Plano Geral de Testes - Metodologia para Analista de Qualidade

Análise de Qualidade

Versão 2.0 - Baseada em Metodologia de Testes Funcional, Estrutural e de Mutação

Matheus Junio da Silva - 5382

Florestal-MG

2025

---

1. Objetivo do Plano

Este plano define um processo repetível de testes que cada analista de qualidade seguirá ao receber uma entrega de código. O documento funciona como:

- Guia prático para a equipe de análise
- Guia estruturado para executar testes seguindo uma metodologia prévia
- Checklist para garantir cobertura completa

---

2. Fluxo Geral: Antes de Começar os Testes

Quando você receber uma entrega de código, primeiro faça:

1. **Obtenha a documentação necessária:**
    - Diagrama de Classes (modelo)
    - Caso de uso
    - Código-fonte da implementação
2. **Identifique o escopo:**
    - Quais funcionalidades foram implementadas?
    - Quais classes/métodos foram alteradas?
    - Há regressões possíveis em funcionalidades anteriores?

3. **Prepare o ambiente:**
    - Sistema em funcionamento
    - Dados de teste disponíveis

---

3. Etapa 1: Teste Funcional (Caixa Preta)

O que fazer: Verificar se o sistema funciona conforme os requisitos especificados. **Foco:** Dados de entrada fornecidos e respostas produzidas (sem conhecimento do código).

3.1 Aplicar Particionamento em Classes de Equivalência

Passo a passo:

1. Para cada **campo de entrada** da funcionalidade:
    - Identifique **classes válidas** (valores que devem funcionar)inte
    - Identifique **classes inválidas** (valores que devem gerar erro)
2. Exemplo (se a funcionalidade recebe um intervalo):
    - Entrada: limite_inferior e limite_superior
    - Classe válida de limite_inferior: 0 ≤ limite_inferior ≤ limite_superior
    - Classe inválida de limite_inferior: limite_inferior < 0 E limite_inferior > limite_superior
3. **Crie um caso de teste por classe** (representativo)
    - Selecione UM valor de cada classe
    - Registre o resultado esperado

3.2 Aplicar Análise de Valor Limite

Passo a passo:

1. Identifique os **limites de cada classe de equivalência**
2. Crie casos de teste **nas bordas**:
    - Valor mínimo válido
    - Valor máximo válido
    - Valor um pouco abaixo do mínimo
    - Valor um pouco acima do máximo
3. Exemplo:
    - Se a classe válida é 0 ≤ x ≤ 100:
        - Teste com -1 (inválido)
        - Teste com 0 (limite válido)

        - Teste com 100 (limite válido)
        - Teste com 101 (inválido)

3.3 Aplicar Grafo Causa-Efeito

Passo a passo:

1. Se a funcionalidade tem **múltiplas condições de entrada que se combinam**, aplique este critério:
    - **Causas** = condições de entrada
    - **Efeitos** = ações/saídas esperadas
2. Identifique todas as **combinações lógicas** importantes
3. Crie casos de teste para cada combinação significativa
4. Exemplo (carrinho de compras):
    - Causa 1: valor_compra > R$60
    - Causa 2: quantidade_produtos < 3
    - Efeito: frete_grátis = SIM
    - Teste: (61, 2) → frete grátis ✓
    - Teste: (61, 3) → frete cobrado ✓
    - Teste: (60, 2) → frete cobrado ✓

3.4 Teste Baseado em Casos de Uso

Passo a passo:

1. Para cada **caso de uso** implementado:
    - Identifique os cenários (caminho feliz, caminhos alternativos, exceções)
    - Para cada cenário:
        - Liste as condições de entrada
        - Liste as saídas esperadas
        - Combine com classes de equivalência e valores limites quando possível
2. Crie casos de teste documentados:
    - **Cenário:** descrição
    - **Pré-condições:** estado inicial do sistema
    - **Passos:** ações realizadas pelo usuário
    - **Resultado esperado:** o que deve acontecer
    - **Dados de teste:** valores específicos
3. **Registre os resultados:**
    - ✓ Passou / ✗ Falhou
    - Se falhou: descreva o comportamento real vs. esperado

---

4. Etapa 2: Teste Estrutural (Caixa Branca)

O que fazer: Verificar o funcionamento interno do código e cobertura lógica. **Foco:** Estrutura, fluxo de controle, fluxo de dados. **Requer:** Acesso ao código-fonte.

4.1 Aplicar Critério de McCabe (Complexidade Ciclomática)

Passo a passo:

1. Para cada **método/função** implementada:
    - Construa o **Grafo de Fluxo de Controle (GFC)**
        - Cada nó = bloco de comandos
        - Cada aresta = desvio (if, loop, etc.)
2. Calcule a **complexidade ciclomática V(G)**:
    - V(G) = número de caminhos linearmente independentes
    - Fórmula simples: V(G) = E - N + 2 (E = arestas, N = nós)
3. V(G) = **número mínimo de casos de teste** necessários para:
    - Executar todas as instruções pelo menos uma vez
    - Cobrir todos os caminhos básicos
4. **Crie casos de teste** para cada caminho independente
5. **Execute e registre** se todos os caminhos foram cobertos

4.2 Aplicar Critério Todos-Nós

Passo a passo:

1. **Objetivo:** Cada comando (nó) do programa é executado pelo menos uma vez
2. **Para cada nó:**
    - Crie um caso de teste que force a execução daquele nó
    - Registre se o nó foi atingido
3. **Resultado:** Lista de cobertura de nós (ex: 15/15 nós cobertos = 100%)

4.3 Aplicar Critério Todas-Arestas

Passo a passo:

1. **Objetivo:** Cada decisão (if/else, loops) é executada em ambas as direções

2. **Para cada aresta:**
    - Crie um caso de teste que force a execução dessa aresta
    - Registre se a aresta foi atingida
3. **Resultado:** Lista de cobertura de arestas (ex: 20/25 arestas cobertos = 80%)

4.4 Aplicar Critério Fluxo de Dados (Todas-Definições e Todos-Usos)

Passo a passo:

1. **Todas-Definições:**
    - Para cada variável que é definida (atribuição):
        - Crie um caso de teste que execute essa definição
        - Registre se a definição foi atingida
2. **Todos-Usos:**
    - Para cada par (definição → uso):
        - Crie um caso de teste que force esse caminho livre de definição
        - Registre se o caminho foi coberto
3. **Resultado:** Cobertura de fluxo de dados (ex: 25/30 = 83%)

---

5. Etapa 3: Teste de Mutação (Validação de Qualidade dos Testes)

O que fazer: Verificar se seus casos de teste são eficazes em detectar erros. **Foco:** Criar versões modificadas do código e verificar se os testes as identificam.

5.1 Selecionar Operadores de Mutação

Aplicar mutações comuns:

1. **Operador SSdL (Eliminação de Comandos):**
    - Remova um comando por vez
    - Teste se seus casos de teste conseguem identificar a diferença
2. **Operador ORRN (Troca de Operador Relacional):**
    - Troque `>` por `<`, `==` por `!=`, etc.
    - Teste se seus casos de teste identificam a mudança
3. **Operador Vsrr (Troca de Variáveis):**
    - Substitua uma variável por outra

- Teste se seus casos de teste identificam o erro

5.2 Executar Mutantes e Registrar Resultados

Passo a passo:

1. Para cada mutante:
    - Execute seu conjunto de casos de teste COMPLETO
    - Registre o resultado:
        - **Morto** = casos de teste detectaram a diferença ✓
        - **Vivo** = casos de teste NÃO detectaram a diferença ✗
2. **Calcule o Score de Mutação:**
    - Score = (Mutantes Mortos) / (Total de Mutantes)
    - Escala: 0 a 1 (1.0 = 100% de efetividade)
3. **Se Score < 0.8 (80%):**
    - Crie casos de teste adicionais para matar mutantes vivos
    - Re-execute e recalcule o Score
    - Repita até Score ≥ 0.8

---

6. Etapa 4: Testes Não-Funcionais (Se Aplicável)

O que fazer: Testar requisitos além da funcionalidade básica.

6.1 Teste de Usabilidade

Passo a passo:

1. **Avaliação Heurística (especialista):**
    - Navegue pela interface
    - Verifique contra as 10 Heurísticas de Nielsen:
        1. Visibilidade do status do sistema
        2. Correspondência entre sistema e mundo real
        3. Controle e liberdade do usuário
        4. Consistência e padrões
        5. Prevenção de erros
        6. Reconhecimento vs. memorização
        7. Flexibilidade e eficiência de uso
        8. Design estético e minimalista
        9. Ajude os usuários a reconhecer, diagnosticar e recuperar-se de erros
        10. Ajuda e documentação

6.2 Teste de Desempenho

Passo a passo:

1. **Teste de Carga:**
    - Simule quantidade de usuários simultaneamente esperada
    - Meça: tempo de resposta, transações/minuto
    - Registre: sistema conseguiu processar?
2. **Teste de Stress:**
    - Sobrecargue o sistema (acima do esperado)
    - Meça: ponto de quebra, recuperação
3. **Teste de Estabilidade:**
    - Execute por período prolongado
    - Registre: comportamento mantém-se consistente?

6.3 Teste de Segurança

Passo a passo:

1. **Vulnerabilidades Comuns:**
    - Tente SQL Injection em campos de entrada
    - Tente acessar recursos sem autenticação
    - Verifique se senhas estão expostas
2. **Registre:**
    - Vulnerabilidades encontradas
    - Nível de criticidade
    - Recomendações de correção

---

7. Documentação de Resultados

Para cada entrega, crie um relatório contendo:

7.1 Resumo Executivo

- Funcionalidades testadas
- Resultado geral (PASSOU/FALHOU)
- Defeitos críticos encontrados

7.2 Testes Funcionais

Caso de Teste	Entrada	Resultado Esperado	Resultado Obtido	Status
CT01	valor1, valor2	comportamento1	comportamento_observado	✓ / ✗
CT02	...	...	...	✓ / ✗

7.3 Testes Estruturais

- Cobertura de Nós: X/Y (Z%)
- Cobertura de Arestas: X/Y (Z%)
- Cobertura de Fluxo de Dados: X/Y (Z%)
- Complexidade Ciclomática testada: V(G) = N, N caminhos cobertos

7.4 Testes de Mutação

- Total de Mutantes Gerados: N
- Mutantes Mortos: N (X%)
- Mutantes Vivos: N
- Score de Mutação: X.X (deve ser ≥ 0.80)

7.5 Defeitos Encontrados

ID	Descrição	Severidade	Status
DEF001	Campo X não valida entrada Y	Alta	Pendente/Corrigido

7.6 Conclusão

- Recomendações
- Próximas ações necessárias

---

8. Critérios de Aceitação para Aprovação de Código

O código é **APROVADO** quando atende a TODOS os critérios:

- ✓ Todos os testes funcionais passaram
- ✓ Cobertura de nós ≥ 90%
- ✓ Cobertura de arestas ≥ 85%
- ✓ Score de Mutação ≥ 0.80
- ✓ Sem defeitos críticos
- ✓ Funcionalidades não-funcionais dentro do esperado

9. Checklist resumido

```
None
Novas funcionalidades implementadas: [LISTAR FUNCIONALIDADES]

Siga este processo:

1. TESTE FUNCIONAL:
   - Aplique particionamento em classes de equivalência para
[LISTAR ENTRADAS]
   - Crie casos de teste de valor limite para [LISTAR CAMPOS]
   - Se aplicável, aplique grafo causa-efeito para [DESCREVER
LÓGICA]
   - Teste todos os cenários dos casos de uso: [LISTAR CASOS
DE USO]

2. TESTE ESTRUTURAL:
   - Para cada método [LISTAR MÉTODOS]:
     - Calcule complexidade ciclomática
     - Crie casos de teste para atingir todos os nós
     - Crie casos de teste para cobrir todas as arestas

3. TESTE DE MUTAÇÃO:
   - Aplique operador SSdL (remova comandos)
   - Aplique operador ORRN (troque operadores relacionais)
   - Execute todos os casos de teste contra cada mutante
   - Calcule Score de Mutação

4. DOCUMENTE:
   - Tabela de resultados funcionais
   - Cobertura estrutural (%)
   - Score de mutação
   - Lista de defeitos (se encontrados)
   - Recomendações

Formato de saída: [ESPECIFICAR FORMATO ESPERADO]
```

---

10. Timeline Esperada por Sprint

- **Início:** Receber entrega de código
- **Teste Funcional:** 1-2 dias
- **Teste Estrutural:** 1-2 dias
- **Teste de Mutação:** 1-2 dias
- **Relatório Final:** 0.5-1 dia
- **Total:** 3-7 dias (depende da complexidade)

---

11. Referências

- IEEE SWEBOK 4.0 - Capítulo 5 (Testing)
- Delamaro, M. E.; Maldonado, J. C.; Jino, M. "Introdução ao Teste de Software"
- 10 Heurísticas de Nielsen: https://www.nngroup.com/articles/ten-usability-heuristics/
