais como tabelas capazes de ordenar, filtrar e selecionar linhas, organização do conteúdo em diversos tipos de painéis e menus, exibição de coleções em listas e árvores, etc, além da possibilidade de escolha de diversos temas. Existem diversas bibliotecas disponíveis. As mais conhecidas são: PrimeFaces, RichFaces e ICEFaces. Nos sistemas atualmente desenvolvidos pela SeTIC baseados em Java, utiliza-se a biblioteca PrimeFaces. A Figura 3 mostra um exemplo de tabela feita com essa biblioteca.

2.2.5 Java Persistence API

A API de Persistência do Java (JPA - Java Persistence API), possibilita o gerenciamento de dados relacionais nas aplicações Java através de um mapeamento objeto-relacional. Esse mapeamento é feito através de *entidades* - objetos do domínio de persistência. Uma entidade representa uma tabela de um banco de dados no modelo relacional, e suas instâncias representam as linhas desta tabela. Segundo Jendrock et al. (2014), para ser considerada uma entidade, uma classe precisa:

1. Possuir a anotação `javax.persistence.entity`;
2. Pelo menos um construtor sem argumentos com visibilidade *public* ou *protected*;

Figura 3 – Componente DataTable da biblioteca PrimeFaces. Evidencia-se o uso de um *paginator* no topo da tabela e o uso do tema *bluesky*.



Fonte: PrimeFaces. Disponível em <https://www.primefaces.org/showcase/>

3. A classe, suas instâncias persistíveis e seus métodos não podem ser declarados *final*;

4. Instâncias persistíveis devem ser declaradas como *private*, *protected* ou *package private* e podem ser acessadas somente através dos métodos da classe.

O estado de uma entidade é definido através de seus campos persistíveis, os quais usam anotações de mapeamento objeto-relacional para mapear as entidades e seus relacionamentos aos dados da base de dados. As restrições impostas anteriormente e o relacionamento entre campos podem ser vistos na entidade `PoloCursoNivel` do Algoritmo 3, no qual as restrições foram destacadas com comentários.

Pode-se perceber o relacionamento entre as entidades PoloCursoNivel e Polo através das anotações `@ManyToOne` e `@JoinColumn`. Neste caso, várias instâncias de `PoloCursoNivel` podem referenciar a mesma instância de `Polo`. No banco de dados, a tabela `poloCursoNivel_cnp`

(mapeada através da anotação `@table`) contém a coluna `cd_polo_pol` (indicada pelo parâmetro `name`), que é uma chave estrangeira para a coluna de mesmo nome na tabela mapeada da entidade `Polo` (anotada na classe `Polo` e, portanto, não mostrada na imagem).

Algoritmo 3 – Entidade PoloCursoNivel e seus mapeamentos.

```
@Entity // 1)
@Table(name="poloCursoNivel_cnp", catalog = "capg",
       schema = "dbo")
public class PoloCursoNivel implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    private PoloCursoNivelPK id;

    @ManyToOne
    @JoinColumn(name="cd_polo_pol",
                referencedColumnName="cd_polo_pol")
    private Polo polo;  // 3 e 4)

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name = "cd_curso_cur",
                    referencedColumnName = "cd_curso_cur"),
        @JoinColumn(name = "cd_nivel_niv",
                    referencedColumnName = "cd_nivel_niv")
    })
    private Curso curso;

    // Demais campos

    public PoloCursoNivel() {  // 2)
        super();
    }

    // Getters e setters
}
```

Fonte: Adaptado de Keith e Schincariol (2013).

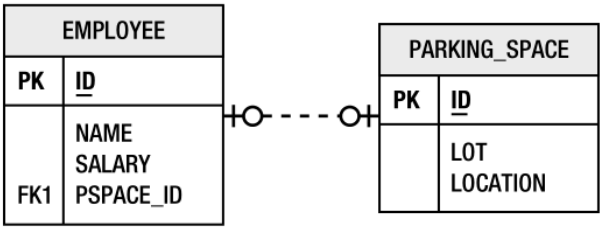
Keith e Schincariol (2013) definem outras anotações utilizadas no relacionamento de entidades além de `@ManyToOne`. As principais e/ou mais utilizadas são discutidas brevemente nos itens a seguir.

2.2.5.1 Relacionamento `@OneToOne`

Apenas uma instância da entidade fonte[7] pode referenciar a mesma instância da entidade alvo. Em termos do banco de dados, isso implica na existência de uma restrição de unicidade na coluna da chave estrangeira da tabela fonte. Ainda, o JPA espera que o nome dessa coluna seja igual ao nome do atributo da entidade fonte seguido de um *underscore* e do nome da coluna da chave primária da tabela alvo. Caso esse não seja o caso, então deve-se fornecer o nome da coluna na anotação `@JoinColumn`.

Para ilustrar essa situação, considere as tabelas ilustradas na Figura 4. O mapeamento feito pelo JPA espera que a coluna com a chave estrangeira a ser utilizada seja chamada de `PARKINGSPACE_ID`, porém essa coluna chama-se `PSPACE_ID`. Neste caso, utiliza-se a anotação `@JoinColumn(name = "PSPACE_ID")`, conforme Algoritmo 4.

Figura 4 – Tabelas EMPLOYEE e PARKING_SPACE.



Fonte: Retirado de Keith e Schincariol (2013).

É interessante notar que o campo `name` não possui nenhuma anotação relacionada à sua persistência. Isso se deve ao fato de existir uma coluna de mesmo nome na tabela `EMPLOYEE`.

[7] A entidade na qual se está fazendo a anotação.

Algoritmo 4 – Classe Employee e seus relacionamentos.

```
@Entity
public class Employee {

  @Id private long id;

  @OneToOne
  @JoinColumn(name = "PSPACE_ID")
  private ParkingSpace parkingSpace;
  // ...
}
```

Fonte: Adaptado de Keith e Schincariol (2013).

Quando a entidade alvo do relacionamento também possuir uma referência à entidade fonte, tem-se então um relacionamento bidirecional, em que uma das entidades é dita ser a *dona* do relacionamento por conter a coluna com a chave estrangeira da outra entidade[8]. Nessa situação, a anotação `@JoinColumn` da entidade que *não* é a dona do relacionamento recebe o elemento `mappedBy = "X"`, onde X refere-se ao nome do campo da entidade que é dona do relacionamento, como pode ser visto no Algoritmo 5.

Algoritmo 5 – Classe ParkingSpace e seus relacionamentos.

```
@Entity
public class ParkingSpace {

  @Id private long id;
  private String location;

  @OneToOne(mappedBy = "parkingSpace")
  private Employee employee;
  // ...
}
```

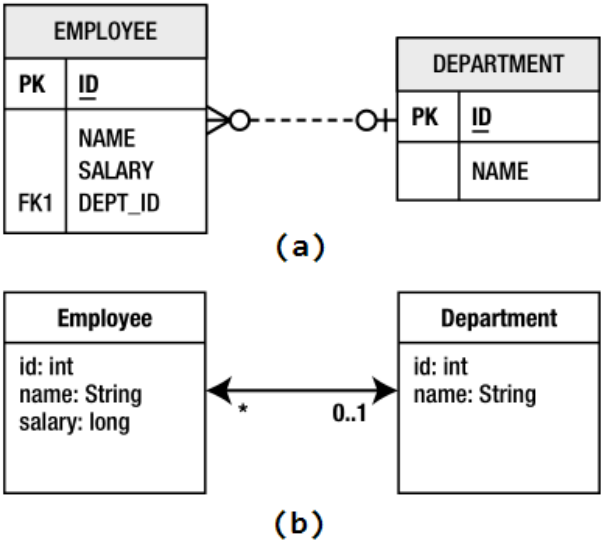
Fonte: Adaptado de Keith e Schincariol (2013).

[8] A escolha de qual das tabelas deverá conter a chave estrangeira da outra tabela é uma decisão de modelagem do banco de dados.

2.2.5.2 Relacionamento `@OneToMany`

Este tipo de relacionamento ocorre quando uma entidade está associada a uma coleção de entidades. Considere as tabelas `EMPLOYEE` e `DEPARTMENT` ilustradas na Figura 5. O relacionamento *um para muitos* feito em UML pode ser visto em (b).

Figura 5 – Tabelas `EMPLOYEE` e `DEPARTMENT` em (a) e seu relacionamento em (b).



Fonte: Adaptado de Keith e Schincariol (2013).

É importante notar que esse relacionamento implica em um relacionamento *muitos para um* quando analisado a partir da tabela `EMPLOYEE` e, portanto, é um relacionamento naturalmente bidirecional. Daí que uma das entidades fará o papel de dona do relacionamento, conforme discutido anteriormente no caso do relacionamento *um para um* bidirecional. Devido ao fato de não existir uma maneira escalonável de guardar inúmeras chaves em uma única linha de uma tabela, a entidade dona do relacionamento nessa situação deve ser a que estiver

no lado "muitos". Nesse caso, a entidade dona é, portanto, a `EMPLOYEE` e nela estarão as chaves estrangeiras para as linhas da tabela `DEPARTMENT`. Por sua vez, isso implica na adição do elemento `mappedBy` na anotação `@JoinColumn` dessa última entidade, conforme Algoritmo 6.

Algoritmo 6 – Classe Department e seus relacionamentos.

```
@Entity
  public class Department {

  @Id
  private long id;
  private String name;

  @OneToMany(mappedBy = "department")
  private Collection<Employee> employees;
  // ...
}
```

Fonte: Adaptado de Keith e Schincariol (2013).

Mais uma vez, devido à utilização de um nome de coluna diferente do valor padrão esperado para esse relacionamento, utiliza-se o elemento `name` na anotação `@JoinColumn` da entidade Employee (vide Algoritmo 7).

Algoritmo 7 – Classe Employee e seus relacionamentos.

```
@Entity
public class Employee {

  @Id
  private long id;

  @ManyToOne
  @JoinColumn(name = "DEPT_ID")
  private Department department;
  // ...
}
```

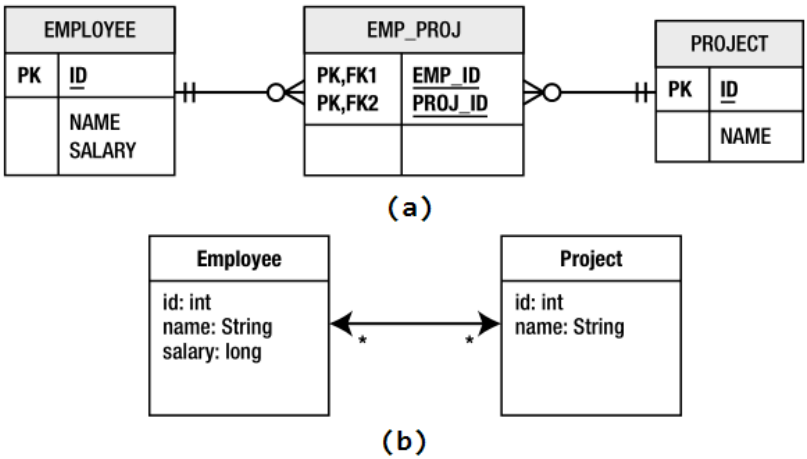
Fonte: Adaptado de Keith e Schincariol (2013).

2.2.5.3 Relacionamento `@ManyToMany`

O último tipo de relacionamento entre entidades apresentado é chamado de *muitos para muitos*. Esse relacionamento ocorre quando uma ou mais entidades fonte estão associadas a uma coleção de entidades alvo, e estas entidades alvo estão associadas a uma coleção de entidades fonte. A representação dessa situação em UML pode ser vista no item (b) da Figura 6 para as entidades Employee e Project.

Considere agora as tabelas ilustradas no item (a) da Figura 6. Uma linha de `EMPLOYEE` pode estar associada a diversas linhas de `PROJECT` e vice-versa. A existência da tabela `EMP_PROJ` justifica-se por não ser escalável guardar as chaves estrangeiras em cada linha de *ambas* as tabelas. Nesse caso, `EMP_PROJ` possui apenas duas colunas e estas guardam as chaves estrangeiras de cada uma das tabelas. Portanto, `EMPLOYEE` não possui uma coluna com as chaves estrageiras de `PROJECT` e vice-versa (KEITH; SCHINCARIOL, 2013).

Figura 6 – Tabelas `EMPLOYEE`, `EMP_PROJ` e `PROJECT` em (a) e seu relacionamento em (b).



Fonte: Adaptado de Keith e Schincariol (2013).

Se ambas as tabelas `EMPLOYEE` e `PROJECT` não possuem uma coluna de chaves estrangeiras, quem é a entidade dona do relacionamento? A resposta é: tanto faz, desde que essa decisão seja indicada ao JPA. As entidades Employee e Project resultantes do mapeamento podem ser vistas no Algoritmo 8. Evidencia-se o uso da anotação `joinColumns`, indicando a entidade dona do relacionamento.

Algoritmo 8 – Classes Employee e Project.

```
@Entity
public class Employee {
  @Id
  private long id;
  private String name;
  @ManyToMany
  @JoinTable(name="EMP_PROJ",
      joinColumns=@JoinColumn(name="EMP_ID"),
      inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
  private Collection<Project> projects;
  // ...
}

@Entity
public class Project {
  @Id
  private long id;
  private String name;
  @ManyToMany(mappedBy="projects")
  private Collection<Employee> employees;
  // ...
}
```

Fonte: Adaptado de Keith e Schincariol (2013).

3 ANÁLISE E PROJETO - MÓDULO DE INSCRIÇÃO

Sommerville (2011) explica que um *processo de software* é um conjunto de atividades relacionadas que levam à produção de um produto de software. Embora existam diversos processos de software, todos devem incluir as seguintes atividades fundamentais: especificação, projeto e implementação, validação, e evolução do software. Cada uma destas atividades é uma tarefa complexa em si mesma e inclui subatividades.

Por não existir um modelo de processo de software único a ser utilizado na SeTIC, o autor decidiu-se por utilizar o mesmo "modelo" de processo de software dos projetos anteriores em que participou na SeTIC como estagiário: as atividades fundamentais supracitadas são executadas de forma iterativa, seguindo as boas práticas especificadas pelo *Rational Unified Process* (RUP), um modelo de processo moderno e híbrido, derivado de trabalhos sobre a UML (Unified Modelling Language).

Ressalta-se que o desenvolvimento se dá por meio da extensão e melhoria de um sistema existente. Far-se-á uso das tabelas utilizadas nesse sistema, bem como dos campos presentes no formulário de inscrição. Não obstante, novas tabelas e campos serão criados, e a implementação das regras de negócio será realizada pelo autor deste trabalho.

3.1 ELICITAÇÃO, ANÁLISE E DEFINIÇÃO DE REQUISITOS

O desenvolvimento de um software/sistema segundo o modelo adotado pode ser dividido em diversos estágios. O primeiro consiste na elicitação, análise e definição de requisitos. Segundo Sommerville (2011), os requisitos de um sistema são as descrições do que o sistema deve fazer, os serviços que oferece e as restrições a seu funcionamento. Esses requisitos refletem as necessidaeds dos clientes para um sistema que serve a uma finalidade determinada, como controlar um dispositivo, fazer um pedido ou encontrar informações. Esta etapa é muito importante pois, além de servir de base para as demais etapas, fornece um rumo a ser seguido e um objetivo a ser alcançado. Não menos importante, evita a implementação de funcionalidades desnecessárias.

A elicitação ou descoberta dos requisitos do sistema pode ser feita de diversas maneiras: entrevistas, cenários, casos de uso, etnografia, etc. A abordagem adotada aqui, comum aos demais sistemas desenvolvidos na SeTIC, foi a de realização de entrevistas abertas, isto é, uma série de questões relacionadas ao uso do sistema foi explorada junto ao cliente. Dessa forma, pôde-se obter uma melhor compreensão de suas necessidades.

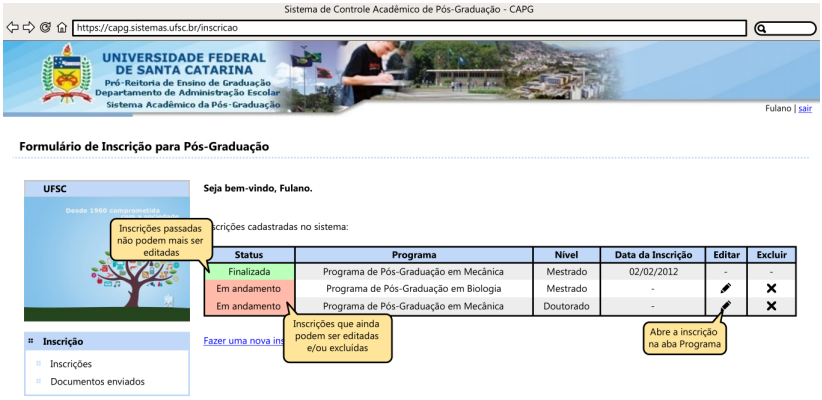
Até o momento da confecção deste relatório, duas reuniões foram realizadas com o cliente. Um conjunto de *wireframes* das páginas do sistema foi desenvolvido[1] e apresentado com o objetivo de validar a estrutura, o conteúdo, as diversas funções e os diversos caminhos de interações dos usuários com o sistema. A confecção deste conjunto de *wireframes* teve como base o sistema atual. Em cima deste foram aplicadas todas as mudanças que se faziam necessárias e que justificam o desenvolvimento do novo sistema. A Figura 7 fornece um exemplo retirado deste conjunto: após ter feito *login* no sistema, o usuário Fulano é recebido com um conjunto de inscrições previamente realizadas por si. Evidencia-se o uso de balões explicativos para agilizar o entendimento dos diversos elementos da página pelo cliente. Ressalta-se que o visual da página não é o foco do *wireframe*, e sim o que ela permite que o usuário realize.

Na reunião com o cliente, os requisitos propostos e exemplificados nos *wireframes* foram analisados. Destes, alguns foram removidos, outros adicionados. No fim, um conjunto de requisitos iniciais[2] para o sistema foi aprovado e servirá como uma especificação formal para o desenvolvimento do sistema. Embora a distinção entre os diferentes tipos de requisitos não seja tão clara, os requisitos de software são frequentemente classificados como *funcionais* e *não funcionais*. Essa classificação é abordada nos itens a seguir.

[1] Utilizou-se um *plugin* do WireframeSketcher feito para o Eclipse.

[2] Em princípio, segundo o modelo em cascata, cada estágio só se inicia após o término do anterior. Na prática, há uma sobreposição dos estágios: problemas com os requisitos são identificados durante o projeto e erros de programa aparecem durante a fase de operação e manutenção, exigindo uma reavaliação dos requisitos.

Figura 7 – *Wireframe* que representa a página de inscrições realizadas pelo usuário Fulano após este ter feito *login* no sistema.



Fonte: Produzido pelo autor.

3.1.1 Requisitos Funcionais

Os requisitos funcionais descrevem os serviços que o sistema deve fornecer, como deve reagir a entradas específicas e como deve se comportar em determinadas situações. A sua descrição é feita aqui de forma abstrata (em alto nível) para que sejam compreendidos pelos diversos *stakeholders* do projeto.

Requisito 1 - Visualizar programas oferecidos. O usuário poderá visualizar a lista de programas de pós-graduação oferecidos pela UFSC sem a necessidade de realizar *login* no sistema.

Requisito 2 - Realizar cadastro no CAS. O usuário poderá se cadastrar no Sistema de Autenticação Centralizada a partir da página inicial do sistema.

Requisito 3 - Acessar o sistema. O usuário poderá acessar o sistema através de *login*.

Requisito 4 - Sair do sistema. O usuário poderá sair do sistema a qualquer momento fazendo *logout*.

Requisito 5 - Salvar a inscrição. O usuário poderá salvar as informações inseridas no formulário para um acesso futuro.

Requisito 6 - Alterar informações. O usuário poderá alterar as informações inseridas durante todo o período de inscrição do programa selecionado, desde que não tenha finalizado a sua inscrição.

Requisito 7 - Finalizar a inscrição. O usuário poderá finalizar a sua inscrição. Após finalizada, o usuário receberá um número de inscrição e os dados inseridos não poderão mais ser modificados.

Requisito 8 - Imprimir um comprovante de inscrição. Após finalizada a inscrição, o usuário poderá imprimir um comprovante de inscrição.

Requisito 9 - Importar dados do CAS. O usuário poderá importar os dados que foram utilizados para fazer o cadastro no CAS.

Requisito 10 - Importar dados de inscrições anteriores. O usuário poderá importar alguns dados de inscrições anteriores ao iniciar uma nova inscrição. Os dados que poderão ser importados serão definidos pela PROPG.

Requisito 11 - Fazer *upload* de arquivos. O usuário poderá fazer o *upload* de arquivos solicitados pelo programa escolhido.

Requisito 12 - Importar arquivos. O usuário poderá importar arquivos utilizados em inscrições anteriores.

Requisito 13 - Visualizar inscrição. O usuário poderá visualizar os dados de sua inscrição em uma página única antes de decidir-se por finalizar a inscrição.

Requisito 14 - Inserir informações. O usuário poderá inserir diversos tipos de informações, as quais são agrupadas em dados do programa, pessoais, econômicos, de contato e de formação. Embora

esses dados não estejam descritos em pormenores neste relatório, estavam presentes nos protótipos apresentados à PROPG na reunião de definição dos requisitos.

3.1.2 Requisitos Não Funcionais

Segundo Sommerville (2011), os requisitos não funcionais são aqueles que não estão diretamente relacionados com os serviços específicos oferecidos pelo sistema a seus usuários. Relacionam-se às propriedades emergentes do sistema e definem restrições sobre a sua implementação. Foram identificados os seguintes requisitos:

- O sistema deve possibilitar o acesso dos usuários durante todos os dias do ano para que possam imprimir comprovantes de inscrição.
- Usuários do sistema devem autenticar-se utilizando o Sistema de Autenticação Centralizada (CAS).
- A aplicação Web deve ser feita utilizando a linguagem de programação Java e as tecnologias que são comumente utilizadas no desenvolvimento Web com Java e que estão disponíveis na SeTIC (vide Seção 2.2).
- A aplicação Web deve utilizar a estrutura e seguir o visual dos demais sistemas desenvolvidos na SeTIC. Para tanto, deve-se utilizar o projeto denominado *Projeto Base*, criado por desenvolvedores da SeTIC para o desenvolvimento de novos sistemas.

3.2 PROJETO DO SISTEMA

O segundo estágio do modelo de desenvolvimento de software adotado consiste em desenvolver a arquitetura geral do sistema. Aqui descrevem-se as principais abstrações do sistema e seus relacionamentos através dos diagramas de casos de uso e de classes.

3.2.1 Diagrama de Casos de Uso

Diagramas de casos de uso fornecem uma representação em alto nível da interação entre os diversos usuários (chamados de atores) com o sistema. As interações são representadas por elipses e os atores por bonecos palito. A Figura 8 abaixo relaciona os casos de uso identificados para o ator Candidato, único ator do módulo de inscrição.

Figura 8 – Casos de uso identificados para o ator Candidato.

Fonte: Elaborada pelo autor.

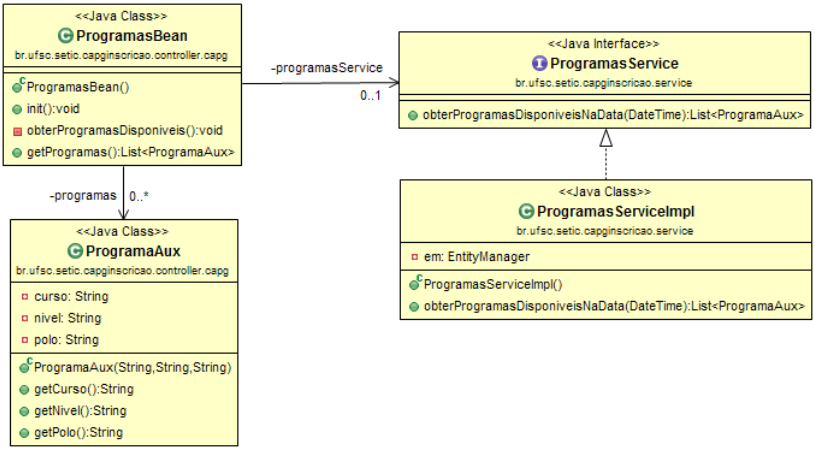
3.2.2 Diagrama de Classes

Um diagrama de classes é responsável por mostrar as diversas classes de uma aplicação e as associações (agregação, composição, generalização e dependências) entre elas.

O diagrama de classes do sistema foi dividido em duas imagens por questão de espaço. A Figura 9 mostra as classes envolvidas na obtenção dos programas disponíveis em uma determinada data. Pode-se perceber

uma associação de generalização entre as classes `ProgramasService` e `ProgramasServiceImpl` e dependências entre as classes `ProgramasBean` e `ProgramaAux`, e `ProgramasBean` e `ProgramasService`.

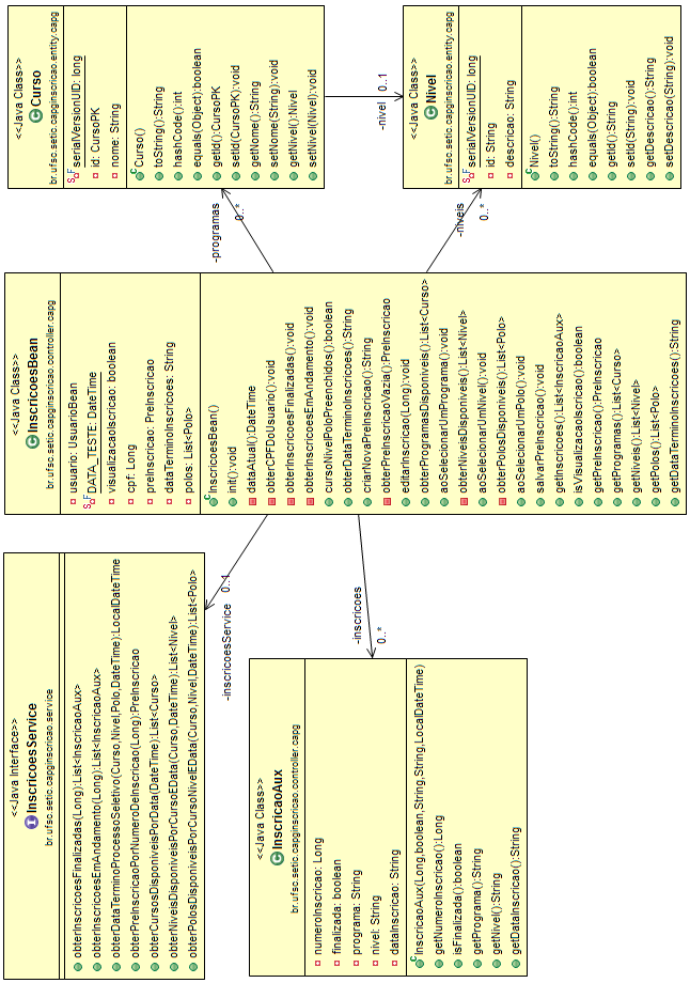
Figura 9 – Modelo entidade-relacionamento relacionado à obtenção de programas.



Fonte: Elaborada pelo autor com a ferramenta ObjectAid UML Explorer.

A Figura 10 ilustra a classe `InscricoesBean` e algumas das suas associações de dependência. As dependências dessas classes não são mostradas (este diagrama pode crescer substancialmente). As classes `Curso` e `Nivel` são entidades e estão relacionadas entre si. `InscricoesService` é uma interface que provê serviços à classe `InscricoesBean` (através de uma classe que a implemente). Finalmente, `InscricaoAux` é uma classe auxiliar que é utilizada junto à *view* para exibir informações das inscrições cadastradas para um determinado usuário.

Figura 10 – Modelo entidade-relacionamento relacionado às inscrições.

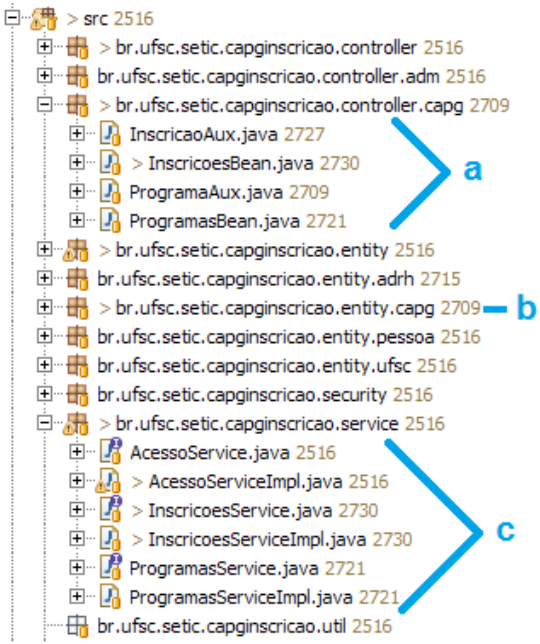


Fonte: Elaborada pelo autor com a ferramenta ObjectAid UML Explorer.

3.3 IMPLEMENTAÇÃO

Após definida a estrutura inicial do sistema, deu-se início ao processo de implementação. O projeto foi configurado para atender às necessidades do sistema. Além de arquivos de configuração, diversos pacotes foram criados para a separação lógica das classes, como pode ser visto na Figura 11.

Figura 11 – Alguns pacotes e classes da aplicação vistos no Eclipse.



Fonte: Elaborada pelo autor.

O pacote `(a)` é composto de *beans* e classes auxiliares da camada Web (vide Seção 2.2.1.2) que, em conjunto com os arquivos .xhtml, são responsáveis por gerar as páginas da aplicação.

O mapeamento objeto-relacional se encontra no pacote destacado por `(b)`. Aqui se encontram as classes das inúmeras entidades do sistema

(vide Seção 2.2.1.3) e, por serem muitas, não foram incluídas na imagem.

As classes representadas por `(c)`, assim como as classes do item anterior, pertencem à camada de negócios. São classes ditas de serviço, responsáveis pelas operações básicas de persistência: *Create*, *Read*, *Update*, *Delete*, comumente abreaviadas por CRUD. Na imagem, as classes `AcessoService`, `InscricoesService` e `InscricoesService` são interfaces[3], enquanto que `AcessoServiceImpl`, `InscricoesServiceImpl` e `InscricoesServiceImpl` representam suas implementações.

Para ilustrar a operação *read*, o Algoritmo 9 contém o método que obtém a lista de programas disponíveis em uma determinada data. Percebe-se que o tipo do elemento contido na lista é um `ProgramaAux`, uma das classes auxialiares mencionadas no item `(a)`. Ainda, `em` é uma instância de um `EntityManager` que está associada a um contexto de persistência.

Ressalta-se que a maioria das classes presentes na Figura 11 já existiam no Projeto Base e provêm funcionalidades como segurança (através do Spring), acesso a Web Services, comunição com a base de dados Centura, validadores e conversores do JSF, etc.

[3] Em Java, uma interface é um tipo abstrato que especifica um comportamento que deve ser implementado.

Algoritmo 9 – Obtenção da lista de programas disponíveis.

```
public List<ProgramaAux> obterProgramasDisponiveisNaData(DateTime data) {
    StringBuffer query = new StringBuffer();
    query.append("SELECT new br.ufsc.setic.capginscricao.controller.capg.ProgramaAux(");
    query.append("ps.curso.nome, ps.nivel.descricao, ps.polo.nome) ");
    query.append("FROM ProcessoSeletivo ps ");
    query.append("WHERE :data BETWEEN ps.dtInicioProcesso AND ps.dtTerminoProcesso ");
    query.append("ORDER BY ps.curso");

    return em.createQuery(query.toString(), ProgramaAux.class)
            .setParameter("data", data.toLocalDateTime())
            .getResultList();
}
```

Fonte: Produzido pelo autor.

3.3.1 Funcionalidades

A Tabela 1 lista os casos de uso relativos aos requisitos especificados na Seção 3.1.1 e que haviam sido implementados *parcialmente* no momento da confecção deste relatório.

Tabela 1 – Casos de uso implementados e respectivas páginas envolvidas.

Caso de Uso	Página da Aplicação
Visualizar programas disponíveis	programas.xhtml
Fazer *login* Fazer *logout*	login.xhtml
Criar inscrição Editar inscrição Excluir inscrição	inscricoes.xhtml

Fonte: Elaborada pelo autor.

Faz-se necessário justificar o uso do termo “parcialmente” no parágrafo anterior. Embora a criação e a edição de inscrições tenham sido implementadas, diversas variáveis de controle e mesmo de estado serão criadas ao longo do desenvolvimento deste projeto para atender aos demais requisitos especificados. Muitas dessas variáveis serão utilizadas na criação e edição de inscrições, implicando em uma constante refatoração do código já implementado. Daí que não se pode afirmar que os casos de uso citados tenham sido implementados em sua totalidade.

4 CONCLUSÕES

O sistema de inscrição atual é utilizado por apenas alguns dos cursos de Pós-Graduação da UFSC. Alguns cursos utilizam seus próprios sistemas, dificultando a sua mantenibilidade e introduzindo duplicações de código e vulnerabilidades.

Torna-se imprescindível, portanto, o desenvolvimento de um novo sistema de inscrição unificado, que dê suporte a novas funcionalidades e que utilize as tecnologias atualmente em uso na SeTIC.

Algumas atividades previstas na proposta do TCC foram desenvolvidas neste primeiro semestre da disciplina de Projetos, especificamente:

- Levantamento das funcionalidades;
- Especificação dos requisitos do sistema a partir destas funcionalidades
- Definição da arquitetura do sistema;
- Adequação do Projeto Base para atender a estrutura definida;
- implementação dos casos de uso discutidos na Seção 3.3.1.

REFERÊNCIAS

ALMEIDA, A. *Entenda os MVCs e os frameworks Action e Component Based*. 2012. [Acesso em 25/04/2017]. Disponível em: <http://blog.caelum.com.br/entenda-os-mvcs-e-os-frameworks-action-e-component-based/>. Citado na página 15.

BUSCHMANN, F. et al. *Pattern-Oriented Software Architecture: A System Of Patterns*. Chichester, Inglaterra: John Wiley & Sons Ltd, 1996. Citado na página 14.

HUNTER, J.; CRAWFORD, W. *Java Servlet Programming*. Sebastopol, California: O'Reilly & Associates, Inc., 1998. Citado na página 12.

JENDROCK, E. et al. *Java Platform, Enterprise Edition: The Java EE Tutorial*. 2014. [Acesso em 22/02/2017]. Disponível em: <https://docs.oracle.com/javaee/7/tutorial/>. Citado 4 vezes nas páginas 9, 10, 16 e 17.

KEITH, M.; SCHINCARIOL, M. *Pro JPA 2*: A definitive guide to mastering the java persistence api. Second. Chichester, Inglaterra: Apress, 2013. Citado 7 vezes nas páginas 19, 20, 21, 22, 23, 24 e 25.

LHOTKA, R. *Should all apps be n-tier?* 2005. [Acesso em 23/02/2017]. Disponível em: <http://www.lhotka.net/weblog/ShouldAllAppsBeNtier.aspx>. Citado na página 9.

SOMMERVILLE, I. *Engenharia de Software*. 9. ed. São Paulo: Pearson Prentice Hall, 2011. Citado 2 vezes nas páginas 27 e 31.