v7.n8.2026

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

REESTRUTURAÇÃO E MODERNIZAÇÃO DO SISTEMA PLANKGYM: UMA ABORDAGEM PRÁTICA DE REFATORAÇÃO E SEGURANÇA EM SISTEMAS WEB

RESTRUCTURING AND MODERNIZATION OF THE PLANKGYM SYSTEM: A PRACTICAL APPROACH TO REFACTORING AND SECURITY IN WEB SYSTEMS

REESTRUCTURACIÓN Y MODERNIZACIÓN DEL SISTEMA PLANKGYM: UN ENFOQUE PRÁCTICO DE REFACTORIZACIÓN Y SEGURIDAD EN SISTEMAS WEB

Edward Lopes Rodrigues de Candia[1], Gabriel do Amaral Abreu[1], Olavo Olimpio de Matos Junior[2]

e788841
https://doi.org/10.47820/recima21.v7i8.8841

PUBLICADO: 08/2026

RESUMO

A transformação digital em micro e pequenas empresas impõe desafios significativos quanto à segurança da informação e manutenção de *software*, sobretudo em sistemas desenvolvidos por equipes reduzidas e sem padronização técnica. Este artigo apresenta um estudo de caso prático sobre a análise, refatoração e modernização do PlankGYM, um sistema *web* para gestão de academias. O artigo demonstra como uma aplicação que apresentava hibridismo de linguagens, ausência de tratamento centralizado de erros e vulnerabilidades classificadas nas categorias A01, A03 e A05 do OWASP *Top Ten* foi transformada em uma arquitetura limpa, tipada e segura. A partir do diagnóstico por inspeção estática de código, foram aplicadas práticas de *Clean Code* e mecanismos de segurança, incluindo autenticação por JWT, controle de acesso baseado em funções (RBAC) em dois níveis e validação de entrada. Os resultados demonstram que seis das dez categorias do OWASP *Top Ten* 2021 foram diretamente mitigadas, a totalidade da lógica de servidor passou a ser coberta por TypeScript e a duplicação identificada nas operações de banco de dados foi eliminada, sem alteração das regras de negócio já consolidadas. O estudo contribui com um roteiro replicável de refatoração para sistemas *web* de pequeno porte desenvolvidos em contextos acadêmicos ou por equipes reduzidas.

PALAVRAS-CHAVE: Refatoração. Código Limpo. Engenharia de *Software*. OWASP. Segurança da Informação.

ABSTRACT

Digital transformation in micro and small businesses poses significant challenges regarding information security and software maintenance, especially in systems developed by small teams without technical standardization. This article presents a practical case study on the analysis, refactoring, and modernization of PlankGYM, a web-based system for gym management. The article demonstrates how an application that exhibited hybrid languages, lack of centralized error handling, and vulnerabilities classified in categories A01, A03, and A05 of the OWASP Top Ten was transformed into a clean, typed, and secure architecture. Based on a static code inspection diagnosis, Clean Code practices and security mechanisms were applied, including JWT authentication, role-based access control (RBAC) at two levels, and input validation. The results show that six of the ten OWASP Top Ten 2021 categories were directly mitigated, all server logic was covered by TypeScript, and the duplication identified in database operations was eliminated, without altering the already established business rules. This study contributes a replicable refactoring roadmap for small-scale web systems developed in academic contexts or by small teams.

[1] Instituto Federal de Educação, Ciência e Tecnologia de São Paulo (IFSP), Graduando em Tecnologia em Análise e Desenvolvimento de Sistemas.
[2] Instituto Federal de Educação, Ciência e Tecnologia de São Paulo (IFSP), Mestre em Informática.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

KEYWORDS: *Refactoring. Clean Code. Software Engineering. OWASP. Information Security.*

RESUMEN
La transformación digital en microempresas y pequeñas empresas plantea importantes desafíos en materia de seguridad de la información y mantenimiento de software, especialmente en sistemas desarrollados por equipos reducidos sin estandarización técnica. Este artículo presenta un estudio de caso práctico sobre el análisis, la refactorización y la modernización de PlankGYM, un sistema web para la gestión de gimnasios. El artículo demuestra cómo una aplicación que presentaba lenguajes híbridos, falta de gestión centralizada de errores y vulnerabilidades clasificadas en las categorías A01, A03 y A05 del OWASP Top Ten se transformó en una arquitectura limpia, tipada y segura. A partir de un diagnóstico de inspección estática del código, se aplicaron prácticas de Código Limpio y mecanismos de seguridad, incluyendo autenticación JWT y control de acceso basado en roles (RBAC) en dos niveles y validación de entrada. Los resultados muestran que seis de las diez categorías del OWASP Top Ten 2021 se mitigaron directamente, toda la lógica del servidor se cubrió con TypeScript y se eliminó la duplicación identificada en las operaciones de la base de datos, sin alterar las reglas de negocio ya establecidas. Este estudio aporta una hoja de ruta de refactorización replicable para sistemas web a pequeña escala desarrollados en contextos académicos o por equipos reducidos.

PALABRAS CLAVE*: Refactorización. Código limpio. Ingeniería de Software. OWASP. Seguridad de la información.*

INTRODUÇÃO

A crescente adoção de sistemas de informação por micro e pequenas empresas impõe desafios que vão além da simples disponibilidade funcional do *software*. Em diversos setores, como o de gestão de academias, nos quais a rotina operacional depende de cadastros de alunos, controle de pagamentos, organização de treinos e autenticação de usuários, a qualidade técnica do sistema subjacente determina diretamente a confiabilidade do serviço prestado. Nesse contexto, aplicações desenvolvidas de forma incremental e sem padronização acumulam o que a literatura de engenharia de *software* denomina dívida técnica, um conjunto de decisões de curto prazo que comprometem a manutenibilidade, a escalabilidade e a segurança do sistema a longo prazo (FOWLER, 1999; MARTIN, 2008).

A motivação para o desenvolvimento deste estudo é verificar de que forma a aplicação sistemática de práticas de refatoração, boas práticas de engenharia de *software* e endurecimento de segurança é capaz de transformar um sistema *web* com uma dívida técnica notória em uma arquitetura limpa, segura e sustentável, sem comprometer as regras de negócio que já estão consolidadas. Esse problema é materializado no sistema “PlankGYM”, uma aplicação *web* para gestão de academias que, em seu estado original, apresentava hibridismo de linguagens JavaScript e TypeScript no mesmo projeto, ausência de tratamento unificado de erros, rotas de autenticação sem hierarquia de autorização e consultas ao banco de dados sem parametrização,

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

configurando vulnerabilidades classificadas nas categorias A01, A03 e A05 do OWASP *Top Ten* (OWASP FOUNDATION, 2021).

O estudo baseia-se em dois aspectos que se complementam. O primeiro é prático: sistemas de gestão para pequenas empresas frequentemente são desenvolvidos por equipes reduzidas, em prazos exíguos, com foco na entrega funcional em detrimento da qualidade estrutural. O resultado é um acúmulo silencioso de fragilidades que só se manifestam quando o sistema precisa ser estendido, sustentado por novos membros ou quando sofre uma tentativa de exploração. O segundo aspecto é acadêmico: embora a literatura sobre refatoração e segurança de aplicações *web* seja vasta, há escassez de estudos de caso práticos que demonstrem, com evidências, a aplicação conjunta dessas abordagens em um único sistema de pequeno porte. Este estudo preenche essa lacuna ao documentar um processo completo, desde o diagnóstico até a validação, em um sistema funcional desenvolvido como Trabalho de Conclusão de Curso em uma instituição de ensino superior.

Esta pesquisa consiste em apresentar de que maneira a adoção de boas práticas de *Clean Code*, aliada às normativas de segurança da OWASP, viabiliza a reestruturação de um sistema *web* legado para uma arquitetura limpa, segura e com tipagem robusta. Tal premissa é validada empiricamente por intermédio de um estudo de caso no sistema PlankGYM. Para o alcance dessa finalidade, foram estabelecidos os seguintes objetivos específicos: (i) mapeamento da arquitetura original do sistema e identificação de suas inconsistências técnicas e vulnerabilidades de segurança por meio de inspeção estática de código; (ii) aplicação de práticas de refatoração em camadas com centralização do tratamento de erros e adoção de tipagem estática via *TypeScript*; (iii) implementação de mecanismos de segurança voltados à mitigação das vulnerabilidades identificadas, incluindo autenticação por JWT, controle de acesso baseado em funções (RBAC) e validação de entradas com *Joi*; e (iv) comparação do estado anterior e posterior do sistema em requisitos mensuráveis de qualidade e segurança.

1. REFERENCIAL TEÓRICO

Visando fornecer o embasamento técnico e científico para as escolhas de projeto assumidas durante a reestruturação do sistema PlankGYM, este referencial elucida os principais conceitos, princípios e tecnologias subjacentes ao desenvolvimento. A abordagem estruturou-se em três eixos temáticos complementares: (i) arquitetura de *software* e separação de responsabilidades; (ii) refatoração, dívida técnica e qualidade de código; e (iii) segurança em

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

aplicações *web*. Cada eixo é sustentado por literatura acadêmica e documentos normativos da área.

1.1. Arquitetura de *software* e separação de responsabilidades

O desenvolvimento de aplicações *web* modernas é frequentemente estruturado segundo o modelo Cliente-Servidor, no qual as responsabilidades de apresentação, lógica de negócio e persistência de dados são distribuídas em camadas distintas e comunicantes. Nesse modelo, o cliente, geralmente um navegador, realiza requisições a um servidor que processa a lógica de domínio e devolve respostas estruturadas, tipicamente no formato JSON sobre o protocolo HTTP (TANENBAUM; WETHERALL, 2011). Essa separação favorece a escalabilidade horizontal, a testabilidade independente de cada camada e a manutenção sem acoplamento entre interface e regras de negócio.

Essa organização reflete os princípios da *Clean Architecture* proposta por Martin (2017), segundo a qual as dependências entre módulos devem sempre apontar em direção às regras de negócio, isolando-as de *frameworks*, bancos de dados e detalhes de infraestrutura. Na prática, essa diretriz se traduz na separação entre a camada que é responsável exclusivamente pelo ciclo requisição e resposta HTTP e a lógica de acesso a dados, que permanece encapsulada, sem vazamento de detalhes de infraestrutura para as camadas superiores.

A adoção do TypeScript como linguagem primária do *backend* representa uma decisão arquitetural com impacto direto na qualidade do sistema. Bogner e Merkel (2022), em estudo comparativo sistemático envolvendo projetos *open source* em JavaScript e TypeScript, constataram que aplicações TypeScript apresentam menor densidade de defeitos e maior facilidade de manutenção, atribuindo esse resultado à capacidade da tipagem estática de detectar, em tempo de compilação, classes inteiras de erros que em JavaScript puro só seriam identificados em tempo de execução. A coexistência não deliberada de módulos tipados e não tipados em um mesmo projeto, por outro lado, tende a anular parcialmente esses benefícios, uma vez que o compilador deixa de exercer sua função de verificação estática sobre a totalidade da base de código.

1.2. Refatoração, dívida técnica e *Clean Code*

Refatoração é definida como o processo disciplinado de reestruturar o código-fonte existente sem alterar seu comportamento externo observável, com o objetivo de melhorar sua estrutura interna (FOWLER, 1999). Trata-se de uma prática central na engenharia de *software* moderna, especialmente em projetos que evoluem de forma incremental, nos quais a pressão

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

por entregas rápidas frequentemente resulta em soluções funcionais, mas estruturalmente frágeis.

Associado à refatoração, o conceito de dívida técnica descreve o custo implícito acumulado quando decisões de projeto de curto prazo comprometem a manutenção futura do sistema. Yanakiev *et al.* (2025) descrevem um processo estruturado de identificação, priorização e pagamento de dívida técnica arquitetural utilizando os princípios *SOLID* como guia, evidenciando redução mensurável de acoplamento entre módulos e melhora na cobertura de testes sem impacto nas regras de negócio.

Martin (2008) estabelece um conjunto de práticas para a escrita de código de qualidade: funções com responsabilidade única, nomes expressivos para variáveis e métodos, ausência de duplicação e tratamento consistente de erros. O princípio da responsabilidade única, principal ponto do que o autor denomina código limpo, preconiza que cada módulo deve ter apenas uma razão para mudar, o que, em termos práticos, orienta a separação entre os componentes responsáveis pela representação de um erro e aqueles responsáveis pelo seu tratamento e serialização em uma resposta.

Um segundo pilar do *Clean Code* é a eliminação de duplicação, formalizada no princípio *DRY* (*Don't Repeat Yourself*) por Hunt e Thomas (1999), segundo o qual toda peça de conhecimento deve ter uma representação única, não ambígua e autoritativa dentro de um sistema. Esse princípio se aplica tanto a projeções de consulta ao banco de dados quanto ao fluxo de controle dos *controllers*: Latte *et al.* (2019) identificam o tratamento de erros disperso por múltiplos pontos do código como um dos principais fatores de degradação da manutenibilidade em sistemas *web* de pequeno e médio porte, reforçando a importância de mecanismos que centralizam essa responsabilidade.

Por fim, Cabral *et al.* (2024) evidenciam que a aplicação dos princípios *SOLID* melhora a compreensão do código, aumenta a modularidade e simplifica a extensão de funcionalidades sem quebrar comportamentos existentes. São benefícios especialmente relevantes quando um sistema centraliza a categorização de erros e status em um único artefato tipado, garantindo consistência semântica sem a necessidade de valores fixos repetidos ao longo do código.

1.3. Segurança em aplicações *web*

A segurança de aplicações *web* é um campo em constante evolução, impulsionado pela crescente sofisticação dos ataques e pela ampla adoção de sistemas baseados no protocolo HTTP. A principal referência normativa da área é o OWASP *Top Ten* (*Open Web Application Security Project*), documento de conscientização que elenca as dez categorias de riscos mais

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

críticos em aplicações *web*, atualizado periodicamente com base em dados coletados da indústria (OWASP FOUNDATION, 2021).

1.3.1. Autenticação e controle de acesso

A autenticação por *JSON Web Tokens* (JWT) é amplamente adotada em aplicações *web stateless* por sua capacidade de transmitir identidade e permissões de forma assinada e verificável, sem necessidade de consulta ao banco de dados a cada requisição (JONES; BRADLEY; SAKIMURA, 2015). O mecanismo consiste na emissão de um *token* estruturado em três partes (*header, payload e signature*), que o cliente armazena e reenvia em cada requisição subsequente. A robustez desse modelo depende diretamente de práticas de gestão segura da chave de assinatura e de uma política de expiração adequada, de modo a limitar a janela de exposição em caso de comprometimento de um *token* válido.

Para o controle de autorização, o modelo *Role-Based Access Control* (RBAC) associa permissões a papéis, e não diretamente a usuários individuais, permitindo que o acesso a recursos seja concedido ou restrito de acordo com a função exercida por cada ator no sistema. Butt *et al.* (2023) demonstram que sistemas baseados em hierarquias de papéis bem definidas reduzem significativamente a superfície de ataque ao garantir que cada ator acesse apenas os recursos autorizados para sua função.

1.3.2. Proteção contra vulnerabilidades comuns

A injeção de SQL (categoria A03 do OWASP *Top Ten)* figura entre os riscos mais críticos listados pelo OWASP e decorre da construção dinâmica de comandos SQL a partir de entradas não confiáveis do usuário. A contramedida primária recomendada é o uso de *queries* parametrizadas, técnica que separa estruturalmente o código SQL dos dados fornecidos pelo usuário, inviabilizando a manipulação do comando por entradas maliciosas (OWASP *FOUNDATION*, 2021). De forma complementar, a validação de esquema das entradas antes de qualquer acesso à camada de persistência atua como uma primeira barreira de defesa, rejeitando dados malformados ou campos não previstos antes que alcancem a lógica de negócio.

No plano da configuração de protocolo HTTP, a categoria A05 *(Security Misconfiguration*) do OWASP *Top Ten* trata da ausência de cabeçalhos de segurança adequados e de políticas de acesso entre origens (*CORS*) mal definidas, que podem expor a aplicação a vetores como *clickjacking*, *MIME sniffing* e injeção de *scripts*. A separação formal entre configurações de ambiente de desenvolvimento e produção é apontada pelo OWASP como prática essencial para evitar que configurações permissivas de desenvolvimento sejam

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

inadvertidamente utilizadas em produção. Por fim, a categoria A09 (*Security Logging and Monitoring Failures*) trata da ausência de trilhas de auditoria: o registro sistemático de acessos é apontado como requisito para viabilizar a análise forense de incidentes e a detecção de padrões de acesso anômalos.

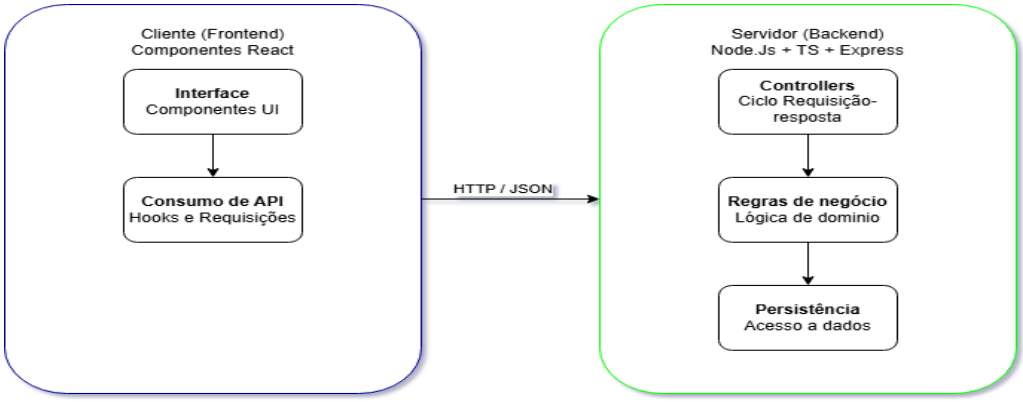
2. METODOLOGIA

O desenvolvimento e a reestruturação do PlankGYM foram conduzidos sob controle de versão distribuído por meio do *Git*, com repositório hospedado na plataforma *GitHub*. Essa escolha viabilizou o rastreamento incremental de cada intervenção realizada no sistema, desde alterações pontuais de configuração até refatorações estruturais de camadas inteiras, permitindo que o histórico de *commits* funcionasse como registro auditável do processo de modernização.

A evolução do esquema de banco de dados foi gerenciada por meio de migrações versionadas e incrementais, numeradas sequencialmente, localizadas no diretório *backend/src/migrations/.* Cada *migração* representa uma unidade atômica de alteração estrutural, como adição de colunas, criação de tabelas auxiliares e ajuste de *constraints*, executada de forma ordenada e sem dependência de intervenção manual no banco de dados de produção. Essa abordagem segue a prática recomendada por Fowler (1999) de tratar mudanças em banco de dados com o mesmo rigor e rastreabilidade aplicados ao código-fonte.

O ambiente de desenvolvimento foi constituído pelas seguintes tecnologias: *Node.js* 18 e *Express* 5 no *backend*; *React* 18 com *Vite* 4 no *frontend*; e *PostgreSQL* como sistema gerenciador de banco de dados relacional. A Figura 1 ilustra essa organização em camadas, com a separação entre cliente e servidor e a estrutura interna do *backend* detalhada nas seções seguintes.

Figura 1. Diagrama de camadas: arquitetura Cliente-Servidor



Fonte: Elaborada pelos autores (2026).

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

As dependências de segurança utilizadas incluem bibliotecas que auxiliam em diversos aspectos, como validação, tokenização, respostas HTTP e criptografia de entradas. As variáveis de ambiente sensíveis são gerenciadas por meio de arquivo *.env*, cujo modelo público *.env.example* está versionado no repositório, documentando as chaves necessárias sem expor valores reais. Essa prática endereça diretamente a categoria A05 (*Security Misconfiguration*) do OWASP *Top Ten,* ao garantir que configurações de produção nunca sejam acidentalmente *commitadas* no repositório.

A etapa de diagnóstico consistiu em uma inspeção estática do código-fonte do sistema em seu estado anterior à refatoração, conduzida por meio de análise manual do repositório. O processo seguiu dois eixos de avaliação paralelos e complementares: a identificação de débitos técnicos estruturais e o mapeamento de vulnerabilidades de segurança, tendo o OWASP Top Ten (OWASP FOUNDATION, 2021) como guia analítico.

O primeiro ponto crítico identificado foi o hibridismo de linguagens no mesmo projeto. A distribuição de linguagens registrada pelo próprio repositório (64,3% *JavaScript* e 35,1% *TypeScript* via *GitHub*) evidenciava uma base de código inconsistente, na qual módulos escritos em *TypeScript* coexistiam com arquivos *JavaScript* sem tipagem, sem que houvesse uma fronteira clara ou deliberada entre os dois. Essa coexistência impede que o compilador *TypeScript* exerça sua função de verificação estática sobre a totalidade do projeto, anulando parcialmente os benefícios da tipagem que motivaram sua adoção original (BOGNER; MERKEL, 2022).

O segundo ponto identificado foi a ausência de um mecanismo centralizado de tratamento de erros. No estado original, cada *controller* tratava (ou ignorava) suas próprias exceções de forma independente, sem padrão de resposta, sem categorização de tipos de erro e sem distinção entre erros operacionais esperados e falhas inesperadas do sistema. Latte *et al.* (2019) identificam exatamente essa dispersão como um dos principais fatores de degradação da manutenibilidade em sistemas *web*, pois qualquer alteração no comportamento de resposta em erros exige intervenção em múltiplos pontos do código simultaneamente.

O terceiro ponto foi a ausência do princípio *DRY* (*Don't Repeat Yourself*) em operações de banco de dados. A projeção SQL utilizada para consultas sobre alunos estava replicada individualmente em cada operação do *CRUD*, sem extração para uma constante reutilizável. Essa duplicação eleva o custo de manutenção: qualquer alteração no conjunto de campos retornados exige intervenção manual em múltiplos pontos do código, com risco de inconsistência entre operações (HUNT; THOMAS, 1999).

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

O mapeamento de vulnerabilidades identificou ocorrências em quatro categorias do OWASP *Top Ten*. A categoria A01 *Broken Access Control* foi evidenciada pela ausência de hierarquia de autorização nas rotas da API: não havia distinção de acesso entre os perfis de administrador, atendente e aluno, expondo operações administrativas sensíveis a qualquer usuário autenticado. A categoria A02 *Cryptographic Failures* foi identificada na ausência de *hashing* robusto de senhas em partes do sistema, com risco de armazenamento de credenciais em formato reversível. A categoria A03 *Injection* foi verificada pela presença de consultas SQL sem parametrização, vulneráveis à injeção de comandos arbitrários por meio de entradas do usuário. Por fim, a categoria A05 *Security Misconfiguration* foi constatada pela ausência de cabeçalhos HTTP de segurança, política de CORS irrestrita e ausência de separação formal entre configurações de desenvolvimento e produção.

Os resultados desta etapa de diagnóstico consolidaram o conjunto de intervenções a realizar, distribuídas entre refatoração estrutural e endurecimento de segurança.

Baseada nos pontos críticos identificados na inspeção estática, a intervenção estrutural foi conduzida em três frentes simultâneas: centralização do tratamento de erros, eliminação de duplicação em operações de banco de dados e padronização da separação de responsabilidades entre as camadas da aplicação. Cada intervenção foi implementada de forma incremental, preservando o comportamento externo do sistema, ou seja, sem alteração nas regras de negócio relativas ao domínio da academia.

A primeira e mais abrangente intervenção foi a criação do módulo *errorHandler.ts*, que consolidou em um único artefato toda a lógica de representação, classificação e serialização de erros da aplicação. O módulo define três elementos interdependentes: a classe *ApiError*, que estende a classe nativa *Error* do *JavaScript* e adiciona os campos *statusCode*, *isOperational* e *errorType*; o *enum ErrorTypes*, que mapeia sete categorias semânticas de erro HTTP para seus respectivos códigos de status; e o *errorMiddleware*, registrado como última camada do *pipeline Express*, responsável por interceptar qualquer exceção propagada via *next(err)* e serializar uma resposta JSON padronizada ao cliente. Essa separação entre a criação do erro (*ApiError*) e seu tratamento (*errorMiddleware*) é uma aplicação direta do princípio da responsabilidade única descrito por Martin (2008): cada classe possui exatamente uma razão para mudar. O *errorMiddleware* implementa ainda uma distinção de ambiente por meio de verificação de *NODE_ENV*: em produção, o campo *stack* é omitido da resposta, prevenindo a exposição de detalhes internos da aplicação a potenciais atacantes; em desenvolvimento, o *stack trace* completo é incluído para facilitar a depuração.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

O utilitário *asyncHandler*, definido no mesmo módulo *errorHandler.ts*, encapsula funções assíncronas do *Express* em uma *closure* que captura automaticamente rejeições de *Promise* e as encaminha ao *next()* do *framework*, eliminando a necessidade de blocos *try/catch* individuais em cada um dos *controllers* da aplicação. Essa intervenção reduziu a superfície de código repetitivo, concentrou a responsabilidade de captura de erros em um único ponto e tornou os *controllers* visualmente mais limpos e focados exclusivamente na lógica de domínio. No plano das operações de banco de dados, a duplicação identificada no diagnóstico foi endereçada pela extração da constante ALUNO_SELECT no módulo *alunosController.ts*. Essa constante define, em um único lugar, os doze campos retornados em todas as operações de consulta sobre alunos, eliminando a necessidade de lógica de derivação em camadas superiores. Qualquer alteração futura no conjunto de campos ou na lógica de cálculo de status passa a exigir intervenção em um único ponto do código, reduzindo o risco de inconsistência entre operações e o custo de manutenção (HUNT; THOMAS, 1999).

A separação de responsabilidades foi reforçada pela criação de três *middlewares* de autenticação com escopos distintos e não sobrepostos: *authenticateUser*, *authenticateAluno* e *authenticateAny*. Antes da refatoração, a lógica de verificação de identidade estava misturada à lógica de autorização nos próprios *controllers*. Após a intervenção, cada *middleware* passou a ter uma única responsabilidade declarada (verificar a identidade de um tipo específico de ator), enquanto a lógica de autorização por cargo foi extraída para o *middleware authorizeRoles*, composto de forma independente sobre qualquer rota que exija restrição por perfil.

A validação de entradas, anteriormente ausente ou dispersa, foi centralizada no módulo *validator.ts* com a biblioteca *Joi*. Os *schemas* definidos cobrem os pontos de entrada mais sensíveis da aplicação. A opção *abortEarly*: *false* garante que todos os erros de validação sejam retornados em uma única resposta, e *stripUnknown: true* remove automaticamente campos não declarados nos *schemas*, prevenindo a injeção de propriedades inesperadas no corpo das requisições. Essa camada de validação passou a atuar antes de qualquer acesso ao banco de dados, garantindo que apenas dados estruturalmente válidos cheguem aos *controllers.*

O endurecimento de segurança foi conduzido de forma paralela à intervenção estrutural, organizando-se em quatro frentes distintas: proteção de protocolo HTTP, autenticação e gestão de sessão, controle de autorização por perfil e proteção de entradas. Cada frente endereça diretamente uma ou mais categorias de vulnerabilidade identificadas no diagnóstico, tendo o *OWASP Top Ten* (OWASP FOUNDATION, 2021) como referência normativa para validação das contramedidas adotadas.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

A primeira camada de proteção foi aplicada diretamente no pipeline do Express: o *middleware helmet*, registrado como primeira camada, configura automaticamente um conjunto de cabeçalhos HTTP de segurança que reduzem a superfície de ataque a vetores como *clickjacking*, *MIME sniffing* e injeção de *scripts* via navegador. Complementarmente, a política de CORS foi diferenciada por ambiente: qualquer origem *localhost* é aceita em desenvolvimento, enquanto em produção apenas as origens declaradas em *CORS_ORIGIN* são autorizadas, e o *payload* máximo aceito foi limitado a 1 MB, reduzindo a exposição a ataques de negação de serviço por sobrecarga de requisição. Em conjunto, essas medidas endereçam a categoria A05 *Security Misconfiguration* do OWASP *Top Ten*.

O registro de acessos foi implementado por meio do *middleware morgan*, com persistência em arquivo de log, viabilizando a análise forense de incidentes e o monitoramento de padrões de acesso anômalos, endereçando a categoria A09 *Security Logging and Monitoring Failures*.

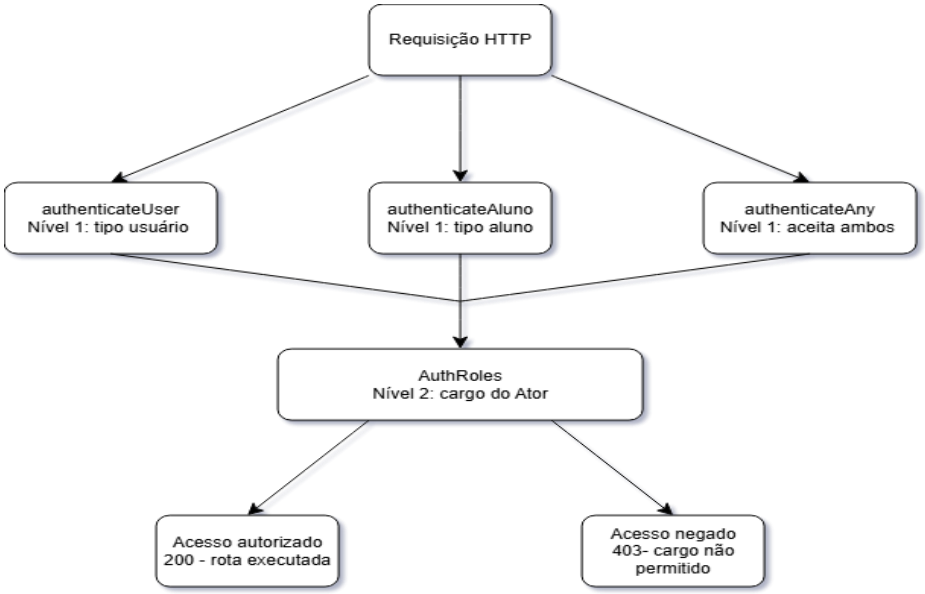
A autenticação por JWT foi implementada com um mecanismo de proteção crítico de ambiente no módulo *auth.ts*: caso a variável *JWT_SECRET* não esteja definida em produção, o processo *Node.js* é interrompido imediatamente com lançamento de erro fatal, prevenindo que a aplicação seja iniciada com uma chave padrão insegura. Em desenvolvimento, um valor *fallback* é aceito para facilitar o trabalho local sem exigir configuração adicional. Essa abordagem, denominada *fail-fast,* garante que falhas de configuração críticas sejam detectadas na inicialização do serviço, e não em tempo de execução durante uma requisição real.

A expiração de sessão foi configurada em oito horas via constante *JWT_CONFIG*, e a verificação de *tokens* diferencia semanticamente dois cenários de falha: ausência de *token*, respondida com HTTP 401 e mensagem "*Token* não fornecido"; e *token* expirado (*TokenExpiredError*), respondida com mensagem distinta "*Token* expirado. Faça login novamente." Essa diferenciação melhora a rastreabilidade de incidentes, pois distingue tentativas de acesso não autenticado de sessões legítimas encerradas por tempo, endereçando a categoria A07 *Identification and Authentication Fa*ilures.

As senhas de todos os atores do sistema são armazenadas com *hash* produzido pela biblioteca *bcryptjs* com fator de custo 10, garantindo que o custo computacional de geração do *hash* seja suficientemente alto para tornar inviável a quebra por força bruta, mesmo em caso de vazamento da base de dados. Essa contramedida endereça a categoria A02 *Cryptographic Failures*.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

Figura 2. Fluxo de controle de autorização por perfil



Fonte: Elaborada pelos autores (2026).

A Figura 2 ilustra o fluxo de autorização implementado. A requisição passa primeiro por um *middleware* de autenticação que identifica o tipo de ator, funcionário ou aluno, e valida o *token* JWT correspondente; só então o *middleware authorizeRoles* verifica se o cargo específico do ator tem permissão para a rota solicitada. Essa separação em dois níveis permite que uma mesma rota aceite múltiplos tipos de ator (por meio de *authenticateAny*) enquanto ainda restringe operações sensíveis a cargos específicos, evitando a duplicação de lógica de autorização em cada *controller* individualmente.

A proteção contra injeção de SQL foi garantida pelo uso exclusivo de *queries* parametrizadas com marcadores posicionais ($1, $2, ..., $n) em todos os *controllers* inspecionados, padrão da biblioteca *node-postgres*. Essa técnica separa estruturalmente o código SQL dos dados fornecidos pelo usuário, inviabilizando a manipulação do comando por entradas maliciosas e endereçando a categoria A0*3 Injection.*

A validação de esquema atua como camada complementar de defesa contra injeção, endereçando a categoria A03.

3. RESULTADOS E DISCUSSÃO

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

Os resultados obtidos com as intervenções descritas na metodologia são organizados nos mesmos três eixos que orientaram o diagnóstico: qualidade e *Clean Code*, segurança e arquitetura. Os resultados são discutidos em formato comparativo, estado anterior identificado na inspeção estática versus estado posterior verificado na análise do código refatorado.

3.1. Qualidade e Clean Code

No estado anterior ao diagnóstico, cada *controller* tratava (ou ignorava) suas próprias exceções de forma independente, sem padrão de resposta, nem distinção entre falhas operacionais esperadas e erros inesperados. Após a centralização do tratamento de erros, um único módulo passou a cobrir os 15 *controllers* da aplicação com um único ponto de intervenção: qualquer alteração no formato de resposta de erro, que antes exigia intervenção manual em múltiplos *controllers* simultaneamente, passou a exigir modificação em um único artefato. Martin (2008) descreve esse tipo de centralização como uma das métricas mais concretas de redução de dívida técnica. O custo de manutenção cai proporcionalmente ao número de pontos que antes replicavam essa lógica.

A tipagem de erros por categorias semânticas eliminou a prática de fixar códigos HTTP numéricos diretamente nos *controllers*, substituindo-os por referências verificadas em tempo de compilação. Cabral *et al.* (2024) associam esse tipo de centralização tipada à redução de defeitos introduzidos durante a manutenção, o que é consistente com o ganho observado aqui.

A eliminação de duplicação na projeção de consultas sobre alunos produziu resultado igualmente verificável. Antes da extração, a mesma projeção era replicada individualmente em cada operação do *CRUD*. Após a refatoração, qualquer alteração no conjunto de campos ou na lógica de cálculo de status requer intervenção em um único lugar, com propagação automática para todas as operações que a consomem, representando uma aplicação direta do princípio *DRY* (HUNT; THOMAS, 1999).

O encapsulamento de funções assíncronas eliminou a necessidade de blocos *try/catch* individuais em cada um dos 15 *controllers*. Latte *et al.* (2019) identificam que a dispersão de tratamento de erros é um dos principais vetores de introdução de defeitos silenciosos em sistemas *web* de pequeno e médio porte; após a intervenção, esse cenário (um *controller* omitir por descuido o bloco *catch*) torna-se estruturalmente impossível, já que qualquer exceção não capturada é automaticamente encaminhada ao tratamento central.

Por fim, a distinção de ambiente implementada no tratamento de erros (*stack trace* completo em desenvolvimento, omitido em produção) produziu um resultado de segurança com impacto direto na qualidade operacional: informações internas da aplicação deixaram de ser

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

expostas em respostas de erro em produção, sem comprometer a depuração em desenvolvimento.

3.2. Segurança (OWASP)

Das dez categorias elencadas pelo OWASP *Top Ten* 2021, seis foram diretamente endereçadas no PlankGYM. Os resultados são apresentados em correspondência direta com cada categoria e com foco no estado anterior versus posterior.

A01 — *Broken Access Contro*l: no estado anterior ao diagnóstico, não havia distinção de acesso entre os diferentes perfis de administrador, atendente e aluno, expondo operações administrativas sensíveis a qualquer usuário. Após o modelo RBAC implementado em dois níveis, todas as 14 rotas registradas passaram a exigir pelo menos um *middleware* de autenticação, e operações sensíveis, como exclusão de registros e gestão financeira, ficaram restritas ao perfil administrador. Butt *et al.* (2023) associam esse tipo de arquitetura em camadas à redução significativa da superfície de ataque.

A02 — *Cryptographic Failures*: no estado anterior, havia risco de armazenamento de credenciais em formato reversível. Após a intervenção, todas as senhas, incluindo as de funcionários e alunos e a senha inicial derivada do CPF, passaram a ser armazenadas com *hash* de fator de custo 10, equilibrando a inviabilização de ataques de força bruta e a preservação do tempo de resposta do *endpoint* de autenticação.

A03 — *Injection:* no estado anterior, consultas sem parametrização eram vulneráveis à injeção de comandos arbitrários. Após a adoção exclusiva de *queries* parametrizadas, nenhuma operação do sistema constrói SQL dinamicamente por concatenação, o vetor primário dessa categoria (OWASP FOUNDATION, 2021), e a camada de validação de esquema anterior ao banco de dados reduz ainda mais a superfície de entrada não confiável.

A05 — *Security Misconfiguration*: foi o eixo com maior número de intervenções simultâneas no estado anterior (ausência de headers de segurança, CORS irrestrito, sem separação formal entre ambientes). Após a intervenção, essas três lacunas foram fechadas de forma coordenada, incluindo o *fail-fast* na ausência de chave de assinatura em produção, garantindo que nenhuma configuração insegura de desenvolvimento chegue à produção.

A07 — *Identification and Authentication Failures*: a diferenciação entre cenários de falha de autenticação produziu duplo impacto: usabilidade (mensagens distintas para *token* ausente versus expirado) e rastreabilidade operacional. A expiração de sessão em oito horas limita a janela de exposição em caso de comprometimento de um *token* válido.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

A09 — *Security Logging and Monitoring Failures*: no estado anterior, a ausência de registro de acessos impedia qualquer análise forense em caso de incidente de segurança. Após a intervenção, o registro contínuo de requisições viabiliza tanto a detecção de padrões de acesso anômalos quanto a reconstituição da sequência de eventos em caso de incidente.

Limitação identificada — A07 parcial: o único vetor não completamente mitigado foi a ausência de *rate limiting* nas rotas de autenticação; o *endpoint* de login permanece tecnicamente suscetível a ataques de força bruta por volume, embora o custo do *hash bcrypt* já eleve significativamente o custo computacional de cada tentativa.

3.3. Arquitetura (*TypeScript* e modelagem de relacionamento)

Os resultados arquiteturais cobrem duas dimensões: cobertura de tipagem estática no *backend* e modelagem relacional do banco de dados.

A totalidade da lógica de servidor está coberta por *TypeScript*, o que permite ao compilador detectar em tempo de compilação classes inteiras de erros (acesso a propriedades inexistentes, incompatibilidade de tipos, retornos inesperados de *query*) que em *JavaScript* puro só seriam identificados em tempo de execução. Bogner e Merkel (2022) associam esse tipo de cobertura a menor taxa de bugs reportados por linha de código; no PlankGYM, esse benefício é especialmente relevante nos *middlewares* de autenticação, nos quais o compilador impede que dados de um tipo de ator sejam acessados em rotas restritas a outro. O *frontend*, no entanto, permanece majoritariamente em *JavaScript*, assimetria já identificada no diagnóstico e apontada como trabalho futuro.

O esquema relacional organiza o domínio em 11 tabelas inter-relacionadas, com integridade garantida por oito *foreign keys* e exclusão em cascata aplicada onde há dependência forte entre entidades, prevenindo o acúmulo de registros órfãos. Restrições de unicidade (CPF, e-mail) atuam como salvaguarda de domínio independente da camada de aplicação, redundância intencional consistente com o princípio de defesa em profundidade também aplicado no eixo de segurança. A evolução do esquema via *migrations* versionadas produz, como efeito colateral, um artefato de documentação viva do domínio, rastreável no histórico do repositório.

Após análise em conjunto, os três eixos — qualidade, segurança e arquitetura — produziram melhorias interdependentes: a tipagem estática tornou o tratamento de erros mais confiável, que por sua vez tornou as respostas de segurança mais consistentes. Essa interdependência, em que o ganho em um eixo amplifica os demais, é consistente com os

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

achados de Yanakiev *et al.* (2025) sobre reestruturação de sistemas legados com princípios *SOLID*.

4. CONSIDERAÇÕES FINAIS

Este estudo teve como objetivo demonstrar, por meio de um estudo de caso aplicado ao sistema PlankGYM, como a refatoração estrutural orientada por princípios de *Clean Code* e o endurecimento de segurança baseado nas diretrizes OWASP transformam um sistema *web* legado em uma arquitetura limpa, tipada e segura. Os resultados indicaram que esse objetivo foi alcançado nos três eixos propostos, sem alteração das regras de negócio já consolidadas no domínio da academia.

Os três eixos investigados neste estudo — qualidade, segurança e arquitetura —, alinhados à prática de *Clean Code*, confirmam, de forma mensurável, a resposta ao problema de pesquisa proposto. No eixo de qualidade, a centralização do tratamento de erros e a eliminação de duplicação em operações de banco de dados reduziram o número de pontos de intervenção necessários para qualquer alteração futura, confirmando a relação entre centralização e redução de custo de manutenção descrita por Martin (2008) e Hunt e Thomas (1999). No eixo de segurança, seis das dez categorias do OWASP *Top Ten* 2021 foram diretamente mitigadas por meio da combinação de RBAC em dois níveis, autenticação JWT com salvaguardas de ambiente, *queries* parametrizadas e trilha de auditoria. No eixo de arquitetura, a cobertura total de *TypeScript* no *backend* e a modelagem relacional com restrições de integridade formalizaram, em nível de compilador e de banco de dados, garantias que antes dependiam exclusivamente da camada de aplicação. Em conjunto, esses resultados confirmam que a aplicação sistemática de práticas de refatoração, boas práticas de engenharia de *software* e endurecimento de segurança é capaz de transformar um sistema *web* com dívida técnica notória em uma arquitetura limpa, segura e sustentável, sem comprometer as regras de negócio já consolidadas.

Duas limitações identificadas ao longo deste trabalho apontam direções concretas para a continuidade. A primeira é a ausência de *rate limiting* nas rotas de autenticação: o *endpoint* de *login* permanece tecnicamente exposto a ataques de força bruta por volume, embora o custo computacional do *hash bcrypt* já dificulte esse vetor, a implementação de uma camada de limitação de taxa é o próximo passo natural do endurecimento de segurança. A segunda é a assimetria de tipagem entre *backend* e *frontend*: enquanto o servidor está integralmente coberto por *TypeScript*, o cliente permanece majoritariamente em JavaScript, e sua migração é identificada como continuidade natural do processo de modernização. Como trabalho futuro de

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

caráter mais amplo, propõe-se ainda a validação empírica deste roteiro de refatoração em outros sistemas de pequeno porte, verificando se os ganhos observados no PlankGYM se replicam em contextos com *stacks* e domínios distintos.

REFERÊNCIAS

BOGNER, Justus; MERKEL, Manuel. To type or not to type? A systematic comparison of the *software* quality of JavaScript and TypeScript applications on GitHub. In: **INTERNATIONAL CONFERENCE ON MINING *SOFTWARE* REPOSITORIES**, 19., 2022, Pittsburgh. *Proceedings…* New York: ACM, 2022. p. 658–669. DOI: 10.1145/3524842.3528454. Disponível em: https://arxiv.org/abs/2203.11115. Acesso em: jul. 2026.

BUTT, Ateeq Ur Rehman *et al.* An optimized role-based access control using trust mechanism in e-health cloud environment. *IEEE Access*, v. 11, p. 138813–138826, 2023. DOI: 10.1109/ACCESS.2023.3335984. Disponível em: https://ieeexplore.ieee.org/document/10327710. Acesso em: jul. 2026.

CABRAL, Raphael *et al.* Investigating the impact of SOLID design principles on machine learning code understanding. In: **IEEE/ACM INTERNATIONAL CONFERENCE ON AI ENGINEERING — *SOFTWARE* ENGINEERING FOR AI**, 3., 2024, Lisboa. *Proceedings…* New York: ACM, 2024. p. 7–17. DOI: 10.1145/3644815.3644957. Disponível em: https://arxiv.org/abs/2402.05337. Acesso em: jul. 2026.

FOWLER, Martin. *Refactoring: improving the design of existing code.* Boston: Addison-Wesley Professional, 1999. ISBN 978-0-201-48567-7.

HUNT, Andrew; THOMAS, David. *The pragmatic programmer: from journeyman to master.* Boston: Addison-Wesley Professional, 1999. ISBN 978-0-201-61622-4.

JONES, Michael B.; BRADLEY, John; SAKIMURA, Nat. JSON Web Token (JWT). *RFC 7519*, Internet Engineering Task Force (IETF), maio 2015. Disponível em: https://datatracker.ietf.org/doc/html/rfc7519. Acesso em: jul. 2026.

LATTE, Benedikt; HENNING, Sven; WOJCIESZAK, Marek. Clean code: on the use of practices and tools to produce maintainable code for long-living *software*. In: **COLLABORATIVE WORKSHOP ON EVOLUTION AND MAINTENANCE OF LONG-LIVING SYSTEMS (EMLS)**, 6., 2019, Stuttgart. *Proceedings…* [S.l.]: CEUR-WS, 2019. p. 97–101. Disponível em: https://ceur-ws.org/Vol-2308/emls2019paper01.pdf. Acesso em: jul. 2026.

MARTIN, Robert C. *Clean architecture: a craftsman's guide to software structure and design.* Upper Saddle River: Prentice Hall, 2017. ISBN 978-0-13-449416-6. Disponível em: https://dl.acm.org/doi/10.5555/3175742. Acesso em: jul. 2026.

MARTIN, Robert C. *Clean code: a handbook of agile software craftsmanship.* Upper Saddle River: Prentice Hall, 2008. ISBN 978-0-13-235088-4. Disponível em: https://dl.acm.org/doi/10.5555/1388398. Acesso em: jul. 2026.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

OWASP FOUNDATION. *OWASP Top Ten 2021: the ten most critical web application security risks.* OWASP Foundation, 2021. Disponível em: https://owasp.org/www-project-top-ten. Acesso em: jul. 2026.

TANENBAUM, Andrew S.; WETHERALL, David J. *Computer networks.* 5. ed. Upper Saddle River: Pearson Prentice Hall, 2011. ISBN 978-0-13-212695-3.

YANAKIEV, Ivan; LAZAR, Bogdan-Mihai; CAPILUPPI, Andrea. Applying SOLID principles for the refactoring of legacy code: an experience report. *Journal of Systems and Software*, v. 220, art. 112254, fev. 2025. DOI: 10.1016/j.jss.2024.112254. Disponível em: https://www.sciencedirect.com/science/article/pii/S016412122400298X. Acesso em: jul. 2026.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.