UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

NICOLAS SIQUEIRA MORAES

REFATORAÇÃO DO SISTEMA SGTCC PARA MELHORIA DA QUALIDADE DO CÓDIGO COM BASE EM BOAS PRÁTICAS DE ENGENHARIA DE SOFTWARE

GUARAPUAVA

2026

NICOLAS SIQUEIRA MORAES

REFATORAÇÃO DO SISTEMA SGTCC PARA MELHORIA DA QUALIDADE DO CÓDIGO COM BASE EM BOAS PRÁTICAS DE ENGENHARIA DE SOFTWARE

Refactoring the SGTCC System to Improve Code Quality Based on Software Engineering Best Practices

Projeto de Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Tecnólogo em Tecnologia em Sistemas para Internet do Curso Superior de Tecnologia em Sistemas para Internet da Universidade Tecnológica Federal do Paraná.

Orientadora: Profª. Drª. Renata Luiza Stange

Coorientador: Prof. Dr. Diego Marczal

GUARAPUAVA

2026

RESUMO

Este trabalho aborda a refatoração do Sistema de Gestão de Trabalhos de Conclusão de Curso (SGTCC), desenvolvido no contexto acadêmico da Universidade Tecnológica Federal do Paraná. Ao longo de sua evolução incremental, o sistema passou a apresentar problemas estruturais, como alto acoplamento entre componentes, duplicação de lógica e inconsistências na representação de dados, o que compromete sua manutenibilidade e evolução. Diante desse cenário, o presente trabalho tem como objetivo propor melhorias na organização do código, por meio da aplicação de técnicas de refatoração e boas práticas de engenharia de software, sem alteração de suas funcionalidades. A metodologia adotada caracteriza-se como uma pesquisa aplicada, com abordagem qualitativa e conduzida por meio de estudo de caso, envolvendo a análise da base de código, definição de estratégias de refatoração e implementação incremental das melhorias propostas. Como resultado esperado, busca-se tornar o sistema mais modular, compreensível e flexível, facilitando sua manutenção e adaptação a novas demandas. Além disso, espera-se reduzir problemas estruturais existentes, contribuindo para a melhoria da qualidade interna do software. Como principal contribuição, este trabalho evidencia a importância da refatoração como estratégia para a evolução sustentável de sistemas desenvolvidos de forma incremental.

Palavras-chave: refatoracao; engenharia de software; manutenibilidade; arquitetura de software; sistemas web.

ABSTRACT

This work addresses the refactoring of the Final Course Project Management System (SGTCC), developed in the academic context of the Federal University of Technology – Paraná. Throughout its incremental evolution, the system has presented structural issues such as high coupling between components, code duplication, and inconsistencies in data representation, which compromise its maintainability and evolution. Given this scenario, this work aims to propose improvements in code organization through the application of refactoring techniques and software engineering best practices, without altering its functionalities. The adopted methodology is characterized as applied research, with a qualitative approach and conducted through a case study, involving codebase analysis, definition of refactoring strategies, and incremental implementation of the proposed improvements. As an expected result, the system is intended to become more modular, understandable, and flexible, facilitating its maintenance and adaptation to new demands. Additionally, it is expected to reduce existing structural issues, contributing to the improvement of the internal quality of the software. As the main contribution, this work highlights the importance of refactoring as a strategy for the sustainable evolution of systems developed incrementally.

Keywords: refactoring; software engineering; maintainability; software architecture; web systems.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

Siglas

SGTCC	Sistema de Gestão de Trabalhos de Conclusão de Curso
SI	Sistemas para Internet
SOLID	Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation e Dependency Inversion
TCC	Trabalho de Conclusão de Curso
TSI	Tecnologia em Sistemas para Internet
UTFPR	Universidade Tecnológica Federal do Paraná

SUMÁRIO

1 INTRODUÇÃO

No curso de graduação em Sistemas para Internet (SI) da Universidade Tecnológica Federal do Paraná (UTFPR), Campus Guarapuava, os estudantes desenvolvem projetos voltados à criação de soluções tecnológicas ao longo de sua formação, sendo o Trabalho de Conclusão de Curso (TCC) uma das etapas finais para a formação. Inicialmente, a gestão das atividades relacionadas ao TCC era realizada de forma manual, envolvendo a utilização de documentos físicos para submissão, avaliação e correção dos trabalhos. Com a busca por maior praticidade no gerenciamento dessas atividades, esse modelo foi gradativamente substituído por uma abordagem automatizada, permitindo a centralização e organização das informações em um sistema computacional.

Nesse contexto, foi desenvolvido o SGTCC, que vem sendo aprimorado ao longo dos anos para atender às demandas acadêmicas da instituição. Desde sua criação, o sistema passou por diferentes etapas de evolução, conforme apresentado em trabalhos como Ferreira (2015), Silva (2019), Lima (2023) e Luz (2025), incorporando novas funcionalidades, melhorias de usabilidade e atualizações tecnológicas. Entretanto, assim como ocorre em sistemas que evoluem continuamente, o crescimento incremental do SGTCC contribuiu para o surgimento de limitações relacionadas à organização do código, aumento do acoplamento entre componentes e dificuldades de manutenção. Esses problemas são amplamente discutidos na área de engenharia de software

De acordo com Somerville (2019), a engenharia de software envolve a aplicação de princípios sistemáticos, disciplinados e quantificáveis para o desenvolvimento, operação e manutenção de sistemas de software. Nesse sentido, problemas como alto acoplamento, baixa coesão e dificuldades de manutenção, observados no SGTCC, indicam a necessidade de adoção de boas práticas que favoreçam a qualidade do código e a organização da arquitetura, especialmente em sistemas que evoluem continuamente ao longo do tempo.

Diante desse cenário, torna-se necessário aprimorar a estrutura interna do sistema, assegurando uma base de código mais organizada, compreensível e preparada para evolução contínua. Nesse contexto, o conceito de Código Limpo (*Clean Code*) refere-se à escrita de código claro, legível e de fácil manutenção, enquanto a Arquitetura Limpa (*Clean Architecture*) diz respeito à organização do sistema em camadas bem definidas, com separação de responsabilidades e baixo acoplamento entre componentes.

A adoção desses princípios, aliada a técnicas de refatoração, apresenta-se como uma abordagem adequada para lidar com esses desafios. Segundo Martin (2009), a escrita de código claro e expressivo facilita sua compreensão e modificação, enquanto Martin (2017) destaca a importância da separação de responsabilidades para promover sistemas mais flexíveis e sustentáveis ao longo do tempo.

A refatoração, conforme definido por Fowler (2019), consiste na reestruturação do código-fonte com o objetivo de melhorar sua estrutura interna sem alterar seu comportamento

externo. Essa prática é especialmente relevante em sistemas que evoluem continuamente, como o SGTCC, permitindo a melhoria da qualidade do código e a redução de problemas estruturais acumulados ao longo do tempo.

Entre os problemas identificados na base de código do sistema, destacam-se inconsistências na representação de dados, como o uso de valores enumerados de forma não padronizada, sendo representados em alguns momentos como valores numéricos e, em outros, como valores textuais, além da presença de alto acoplamento entre componentes, especialmente entre modelos e controladores. Esses aspectos, dentre outros observados, dificultam a compreensão do código, aumentam a probabilidade de erros e tornam o processo de manutenção mais complexo.

Diante do contexto apresentado, este trabalho tem como objetivo realizar a refatoração do SGTCC, com foco na melhoria da organização do código, na redução da complexidade estrutural e no aumento da manutenibilidade, sem alteração de suas funcionalidades. Para isso, serão analisados os principais problemas existentes na base de código, com o objetivo de aplicar técnicas e boas práticas de engenharia de software que contribuam para a evolução do sistema.

Dessa forma, este trabalho busca não apenas melhorar a estrutura interna do SGTCC, mas também contribuir para a aplicação prática de conceitos de engenharia de software em um sistema real, desenvolvido em ambiente acadêmico. A partir da identificação de problemas estruturais e da aplicação de técnicas de refatoração, pretende-se evidenciar os benefícios dessas práticas na melhoria da qualidade do código e na sustentabilidade do sistema ao longo do tempo. Além disso, o estudo delimita-se à análise e refatoração da camada de aplicação, sem alterações nas regras de negócio existentes, preservando o comportamento funcional do sistema.

1.1 Objetivos

Diante do contexto apresentado, que evidencia a evolução do SGTCC e os desafios relacionados à qualidade e manutenção de sua base de código, este trabalho propõe a aplicação de técnicas de refatoração fundamentadas em boas práticas de engenharia de software. Para orientar o desenvolvimento da pesquisa e delimitar seu escopo, são definidos, a seguir, o objetivo geral e os objetivos específicos do trabalho.

1.1.1 Objetivo Geral

Melhorar a qualidade e a manutenibilidade do SGTCC por meio da refatoração do código, da padronização de dados, da redução de acoplamento e da reorganização estrutural da aplicação e dos testes, fundamentada em boas práticas de engenharia de software.

1.1.2 Objetivos Específicos

1. Identificar e analisar problemas estruturais na base de código do SGTCC, relacionados à duplicação de lógica, alto acoplamento e inconsistências na representação de dados;
2. Padronizar a representação de dados no sistema por meio da adoção consistente de valores enumerados (`enums`);
3. Reduzir o acoplamento entre componentes do sistema, especialmente entre `models`, `controllers` e demais camadas, aplicando princípios de separação de responsabilidades;
4. Refatorar as `factories` de testes, visando melhorar a organização, reutilização e previsibilidade dos dados gerados;
5. Aprimorar os testes automatizados para aumentar sua legibilidade, reduzir redundâncias e ampliar a cobertura de cenários relevantes;
6. Reestruturar a organização dos `controllers`, `models` e `concerns`, promovendo a separação de responsabilidades e a adequada distribuição das regras de negócio em camadas específicas da aplicação;
7. Aplicar boas práticas de engenharia de software (como princípios SOLID e refatoração contínua) para melhorar a manutenibilidade e facilitar a evolução do sistema.

2 EVOLUÇÃO DO SGTCC E MANUTENÇÃO DE SOFTWARE

Este capítulo apresenta a evolução do Sistema de Gestão de Trabalhos de Conclusão de Curso (SGTCC) ao longo dos anos, bem como sua relação com os conceitos de manutenção e evolução de software discutidos na literatura. A análise dessa trajetória permite compreender os desafios estruturais acumulados no sistema e fundamenta a necessidade de intervenções voltadas à melhoria de sua qualidade interna.

2.1 Evolução do SGTCC

O SGTCC desempenha um papel relevante no contexto acadêmico do curso de Tecnologia em Sistemas para Internet (TSI) da UTFPR, atuando como uma ferramenta de apoio à gestão e acompanhamento das atividades relacionadas aos TCCs. Por meio da centralização de informações, o sistema permite o armazenamento e a organização de dados referentes a alunos, orientadores, bancas avaliadoras e documentos acadêmicos, facilitando o acesso e a consulta dessas informações por diferentes usuários. Essa centralização contribui para a redução de processos manuais, aumento da eficiência administrativa e maior confiabilidade no gerenciamento dos dados.

Além disso, o SGTCC possibilita o acompanhamento do ciclo completo dos trabalhos de conclusão de curso, desde a definição do tema até a avaliação final, promovendo maior transparência e controle sobre as etapas do processo. Essa característica é especialmente relevante para docentes e coordenação, que passam a dispor de informações consolidadas para apoio à tomada de decisão.

Outro aspecto importante é a utilização contínua do sistema como objeto de estudo em atividades acadêmicas, incluindo disciplinas e TCC. Nesse contexto, o SGTCC deixa de ser apenas uma ferramenta operacional e passa a atuar também como um ambiente prático para aplicação de conceitos de engenharia de software, desenvolvimento web e melhoria contínua de sistemas.

Essa característica contribui para a formação dos estudantes, que têm a oportunidade de trabalhar com um sistema real, enfrentando desafios relacionados à manutenção, evolução e qualidade de software. Além disso, a continuidade do uso do sistema ao longo dos anos reforça sua relevância institucional e evidencia a importância de garantir sua sustentabilidade e qualidade estrutural.

Dessa forma, o SGTCC configura-se não apenas como um sistema de apoio administrativo, mas também como um elemento integrador entre ensino, prática e evolução tecnológica, justificando a necessidade de iniciativas voltadas à sua melhoria contínua, como a proposta apresentada neste trabalho.

Diante da análise de sua evolução ao longo dos anos, observa-se que o sistema foi desenvolvido de forma incremental, com cada trabalho focando em aspectos específicos da

aplicação. No entanto, essa abordagem, embora eficaz para a evolução contínua do sistema, contribuiu para o surgimento de problemas estruturais, como inconsistências no código, alto acoplamento entre componentes e dificuldades de manutenção. Esses aspectos evidenciam a necessidade de uma abordagem focada na qualidade interna do sistema, como a proposta neste trabalho.

Nesse contexto, o presente trabalho se diferencia dos anteriores ao focar na refatoração do sistema, com ênfase na melhoria da qualidade do código e na organização estrutural da aplicação. Enquanto os trabalhos anteriores priorizaram a adição de funcionalidades e melhorias pontuais, esta proposta concentra-se na qualidade interna do sistema, visando garantir sua sustentabilidade e evolução a longo prazo.

2.2 Linha do Tempo

A evolução do SGTCC ocorreu de forma incremental ao longo dos anos, por meio de diferentes trabalhos acadêmicos desenvolvidos no curso de TSI da UTFPR. A Figura 1 apresenta uma visão geral dessa evolução, destacando as principais contribuições realizadas em cada período.

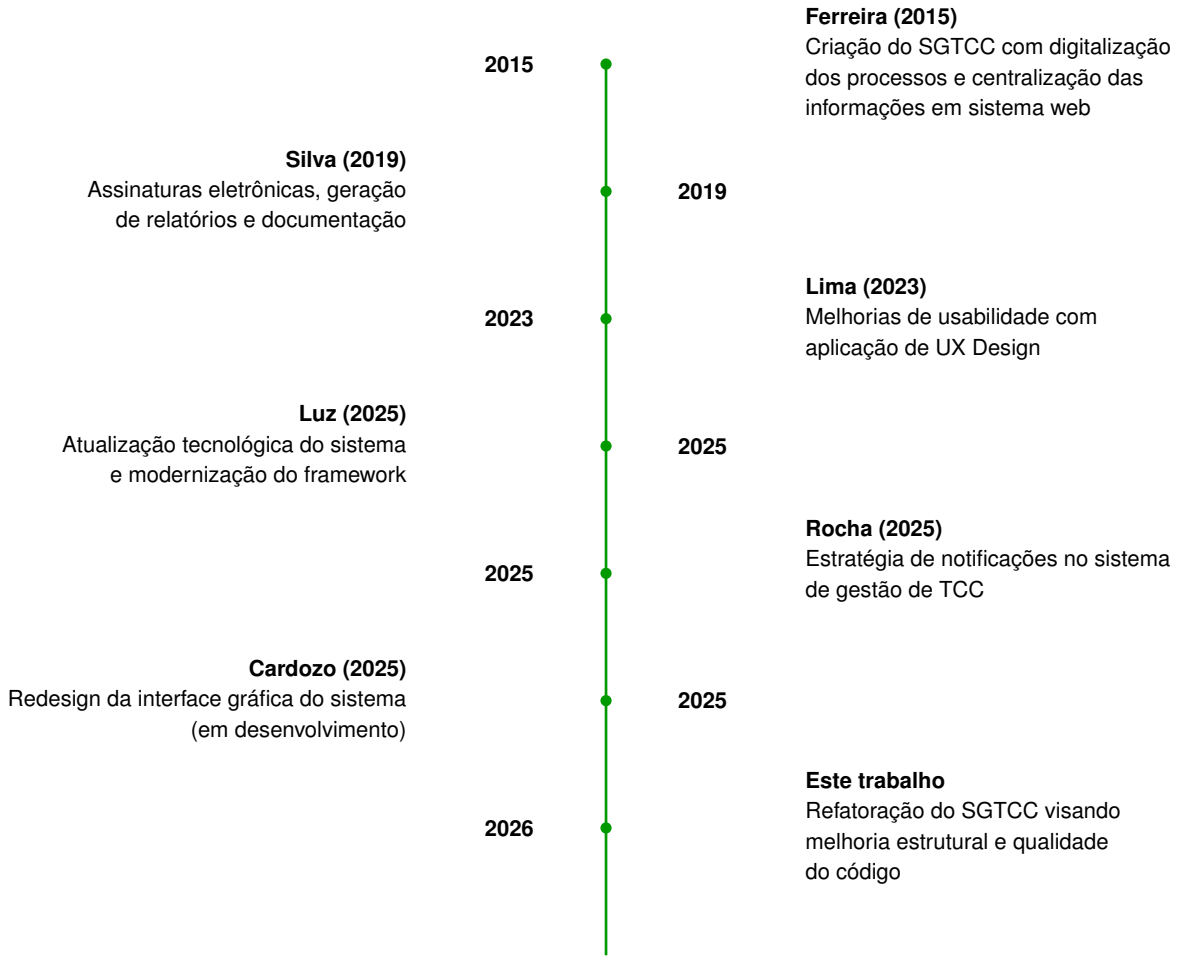

Figura 1 – Linha do tempo da evolução do SGTCC

O trabalho de Ferreira (2015) foi responsável pela criação inicial do SGTCC, propondo a digitalização dos processos relacionados ao TCC, que anteriormente eram realizados de forma manual. A principal contribuição foi a centralização das informações em um sistema web, permitindo o gerenciamento de documentos e atividades acadêmicas.

Posteriormente, Silva (2019) aperfeiçoou o SGTCC por meio da digitalização dos processos de TCC, substituindo documentos físicos por assinaturas eletrônicas e incorporando funcionalidades como geração de relatórios e documentação do sistema.

Em seguida, Lima (2023) direcionou seus esforços para a melhoria da usabilidade do sistema, aplicando conceitos de UX Design com o objetivo de tornar a interface mais intuitiva e eficiente para os usuários.

Mais recentemente, Luz (2025) realizou a atualização tecnológica do sistema, modernizando o *framework* Rails e suas dependências, garantindo maior segurança e continuidade da aplicação.

Também em 2025, Rocha (2025) contribuiu para a evolução do sistema por meio da definição e implementação de uma estratégia de notificações no SGTCC, ampliando os recursos de comunicação e acompanhamento das atividades acadêmicas.

Além disso, o trabalho de Cardozo (2025), ainda em desenvolvimento, propõe o redesign da interface gráfica do sistema, buscando aprimorar aspectos visuais e de usabilidade da aplicação.

Cabe destacar também que, além dos TCCs, o SGTCC foi utilizado como objeto de estudo na disciplina de Desenvolvimento para Web 5 nos anos de 2022 e 2023, reforçando sua relevância como ferramenta acadêmica e prática no contexto do curso.

2.3 Evolução de Software e Manutenção

A evolução de software é um fenômeno inerente a sistemas que permanecem em operação ao longo do tempo. Conforme discutido por Somerville (2019), a maior parte do esforço associado ao ciclo de vida de um sistema está relacionada às atividades de manutenção, e não ao seu desenvolvimento inicial.

A manutenção de software pode ser classificada em diferentes categorias, de acordo com seus objetivos (SOMERVILLE, 2019):

- **Manutenção corretiva**, voltada à correção de falhas identificadas durante o uso do sistema;
- **Manutenção adaptativa**, relacionada à adequação do sistema a mudanças no ambiente tecnológico;
- **Manutenção perfectiva**, associada à melhoria do sistema, incluindo otimizações e reorganização estrutural;

- **Manutenção preventiva**, direcionada à redução de problemas futuros por meio da melhoria da qualidade interna do software.

Dentre essas categorias, a manutenção perfectiva e a preventiva são particularmente relevantes em sistemas que evoluem de forma incremental. A ausência de intervenções estruturais ao longo do tempo pode levar à degradação progressiva da qualidade do código, dificultando sua manutenção e evolução (SOMERVILLE, 2019). Esse fenômeno está associado ao conceito de dívida técnica, no qual decisões de curto prazo resultam em custos adicionais futuros. Segundo Martin (2009), a falta de organização e de aplicação de boas práticas pode comprometer significativamente a sustentabilidade do software.

2.4 Relação entre a Evolução do SGTCC e a Manutenção

A análise da trajetória do SGTCC evidencia características típicas de sistemas que evoluem de forma incremental. Ao longo dos anos, diferentes trabalhos contribuíram para a ampliação de funcionalidades e para a melhoria da experiência do usuário, sem, contudo, uma abordagem sistemática voltada à qualidade estrutural do sistema.

Como consequência, observa-se o acúmulo de problemas como:

- alto acoplamento entre componentes;
- duplicação de lógica;
- inconsistências na representação de dados;
- dificuldades de manutenção e evolução.

Esses aspectos indicam a necessidade de adoção de estratégias de manutenção voltadas à melhoria da qualidade interna do sistema. Nesse contexto, a refatoração surge como uma prática essencial, alinhada às atividades de manutenção perfectiva e preventiva. Conforme destacado por Fowler (2019), a refatoração permite melhorar a organização do código sem alterar seu comportamento externo, contribuindo para a redução da complexidade e para a facilitação de futuras modificações.

3 METODOLOGIA

Este capítulo descreve os procedimentos metodológicos adotados para o desenvolvimento deste trabalho, com foco na refatoração do SGTCC. A abordagem utilizada é de natureza aplicada, uma vez que busca solucionar problemas reais identificados em um sistema em uso, e qualitativa, por envolver a análise estrutural do código e a aplicação de boas práticas de engenharia de software.

3.1 Caracterização da Pesquisa

Este trabalho caracteriza-se como de natureza aplicada, com abordagem predominantemente qualitativa, sendo conduzida por meio de um estudo de caso, conforme a classificação proposta por Gil (2010). A natureza aplicada justifica-se pelo foco na resolução de problemas reais identificados na base de código do SGTCC, em consonância com o objetivo de melhorar a qualidade e a manutenibilidade do sistema. A abordagem qualitativa fundamenta-se na análise estrutural do código e na interpretação dos impactos das refatorações propostas, considerando atributos como organização, legibilidade, coesão e acoplamento. Adicionalmente, serão utilizados indicadores quantitativos como apoio à avaliação dos resultados, empregados como meio de mensuração complementar da qualidade do software. O estudo de caso será adotado por possibilitar a análise aprofundada de um sistema real, permitindo identificar, analisar e tratar problemas estruturais.

3.2 Visão Geral do Processo

O desenvolvimento deste trabalho é conduzido de forma incremental e iterativa, organizado em seis etapas sequenciais que abrangem desde o diagnóstico técnico até a validação dos resultados.

3.2.1 Etapa 1: Análise da Base de Código

Esta fase inicial compreende uma análise exploratória e sistemática do código-fonte do SGTCC. O objetivo é realizar um diagnóstico detalhado para identificar gargalos de manutenção e problemas estruturais. A análise é pautada na leitura técnica e na detecção de `code smells`, priorizando a identificação de:

- **Duplicação de código:** trechos redundantes que dificultam a evolução;
- **Acoplamento e Coesão:** componentes com excesso de dependências ou responsabilidades difusas;

- **Inconsistências:** falhas na padronização da representação de dados.

3.2.2 Etapa 2: Definição das Estratégias de Refatoração

Com o diagnóstico em mãos, são estabelecidas as diretrizes de intervenção baseadas nos princípios de *Clean Code* e *Clean Architecture*. A estratégia de refatoração não visa apenas a correção estética, mas a melhoria estrutural do software. A priorização das tarefas é definida pelo binômio impacto-complexidade, avaliando o quão crítica é a funcionalidade para o sistema e o risco técnico da alteração.

3.2.3 Etapa 3: Implementação das Refatorações

Nesta etapa, o plano é executado de forma incremental. As intervenções focam na modernização da arquitetura interna, destacando-se:

- A padronização de domínios através do uso de `enums`;
- A introdução de camadas de `services` para desacoplar a lógica de negócio dos `controllers`;
- A aplicação de técnicas clássicas de refatoração, como a extração de métodos e a eliminação de literais "mágicos".

3.2.4 Etapa 4: Reestruturação dos Testes

Paralelamente à melhoria do código, a infraestrutura de testes é revitalizada. O foco recai sobre a refatoração das `factories` de testes e a reorganização dos testes automatizados. Busca-se garantir que a suíte de testes seja não apenas um mecanismo de verificação, mas também uma documentação viva do sistema, com alta legibilidade e reaproveitamento de código.

3.2.5 Etapa 5: Validação das Alterações

Para assegurar que as melhorias estruturais não introduziram regressões funcionais, o sistema passa por um rigoroso processo de validação. Esta fase utiliza a suíte de testes reestruturada e novos cenários de teste criados especificamente para as áreas refatoradas. A metodologia adota preceitos do desenvolvimento orientado a testes (TDD), conforme preconizado por (BECK, 2003), garantindo a preservação do comportamento externo do SGTCC.

3.2.6 Etapa 6: Avaliação dos Resultados

Por fim, o impacto das alterações é mensurado comparando-se o estado *as-is* (inicial) com o *to-be* (final). A eficácia da refatoração é validada por meio de métricas objetivas, tais como o percentual de redução de código duplicado, a melhoria nos índices de acoplamento e o aumento da cobertura de testes, fornecendo evidências quantitativas e qualitativas da evolução do software.

3.3 Ferramentas de Análise e Coleta de Métricas

Para apoiar a análise quantitativa da qualidade do código e complementar a avaliação qualitativa conduzida neste trabalho, serão utilizadas ferramentas consolidadas do ecossistema Ruby, amplamente empregadas na prática de engenharia de software.

Dentre as ferramentas adotadas, destaca-se o uso do **RubyCritic**, responsável por fornecer uma visão integrada da qualidade do código. Essa ferramenta agrega métricas provenientes de diferentes analisadores estáticos, permitindo avaliar aspectos como complexidade, duplicação de código e presença de *code smells*.

O RubyCritic integra, entre outras, as seguintes ferramentas:

- **Reek**: utilizada para identificação de `code smells`, permitindo detectar problemas estruturais como responsabilidades excessivas, métodos longos e uso inadequado de abstrações;
- **Flog**: empregada para análise da complexidade dos métodos, fornecendo métricas que auxiliam na identificação de trechos de código com alta dificuldade de compreensão e manutenção;
- **Flay**: utilizada para detecção de duplicação de código, contribuindo para a identificação de padrões repetitivos e oportunidades de refatoração;
- **SimpleCov**: utilizada para mensurar a cobertura de testes automatizados, permitindo avaliar o percentual de código exercitado pelos testes;
- **RuboCop**: utilizada para análise estática e verificação de conformidade com boas práticas e padrões de estilo de código.

A utilização dessas ferramentas permitirá a coleta de métricas objetivas antes e após a aplicação das refatorações, possibilitando a comparação entre os estados do sistema e a avaliação dos impactos das melhorias implementadas.

Ressalta-se que as métricas obtidas por meio dessas ferramentas serão utilizadas como apoio à análise qualitativa, não sendo empregadas com finalidade estatística inferencial, mas sim como indicadores complementares da qualidade do software.

3.4 Relação entre Objetivos, Métodos e Métricas

O Quadro 1 apresenta a relação entre os objetivos específicos, os procedimentos metodológicos adotados, as evidências esperadas e os indicadores de avaliação utilizados no trabalho.

Tabela 1 – Relação entre objetivos específicos, procedimentos metodológicos, evidências e indicadores de avaliação

Objetivo Específico	**Método / Procedimento**	**Evidência de Avaliação**	**Indicadores/Métricas**
Identificar e analisar problemas estruturais na base de código	Análise exploratória do código com identificação de `code smells`	Lista de problemas identificados e trechos analisados	Quantidade de `code smells` identificados
Padronizar a representação de dados por meio de `enums`	Refatoração da modelagem de dados com substituição de valores inconsistentes	Padronização observada no código	Número de `enums` criados e redução de valores duplicados
Reduzir o acoplamento entre componentes do sistema	Aplicação de separação de responsabilidades entre `models`, `controllers`, `concerns` e `services`	Melhoria na modularização do código	Redução de dependências diretas entre camadas
Refatorar as `factories` de testes	Reorganização das `factories` com uso de `traits` e eliminação de duplicações	Melhoria na reutilização e previsibilidade dos dados de teste	Redução de duplicação nas `factories`
Aprimorar os testes automatizados	Reestruturação dos testes e inclusão de novos cenários	Aumento da cobertura e legibilidade dos testes	Cobertura de testes e número de cenários adicionados
Reestruturar `controllers`, `models` e `concerns`	Redistribuição de responsabilidades, extraindo regras de negócio para `services` ou classes de domínio	Redução da complexidade em `controllers` e `models`	Número de métodos simplificados, redução de linhas de código e responsabilidades extraídas
Aplicar boas práticas de engenharia de software	Aplicação de princípios SOLID, Código Limpo e refatoração contínua	Melhoria geral da organização estrutural	Redução de duplicação, acoplamento e aumento de coesão

4 PROPOSTA DE REFATORAÇÃO

Este capítulo apresenta a proposta de refatoração da base de código do SGTCC, elaborada a partir dos problemas estruturais identificados na etapa de análise descrita na metodologia.

As intervenções propostas têm como objetivo melhorar a organização do código, reduzir o acoplamento entre componentes e aumentar a manutenibilidade do sistema, em conformidade com os objetivos definidos neste trabalho.

As propostas estão organizadas de acordo com os principais aspectos estruturais identificados, incluindo padronização de dados, reorganização de componentes, redução de duplicações e aprimoramento dos testes automatizados.

4.1 Padronização de dados

A solução proposta inclui a padronização da representação de dados no sistema, especialmente no uso de valores enumerados (`enums`), que atualmente apresentam inconsistências ao serem representados como valores numéricos e textuais em diferentes partes da aplicação. Essa padronização visa garantir maior consistência, reduzir ambiguidades e facilitar a manutenção do código.

Como exemplo, observa-se que o campo relacionado ao semestre é representado de formas distintas no sistema, sendo utilizado em alguns trechos como valor numérico (1 e 2) e, em outros, como valor textual ("*one*"e "*two*"). Essa inconsistência pode gerar ambiguidades e dificultar a manutenção do código.

Exemplo numérico

```
@orientations =
↪  Orientation.includes(:calendars).where(calendars: { tcc: 2
↪  })
                                    .order('orientations.created_at
                                    ↪  DESC')
```

Exemplo textual

```
def calendar_tcc_url
  return responsible_calendars_tcc_one_path if @calendar.tcc
  ↪  == 'one'

  responsible_calendars_tcc_two_path
end
```

Figura 2 – Representações distintas do campo `tcc` no código do sistema

A padronização proposta consiste na adoção de uma única forma de representação para esses valores ao longo de toda a aplicação. Dessa forma, busca-se reduzir ambiguidades na interpretação dos dados, melhorar a integridade das informações e evitar inconsistências entre diferentes camadas do sistema, como `models`, `services` e `controllers`. Além disso, a padronização contribui para tornar as consultas e validações mais claras, reduzindo a ocorrência de erros relacionados ao uso de valores distintos para representar o mesmo conceito.

4.2 Reestruturação de `controllers`

Os `controllers` serão reestruturados com o objetivo de promover a separação de responsabilidades, removendo regras de negócio e delegando essas responsabilidades para camadas apropriadas. Essa abordagem contribui para a redução do acoplamento e facilita tanto a manutenção quanto a testabilidade do sistema.

Como forma de solucionar esse problema, propõe-se a introdução de uma camada intermediária denominada `services`, responsável por centralizar regras de negócio que não pertencem diretamente aos controllers ou `models`. Essa abordagem contribui para a organização do código, promovendo maior reutilização e facilitando a realização de testes.

Dessa forma, os controllers passam a atuar apenas como intermediadores entre a requisição e a execução da lógica de negócio, delegando essas responsabilidades aos services. Os models, por sua vez, permanecem responsáveis pela persistência e manipulação dos da-

dos. Essa separação permite reduzir o acoplamento entre componentes e torna o sistema mais modular e flexível a mudanças futuras.

Essa abordagem está alinhada a princípios de arquitetura limpa, que preconizam a separação de responsabilidades entre as camadas da aplicação.

Como exemplo, observa-se que o método `tcc_one` do A refatoração proposta consiste na extração dessa lógica para uma camada de `service`, permitindo que o `controller` atue apenas como intermediador da requisição. A seguir, apresenta-se um exemplo simplificado dessa transformação. concentra a construção de consultas complexas e a aplicação de filtros diretamente na camada de apresentação, caracterizando a presença de regras de negócio no A refatoração proposta consiste na extração dessa lógica para uma camada de `service`, permitindo que o `controller` atue apenas como intermediador da requisição. A seguir, apresenta-se um exemplo simplificado dessa transformação..

A refatoração proposta consiste na extração dessa lógica para uma camada de `service`, permitindo que o `controller` atue apenas como intermediador da requisição. A seguir, apresenta-se um exemplo simplificado dessa transformação.

Antes da refatoração

```
  def tcc_one
    @orientations = Orientation.joins(:calendars)
                               .group('orientations.id')
                               .having('COUNT(DISTINCT
                               ↪  calendars.tcc) = 1
                                       AND MIN(calendars.tcc)
↪  = 1')
                               .order('orientations.created_at
                               ↪  DESC')

    @orientations = @orientations.where(status:
    ↪  params[:status]) if params[:status].present?
    @orientations =
    ↪  @orientations.search(params[:term]).page(params[:page])

    @search_url = responsible_orientations_search_tcc_one_path

    render :index
  end
```

Após a refatoração

```
def tcc_one
  @orientations = OrientationSearchService.tcc_one(params)

  @search_url = responsible_orientations_search_tcc_one_path

  render :index
end
```

Figura 3 – Comparação entre a implementação antes e após a refatoração do método `tcc_one`

Com essa abordagem, a lógica de negócio torna-se mais reutilizável e testável de forma isolada, contribuindo para a redução do acoplamento e para a melhoria da organização do sistema.

4.3 Refatoração de `factories`

As `factories` utilizadas nos testes automatizados serão refatoradas com o objetivo de reduzir duplicações e melhorar a reutilização de código. Observa-se que a implementação atual apresenta múltiplas variações de `factories` para um mesmo modelo, com lógica repe-

tida e forte acoplamento a regras de negócio, o que dificulta a manutenção e a compreensão dos testes.

Como exemplo, a `factory` de calendário possui diversas definições específicas, como `current_calendar_tcc_one`, `next_calendar_tcc_two`, entre outras, contendo lógica condicional duplicada para cálculo de ano e semestre.

Antes da refatoração (trecho simplificado)

```
factory :current_calendar_tcc_one do
  tcc      { :one }
  year     { Calendar.current_year }
  semester { { 1 => :one, 2 => :two
  ↪  }[Calendar.current_semester] }
end

factory :next_calendar_tcc_one do
  tcc { :one }

  year do
    base = Calendar.find_by(...)
    base.semester.to_i == 1 ? base.year.to_i : base.year.to_i +
    ↪  1
  end

  semester do
    base = Calendar.find_by(...)
    base.semester.to_i == 1 ? :two : :one
  end
end
```

Figura 4 – Exemplo de duplicação de lógica nas `factories` antes da refatoração

Essa abordagem resulta em duplicação de lógica, dificuldade de manutenção e menor flexibilidade na criação de cenários de teste.

Como forma de melhoria, foi adotado o uso de *traits*, permitindo a composição de comportamentos reutilizáveis e a redução da quantidade de `factories` específicas.

Após a refatoração

```
factory :calendar do
  year { Calendar.current_year }
  semester { Calendar.current_semester }

  trait :tcc_one do
    tcc { Calendar.tccs.values.first }
  end

  trait :tcc_two do
    tcc { Calendar.tccs.values.last }
  end

  trait :current do
    year { Calendar.current_year }
    semester { Calendar.current_semester }
  end

  trait :previous do
    year { Calendar.current_semester == 2 ?
    ↪  Calendar.current_year : Calendar.current_year - 1 }
    semester { Calendar.current_semester == 1 ? 2 : 1 }
  end

  trait :next do
    year { Calendar.current_semester == 1 ?
    ↪  Calendar.current_year : Calendar.current_year + 1 }
    semester { Calendar.current_semester == 1 ? 2 : 1 }
  end
end
```

Figura 5 – Refatoração das `factories` utilizando *traits*

Com essa abordagem, a criação de cenários de teste torna-se mais simples e expressiva, permitindo a composição de diferentes estados de forma declarativa, como nos exemplos a seguir:

Exemplos de uso

```
create(:calendar, :current, :tcc_one)
create(:calendar, :previous, :tcc_two)
create(:calendar, :next, :tcc_one)
```

Figura 6 – Exemplos de composição de `traits` nas `factories`

Essa refatoração contribui para a redução da duplicação de código, melhora a legibilidade dos testes e facilita a manutenção da suíte de testes, além de promover maior flexibilidade na definição de cenários.

4.4 Desacoplamento e arquitetura

Como parte da proposta de refatoração, será realizada a introdução de uma camada de `services` e, quando necessário, de classes de domínio em `domains`, com o objetivo de centralizar regras de negócio que atualmente se encontram distribuídas entre `controllers`, `models` e `concerns`. No contexto de aplicações Rails, um `service` pode ser entendido como uma classe responsável por encapsular operações ou fluxos de negócio que não pertencem diretamente a um `controller` ou a um `model`. Essa reorganização busca reduzir o acoplamento entre componentes e promover uma separação mais clara de responsabilidades.

Atualmente, observa-se que alguns `controllers` concentram lógica relacionada tanto ao fluxo de requisições quanto às regras de negócio, o que dificulta a manutenção e a reutilização de código. Da mesma forma, alguns `models` apresentam acúmulo de responsabilidades que extrapolam a persistência e manipulação de dados, comprometendo sua coesão. Além disso, serão analisados os `concerns` existentes, especialmente nos casos em que esses módulos concentram responsabilidades que poderiam estar em camadas mais adequadas, como `services` ou classes de domínio em `domains`.

Com a adoção da camada de `services` e de classes de domínio em `domains`, as regras de negócio passam a ser encapsuladas em componentes específicos, permitindo que:

(i) `controllers` atuem exclusivamente como intermediadores das requisições;

(ii) `models` permaneçam responsáveis pela persistência e representação dos dados;

(iii) `services` concentrem fluxos e operações de negócio de forma reutilizável e testável;

(iv) classes em `domains` representem regras específicas do domínio da aplicação, reduzindo o acúmulo de responsabilidades nos `models`;

(v) `concerns` sejam utilizados apenas quando houver responsabilidade compartilhada adequada entre componentes.

Além disso, essa abordagem favorece a redução de dependências diretas entre camadas, contribuindo para um menor acoplamento e maior modularidade do sistema.

A proposta também está alinhada a princípios de arquitetura de software, como separação de responsabilidades e alta coesão, além de práticas associadas à Arquitetura Limpa, que incentivam a organização do sistema em camadas bem definidas e independentes.

Nos `models`, serão realizadas melhorias complementares com foco na remoção de lógicas duplicadas e na simplificação de responsabilidades. Sempre que identificado código que não pertence diretamente à responsabilidade da entidade, este será extraído para `services`, classes de domínio em `domains`, `concerns` mais adequados ou outros componentes apropriados.

Com essas mudanças, espera-se tornar o sistema mais organizado, flexível e preparado para evoluções futuras, facilitando tanto a manutenção quanto a implementação de novas funcionalidades.

4.5 Reestruturação de `components` e da classe `Calendar`

Além das melhorias relacionadas aos `controllers`, `models`, `services` e `concerns`, apresentadas anteriormente, a proposta também contempla a análise de outros pontos da aplicação que impactam sua organização e manutenção. Entre eles, destacam-se a extração de `partials` para `components` e a reestruturação da classe `Calendar`.

A extração de `partials` para `components` será considerada nos casos em que trechos de interface apresentem repetição estrutural, lógica de apresentação ou necessidade de reutilização em diferentes telas. Embora os `partials` sejam úteis para dividir partes da camada de visualização, os `components` permitem encapsular melhor a estrutura visual e o comportamento associado, favorecendo maior organização e reutilização da camada de apresentação.

Outro ponto importante envolve a classe `Calendar`, que possui papel central no sistema por estar relacionada à organização dos períodos acadêmicos, atividades e vínculos com os Trabalhos de Conclusão de Curso. Como essa classe concentra regras relacionadas a ano, semestre, datas de início e fim e identificação de períodos atuais, sua complexidade tende a aumentar conforme novas funcionalidades são adicionadas.

Dessa forma, pretende-se revisar as responsabilidades da classe `Calendar`, padronizar a representação dos dados relacionados ao calendário e, quando necessário, extrair regras específicas para `services` ou classes de domínio em `domains`. Com isso, espera-se tornar essa parte do sistema mais coesa, compreensível e preparada para futuras evoluções.

4.6 Melhoria dos testes

Os testes automatizados serão aprimorados por meio da reorganização de sua estrutura, redução de redundâncias e adoção de boas práticas, como o uso de `build` em substituição ao `create` quando não há necessidade de persistência. Além disso, serão incluídos novos cenários de teste, incluindo casos negativos, com o objetivo de aumentar a confiabilidade do sistema.

A reorganização também envolverá a estrutura de diretórios dos testes, buscando separar arquivos extensos em unidades menores e mais específicas, de acordo com a responsabilidade de cada conjunto de testes. Dessa forma, testes relacionados a validações, associações, buscas, `callbacks`, regras de negócio e métodos auxiliares poderão ser organizados em arquivos distintos, facilitando a localização, leitura e manutenção dos cenários.

Como exemplo, observa-se a repetição de testes semelhantes para diferentes atributos de busca:

Antes da refatoração

```
it 'returns academic by name' do
  results = described_class.search(academic.name)
  expect(results.first.name).to eq(academic.name)
end

it 'returns academic by email' do
  results = described_class.search(academic.email)
  expect(results.first.email).to eq(academic.email)
end
```

Figura 7 – Exemplo de duplicação de testes para diferentes atributos

Essa abordagem apresenta duplicação de lógica nos testes, tornando a manutenção mais trabalhosa e aumentando a quantidade de código necessário para cobrir cenários semelhantes.

Como forma de melhoria, pode-se utilizar estruturas iterativas para reduzir a repetição e tornar os testes mais concisos e expressivos.

Após a refatoração

```
[:name, :email, :ra].each do |attribute|
  it "returns academic by #{attribute}" do
    results =
    ↪  described_class.search(academic.public_send(attribute))

    expect(results).to include(academic)
  end
end
```

Figura 8 – Refatoração de testes utilizando estrutura iterativa

Essa refatoração reduz a duplicação de código, melhora a legibilidade dos testes e facilita a inclusão de novos cenários, bastando adicionar novos atributos à estrutura iterativa.

Além da reorganização dos testes, também será utilizada a gem `shoulda-matchers`, que permite escrever testes mais concisos para validações e associações em aplicações Rails. Essa ferramenta contribui para reduzir a repetição de código em cenários comuns, como validação de presença de atributos, unicidade e relacionamentos entre `models`, tornando os testes mais legíveis e diretos.

Exemplo de uso da gem `shoulda-matchers`

```
it { is_expected.to validate_presence_of(:name) }
it { is_expected.to validate_presence_of(:email) }
it { is_expected.to
↪  validate_uniqueness_of(:email).case_insensitive }

it { is_expected.to have_many(:orientations) }
it { is_expected.to
↪  have_many(:examination_boards).through(:orientations) }
```

Figura 9 – Exemplo de testes concisos utilizando a gem `shoulda-matchers`

Com essas alterações, espera-se melhorar o isolamento dos testes, reduzindo dependências desnecessárias entre cenários e promovendo maior previsibilidade nos resultados. Es-

sas melhorias contribuem para um conjunto de testes mais robusto, confiável e de fácil manutenção.

5 RESULTADOS PRELIMINARES

Este capítulo descreve os resultados iniciais obtidos a partir das atividades de refatoração e evolução do sistema SGTCC. As ações focaram na organização da base de dados e na reestruturação de componentes centrais para aumentar a flexibilidade da aplicação.

5.1 Integração de Dados de Períodos Anteriores

A primeira etapa do trabalho consistiu no levantamento e organização de dados relativos a TCCs de semestres passados que ainda não constavam no sistema.

A centralização dessas informações visa consolidar o SGTCC como a base única de registros do curso. Para viabilizar essa integração, foram desenvolvidos `scripts` de migração que estruturaram os dados dispersos e os inseriram de forma padronizada no banco de dados atual. Esta ação permite que o sistema forneça estatísticas mais precisas sobre a produção acadêmica ao longo do tempo.

5.2 Alteração do Modelo de Calendário Acadêmico

Identificou-se que o modelo de calendário original apresentava limitações de uso. A representação baseada estritamente em ano e semestre não suportava cenários comuns, como períodos letivos com datas especiais ou que ultrapassam o ano civil.

5.2.1 Flexibilização do Período Letivo

Para solucionar essa limitação, o modelo foi reestruturado para incluir campos de data inicial e data final. A Tabela 2 apresenta a diferença entre a estrutura anterior e a nova implementação.

Tabela 2 – Comparação entre as estruturas do calendário.

Modelo Anterior	Modelo Atualizado
Ano: 2024	Ano: 2024
Semestre: 1	Semestre: 1
	Data Inicial: 01-03-2024
	Data Final: 01-07-2024

Essa alteração permite que o sistema execute validações automáticas de prazos e identifique o período ativo com base no intervalo de datas configurado.

5.2.2 Ajustes de Consistência e Testes

Para garantir que a mudança não afetasse o funcionamento das funcionalidades existentes, foram realizadas as seguintes tarefas:

- **Migração de registros:** Atualização dos dados antigos para o novo formato de datas.
- **Atualização de testes:** Adaptação das suítes de testes automatizados e das *factories* para contemplar as novas regras de validação.

5.3 Análise das Dificuldades Encontradas

Durante a reestruturação do calendário, foram observados pontos de melhoria na arquitetura interna do sistema. A presença de alta dependência entre módulos (acoplamento) e a falta de padronização em valores enumerados dificultaram a implementação inicial.

Essas observações confirmam a relevância das refatorações propostas neste trabalho, evidenciando que a melhoria na organização do código é essencial para que o sistema possa evoluir e receber novas funcionalidades de forma sustentável.

5.4 Impactos Gerados

As modificações realizadas trouxeram benefícios imediatos para o SGTCC:

- **Precisão:** O calendário agora reflete fielmente os períodos acadêmicos reais.
- **Centralização:** A inclusão de dados anteriores unificou o repositório de trabalhos.
- **Confiabilidade:** A atualização dos testes automatizados garante maior segurança para futuras manutenções.

6 CONSIDERAÇÕES FINAIS

Este trabalho tem como objetivo propor a refatoração do SGTCC, com foco na melhoria da organização do código, na redução da complexidade estrutural e no aumento da manutenibilidade, sem alteração de suas funcionalidades.

A partir da análise da evolução do sistema e da identificação de problemas estruturais na base de código, como inconsistências na representação de dados, duplicação de lógica, alto acoplamento entre componentes e concentração de responsabilidades em `controllers`, `models` e `concerns`, foi possível definir um conjunto de estratégias de refatoração fundamentadas em boas práticas de engenharia de software.

A proposta contempla a padronização de dados, a reestruturação de componentes da aplicação, a introdução de uma camada de `services`, a análise de classes de domínio em `domains`, a reorganização de `concerns` e a melhoria na estrutura dos testes automatizados. Essas ações buscam promover uma melhor separação de responsabilidades e tornar o sistema mais modular, compreensível e preparado para futuras evoluções.

Além disso, foram identificadas oportunidades de melhoria relacionadas ao modelo de calendário, à integração de dados históricos e à organização da camada de apresentação, especialmente por meio da possível extração de `partials` para `components`. Esses pontos evidenciam que a refatoração proposta não se limita à reorganização interna do código, mas também contribui para a sustentabilidade e evolução contínua da aplicação.

Espera-se que, com a implementação das refatorações propostas, o SGTCC se torne mais preparado para manutenção e evolução, reduzindo a complexidade do código, facilitando a adaptação a novas demandas e diminuindo o risco de introdução de erros em alterações futuras.

No entanto, por se tratar de uma proposta a ser desenvolvida em um curto espaço de tempo, este trabalho apresenta limitações quanto à abrangência das refatorações. Dessa forma, não se pretende contemplar todos os módulos do sistema, mas priorizar os trechos mais relevantes e com maior impacto na organização do código. Além disso, durante a implementação, podem surgir problemas inesperados relacionados a dependências entre componentes, efeitos colaterais em funcionalidades existentes e necessidade de ajustes em testes automatizados. Esses fatores podem exigir adaptações no planejamento inicial e reforçam a importância de uma execução incremental, acompanhada de validação contínua.

Dessa forma, reforça-se a importância da refatoração como prática essencial para a melhoria contínua da qualidade de software, especialmente em sistemas que evoluem ao longo do tempo e acumulam complexidade estrutural em decorrência de sucessivas alterações.

REFERÊNCIAS

BECK, K. **Test-Driven Development: By Example**. Boston: Addison-Wesley, 2003.

CARDOZO, N. H. F. **Redesign da interface gráfica do sistema de gestão de trabalho de conclusão de curso**. 2025. Trabalho de Conclusão de Curso em andamento (Tecnologia em Sistemas para Internet) – Universidade Tecnológica Federal do Paraná, Câmpus Guarapuava. Disponível em: https://tcc.tsi.pro.br/uploads/academic_activity/pdf/406/GP_COINT_2025_1_NESTOR_HUGO_FERREIRA_CARDOZO_PROJETO.pdf.

FERREIRA, D. **Desenvolvimento de um sistema para o gerenciamento do processo de trabalho de conclusão de curso do curso de Tecnologia em Sistemas para Internet da UTFPR Câmpus Guarapuava**. 2015. Trabalho de Conclusão de Curso (Tecnologia em Sistemas para Internet) – Universidade Tecnológica Federal do Paraná, Câmpus Guarapuava.

FOWLER, M. **Refactoring: Improving the Design of Existing Code**. 2. ed. [*S.l.*]: Addison-Wesley, 2019.

GIL, A. C. **Como elaborar projetos de pesquisa**. [*S.l.*]: Atlas, 2010.

LIMA, A. C. **Projeto e implementação de interface baseada na experiência do usuário para um sistema de gerenciamento de trabalho de conclusão de curso**. 2023. Trabalho de Conclusão de Curso (Tecnologia em Sistemas para Internet) – Universidade Tecnológica Federal do Paraná, Câmpus Guarapuava. Disponível em: https://tcc.tsi.pro.br/uploads/academic_activity/pdf/227/GP_COINT_2023_2_AMANDA_CAROLYNE_DE_LIMA_MONOGRAFIA.pdf.

LUZ, G. S. **Atualização do framework rails para garantia da evolução do sitema de gestão de TCC**. 2025. Trabalho de Conclusão de Curso em andamento (Tecnologia em Sistemas para Internet) – Universidade Tecnológica Federal do Paraná, Câmpus Guarapuava. Disponível em: https://tcc.tsi.pro.br/uploads/academic_activity/pdf/328/GP_COINT_2024_2_GUILHERME_STACIAKI_DA_LUZ_PROJETO.pdf.

MARTIN, R. C. **Clean Code: A Handbook of Agile Software Craftsmanship**. Upper Saddle River, NJ: Prentice Hall, 2009.

MARTIN, R. C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. Boston, MA: Prentice Hall, 2017.

ROCHA, R. F. **Estratégia de notificações no sistema de gestão de TCC do curso de Sitemas para Internet: Definição e implementação**. 2025. Trabalho de Conclusão de Curso em andamento (Tecnologia em Sistemas para Internet) – Universidade Tecnológica Federal do Paraná, Câmpus Guarapuava. Disponível em: https://tcc.tsi.pro.br/uploads/academic_activity/pdf/404/GP_COINT_2025_1_RAUL_FERREIRA_DA_ROCHA_PROJETO.pdf.

SILVA, R. G. A. **Aperfeiçoamento do sistema de gestão de processos de trabalho de conclusão de curso de Tecnologia em Sistemas para Internet da UTFPR Câmpus Guarapuava**. 2019. Trabalho de Conclusão de Curso (Tecnologia em Sistemas para Internet) – Universidade Tecnológica Federal do Paraná, Câmpus Guarapuava. Disponível em: https://tcc.tsi.pro.br/uploads/academic_activity/pdf/20/GP_COINT_2019_2_RENAN_GABRIEL_ALMEIDA_SILVA_MONOGRAFIA.pdf.

SOMERVILLE, I. **Software Engineering**. 10. ed. [*S.l.*]: Pearson, 2019.