Instruções para o Projeto - BPP 2025.2 - Unidade 3

Visão Geral da Entrega Final

A entrega da **Unidade 3** consiste na **versão final do MVP** com foco em aspectos avançados de qualidade de software que não foram abordados na U1. Esta entrega complementa o trabalho anterior, incorporando:

1. **Automação de Testes** - Suite completa de testes unitários
2. **Análise de Cobertura** - Métricas e relatórios de cobertura de código
3. **Técnicas de Depuração** - Documentação de bugs encontrados e corrigidos
4. **Análise de Desempenho** - Identificação e correção de gargalos
5. **Gerenciamento de Memória** - Análise e otimização (quando aplicável)

Novos Requisitos para U3

1. Automação de Testes (Peso: 30%)

1.1 Suite de Testes Obrigatória

Seu projeto deve incluir:

Testes Unitários:

- Mínimo de **10 testes unitários** cobrindo funcionalidades principais
- Testes para casos de sucesso, falha e casos limites (edge cases)
- Seguir o padrão **AAA (Arrange-Act-Assert)**

1.2 Organização dos Testes

```
projeto/
├── src/
│   └── (código fonte)
├── tests/
│   ├── unit/
│   │   ├── test_modelo1.py
│   │   ├── test_modelo2.py
│   │   └── test_servicos.py
│   └── conftest.py (fixtures compartilhados)
└── README.md
```

1.3 Qualidade dos Testes

Seus testes devem seguir os princípios **FIRST**:

- ✓ **Fast** - Executam rapidamente (suite completa < 30 segundos)
- ✓ **Independent** - Não dependem de outros testes

- ✓ **Repeatable** - Resultados consistentes
- ✓ **Self-validating** - Pass/Fail claro, sem verificação manual
- ✓ **Timely** - Escritos junto com o código

1.4 Exemplo de Teste Bem Estruturado

```
def test_criar_usuario_com_dados_validos():
    # ARRANGE - Preparar dados e dependências
    dados_usuario = {
        "nome": "João Silva",
        "email": "joao@example.com",
        "senha": "SenhaSegura123!"
    }
    repositorio = UsuarioRepositorioMock()
    servico = UsuarioService(repositorio)

    # ACT - Executar a ação
    usuario_criado = servico.criar_usuario(dados_usuario)

    # ASSERT - Verificar resultados
    assert usuario_criado.nome == "João Silva"
    assert usuario_criado.email == "joao@example.com"
    assert usuario_criado.senha != "SenhaSegura123!"  # senha deve ser
hasheada
    assert len(repositorio.usuarios) == 1
```

---

2. Análise de Cobertura de Código (Peso: 20%)

2.1 Métricas Mínimas Exigidas

- **Cobertura de Linhas:** Mínimo **70%** do código total
- **Cobertura de Branches:** Mínimo **60%** das ramificações
- **Módulos Críticos:** Mínimo **85%** de cobertura

2.2 Relatório de Cobertura

Inclua na documentação:

Relatório Inicial vs Final:

```
## Evolução da Cobertura

### Primeira Análise (Sprint 1)
- Cobertura de Linhas: 45%
- Cobertura de Branches: 30%
- Linhas não cobertas: 234

### Análise Final (Sprint 3)
```

```
- Cobertura de Linhas: 78%
- Cobertura de Branches: 65%
- Linhas não cobertas: 89
```

Justificativa para Código Não Coberto:

```
## Código Não Coberto (22%)

### Módulo de Configuração (config.py)
- **Razão:** Código de inicialização executado apenas no startup
- **Linhas:** 12-25
- **Justificativa:** Teste exigiria ambiente completo de deploy

### Tratamento de Exceções Raras (error_handler.py)
- **Razão:** Exceções de sistema pouco prováveis
- **Linhas:** 78-92
- **Justificativa:** Cenários difíceis de reproduzir em ambiente de teste
```

2.3 Ferramentas de Cobertura

Por Linguagem:

Linguagem	Ferramenta	Comando
Python	coverage.py	`coverage run -m pytest && coverage report`
JavaScript	Jest + Istanbul	`jest --coverage`
Java	JaCoCo	Configurar no pom.xml/build.gradle
C/C++	gcov/lcov	`gcc --coverage && gcov arquivo.c`

2.4 Entregáveis de Cobertura

- [ ] Relatório HTML de cobertura (pasta `tests/coverage-results/`)
- [ ] Screenshot do relatório mostrando métricas principais
- [ ] Documento justificando código não coberto (se cobertura < 80%)

---

3. Técnicas de Depuração (Peso: 15%)

3.1 Log de Bugs e Correções

Crie um arquivo `docs/debugging-log.md` documentando:

Template de Registro de Bug:

```
## Bug #1: Erro ao Calcular Desconto
```

```
### Identificação
- **Data:** 2025-10-15
- **Reportado por:** Teste automatizado
- **Severidade:** Alta
- **Módulo:** services/carrinho.py

### Descrição
Ao aplicar desconto de 20% em produtos, o valor final está incorreto
para carrinhos com múltiplos produtos.

### Reprodução
1. Adicionar 3 produtos ao carrinho
2. Aplicar cupom DESC20
3. Resultado esperado: R$ 240,00
4. Resultado obtido: R$ 300,00

### Investigação
**Técnica utilizada:** Debugger + logging estratégico

**Código problemático:**
```python
def aplicar_desconto(self, cupom):
    for item in self.itens:
        item.preco = item.preco * 0.8  # BUG: modifica preço original!
    return self.calcular_total()
```

Causa raiz: Método estava modificando o preço original dos produtos ao invés de calcular desconto no total.

Correção

```python
def aplicar_desconto(self, cupom):
    subtotal = self.calcular_total()
    return subtotal * 0.8  # Aplica desconto no total
```

Verificação

- ✓ Teste automatizado passou
- ✓ Teste manual confirmou correção
- ✓ Novos testes adicionados para evitar regressão

Lições Aprendidas

- Sempre testar modificações de estado
- Adicionar testes para casos com múltiplos itens
- Preferir cálculos imutáveis
```

````
#### 3.2 Técnicas de Depuração Utilizadas

Documente pelo menos **3 técnicas diferentes** que você utilizou:

**Exemplos de técnicas:**
- Uso de debugger (breakpoints, step-by-step)
- Logging estratégico
- Testes automatizados para isolar problema
- Binary search (desabilitar partes do código)
- Análise de stack trace
- Rubber duck debugging
- Git bisect para encontrar commit problemático

#### 3.3 Requisitos Mínimos

- [ ] Documentar pelo menos **3 bugs diferentes** encontrados e corrigidos
- [ ] Incluir código antes/depois da correção
- [ ] Explicar técnica de depuração utilizada
- [ ] Adicionar testes que detectam o bug

---

### 4. Análise de Desempenho e Detecção de Gargalos (Peso: 20%)

#### 4.1 Análise de Performance

Identifique e otimize pelo menos **2 gargalos de desempenho** no seu
código.

**Template de Análise:**
```markdown
## Gargalo #1: Busca Linear em Lista Grande

### Identificação
- **Módulo:** services/busca.py
- **Função:** buscar_produto_por_nome()
- **Problema:** Lentidão com > 1000 produtos

### Medição Inicial
**Ferramenta:** timeit / cProfile

```python
import timeit

tempo = timeit.timeit(
    'buscar_produto_por_nome("Notebook")',
    setup='from services.busca import buscar_produto_por_nome',
    number=1000
)
print(f"Tempo médio: {tempo/1000:.4f}s")
````

Resultado: 0.0234s por busca (muito lento!)

Código Original

```
def buscar_produto_por_nome(nome):
    for produto in todos_produtos:  # O(n) - linear
        if produto.nome == nome:
            return produto
    return None
```

Análise

- **Complexidade:** O(n) - percorre toda lista
- **Impacto:** Aumenta linearmente com número de produtos
- **Gargalo:** Busca sequencial sem índice

Otimização Aplicada

```
# Criar índice em memória
produtos_por_nome = {p.nome: p for p in todos_produtos}

def buscar_produto_por_nome(nome):
    return produtos_por_nome.get(nome)  # O(1) - hash lookup
```

Medição Final

Resultado: 0.0001s por busca (234x mais rápido!)

Ganho de Performance

- **Redução de tempo:** 99.6%
- **Escalabilidade:** Agora constante O(1) independente do tamanho

Trade-offs

- ✓ Ganho: Busca muito mais rápida
- ✗ Custo: Uso adicional de memória
- ⚠ Atenção: Índice precisa ser atualizado quando produtos mudam

```


#### 4.2 Ferramentas de Profiling

**Recomendadas por linguagem:**

| Linguagem | Ferramenta | Uso |
|-----------|------------|-----|
| Python | cProfile, line_profiler | `python -m cProfile script.py` |
```

```
| JavaScript | Chrome DevTools, clinic.js | F12 → Performance |
| Java | JProfiler, VisualVM | Analisar CPU e memória |
| C/C++ | gprof, Valgrind (callgrind) | `gcc -pg && gprof` |

#### 4.3 Métricas de Performance

Documente para cada otimização:
- **Tempo de execução antes vs depois**
- **Complexidade algorítmica (Big O)**
- **Uso de memória (se relevante)**
- **Trade-offs da otimização**

---

### 5. Gerenciamento de Memória (Peso: 15% - quando aplicável)

#### 5.1 Análise de Memória

**Para linguagens com gerenciamento manual (C/C++):**

Utilize ferramentas como **Valgrind** para detectar:
- Memory leaks (vazamentos)
- Invalid memory access
- Use after free
- Double free

**Exemplo de relatório:**
```markdown
## Análise de Memória - Valgrind

### Execução Inicial
```bash
valgrind --leak-check=full ./meu_programa
```

Resultado:

```
HEAP SUMMARY:
    in use at exit: 1,024 bytes in 10 blocks
  total heap usage: 100 allocs, 90 frees, 10,240 bytes allocated

LEAK SUMMARY:
    definitely lost: 1,024 bytes in 10 blocks
```

Problema Identificado

```
// Código com memory leak
char* criar_mensagem(const char* texto) {
    char* msg = malloc(strlen(texto) + 1);
    strcpy(msg, texto);
```

```
    return msg;  // Caller precisa fazer free!
}
```

Correção

```
// Solução: documentação + uso correto
char* criar_mensagem(const char* texto) {
    // NOTA: Caller é responsável por liberar memória
    char* msg = malloc(strlen(texto) + 1);
    if (msg != NULL) {
        strcpy(msg, texto);
    }
    return msg;
}

// No código que chama:
char* msg = criar_mensagem("Hello");
// ... usar msg ...
free(msg);  // Libera memória
```

```
**Para linguagens com GC (Python, Java, JavaScript):**

Documente otimizações de uso de memória:
- Uso eficiente de estruturas de dados
- Evitar manter referências desnecessárias
- Generators/iteradores para grandes volumes
- Cache limitado (LRU cache)

---

## Estrutura de Documentação U3

### Arquivos Obrigatórios
```

entrega-u3/ ├── README.md (visão geral do projeto) ├── docs/ | ├── testing-report.md (relatório de testes) | ├── coverage-report.md (análise de cobertura) | ├── debugging-log.md (bugs encontrados/corrigidos) | ├── performance-analysis.md (gargalos e otimizações) | └── memory-analysis.md (se aplicável) ├── tests/ | ├── unit/ | └── coverage-results/ (relatórios HTML) ├── src/ | └── (código fonte refatorado) └── video-presentation.md (link para vídeo)

```
### Template do Relatório Principal (testing-report.md)

```markdown
```
```

```
# Relatório de Testes e Qualidade - [Nome do Projeto]

## 1. Suite de Testes

### 1.1 Visão Geral
- **Total de testes:** 15
- **Testes unitários:** 15
- **Status:** ✓ Todos passando

### 1.2 Estatísticas de Execução
- **Tempo total:** 8.4s
- **Testes mais lentos:**
  - test_processamento_completo: 2.1s
  - test_validacao_complexa: 1.8s

### 1.3 Organização dos Testes
[Descrever estrutura de pastas e convenções]

## 2. Cobertura de Código

### 2.1 Métricas Gerais
- **Cobertura de linhas:** 78%
- **Cobertura de branches:** 65%
- **Arquivos com 100% cobertura:** 8
- **Arquivos com < 50% cobertura:** 2

### 2.2 Cobertura por Módulo
| Módulo | Linhas | Branches | Status |
|--------|--------|----------|--------|
| models/ | 92% | 85% | ✓ Excelente |
| services/ | 81% | 72% | ✓ Bom |
| controllers/ | 65% | 48% | ⚠ Melhorar |
| utils/ | 100% | 100% | ✓ Perfeito |

### 2.3 Código Não Coberto
[Justificar linhas não cobertas]

## 3. Bugs e Depuração

### 3.1 Resumo
- **Bugs encontrados:** 5
- **Bugs corrigidos:** 5
- **Bugs conhecidos:** 0

### 3.2 Técnicas de Depuração Utilizadas
[Descrever técnicas e ferramentas]

## 4. Análise de Performance

### 4.1 Gargalos Identificados
- [Listar gargalos]

### 4.2 Otimizações Realizadas
- [Detalhar otimizações]
```

```
### 4.3 Ganhos Obtidos
[Métricas before/after]

## 5. Gerenciamento de Memória

### 5.1 Análise Realizada
[Se aplicável]

### 5.2 Problemas Encontrados e Corrigidos
[Detalhar]

## 6. Ferramentas Utilizadas

| Categoria | Ferramenta | Versão |
|-----------|------------|--------|
| Testes | pytest | 7.4.0 |
| Cobertura | coverage.py | 7.3.0 |
| Profiling | cProfile | - |
| Linting | pylint | 2.17.0 |

## 7. Lições Aprendidas

### 7.1 O que funcionou bem
[Descrever]

### 7.2 Desafios enfrentados
[Descrever]

### 7.3 Melhorias futuras
[Listar]
```

Critérios de Avaliação Detalhados - U3

Critério	Peso	Detalhamento
Automação de Testes	30%	Suite completa, qualidade dos testes, organização
Cobertura de Código	20%	Métricas atingidas, relatórios, justificativas
Depuração Documentada	15%	Bugs identificados, técnicas, correções
Análise de Desempenho	20%	Gargalos encontrados, otimizações, métricas
Gerenciamento de Memória	15%	Análise realizada, problemas corrigidos (quando aplicável)

Detalhamento por Critério

Automação de Testes (30%)

- **Quantidade e qualidade (12%):**

    - Mínimo 10 testes unitários
    - Seguem padrão AAA
    - Nomes descritivos
    - Cobrem casos de sucesso, falha e edge cases

- **Organização e estrutura (8%):**

    - Testes bem organizados em pastas
    - Fixtures reutilizáveis (quando aplicável)
    - Setup/teardown apropriados

- **Princípios FIRST (10%):**

    - Fast: suite executa rapidamente
    - Independent: testes independentes
    - Repeatable: resultados consistentes
    - Self-validating: assertions claros
    - Timely: cobrem funcionalidades

Cobertura de Código (20%)

- **Métricas atingidas (10%):**

    - Cobertura de linhas ≥ 70%
    - Cobertura de branches ≥ 60%
    - Módulos críticos ≥ 85%

- **Relatórios e análise (10%):**

    - Relatório HTML incluído
    - Justificativa para código não coberto
    - Evolução da cobertura documentada

Depuração Documentada (15%)

- **Identificação de bugs (5%):**

    - Pelo menos 3 bugs documentados
    - Descrição clara do problema

- **Técnicas aplicadas (5%):**

    - Uso de debugger
    - Logging estratégico
    - Testes para reprodução

- **Correções e aprendizado (5%):**

    - Código antes/depois
    - Testes para prevenir regressão
    - Lições aprendidas

Análise de Desempenho (20%)

- **Identificação de gargalos (8%):**
  - Pelo menos 2 gargalos encontrados
  - Uso de ferramentas de profiling
  - Medições before/after
- **Otimizações realizadas (12%):**
  - Melhorias implementadas
  - Ganhos de performance quantificados
  - Trade-offs documentados
  - Análise de complexidade (Big O)

Gerenciamento de Memória (15%)

- **Para C/C++ (15%):**
  - Análise com Valgrind
  - Memory leaks corrigidos
  - Uso adequado de malloc/free
- **Para outras linguagens (15%):**
  - Otimização de estruturas de dados
  - Uso eficiente de memória
  - Prevenção de referências circulares

Vídeo de Apresentação (10-12 minutos)

Estrutura Recomendada

Minutos 1-2: Recapitulação

- Visão do produto (breve)
- Evolução desde U1
- Principais funcionalidades do MVP

Minutos 3-4: Demonstração de Testes

- Mostrar execução da suite de testes
- Destacar testes mais importantes
- Mostrar relatório de cobertura

Minutos 5-6: Depuração

- Demonstrar 1-2 bugs interessantes
- Explicar como foram encontrados
- Mostrar correção

Minutos 7-8: Performance

- Demonstrar gargalos encontrados
- Mostrar métricas before/after
- Explicar otimizações

Minutos 9-10: Qualidade Final

- Métricas gerais do projeto
- Principais melhorias desde U1
- Dívida técnica conhecida

Minutos 11-12: Lições Aprendidas

- Principais desafios
- O que funcionou bem
- Próximos passos

Checklist de Entrega

Documentação

- [ ] README.md atualizado com instruções de teste
- [ ] testing-report.md completo
- [ ] coverage-report.md com métricas e justificativas
- [ ] debugging-log.md com pelo menos 3 bugs
- [ ] performance-analysis.md com 2+ otimizações
- [ ] memory-analysis.md (se aplicável)

Código

- [ ] Mínimo 10 testes unitários
- [ ] Todos os testes passando
- [ ] Cobertura ≥ 70% linhas
- [ ] Cobertura ≥ 60% branches
- [ ] Código refatorado desde U1

Testes e Análise

- [ ] Suite de testes executa sem erros
- [ ] Relatório HTML de cobertura incluído
- [ ] Screenshots de métricas
- [ ] Gargalos de performance identificados
- [ ] Otimizações implementadas e medidas

Apresentação

- [ ] Vídeo de 10-12 minutos
- [ ] Demonstração de testes rodando

- [ ] Explicação de bugs e correções
- [ ] Métricas de performance
- [ ] Link público funcionando

Entrega

- [ ] Arquivo ZIP com estrutura correta
- [ ] Link para repositório GitHub
- [ ] Link para vídeo (YouTube/Drive)
- [ ] Todos os PDFs incluídos

Observações Importantes

Trabalho em Grupo

- Projetos podem ser individuais ou em grupos de 2-4 pessoas
- Em grupos, descrever claramente contribuição de cada membro
- Todos devem estar aptos a explicar qualquer parte do código

Plágio e Integridade

- Testes copiados sem adaptação serão penalizados
- Código gerado por IA deve ser revisado e compreendido
- Cite fontes de código adaptado de terceiros

Tecnologias

- Linguagem de programação: livre escolha
- Frameworks de teste: escolher apropriado para linguagem
- Ferramentas de cobertura: usar oficial da linguagem

Suporte

- **Atendimento:** Segundas 14h-16h (online)
- **Discord:** https://discord.gg/bbMFJBQRT8
- **GitHub:** https://github.com/fmarquesfilho/bpp-2025-2
- **Plantão de dúvidas:** 05/12 no horário da aula (online)

Data e Formato de Entrega

Data limite: 05/12/2025 às 23:59
Plataforma: SIGAA - Tarefa "Entrega U3"
Formato: Arquivo ZIP único contendo:

```
projeto-nome-grupo.zip
├── README.md
├── docs/
├── src/
```

```
├── tests/
└── links.txt (repositório + vídeo)
```

Nome do arquivo: `u3-[nome-projeto]-[nomes-integrantes].zip`
Exemplo: `u3-taskmanager-joao-maria.zip`

Recursos de Apoio

Documentação Oficial

- **pytest:** https://docs.pytest.org/
- **JUnit:** https://junit.org/junit5/docs/current/user-guide/
- **Jest:** https://jestjs.io/docs/getting-started
- **Coverage.py:** https://coverage.readthedocs.io/

Ferramentas de Análise

- **SonarCloud:** https://sonarcloud.io/ (análise gratuita)
- **Codecov:** https://codecov.io/ (cobertura online)
- **GitHub Actions:** Automação de testes no CI/CD

Materiais Complementares

- Clean Code, Cap. 9 - Unit Tests
- The Art of Unit Testing, Roy Osherove
- Working Effectively with Legacy Code, Michael Feathers

Catálogos

- Code Smells: https://luzkan.github.io/smells/
- Refactoring Catalog: https://refactoring.com/catalog/
- Test Smells: http://testsmells.org/

Dúvidas Frequentes

P: Posso usar o mesmo projeto da U1? R: Sim! A U3 é a evolução natural do projeto da U1, adicionando testes e análises de qualidade.

P: Preciso ter 100% de cobertura? R: Não. Mínimo de 70% é suficiente. Foque em testar código crítico e complexo.

P: E se meu projeto não tem gargalos de performance? R: Identifique oportunidades de otimização mesmo que o código já seja rápido. Exemplo: trocar busca linear por hash, usar cache, etc.

P: Linguagens compiladas precisam de análise de memória? R: Para C/C++, sim. Para outras linguagens, foque em uso eficiente de estruturas de dados.

P: Quantos bugs preciso documentar? R: Mínimo de 3 bugs reais que você encontrou e corrigiu durante o desenvolvimento.

P: Posso usar testes de outras pessoas como base? R: Pode se inspirar, mas deve adaptar para seu contexto e citar a fonte.

P: Como executar testes sem instalar nada? R: Use ambientes online como Replit, CodeSandbox, ou GitHub Codespaces.

P: Por que não são exigidos testes de integração? R: Para simplificar a entrega e focar na qualidade dos testes unitários. Testes de integração são opcionais e podem ser incluídos como diferencial, mas não são obrigatórios.

Boa sorte!

Código com boa cobertura de testes é código confiável.
Testes automatizados são investimento, não custo.