lab-02-requirements-to-code

Contents

Lab 2: From Requirements to Code with GitHub Copilot

Duration: 45 minutes
Learning Objectives:

- Transform user stories into actionable backlog items using Copilot
- Generate acceptance criteria with AI assistance
- Create test-driven implementations from requirements
- Practice the full software development workflow with AI

Overview

In this lab, you'll experience the complete journey from a vague user requirement to working, tested code. You'll use GitHub Copilot to:

1. **Decompose** a user story into specific backlog items
2. **Generate** acceptance criteria for each item
3. **Create** test cases from acceptance criteria
4. **Implement** features using TDD principles from Lab 1

This simulates real-world Agile development where requirements are refined into actionable work items.

Prerequisites

- Completed Lab 1 (TDD with NotificationService)
- Understand Red-Green-Refactor cycle
- Familiar with Copilot Chat and slash commands
- Repository at clean state: `git status` shows no uncommitted changes

---

Part 1: User Story Analysis (15 minutes)

Scenario: Task Manager Enhancement

Your product owner provides this user story:

> **User Story**: As a user, I want to manage a list of tasks with priorities and due dates so I can track my work and focus on what's most important.

This is intentionally vague - typical of early requirements. Your job is to break it down into concrete, implementable features.

1.1 Generate Backlog Items

Open Copilot Chat (`Ctrl+Alt+I` / `Cmd+Shift+I`) and enter:

```
I have this user story: "As a user, I want to manage a list of tasks with priorities and due

Generate 5 backlog items with acceptance criteria for implementing this feature in our Task
```

1.2 Review Generated Backlog

Copilot should generate something like:

Backlog Item 1: Add Priority to Task Entity **Description**: Extend the Task domain entity to support priority levels (Low, Medium, High, Critical).

Acceptance Criteria:

- Task entity has a Priority property (enum)
- Priority is required when creating a task
- Priority can be updated after creation
- Priority is persisted to the database
- Invalid priority values are rejected

Technical Notes:

- Update Task entity in Domain layer
- Add Priority enum as value object
- Update database schema/migrations
- Respect DDD aggregate rules

Backlog Item 2: Add Due Date to Task Entity **Description**: Extend the Task domain entity to support due dates for deadline tracking.

Acceptance Criteria:

- Task entity has a DueDate property (nullable DateTime)
- Due date is optional when creating a task
- Due date can be set/updated/cleared after creation
- Due date must be in the future when set
- Due date is persisted to the database

Technical Notes:

- Update Task entity in Domain layer
- Add validation for future dates only
- Handle null case for tasks without deadlines
- Consider timezone handling

Backlog Item 3: Create Task with Priority and Due Date Description: Update task creation logic to accept priority and optional due date.

Acceptance Criteria:

- POST /tasks endpoint accepts priority and dueDate fields
- Priority is required in the request
- DueDate is optional in the request
- Invalid priority values return 400 Bad Request
- Past due dates return 400 Bad Request with clear error message
- Created task returns 201 with full task details including priority and due date

Technical Notes:

- Update CreateTaskCommand in Application layer
- Add validation logic
- Update API endpoint and request DTOs
- Add unit tests for all validation rules

Backlog Item 4: Filter Tasks by Priority Description: Allow users to retrieve tasks filtered by priority level.

Acceptance Criteria:

- GET /tasks?priority={level} endpoint supports filtering
- Valid priority values: Low, Medium, High, Critical
- Multiple priority values can be specified: ?priority=High&priority=Critical
- Invalid priority values return 400 Bad Request
- Returns empty array if no tasks match
- Results are ordered by due date (earliest first, nulls last)

Technical Notes:

- Add query parameters to endpoint
- Implement filtering in repository
- Add sorting by due date
- Unit test filtering logic
- Integration test endpoint behavior

Backlog Item 5: Get Tasks Due Soon **Description**: Provide an endpoint to retrieve tasks with approaching deadlines.

Acceptance Criteria:

- GET /tasks/due-soon endpoint returns tasks due in next 7 days
- Tasks without due dates are excluded
- Results ordered by due date (soonest first)
- Configurable time window (default 7 days, max 30 days)
- Returns 200 with array of tasks
- Query parameter: ?days={number} to customize window

Technical Notes:

- Add new endpoint in API layer
- Create query handler in Application layer
- Calculate date range based on current time
- Add unit tests for date calculations
- Consider timezone implications

1.3 Select Your Backlog Item

For this lab, we'll implement **Backlog Item 3: Create Task with Priority and Due Date**.

Why this item?

- Touches all layers (Domain, Application, Infrastructure, API)
- Demonstrates validation logic
- Requires TDD approach
- Foundation for other items (Items 1 & 2 are prerequisites)

Part 2: Implement Prerequisites (15 minutes)

TDD REMINDER: In this section, we'll follow Red-Green-Refactor:

1. **RED**: Write tests FIRST that fail
2. **GREEN**: Implement code to make tests pass
3. **REFACTOR**: Improve code quality while keeping tests green

Before we can create tasks with priority and due date, we need to add these properties to the Task entity (Items 1 & 2).

2.1 Add Priority Enum (Item 1)

Ask Copilot Chat:

Create a Priority enum as a value object in the Domain layer following DDD patterns. Include

Expected Output - src/TaskManager.Domain/ValueObjects/Priority.cs:

```
namespace TaskManager.Domain.ValueObjects;

public enum Priority
{
    Low = 0,
    Medium = 1,
    High = 2,
    Critical = 3
}
```

Or, for a more DDD approach with value object:

```
namespace TaskManager.Domain.ValueObjects;

public sealed record Priority
{
    public static readonly Priority Low = new(0, nameof(Low));
    public static readonly Priority Medium = new(1, nameof(Medium));
    public static readonly Priority High = new(2, nameof(High));
    public static readonly Priority Critical = new(3, nameof(Critical));

    public int Value { get; }
    public string Name { get; }

    private Priority(int value, string name)
    {
        Value = value;
        Name = name;
    }

    public static Priority FromValue(int value) => value switch
    {
        0 => Low,
        1 => Medium,
        2 => High,
        3 => Critical,
        _ => throw new ArgumentException($"Invalid priority value: {value}", nameof(value))
    };

    public static Priority FromName(string name) => name?.ToLowerInvariant() switch
    {
        "low" => Low,
        "medium" => Medium,
```

```
        "high" => High,
        "critical" => Critical,
        _ => throw new ArgumentException($"Invalid priority name: {name}", nameof(name))
    };
}
```

> **Note**: Our implementation uses three priority levels (Low, Medium, High) for simplicity. The Critical level is optional.

2.2 Write Tests for Task Entity (RED Phase)

Following TDD: Write tests FIRST before implementing!

Use the `/tests` command or ask Copilot Chat:

```
Generate xUnit tests for the Task entity in tests/TaskManager.UnitTests/Domain/Entities/Task
- Task.Create with valid title and priority succeeds
- Task.Create with valid title, priority, and future due date succeeds
- Task.Create with null/empty/whitespace title throws ArgumentException
- Task.Create with past due date throws ArgumentException
- Task.Create with null due date is allowed
- UpdatePriority updates the priority correctly
- UpdateDueDate with future date succeeds
- UpdateDueDate with past date throws ArgumentException
- MarkAsCompleted sets IsCompleted and CompletedAt

Use FakeItEasy for any dependencies if needed.
```

Run tests - they should **FAIL** because the Task entity doesn't exist yet or doesn't have these properties:

```
dotnet test
```

Expected result: Tests fail with compilation errors or NotImplementedException. This is the **RED** phase!

2.3 Implement Task Entity (GREEN Phase)

Now that we have failing tests, implement the code to make them pass.

Use `@workspace` to find the Task entity:

```
@workspace Where is the Task entity defined?
```

Then ask Copilot to update it:

```
Update the Task entity in #file:src/TaskManager.Domain/Entities/Task.cs to add:
1. Priority property (required)
2. DueDate property (nullable DateTime)
3. Validation: DueDate must be in future if provided
```

4. Factory method to create tasks with these properties
Follow DDD patterns: private constructor, factory method, invariant enforcement

Example Updated Entity:

```
namespace TaskManager.Domain.Entities;

public sealed class Task
{
    public Guid Id { get; private set; }
    public string Title { get; private set; }
    public string? Description { get; private set; }
    public Priority Priority { get; private set; }
    public DateTime? DueDate { get; private set; }
    public bool IsCompleted { get; private set; }
    public DateTime CreatedAt { get; private set; }
    public DateTime? CompletedAt { get; private set; }

    // Private constructor - use factory method
    private Task(
        Guid id,
        string title,
        string? description,
        Priority priority,
        DateTime? dueDate)
    {
        Id = id;
        Title = title;
        Description = description;
        Priority = priority;
        DueDate = dueDate;
        IsCompleted = false;
        CreatedAt = DateTime.UtcNow;
    }

    public static Task Create(
        string title,
        string? description,
        Priority priority,
        DateTime? dueDate = null)
    {
        if (string.IsNullOrWhiteSpace(title))
            throw new ArgumentException("Title cannot be null or empty", nameof(title));

        if (dueDate.HasValue && dueDate.Value <= DateTime.UtcNow)
            throw new ArgumentException("Due date must be in the future", nameof(dueDate));
```

```
        return new Task(Guid.NewGuid(), title, description, priority, dueDate);
    }

    public void UpdatePriority(Priority priority)
    {
        Priority = priority ?? throw new ArgumentNullException(nameof(priority));
    }

    public void UpdateDueDate(DateTime? dueDate)
    {
        if (dueDate.HasValue && dueDate.Value <= DateTime.UtcNow)
            throw new ArgumentException("Due date must be in the future", nameof(dueDate));

        DueDate = dueDate;
    }

    public void MarkAsCompleted()
    {
        if (IsCompleted)
            throw new InvalidOperationException("Task is already completed");

        IsCompleted = true;
        CompletedAt = DateTime.UtcNow;
    }
}
```

Run tests again:

```
dotnet test
```

Expected result: All tests pass! This is the **GREEN** phase!

2.4 Refactor (If Needed)

Review the code and tests:

- Are there any code smells?
- Can validation logic be extracted?
- Are error messages clear?
- Is the code following DDD patterns?

If you make changes, re-run tests to ensure they still pass:

```
dotnet test
```

Part 2 Complete! You've successfully added Priority and DueDate to the Task entity using proper TDD.

Part 3: Implement Backlog Item 3 (TDD) (15 minutes)

Now implement the full feature: Create Task with Priority and Due Date through the API.

3.1 Design the Application Layer

Step 1: Create the Command Ask Copilot Chat:

```
Create a CreateTaskCommand in the Application layer with properties:
- Title (required)
- Description (optional)
- Priority (required, string)
- DueDate (optional, DateTime?)

Include validation attributes and follow CQRS patterns.
```

Expected Output - `src/TaskManager.Application/Commands/CreateTaskCommand.cs`:

```
namespace TaskManager.Application.Commands;

public sealed record CreateTaskCommand
{
    public required string Title { get; init; }
    public string? Description { get; init; }
    public required string Priority { get; init; }
    public DateTime? DueDate { get; init; }
}
```

Step 2: Write Command Handler Tests (RED) Ask Copilot:

```
Create xUnit tests for CreateTaskCommandHandler in tests/TaskManager.UnitTests/Commands/Crea
- Valid command creates task with correct properties
- Invalid priority string throws exception
- Past due date throws exception
- Null title throws exception
Use FakeItEasy for ITaskRepository and ILogger
```

Run tests - they should **FAIL** (handler doesn't exist yet):

```
dotnet test
```

Step 3: Implement Command Handler (GREEN) Ask Copilot:

```
Implement CreateTaskCommandHandler in Application layer that:
1. Parses priority string to Priority value object
2. Validates due date is in future (if provided)
3. Creates Task entity using factory method
4. Saves via ITaskRepository
```

5. Returns created task
Follow Clean Architecture and use ILogger for structured logging

Expected Output - src/TaskManager.Application/Commands/CreateTaskCommandHandler.cs:

```
namespace TaskManager.Application.Commands;

public sealed class CreateTaskCommandHandler
{
    private readonly ITaskRepository _repository;
    private readonly ILogger<CreateTaskCommandHandler> _logger;

    public CreateTaskCommandHandler(
        ITaskRepository repository,
        ILogger<CreateTaskCommandHandler> logger)
    {
        _repository = repository ?? throw new ArgumentNullException(nameof(repository));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    public async Task<Domain.Entities.Task> HandleAsync(
        CreateTaskCommand command,
        CancellationToken cancellationToken = default)
    {
        if (command == null)
            throw new ArgumentNullException(nameof(command));

        _logger.LogInformation(
            "Creating task with title {Title} and priority {Priority}",
            command.Title,
            command.Priority);

        // Parse priority from string
        var priority = Priority.FromName(command.Priority);

        // Create task entity (validates due date)
        var task = Domain.Entities.Task.Create(
            command.Title,
            command.Description,
            priority,
            command.DueDate);

        // Save via repository
        await _repository.AddAsync(task, cancellationToken);

        _logger.LogInformation(
```

```
            "Task created successfully with ID {TaskId}",
            task.Id);

        return task;
    }
}
```

Run tests - they should **PASS**:

```
dotnet test
```

3.2 Update the API Layer (Following TDD)

Step 1: Create Request/Response DTOs Ask Copilot:

```
Create CreateTaskRequest and TaskResponse DTOs in API layer (src/TaskManager.Api/Models/) fo
```

Expected Output - Two DTO files that map between HTTP and Application layers. These are simple data structures, so no tests are needed.

Step 2: Write Integration Tests FIRST (RED Phase) Following TDD: Write integration tests BEFORE implementing the endpoint!

Use `@workspace` to find the endpoint extensions:

```
@workspace Where are the API endpoints defined?
```

Then create integration tests FIRST:

```
Create integration tests for POST /tasks endpoint in tests/TaskManager.IntegrationTests/Api/
- Valid request with all fields returns 201 Created with task details and Location header
- Valid request with only required fields returns 201 Created
- Invalid priority returns 400 Bad Request with ProblemDetails
- Past due date returns 400 Bad Request with ProblemDetails
- Missing/empty/whitespace title returns 400 Bad Request
- Optional fields (description, dueDate) handled correctly

Use WebApplicationFactory<Program> pattern and xUnit.
```

Run the integration tests - they should **FAIL** with 404 Not Found (endpoint doesn't exist yet):

```
dotnet test tests/TaskManager.IntegrationTests/
```

Expected result: All integration tests fail. This is the **RED** phase!

Step 3: Implement the Endpoint (GREEN Phase) Now implement the endpoint to make the tests pass:

```
Implement POST /tasks endpoint in #file:src/TaskManager.Api/Extensions/EndpointExtensions.cs
1. Maps CreateTaskRequest DTO to CreateTaskCommand
2. Calls CreateTaskCommandHandler to create the task
3. Maps the domain Task entity to TaskResponse DTO
4. Returns 201 Created with Location header and TaskResponse body
5. Handles ArgumentException (validation errors) → 400 Bad Request with ProblemDetails
6. Handles unexpected exceptions → 500 Internal Server Error with ProblemDetails

Use minimal API pattern, dependency injection for handler, and ILogger for logging.
```

Run the integration tests again:

```
dotnet test tests/TaskManager.IntegrationTests/
```

Expected result: All integration tests pass! This is the **GREEN** phase!

Step 4: Create Manual Testing File Create a `tasks.http` file in the API project for manual testing with the REST Client extension:

```
Create a tasks.http file in src/TaskManager.Api/ with test scenarios for POST /tasks endpoin
- Valid requests with all fields
- Valid requests with required fields only
- All priority levels (Low, Medium, High)
- Invalid priority
- Missing/empty/whitespace title
- Past due date
- Future due date
- Optional field combinations

Use REST Client format with @baseUrl variable set to http://localhost:5215
```

This file allows manual testing without writing curl commands repeatedly.

3.3 Run Full Test Suite

```
dotnet build
dotnet test
```

All tests should pass!

Expected output:

- Unit tests: All passing (14+ for CreateTaskCommandHandler, 11+ for Task entity)
- Integration tests: All passing (8 for TaskEndpointsTests)
- Build: 0 warnings, 0 errors

Part 4: Manual Testing & Validation (5 minutes)

4.1 Run the API

```
cd src/TaskManager.Api
dotnet run --launch-profile http
```

The API will start on `http://localhost:5215` (configured in `Properties/launchSettings.json`).

4.2 Test with REST Client Extension (Recommended)

If you created the `tasks.http` file in Step 3.2.4:

1. Install the **REST Client** extension in VS Code (by Huachao Mao)
2. Open `src/TaskManager.Api/tasks.http`
3. Click **"Send Request"** above any test scenario
4. View the response in a split pane

This is the easiest way to test your API!

4.3 Test with curl (Alternative)

Valid Request:

```
curl -X POST http://localhost:5215/tasks \
  -H "Content-Type: application/json" \
  -d '{
    "title": "Complete Lab 2",
    "description": "Finish requirements to code lab",
    "priority": "High",
    "dueDate": "2025-10-25T17:00:00Z"
  }'
```

Expected Response: 201 Created with Location header

```
{
  "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "title": "Complete Lab 2",
  "description": "Finish requirements to code lab",
  "priority": "High",
  "status": "Todo",
  "dueDate": "2025-10-25T17:00:00Z",
  "createdAt": "2025-10-20T14:30:00Z",
  "updatedAt": "2025-10-20T14:30:00Z"
}
```

Invalid Priority:

```
curl -X POST http://localhost:5215/tasks \
  -H "Content-Type: application/json" \
```

```
  -d '{
    "title": "Test Task",
    "priority": "SuperUrgent",
    "dueDate": "2025-10-25T17:00:00Z"
  }'
```

Expected Response: 400 Bad Request with ProblemDetails

```
{
  "type": "https://tools.ietf.org/html/rfc9110#section-15.5.1",
  "title": "Validation Error",
  "status": 400,
  "detail": "Invalid priority name: SuperUrgent (Parameter 'name')"
}
```

Past Due Date:

```
curl -X POST http://localhost:5215/tasks \
  -H "Content-Type: application/json" \
  -d '{
    "title": "Test Task",
    "priority": "Low",
    "dueDate": "2020-01-01T00:00:00Z"
  }'
```

Expected Response: 400 Bad Request with ProblemDetails

4.4 Verify Test Results

Confirm that:

- Valid requests return 201 Created with Location header
- Invalid priority returns 400 Bad Request with clear error message
- Past due dates return 400 Bad Request with validation error
- Missing/empty title returns 400 Bad Request
- Optional fields (description, dueDate) can be omitted
- Response includes all task properties (id, title, priority, status, timestamps)

Key Learning Points

AI-Assisted Requirements Analysis

1. **Decomposition**: Copilot helped break vague user story into concrete items
2. **Acceptance Criteria**: Generated testable, specific criteria for each item

3. **Technical Context**: Understood existing architecture and suggested appropriate patterns
4. **Comprehensive Coverage**: Identified edge cases and validation rules

Full-Stack TDD Workflow

1. **Red-Green-Refactor Applied**: Tests written FIRST at every layer
2. **Domain Layer TDD**: Task entity tests → implementation → refactor
3. **Application Layer TDD**: Handler tests → implementation → validation
4. **API Layer TDD**: Integration tests → endpoint implementation → manual testing
5. **Test Coverage**: Unit tests for logic, integration tests for full stack
6. **All Layers Tested**: Each layer validated independently with proper test pyramid

Clean Architecture Maintained

1. **Dependencies Flow Inward**: API → Application → Domain
2. **Domain Purity**: No infrastructure concerns in entities
3. **Application Logic**: Commands and handlers orchestrate use cases
4. **API Responsibility**: Only request/response mapping, no business logic

---

Extension Exercises (If Time Permits)

Exercise 1: Implement Item 4 (Filter by Priority)

1. Generate acceptance criteria tests
2. Implement repository filtering
3. Add API endpoint with query parameters
4. Test with multiple priority filters

Exercise 2: Implement Item 5 (Due Soon)

1. Write tests for date range calculations
2. Create query handler in Application layer
3. Add API endpoint
4. Test edge cases (timezone boundaries)

Exercise 3: Add Update Task Endpoint

1. Generate backlog item with acceptance criteria
2. Create UpdateTaskCommand
3. Implement PUT /tasks/{id} endpoint
4. Test validation and error cases

Success Criteria

You've completed this lab successfully when:

- User story decomposed into 5 backlog items with acceptance criteria
- Priority value object created in Domain layer
- Task entity updated with Priority and DueDate
- CreateTaskCommand and handler implemented with tests
- POST /tasks endpoint working with proper validation
- All tests passing (unit and integration)
- Manual testing confirms expected behavior
- Clean Architecture principles maintained throughout

Troubleshooting

Copilot Generates Generic Backlog Items

Problem: Backlog items don't consider existing architecture
Solution: Use `@workspace` to give context: "Given our Clean Architecture structure..."

Tests Don't Cover Edge Cases

Problem: Missing validation tests
Solution: Explicitly ask: "Generate tests for all guard clauses and edge cases"

Repository Pattern Not Working

Problem: ITaskRepository doesn't have needed methods
Solution: Update repository interface first, then implement in Infrastructure layer

Date Validation Issues

Problem: Due date validation fails unexpectedly
Solution: Use `DateTime.UtcNow` consistently, consider timezone handling

Next Steps

Move on to **Lab 3: Code Generation & Refactoring** where you'll:

- Scaffold complete API endpoints with Copilot
- Refactor legacy code using `/refactor` command
- Apply Object Calisthenics principles
- Use `@workspace` for cross-file understanding

Documenting Architectural Decisions (ADR)

Why ADRs?

As you make key design or architectural choices (e.g., how to model priorities, validation, or API structure), it's best practice to capture your reasoning in an Architecture Decision Record (ADR). This helps your team understand why decisions were made and makes future changes easier to justify.

Sample Copilot Prompt:

```
Write an Architecture Decision Record (ADR) for our approach to modeling task priorities as
- Context and alternatives considered
- Decision summary
- Consequences (tradeoffs, future impact)
Format as Markdown.
```

Where to put ADRs:

- Save ADRs in the `docs/adr/` folder (create it if it doesn't exist).
- Use a clear filename, e.g., `docs/adr/0001-task-priority-value-object.md`.

Learn more: ADR GitHub site

Additional Resources

- User Story Best Practices
- CQRS Pattern
- Clean Architecture
- DDD Value Objects