ARQUITETURA DE SISTEMAS

HENRIQUE EDUARDO DA COSTA MENEL

JOANA JENSEN SCHIFTER

JUSCELINO RODRIGUES BRANDÃO

JARAGUÁ DO SUL

2025

HENRIQUE EDUARDO DA COSTA MENEL

JOANA JENSEN SCHIFTER

JUSCELINO RODRIGUES BRANDÃO

ARQUITETURA DE SISTEMAS

Desenvolvimento de código e documentação técnica baseado nos conteúdos desenvolvidos em sala. Este documento será utilizado como critério avaliativo para o curso de Desenvolvimento de Sistemas na unidade curricular Arquitetura de Sistemas, solicitada pelo SENAI-CentroWeg e apresentada ao docente Matheus Quost.

JARAGUÁ DO SUL

2025

Sumário

1 – Introdução

Durante a disciplina de Arquitetura de Sistemas, foi proposto a nós alunos que desenvolvêssemos um sistema capaz de atender as necessidades de uma empresa fictícia unindo código e documentação. A seguir será apresentado todas as funcionalidades aplicadas no programa, como os princípios SOLID aprendidos em sala e Padrões de Arquitetura.

2 – Análise do Problema e Requisitos

2.1 – Especificação do problema

Durante o programa de capacitação de aprendizes CentroWeg, as matérias disciplinares dos cursos são divididas em unidades curriculares, em visão de melhor organização. Para que a contabilização de presença nas aulas seja feita, o professor precisa acessar um sistema de chamada online onde marca quais alunos estão presentes e quais estão faltantes.

O principal problema enfrentado pelos alunos, professores e equipe pedagógica, no contexto de provas, é a confusão que os alunos encontram quando faltam e por ventura perdem uma atividade avaliativa feita no dia.

Atualmente, os pedidos de segunda chamada são realizados através do sistema disponibilizado pelo SENAI chamado “Espaço do Estudante”, onde os alunos enviam seus atestados (declarações, etc), onde aguardam a autorização da solicitação. Todavia, para que esta solicitação seja realizada e enviada para análise, além do atestado é necessário que o aluno selecione a data exata em que a atividade foi realizada, fazendo com que os estudantes precisam questionar seus colegas e professores, causando possibilidade de falhas de comunicação ou aborrecimento.

2.2 – Principais funcionalidades

1. Verificação de segurança
    - Apenas dois atores podem ter acesso ao sistema, sendo eles, o professor e o administrador, cada um com suas respectivas responsabilidades
    - As verificações de segurança são feitas através de uma simulação de “login” com usuário e senha que serão previamente cadastrados no banco de dados

2. Cadastro de alunos

- Os valores que devem ser inseridos para o cadastro funcional de alunos deve ser: nome, matrícula, turma e status (o status é definido de acordo com a presença do aluno, sendo ‘presente’ o valor default)
- Apenas usuários autenticados como administrador podem cadastrar novos alunos, excluir ou alterar alunos já existentes.

3. Cadastro de professores
    - Os valores que devem ser inseridos para o cadastro funcional de professores deve ser: nome, telefone e CPF
    - Apenas usuários autenticados como administrador podem cadastrar novos professores, excluir ou alterar professores já existentes.

4. Cadastro de unidades curriculares
    - Os valores que devem ser inseridos para o cadastro funcional de unidades curriculares deve ser: nome, data de início e professor
    - Apenas usuários autenticados como administrador podem cadastrar novas unidades curriculares

5. Registro diário de alunos
    - Os alunos serão apresentados em um menu ao professor como uma lista, apresentando seus nomes, matrícula e turma
    - O professor deve selecionar os alunos faltantes através do Id, alterando seu status do dia para “ausente”

6. Sistema de notificação
    - Identifica o aluno faltante e permite ao administrador que envie uma notificação (via e-mail, sms, etc) ao aluno ausente

2.3 – Atores envolvidos

Os atores envolvidos no sistema podem ser identificados através das entidades “Professor” e “Administrador”, já que ambas atuam com frequência nos fluxos do sistema, ditando quais atitudes o programa pode ceder a cada uma delas.

2.4 – Fluxos iniciais

Para o fluxo inicial, identificamos o login do usuário, sendo que ele pode se identificar como “professor” e “administrador”, cada um com suas respectivas funções.

Começaremos com o login de administrador, inserindo alunos, professores e unidades curriculares dentro do banco de dados do sistema.

Em seguida, o professor analisa quais alunos estão presentes no dia, para que assim que um aluno estiver ausente no dia de avaliação, o administrador conseguir enviar uma notificação a ele, indicando o dia da avaliação e a unidade curricular que foi perdida para que ele consiga recuperar.

2.5 – Requisitos funcionais

1. Desempenho
    - O sistema deve ser capaz de processar até 500 registros diários de presença por minuto sem que ocorra interrupções ou degradação da performance
    - Consultas aos registros de alunos, professores ou unidades curriculares devem retornar resultados em menos de 2 segundos

2. Segurança
    - Autenticação de entidades obrigatória antes que o usuário possa acessar o sistema
    - Senhas armazenadas de forma criptografada
    - Apenas usuários com login de administrador podem cadastrar alunos, professores e enviar notificações
    - O sistema registra logs de acesso e alterações realizadas, garantindo rastreabilidade dos usuários que realizarem alterações
    - A comunicação entre cliente e servidor é feita através do protocolo HTTPS

3. Escalabilidade
    - O sistema permite expansão até 10mil alunos e 500 professores sem alterações significativas na arquitetura do sistema
    - O design permite a inclusão de novos módulos, como a integração com sistemas externos e envio automatizado de notificações

4. Disponibilidade
    - O sistema garante a disponibilidade mínima de 95% em horário escolar (das 7h até as 22h)
    - Mensagens de erro amigáveis são exibidas em caso de indisponibilidade temporária

5. Usabilidade
    - Interface simples e clara, com linguagem acessível ao usuário leigo
    - Menus e fluxos permitem que professores registrem faltas em até 3 cliques
    - Validação de dados é feita na entrada para evitar cadastros incorretos (ex: matrícula inválida, CPF mal formatado)

6. Manutenibilidade
    - Código organizado com forme padrões de fluxo e arquitetura para facilitar futuras alterações
    - Utilização de DAO e padrões de projeto para separação de responsabilidades e facilidade de manutenção

7. Portabilidade
    - O sistema é executável em Windows, Linux e MacOS (ou em container, caso o usuário esteja utilizando o sistema via WEB)
    - Possibilidade de migração para diferentes bancos de dados relacionais (PostgreSQL, Azure, MariaDB)

8. Confiabilidade
    - Registros de presença não são perdidos em caso de falha
    - Operações críticas, como a marcação de faltas ou envio de notificações, são transacionais (ou seja, ou se completam totalmente ou não se executam)

3 – Escolha e Justificativa da Arquitetura

3.1 – Escolha de estilo arquitetural

Para melhor adequarmos o nosso sistema, escolhemos o padrão arquitetural MVC (Model-View-Controller) que separa o sistema em três camadas principais.

3.2 – Justificativa Técnica

A escolha da arquitetura MVC (Model-View-Controller) para o desenvolvimento do sistema foi baseada em requisitos específicos que visam facilitar a manutenção, escalabilidade e integração com outros sistemas já existentes no contexto do CentroWeg. A arquitetura MVC oferece uma clara separação de responsabilidades, onde cada camada (Model, View e Controller) é responsável por funções distintas. O Model lida com a lógica de negócios e o acesso a dados, o Controller gerencia a interação entre o Model e a View, enquanto a View apresenta os dados ao usuário. Essa divisão facilita a organização do código, permitindo uma manutenção mais eficiente e uma melhor compreensão do funcionamento do sistema, características especialmente importantes em um projeto acadêmico, onde a clareza e a didática do código são essenciais.

Outro ponto importante da arquitetura MVC é a integração com o DAO (Data Access Object), que se comunica diretamente com a camada Model para realizar as operações de persistência. O uso do DAO garante que a lógica de negócios seja isolada

da camada de acesso a dados, o que simplifica a manutenção e amplia a flexibilidade para futuras modificações no sistema, como a mudança do banco de dados ou a adição de novas fontes de dados.

A escalabilidade e a facilidade de manutenção futura também são características atendidas pela arquitetura MVC. Novas funcionalidades, como o envio de notificações automáticas para alunos ausentes ou a inclusão de novos tipos de usuários, podem ser adicionadas sem impactar outras camadas do sistema. Isso permite que o sistema evolua de forma modular, minimizando o risco de introdução de erros e facilitando a adaptação a novas demandas ao longo do tempo.

A flexibilidade do MVC em relação a diferentes interfaces de usuário também é uma vantagem significativa. A camada de View pode ser modificada para suportar diferentes tipos de interfaces, como console, web ou mobile, sem a necessidade de alterar a lógica central do Model ou Controller. Isso garante que o sistema possa ser facilmente adaptado a diferentes plataformas e dispositivos, sem comprometer a consistência do código.

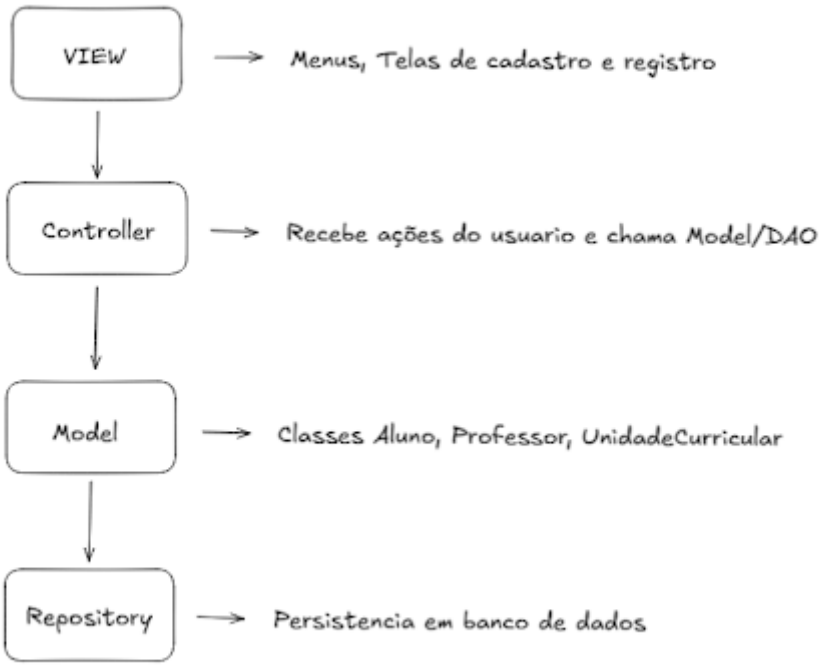
Portanto, a arquitetura MVC não só atende aos requisitos técnicos do sistema, mas também oferece uma base sólida para o desenvolvimento de uma aplicação escalável, modular e fácil de manter, ao mesmo tempo em que permite a integração com os sistemas já existentes no SENAI.

3.3 – Rationale Arquitetural

A arquitetura do sistema proposta visa garantir a organização, modularidade e facilidade de manutenção. A ideia do sistema busca proporcionar uma base sólida para a construção de funcionalidades futuras, ao mesmo tempo em que assegura uma estrutura compreensível para a apresentação de conceitos fundamentais, como o padrão DAO, a implementação de um login seguro e a clara separação de responsabilidades dentro do código. A modularidade facilita a manutenção do código, permitindo que novos módulos ou funcionalidades sejam adicionados sem comprometer o funcionamento das partes já existentes, o que também permite a expansão futura do sistema de maneira eficiente.

No entanto, essa abordagem também apresenta alguns trade-offs. A separação em diversas camadas e a modularização do sistema resultam em um maior número de classes quando comparado a uma abordagem monolítica mais simples. Essa complexidade adicional pode ser vista como um obstáculo em sistemas menores ou em ambientes onde a agilidade no desenvolvimento é mais importante do que a manutenção a longo prazo. Contudo, essa estrutura modular se justifica, especialmente pela clareza que proporciona, o que torna o aprendizado sobre conceitos fundamentais mais acessível, além de garantir que o sistema possa ser mantido e expandido de forma sustentável ao longo do tempo.

3.4 – Visão Geral da Arquitetura Proposta



(Imagem criada pelos alunos)

4 – Aplicação de Padrões de Projeto

4.1 - Padrões aplicados

Tipo	Padrão	Onde foi usado	Função
Criacional	Factory Method	Sistema de Notificação (NotificadorFactory)	Cria objetos de notificação (E-mail, SMS) de forma desacoplada
Estrutural	Facade	Sistema de Notificação (NotificadorFacade)	Simplifica a interface de envio de notificações para o Controller
Comportamental	Observer	Sistema de Notificação (NotificadorObserver)	Alunos observam mudanças de status; notificações são disparadas automaticamente

4.2 - Local onde cada padrão foi utilizado no sistema

Factory Method –

Local: *service.notificacao*
Objetivo: *permitir a criação do Notificador (E-mail ou SMS) sem acoplar o Controller a implantação concreta.*
Classes: *Notificador, Email, SMS, NotificadorFactory, EmailFactory, SMSFactory*

Facade –

Local: *service.facade*
Facade: *simplificar a interface de notificação combinando o Factory com a lógica de envio.*
Classe: *NotificadorFacade*

Observer –

Local: *service.observer*
Objetivo: *monitorar as alterações no status do aluno e disparar notificações ao sistema automaticamente.*
Classes: *Observador, AlunoObservador, RegistroPresenca*

4.3 - Código-fonte comentado

Factory Method –

```
package org.example.patterns;

import org.example.model.*;

public class UserFactory {

   // Método fábrica que cria instâncias de usuários de forma padronizada
   public static User createUser(String tipo, String nome, String email, String senha,
                                 String telefone, String cpf, String matricula) {
      return switch (tipo.toUpperCase()) {
         case "ADMIN" -> new Admin(nome, email, senha, telefone, cpf);
         case "PROFESSOR" -> new Professor(nome, email, senha, telefone, cpf);
         case "ALUNO" -> new Aluno(nome, email, senha, telefone, cpf, matricula);
         default -> throw new IllegalArgumentException("Tipo inválido: " + tipo);
      };
   }
}
```

Facade –

```
package org.example.patterns;

import org.example.repository.TurmaDAO;
import org.example.repository.UserDAO;
import org.example.model.*;

public class AdminFacade {
```

```
    private final UserDAO userDAO = new UserDAO();
    private final TurmaDAO turmaDAO = new TurmaDAO();

    // Cadastra um aluno completo com ajuda da fábrica
    public int cadastrarAluno(String nome, String email, String senha, String telefone,
                              String cpf, String matricula){
        Aluno a = (Aluno) UserFactory.createUser("ALUNO", nome, email, senha, telefone, cpf,
matricula);
        return userDAO.saveUser(a);
    }

    // Cadastra um professor
    public int cadastrarProfessor(String nome, String email, String senha, String telefone, String cpf){
        Professor p = (Professor) UserFactory.createUser("PROFESSOR", nome, email, senha,
telefone, cpf, null);
        return userDAO.saveUser(p);
    }

    // Cria uma nova turma
    public void criarTurma(String nome, String curso){
        turmaDAO.inserir(new Turma(nome, curso));
    }
}
```

Observer –

- Interface Observer

```
package org.example.patterns.observer;

import org.example.model.Notificacao;

public interface Observer {
    void update(Notificacao notificacao);
}
```

- Interface Subject

```
package org.example.patterns.observer;

import org.example.model.Notificacao;

public interface Subject {
    void addObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers(Notificacao notificacao);
}
```

- Classe concreta NotificacaoSubject

```
package org.example.patterns.observer;

import org.example.model.Notificacao;
import java.util.ArrayList;
import java.util.List;

public class NotificacaoSubject implements Subject {
    private final List<Observer> observers = new ArrayList<>();

    @Override
    public void addObserver(Observer o) {
        observers.add(o);
    }

    @Override
    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    @Override
    public void notifyObservers(Notificacao notificacao) {
        for (Observer o : observers) {
            o.update(notificacao);
        }
    }
}
```

- Observador concreto — AlunoObserver

```
package org.example.patterns.observer;

import org.example.model.Notificacao;

public class AlunoObserver implements Observer {
    private final int alunoId;

    public AlunoObserver(int alunoId) {
        this.alunoId = alunoId;
    }

    @Override
    public void update(Notificacao notificacao) {
        if (notificacao.getAlunoId() == alunoId) {
            System.out.println("🔔 Nova notificação recebida: " + notificacao.getMensagem());
        }
    }
}
```

- Integração no NotificationService

```
package org.example.service;
```

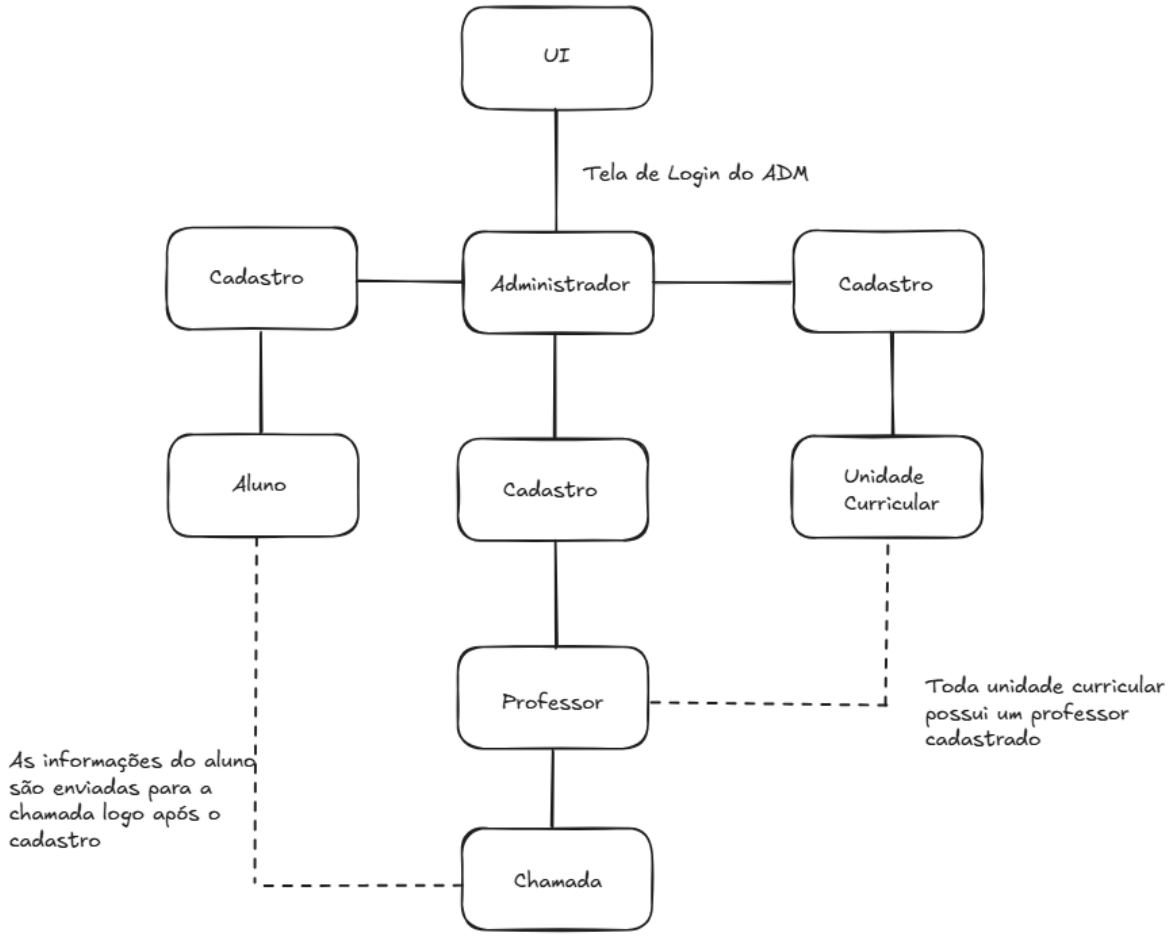
```
import org.example.model.Notificacao;
import org.example.patterns.observer.NotificacaoSubject;
import org.example.repository.NotificacaoDAO;
import java.util.List;

public class NotificationService {
    private final NotificacaoDAO repository = new NotificacaoDAO();
    private final NotificacaoSubject subject = new NotificacaoSubject();

    public void adicionarObserverAluno(int alunoId) {
        subject.addObserver(new org.example.patterns.observer.AlunoObserver(alunoId));
    }

    public void notificarFalta(int alunoId, String mensagem) {
        Notificacao n = new Notificacao(alunoId, mensagem);
        repository.salvar(n);
        subject.notifyObservers(n);
    }

    public List<Notificacao> listarNotificacoes(int alunoId) {
        return repository.buscarPorAluno(alunoId);
    }
}
```

5 – Modelagem Arquitetural

5.1 – Diagramas

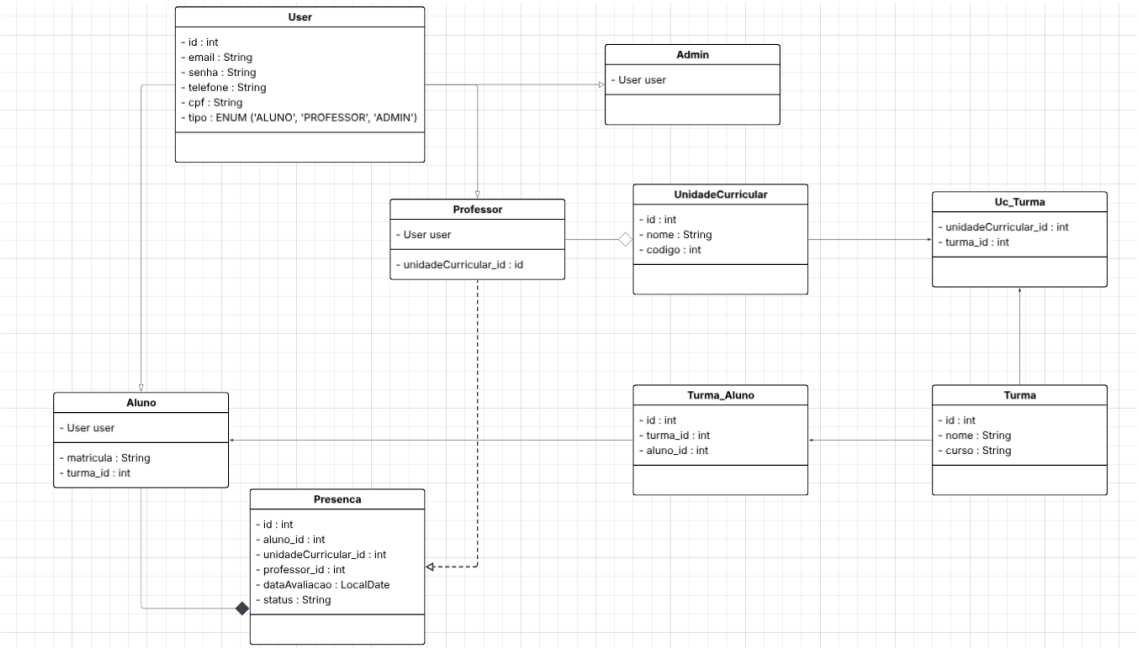
A seguir, diagramas que nos guiaram para desenvolver nosso sistema, auxiliando e fazendo com que consigamos nos manter em ordem conversando com todos os desenvolvedores do sistema, tornando mais fácil de explicar para os outros.

5.1.1 – Diagrama de Componentes



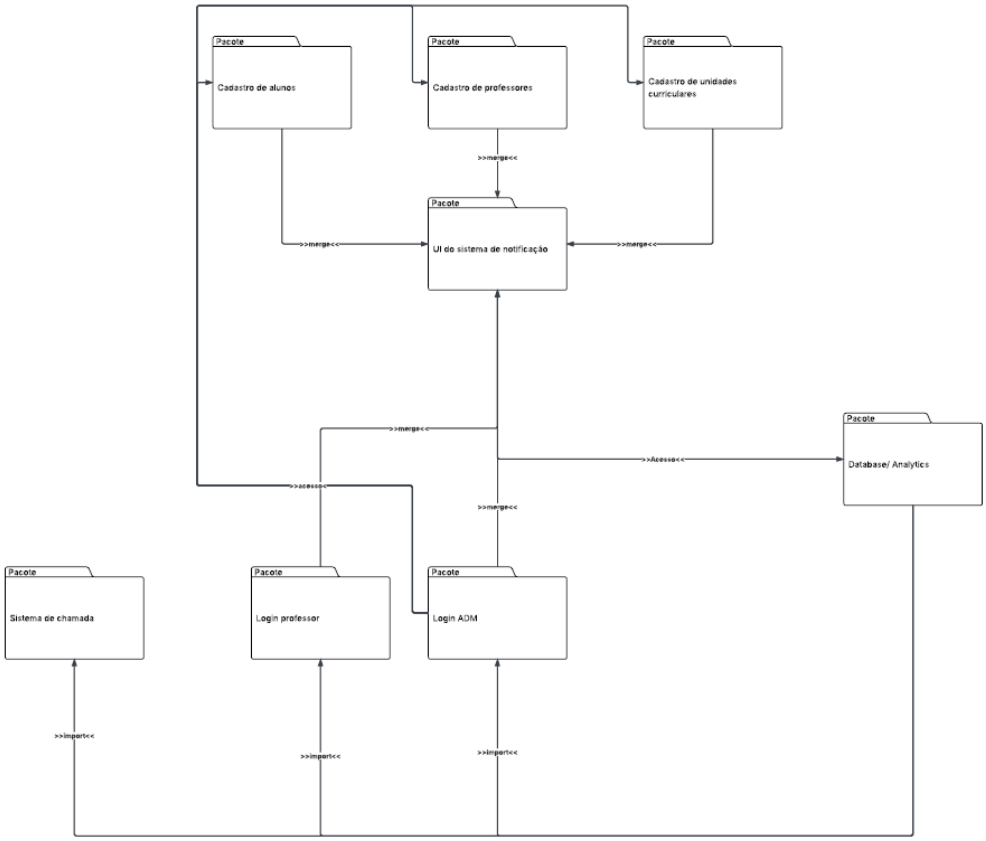
(Imagem criada pelos alunos)

5.1.2 – Diagrama de Classes



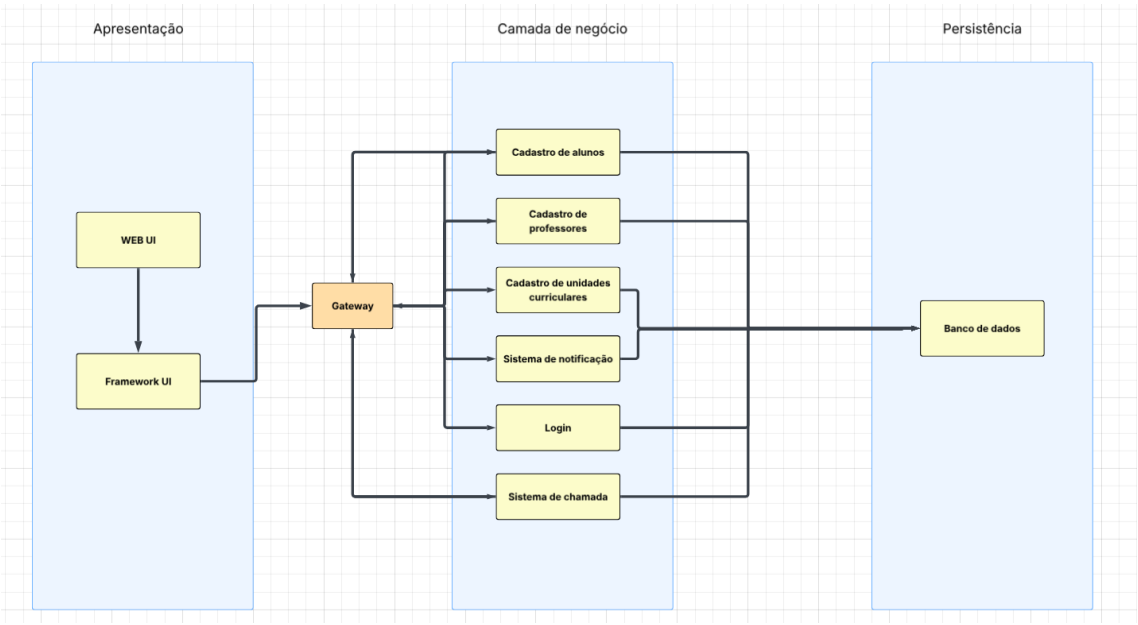
(Imagem criada pelos alunos)

5.1.3 – Diagrama de pacotes



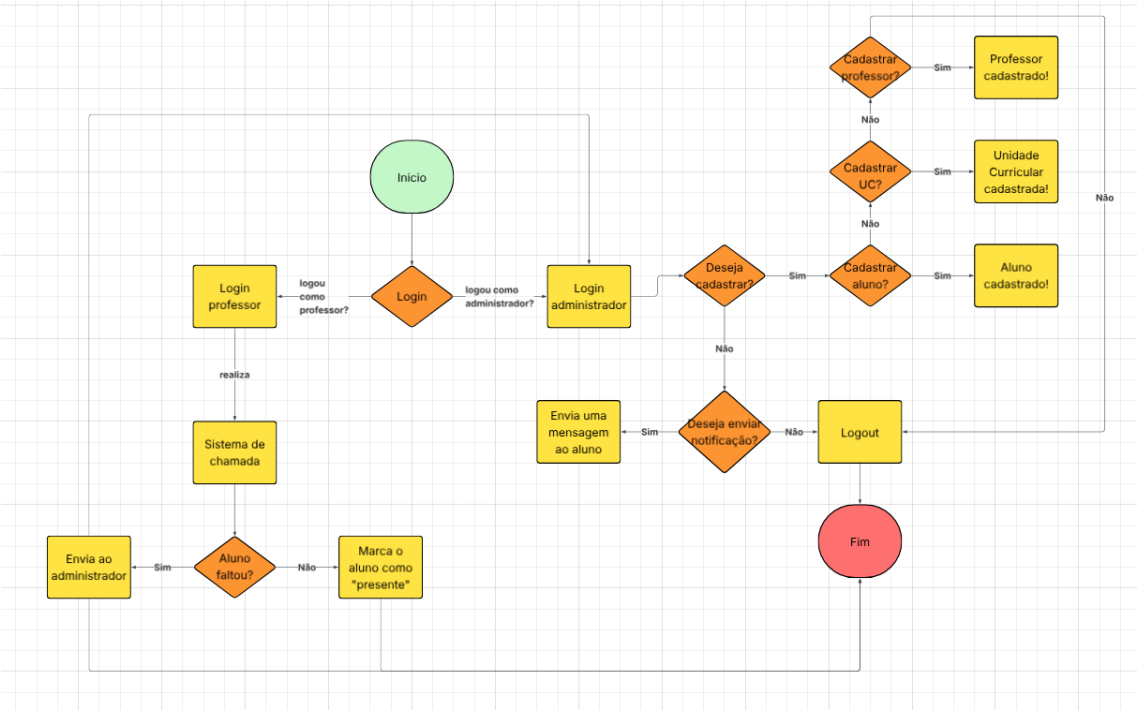
(Imagem criada pelos alunos)

5.1.4 - Diagrama de Camadas



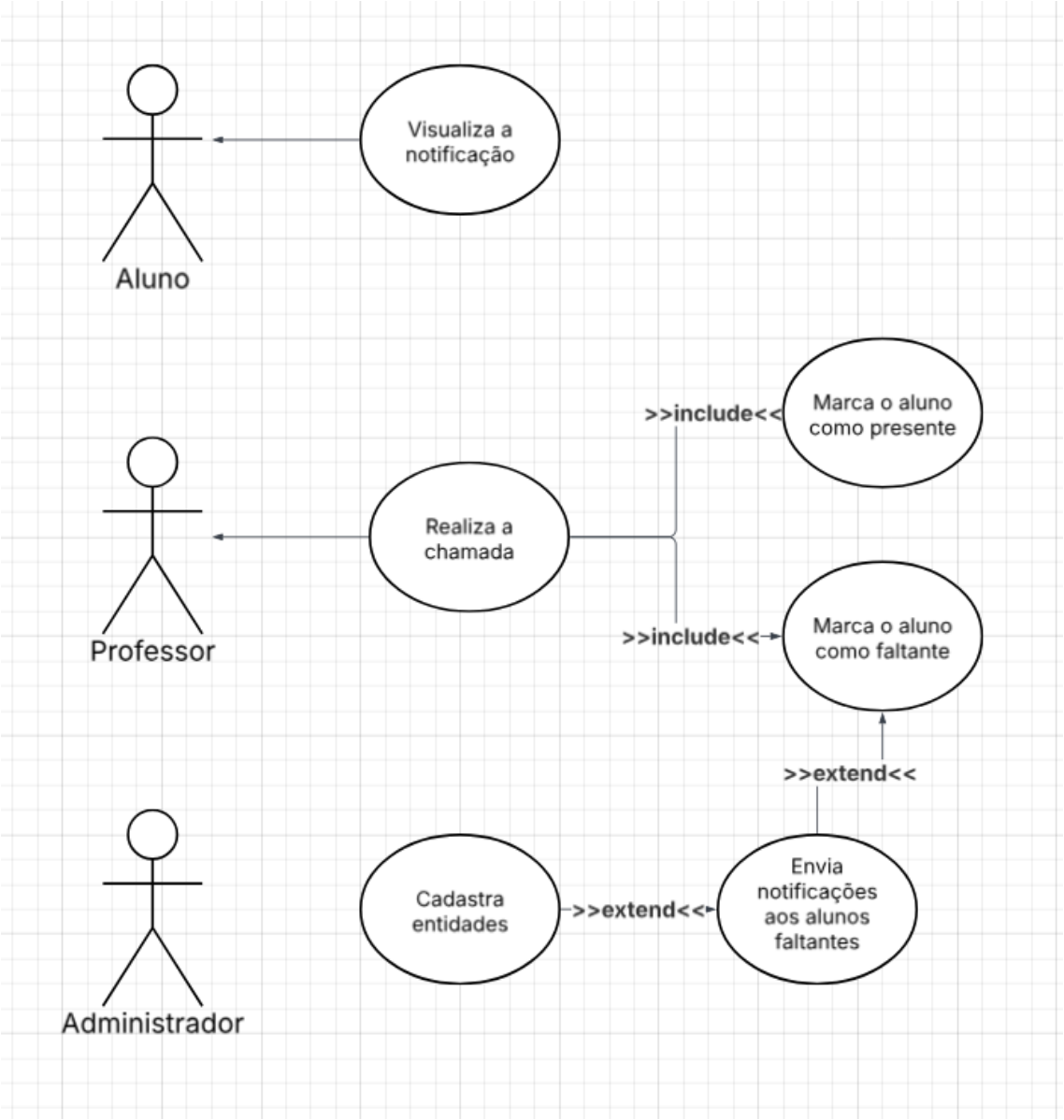
(Imagem criada pelos alunos)

5.1.5 - Diagrama de Conectores



(Imagem criada pelos alunos)

5.1.6 - Diagramas de Casos de Uso



(Imagem criada pelos alunos)

6 – Estratégias de Segurança e Qualidade

6.1 - Pontos críticos de segurança

Os principais pontos que implicam falhas de segurança no nosso sistema são: o armazenamento de senhas sem *hash*, tornando o sistema suscetível a invasões; a injeção SQL, por mais que seja mitigado com *PreparedStatement* ainda é perigoso; o controle de acesso que pode acabar permitindo um professor editar/cadastrar sem ter a devida permissão, por conta da falta do *SpringSecurity*; a exposição de dados pessoais dos alunos através das tabelas SQL; a falta de logs para as alterações críticas, o'que faz com que os técnicos do sistema fiquem sem saber exatamente onde as alterações estão; a falta de backup / recuperação de dados.

6.2 - Estratégias aplicadas

As estratégias aplicadas para o desenvolvimento do projeto foram: a autenticação com fluxo de login com verificação de e-mail e senha; autorização e controle de acesso através dos papéis de “admin”, “professor”, “aluno” que verifica o papel do usuário antes de permitir o acesso as partes do sistema; validações através de tamanho de campos na hora do preenchimento, restrições (ex: matricula) e validação de formatos.

6.3 - Aplicação de princípios SOLID

- **Single Responsibility**: cada camada (DAO, Service, Controller, View) tem responsabilidade única.
- **Open/Closed**: serviços permitem extensão (ex.: adicionar canal de notificação).
- **Liskov**: subtipos Aluno, Professor, Admin seguem User.
- **Interface Segregation**: separar contratos (por exemplo, NotificationSender interface).
- **Dependency Inversion**: controllers dependem de abstrações (por ex., NotificationService interface).

6.4 - Separação clara de responsabilidades

A separação de responsabilidades foi feita da seguinte forma:

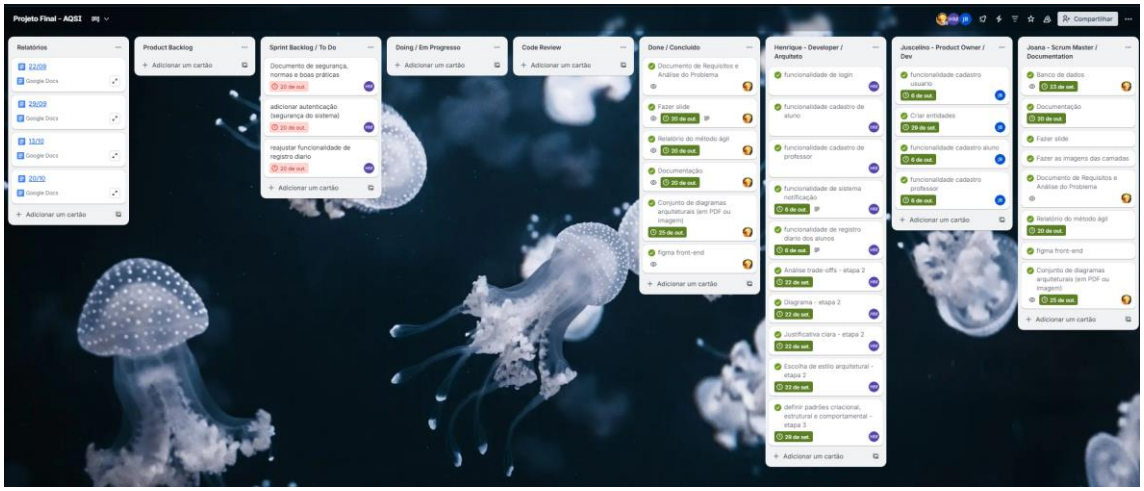
- Model
- DAO
- Service
- Controller
- View
- Infra (conexão ao banco de dados)

7 – Organização do Trabalho e Metodologia

7.1 - Método ágil adotado

O método ágil escolhido pela nossa equipe foi o Scrum, já que nosso time tem mais conhecimento e facilidade com o método.

7.2 - Registro de tarefas



Nosso registro de tarefas foi feito através do Trello.

7.3 - Divisão de papéis

A divisão de papéis foi feita a partir da facilidade que cada um possui, para agilizar os processos, o Henrique e o Juscelino ficaram com a maior parte da programação e aplicação do método ágil, comigo (Joana) sendo responsável por ter certeza que os padrões de projeto e arquitetura fossem aplicados e registrar e escrever todo processo do projeto.

Especificamente, nos dividimos em:

- Henrique: Developer

- Joana: Scrum Master / Developer

- Juscelino: Product Owner / Developer

7.4 - Reflexão final

A metodologia Scrum, por ser a mais conhecida por nosso time, tornou-se realmente a melhor escolha, por vezes, membros da equipe faltaram, tornando o processo dificultoso de seguir exatamente as tarefas definidas no escopo, já que cada

participante teve que fazer um pouco do trabalho do outro. Porém, de todo modo, conseguimos ter um processo proveitoso.