Plano de Análise de Defeitos – Projeto Integrador

Equipe:

- Lucas de Souza Silva
- Amanda Priscila
- Ana Beatriz
- Kellvyn

Data: 07 de outubro de 2025

---

1. Introdução

1.1. Objetivo

Este documento estabelece o Plano de Análise de Defeitos para o Projeto Integrador deste semestre. Seu objetivo é definir um processo claro, padronizado e mensurável para identificar, registrar, priorizar, investigar, corrigir e prevenir defeitos ao longo de todo o ciclo de vida do software. A adoção deste plano visa transformar a gestão de bugs em uma prática integrada ao nosso fluxo de desenvolvimento (CI/CD), garantindo a entrega de um produto final com maior qualidade, estabilidade e confiabilidade.

1.2. Escopo

Este plano abrange todas as camadas do projeto, incluindo:

- **Hardware e Firmware:** Componentes físicos (placa Arduino/ESP32, sensores, atuadores) e o código embarcado.
- **Backend:** A aplicação do servidor, incluindo APIs, lógica de negócios e conexão com o banco de dados.
- **Frontend (Dashboard Web):** A interface do usuário, incluindo responsividade, usabilidade e interações.
- **Infraestrutura:** O ambiente de CI/CD, configurações de deploy e serviços de nuvem.

---

2. Processo de Gestão de Defeitos

Para garantir uma gestão eficaz, seguiremos um ciclo de vida de defeitos estruturado em cinco etapas principais.

2.1. Identificação e Registro

Qualquer comportamento inesperado ou falha encontrada deve ser formalmente registrada para garantir que não seja esquecida.

- **Ferramenta de Registro:** Utilizaremos a funcionalidade **"Issues"** do nosso repositório no **GitHub** como sistema centralizado para o rastreamento de todos os defeitos.
- **Template de Registro:** Todo novo defeito registrado deverá seguir o template abaixo para garantir a clareza e a completude das informações:

- **Título:** Um resumo claro e conciso do problema. (Ex: "Dashboard não atualiza dados do sensor de temperatura em tempo real").
- **Descrição:** Detalhamento do problema.
- **Passos para Reproduzir:** Sequência exata de ações que levam ao erro.
- **Comportamento Esperado:** O que deveria ter acontecido.
- **Comportamento Atual:** O que de fato aconteceu.
- **Ambiente:** Onde o erro ocorreu (ex: Hardware, Desenvolvimento Local, Produção).
- **Anexos:** Screenshots, vídeos ou logs que ajudem a ilustrar o defeito.

2.2. Priorização

Após o registro, cada defeito será classificado para que a equipe possa focar nos problemas mais críticos primeiro.

- **Sistema de Priorização:** Usaremos **labels (etiquetas)** no GitHub para definir a prioridade, com base no impacto e na urgência.
  - `priority:critical`: Bloqueia funcionalidades essenciais do sistema ou causa perda de dados. Impede o uso do produto.
  - `priority:high`: Afeta uma funcionalidade importante, mas existem formas de contornar o problema (workaround).
  - `priority:medium`: Defeito de menor impacto que não impede o uso geral do sistema, mas prejudica a experiência.
  - `priority:low`: Problema cosmético (ex: desalinhamento de um botão), erro de digitação ou sugestão de melhoria de baixo impacto.

2.3. Investigação e Correção

Uma vez priorizado, o defeito é atribuído a um membro da equipe para ser solucionado.

- **Atribuição:** O defeito será atribuído a um responsável (`Assignee` no GitHub), que confirmará a capacidade de reproduzir o erro e iniciará a investigação.
- **Fluxo de Correção (Git Flow):**
  1. O responsável criará uma nova *branch* a partir da `develop` ou `main`, seguindo o padrão `fix/nome-do-defeito`.
  2. A correção será implementada e testada localmente nesta *branch*.
  3. Após a correção, um **Pull Request (PR)** será aberto para mesclar a *branch* de correção na *branch* principal. O PR deve referenciar a *Issue* correspondente.

2.4. Verificação e Fechamento

A correção precisa ser validada antes de ser considerada concluída.

- **Revisão de Código (Code Review):** O Pull Request deverá ser revisado e aprovado por, no mínimo, **um outro membro da equipe**. A revisão deve focar não apenas na correção, mas também na qualidade e no impacto do novo código.
- **Verificação da Correção:** Após o *merge* do PR, a pessoa que registrou o defeito (ou outro membro designado) será responsável por testar e confirmar que o problema foi resolvido no ambiente de desenvolvimento/homologação.

- **Fechamento:** Se a correção for validada com sucesso, a *Issue* no GitHub será fechada. Caso contrário, ela será reaberta com comentários adicionais.

2.5. Prevenção

Além de corrigir, devemos aprender com os defeitos para evitar que se repitam.

- **Análise de Causa Raiz:** Para defeitos críticos, a equipe discutirá a causa raiz do problema após a sua resolução.
- **Ações Preventivas:** As lições aprendidas serão usadas para melhorar nossos processos. Exemplos:
  - Adicionar novos passos de verificação ao nosso checklist (Anexo A).
  - Criar testes automatizados (unitários, de integração) que cubram o cenário do defeito.
  - Melhorar a documentação para evitar erros de configuração ou uso.

---

3. Ferramentas de Suporte

- **Rastreamento de Defeitos:** GitHub Issues
- **Controle de Versão:** Git e GitHub
- **Comunicação da Equipe:** Microsoft Teams / Discord / WhatsApp
- **Automação (CI/CD):** GitHub Actions

Anexo A: Checklist de Verificação de Defeitos Comuns

Esta lista serve como um guia de referência durante as fases de desenvolvimento e teste para identificar proativamente os defeitos mais comuns em nosso projeto.

A.1. Defeitos de Hardware – Arduino

- **Conectividade e Integridade:**
  - ☐ Verificar se todos os componentes (placa, sensores, atuadores) respondem a testes básicos.
  - ☐ Revisar a montagem do circuito, identificando ligações incorretas ou inversão de polaridade.
  - ☐ Utilizar multímetro para verificar continuidade, tensão e corrente em pontos críticos.
  - ☐ Inspecionar jumpers, conectores e soldas em busca de mau contato.
  - ☐ Verificar a integridade da porta USB e dos cabos de alimentação/programação.
- **Operação:**
  - ☐ Avaliar superaquecimento de componentes durante o uso contínuo.
- **Recursos:**
  - ☐ Manter peças de reposição disponíveis para substituição rápida.

A.2. Defeitos no Código – Arduino (Firmware)

- **Compatibilidade e Configuração:**
  - ☐ Garantir que o código é compatível com a placa utilizada (ex: Uno vs. ESP32).
  - ☐ Verificar se o driver da placa está instalado e atualizado no computador.
  - ☐ Confirmar se a porta USB e o modelo da placa (board) estão configurados corretamente na IDE.
- **Boas Práticas e Depuração:**

- [ ] Implementar logs seriais (`Serial.print`) para rastreabilidade e diagnóstico.
- [ ] Verificar a importação de bibliotecas e suas versões.
- [ ] Revisar erros de sintaxe, digitação ou lógica no código.
- [ ] Evitar o uso excessivo de `delay()` que prejudique a resposta em tempo real.
- [ ] Implementar tratamento de exceções e validação de entradas de sensores.
- [ ] Modularizar o código em funções para facilitar manutenção e testes.

A.3. Defeitos no Frontend – Dashboard Web

- **Estrutura e Semântica:**
  - [ ] Utilizar tags HTML semânticas e evitar a mistura de HTML, CSS e JS em um único arquivo.
  - [ ] Garantir que classes e IDs no CSS sejam específicos para evitar estilos inesperados.
- **Responsividade e Compatibilidade:**
  - [ ] Testar `media queries` em diferentes dispositivos para garantir a responsividade.
  - [ ] Verificar a compatibilidade da aplicação em diferentes navegadores (cross-browser).
- **Usabilidade e Acessibilidade:**
  - [ ] Incluir atributo `alt` em todas as imagens.
  - [ ] Garantir bom contraste de cores e navegação funcional via teclado.
- **Desempenho e Validação:**
  - [ ] Otimizar imagens e arquivos estáticos.
  - [ ] Utilizar `async` ou `defer` em scripts que possam bloquear o carregamento da página.
  - [ ] Implementar validação de formulários no frontend.

A.4. Defeitos no Backend

- **Configuração e Ambiente:**
  - [ ] Verificar se o ambiente (`.env`) está completo e com as variáveis corretas.
  - [ ] Garantir que a string de conexão com o banco de dados está correta e há tratamento para reconexão.
- **Segurança e Validação:**
  - [ ] Proteger rotas com mecanismos de autenticação e autorização (ex: JWT).
  - [ ] Validar e sanitizar todas as entradas de usuário para prevenir SQL Injection, XSS, etc.
  - [ ] Configurar CORS e outros headers de segurança corretamente.
- **Boas Práticas e Desempenho:**
  - [ ] Implementar tratamento de erros robusto (`try/catch`) com mensagens claras.
  - [ ] Padronizar e versionar os endpoints da API.
  - [ ] Separar a lógica de negócios em camadas (ex: não sobrecarregar os controladores).
  - [ ] Otimizar consultas ao banco de dados e usar cache quando aplicável.
  - [ ] Evitar código síncrono bloqueante em tarefas pesadas.
- **Qualidade e CI/CD:**
  - [ ] Manter dependências atualizadas.
  - [ ] Implementar testes unitários e de integração.
  - [ ] Padronizar logs com níveis de severidade.
  - [ ] Integrar ferramentas de linting e testes automatizados no pipeline de CI/CD.