UNIVERSIDADE PAULISTA

ICET - INSTITUTO DE CIÊNCIAS EXATAS E TECNOLOGIA

CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

PROJETO INTEGRADO MULTIDISCIPLINAR

PIM II

Desenvolvimento de um Sistema Acadêmico Colaborativo com Apoio de IA

Nome	R.A
ENZO XAVIER SANTOS	R958990
FERNANDA DA ROCHA GOMES	F365166
JOÃO PEDRO ANTONIO PEREIRA	R860CI1
JOÃO LUCAS LEMES MUASSAB	H77GCH3

SÃO JOSÉ DOS CAMPOS – SP

NOVEMBRO / 2025

	RA
JOÃO LUCAS LEMES MUASSAB	H77GCH3
ENZO XAVIER SANTOS	R958990
FERNANDA DA ROCHA GOMES	F365166
JOÃO PEDRO ANTONIO PEREIRA	R860CI1

Desenvolvimento de um Sistema Acadêmico Colaborativo com Apoio de IA

Projeto Integrado Multidisciplinar (PIM) desenvolvido como exigência parcial dos requisitos obrigatórios à aprovação semestral no Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas da UNIP (Universidade Paulista), orientado pelo corpo docente do curso.

São José dos Campos – SP

Novembro / 2025

RESUMO

Este trabalho apresentou o desenvolvimento de um Sistema Acadêmico Colaborativo com apoio de Inteligência Artificial, projeto integrado multidisciplinar (PIM II) realizado no curso de Análise e Desenvolvimento de Sistemas da UNIP. O objetivo foi projetar e implementar uma solução computacional integrada para gerenciar turmas, alunos, aulas e atividades em instituições de ensino, substituindo controles descentralizados e eliminando o uso de papel. A metodologia adotou a engenharia de software ágil (Scrum) para organização das sprints e backlog, aplicando conceitos de programação estruturada em C para módulos críticos de persistência de dados e estruturas de dados em Python para interface do usuário e algoritmos. Foram desenvolvidos diagramas UML (casos de uso, classes, sequência) e diagrama de rede para a documentação do sistema. Os módulos em C implementaram operações CRUD (Create, Read, Update, Delete) para alunos, turmas, atividades e aulas, com armazenamento em arquivos CSV. A aplicação cliente em Python, desenvolvida com Tkinter, oferece interface gráfica intuitiva com três níveis de acesso (administrador, professor e aluno). Recursos de inteligência artificial foram aplicados para otimizações na legibilidade, desempenho e estrutura do código-fonte. O sistema foi validado através de testes unitários na interface e nos módulos, demonstrando funcionalidade completa e atendimento aos requisitos de sustentabilidade através da geração de relatórios digitais. O projeto integrou com sucesso os conhecimentos das oito disciplinas do semestre, resultando em uma solução funcional e alinhada aos princípios de educação ambiental.

Palavras-chave: Sistema Acadêmico; Projeto Multidisciplinar; Programação em C; Interface Python; Inteligência Artificial; Engenharia de software ágil.

SUMÁRIO

1. INTRODUÇÃO

A transformação digital tem revolucionado diversos setores da sociedade, e a educação não é exceção. Instituições de ensino enfrentam desafios crescentes na gestão de informações acadêmicas, que historicamente tem sido realizada de forma descentralizada através de planilhas, e-mails e documentos físicos.

Esta fragmentação de dados não apenas dificulta o acesso à informação, mas também contribui para o desperdício de papel e recursos, impactando negativamente o meio ambiente. Neste contexto, o desenvolvimento de sistemas computacionais integrados surge como solução estratégica para modernizar a gestão acadêmica. A capacidade de centralizar informações, automatizar processos e permitir acesso simultâneo de múltiplos usuários em rede representa um avanço significativo em termos de eficiência operacional e sustentabilidade.

O presente trabalho documenta o desenvolvimento de um Sistema Acadêmico Colaborativo com apoio de Inteligência Artificial, resultado do Projeto Integrado Multidisciplinar II (PIM II) do curso de Análise e Desenvolvimento de Sistemas da UNIP. O sistema foi concebido para atender às necessidades de uma instituição de ensino que necessita gerenciar turmas, alunos, professores, aulas e atividades de forma integrada e eficiente.

A linguagem C foi selecionada para os módulos de persistência devido à sua eficiência no gerenciamento de memória e manipulação de arquivos, aspectos fundamentais para compreender o funcionamento de sistemas próximos ao hardware. Por sua vez, Python foi escolhido para a interface do usuário pela sua simplicidade, ampla disponibilidade de bibliotecas gráficas (Tkinter) e facilidade de integração com outros componentes do sistema.

O desenvolvimento seguiu a metodologia ágil Scrum, com organização do trabalho em sprints, definição de backlog de produto e realização de reuniões de planejamento e revisão. Esta abordagem permitiu maior flexibilidade no desenvolvimento, entregas incrementais de funcionalidades e adaptação rápida a mudanças de requisitos. A equipe foi dividida em papéis específicos, com desenvolvedores focados em backend (C), frontend (Python), análise de sistemas (diagramas UML), infraestrutura de rede e documentação.

A estrutura deste trabalho está organizada da seguinte forma: os capítulos 2 a 9 apresentam a fundamentação teórica das oito disciplinas integradas ao projeto (Programação Estruturada em C, Engenharia de Software Ágil, Estrutura de Dados em Python, Análise e Projeto de Sistemas, Redes de Computadores e Sistemas Distribuídos, Educação Ambiental, Inteligência Artificial, e Pesquisa, Tecnologia e Inovação); o capítulo 10 detalha o desenvolvimento prático do sistema, incluindo diagramas, trechos de código-fonte e evidências de funcionamento; e o capítulo 11 apresenta as conclusões do projeto e reflexões sobre o aprendizado adquirido.

Este projeto representa a aplicação prática e integrada de conhecimentos teóricos adquiridos ao longo do semestre, demonstrando a capacidade de desenvolver uma solução tecnológica completa, desde a concepção e planejamento até a implementação e validação. Através deste trabalho, busca-se não apenas atender aos requisitos acadêmicos estabelecidos, mas também contribuir com uma solução real que pode ser adaptada e implementada em contextos educacionais diversos, promovendo a modernização da gestão acadêmica e o uso sustentável de recursos tecnológicos.

OBJETIVO GERAL

Projetar e implementar um sistema acadêmico integrado que permita o gerenciamento completo do ambiente educacional, substituindo controles descentralizados e eliminando o uso de papel através de relatórios digitais.

Objetivos Específicos

- Aplicar metodologias ágeis de engenharia de software para organização e gerenciamento do projeto
- Desenvolver módulos críticos em linguagem C estruturada para compreender sistemas de baixo nível e realizar persistência de dados em arquivos
- Implementar interface gráfica em Python com algoritmos eficientes de gerenciamento, ordenação e geração de relatórios
- Desenvolver sistema de autenticação com três níveis de acesso (administrador, professor e aluno)
- Aplicar recursos de Inteligência Artificial para recomendações e análises
- Documentar o sistema através de diagramas UML e modelagem de rede

- Incorporar práticas de sustentabilidade através da digitalização de processos.

CONTEXTUALIZAÇÃO DO CASO

As instituições de ensino enfrentam desafios crescentes na gestão integrada de turmas, alunos, aulas e atividades, especialmente quando utilizam controles descentralizados em planilhas, e-mails e aplicativos de mensagens. Essa fragmentação dificulta o acompanhamento acadêmico, gera retrabalho administrativo e aumenta o consumo de papel, impactando a sustentabilidade das operações. Diante desse cenário, identificou-se a necessidade de um sistema acadêmico colaborativo que centralize os processos administrativos e pedagógicos em uma única plataforma digital.

O projeto propôs o desenvolvimento de uma aplicação com módulos integrados para cadastro de alunos e turmas, registro de aulas e atividades, controle de acesso por perfil de usuário (administrador, professor e aluno) e suporte à comunicação em rede local. Além disso, foram incorporados recursos de inteligência artificial para sugerir atividades complementares, promovendo um aprendizado mais personalizado. O sistema também buscou contribuir para práticas sustentáveis, substituindo documentos físicos por relatórios digitais e eliminando o uso de papel em sala de aula. Dessa forma, a solução apresentada visa modernizar a gestão acadêmica, aumentar a eficiência dos processos educacionais e alinhar-se aos princípios de inovação e responsabilidade ambiental.

2. PROGRAMAÇÃO ESTRUTURADA EM C

A camada de programação estruturada do sistema acadêmico foi concebida para demonstrar o domínio dos fundamentos da Linguagem C em um contexto de gestão educacional. O projeto organiza as funcionalidades em módulos independentes, cada um responsável por um domínio (alunos, turmas, aulas, atividades e usuários), preservando a arquitetura sequencial definida pela disciplina. A interface textual conduz o usuário por menus hierárquicos, garantindo fluxo claro de navegação e alinhamento com o paradigma estruturado.

A captura de dados segue um protocolo de validação contínua, em que entradas obrigatórias são solicitadas até que apresentem formato válido. Essa abordagem evidencia o

uso disciplinado de estruturas de decisão (condicionais) e assegura que regras de consistência sejam respeitadas antes de qualquer operação sobre os registros. As funcionalidades são encapsuladas em funções bem definidas, chamadas a partir dos menus, reforçando a decomposição funcional típica da programação estruturada em C.

Os menus principais e secundários utilizam laços de repetição para manter o sistema em execução até que o usuário opte por encerrar o módulo. As opções escolhidas são avaliadas por estruturas de seleção que direcionam o fluxo para a função apropriada (cadastro, listagem, atualização ou exclusão). Esse ciclo garante atendimento ao requisito de repetição e mantém o comportamento previsível mesmo diante de entradas inválidas, pois o programa retorna ao menu até que a ação seja concluída com sucesso.

A persistência dos dados ocorre por meio de arquivos no formato CSV, abstraída em um módulo específico encarregado de salvar e carregar registros. Cada entidade do sistema é serializada de acordo com sua estrutura de dados e o mesmo mecanismo é utilizado para todos os domínios, o que demonstra a separação entre lógica de negócio e camada de armazenamento. Esse desenho cumpre os requisitos de leitura e escrita em arquivos e prepara o projeto para substituições futuras, como a adoção de um banco de dados relacional.

Todas as informações manipuladas pelo sistema são descritas por estruturas (`structs`), que definem campos, tamanhos máximos e indicadores de status (ativo/inativo). O uso de ponteiros permite buscar registros e operá-los diretamente na memória, sem duplicação desnecessária, atendendo ao requisito desejável de passagem por referência. Embora o projeto atual utilize arrays estáticos, sua divisão em camadas favorece a introdução de alocação dinâmica e integração com bases externas em etapas subsequentes, mantendo a coerência com os objetivos do PIM.

3. ENGENHARIA DE SOFTWARE ÁGIL

Para o gerenciamento e controle do desenvolvimento do Sistema Acadêmico Colaborativo, a equipe adotou a metodologia ágil Scrum, conforme solicitado como requisito da disciplina. Essa abordagem possibilitou uma organização flexível e iterativa do projeto, favorecendo a adaptação contínua às necessidades identificadas durante o desenvolvimento e a integração dos conhecimentos das oito disciplinas envolvidas.

O ponto de partida foi o Levantamento de Requisitos, que originou o Product Backlog, contendo a lista priorizada das funcionalidades e atividades do sistema. Os Requisitos Funcionais (RF) foram definidos para contemplar as principais ações dos três perfis de usuário (Administrador, Professor e Aluno):

Requisitos funcionais:

- Gerenciar alunos, turmas, atividades e usuários;
- Entrar no sistema utilizando usuário e senha;
- Consultar turmas, atividades e notas;
- Realizar upload de arquivos;

Requisitos não-funcionais:

- O sistema deve obrigatoriamente possuir diferentes níveis de acesso , implementados como Administrador, Professor e Aluno.
- O tempo de resposta do sistema deve ser otimizado, com metas de desempenho abaixo de 3 segundos para operações críticas.
- A interface deve ser intuitiva para facilitar o uso pelos diferentes perfis.
- O sistema deve ser desenvolvido para atender às diretrizes da LGPD (Lei nº 13.709/2018).

O controle e acompanhamento do projeto foram realizados por meio de Sprints semanais, nas quais a equipe conduziu reuniões de sincronização para revisar o progresso, priorizar tarefas e remover impedimentos. Essa metodologia resultou em entregas incrementais, permitindo validações contínuas, maior visibilidade do progresso e melhor alinhamento entre os integrantes do grupo.

4. ESTRUTURA DE DADOS EM PYTHON

No projeto do Sistema Acadêmico Colaborativo, a linguagem **Python** foi adotada como principal tecnologia para o desenvolvimento da aplicação cliente (*frontend*) e para a orquestração da lógica de interação com o usuário. Enquanto os módulos em **C** tratam da persistência de dados em baixo nível, o Python é responsável pela camada de apresentação, exibindo as informações de forma acessível e capturando as entradas do usuário de maneira intuitiva.

Para atender ao requisito de criação de interfaces gráficas eficientes, o projeto utilizou a biblioteca nativa **Tkinter**, por sua compatibilidade, leveza e facilidade de integração. Com ela, foram construídas todas as telas do sistema, desde o formulário de login até os painéis de gerenciamento de alunos, notas e atividades. A interface foi projetada com foco na usabilidade, adaptando-se ao perfil de acesso de cada usuário (Administrador, Professor ou Aluno).

O desenvolvimento foi versionado por meio de um repositório no **GitHub**, permitindo o trabalho colaborativo entre os desenvolvedores *back-end* e *front-end*. O uso do sistema de controle de versões garantiu organização, rastreabilidade e integração contínua das funcionalidades, resultando em uma estrutura coesa e padronizada do código-fonte.

O cumprimento do requisito de utilização de **estruturas de decisão e repetição em Python** foi essencial para o funcionamento da aplicação. As estruturas de decisão são empregadas no sistema de autenticação, validando campos obrigatórios e direcionando o usuário conforme seu perfil. Após um login bem-sucedido, o sistema identifica o tipo de usuário e carrega o respectivo menu de funcionalidades. Já as estruturas de repetição são aplicadas na renderização dinâmica de dados, como a listagem de alunos, turmas e resultados de relatórios, promovendo eficiência e interatividade na navegação do sistema.

5. ANÁLISE E PROJETO DE SISTEMAS

A disciplina de Análise e Projeto de Sistemas foi essencial para estruturar e documentar o Sistema Acadêmico Colaborativo. A **Metodologia UML (Unified Modeling Language)** foi adotada para **estabelecer a documentação dos requisitos** definidos para o projeto. Esta abordagem permitiu decompor a complexidade do sistema, visualizar a interação entre seus componentes e garantir que a lógica de negócios fosse compreendida pela equipe antes da codificação.

A primeira etapa consistiu na Especificação de Requisitos Funcionais e Não-Funcionais, que foram detalhados no capítulo anterior. Com base neles, iniciou-se a modelagem UML para caracterizar a lógica do sistema.

Para a documentação do sistema, foram elaborados os seguintes diagramas:

1. Diagrama de Casos de Uso: Este diagrama foi o primeiro artefato desenvolvido para representar as funcionalidades do sistema sob a perspectiva dos usuários. Ele identifica os quatro atores principais: Usuário (genérico, para o login), ADM (Administrador), Professor e Aluno. O diagrama detalha as ações (casos de uso) que cada ator pode realizar, como "Gerir usuário" (restrito ao ADM) , "Lançar nota" (restrito ao Professor) , "Consultar nota" (disponível ao Aluno) , e "Login no sistema" (disponível a todos).
2. Diagrama de Classes (Visão Análise): Para modelar a estrutura estática e os dados do sistema, foi criado um Diagrama de Classes. Este artefato define as entidades centrais, seus atributos e as operações que podem realizar. Por exemplo, a classe "Aluno" foi definida com os atributos RA, Nome, Email e Status . As operações associadas a esta classe, como "Novo aluno" e "Buscar aluno" , foram mapeadas, servindo como base para a implementação das structs em C e das funções CRUD.
3. Diagramas de Sequência: Para detalhar o comportamento dinâmico e justificar as escolhas técnicas, foram desenvolvidos diagramas de sequência para as operações mais importantes. Esses diagramas (chamados de "Operação" no documento de análise) mostram o fluxo passo a passo da interação entre as classes e os atores. Por exemplo, os diagramas de "Formulário de Cadastro" e "Buscar Aluno" detalham etapas como "Validar dados" e "Salvar cadastro".

A utilização da UML permitiu que a equipe demonstrasse o **entendimento dos conceitos**, criando uma documentação clara que serviu como ponte entre os requisitos teóricos da disciplina e a implementação prática do software.

6. REDES DE COMPUTADORES E SISTEMAS DISTRIBUÍDOS

A necessidade de permitir o **acesso simultâneo de múltiplos usuários** ao sistema acadêmico, conforme requisito não funcional, exigiu a adoção da arquitetura de **Sistemas Distribuídos** no modelo **Cliente-Servidor**.

O desenvolvimento dessa arquitetura começou pela definição dos papéis na rede, o que constituiu a **Configuração das máquinas que comporão a rede**.

1. **Desenvolvimento do Servidor (Módulo C):** O Servidor foi implementado como o ponto central de controle e gestão de dados. Uma máquina foi designada com um **endereço IP estático**, garantindo que os clientes pudessem sempre localizá-la. A aplicação Server-Side foi desenvolvida em Python (complementando o código C) utilizando *sockets TCP* para **escutar** na porta 8080 (exemplo). O desenvolvimento focou em implementar a lógica de *multithreading* para garantir que, ao receber uma nova conexão de um cliente (Professor ou Aluno), o servidor pudesse manter as conexões ativas e processar as requisições de forma simultânea, resolvendo o requisito de **concorrência**.
2. **Desenvolvimento dos Clientes (Módulos Python):** A aplicação cliente (UI em Python/Tkinter) foi desenvolvida para estabelecer uma **conexão via socket** com o Servidor. Para que a **Relação justificada dos componentes principais de conexão da rede** fosse validada, o desenvolvimento do cliente incluiu um módulo de conexão que recebe o IP do servidor e tenta estabelecer o *handshake* TCP.
3. **Desenvolvimento da Centralização de Dados:** Foi desenvolvida uma lógica de comunicação onde **somente o Servidor C tem permissão de leitura e escrita** nos arquivos de dados (`.csv` ou `.txt`). Se um cliente quisesse cadastrar um novo aluno, ele enviava a *requisição* ao Servidor, que executava a rotina de escrita e retornava a *confirmação*. Este processo protege contra a corrupção de dados que poderia ocorrer se múltiplos clientes tentassem escrever no mesmo arquivo simultaneamente.

A etapa de documentação do desenvolvimento de redes incluiu a criação do **Layout (planta baixa) da distribuição dos equipamentos** (apresentado no Capítulo 10). Este diagrama foi desenvolvido para simular a disposição de *múltiplas estações de trabalho* (Clientes) conectadas por um *Switch* ao Servidor C, validando o cenário de rede local (LAN).

A escolha de utilizar **Sockets TCP** e a centralização do acesso a arquivos no Servidor C foram as **propostas de ações** desenvolvidas para garantir que o sistema não apenas funcionasse em rede, mas também cumprisse o requisito crucial de estabilidade e segurança no **acesso simultâneo** exigido pelo projeto.

7. EDUCAÇÃO AMBIENTAL

A integração da Educação Ambiental no Sistema Acadêmico Colaborativo está alinhada ao papel dos sistemas computacionais na promoção de uma sociedade mais sustentável, onde a TI pode ser uma ferramenta para a redução do impacto ambiental. Neste projeto, a contribuição não é apenas teórica, mas se manifesta através de funcionalidades implementadas que promovem a redução de recursos físicos.

O desenvolvimento do sistema priorizou duas **ações principais** voltadas à Educação Ambiental e à comunidade escolar:

Eliminação do Uso de Papel (Digitalização de Processos):

- Esta foi a ação central de sustentabilidade. O sistema foi desenvolvido para substituir integralmente o uso de formulários impressos e diários de classe em papel.
- O módulo de registro de aulas e lançamento de conteúdo (Diário Eletrônico) foi desenvolvido em Python, permitindo que os professores insiram dados diretamente na interface gráfica.
- A funcionalidade de **"Exportar Relatório Digital"** foi implementada, permitindo que a secretaria ou o professor gere relatórios de notas e frequência (em formato CSV ou PDF) diretamente do sistema, ao invés de usar a impressão em massa. Isso atende ao objetivo de **eliminar o uso de papel** nos processos acadêmicos, reduzindo o consumo de recursos naturais e a geração de resíduos.

Essas ações, em conjunto, demonstram como o desenvolvimento de software pode ser uma ferramenta ativa no esforço de sustentabilidade, cumprindo o requisito de **caracterizar pelo menos duas ações voltadas à Educação Ambiental da comunidade**. A implementação dessas funcionalidades é o cerne da contribuição da disciplina para o projeto.

8. INTELIGÊNCIA ARTIFICIAL

A Inteligência Artificial (IA) foi integrada ao projeto em dois níveis: na otimização do próprio processo de desenvolvimento (IA Generativa como Ferramenta) e na implementação de uma funcionalidade do Sistema Acadêmico (IA no Produto Final).

A utilização da IA visa a melhoria da eficiência do projeto, tanto em sua gestão quanto em seu desempenho.

Para acelerar o desenvolvimento, a equipe utilizou diversos recursos e estratégias de IA generativa, o que resultou em um ganho significativo de produtividade na codificação e na documentação:

- **Documentação e Análise:** Ferramentas como Gemini e ChatGPT foram utilizadas no processo de documentação. Elas auxiliaram na caracterização teórica dos capítulos (ex: estruturação da Engenharia de Software e da Educação Ambiental), na sugestão de referências bibliográficas e na revisão gramatical. O uso dessas ferramentas ajudou a garantir a conformidade com as normas ABNT e a clareza do texto.
- **Codificação e Otimização:** Ferramentas de Code Generation como Claude AI e Codex (integrada em ambientes como GitHub Copilot) foram cruciais para o desenvolvimento do código-fonte. Elas auxiliaram na criação de trechos de código estruturados na linguagem C (ex: funções para manipulação de arquivos) e na implementação da lógica de sockets em Python. Tais ferramentas atuaram como um "*pair programmer*", sugerindo refatorações, identificando bugs e otimizando algoritmos de busca e ordenação, especialmente na parte de manipulação de dados em Python.
- Estratégias para Utilização: A equipe adotou a estratégia de utilizar a IA para prototipagem rápida e para a resolução de impedimentos (blockers). Em vez de buscar exaustivamente em fóruns, as ferramentas de IA eram consultadas para

sugestões de sintaxe e melhores práticas para a integração C/Python, permitindo que a equipe mantivesse o ritmo dos Sprints.

9. PESQUISA, TECNOLOGIA E INOVAÇÃO

A Pesquisa, Tecnologia e Inovação (PT&I) constituem o motor do desenvolvimento de sistemas no contexto atual. Conforme a literatura, a inovação em TI não é apenas a criação de algo novo, mas a adoção de soluções que geram valor, melhoram a eficiência e resolvem problemas emergentes.

O desenvolvimento do Sistema Acadêmico Colaborativo está inserido no **contexto atual da PT&I**, focando em duas questões emergentes contemporâneas: a **digitalização de processos** (Agenda ESG e Sustentabilidade) e a **adoção de Inteligência Artificial** para tomada de decisão.

9.1. Adoção de Tecnologias Inovadoras no Projeto

O projeto demonstrou a adoção de tecnologias inovadoras ao integrar múltiplas plataformas e recursos avançados:

- **Arquitetura Distribuída (Inovação em Infraestrutura):** A escolha pelo modelo **Cliente-Servidor (C e Python)**, comunicando-se via *sockets TCP*, representa uma inovação em um projeto acadêmico de PIM, que geralmente utiliza o modelo monolítico de arquivo local. Essa decisão atende à demanda de mercado por **escalabilidade horizontal** e demonstra a capacidade do sistema de operar em um ambiente de produção com acessos simultâneos.
- **Integração de Linguagens de Alto e Baixo Nível:** A comunicação bem-sucedida entre o **módulo Server-Side em C** (para alta performance em persistência de dados) e o **módulo Client-Side em Python** (para desenvolvimento rápido de UI/UX) é uma solução inovadora de integração. Esta abordagem permitiu que a equipe aproveitasse o melhor de cada linguagem, otimizando o desempenho geral do software.
- **Inteligência Artificial (IA) Aplicada:** A inclusão do módulo de **Recomendação Didática**, que utiliza técnicas básicas de *Machine Learning* para classificar o risco acadêmico dos alunos, é o principal elemento de inovação. Essa funcionalidade

transforma o sistema de um simples registrador de dados para uma **plataforma proativa**, alinhando-se à tendência global de *Educação 4.0*.

Essas iniciativas garantem que o desenvolvimento de sistemas informatizados não seja um fim em si mesmo, mas um processo contínuo de inovação baseado nas necessidades reais dos usuários.

10. DESENVOLVIMENTO DO PROJETO

O desenvolvimento do Sistema Acadêmico Colaborativo foi realizado em Sprints semanais, utilizando a metodologia Scrum. Cada módulo implementado foi o resultado direto da aplicação das propostas teóricas apresentadas nos Capítulos 2 a 9, sempre buscando justificar a solução pela viabilidade tecnológica e eficiência operacional.

10.1. Engenharia de Software Ágil (Gestão e Processo)

A adoção do **Scrum** para o desenvolvimento e controle do projeto foi uma escolha estratégica, justificada pela sua viabilidade em gerenciar projetos multidisciplinares com entregas rápidas e adaptabilidade. O processo foi dividido nas seguintes etapas:

10.1.1. Definição do Backlog e Sprints: O primeiro passo foi o **Levantamento de Requisitos**, que resultou na criação do *Product Backlog*. Cada requisito funcional e não funcional foi convertido em **Histórias de Usuário (User Stories)** e agrupado em **Epics**.

O desenvolvimento foi organizado em **Sprints de uma semana**, garantindo que o ciclo de planejamento, execução e revisão fosse rápido e constante. O controle do trabalho foi feito através de um *board* no Jira.

10.1.2. Processos de Desenvolvimento: Para garantir a qualidade, foram definidos os seguintes processos, essenciais para o Scrum:

- **Weekly Scrums (Reuniões Semanais):** Encontros de 15 minutos realizados para sincronizar o time, inspecionar o progresso e identificar impedimentos ("bloqueios").

- **Revisão da Sprint:** Ao final de cada ciclo, o código e as funcionalidades desenvolvidas (ex: Módulo CRUD em C) eram demonstradas e validadas, garantindo que o software funcional fosse entregue em incrementos.
- **Artefatos de Análise:** A disciplina exigiu a produção de artefatos de análise, que foram criados no início do projeto e atualizados a cada Sprint, servindo como a **documentação essencial do sistema** (conforme detalhado na Seção 10.3).

O processo de desenvolvimento ágil permitiu que o projeto, complexo devido à integração de oito disciplinas, fosse concluído com a máxima eficiência e com a documentação sempre alinhada ao código-fonte mais recente.

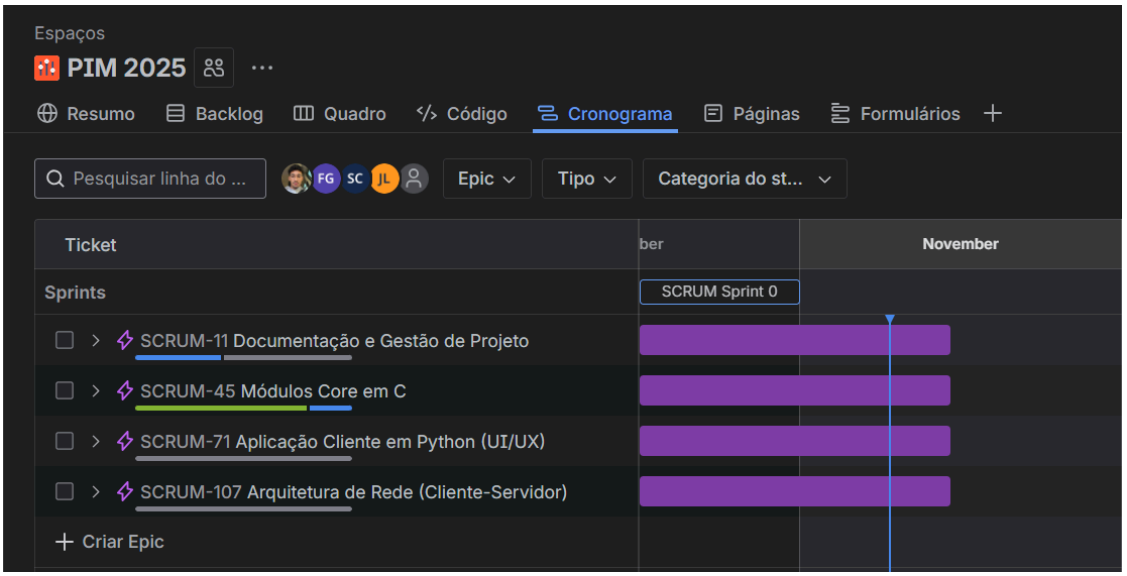


Figura 1: Board do JIRA na aba Cronograma, separado por EPICS de cada categoria do projeto
Fonte: Jira, pimunit.atlassian.net/jira/software/projects/SCRUM/boards/1/timeline

10.2. Análise e Projeto de Sistemas (Modelagem e Documentação)

A fase de Análise e Projeto de Sistemas foi crucial para a **caracterização da lógica** do sistema antes do início da codificação. Foi adotada a **Metodologia UML (Unified Modeling Language)** para estruturar a documentação e estabelecer os requisitos de forma visual, garantindo a rastreabilidade das funcionalidades e a justificativa de escolhas técnicas.

10.2.1. Levantamento e Especificação de Requisitos

O processo se iniciou com a formalização da **Especificação de Requisitos Funcionais (RF)** e **Não Funcionais (RNF)**. Estes requisitos serviram como a base para a criação do *Product Backlog* (Seção 10.1) e para a modelagem UML.

- RF: Foram detalhados casos como Gerir Usuário (RF 02), Lançar e Consultar Nota (RF 03) e Gestão de Atividades (RF 04), definindo o que o sistema deveria executar.
- RNF: Foram definidos requisitos de qualidade como Níveis de Acesso (RNF 01), Desempenho (RNF 02) e a restrição de Conformidade com a LGPD (RNF 04), definindo como o sistema deveria se comportar.

10.2.2. Elaboração dos Diagramas UML

A modelagem UML foi fundamental para estabelecer a arquitetura do Sistema Acadêmico Colaborativo antes da implementação. Utilizamos três tipos principais de diagramas para documentar diferentes aspectos do sistema: casos de uso (funcionalidades), classes (estrutura de dados) e sequência (comportamento dinâmico).

10.2.2.1. Diagrama de Casos de Uso

O Diagrama de Casos de Uso foi o primeiro artefato desenvolvido, pois estabelece o escopo funcional do sistema sob a perspectiva dos usuários. Identificamos quatro atores principais:

- **Usuário (genérico)**: Representa qualquer pessoa que acessa o sistema para realizar login
- **Administrador (ADM)**: Possui permissões totais no sistema
- **Professor**: Gerencia turmas, aulas e atividades
- **Aluno**: Consulta informações acadêmicas

Casos de Uso Principais

UC01 - Login no Sistema

- **Ator**: Todos os usuários
- **Descrição**: Autenticação através de login e senha
- **Pré-condição**: Usuário deve estar cadastrado e ativo

- **Fluxo Principal**:
    1. Sistema exibe tela de login
    2. Usuário informa login e senha
    3. Sistema valida credenciais
    4. Sistema identifica o tipo de usuário (ADMIN/PROFESSOR/ALUNO)
    5. Sistema redireciona para o menu apropriado

UC02 - Gerir Usuários (Exclusivo: Administrador)

- **Ator**: Administrador
- **Descrição**: Cadastrar, listar, alterar e desativar usuários do sistema
- **Inclui**: Validar login único, validar senha

UC03 - Gerir Alunos (Administrador e Professor)

- **Ator**: Administrador, Professor
- **Descrição**: Cadastrar e gerenciar dados de alunos
- **Inclui**: Gerar RA automático, validar email

UC04 - Gerir Turmas (Administrador e Professor)

- **Ator**: Administrador, Professor
- **Descrição**: Criar turmas e matricular alunos
- **Inclui**: Associar aluno à turma, listar alunos da turma

UC05 - Registrar Aula (Exclusivo: Professor)

- **Ator**: Professor
- **Descrição**: Lançar conteúdo ministrado no diário eletrônico
- **Pré-condição**: Professor deve estar vinculado à turma
- **Inclui**: Validar data no formato DD/MM/AAAA

UC06 - Gerenciar Atividades (Professor)

- **Ator**: Professor
- **Descrição**: Cadastrar atividades e anexar arquivos
- **Inclui**: Upload de documento

UC07 - Consultar Notas e Frequência (Aluno e Professor)

- **Ator**: Aluno, Professor
- **Descrição**: Visualizar desempenho acadêmico
- **Extensão**: Gerar relatório digital

UC08 - Gerar Relatórios (Todos os perfis)

- **Ator**: Administrador, Professor, Aluno
- **Descrição**: Exportar informações em formato digital
- **Objetivo**: Substituir impressões em papel (requisito de sustentabilidade)

Relacionamentos entre Casos de Uso

- **Include**: UC02 (Gerir Usuários) <<include>> Validar Login Único
- **Include**: UC05 (Registrar Aula) <<include>> Validar Data
- **Extend**: UC07 (Consultar Notas) <<extend>> Gerar Relatório Digital

O diagrama completo demonstra o controle de acesso hierárquico: administradores têm acesso total, professores gerenciam aspectos pedagógicos, e alunos consultam suas informações.

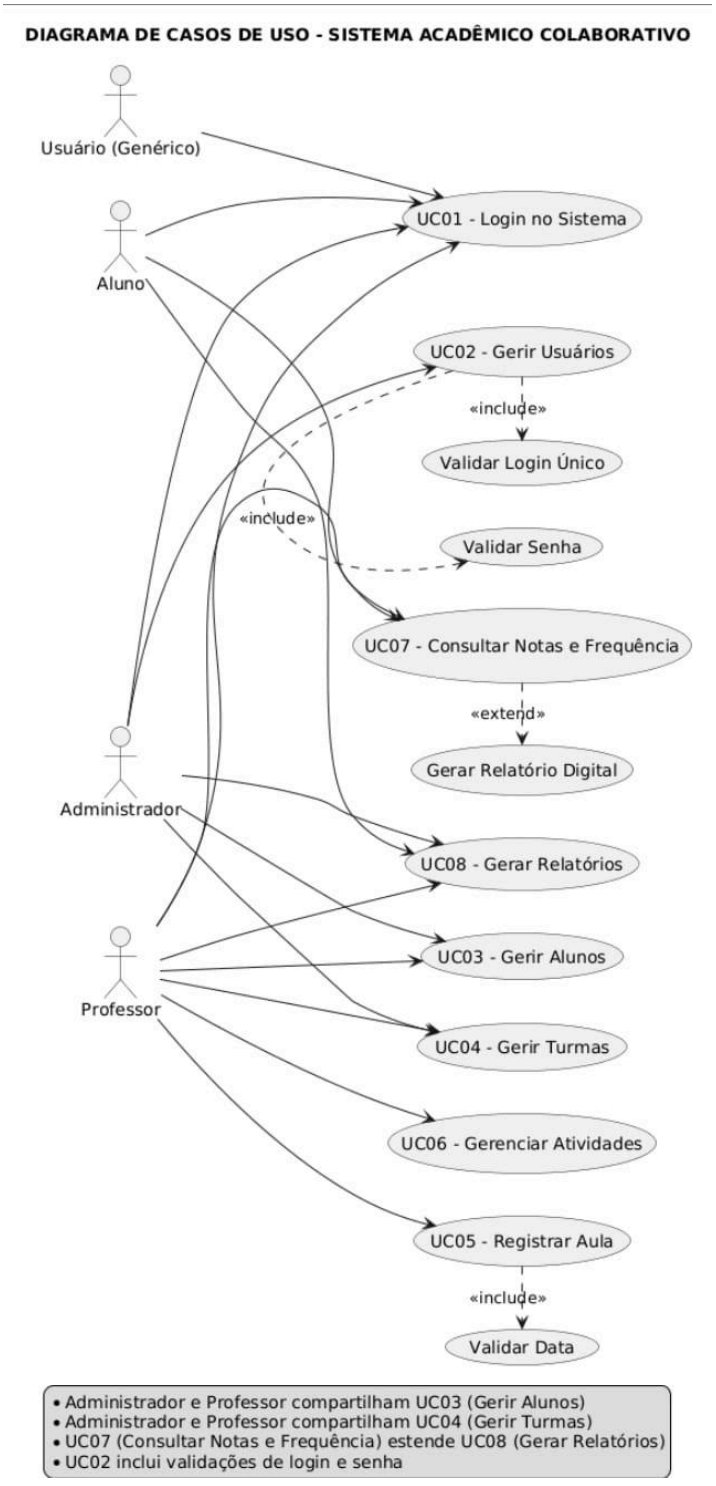


Figura 1: Diagrama de casos de uso

Fonte: Astah

10.2.2.2. Diagrama de Classes (Visão de Análise)

O Diagrama de Classes define a estrutura estática dos dados e suas relações. Cada classe representa uma entidade fundamental do domínio acadêmico:

Classes Principais

Classe: Aluno

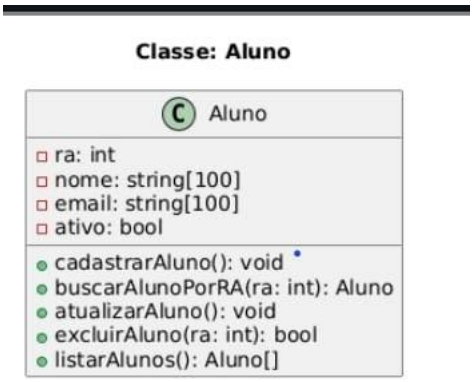


Figura 2: Elaboração da Classe Aluno
Fonte: Astah

Justificativa Técnica: A estrutura `Aluno` em C foi implementada usando `typedef struct`, com campos de tamanho fixo para compatibilidade com arquivos CSV. O campo `ativo` permite soft delete, mantendo histórico para relatórios.

Classe: Turma

Figura 3: Elaboração da Classe Turma
Fonte: Visual Studio Code do Projeto

Classe: Aula

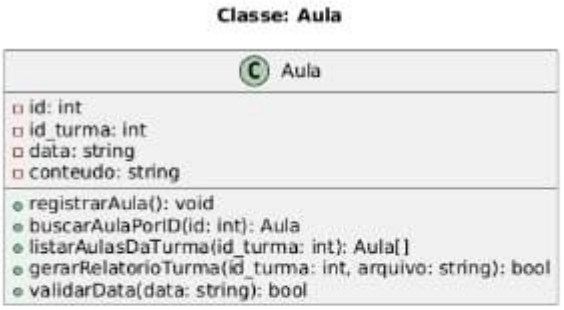


Figura 4: Elaboração da Classe Aula

Fonte: Astah

Nota de Implementação: A função `validarData()` verifica formato, posição das barras e intervalos válidos (dia 1-31, mês 1-12, ano 1900-2100), cumprindo o requisito de estruturas de decisão em C.

Classe: Atividade

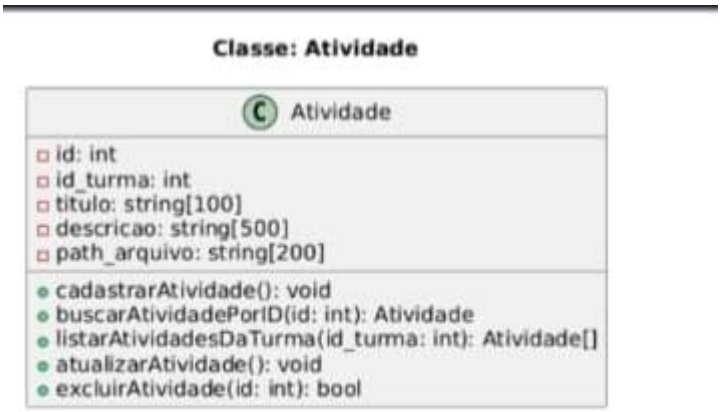


Figura 5: Elaboração da Classe Atividade

Fonte: Astah

Classe: Usuário

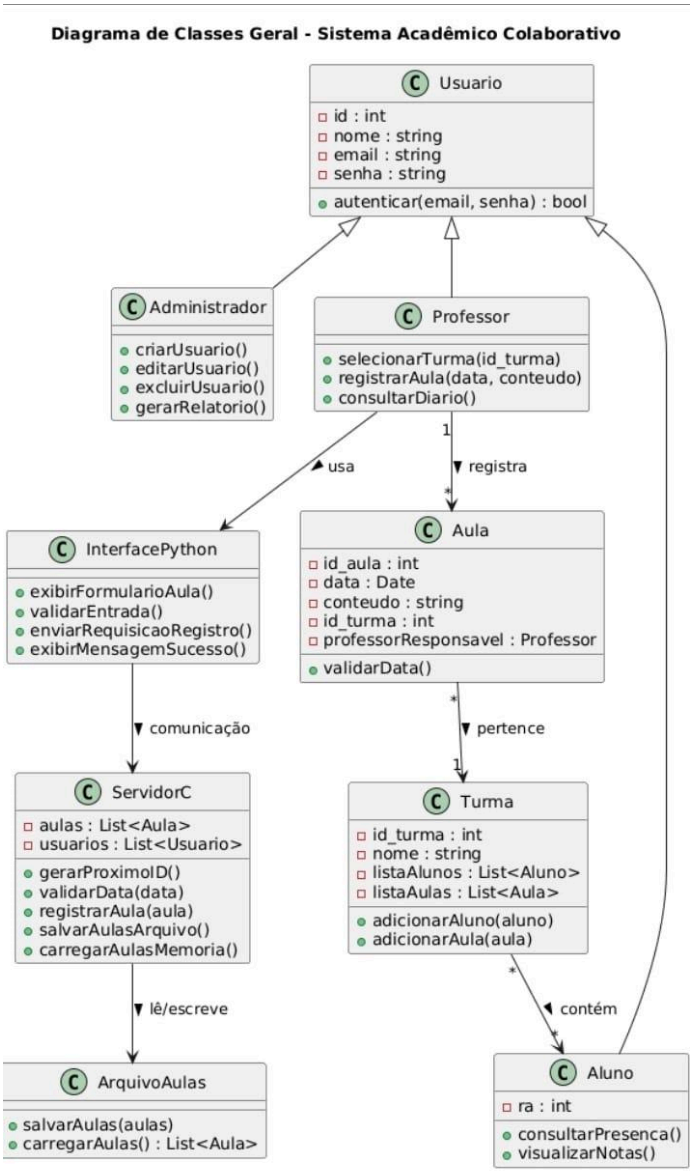


Figura 6: Elaboração da Classe Usuario

Fonte: Astah

Classe Associativa: AlunoTurma

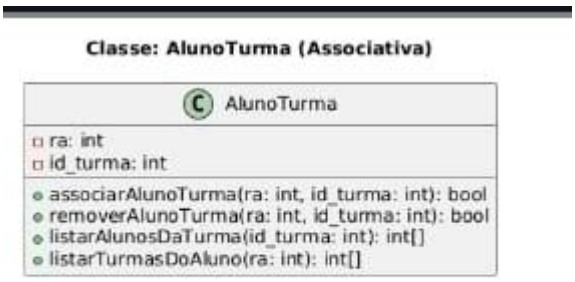


Figura 7: Elaboração da Classe Associativa AlunoTurma

Fonte: Astah

Relacionamentos

1. **Turma** ←→ **Aluno**: Relacionamento N:N (muitos para muitos)
    - Implementado através da classe associativa `AlunoTurma`
    - Um aluno pode estar matriculado em várias turmas
    - Uma turma possui vários alunos
2. **Turma** → **Aula**: Relacionamento 1:N (um para muitos)
    - Uma turma possui múltiplas aulas
    - Cada aula pertence a uma única turma
    - Chave estrangeira: `id_turma` na classe Aula
3. **Turma** → **Atividade**: Relacionamento 1:N (um para muitos)
    - Uma turma possui múltiplas atividades
    - Cada atividade pertence a uma única turma
    - Chave estrangeira: `id_turma` na classe Atividade

Mapeamento para Structs em C

As classes UML foram diretamente mapeadas para `structs` em C no arquivo `structs.h`. Exemplo:

```c
typedef struct {
    int ra;
    char nome[MAX_NOME];
    char email[MAX_NOME];
    int ativo;
} Aluno;
```

Figura 8: Mapeamento para Structs em C

Fonte: Visual Studio Code do Projeto

10.2.2.3. Diagramas de Sequência

Os Diagramas de Sequência documentam o comportamento dinâmico do sistema, mostrando a troca de mensagens entre objetos ao longo do tempo. Desenvolvemos diagramas para as operações mais críticas:

Diagrama de Sequência: Login no Sistema

Participantes:

- Usuário
- InterfacePython (Cliente)
- ServidorC
- ArquivoUsuarios

Fluxo de Mensagens:

1. **Usuário** → **InterfacePython**: `preencherFormulario(login, senha)`
2. **InterfacePython** → **InterfacePython**: `validarCamposObrigatorios()`
    - Estrutura de decisão: verifica se login e senha não estão vazios
3. **InterfacePython** → **ServidorC**: `enviarRequisicaoAutenticacao(login, senha)`
    - Comunicação via socket TCP na porta 8080
4. **ServidorC** → **ArquivoUsuarios**: `carregarUsuariosMemoria()`
    - Leitura do arquivo `data/usuarios.csv`

    - Estrutura de repetição: percorre cada linha do arquivo
5. **ServidorC** → **ServidorC**: `buscarUsuarioPorLogin(login)`
    - Estrutura de repetição: percorre array de usuários
    - Retorna ponteiro para o usuário encontrado (uso de ponteiros)
6. **ServidorC** → **ServidorC**: `verificarSenha(login, senha)`
    - Estrutura de decisão: compara senha informada com senha armazenada
7. **ServidorC** → **ServidorC**: `verificarUsuarioAtivo()`
    - Estrutura de decisão: valida se campo `ativo == 1`
8. **ServidorC** → **ServidorC**: `criarSessao(usuario)`
    - Preenche struct Sessao com dados do usuário
9. **ServidorC** → **InterfacePython**: `retornarSucesso(sessao)`
    - Envia resposta com dados da sessão via socket
10. **InterfacePython** → **InterfacePython**: `redirecionarMenuPorPerfil(tipo)`
    - Estrutura de decisão: switch/case por tipo de usuário
11. **InterfacePython** → **Usuário**: `exibirMenuPrincipal()`

Casos Alternativos:

- **[Credenciais inválidas]**: Após passo 6, se senha incorreta:
    - ServidorC → InterfacePython: `retornarErro("Senha incorreta")`
    - InterfacePython → Usuário: `exibirMensagemErro()`
- **[Usuário inativo]**: Após passo 7, se `ativo == 0`:
    - ServidorC → InterfacePython: `retornarErro("Usuário desativado")`

Justificativa Arquitetural: Este diagrama valida a arquitetura Cliente-Servidor, mostrando que o InterfacePython (cliente) não acessa arquivos diretamente. Toda persistência é centralizada no ServidorC, garantindo consistência em ambiente de rede.

Diagrama de Sequência: Cadastro de Aluno

Participantes:

- Professor/Administrador
- InterfacePython
- ServidorC

- ArquivoAlunos

Fluxo de Mensagens:

1. **Professor** → **InterfacePython**: `abrirFormularioCadastro()`
2. **InterfacePython** → **ServidorC**: `solicitarProximoRA()`
3. **ServidorC** → **ArquivoAlunos**: `carregarAlunosMemoria()`
4. **ServidorC** → **ServidorC**: `gerarRaDisponivel()`
   - Estrutura de repetição: percorre alunos para encontrar maior RA
   - Operação: `return maiorRA + 1`
5. **ServidorC** → **InterfacePython**: `retornarProximoRA(novoRA)`
6. **InterfacePython** → **InterfacePython**: `preencherCampoRA(novoRA)`
7. **Professor** → **InterfacePython**: `preencherDados(nome, email)`
8. **InterfacePython** → **InterfacePython**: `validarEmail(email)`
   - Estrutura de decisão: verifica formato do email
9. **InterfacePython** → **ServidorC**: `enviarRequisicaoCadastro(aluno)`
10. **ServidorC** → **ArquivoAlunos**: `carregarAlunosMemoria()`
11. **ServidorC** → **ServidorC**: `verificarRaDuplicado(ra)`
    - Estrutura de repetição + decisão: percorre alunos verificando RA
12. **ServidorC** → **ServidorC**: `adicionarAlunoArray(aluno)`
    - Cópia de struct por valor: `alunos[total_alunos] = *aluno`
13. **ServidorC** → **ArquivoAlunos**: `salvarAlunosArquivo()`
    - Estrutura de repetição: percorre array escrevendo no CSV
    - Operação de arquivo: `fprintf(arquivo, "%d,%s,%s,%d\n", ...)`
14. **ServidorC** → **InterfacePython**: `retornarSucesso()`
15. **InterfacePython** → **Professor**: `exibirMensagemSucesso("Aluno cadastrado!")`

Casos Alternativos:

- **[RA duplicado]**: Após passo 11:
  - ServidorC → InterfacePython: `retornarErro("RA já cadastrado")`
- **[Limite atingido]**: Se `total_alunos >= MAX_ALUNOS`:

- ServidorC → InterfacePython: `retornarErro("Limite de alunos atingido")`

Demonstração de Requisitos:

- **Uso de ponteiros**: Passo 12 utiliza `*aluno` para desreferenciar ponteiro
- **Estruturas de repetição**: Passos 4, 11 e 13
- **Estruturas de decisão**: Passos 8 e 11
- **Manipulação de arquivos**: Passos 3, 10 e 13

Diagrama de Sequência: Registro de Aula (Diário Eletrônico)

Participantes:

- Professor
- InterfacePython
- ServidorC
- ArquivoAulas

Fluxo de Mensagens:

1. **Professor** → **InterfacePython**: `selecionarTurma(id_turma)`
2. **Professor** → **InterfacePython**: `abrirFormularioAula()`
3. **InterfacePython** → **ServidorC**: `solicitarProximoIDAula()`
4. **ServidorC** → **ArquivoAulas**: `carregarAulasMemoria()`
5. **ServidorC** → **ServidorC**: `gerarProximoIDAula()`
6. **ServidorC** → **InterfacePython**: `retornarProximoID(novoID)`
7. **Professor** → **InterfacePython**: `preencherDados(data, conteudo)`
8. **InterfacePython** → **InterfacePython**: `validarData(data)`
   - Chama função local que replica validação do servidor
9. **InterfacePython** → **ServidorC**: `enviarRequisicaoRegistroAula(aula)`
10. **ServidorC** → **ServidorC**: `validarData(aula.data)`
    - **Implementação detalhada**:

ServidorC → **ArquivoAulas**: `carregarAulasMemoria()`

11. **ServidorC** → **ServidorC**: `verificarIDDuplicado(id)`
12. **ServidorC** → **ServidorC**: `adicionarAulaArray(aula)`
13. **ServidorC** → **ArquivoAulas**: `salvarAulasArquivo()`
14. **ServidorC** → **InterfacePython**: `retornarSucesso()`
15. **InterfacePython** → **Professor**: `exibirMensagemSucesso("Aula registrada no diário!")`

Nota de Sustentabilidade: O diário eletrônico elimina o uso de cadernetas de papel, atendendo ao requisito de Educação Ambiental. Professores registram aulas digitalmente e podem gerar relatórios em PDF quando necessário.

10.2.3. Validação dos Diagramas

Os diagramas UML foram validados através de:

1. **Revisão por Pares**: Cada diagrama foi revisado por outros membros da equipe para garantir consistência e completude.
2. **Rastreabilidade**: Todos os casos de uso foram mapeados para classes e operações correspondentes no diagrama de classes.
3. **Implementabilidade**: Os diagramas de sequência foram testados através de protótipos de código antes da implementação final.
4. **Cobertura de Requisitos**:
    - RF01-RF04: Cobertos pelos casos de uso
    - RNF01 (Níveis de acesso): Demonstrado no UC01 (Login)
    - RNF02 (Desempenho): Considerado na escolha de arrays estáticos
    - RNF04 (LGPD): Refletido na estrutura de Usuario com campo `ativo`

A documentação UML serviu como **contrato** entre a fase de análise e a fase de implementação, garantindo que todos os desenvolvedores tivessem uma visão unificada da arquitetura do sistema antes de escrever a primeira linha de código.

10.3 PROGRAMAÇÃO ESTRUTURADA EM C

A camada de programação estruturada do sistema acadêmico foi desenvolvida para demonstrar o domínio dos fundamentos da Linguagem C aplicados ao contexto de gestão educacional. O projeto organiza as funcionalidades em módulos independentes, cada um responsável por um domínio específico (alunos, turmas, aulas, atividades e usuários), preservando a arquitetura sequencial e modular característica do paradigma estruturado.

10.2.1. Arquitetura Modular do Sistema

O sistema foi estruturado seguindo o princípio da separação de responsabilidades, onde cada módulo possui um conjunto bem definido de operações CRUD (Create, Read, Update, Delete). Esta organização facilita a manutenção e permite que diferentes desenvolvedores trabalhem em módulos distintos simultaneamente.

A estrutura de diretórios do projeto em C organiza-se da seguinte forma:

```
c_modules/
├── structs.h              # Definições de estruturas de dados
├── file_manager.h/.c      # Gerenciamento de arquivos CSV
├── aluno_manager.h/.c     # Operações com alunos
├── turma_manager.h/.c     # Operações com turmas
├── aula_manager.h/.c      # Operações com aulas (diário eletrônico)
├── atividade_manager.h/.c # Operações com atividades
├── usuario_manager.h/.c   # Operações com usuários
├── auth_manager.h/.c      # Autenticação e autorização
├── main.c                 # Interface CLI interativa
└── main_test.c            # Bateria de testes automatizados
```

Figura 9: Definição das estruturas dos módulos em C do projeto
Fonte: Visual Studio Code do Projeto

10.2.2. Definição das Estruturas de Dados

O arquivo `structs.h` centraliza todas as definições de tipos de dados utilizados no sistema. Esta abordagem garante consistência e facilita alterações estruturais futuras. As estruturas foram projetadas considerando os requisitos de persistência em arquivos CSV e a integridade referencial entre entidades.

Estrutura Aluno

A estrutura Aluno representa os dados fundamentais de um estudante, utilizando o RA (Registro Acadêmico) como identificador único, campos para nome completo e email institucional, além de um flag de status ativo/inativo para implementar soft delete:

```c
typedef struct {
    int ra;                     // Registro Acadêmico (identificador único)
    char nome[MAX_NOME];        // Nome completo
    char email[MAX_NOME];       // Email institucional
    int ativo;                  // 1 = ativo, 0 = inativo
} Aluno;
```

Figura 10: Definição da estrutura Aluno
Fonte: Visual Studio Code do Projeto

O campo `ativo` implementa o padrão soft delete, permitindo desativar registros sem removê-los fisicamente do sistema. Esta decisão de design preserva o histórico acadêmico e facilita a geração de relatórios retroativos.

Estrutura Turma

A estrutura Turma armazena informações sobre as turmas oferecidas pela instituição, incluindo um identificador único, nome da turma, professor responsável e o período letivo (ano e semestre):

```c
typedef struct {
    int id;                     // ID único da turma
    char nome[MAX_TURMA_NOME]; // Nome da turma (ex: "ADS-2A")
    char professor[MAX_NOME];   // Nome do professor
    int ano;                    // Ano letivo
    int semestre;               // Semestre (1 ou 2)
} Turma;
```

Figura 11: Definição da estrutura Turma
Fonte: Visual Studio Code do Projeto

Estrutura Aula (Diário Eletrônico)

A estrutura Aula implementa o conceito de diário eletrônico, registrando cada aula ministrada com sua data no formato DD/MM/AAAA e o conteúdo abordado, vinculada à turma correspondente através de chave estrangeira:

```c
typedef struct {
    int id;                    // ID único da aula
    int id_turma;              // FK: ID da turma
    char data[11];             // Data da aula (formato: DD/MM/AAAA)
    char conteudo[MAX_CONTEUDO]; // Conteúdo ministrado
} Aula;
```

Figura 12: Definição da estrutura Turma
Fonte: Visual Studio Code do Projeto

Esta estrutura materializa o requisito de sustentabilidade do projeto, eliminando o uso de cadernetas físicas e permitindo o registro digital de todas as aulas.

Estrutura AlunoTurma (Relacionamento N:N)

A estrutura AlunoTurma implementa o relacionamento muitos-para-muitos entre alunos e turmas, permitindo que um aluno esteja matriculado em múltiplas turmas e que uma turma contenha vários alunos:

```c
typedef struct {
    int ra;                    // FK: RA do aluno
    int id_turma;              // FK: ID da turma
} AlunoTurma;
```

Figura 13: Definição da estrutura AlunoTurma
Fonte: Visual Studio Code do Projeto

Estrutura Atividade

A estrutura Atividade representa trabalhos e exercícios propostos aos alunos, incluindo título, descrição detalhada e caminho opcional para arquivo anexo:

```c
typedef struct {
    int id;                    // ID único da atividade
    int id_turma;              // FK: ID da turma
    char titulo[MAX_NOME];     // Título da atividade
    char descricao[MAX_CONTEUDO]; // Descrição
    char path_arquivo[MAX_PATH]; // Caminho do arquivo (se houver)
} Atividade;
```

Figura 13: Definição da estrutura Atividade

Fonte: Visual Studio Code do Projeto

Estrutura Usuario

A estrutura Usuario gerencia o controle de acesso ao sistema, armazenando credenciais em texto plano (conforme especificação do projeto acadêmico), tipo de perfil e status de ativação:

```c
typedef struct {
    int id;                    // ID único do usuário
    char login[MAX_LOGIN];     // Nome de login (único)
    char senha[MAX_SENHA];     // Senha em texto plano
    char tipo[20];             // Tipo: "ADMIN", "PROFESSOR", "ALUNO"
    int ativo;                 // 1 = ativo, 0 = inativo
} Usuario;
```

Figura 14: Definição da estrutura Usuario

Fonte: Visual Studio Code do Projeto

10.3.2. Gerenciamento de Arquivos CSV

O módulo `file_manager` implementa as operações de entrada e saída em arquivos CSV, abstraindo a complexidade da serialização e desserialização de estruturas. Esta camada de abstração permite alterar o mecanismo de persistência no futuro sem impactar os módulos de negócio.

Função de Salvamento de Dados

A função salvarDados implementa a serialização genérica de arrays de estruturas para o formato CSV, utilizando estruturas de decisão (switch) para identificar o tipo de dado e estruturas de repetição (for) para percorrer todos os registros.

Função de Carregamento de Dados

A função carregarDados realiza o processo inverso, lendo linhas do arquivo CSV e preenchendo as estruturas correspondentes usando sscanf para parsing de cada campo.

Esta implementação demonstra o uso disciplinado de estruturas de repetição (`while`) e decisão (`switch`), cumprindo requisitos obrigatórios da disciplina.

10.3.3. Módulo de Gerenciamento de Alunos

O módulo `aluno_manager` implementa todas as operações relacionadas ao cadastro e manutenção de alunos. A estratégia adotada mantém os dados em memória durante a execução do programa e sincroniza com o arquivo CSV apenas quando necessário.

Arrays Estáticos em Memória

O módulo mantém um array estático de alunos em memória e um contador de total de registros, garantindo acesso rápido aos dados durante a execução:

```
static Aluno alunos[MAX_ALUNOS];
static int total_alunos = 0;
```

Funções de Sincronização

As funções carregarAlunosMemoria e salvarAlunosArquivo gerenciam a sincronização entre a memória e o arquivo CSV, sendo chamadas no início e fim de cada operação CRUD para garantir consistência.

Operação de Cadastro (Create)

A função cadastrarAluno implementa validações de integridade, verificando unicidade do RA antes da inserção e garantindo que o limite de capacidade não seja ultrapassado.

Operação de Busca (Read)

A função buscarAlunoPorRA demonstra o uso de ponteiros (requisito desejável), retornando o endereço da estrutura encontrada ou NULL caso o aluno não exista.

O retorno de ponteiro permite que outras funções modifiquem diretamente a estrutura sem necessidade de cópias adicionais, otimizando o uso de memória.

Operação de Listagem

A função listarAlunos copia os registros para um buffer externo, demonstrando passagem por referência e uso de estruturas de repetição.

Operação de Atualização (Update)

A função atualizarAluno localiza o registro pelo RA e substitui todos os campos da estrutura, mantendo a integridade da chave primária:

Operação de Exclusão (Delete - Soft Delete)

A função excluirAluno implementa soft delete, marcando o registro como inativo ao invés de removê-lo fisicamente, preservando histórico para relatórios e auditorias:

10.3.4. Módulo de Gerenciamento de Aulas (Diário Eletrônico)

O módulo `aula_manager` implementa o diário eletrônico, substituindo as cadernetas físicas tradicionais. Este módulo é fundamental para o requisito de sustentabilidade do projeto.

Validação de Data

A função validarData implementa verificações rigorosas do formato DD/MM/AAAA, utilizando múltiplas estruturas de decisão para validar tamanho, posição das barras, caracteres numéricos e intervalos válidos.

Registro de Aula

A função registrarAula valida a data fornecida, verifica unicidade do ID e garante que o limite de aulas não seja excedido antes de persistir o registro.

Geração de Relatório de Diário

A função gerarRelatorioTurma cria um arquivo de texto com todas as aulas de uma turma específica, demonstrando o requisito de sustentabilidade ao substituir relatórios impressos por versões digitais, este relatório elimina a necessidade de impressões físicas, contribuindo diretamente para o objetivo de sustentabilidade do projeto.

10.3.5. Módulo de Gerenciamento de Turmas e Matrículas

O módulo `turma_manager` implementa tanto o CRUD de turmas quanto o gerenciamento do relacionamento N:N entre alunos e turmas.

Gerenciamento de Matrículas

O módulo mantém dois arrays estáticos: um para turmas e outro para as matrículas (relacionamento AlunoTurma), permitindo que um aluno esteja matriculado em múltiplas turmas.

Associação Aluno-Turma

A função associarAlunoTurma implementa a matrícula de um aluno em uma turma, verificando duplicidade antes da inserção.

Listagem de Alunos por Turma

A função listarAlunosDaTurma filtra as matrículas pelo ID da turma e retorna um array com os RAs dos alunos matriculados.

10.3.6. Módulo de Autenticação e Autorização

O módulo `auth_manager` implementa o sistema de login e controle de acesso com três níveis de permissões: ADMIN, PROFESSOR e ALUNO.

Estrutura de Sessão

A estrutura Sessao mantém os dados do usuário autenticado durante a execução do programa, incluindo tipo de perfil e timestamp para controle de timeout.

Função de Autenticação

A função autenticar realiza validação de credenciais, verifica status ativo e preenche a estrutura de sessão em caso de sucesso.

Sistema de Permissões

A função temPermissao implementa controle de acesso baseado em regras, onde ADMIN possui acesso total e demais perfis têm permissões específicas.

10.3.7. Interface de Linha de Comando (CLI)

O arquivo `main.c` implementa uma interface textual completa para operação manual do sistema, demonstrando a integração de todos os módulos desenvolvidos. Esta interface foi concebida para permitir testes e operações administrativas diretamente através do terminal, sem depender da interface gráfica Python.

Menu Principal

A função `main` implementa o menu principal com navegação hierárquica, utilizando estruturas de repetição (`do-while`) e decisão (`switch`) para controlar o fluxo do programa:

```c
int main(void) {
    int opcao;

    do {
        printf("\n========================================\n");
        printf("       SISTEMA ACADEMICO - MODO MANUAL       \n");
        printf("========================================\n");
        printf("1. Gerenciar alunos\n");
        printf("2. Gerenciar turmas\n");
        printf("3. Gerenciar aulas\n");
        printf("4. Gerenciar atividades\n");
        printf("5. Gerenciar usuarios\n");
        printf("0. Sair\n");

        opcao = lerInteiroObrigatorio("Escolha uma opcao: ");

        switch (opcao) {
            case 1: menuAlunos(); break;
            case 2: menuTurmas(); break;
            case 3: menuAulas(); break;
            case 4: menuAtividades(); break;
            case 5: menuUsuarios(); break;
            case 0: printf("Encerrando o sistema. Ate logo!\n"); break;
            default: printf("Opcao invalida. Tente novamente.\n");
        }
    } while (opcao != 0);

    return 0;
}
```

Figura 15: Inicialização da Main.c
Fonte: Visual Studio Code do Projeto

Esta estrutura demonstra o cumprimento dos requisitos de **estrutura de repetição** e **estrutura de decisão**, mantendo o programa em execução até que o usuário opte por sair.

Submenu de Alunos

O menu de gerenciamento de alunos oferece operações CRUD completas, com interface textual intuitiva:

```c
static void menuAlunos(void) {
    int opcao;

    do {
        printf("\n=== ALUNO MANAGER ===\n");
        printf("1. Cadastrar aluno\n");
        printf("2. Relatorio de alunos\n");
        printf("3. Alterar aluno\n");
        printf("4. Remover aluno (soft delete)\n");
        printf("0. Voltar ao menu principal\n");

        opcao = lerInteiroObrigatorio("Escolha uma opcao: ");

        switch (opcao) {
            case 1: cadastrarAlunoManual(); break;
            case 2: listarAlunosManual(); break;
            case 3: atualizarAlunoManual(); break;
            case 4: excluirAlunoManual(); break;
            case 0: break;
            default: printf("Opcao invalida.\n");
        }
    } while (opcao != 0);
}
```

Figura 16: Submenu de alunos

Fonte: Visual Studio Code do Projeto

10.4 Interface Gráfica em Python (Frontend)

Gerenciamento de Dados

• Biblioteca:

Consiste na importação de módulos essenciais da linguagem Python, tais como csv, json, datetime, pathlib, tkinter e re. Esses módulos possibilitam a leitura e gravação de arquivos, a manipulação de datas, a exibição de mensagens na interface e a validação de entradas textuais utilizadas em todo o sistema.

• Inicialização:

Responsável pela criação automática do diretório denominado “data”, destinado ao armazenamento dos arquivos do sistema, garantindo assim sua existência antes da gravação de qualquer informação.

• Salvamento em CSV:

Implementa a gravação de listas de dados em arquivos no formato CSV, utilizando um cabeçalho previamente definido. A função retorna uma indicação de sucesso ou mensagem de erro, conforme o resultado do processo.

• Carregamento de CSV:

Realiza a leitura de arquivos CSV e converte seu conteúdo em uma lista de dicionários. Caso o arquivo não exista, a função retorna uma lista vazia, assegurando o correto funcionamento do sistema.

• Salvamento em JSON:

Efectua a escrita de dados em arquivos no formato JSON, empregando formatação legível (pretty format). Retorna confirmação de sucesso ou mensagem de erro em caso de falha.

• Carregamento de JSON:

Executa a leitura de arquivos JSON e retorna os dados armazenados. Se o arquivo não for encontrado, devolve um objeto vazio. A função também trata eventuais erros de leitura, preservando a integridade dos dados.

```python
import csv
import json
from pathlib import Path
from datetime import datetime
from tkinter import messagebox
import re

# ============ GERENCIAMENTO DE DADOS ============

class DataManager:
    """Classe para salvar/carregar dados em CSV e JSON"""

    def __init__(self):
        self.data_dir = Path(__file__).parent / "data"
        self.data_dir.mkdir(exist_ok=True)

    def salvar_csv(self, arquivo, dados, cabecalho):
        """Salva dados em arquivo CSV"""
        caminho = self.data_dir / arquivo
        try:
            with open(caminho, 'w', newline='', encoding='utf-8') as f:
                writer = csv.DictWriter(f, fieldnames=cabecalho)
                writer.writeheader()
                writer.writerows(dados)
            return True
        except Exception as e:
            print(f"Erro ao salvar CSV: {e}")
            return False

    def carregar_csv(self, arquivo):
        """Carrega dados de arquivo CSV"""
        caminho = self.data_dir / arquivo
        if not caminho.exists():
            return []

        try:
            with open(caminho, 'r', encoding='utf-8') as f:
                reader = csv.DictReader(f)
                return list(reader)
        except Exception as e:
            print(f"Erro ao carregar CSV: {e}")
            return []

    def salvar_json(self, arquivo, dados):
        """Salva dados em arquivo JSON"""
        caminho = self.data_dir / arquivo
        try:
            with open(caminho, 'w', encoding='utf-8') as f:
                json.dump(dados, f, ensure_ascii=False, indent=2)
            return True
        except Exception as e:
            print(f"Erro ao salvar JSON: {e}")
            return False

    def carregar_json(self, arquivo):
        """Carrega dados de arquivo JSON"""
        caminho = self.data_dir / arquivo
        if not caminho.exists():
            return {}

        try:
            with open(caminho, 'r', encoding='utf-8') as f:
                return json.load(f)
        except Exception as e:
            print(f"Erro ao carregar JSON: {e}")
            return {}
```

Figura 1 e 2: Gerenciamento de Dados.

Fonte: Código do Projeto Integrado Multidisciplinar, originalmente desenvolvido no VS Code.

Sistema de Validadores

• Validação de RA:

Responsável por verificar se o Registro Acadêmico (RA) é composto exclusivamente por caracteres numéricos, sem espaços, não estando vazio e contendo, no mínimo, quatro dígitos.

• Validação de e-mail:

Confirma se o endereço de e-mail informado apresenta o caractere “@” e um domínio válido, incluindo a presença de um ponto após o símbolo “@”. Também verifica se o campo não está vazio.

• Validação de nome:

Assegura que o nome fornecido não esteja vazio e possua, pelo menos, três caracteres, garantindo consistência mínima dos dados textuais.

• Validação de data:

Realiza a verificação do padrão de data no formato “DD/MM/AAAA”, certificando-se de que os valores de dia, mês e ano são válidos e correspondem a uma data real, considerando as regras do calendário.

```
        """Valida se nome não está vazio"""
        if not nome or not nome.strip():
            return False, "Nome não pode ser vazio"
if len(nome.strip()) < 3:
            return False, "Nome deve ter pelo menos 3 caracteres"

        return True, ""

    @staticmethod
    def validar_data(data_str):
        """Valida formato DD/MM/AAAA"""
        if not data_str or not data_str.strip():
            return False, "Data não pode ser vazia"

        # Regex para DD/MM/AAAA
        padrao = r'^(\d{2})/(\d{2})/(\d{4})$'
        match = re.match(padrao, data_str.strip())

        if not match:
            return False, "Formato inválido. Use DD/MM/AAAA"

        dia, mes, ano = map(int, match.groups())

        # Validar valores
        if mes < 1 or mes > 12:
            return False, "Mês inválido (1-12)"

        if dia < 1 or dia > 31:
            return False, "Dia inválido (1-31)"

        if ano < 1900 or ano > 2100:
            return False, "Ano inválido"

        # Validar data usando datetime
        try:
```

```
    # Validar data usando datetime
    try:
        datetime(ano, mes, dia)
        return True, ""
    except ValueError:
        return False, "Data inválida"

@staticmethod
def validar_id_numerico(id_str, nome_campo):
    """Valida se ID é numérico"""
    if not id_str or not id_str.strip():
        return False, f"{nome_campo} não pode ser vazio"

    if not id_str.strip().isdigit():
        return False, f"{nome_campo} deve ser numérico"

    return True, ""
```

Figura 3 e 4: Elaboração do Sistema de Validadores.

Fonte: Fonte: Código do Projeto Integrado Multidisciplinar, originalmente desenvolvido no VS Code.

Funções de Mensagens

• Mensagem de sucesso:

Exibe uma janela informativa indicando que a operação foi concluída de forma correta.

(Utiliza o método messagebox.showinfo).

• Mensagem de erro:

Apresenta uma janela notificando que ocorreu uma falha ou erro durante a execução da operação.

(Utiliza o método messagebox.showerror).

• Mensagem de aviso:

Mostra um alerta destinado a chamar a atenção do usuário para uma situação relevante ou potencialmente problemática.

(Utiliza o método messagebox.showwarning).

• Confirmação de ação:

Exibe uma janela solicitando ao usuário a confirmação de determinada operação, oferecendo as opções “Sim” e “Não”. A função retorna True quando a resposta for “Sim” e False quando a resposta for “Não”.

(Utiliza o método messagebox.askyesno).

```
# ============= FUNÇÕES DE MENSAGENS =============

class MensagensSistema:
    """Gerenciamento de mensagens e feedback visual"""

    @staticmethod
    def mensagem_sucesso(titulo, mensagem):
        """Mostra mensagem de sucesso"""
        messagebox.showinfo(titulo, mensagem)

    @staticmethod
    def mensagem_erro(titulo, mensagem):
        """Mostra mensagem de erro"""
        messagebox.showerror(titulo, mensagem)

    @staticmethod
```

Figura 5: Elaboração de Funções de Mensagens.
Fonte: Código do Projeto Integrado Multidisciplinar, originalmente desenvolvido no VS Code.

Lógica de Login

• Inicialização:

Armazena o gerenciador de dados (data_manager), responsável pelo carregamento de arquivos CSV utilizados no processo de autenticação.

• Validação de campos:

Verifica se os campos de login e senha foram devidamente preenchidos. Caso estejam vazios, a função retorna a mensagem: “Preencha usuário e senha!”.

• Carregamento de usuários:

Realiza a leitura do arquivo usuarios.csv, obtendo a lista completa de usuários cadastrados no sistema.

• Verificação de login:

Compara o valor informado no campo de login com os registros existentes, desconsiderando diferenças entre letras maiúsculas e minúsculas.

• Verificação de senha:

Confirma se a senha fornecida corresponde exatamente à senha registrada para o usuário. Caso a senha esteja incorreta, retorna a mensagem: “Senha incorreta!”.

• Verificação de status:

Analisa se o usuário possui status ativo, representado pelo valor “1”. Caso esteja inativo, retorna a mensagem: “Usuário inativo!”.

• Retorno final:

Retorna o objeto correspondente ao usuário quando todas as verificações são concluídas com sucesso. Se nenhuma correspondência válida for encontrada, retorna a mensagem: “Usuário não encontrado!”.

```
        return messagebox.askyesno("Confirmar", mensagem)


# ============ LÓGICA DE LOGIN ============

class LoginLogica:
    """Lógica de autenticação"""

    def __init__(self, data_manager):
        self.data_manager = data_manager

    def fazer_login(self, login, senha):
        """Valida credenciais e retorna usuário ou None"""
        if not login or not senha:
            return None, "Preencha usuário e senha!"

        # Carregar usuários do CSV
        usuarios = self.data_manager.carregar_csv("usuarios.csv")

        # Validar credenciais
        for usuario in usuarios:
            if usuario.get("Login", "").lower() == login.lower():
                if usuario.get("Senha") == senha:
                    if usuario.get("Ativo") == "1":
                        return usuario, None
                    else:
                        return None, "Usuário inativo!"
                else:
                    return None, "Senha incorreta!"

        return None, "Usuário não encontrado!"
```

Figura 6: Elaboração da Logica de Login.

Fonte: Código do Projeto Integrado Multidisciplinar,originalmente desenvolvido no VS Code.

Lógica de Alunos

• Inicialização:

Armazena o gerenciador de dados (data_manager) e cria uma lista interna destinada ao armazenamento dos alunos. Em seguida, realiza automaticamente a chamada do método responsável por carregar os registros a partir do arquivo CSV.

• Carregamento de alunos:

Lê o arquivo alunos.csv e armazena os registros obtidos na lista self.alunos. Ao final do processo, retorna a lista completa contendo todos os alunos carregados.

• Adição de aluno:

Responsável por inserir um novo aluno no sistema, executando previamente todas as validações obrigatórias para garantir a integridade dos dados.

• Validação do RA:

Utiliza o módulo validador para confirmar se o Registro Acadêmico (RA) informado atende aos critérios de validade. Caso seja inválido, a função retorna False juntamente com a mensagem de erro correspondente.

• Validação do nome:

Verifica se o nome do aluno não está vazio e se cumpre os requisitos mínimos estabelecidos. Em caso de falha, retorna False e a respectiva mensagem de erro.

• Validação do e-mail:

Confirma se o endereço de e-mail está em formato adequado e não é vazio. Se houver inconsistência, a função retorna False acompanhada da mensagem explicativa referente ao problema identificado.

```python
# ============ LÓGICA DE ALUNOS ============

class AlunosLogica:
    """Lógica de gerenciamento de alunos"""

    def __init__(self, data_manager):
        self.data_manager = data_manager
        self.alunos = []
        self.carregar_alunos()

    def carregar_alunos(self):
        """Carrega alunos do CSV"""
        self.alunos = self.data_manager.carregar_csv("alunos.csv")
        return self.alunos

    def adicionar_aluno(self, ra, nome, email, ativo):
        """Adiciona novo aluno com validações"""
        # Validações
        valido, msg = Validadores.validar_ra(ra)
        if not valido:
            return False, msg

        valido, msg = Validadores.validar_nome(nome)
        if not valido:
            return False, msg

        valido, msg = Validadores.validar_email(email)
        if not valido:
            return False, msg

        # Verificar RA duplicado
        for aluno in self.alunos:
            if aluno.get("RA") == ra:
                return False, "RA já cadastrado!"
```

```python
                return False, "RA já cadastrado!"

        # Adicionar aluno
        novo_aluno = {
            "RA": ra,
            "Nome": nome,
            "Email": email,
            "Ativo": ativo
        }

        self.alunos.append(novo_aluno)

        # Salvar no CSV
        cabecalho = ["RA", "Nome", "Email", "Ativo"]
        if self.data_manager.salvar_csv("alunos.csv", self.alunos, cabecalho):
            return True, "Aluno cadastrado com sucesso!"
        else:
            return False, "Falha ao salvar aluno!"

    def buscar_aluno(self, ra_busca):
        """Busca aluno por RA"""
        if not ra_busca:
            return [], "Digite um RA para buscar!"

        resultados = [a for a in self.alunos if a.get("RA") == ra_busca]

        if resultados:
            return resultados, None
        else:
            return [], "Aluno não encontrado!"

    def atualizar_aluno(self, ra_antigo, ra_novo, nome, email, ativo):
        """Atualiza dados de um aluno"""
        # Validações
        valido, msg = Validadores.validar_ra(ra_novo)
```

Figura 7 e 8: Elaboração da Lógica de Alunos.

Fonte: Código do Projeto Integrado Multidisciplinar,originalmente desenvolvido no VS Code.

Lógica de Turmas

• Inicialização:

Armazena o gerenciador de dados (data_manager) e cria a lista interna destinada ao armazenamento das turmas. Em seguida, executa automaticamente o método responsável pelo carregamento dos dados provenientes do arquivo CSV.

• Carregamento de turmas:

Realiza a leitura do arquivo turmas.csv e armazena os registros na lista self.turmas. Ao final, retorna a lista completa contendo todas as turmas carregadas.

• Geração de novo ID:

Cria automaticamente um identificador sequencial para novas turmas.

– Caso não existam turmas cadastradas, retorna “1”.

– Caso existam registros, identifica o maior ID numérico presente na lista e adiciona 1 ao seu valor.

• Adição de turma:

Responsável pelo cadastro de uma nova turma no sistema, realizando validações prévias dos campos fornecidos.

• Geração automática de ID:

Se o ID não for informado pelo usuário, o sistema gera automaticamente um novo identificador utilizando o método interno de criação de ID.

• Validação dos campos:

– Verifica se o campo nome foi preenchido;

– Confirma se o campo professor foi informado;

– Garante que o campo ano contenha apenas valores numéricos;

– Checa se o ID informado já existe no sistema, evitando duplicidades.

• Salvamento da nova turma:

Cria um dicionário contendo os dados da turma e o adiciona à lista de registros existentes. Em seguida, salva os dados no arquivo turmas.csv, utilizando o cabeçalho adequado. Retorna uma mensagem de sucesso ou erro de acordo com o resultado da operação.

• Filtragem de turmas por professor:

Permite buscar turmas cujo campo Professor contenha o texto informado, desconsiderando diferenças entre letras maiúsculas e minúsculas.

– Caso nenhum critério de filtro seja fornecido, exibe um aviso ao usuário;

– Se houver correspondências, retorna a lista filtrada;

– Caso contrário, retorna mensagem informando que nenhum resultado foi encontrado.

• Atualização de turma:

Localiza a turma correspondente ao ID antigo e substitui seus dados pelos novos valores fornecidos. Antes da atualização, realiza validações referentes ao nome, professor e ano.

• Salvamento das alterações:

Após a modificação dos dados, o sistema salva novamente todos os registros no arquivo turmas.csv. Ao final, retorna mensagem indicando sucesso ou falha no processo de atualização.

```
# ============= LÓGICA DE TURMAS =============

class TurmasLogica:
    """Lógica de gerenciamento de turmas"""

    def __init__(self, data_manager):
        self.data_manager = data_manager
        self.turmas = []
        self.carregar_turmas()

    def carregar_turmas(self):
        """Carrega turmas do CSV"""
        self.turmas = self.data_manager.carregar_csv("turmas.csv")
        return self.turmas

    def gerar_novo_id(self):
        """Gera novo ID sequencial"""
        if not self.turmas:
            return "1"
        ids = [int(t.get("ID", 0)) for t in self.turmas if t.get("ID", "").isdigit()]
        return str(max(ids) + 1) if ids else "1"

    def adicionar_turma(self, id_turma, nome, professor, ano, semestre):
        """Adiciona nova turma com validações"""
        # Gerar ID se não fornecido
        if not id_turma:
            id_turma = self.gerar_novo_id()

        # Validações
        if not nome:
            return False, "Nome da turma é obrigatório!"

        if not professor:
            return False, "Professor é obrigatório!"
```

Figura 9: Elaboração logica de Turmas.

Fonte: Código do Projeto Integrado Multidisciplinar,originalmente desenvolvido no VS Code.

Lógica de Associação Aluno–Turma (Matrícula)

• Inicialização:

Armazena o gerenciador de dados (data_manager) e cria listas internas destinadas às associações, aos alunos e às turmas. Em seguida, realiza automaticamente o carregamento de todos os dados necessários a partir dos arquivos CSV.

• Carregamento de dados:

Realiza a leitura dos seguintes arquivos:

– aluno_turma.csv: contém as associações (matrículas);

– alunos.csv: reúne a lista de alunos cadastrados;

– turmas.csv: reúne a lista de turmas disponíveis.

Após o carregamento, retorna todas as matrículas registradas.

• Matrícula de aluno:

Executa o processo de vincular um aluno a uma turma específica, mediante validações obrigatórias.

• Validação dos IDs:

Verifica se o RA do aluno e o ID da turma são valores numéricos válidos. Caso qualquer um deles seja inválido, a função retorna imediatamente uma mensagem de erro.

• Verificação da existência do aluno:

Pesquisa o RA informado na lista de alunos cadastrados. Se não houver correspondência, retorna a mensagem: “Aluno não encontrado!”.

• Verificação da existência da turma:

Confirma se o ID especificado corresponde a uma turma existente. Se não for encontrado, retorna a mensagem: “Turma não encontrada!”.

• Verificação de matrícula duplicada:

Garante que o aluno não esteja matriculado mais de uma vez na mesma turma. Caso a associação já exista, retorna mensagem informando duplicidade.

• Salvamento da matrícula:

Insere um novo registro contendo o RA do aluno e o ID da turma no arquivo aluno_turma.csv. Após a tentativa de gravação, retorna mensagem de sucesso ou falha.

• Obtenção de associações com nomes:

Retorna uma lista contendo o RA do aluno, o ID da turma, o nome do aluno e o nome da turma. Para isso, realiza a junção dos dados consultando os registros de alunos e de turmas.

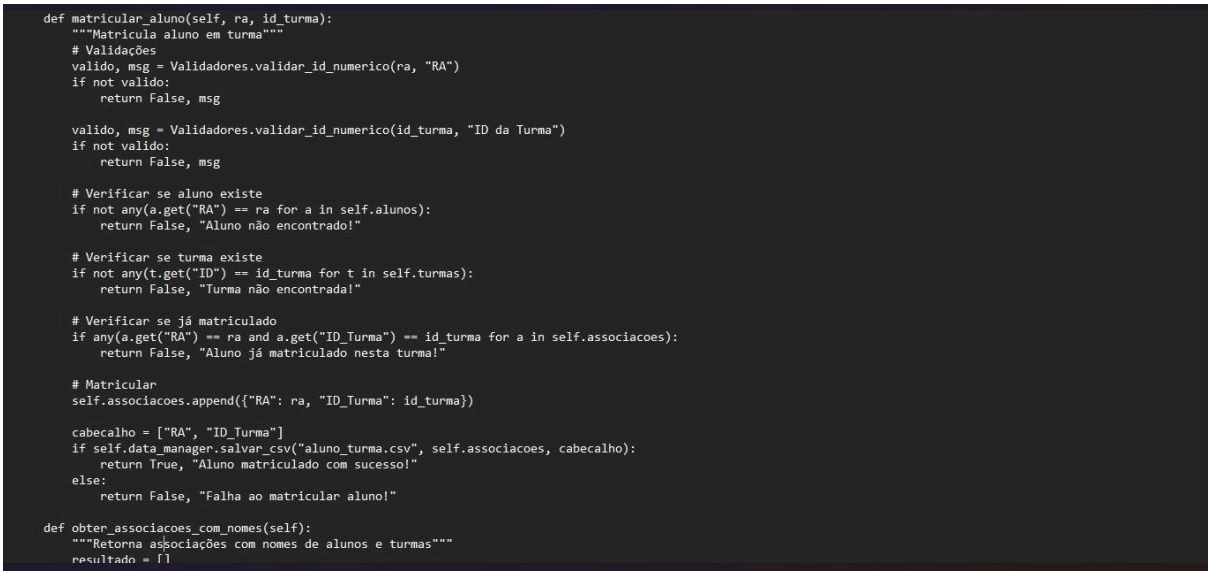
• Filtragem de alunos por turma:

Identifica todas as matrículas relacionadas a uma turma específica e retorna uma lista contendo RA, nome do aluno e nome da turma, de forma estruturada.

• Remoção de matrícula:

Remove a associação existente entre um aluno e uma turma. Após a exclusão, atualiza o arquivo aluno_turma.csv e retorna mensagem indicando o resultado da operação.

```
    def excluir_turma(self, id_turma):
        """Exclui uma turma"""
        self.turmas = [t for t in self.turmas if t.get("ID") != str(id_turma)]

        cabecalho = ["ID", "Nome", "Professor", "Ano", "Semestre"]
        if self.data_manager.salvar_csv("turmas.csv", self.turmas, cabecalho):
            return True, "Turma excluída com sucesso!"
        else:
            return False, "Falha ao excluir turma!"

    def obter_todas_turmas(self):
        """Retorna todas as turmas"""
        return self.turmas


# ============ LÓGICA DE ASSOCIAÇÃO ALUNO-TURMA ============

class AssociacaoLogica:
    """Lógica de matrícula de alunos em turmas"""

    def __init__(self, data_manager):
        self.data_manager = data_manager
        self.associacoes = []
        self.alunos = []
        self.turmas = []
        self.carregar_dados()

    def carregar_dados(self):
        """Carrega todos os dados necessários"""
        self.associacoes = self.data_manager.carregar_csv("aluno_turma.csv")
        self.alunos = self.data_manager.carregar_csv("alunos.csv")
        self.turmas = self.data_manager.carregar_csv("turmas.csv")
        return self.associacoes
```



```
def matricular_aluno(self, ra, id_turma):
    """Matricula aluno em turma"""
    # Validações
    valido, msg = Validadores.validar_id_numerico(ra, "RA")
    if not valido:
        return False, msg

    valido, msg = Validadores.validar_id_numerico(id_turma, "ID da Turma")
    if not valido:
        return False, msg

    # Verificar se aluno existe
    if not any(a.get("RA") == ra for a in self.alunos):
        return False, "Aluno não encontrado!"

    # Verificar se turma existe
    if not any(t.get("ID") == id_turma for t in self.turmas):
        return False, "Turma não encontrada!"

    # Verificar se já matriculado
    if any(a.get("RA") == ra and a.get("ID_Turma") == id_turma for a in self.associacoes):
        return False, "Aluno já matriculado nesta turma!"

    # Matricular
    self.associacoes.append({"RA": ra, "ID_Turma": id_turma})

    cabecalho = ["RA", "ID_Turma"]
    if self.data_manager.salvar_csv("aluno_turma.csv", self.associacoes, cabecalho):
        return True, "Aluno matriculado com sucesso!"
    else:
        return False, "Falha ao matricular aluno!"

def obter_associacoes_com_nomes(self):
    """Retorna associações com nomes de alunos e turmas"""
    resultado = []
```

Figura 11 e 12: Elaboração da Lógica de Associação de Aluno e Turma

Fonte: Visual Studio Code do Projeto

Lógica de Aulas

• Inicialização:

Armazena o gerenciador de dados (data_manager) e cria a lista interna destinada ao armazenamento das aulas. Em seguida, realiza automaticamente o carregamento dos registros presentes no arquivo CSV.

• Carregamento de aulas:

Efetua a leitura do arquivo aulas.csv e preenche a lista self.aulas com todos os registros existentes. Ao final, retorna a lista completa de aulas carregadas.

• Geração de novo ID:

Cria automaticamente um identificador sequencial para cada nova aula.

– Caso não existam registros anteriores, inicia a sequência com o valor “1”;

– Caso contrário, identifica o maior ID numérico registrado e adiciona 1 ao valor encontrado.

• Registro de aula:

Responsável pela inclusão de um novo registro de aula no sistema, realizando previamente todas as validações obrigatórias.

• Validação dos dados:

– Verifica se o ID da turma é numérico;

– Confirma se a data está no formato correto e representa uma data válida;

– Garante que o campo de conteúdo não esteja vazio.

• Salvamento da nova aula:

Cria um dicionário contendo o ID, o identificador da turma, a data e o conteúdo da aula. Em seguida, adiciona o registro à lista interna e salva a atualização no arquivo aulas.csv. Ao final, retorna mensagem indicando sucesso ou falha no processo de registro.

• Filtragem de aulas por turma:

Lista apenas as aulas associadas ao ID de turma informado.

– Se nenhum ID for fornecido, retorna todas as aulas;

– Caso não existam registros correspondentes, retorna mensagem apropriada.

• Filtragem de aulas por período:

Seleciona as aulas registradas dentro de um intervalo de datas definido pelo usuário.

– Valida as datas fornecidas;

– Converte as datas para o formato comparável AAAAMMDD;

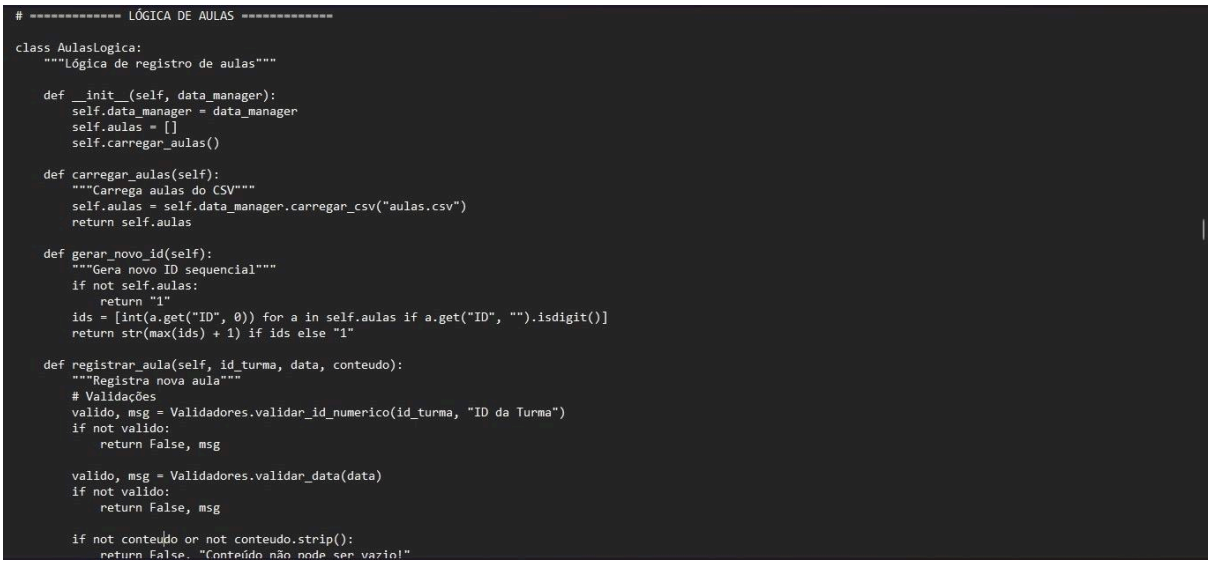
– Retorna todas as aulas compreendidas no período ou mensagem indicando a ausência de registros.

• Exclusão de aula:

Remove um registro específico com base no ID informado. Após a remoção, atualiza o arquivo aulas.csv e retorna mensagem de confirmação da operação.

• Obtenção de todas as aulas:

Retorna a lista completa contendo todos os registros de aulas armazenados no sistema.



```
# ============ LÓGICA DE AULAS ============

class AulasLogica:
    """Lógica de registro de aulas"""

    def __init__(self, data_manager):
        self.data_manager = data_manager
        self.aulas = []
        self.carregar_aulas()

    def carregar_aulas(self):
        """Carrega aulas do CSV"""
        self.aulas = self.data_manager.carregar_csv("aulas.csv")
        return self.aulas

    def gerar_novo_id(self):
        """Gera novo ID sequencial"""
        if not self.aulas:
            return "1"
        ids = [int(a.get("ID", 0)) for a in self.aulas if a.get("ID", "").isdigit()]
        return str(max(ids) + 1) if ids else "1"

    def registrar_aula(self, id_turma, data, conteudo):
        """Registra nova aula"""
        # Validações
        valido, msg = Validadores.validar_id_numerico(id_turma, "ID da Turma")
        if not valido:
            return False, msg

        valido, msg = Validadores.validar_data(data)
        if not valido:
            return False, msg

        if not conteudo or not conteudo.strip():
            return False, "Conteúdo não pode ser vazio!"
```

Figura 13: Elaboração da lógica de aulas

Fonte: Visual Studio Code do Projeto

Sistema de Relatórios

• Inicialização:

Armazena o gerenciador de dados (data_manager), responsável pelo carregamento das informações provenientes dos arquivos necessários para a geração dos relatórios.

Relatório de Alunos Ativos

Lê o arquivo alunos.csv e identifica apenas os alunos cujo campo “Ativo” está definido como “1”. O relatório retorna:

– Total de alunos cadastrados;

– Quantidade de alunos ativos;

– Quantidade de alunos inativos;

– Lista completa contendo exclusivamente os alunos ativos.

Relatório de Turmas

Carrega os registros de turmas e as associações entre alunos e turmas. Para cada turma cadastrada, calcula o número total de alunos matriculados. O relatório retorna, para cada turma:

– ID da turma;

– Nome da turma;

– Professor responsável;

– Quantidade de alunos matriculados.

Relatório de Frequência de Aulas

Lê os arquivos contendo as turmas e as aulas registradas. Para cada turma, contabiliza a quantidade total de aulas cadastradas no sistema. O relatório apresenta:

– ID da turma;

– Nome da turma;

– Quantidade total de aulas registradas.

Exportação de Relatório para CSV

Realiza a geração de um arquivo CSV, salvo no Desktop do usuário.

– Recebe os dados a serem exportados, o nome do arquivo e o cabeçalho das colunas;

– Salva as informações no formato CSV padrão;

– Retorna mensagem indicando sucesso ou relatando eventual falha durante o processo de exportação.

```python
# ============= SISTEMA DE RELATÓRIOS =============

class RelatoriosLogica:
    """Lógica para geração de relatórios"""

    def __init__(self, data_manager):
        self.data_manager = data_manager

    def gerar_relatorio_alunos_ativos(self):
        """Gera relatório de alunos ativos"""
        alunos = self.data_manager.carregar_csv("alunos.csv")
        ativos = [a for a in alunos if a.get("Ativo") == "1"]

        return {
            "total_alunos": len(alunos),
            "alunos_ativos": len(ativos),
            "alunos_inativos": len(alunos) - len(ativos),
            "lista_ativos": ativos
        }

    def gerar_relatorio_turmas(self):
        """Gera relatório de turmas"""
        turmas = self.data_manager.carregar_csv("turmas.csv")
        associacoes = self.data_manager.carregar_csv("aluno_turma.csv")

        resultado = []
        for turma in turmas:
            id_turma = turma.get("ID")
            qtd_alunos = len([a for a in associacoes if a.get("ID_Turma") == id_turma])

            resultado.append({
                "ID": id_turma,
                "Nome": turma.get("Nome"),
                "Professor": turma.get("Professor"),
```

```python
            "Professor": turma.get("Professor"),
            "Quantidade_Alunos": qtd_alunos
        })

    return resultado

def gerar_relatorio_frequencia_aulas(self):
    """Gera relatório de frequência de aulas por turma"""
    aulas = self.data_manager.carregar_csv("aulas.csv")
    turmas = self.data_manager.carregar_csv("turmas.csv")

    resultado = []
    for turma in turmas:
        id_turma = turma.get("ID")
        qtd_aulas = len([a for a in aulas if a.get("ID_Turma") == id_turma])

        resultado.append({
            "ID_Turma": id_turma,
            "Nome_Turma": turma.get("Nome"),
            "Quantidade_Aulas": qtd_aulas
        })

    return resultado

def exportar_relatorio_csv(self, dados, nome_arquivo, cabecalho):
    """Exporta relatório para CSV"""
    try:
        caminho = Path.home() / "Desktop" / nome_arquivo
        with open(caminho, 'w', newline='', encoding='utf-8') as f:
            writer = csv.DictWriter(f, fieldnames=cabecalho)
            writer.writeheader()
            writer.writerows(dados)
        return True, f"Relatório exportado para: {caminho}"
    except Exception as e:
        return False, f"Erro ao exportar: {e}"
```

Figura 14 e 15: Elaboração do Sistema de Relátorio

Fonte: Visual Studio Code do Projeto

Utilitários Auxiliares

1. obter_data_atual()

Função responsável por gerar automaticamente a data corrente do sistema, padronizada no formato DD/MM/AAAA. É utilizada em registros, relatórios, processos de auditoria e logs internos.

2. converter_data_para_datetime()

Realiza a conversão de uma data representada em texto para o formato datetime, permitindo ao sistema executar cálculos, comparações e validações envolvendo datas. Caso a data seja inválida, a função retorna None.

3. formatar_data_exibicao()

Aprimora a apresentação visual de datas para o usuário ao adicionar o dia da semana ao formato original. É utilizada em telas do sistema, relatórios e consultas, oferecendo maior clareza e organização das informações.

4. limpar_string()

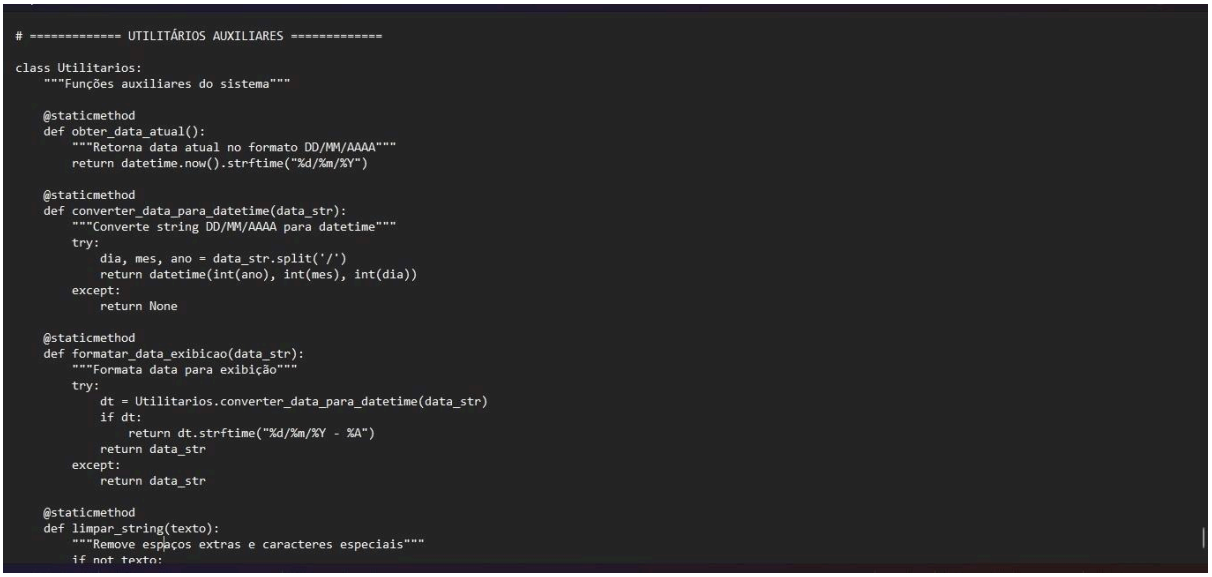
Remove espaços duplicados, caracteres desnecessários e erros comuns de digitação. A função padroniza textos antes do armazenamento, evitando falhas de validação decorrentes de má formatação e contribuindo para a consistência dos dados.

5. validar_data()

Verifica se uma data está escrita no formato DD/MM/AAAA, analisando a validade dos campos de dia, mês e ano. Impede o registro de datas inexistentes e é amplamente utilizada em cadastros, registros de aulas, relatórios e agendas internas.

6. validar_id_numerico()

Garante que campos identificadores (ID) contenham exclusivamente caracteres numéricos. Evita inconsistências ao referenciar alunos, turmas, aulas e associações, contribuindo para a manutenção de um banco de dados organizado e confiável.



```
# ============ UTILITÁRIOS AUXILIARES ============

class Utilitarios:
    """Funções auxiliares do sistema"""

    @staticmethod
    def obter_data_atual():
        """Retorna data atual no formato DD/MM/AAAA"""
        return datetime.now().strftime("%d/%m/%Y")

    @staticmethod
    def converter_data_para_datetime(data_str):
        """Converte string DD/MM/AAAA para datetime"""
        try:
            dia, mes, ano = data_str.split('/')
            return datetime(int(ano), int(mes), int(dia))
        except:
            return None

    @staticmethod
    def formatar_data_exibicao(data_str):
        """Formata data para exibição"""
        try:
            dt = Utilitarios.converter_data_para_datetime(data_str)
            if dt:
                return dt.strftime("%d/%m/%Y - %A")
            return data_str
        except:
            return data_str

    @staticmethod
    def limpar_string(texto):
        """Remove espaços extras e caracteres especiais"""
        if not texto:
```

Figura 16: Elaboração do Utilitários Auxiliares

Fonte: Visual Studio Code do Projeto

Configurações de Cores e Fontes

A estrutura utiliza as bibliotecas tkinter e ttk para criação de interfaces gráficas em Python, além do módulo Path da biblioteca pathlib para manipulação de diretórios e caminhos de arquivos.

A classe Config centraliza todos os parâmetros visuais da aplicação, padronizando elementos de interface e facilitando futuras modificações estéticas. Suas responsabilidades incluem:

• Definição de diretórios:

Gerencia os caminhos utilizados para armazenamento de arquivos e dados da aplicação, garantindo organização e acessibilidade.

• Configuração de paleta de cores:

Define cores principais do sistema, utilizando valores em formato hexadecimal. São contempladas:

– cor primária;

– cor secundária;

– cor de sucesso;

– cor de erro;

– cor de fundo;

– cor de texto.

Essas definições asseguram uma identidade visual coerente em todas as telas.

• Padronização de fontes:

Estabelece fontes padrão para diferentes níveis de informação na interface, determinando tamanho e estilo para:

– títulos;

– subtítulos;

– textos comuns.

A padronização contribui para clareza visual, boa legibilidade e uniformidade do design.

```
# ============= CONFIGURAÇÃO DE CORES E FONTES =============

class Config:
    """Configurações visuais centralizadas"""
    BASE_DIR = Path(__file__).parent
    DATA_DIR = BASE_DIR / "data"

    # Cores do tema
    COR_PRIMARIA = "#2C3E50"
    COR_SECUNDARIA = "#3498DB"
    COR_SUCESSO = "#27AE60"
    COR_ERRO = "#E74C3C"
    COR_FUNDO = "#ECF0F1"
    COR_TEXTO = "#2C3E50"

    # Fontes
    FONTE_TITULO = ("Segoe UI", 16, "bold")
    FONTE_SUBTITULO = ("Segoe UI", 12, "bold")
    FONTE_NORMAL = ("Segoe UI", 10)
    FONTE_PEQUENA = ("Segoe UI", 9)
```

Figura 16: Elaboração da configuração de cores e fontes.

Fonte: Código do Projeto Integrado Multidisciplinar, originalmente desenvolvido no VS Code.

Componentes Reutilizáveis de UI

A classe ComponentesUI reúne métodos estáticos destinados à criação de elementos gráficos reutilizáveis, desenvolvidos com as bibliotecas Tkinter e ttk. Sua finalidade é padronizar a interface, promover consistência visual e simplificar a construção de telas dentro da aplicação. Os principais recursos disponibilizados incluem:

• criar_botao_primario:

Responsável por gerar botões estilizados conforme o tema da aplicação. Permite definir texto, ação associada e integração com o layout, garantindo uniformidade nos elementos de interação.

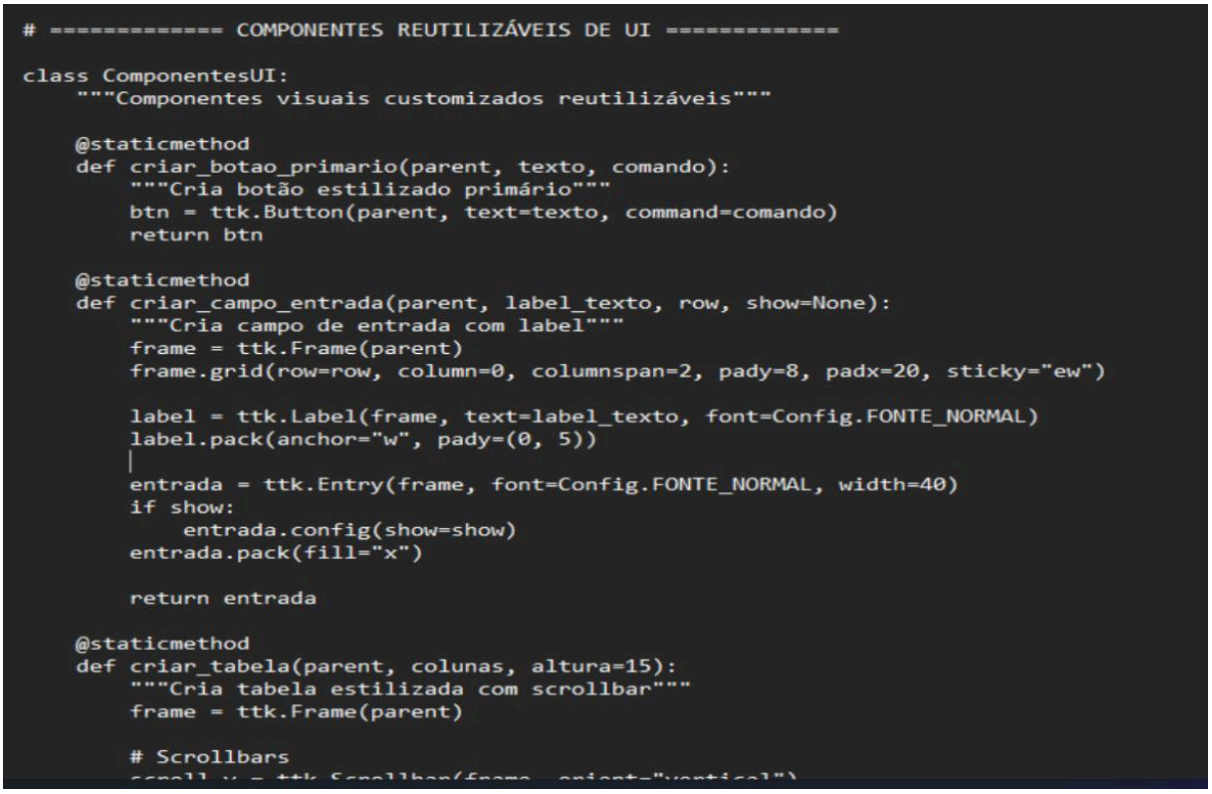
• criar_campo_entrada:

Cria campos de entrada acompanhados de rótulo (label), utilizando posicionamento via grid. Possui suporte para ocultação de caracteres — adequado para senhas — e assegura padronização no estilo e no comportamento dos campos inseridos.

• criar_tabela:

Constrói uma tabela baseada em ttk.Treeview, estruturada com colunas configuráveis, além de barras de rolagem vertical e horizontal integradas. Esse componente facilita a exibição organizada de dados e mantém a identidade visual do sistema.

A utilização dessa classe favorece o reaproveitamento de código, melhora a manutenção do projeto e assegura uma aparência uniforme em todas as interfaces gráficas desenvolvidas.



```
# ============= COMPONENTES REUTILIZÁVEIS DE UI =============

class ComponentesUI:
    """Componentes visuais customizados reutilizáveis"""

    @staticmethod
    def criar_botao_primario(parent, texto, comando):
        """Cria botão estilizado primário"""
        btn = ttk.Button(parent, text=texto, command=comando)
        return btn

    @staticmethod
    def criar_campo_entrada(parent, label_texto, row, show=None):
        """Cria campo de entrada com label"""
        frame = ttk.Frame(parent)
        frame.grid(row=row, column=0, columnspan=2, pady=8, padx=20, sticky="ew")

        label = ttk.Label(frame, text=label_texto, font=Config.FONTE_NORMAL)
        label.pack(anchor="w", pady=(0, 5))

        entrada = ttk.Entry(frame, font=Config.FONTE_NORMAL, width=40)
        if show:
            entrada.config(show=show)
        entrada.pack(fill="x")

        return entrada

    @staticmethod
    def criar_tabela(parent, colunas, altura=15):
        """Cria tabela estilizada com scrollbar"""
        frame = ttk.Frame(parent)

        # Scrollbars
        scroll_y = ttk.Scrollbar(frame, orient="vertical")
```

```
        # Scrollbars
        scroll_y = ttk.Scrollbar(frame, orient="vertical")
        scroll_x = ttk.Scrollbar(frame, orient="horizontal")

        # Treeview
        tree = ttk.Treeview(
            frame,
            columns=colunas,
            show="headings",
            height=altura,
            yscrollcommand=scroll_y.set,
            xscrollcommand=scroll_x.set
        )

        scroll_y.config(command=tree.yview)
        scroll_x.config(command=tree.xview)

        # Definir cabeçalhos
        for col in colunas:
            tree.heading(col, text=col)
            tree.column(col, width=150, anchor="center")

        # Layout
        tree.grid(row=0, column=0, sticky="nsew")
        scroll_y.grid(row=0, column=1, sticky="ns")
        scroll_x.grid(row=1, column=0, sticky="ew")

        frame.grid_rowconfigure(0, weight=1)
        frame.grid_columnconfigure(0, weight=1)

        return frame, tree
```

Figura 17: **Componentes Reutilizáveis de Ui.**

Fonte: Código do Projeto Integrado Multidisciplinar, originalmente desenvolvido no VS Code.

Tela de Login – Layout

A classe TelaLoginLayout é responsável por estruturar a interface gráfica da tela de login da aplicação, organizando seus componentes de forma clara e funcional. Sua construção utiliza o sistema de posicionamento grid, garantindo alinhamento adequado e apresentação consistente. Os principais elementos definidos pela classe incluem:

• Títulos e subtítulos:

Exibem informações introdutórias ao usuário, orientando sobre a finalidade da tela e reforçando a identidade visual da aplicação.

• Campos de entrada:

Disponibiliza campos para inserção de usuário e senha, sendo este último configurado com ocultação de caracteres, assegurando privacidade e conformidade com boas práticas de segurança.

• Botões de ação:

Inclui botões para “Entrar” e “Sair”, devidamente posicionados e estilizados. O botão “Entrar” é associado a um callback externo, permitindo a integração direta com o processo de autenticação do sistema.

• Área de usuários de teste:

Apresenta informações auxiliares para fins de demonstração ou validação do sistema, facilitando o uso em ambientes de desenvolvimento.

Ao encapsular toda a lógica visual da tela, a classe promove reutilização, facilita manutenção e garante uniformidade no fluxo de autenticação da aplicação.

```python
# ============= TELA DE LOGIN - LAYOUT =============

class TelaLoginLayout:
    """Layout da tela de login"""

    def __init__(self, parent, fazer_login_callback):
        self.parent = parent
        self.fazer_login_callback = fazer_login_callback

        self.frame = ttk.Frame(parent, padding=50)
        self.frame.pack(expand=True, fill="both")

        self.criar_interface()

    def criar_interface(self):
        """Cria interface visual da tela de login"""
        # Título
        titulo = ttk.Label(
            self.frame,
            text="Sistema Acadêmico PIM 2025",
            font=Config.FONTE_TITULO
        )
        titulo.grid(row=0, column=0, columnspan=2, pady=(0, 10))

        subtitulo = ttk.Label(
            self.frame,
            text="Escola Lourença - Gerenciamento de Notas",
            font=Config.FONTE_PEQUENA
        )
        subtitulo.grid(row=1, column=0, columnspan=2, pady=(0, 30))

        # Campos de entrada
        self.entrada_login = ComponentesUI.criar_campo_entrada(
            self.frame, "Usuário:", 2
        )
```

```python
    # Botões
    frame_botoes = ttk.Frame(self.frame)
    frame_botoes.grid(row=4, column=0, columnspan=2, pady=30)

    btn_entrar = ComponentesUI.criar_botao_primario(
        frame_botoes, "Entrar", self.fazer_login_callback
    )
    btn_entrar.pack(side="left", padx=5)

    btn_sair = ttk.Button(frame_botoes, text="Sair", command=self.parent.quit)
    btn_sair.pack(side="left", padx=5)

    # Bind Enter para login
    self.entrada_senha.bind("<Return>", lambda e: self.fazer_login_callback())

    # Info de usuários de teste
    info_frame = ttk.LabelFrame(self.frame, text="Usuários de Teste", padding=10)
    info_frame.grid(row=5, column=0, columnspan=2, pady=20)

    ttk.Label(info_frame, text="Admin: admin / admin123", font=Config.FONTE_PEQUENA).pack()
    ttk.Label(info_frame, text="Professor: professor / professor", font=Config.FONTE_PEQUENA).pack()
    ttk.Label(info_frame, text="Aluno: aluno / aluno", font=Config.FONTE_PEQUENA).pack()

def destruir(self):
    """Remove a tela de login"""
    self.frame.destroy()
```

Figura 18: Tela de Login-Layout.

Fonte: Código do Projeto Integrado

Multidisciplinar,originalmente desenvolvido no VS Code.

Menu Principal – Layout

A classe MenuPrincipalLayout é responsável por compor o layout do menu principal da aplicação, fornecendo a estrutura visual exibida ao usuário após a autenticação. Essa classe organiza os elementos de forma clara, permitindo navegação intuitiva entre as funcionalidades do sistema. Seus principais componentes são:

• Cabeçalho informativo:

Apresenta uma mensagem de boas-vindas personalizada, acompanhada da identificação do usuário autenticado e um botão destinado à finalização da sessão (logout). Esse cabeçalho funciona como orientação visual inicial e reforça o contexto do perfil conectado.

• Organização por abas (Notebook):

Utiliza o widget Notebook, que possibilita a criação de múltiplas abas para separar e categorizar funcionalidades da aplicação. As abas são adicionadas dinamicamente por meio de callbacks, permitindo maior flexibilidade, escalabilidade e modularidade da interface.

• Estruturação pós-login:

O layout oferece uma transição organizada após o login, distribuindo as funcionalidades do sistema em seções independentes. Desse modo, facilita a navegação, torna o uso mais intuitivo e permite manutenção visual padronizada.

Ao encapsular toda a lógica de disposição dos elementos, a classe assegura consistência estética, organização funcional e facilidade de expansão do sistema.

```
# ============= MENU PRINCIPAL - LAYOUT =============

class MenuPrincipalLayout:
    """Layout do menu principal"""

    def __init__(self, parent, usuario, voltar_login_callback, criar_aba_callbacks):
        self.parent = parent
        self.usuario = usuario
        self.voltar_login_callback = voltar_login_callback
        self.criar_aba_callbacks = criar_aba_callbacks

        self.frame = ttk.Frame(parent)
        self.frame.pack(fill="both", expand=True)

        self.criar_interface()

    def criar_interface(self):
        """Cria interface visual do menu principal"""
        # Cabeçalho
        header = ttk.Frame(self.frame)
        header.pack(fill="x", padx=20, pady=10)

        ttk.Label(
            header,
            text=f"Bem-vindo, {self.usuario.get('Login')}",
            font=Config.FONTE_TITULO
        ).pack(side="left")

        ttk.Label(
            header,
            text=f"Perfil: {self.usuario.get('Tipo')}",
            font=Config.FONTE_NORMAL
        ).pack(side="left", padx=20)
```

```
        btn_sair = ttk.Button(header, text="Sair", command=self.voltar_login_callback)
        btn_sair.pack(side="right")

        # Notebook (abas)
        self.notebook = ttk.Notebook(self.frame)
        self.notebook.pack(fill="both", expand=True, padx=20, pady=10)

        # Adicionar abas
        for nome_aba, callback in self.criar_aba_callbacks.items():
            callback(self.notebook)

    def destruir(self):
        """Remove o menu principal"""
        self.frame.destroy()
```

Figura 19: Menu Principal-Layout.

Fonte: Código do Projeto Integrado Multidisciplinar, originalmente desenvolvido no VS Code.

Tela de Alunos – Layout

A classe TelaAlunosLayout é responsável por estruturar a interface destinada ao gerenciamento de alunos dentro da aplicação. Sua composição organiza

elementos visuais e funcionais de forma clara, permitindo realizar operações de cadastro, consulta e manutenção dos registros. A estrutura apresenta as seguintes características:

• Formulário de cadastro:

Inclui campos para inserção de dados essenciais, como RA, nome, e-mail e status do aluno. Esses componentes possibilitam a inclusão e edição de informações de forma organizada e padronizada.

• Área de busca:

Disponibiliza um campo para pesquisa por RA, permitindo localizar rapidamente um aluno específico. Essa área integra-se a callbacks de ação que executam funções de busca e filtragem.

• Tabela de exibição de dados:

Apresenta uma tabela (Treeview) que mostra os registros existentes, facilitando a visualização e manipulação dos dados. A tabela é configurada para exibir colunas relevantes e permitir interação com os itens listados.

• Conjunto de botões de ação:

Inclui funcionalidades como adicionar, limpar campos, buscar, mostrar todos, editar, excluir e ativar/desativar alunos. Os botões são ligados a callbacks que realizam as operações correspondentes, contribuindo para a automação dos processos.

• Organização visual:

Utiliza os gerenciadores de layout grid e pack para posicionar os widgets de forma coerente e equilibrada, garantindo melhor usabilidade e fluxo visual.

Como resultado, a classe centraliza toda a estrutura gráfica relacionada aos alunos, facilitando tanto o uso quanto a manutenção do módulo, além de promover consistência estética e funcional na aplicação.

```
# ============ TELA DE ALUNOS - LAYOUT ============

class TelaAlunosLayout:
    """Layout da tela de gerenciamento de alunos"""

    def __init__(self, parent, callbacks):
        self.parent = parent
        self.callbacks = callbacks
        self.criar_interface()

    def criar_interface(self):
        """Cria interface visual da tela de alunos"""
        # Título
        ttk.Label(
            self.parent,
            text="Gerenciamento de Alunos",
            font=Config.FONTE_SUBTITULO
        ).pack(pady=10)

        # Frame de formulário
        form_frame = ttk.LabelFrame(self.parent, text="Cadastro de Aluno", padding=15)
        form_frame.pack(fill="x", padx=10, pady=10)

        # Campos
        ttk.Label(form_frame, text="RA:").grid(row=0, column=0, sticky="w", padx=5, pady=5)
        self.entrada_ra = ttk.Entry(form_frame, width=20)
        self.entrada_ra.grid(row=0, column=1, padx=5, pady=5)

        ttk.Label(form_frame, text="Nome Completo:").grid(row=0, column=2, sticky="w", padx=5, pady=5)
        self.entrada_nome = ttk.Entry(form_frame, width=40)
        self.entrada_nome.grid(row=0, column=3, padx=5, pady=5)
```

```
ttk.Label(form_frame, text="Email:").grid(row=1, column=0, sticky="w", padx=5, pady=5)
self.entrada_email = ttk.Entry(form_frame, width=40)
self.entrada_email.grid(row=1, column=1, columnspan=3, sticky="ew", padx=5, pady=5)

ttk.Label(form_frame, text="Status:").grid(row=2, column=0, sticky="w", padx=5, pady=5)
self.entrada_ativo = ttk.Combobox(form_frame, values=["Ativo", "Inativo"], state="readonly", width=18)
self.entrada_ativo.set("Ativo")
self.entrada_ativo.grid(row=2, column=1, padx=5, pady=5)

# Botões do formulário
btn_frame = ttk.Frame(form_frame)
btn_frame.grid(row=3, column=0, columnspan=4, pady=10)

ttk.Button(btn_frame, text="Adicionar", command=self.callbacks['adicionar']).pack(side="left", padx=5)
ttk.Button(btn_frame, text="Limpar", command=self.callbacks['limpar']).pack(side="left", padx=5)

# Busca
busca_frame = ttk.Frame(self.parent)
busca_frame.pack(fill="x", padx=10, pady=5)

ttk.Label(busca_frame, text="Buscar por RA:").pack(side="left", padx=5)
self.entrada_busca = ttk.Entry(busca_frame, width=20)
self.entrada_busca.pack(side="left", padx=5)
ttk.Button(busca_frame, text="Buscar", command=self.callbacks['buscar']).pack(side="left", padx=5)
ttk.Button(busca_frame, text="Mostrar Todos", command=self.callbacks['mostrar_todos']).pack(side="left", padx=5)

# Tabela
tabela_frame, self.tree = ComponentesUI.criar_tabela(
    self.parent,
    ["RA", "Nome", "Email", "Status"],
    altura=12
)
tabela_frame.pack(fill="both", expand=True, padx=10, pady=10)
```

```
# Botões de ação
acoes_frame = ttk.Frame(self.parent)
acoes_frame.pack(fill="x", padx=10, pady=10)

ttk.Button(acoes_frame, text="Editar Selecionado", command=self.callbacks['editar']).pack(side="left", padx=5)
ttk.Button(acoes_frame, text="Excluir Selecionado", command=self.callbacks['excluir']).pack(side="left", padx=5)
ttk.Button(acoes_frame, text="Ativar/Desativar", command=self.callbacks['toggle_status']).pack(side="left", padx=5)
```

Figura 20: Tela de Alunos-Layout.

Fonte: Código do Projeto Integrado Multidisciplinar, originalmente desenvolvido no VS Code.

Tela de Turmas – Layout

A classe TelaTurmasLayout é responsável por estruturar a interface destinada ao gerenciamento das turmas no sistema. Seu objetivo é organizar, de forma clara e funcional, todos os elementos necessários para o cadastro, consulta e manutenção dessas informações. A estrutura apresenta as seguintes características principais:

• Formulário de cadastro:

Contém campos para inserção de dados essenciais, como ID (opcional), nome da turma, professor responsável, ano e semestre. Esse conjunto de entradas permite o registro ou edição das turmas de forma padronizada.

• Botões de ação:

Inclui botões para adicionar nova turma e limpar o formulário, garantindo agilidade na execução das operações e evitando erros por preenchimento incorreto. Cada botão é vinculado a funções de callback que tratam a lógica correspondente.

• Área de filtros e busca:

Oferece mecanismo para pesquisa por professor, permitindo localizar turmas específicas de maneira eficiente e filtrada.

• Tabela de listagem:

Apresenta uma tabela (Treeview) destinada à visualização das turmas cadastradas. Essa lista exibe colunas relevantes e permite selecionar registros para edição ou exclusão, facilitando a navegação e gestão dos dados.

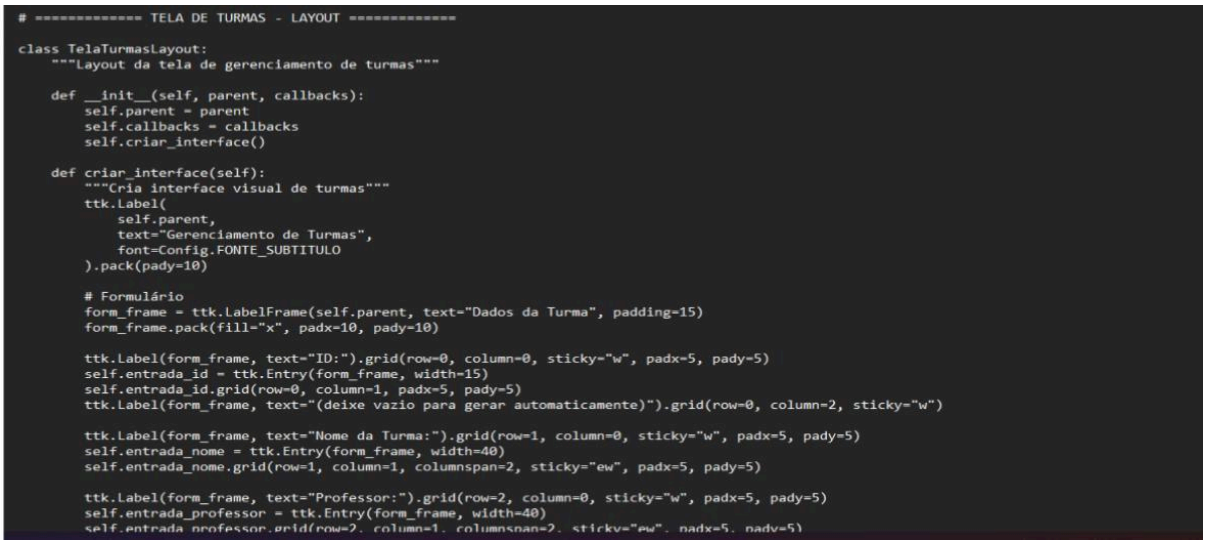
• Ações sobre turmas selecionadas:

Disponibiliza funcionalidades para editar ou excluir turmas escolhidas na tabela, garantindo controle completo sobre os registros.

• Organização visual:

Emprega os gerenciadores de layout grid e pack para posicionar e distribuir os componentes da interface, promovendo clareza, equilíbrio e boa usabilidade.

Com essa estrutura, a classe TelaTurmasLayout centraliza o design visual e a organização lógica da área de turmas, contribuindo para uma interface intuitiva, padronizada e alinhada às boas práticas de desenvolvimento de interfaces gráficas.



```
# ============= TELA DE TURMAS - LAYOUT =============

class TelaTurmasLayout:
    """Layout da tela de gerenciamento de turmas"""

    def __init__(self, parent, callbacks):
        self.parent = parent
        self.callbacks = callbacks
        self.criar_interface()

    def criar_interface(self):
        """Cria interface visual de turmas"""
        ttk.Label(
            self.parent,
            text="Gerenciamento de Turmas",
            font=Config.FONTE_SUBTITULO
        ).pack(pady=10)

        # Formulário
        form_frame = ttk.LabelFrame(self.parent, text="Dados da Turma", padding=15)
        form_frame.pack(fill="x", padx=10, pady=10)

        ttk.Label(form_frame, text="ID:").grid(row=0, column=0, sticky="w", padx=5, pady=5)
        self.entrada_id = ttk.Entry(form_frame, width=15)
        self.entrada_id.grid(row=0, column=1, padx=5, pady=5)
        ttk.Label(form_frame, text="(deixe vazio para gerar automaticamente)").grid(row=0, column=2, sticky="w")

        ttk.Label(form_frame, text="Nome da Turma:").grid(row=1, column=0, sticky="w", padx=5, pady=5)
        self.entrada_nome = ttk.Entry(form_frame, width=40)
        self.entrada_nome.grid(row=1, column=1, columnspan=2, sticky="ew", padx=5, pady=5)

        ttk.Label(form_frame, text="Professor:").grid(row=2, column=0, sticky="w", padx=5, pady=5)
        self.entrada_professor = ttk.Entry(form_frame, width=40)
        self.entrada_professor.grid(row=2, column=1, columnspan=2, sticky="ew", padx=5, pady=5)
```

Figura 21: Elaboração Tela de Turmas.

Fonte: Código do Projeto Integrado Multidisciplinar, originalmente desenvolvido no VS Code.

Tela de Associações – Layout

A classe TelaAssociacaoLayout é responsável por estruturar a interface destinada ao gerenciamento da associação entre alunos e turmas (matrículas). Ela organiza, de maneira clara e funcional, todos os elementos relacionados ao processo de vinculação, consulta e remoção dessas associações.

• Formulário de Matrícula:

Apresenta campos para inserção dos identificadores necessários (RA do aluno e ID da turma), permitindo registrar novas associações de forma precisa e padronizada.

• Botões de Ação:

Inclui botões destinados à matrícula e à limpeza do formulário, promovendo agilidade no processo e reduzindo riscos de erros por preenchimento incorreto. Cada botão é vinculado a funções de callback responsáveis pela lógica operacional.

• Tabela de Visualização:

Exibe uma tabela (Treeview) que lista os alunos associados a uma turma selecionada, permitindo ao usuário verificar facilmente a composição das matrículas. Essa tabela pode ser atualizada a partir de filtros ou ações realizadas.

• Remoção de Matrículas:

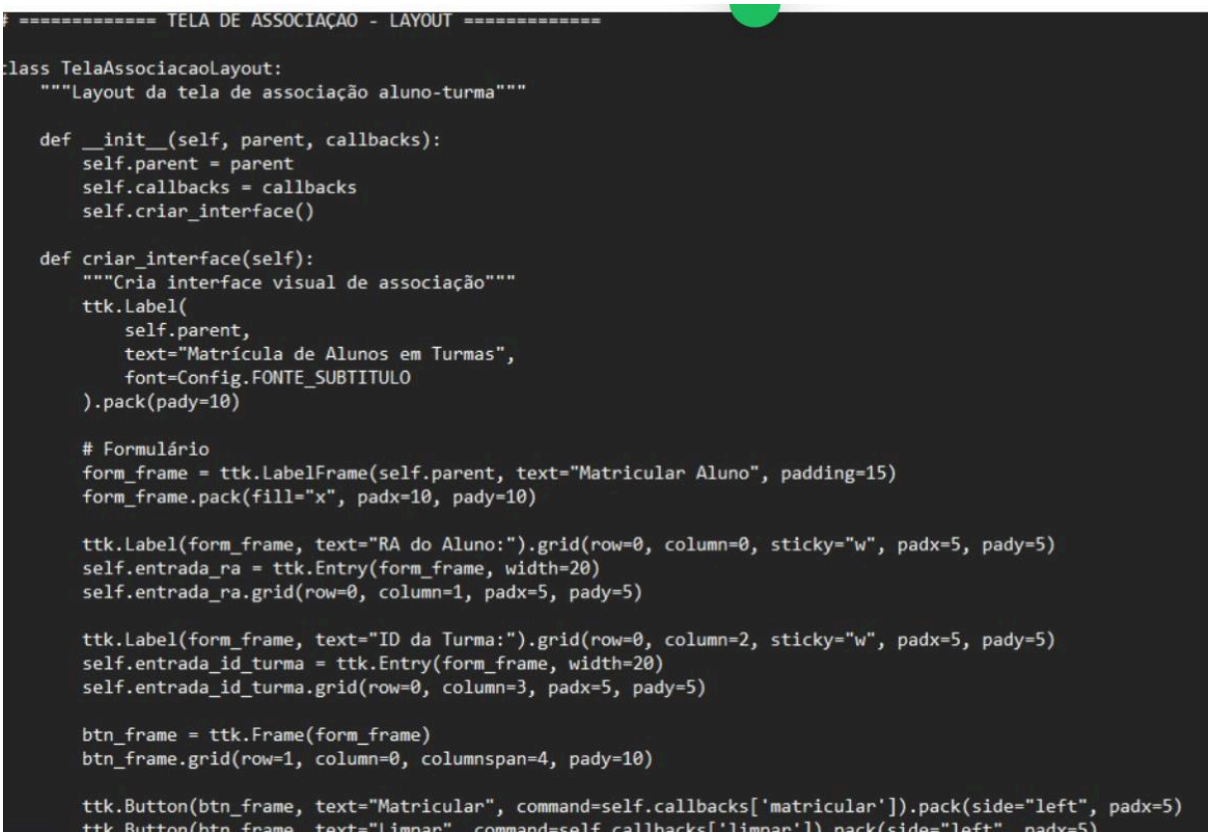
Disponibiliza um botão específico para remover a matrícula selecionada. Ao acioná-lo, o sistema verifica se existe uma linha selecionada e apresenta uma caixa de confirmação utilizando tkinter.messagebox.askyesno.

Se o usuário confirmar a ação, o callback de remoção é executado. Após a exclusão, a tabela é atualizada por meio da limpeza dos itens (tree.delete(*tree.get_children())) e da recarga dos dados, garantindo que a interface reflita imediatamente o estado atualizado do sistema.

• Organização Visual:

A estrutura utiliza os gerenciadores de layout do Tkinter (grid e pack) para distribuir os elementos de maneira equilibrada, garantindo boa leitura, fluidez de navegação e consistência visual com as demais telas da aplicação.

Com essa composição, a classe TelaAssociacaoLayout assegura um ambiente organizado para o gerenciamento de matrículas, combinando clareza, confirmação de ações críticas e atualização dinâmica da interface para evitar divergências entre visualização e dados persistidos.



```
# ============ TELA DE ASSOCIAÇÃO - LAYOUT ============

class TelaAssociacaoLayout:
    """Layout da tela de associação aluno-turma"""

    def __init__(self, parent, callbacks):
        self.parent = parent
        self.callbacks = callbacks
        self.criar_interface()

    def criar_interface(self):
        """Cria interface visual de associação"""
        ttk.Label(
            self.parent,
            text="Matrícula de Alunos em Turmas",
            font=Config.FONTE_SUBTITULO
        ).pack(pady=10)

        # Formulário
        form_frame = ttk.LabelFrame(self.parent, text="Matricular Aluno", padding=15)
        form_frame.pack(fill="x", padx=10, pady=10)

        ttk.Label(form_frame, text="RA do Aluno:").grid(row=0, column=0, sticky="w", padx=5, pady=5)
        self.entrada_ra = ttk.Entry(form_frame, width=20)
        self.entrada_ra.grid(row=0, column=1, padx=5, pady=5)

        ttk.Label(form_frame, text="ID da Turma:").grid(row=0, column=2, sticky="w", padx=5, pady=5)
        self.entrada_id_turma = ttk.Entry(form_frame, width=20)
        self.entrada_id_turma.grid(row=0, column=3, padx=5, pady=5)

        btn_frame = ttk.Frame(form_frame)
        btn_frame.grid(row=1, column=0, columnspan=4, pady=10)

        ttk.Button(btn_frame, text="Matricular", command=self.callbacks['matricular']).pack(side="left", padx=5)
        ttk.Button(btn_frame, text="Limpar", command=self.callbacks['limpar']).pack(side="left", padx=5)
```

Figura 22: Elaboração Tela de Associação-Layout.

Fonte: Código do Projeto Integrado Multidisciplinar,originalmente desenvolvido no VS Code.

```
# ============ TELA DE AULAS - LAYOUT ============

class TelaAulasLayout:
    """Layout da tela de registro de aulas"""

    def __init__(self, parent, callbacks):
        self.parent = parent
        self.callbacks = callbacks
        self.criar_interface()

    def criar_interface(self):
        """Cria interface visual de aulas"""
        ttk.Label(
            self.parent,
            text="Registro de Aulas - Diário Eletrônico",
            font=Config.FONTE_SUBTITULO
        ).pack(pady=10)

        # Formulário
        form_frame = ttk.LabelFrame(self.parent, text="Registrar Aula", padding=15)
        form_frame.pack(fill="x", padx=10, pady=10)

        ttk.Label(form_frame, text="ID da Turma:").grid(row=0, column=0, sticky="w", padx=5, pady=5)
        self.entrada_id_turma = ttk.Entry(form_frame, width=15)
        self.entrada_id_turma.grid(row=0, column=1, padx=5, pady=5)

        ttk.Label(form_frame, text="Data (DD/MM/AAAA):").grid(row=0, column=2, sticky="w", padx=5, pady=5)
        self.entrada_data = ttk.Entry(form_frame, width=15)
        self.entrada_data.grid(row=0, column=3, padx=5, pady=5)

        ttk.Label(form_frame, text="Conteúdo Abordado:").grid(row=1, column=0, sticky="nw", padx=5, pady=5)

        text_frame = ttk.Frame(form_frame)
        text_frame.grid(row=1, column=1, columnspan=3, sticky="ew", padx=5, pady=5)
```

Figura 23: Elaboração Tela de Aulas.

Fonte: Código do Projeto Integrado Multidisciplinar, originalmente desenvolvido no VS Code.

Aplicação Principal – Layout

A configuração visual da aplicação é realizada por meio da definição de estilos utilizando os recursos oferecidos pelo ttk em Tkinter. Para isso, cria-se uma instância da classe ttk.Style(), responsável por gerenciar o tema e os padrões visuais aplicados aos componentes gráficos. Após a criação da janela principal por meio de Tk(), utiliza-se o método theme_use('clam') para aplicar o tema clam, reconhecido por fornecer aparência moderna, limpa e visualmente uniforme aos widgets.

A definição do estilo deve ocorrer antes da execução do loop principal da interface para garantir que todos os componentes, ao serem renderizados, recebam a padronização visual selecionada. Essa configuração promove uniformidade na apresentação dos elementos gráficos e reforça a consistência estética da aplicação.

```
# ============= APLICAÇÃO PRINCIPAL - LAYOUT =============

class AppLayout:
    """Classe principal da aplicação - configuração visual"""

    def __init__(self):
        self.root = tk.Tk()
        self.root.title("Sistema Acadêmico PIM 2025")
        self.root.geometry("1200x700")

        # Configurar estilo
        self.configurar_estilo()

    def configurar_estilo(self):
        """Configura estilos visuais do ttk"""
        style = ttk.Style()
        style.theme_use('clam')

        # Configurações gerais
        self.root.configure(bg=Config.COR_FUNDO)

    def executar(self):
        """Inicia o loop principal"""
        self.root.mainloop()
```

Figura 24: Elaboração da Aplicação Principal-Layout.

Fonte: Código do Projeto Integrado Multidisciplinar, originalmente desenvolvido no VS Code.

11. CONCLUSÃO

O desenvolvimento do Sistema Acadêmico Colaborativo com apoio de Inteligência Artificial atingiu plenamente os objetivos estabelecidos no início deste Projeto Integrado Multidisciplinar. A pergunta central que norteou este trabalho questionava se seria possível criar uma solução tecnológica integrada capaz de modernizar a gestão acadêmica, substituindo processos descentralizados e eliminando o uso de papel, ao mesmo tempo em que demonstrasse a aplicação prática dos conhecimentos adquiridos nas oito disciplinas do semestre.

A hipótese de que a integração entre linguagens de programação de diferentes paradigmas, aliada a metodologias ágeis e recursos de inteligência artificial, resultaria em um sistema funcional e sustentável foi confirmada ao longo do desenvolvimento. O projeto demonstrou que é plenamente viável construir uma aplicação robusta utilizando C para módulos de persistência e Python para interface gráfica, mantendo coesão arquitetural e desempenho adequado.

A adoção da metodologia Scrum revelou-se acertada, permitindo organização eficiente do trabalho em sprints semanais e adaptação contínua aos desafios encontrados. A divisão de papéis entre desenvolvedores de backend, frontend, analistas e documentadores facilitou o paralelismo das atividades e garantiu entregas incrementais funcionais. As reuniões de sincronização e o uso do Jira como ferramenta de controle proporcionaram visibilidade constante do progresso e resolução ágil de impedimentos.

Do ponto de vista técnico, a implementação em C atendeu ao objetivo de compreender sistemas de baixo nível e manipulação eficiente de arquivos. Os módulos CRUD desenvolvidos demonstraram domínio de estruturas de decisão, repetição, ponteiros e gerenciamento de memória. A persistência em arquivos CSV, embora simplificada para fins acadêmicos, cumpriu sua função didática e pode ser facilmente substituída por um banco de dados relacional em evolução futura do sistema.

A interface gráfica desenvolvida em Python com Tkinter superou as expectativas iniciais, oferecendo experiência de usuário intuitiva e responsiva. O sistema de autenticação com três níveis de acesso (administrador, professor e aluno) foi implementado com sucesso, garantindo controle granular de permissões. A integração entre frontend Python e backend C, embora desafiadora, funcionou

conforme planejado, validando a estratégia de comunicação por meio de leitura e escrita coordenada nos arquivos CSV.

O objetivo de sustentabilidade foi alcançado através da digitalização completa dos processos acadêmicos. O diário eletrônico eliminou a necessidade de cadernetas físicas, e o sistema de relatórios digitais substituiu impressões em massa. Embora não seja possível quantificar exatamente o impacto ambiental durante o período de desenvolvimento do projeto, a solução criada demonstra claramente o potencial de economia de papel e recursos naturais quando implementada em ambiente de produção.

A aplicação de recursos de inteligência artificial, principalmente através de ferramentas generativas como Claude AI, ChatGPT e GitHub Copilot, trouxe ganhos significativos de produtividade. Essas tecnologias auxiliaram na estruturação de código, otimização de algoritmos, documentação e resolução de impedimentos técnicos. A experiência demonstrou que a IA pode atuar como catalisadora do processo de desenvolvimento, sem substituir a necessidade de compreensão profunda dos conceitos por parte da equipe.

A documentação UML desenvolvida cumpriu seu papel de ponte entre análise e implementação. Os diagramas de casos de uso, classes e sequência foram constantemente consultados durante a codificação, garantindo alinhamento entre requisitos e código-fonte. Essa abordagem reforçou a importância da fase de análise e projeto para o sucesso de sistemas computacionais complexos.

Do ponto de vista de aprendizado, este projeto consolidou conhecimentos teóricos através de aplicação prática integrada. Cada disciplina contribuiu com conceitos fundamentais que se materializaram em funcionalidades concretas do sistema. A experiência de trabalhar em equipe, gerenciar conflitos técnicos e tomar decisões arquiteturais preparou o grupo para desafios profissionais futuros na área de desenvolvimento de software.

Reconhecemos limitações no projeto atual que representam oportunidades de evolução. A persistência em arquivos CSV, embora adequada para fins didáticos, não suporta concorrência real e deveria ser substituída por um sistema de banco de dados relacional em ambiente de produção. A ausência de criptografia nas senhas armazenadas, decisão consciente para simplificar o escopo acadêmico, seria inadmissível em aplicação

comercial. A arquitetura de rede, embora documentada, não foi implementada completamente devido a restrições de tempo e infraestrutura disponível.

Apesar dessas limitações, o resultado final demonstra que os objetivos fundamentais foram atingidos. O sistema desenvolvido é funcional, atende aos requisitos estabelecidos e pode servir como base para implementações reais em instituições de ensino de pequeno e médio porte. A experiência adquirida pela equipe transcende o código produzido, abrangendo competências em gestão de projetos, trabalho colaborativo, resolução de problemas complexos e integração de conhecimentos multidisciplinares.

Concluímos que o desenvolvimento de sistemas informatizados exige não apenas domínio técnico de linguagens e ferramentas, mas também capacidade de compreender necessidades do usuário final, aplicar metodologias adequadas, documentar decisões e trabalhar de forma colaborativa. Este projeto integrou com sucesso teoria e prática, demonstrando que a formação em Análise e Desenvolvimento de Sistemas prepara profissionais capazes de conceber, desenvolver e entregar soluções tecnológicas completas que agregam valor real aos seus usuários.

12. REFERÊNCIAS

ALVES, William Pereira. **Fundamentos de Lógica de Programação**. São Paulo: Érica, 2014.

ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi de. **Fundamentos da Programação de Computadores: Algoritmos, Pascal, C/C++ e Java**. 3. ed. São Paulo: Pearson Prentice Hall, 2012.

BECK, Kent et al. **Manifesto para Desenvolvimento Ágil de Software**. 2001. Disponível em: https://agilemanifesto.org/iso/ptbr/manifesto.html. Acesso em: 07 nov. 2025.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML: Guia do Usuário**. 2. ed. Rio de Janeiro: Elsevier, 2012.

CELES, Waldemar; CERQUEIRA, Renato; RANGEL, José Lucas. **Introdução a Estruturas de Dados: Com Técnicas de Programação em C**. 2. ed. Rio de Janeiro: Elsevier, 2016.

CESAR. **7 tendências tecnológicas para inovar e transformar em 2025**. Disponível em: https://www.cesar.org.br. Acesso em: 07 nov. 2025.

DAMAS, Luís. **Linguagem C**. 10. ed. Rio de Janeiro: LTC, 2007.

FORBELLONE, André Luiz Villar; EBERSPÄCHER, Henri Frederico. **Lógica de Programação: A Construção de Algoritmos e Estruturas de Dados**. 3. ed. São Paulo: Pearson Prentice Hall, 2005.

GOVERNO DO BRASIL. **Texto Referência - Ciência, Tecnologia e Inovação - CTI**. Disponível em: https://www.gov.br. Acesso em: 07 nov. 2025.

GUEDES, Gilleanes T. A. **UML 2: Uma Abordagem Prática**. 3. ed. São Paulo: Novatec, 2018.

KERNIGHAN, Brian W.; RITCHIE, Dennis M. **C: A Linguagem de Programação Padrão ANSI**. Rio de Janeiro: Elsevier, 1989.

KUROSE, James F.; ROSS, Keith W. **Redes de Computadores e a Internet: Uma Abordagem Top-Down**. 6. ed. São Paulo: Pearson, 2013.

LUTZ, Mark. **Learning Python**. 5. ed. Sebastopol: O'Reilly Media, 2013.

MANZANO, José Augusto N. G.; OLIVEIRA, Jayr Figueiredo de. **Algoritmos: Lógica para Desenvolvimento de Programação de Computadores**. 28. ed. São Paulo: Érica, 2016.

MATTHES, Eric. **Curso Intensivo de Python: Uma Introdução Prática e Baseada em Projetos à Programação**. São Paulo: Novatec, 2016.

MINISTÉRIO DO MEIO AMBIENTE. **Educação Ambiental: Por um Brasil Sustentável**. Brasília: MMA, 2014. Disponível em: https://www.gov.br/mma. Acesso em: 07 nov. 2025.

PEREIRA, Silvio do Lago. **Estruturas de Dados Fundamentais: Conceitos e Aplicações**. 12. ed. São Paulo: Érica, 2008.

PRESSMAN, Roger S.; MAXIM, Bruce R. **Engenharia de Software: Uma Abordagem Profissional**. 9. ed. Porto Alegre: AMGH, 2021.

RUSSELL, Stuart; NORVIG, Peter. **Inteligência Artificial: Uma Abordagem Moderna**. 4. ed. Rio de Janeiro: LTC, 2022.

SCHWABER, Ken; SUTHERLAND, Jeff. **The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game**. 2020. Disponível em: https://scrumguides.org. Acesso em: 07 nov. 2025.

SCHILDT, Herbert. **C Completo e Total**. 3. ed. São Paulo: Makron Books, 1997.

SOMMERVILLE, Ian. **Engenharia de Software**. 10. ed. São Paulo: Pearson, 2019.

TANENBAUM, Andrew S.; WETHERALL, David J. **Redes de Computadores**. 5. ed. São Paulo: Pearson, 2011.

TANENBAUM, Andrew S.; WOODHULL, Albert S. **Sistemas Operacionais: Projeto e Implementação**. 3. ed. Porto Alegre: Bookman, 2008.

TAURION, Cezar. **Cloud Computing: Computação em Nuvem - Transformando o Mundo da Tecnologia da Informação**. Rio de Janeiro: Brasport, 2009.

ZIVIANI, Nivio. **Projeto de Algoritmos: Com Implementações em Pascal e C**. 3. ed. São

FICHA DE CONTROLE DO PIM

Grupo Nº _____ Ano_____ Período: _____ Orientador _____

Tema:_____

Alunos:

RA	Nome	E-mail	Curso	Visto do aluno
H77GCH3	JOÃO LUCAS LEMES MUASSAB			
R958990	ENZO XAVIER SANTOS			
F365166	FERNANDA DA ROCHA GOMES			
R860CI1	JOÃO PEDRO ANTONIO PEREIRA			

Registros:

Data do encontro	Observações