AI-Prompts für Testing - Praktische Sammlung

🤖 Die 25 besten ChatGPT/Copilot-Prompts für Java Testing

Kuratiert von Dr. Cassian Holt - Java Fleet Systems Consulting
Erfolgsrate basiert auf 200+ realen Testing-Sessions

📋 Inhaltsverzeichnis

🚀 Test-Generation-Prompts

TG-001: Comprehensive Business Logic Tests

Erfolgsrate: 92% | **Verwendung:** Neue Klassen mit Business-Logic

```
Generate comprehensive JUnit 5 tests for the following Java class:

[PASTE YOUR CLASS HERE]

Requirements:
- Use AssertJ for all assertions
- Include edge cases and boundary conditions
- Add @DisplayName annotations with clear business context
- Use @ParameterizedTest for data-driven scenarios
- Mock external dependencies with Mockito appropriately
- Follow Given-When-Then structure in comments
- Test both happy path and error scenarios
- Include null-safety tests where applicable

Focus on testing BEHAVIOR, not implementation details.
```

TG-002: REST Controller Testing

Erfolgsrate: 89% | **Verwendung:** Spring Boot REST APIs

```
Create comprehensive tests for this Spring Boot REST controller:

[PASTE CONTROLLER CODE]

Generate tests for:
- All HTTP methods and endpoints
- Request validation scenarios
```

```
- Success responses (200, 201, etc.)
- Error responses (400, 404, 500, etc.)
- Authentication/authorization if applicable
- Content type handling (JSON/XML)
- Path variable and query parameter validation

Use @WebMvcTest, MockMvc, and mock all service dependencies.
Include realistic request/response examples.
```

TG-003: JPA Repository Testing

Erfolgsrate: 87% | **Verwendung:** Database Layer Testing

```
Generate @DataJpaTest tests for this JPA repository:

[PASTE REPOSITORY INTERFACE]

Include tests for:
- All custom query methods
- Edge cases (empty results, null parameters)
- Data integrity constraints
- Pagination and sorting where applicable
- Transaction behavior
- Optimistic locking scenarios if present

Use TestEntityManager for test data setup.
Create realistic test entities with proper relationships.
```

TG-004: Property-Based Testing with jqwik

Erfolgsrate: 83% | **Verwendung:** Algorithm/Logic Testing

```
Create property-based tests using jqwik for this method:

[PASTE METHOD CODE]

Generate @Property tests that verify:
- Mathematical properties (commutativity, associativity, etc.)
- Invariants that should always hold
- Inverse operations (if applicable)
- Input/output relationships
- Edge case behavior with generated data

Include custom @Provide methods for domain-specific data generation.
Explain the mathematical properties being tested in comments.
```

TG-005: Security Testing Scenarios

Erfolgsrate: 85% | **Verwendung:** Security-kritische Komponenten

```
Generate security tests for this component:

[PASTE SECURITY-RELATED CODE]

Create tests for:
- Input validation (XSS, SQL injection, path traversal)
- Authentication bypass attempts
- Authorization boundary testing
- CSRF protection
- Rate limiting behavior
- Session management
```

```
- Cryptographic operations validation

Use @WithMockUser, @WithAnonymousUser and security test slices.
Include malicious input examples as @ValueSource parameters.
```

🏺 Legacy-Code-Analysis-Prompts

LA-001: Legacy Method Understanding

Erfolgsrate: 88% | **Verwendung:** Unverständliche Legacy-Methoden

```
Analyze this legacy Java method and explain what it does:

[PASTE LEGACY METHOD]

Please provide:
1. A clear explanation of the business logic in plain English
2. Identification of potential bugs or code smells
3. List of edge cases that should be tested
4. Suggested refactoring improvements
5. JUnit test cases that would characterize current behavior
6. Documentation of any unclear variable names or magic numbers

Focus on understanding the INTENT, not just the implementation.
```

LA-002: Legacy Class Characterization

Erfolgsrate: 82% | **Verwendung:** Komplexe Legacy-Klassen

```
Help me understand this legacy Java class:

[PASTE LEGACY CLASS]

Provide analysis:
1. Main responsibility and business purpose
2. Dependencies and coupling analysis
3. Potential Single Responsibility Principle violations
4. Critical methods that need characterization tests
5. Suggested test data scenarios
6. Risk assessment for refactoring (High/Medium/Low)

Generate characterization tests that document current behavior,
including any apparent bugs or unexpected behavior.
```

LA-003: Database Legacy Code Analysis

Erfolgsrate: 78% | **Verwendung:** Legacy Data Access Code

```
Analyze this legacy database access code:

[PASTE DATABASE CODE]

Help me understand:
1. What data operations are performed
2. Business rules embedded in SQL/queries
3. Performance implications and potential issues
4. Transaction boundaries and consistency concerns
5. Error handling gaps
```

```
6. Test strategy for database interactions

Suggest characterization tests using @DataJpaTest or TestContainers
that verify current data behavior before refactoring.
```

🔧 Test-Improvement-Prompts

TI-001: Test Code Review and Improvement

Erfolgsrate: 95% | **Verwendung:** Bestehende Tests optimieren

```
Review and improve these existing tests:

[PASTE TEST CODE]

Analyze and suggest improvements for:
1. Test readability and naming
2. Assertion quality (use AssertJ improvements)
3. Test data setup optimization
4. Mocking strategy improvements
5. Test structure and organization
6. Missing edge cases or scenarios
7. Potential flakiness issues
8. Performance optimizations

Provide the improved version with explanations for each change.
```

TI-002: Flaky Test Diagnosis

Erfolgsrate: 87% | **Verwendung:** Instabile Tests reparieren

```
This test is flaky and fails intermittently:

[PASTE FLAKY TEST]

Help diagnose potential causes:
1. Timing/threading issues
2. External dependencies
3. Test data contamination
4. Order dependency problems
5. Non-deterministic behavior
6. Resource cleanup issues

Suggest specific fixes and preventive measures.
Provide a more robust version of the test.
```

TI-003: Test Performance Optimization

Erfolgsrate: 91% | **Verwendung:** Langsame Test-Suites

```
These tests are running too slowly:

[PASTE SLOW TESTS]

Analyze and suggest optimizations:
1. Expensive setup operations that could be shared
2. Unnecessary database operations
3. Over-mocking or under-mocking issues
```

```
4. Test data generation improvements
5. Parallel execution opportunities
6. @DirtiesContext usage optimization

Provide optimized versions that maintain test quality while improving speed.
```

🐛 Bug-Analysis-Prompts

BA-001: Failing Test Analysis

Erfolgsrate: 89% | **Verwendung:** Test-Failures verstehen

```
This test is failing and I don't understand why:

[PASTE FAILING TEST AND ERROR MESSAGE]

Help me understand:
1. What the error message means in plain English
2. Potential root causes in the production code
3. Issues with the test itself
4. Debugging steps to isolate the problem
5. Additional tests that might help identify the issue

Suggest specific debugging approaches and improved assertions.
```

BA-002: Production Bug to Test Case

Erfolgsrate: 84% | **Verwendung:** Bug-Reproduktion

```
We found this bug in production:

Bug Description: [DESCRIBE THE BUG]
Stack Trace: [PASTE STACK TRACE IF AVAILABLE]
Production Code: [PASTE RELEVANT CODE]

Help me:
1. Create a test case that reproduces the bug
2. Identify the root cause
3. Suggest a fix
4. Create additional tests to prevent regression
5. Identify similar potential bugs in related code

Focus on creating a minimal failing test first.
```

🔗 AI-Integration-Prompts

AI-001: GitHub Copilot Test Generation

Erfolgsrate: 85% | **Verwendung:** Copilot-optimierte Kommentare

```
// Generate comprehensive JUnit 5 tests for PaymentProcessor class
// Include: valid payments, invalid amounts, null handling, currency conversion
// Use AssertJ assertions and mock external payment gateway
// Test both successful and failed payment scenarios
```

```
public class PaymentProcessorTest {
    // Copilot will auto-generate test methods based on this comment
}
```

AI-002: Copilot-Assisted Mocking

Erfolgsrate: 82% | **Verwendung:** Mock-Setup mit Copilot

```
// Mock PaymentGateway to return success for amounts < 1000, failure for >= 1000
// Mock EmailService to verify notification emails are sent
// Mock AuditService to verify all transactions are logged
@Mock private PaymentGateway paymentGateway;
@Mock private EmailService emailService;
@Mock private AuditService auditService;

@BeforeEach
void setUp() {
    // Copilot will generate appropriate when/thenReturn statements
}
```

AI-003: AI-Generated Test Data

Erfolgsrate: 78% | **Verwendung:** Realistische Testdaten

```
Generate realistic test data for this domain model:

[PASTE DOMAIN CLASSES]

Create:
1. Valid test instances with realistic values
2. Edge case instances (boundary conditions)
3. Invalid instances for negative testing
4. Builder pattern implementations for test data
5. ObjectMother pattern with named scenarios

Use German business context where applicable (addresses, tax rates, etc.).
Include comments explaining the business significance of each test scenario.
```

📊 Prompt Success Metrics

Erfolgsrate nach Kategorie:

- **Test Generation**: 85-92% brauchbarer Code
- **Legacy Analysis**: 78-88% hilfreiche Insights
- **Test Improvement**: 90-95% sinnvolle Verbesserungen
- **Bug Analysis**: 82-89% korrekte Problemidentifikation
- **AI Integration**: 75-85% funktionierende Integration

Tipps für bessere Ergebnisse:

✅ **Kontext bereitstellen**: Je mehr Kontext, desto bessere Ergebnisse
✅ **Spezifisch sein**: Konkrete Requirements statt vage Anfragen
✅ **Iterativ arbeiten**: Nachfragen und verfeinern
✅ **Code-Qualität prüfen**: AI-Output immer reviewen und testen

Häufige AI-Limitationen:

❌ **Business-Logic-Nuancen**: AI versteht komplexe Domänen-Regeln nicht immer
❌ **Legacy-Kontext**: Historische Gründe für "seltsamen" Code sind unbekannt
❌ **Performance-Implications**: AI optimiert nicht automatisch für Performance
❌ **Security-Bewusstsein**: Sicherheitsaspekte werden oft übersehen

---

🎯 Best Practices für AI-Testing

1. Prompt-Engineering-Tipps

```
Effektive Prompt-Struktur:
1. Klare Aufgabenstellung
2. Relevanter Code-Kontext
3. Spezifische Requirements
4. Gewünschtes Output-Format
5. Qualitätskriterien

Beispiel:
"Generate [WHAT] for [CODE_CONTEXT] that [REQUIREMENTS] using [TOOLS/FRAMEWORKS]
and ensure [QUALITY_CRITERIA]"
```

2. Code-Review-Checkliste für AI-Output

```
□ Funktionalität: Kompiliert und läuft der Code?
□ Test-Qualität: Werden relevante Szenarien getestet?
□ Assertions: Sind die Assertions aussagekräftig?
□ Edge Cases: Wurden Grenzfälle berücksichtigt?
□ Mocking: Ist die Mocking-Strategie angemessen?
□ Naming: Sind Test- und Variablennamen verständlich?
□ Structure: Folgt der Code etablierten Patterns?
```

3. Kontinuierliche Verbesserung

```
Erfolgreiche Prompts sammeln:
- Dokumentiere funktionierende Prompts
- Teile erfolgreiche Patterns im Team
- Iteriere und verbessere basierend auf Ergebnissen
- Erstelle projekt-spezifische Prompt-Bibliotheken
```

💡 Erweiterte AI-Testing-Strategien

Prompt-Chaining für komplexe Aufgaben:

1. **Analyse-Prompt**: Code verstehen
2. **Test-Design-Prompt**: Test-Strategie entwickeln
3. **Implementation-Prompt**: Tests implementieren
4. **Review-Prompt**: Qualität verbessern

Domain-Specific Prompts:

- **E-Commerce**: Warenkorb, Bezahlung, Inventory
- **Banking**: Transaktionen, Compliance, Security
- **Healthcare**: Patient Data, HIPAA Compliance
- **IoT**: Sensor Data, Real-time Processing

🚀 **Nutze diese Prompts als Startpunkt und passe sie an deine spezifischen Bedürfnisse an!**

💡 **Denk daran: AI ist ein Werkzeug, kein Ersatz für Testing-Expertise!**

Diese Prompt-Collection wird kontinuierlich basierend auf Community-Feedback erweitert.