

AI-ASSISTED JAVA DEVELOPMENT

Module 3 — Harwell Prompt Engineering

LEARNING OBJECTIVES

By the end of this module you will be able to:

-
-
-
-
-

LEARNING OBJECTIVES

By the end of this module you will be able to:

- Use AI to generate boilerplate, entities, and service layers in Spring Boot 3
-
-
-
-

LEARNING OBJECTIVES

By the end of this module you will be able to:

- Use AI to generate boilerplate, entities, and service layers in Spring Boot 3
- Use AI to explain legacy code and unfamiliar codebases
-
-
-

LEARNING OBJECTIVES

By the end of this module you will be able to:

- Use AI to generate boilerplate, entities, and service layers in Spring Boot 3
- Use AI to explain legacy code and unfamiliar codebases
- Apply refactoring strategies with AI: modernizing code safely
-
-

LEARNING OBJECTIVES

By the end of this module you will be able to:

- Use AI to generate boilerplate, entities, and service layers in Spring Boot 3
- Use AI to explain legacy code and unfamiliar codebases
- Apply refactoring strategies with AI: modernizing code safely
- Generate unit tests (JUnit 5/Mockito) with AI assistance
-

LEARNING OBJECTIVES

By the end of this module you will be able to:

- Use AI to generate boilerplate, entities, and service layers in Spring Boot 3
- Use AI to explain legacy code and unfamiliar codebases
- Apply refactoring strategies with AI: modernizing code safely
- Generate unit tests (JUnit 5/Mockito) with AI assistance
- Evaluate AI output for correctness, style, and fit before applying

BRIDGE FROM MODULE 2

What we learned last time:

-
-
-

BRIDGE FROM MODULE 2

What we learned last time:

- How to prompt effectively (clarity, context, constraints)
-
-

BRIDGE FROM MODULE 2

What we learned last time:

- How to prompt effectively (clarity, context, constraints)
- Zero-shot, few-shot, chain-of-thought techniques
-

BRIDGE FROM MODULE 2

What we learned last time:

- **How** to prompt effectively (clarity, context, constraints)
- Zero-shot, few-shot, chain-of-thought techniques
- Iterative refinement

The problem:

-
-
-
-

Today: Learn **what** to prompt for in Java development.

The problem:

- **X** You know how to prompt, but what should you ask for?
-
-
-

Today: Learn **what** to prompt for in Java development.

The problem:

- ❌ You know how to prompt, but what should you ask for?
- ❌ How do I explain legacy code I don't understand?
-
-

Today: Learn **what** to prompt for in Java development.

The problem:

- ❌ You know how to prompt, but what should you ask for?
- ❌ How do I explain legacy code I don't understand?
- ❌ Can AI help me refactor safely?
-

Today: Learn **what** to prompt for in Java development.

The problem:

- ❌ You know how to prompt, but what should you ask for?
- ❌ How do I explain legacy code I don't understand?
- ❌ Can AI help me refactor safely?
- ❌ How do I generate tests that actually work?

Today: Learn **what** to prompt for in Java development.

CODE GENERATION: THE PROBLEM

Scenario: You need a Book entity with JPA annotations, repository, and service method

Manual approach:

-
-
-

Result: Slow, repetitive, error-prone

CODE GENERATION: THE PROBLEM

Scenario: You need a Book entity with JPA annotations, repository, and service method

Manual approach:

- **X** Write entity, add annotations, create repository interface, write service method
-
-

Result: Slow, repetitive, error-prone

CODE GENERATION: THE PROBLEM

Scenario: You need a Book entity with JPA annotations, repository, and service method

Manual approach:

- **X** Write entity, add annotations, create repository interface, write service method
- **X** Time-consuming: 15-20 minutes for boilerplate
-

Result: Slow, repetitive, error-prone

CODE GENERATION: THE PROBLEM

Scenario: You need a Book entity with JPA annotations, repository, and service method

Manual approach:

- **✗** Write entity, add annotations, create repository interface, write service method
- **✗** Time-consuming: 15-20 minutes for boilerplate
- **✗** Error-prone: Easy to miss annotations, wrong return types

Result: Slow, repetitive, error-prone

CODE GENERATION: THE SOLUTION

AI approach:

-
-
-

CODE GENERATION: THE SOLUTION

AI approach:

-  Generate complete entity with proper annotations
-
-

CODE GENERATION: THE SOLUTION

AI approach:

-  Generate complete entity with proper annotations
-  Fast: 30 seconds vs. 15 minutes
-

CODE GENERATION: THE SOLUTION

AI approach:

-  Generate complete entity with proper annotations
-  Fast: 30 seconds vs. 15 minutes
-  Consistent: Follows Spring Boot patterns

Prompt example:

“Generate a Spring Boot 3 JPA entity for a Book with fields: id (Long, primary key), title (String), author (String), isbn (String, unique), publishedYear (Integer). Use Java 17.”

CODE GENERATION: PROGRESSIVE BUILDING

Start simple:

- 1.
- 2.
- 3.
- 4.

Build incrementally — don't ask for everything at once.

CODE GENERATION: PROGRESSIVE BUILDING

Start simple:

1. Entity only
- 2.
- 3.
- 4.

Build incrementally — don't ask for everything at once.

CODE GENERATION: PROGRESSIVE BUILDING

Start simple:

1. Entity only
2. Add repository interface
- 3.
- 4.

Build incrementally — don't ask for everything at once.

CODE GENERATION: PROGRESSIVE BUILDING

Start simple:

1. Entity only
2. Add repository interface
3. Add service class with constructor injection
- 4.

Build incrementally — don't ask for everything at once.

CODE GENERATION: PROGRESSIVE BUILDING

Start simple:

1. Entity only
2. Add repository interface
3. Add service class with constructor injection
4. Add controller with REST endpoint

Build incrementally — don't ask for everything at once.

CODE GENERATION: REFINEMENT

Initial output (maybe uses @Autowired):

```
@Autowired  
private BookRepository repository;
```

Refine prompt:

*“Use constructor injection, not
@Autowired”*

Improved output:

```
private final BookRepository repository;  
  
public BookService(BookRepository repository) {  
    this.repository = repository;  
}
```

Key point: Review and refine iteratively.

WHEN TO GENERATE VS. WRITE MANUALLY

Generate with AI	Review carefully	Write manually
Boilerplate (entities, DTOs)	Business logic	Security/auth
CRUD operations	Complex algorithms	Payment processing
Repository interfaces	Performance-critical code	Compliance code

**Generate with
AI**

Review carefully

**Write
manually**

Service stubs

CODE EXPLANATION: THE PROBLEM

Scenario: You inherited a JSF/GWT component you don't understand

Problems:

-
-
-

CODE EXPLANATION: THE PROBLEM

Scenario: You inherited a JSF/GWT component you don't understand

Problems:

- **X** Hard to read: Old patterns, verbose syntax
-
-

CODE EXPLANATION: THE PROBLEM

Scenario: You inherited a JSF/GWT component you don't understand

Problems:

- **X** Hard to read: Old patterns, verbose syntax
- **X** No documentation: Why was it written this way?
-

CODE EXPLANATION: THE PROBLEM

Scenario: You inherited a JSF/GWT component you don't understand

Problems:

- **X** Hard to read: Old patterns, verbose syntax
- **X** No documentation: Why was it written this way?
- **X** Risk: Don't know what it does or what breaks if changed

CODE EXPLANATION: THE SOLUTION

Prompt example:

“Explain what this code does step by step. Why might it have been written this way? What are the dependencies?”

AI provides:



CODE EXPLANATION: THE SOLUTION

Prompt example:

“Explain what this code does step by step. Why might it have been written this way? What are the dependencies?”

AI provides:

-  Functional explanation

-

-

-

CODE EXPLANATION: THE SOLUTION

Prompt example:

“Explain what this code does step by step. Why might it have been written this way? What are the dependencies?”

AI provides:

-  Functional explanation
-  Historical context
-
-

CODE EXPLANATION: THE SOLUTION

Prompt example:

“Explain what this code does step by step. Why might it have been written this way? What are the dependencies?”

AI provides:

-  Functional explanation
-  Historical context
-  Dependency analysis
-

CODE EXPLANATION: THE SOLUTION

Prompt example:

“Explain what this code does step by step. Why might it have been written this way? What are the dependencies?”

AI provides:

-  Functional explanation
-  Historical context
-  Dependency analysis
-  Modern equivalent suggestions

CODE EXPLANATION: LAYERS OF UNDERSTANDING

Layer 1: “What does this do?” (functional explanation)

Layer 2: “Why was it written this way?” (historical context)

Layer 3: “What’s the modern equivalent?” (migration path)

Layer 4: “What are the risks if I change it?” (dependency analysis)

Progressive building: Start with “what,” build to “how to modernize.”

REFACTORING: THE PROBLEM

Example: Procedural method with loops, null checks,
verbose logic

Problems:

-
-
-

Result: Code that works but is hard to maintain

REFACTORING: THE PROBLEM

Example: Procedural method with loops, null checks,
verbose logic

Problems:

- **X** Hard to read: Nested loops, multiple null checks
-
-

Result: Code that works but is hard to maintain

REFACTORING: THE PROBLEM

Example: Procedural method with loops, null checks, verbose logic

Problems:

- **✗** Hard to read: Nested loops, multiple null checks
- **✗** Not modern: Java 8+ features not used
-

Result: Code that works but is hard to maintain

REFACTORING: THE PROBLEM

Example: Procedural method with loops, null checks,
verbose logic

Problems:

- **✗** Hard to read: Nested loops, multiple null checks
- **✗** Not modern: Java 8+ features not used
- **✗** Error-prone: Easy to introduce bugs

Result: Code that works but is hard to maintain

REFACTORING: THE SOLUTION

Prompt example:

“Refactor this method to use Java 17 Streams API. Extract null checks into filters. Improve readability while maintaining the same behavior.”

AI suggests:

-
-
-

REFACTORING: THE SOLUTION

Prompt example:

“Refactor this method to use Java 17 Streams API. Extract null checks into filters. Improve readability while maintaining the same behavior.”

AI suggests:

-  Stream-based refactor
-
-

REFACTORING: THE SOLUTION

Prompt example:

“Refactor this method to use Java 17 Streams API. Extract null checks into filters. Improve readability while maintaining the same behavior.”

AI suggests:

-  Stream-based refactor
-  More readable: Functional style
-

REFACTORING: THE SOLUTION

Prompt example:

“Refactor this method to use Java 17 Streams API. Extract null checks into filters. Improve readability while maintaining the same behavior.”

AI suggests:

-  Stream-based refactor
-  More readable: Functional style
-  Modern: Uses Java 17 features

REFACTORING STRATEGIES

Strategy 1: Extract method (break down large methods)

Strategy 2: Replace loops with Streams

Strategy 3: Add null-safety (Optional, null checks)

Strategy 4: Improve naming and structure

Progressive building: Start with one strategy, add more as needed.

REFACTORING: SAFETY FIRST

Always:

-
-
-

Example: Refactor that breaks a test → fix it → tests pass

Key point: Refactoring is safe when you have tests.

REFACTORING: SAFETY FIRST

Always:

-  Run tests before and after
-
-

Example: Refactor that breaks a test → fix it → tests pass

Key point: Refactoring is safe when you have tests.

REFACTORING: SAFETY FIRST

Always:

- ⚠ Run tests before and after
- ⚠ Review the diff carefully
-

Example: Refactor that breaks a test → fix it → tests pass

Key point: Refactoring is safe when you have tests.

REFACTORING: SAFETY FIRST

Always:

- ⚠ Run tests before and after
- ⚠ Review the diff carefully
- ⚠ Verify behavior hasn't changed

Example: Refactor that breaks a test → fix it → tests pass

Key point: Refactoring is safe when you have tests.

WHEN TO REFACTOR WITH AI

Good for AI refactoring	Be careful	Don't use AI
Readability improvements	Performance-critical code	Security-sensitive logic
Modernizing syntax		
Extracting methods		

**Good for AI
refactoring**

Be careful

Don't use AI

Reducing
duplication

UNIT TESTING: THE PROBLEM

Scenario: You need tests for a service method

Manual approach:

-
-
-

Result: Tests are written slowly or skipped

UNIT TESTING: THE PROBLEM

Scenario: You need tests for a service method

Manual approach:

- **X** Write test class, set up mocks, write assertions
-
-

Result: Tests are written slowly or skipped

UNIT TESTING: THE PROBLEM

Scenario: You need tests for a service method

Manual approach:

- **X** Write test class, set up mocks, write assertions
- **X** Time-consuming: 10-15 minutes per test method
-

Result: Tests are written slowly or skipped

UNIT TESTING: THE PROBLEM

Scenario: You need tests for a service method

Manual approach:

- **✗** Write test class, set up mocks, write assertions
- **✗** Time-consuming: 10-15 minutes per test method
- **✗** Easy to miss: Edge cases, null handling

Result: Tests are written slowly or skipped

UNIT TESTING: THE SOLUTION

Prompt example:

“Generate JUnit 5 and Mockito tests for this service method. Cover happy path, null input, and empty list scenarios. Use Java 17.”

AI generates:

-
-
-

UNIT TESTING: THE SOLUTION

Prompt example:

“Generate JUnit 5 and Mockito tests for this service method. Cover happy path, null input, and empty list scenarios. Use Java 17.”

AI generates:

-  Complete test class
-
-

UNIT TESTING: THE SOLUTION

Prompt example:

“Generate JUnit 5 and Mockito tests for this service method. Cover happy path, null input, and empty list scenarios. Use Java 17.”

AI generates:

-  Complete test class
-  Includes mocks setup
-

UNIT TESTING: THE SOLUTION

Prompt example:

“Generate JUnit 5 and Mockito tests for this service method. Cover happy path, null input, and empty list scenarios. Use Java 17.”

AI generates:

-  Complete test class
-  Includes mocks setup
-  Covers multiple scenarios

UNIT TESTING: PROGRESSIVE QUALITY

Level 1: Basic test (happy path only)

Level 2: Add edge cases (null, empty, exceptions)

Level 3: Add parameterized tests

Level 4: Add integration test suggestions

Build from basic to comprehensive.

UNIT TESTING: RUNNING AND FIXING

Generated test → Run it → One test fails

Diagnosis: What's wrong? (maybe mock not set up correctly)

Refine prompt:

“Fix the mock setup for the repository”

Re-run → Tests pass

Key point: Tests are iterative too — generate, run, refine.

TEST BEST PRACTICES

Cover:

-
-
-
-

Verify:

-
-

TEST BEST PRACTICES

Cover:

-  Happy path
-
-
-

Verify:

-
-

TEST BEST PRACTICES

Cover:

-  Happy path
-  Edge cases (null, empty, exceptions)
-
-

Verify:

-
-

TEST BEST PRACTICES

Cover:

-  Happy path
-  Edge cases (null, empty, exceptions)
-  Proper mocking (Mockito)
-

Verify:

-
-

TEST BEST PRACTICES

Cover:

-  Happy path
-  Edge cases (null, empty, exceptions)
-  Proper mocking (Mockito)
-  Clear assertions

Verify:

-
-

TEST BEST PRACTICES

Cover:

-  Happy path
-  Edge cases (null, empty, exceptions)
-  Proper mocking (Mockito)
-  Clear assertions

Verify:

-  Tests actually test the right thing
-

TEST BEST PRACTICES

Cover:

-  Happy path
-  Edge cases (null, empty, exceptions)
-  Proper mocking (Mockito)
-  Clear assertions

Verify:

-  Tests actually test the right thing
-  Don't trust tests without running them

REVIEW BEFORE PASTE: THE PROBLEM

Example: AI generates code with wrong API

Problems:

-
-
-

Result: Code that doesn't work or introduces bugs

REVIEW BEFORE PASTE: THE PROBLEM

Example: AI generates code with wrong API

Problems:

- **✘** Wrong version: Uses Spring Boot 2 instead of 3
-
-

Result: Code that doesn't work or introduces bugs

REVIEW BEFORE PASTE: THE PROBLEM

Example: AI generates code with wrong API

Problems:

- **✗** Wrong version: Uses Spring Boot 2 instead of 3
- **✗** Wrong patterns: Doesn't match team style
-

Result: Code that doesn't work or introduces bugs

REVIEW BEFORE PASTE: THE PROBLEM

Example: AI generates code with wrong API

Problems:

- **✗** Wrong version: Uses Spring Boot 2 instead of 3
- **✗** Wrong patterns: Doesn't match team style
- **✗** Security issue: Missing validation

Result: Code that doesn't work or introduces bugs

REVIEW BEFORE PASTE: EVALUATION CHECKLIST

Before applying AI output, check:

-
-
-
-
-

REVIEW BEFORE PASTE: EVALUATION CHECKLIST

Before applying AI output, check:

-  **Correctness:** Does it compile? Does it work?
-
-
-
-

REVIEW BEFORE PASTE: EVALUATION CHECKLIST

Before applying AI output, check:

-  **Correctness:** Does it compile? Does it work?
-  **Version:** Right Spring Boot/Java version?
-
-
-

REVIEW BEFORE PASTE: EVALUATION CHECKLIST

Before applying AI output, check:

-  **Correctness:** Does it compile? Does it work?
-  **Version:** Right Spring Boot/Java version?
-  **Style:** Matches team conventions?
-
-

REVIEW BEFORE PASTE: EVALUATION CHECKLIST

Before applying AI output, check:

-  **Correctness:** Does it compile? Does it work?
-  **Version:** Right Spring Boot/Java version?
-  **Style:** Matches team conventions?
-  **Security:** No obvious vulnerabilities?
-

REVIEW BEFORE PASTE: EVALUATION CHECKLIST

Before applying AI output, check:

-  **Correctness:** Does it compile? Does it work?
-  **Version:** Right Spring Boot/Java version?
-  **Style:** Matches team conventions?
-  **Security:** No obvious vulnerabilities?
-  **Tests:** Can I test this?

COMMON ISSUES TO SPOT

Wrong API versions:

-
-

COMMON ISSUES TO SPOT

Wrong API versions:

- Spring Boot 2 vs. 3 APIs
-

COMMON ISSUES TO SPOT

Wrong API versions:

- Spring Boot 2 vs. 3 APIs
- Java 8 vs. Java 17 features

Deprecated patterns:

-
-

Deprecated patterns:

- @Autowired vs. constructor injection
-

Deprecated patterns:

- @Autowired vs. constructor injection
- Old exception handling

Security issues:

-
-
-

Security issues:

- SQL injection risks
-
-

Security issues:

- SQL injection risks
- Missing validation
-

Security issues:

- SQL injection risks
- Missing validation
- XSS vulnerabilities

Performance problems:

-
-

Performance problems:

- N+1 queries
-

Performance problems:

- N+1 queries
- Inefficient algorithms

WHEN TO REJECT AI OUTPUT

Reject if:

-
-
-
-
-

Remember: AI assists, but you're responsible for the code.

WHEN TO REJECT AI OUTPUT

Reject if:

- **X** Wrong version or API
-
-
-
-

Remember: AI assists, but you're responsible for the code.

WHEN TO REJECT AI OUTPUT

Reject if:

- **X** Wrong version or API
- **X** Doesn't match team style
-
-
-

Remember: AI assists, but you're responsible for the code.

WHEN TO REJECT AI OUTPUT

Reject if:

- **X** Wrong version or API
- **X** Doesn't match team style
- **X** Security concerns
-
-

Remember: AI assists, but you're responsible for the code.

WHEN TO REJECT AI OUTPUT

Reject if:

- **X** Wrong version or API
- **X** Doesn't match team style
- **X** Security concerns
- **X** Too complex or unclear
-

Remember: AI assists, but you're responsible for the code.

WHEN TO REJECT AI OUTPUT

Reject if:

- **X** Wrong version or API
- **X** Doesn't match team style
- **X** Security concerns
- **X** Too complex or unclear
- **X** Better to write manually

Remember: AI assists, but you're responsible for the code.

SUMMARY

1.

2.

3.

4.

5.

SUMMARY

1. **Generation:** Boilerplate, entities, services — fast and consistent
- 2.
- 3.
- 4.
- 5.

SUMMARY

1. **Generation:** Boilerplate, entities, services — fast and consistent
2. **Explanation:** Legacy code, unfamiliar patterns — understand before changing
- 3.
- 4.
- 5.

SUMMARY

1. **Generation:** Boilerplate, entities, services — fast and consistent
2. **Explanation:** Legacy code, unfamiliar patterns — understand before changing
3. **Refactoring:** Modernize safely — always test before/after
- 4.
- 5.

SUMMARY

1. **Generation:** Boilerplate, entities, services — fast and consistent
2. **Explanation:** Legacy code, unfamiliar patterns — understand before changing
3. **Refactoring:** Modernize safely — always test before/after
4. **Testing:** Generate tests — run and refine iteratively
- 5.

SUMMARY

1. **Generation:** Boilerplate, entities, services — fast and consistent
2. **Explanation:** Legacy code, unfamiliar patterns — understand before changing
3. **Refactoring:** Modernize safely — always test before/after
4. **Testing:** Generate tests — run and refine iteratively
5. **Review:** Always evaluate before applying — correctness, version, style, security

BRIDGE TO MODULE 4

What we've learned:

-

What's next:

Module 4: Tooling Strategies — IDE integrations, chat tools, workflow optimization.

Apply generation, explanation, refactoring, and testing throughout.

BRIDGE TO MODULE 4

What we've learned:

- **What** to prompt for in Java development (generation, explanation, refactoring, testing)

What's next:

Module 4: Tooling Strategies — IDE integrations, chat tools, workflow optimization.

Apply generation, explanation, refactoring, and testing throughout.

QUESTIONS?

Module 3 — AI-Assisted Java Development

