UNIVERSIDADE SÃO JUDAS TADEU
Gestão e Qualidade de Software

Nome dos Alunos:

Amanda Agustinho Costa	823150503	Ciência da Computação
Evellyn Cruz Souza	823213551	Ciência da Computação
João Pedro Agustinho Costa	823223417	Ciência da Computação
Gabriel Crispino Torres	822140678	Ciência da Computação
Allan Pauluk Medeiros	822142483	Ciência da Computação

PROJETO A3

BUTANTÃ
2025

Relatório de Refatoração e Testes Unitários - Sistema de Diagnóstico Hepático

i. Descrição das Deficiências do Código Original Identificadas

A versão inicial do sistema, comum em protótipos de rápida implementação, apresentava um alto acoplamento e baixa adesão aos princípios do Clean Code, o que comprometia seriamente sua manutenibilidade e testabilidade.

A principal deficiência identificada era a **Baixa Separação de Responsabilidades (SRP)**. O código do servidor Node.js (index.js), por exemplo, misturava funções de roteamento Express com manipulação direta do sistema de arquivos (fs para o diagnosticos.json) e chamadas HTTP externas para o serviço Python. Essa falta de isolamento gerava um **Acoplamento Forte de Serviço**, onde as rotas da API se tornavam totalmente dependentes da rede e do status do disco. Por consequência, a testabilidade era prejudicada, pois era impossível escrever testes unitários para a lógica de negócio sem envolver a rede ou o sistema de arquivos.

No lado Python, embora já existissem classes, a manipulação do modelo (.pkl) e a lógica de logging (via PredictionRepository) estavam acopladas ao *startup* do Flask, e havia uma falta de validação robusta de *payload* nos *endpoints*, aumentando o risco de *crashes* com entradas malformadas.

ii. Justificativas para as Mudanças Feitas no Código

A refatoração seguiu o princípio da **Arquitetura Limpa**, buscando isolar as responsabilidades em camadas bem definidas (Apresentação, Aplicação e Infraestrutura).

No Node.js, a lógica de persistência foi isolada na classe DiagnosisRepository.js. Isso permitiu que o servidor Express passasse a usar métodos simples como repo.list() e repo.create(), sem precisar saber que o armazenamento é um arquivo JSON. Toda a comunicação de rede foi encapsulada na classe PredictionClient.js. Esta mudança é vital, pois o *roteamento* (index.js) agora apenas orquestra, chamando o Client e o Repository mockados nos testes, eliminando a dependência de rede e disco das rotas principais.

No lado Python, a criação de classes HepatitisPredictor e PredictionRepository em prediction_service.py garantiu o isolamento da lógica de ML e de logging. A função create_app() em model_api.py agora instancia o preditor e o repositório **uma única vez** globalmente, garantindo que o modelo seja carregado na memória no *startup* e não a cada requisição, resultando em maior eficiência. Além disso, o tratamento de erro em model_api.py foi aprimorado para incluir o traceback detalhado no retorno 500 apenas quando a variável de ambiente DEBUG=1 está ativa, melhorando a capacidade de diagnóstico.

iii. Descrição dos Testes Unitários Implementados

A implementação de testes unitários foi o resultado direto da refatoração e focou nas unidades lógicas centrais. Nos testes Python, o arquivo **test_prediction_service.py** verifica a integridade da lógica de Machine Learning, garantindo que o HepatitisPredictor normalize corretamente os dados de entrada (tratando valores ausentes e conversões de tipo) e que o pipeline de ML seja carregado ou treinado de forma confiável. Além disso, foram incluídos testes específicos para validar o comportamento diante de payloads inválidos ou incompletos, assegurando que a API retorne 400 quando informações obrigatórias estiverem ausentes ou em formatos incorretos. Também foram criados testes que simulam falhas internas no predictor, como erro ao carregar o modelo ou exceções durante o pipeline de predição, verificando se a API responde com 500 e respeita o comportamento esperado conforme a variável de ambiente DEBUG esteja ou não habilitada. No arquivo **test_model_api.py**, a testagem cobre o contrato completo da API Flask, avaliando o retorno dos endpoints /predict e /train (200, 400, 500) e assegurando que o log, via PredictionRepository mockado, seja chamado corretamente. Adicionalmente, os métodos do repositório foram testados com mocks para confirmar que o registro de logs funciona mesmo sem o acesso real ao sistema de arquivos.

Nos testes JavaScript, o arquivo **DiagnosisRepository.test.js** valida toda a lógica de CRUD (Criação, Leitura, Atualização e Deleção) do repositório, utilizando jest.mock('fs') para simular operações de disco. A bateria de testes cobre também cenários de erro, como JSON corrompido, falha simulada na escrita do arquivo e comportamento correto quando o arquivo de base está vazio. Para o **PredictionClient.js**, os testes garantem que as URLs sejam formatadas de maneira correta e que erros de rede, incluindo timeouts e respostas 500 do serviço Python, sejam tratados de forma adequada. Por fim, os testes do servidor Express (index.js), escritos com supertest, verificam se as rotas utilizam corretamente os mocks do PredictionClient e do DiagnosisRepository e se retornam os códigos HTTP apropriados quando ocorrem falhas, como 502 quando o cliente de predição está indisponível ou 500 quando o repositório apresenta erro interno. Os testes ainda confirmam que, quando NODE_ENV está configurado como produção, o servidor não expõe stack traces no retorno.

Como etapa final, ferramentas de cobertura foram utilizadas (pytest-cov e Jest Coverage), demonstrando que as partes essenciais do sistema estão amplamente testadas: a cobertura média dos módulos Python ultrapassou 90% nas funções centrais, enquanto os módulos do Node.js ficaram entre 80% e 95%, especialmente altos nos arquivos refatorados (Repository, Client e Rotas). Esses resultados reforçam que a arquitetura modular possibilitou testagem muito mais completa e confiável.

iv. Conclusão sobre a Importância do Clean Code na Manutenção de Software

A experiência de refatoração do Sistema de Diagnóstico Hepático demonstrou que o **Clean Code não é apenas sobre estética, mas sim sobre economia e sustentabilidade**.

A separação em módulos de responsabilidade única (como o PredictionClient e o DiagnosisRepository) transformou o sistema. Onde antes havia um código frágil e acoplado, agora existe um conjunto de unidades isoladas que podem ser modificadas e corrigidas com segurança.

A **testabilidade plena**, alcançada após a refatoração, é o maior benefício para a manutenção. Os testes unitários atuam como uma rede de segurança que permite que o desenvolvedor altere o código interno de um repositório (ex: mudar de JSON para MySQL) com a certeza de que as rotas da API que o consomem não serão quebradas.

Em última análise, o Clean Code facilita a colaboração em grupo e garante que o custo de manutenção futura seja minimizado, transformando o software de um passivo complexo em um ativo manutenível e evolutivo.