lab-03-generation-and-refactoring

Contents

Plan First with Agents: Safer, Smarter Refactoring 2

Lab 3: Code Generation & Refactoring with GitHub Copilot 3

Overview 3

Agent Mode Challenge: Go Beyond Ask/Edit 3

Prerequisites 4

Part 1: Generate API Endpoints (20 minutes) 4

Scenario: Complete CRUD Operations 4

1.1 Understand Existing Structure with @workspace 4

1.2 Generate Query: GET /tasks (List All) 4

1.3 Generate Query: GET /tasks/{id} (Get by ID) 6

1.4 Generate Command: PUT /tasks/{id} (Update) 6

1.5 Generate Command: DELETE /tasks/{id} 6

1.6 Run and Test 7

Part 2: Refactor Legacy Code (15 minutes) 7

Scenario: Legacy Task Processor 7

2.1 Find the Legacy Code 8

2.2 Analyze Current Issues 8

2.3 Refactor with /refactor Command 8

2.4 Generate Tests for Refactored Code 11

Part 3: Apply Object Calisthenics (10 minutes) 11

Scenario: Further Code Quality Improvements 11

3.1 Review Object Calisthenics Rules 11

3.2 Apply: Wrap Primitives 12

3.3 Apply: First-Class Collections 12

3.4 Apply: No Abbreviations 13

Part 4: Multi-File Refactoring with Copilot Edits (Optional, if time) . 14

Scenario: Rename Across Multiple Files 14

4.1 Open Copilot Edits 14

4.2 Add Files to Working Set 14

4.3 Describe Change 14

4.4 Review Proposed Changes 14

4.5 Accept or Reject 14

Key Learning Points 15
Context-Aware Code Generation 15
Effective Refactoring Workflow 15
Code Quality Improvements 15
Multi-File Editing 15
Extension Exercises (If Time Permits) 15
Exercise 1: Add Pagination 15
Exercise 2: Add Sorting 16
Exercise 3: Extract API Response Builder 16
Success Criteria 16
Troubleshooting 16
Copilot Generates Inconsistent Patterns 16
Refactoring Breaks Tests 16
Too Many Changes at Once 16
Multi-File Edit Misses Files 17
Next Steps 17
Additional Resources 17

Plan First with Agents: Safer, Smarter Refactoring

Before making major changes, try using Copilot (in Agent Mode) to generate a plan first. This helps you:

- Understand the scope and impact of your changes
- Catch misunderstandings or missing steps early
- Collaborate and iterate on the approach before any code is changed

How to try it:

- In Copilot Chat (Agent Mode), ask: "Propose a step-by-step plan to refactor LegacyTaskProcessor to use async/await, add logging, and follow Object Calisthenics."
- Review the plan. Edit or reorder steps as needed.
- Only then, ask Copilot (or a custom agent like `@engineer`) to implement the plan, one step at a time or all at once.

Custom Agents Demo:

- Use `@planner` to generate/refine the plan
- Use `@engineer` to execute the approved plan

Reflection:

- Did planning first catch any issues you would have missed?
- Was the implementation smoother or more predictable?

Facilitator Tip: Model this workflow live, and encourage participants to always ask for a plan before executing large or multi-file changes.

Lab 3: Code Generation & Refactoring with GitHub Copilot

Duration: 45 minutes
Learning Objectives:

- Generate complete API endpoints using Copilot and context variables
- Refactor legacy code using `/refactor` command and Inline Chat
- Apply Object Calisthenics principles with AI assistance
- Use `@workspace` for understanding and modifying existing code
- Leverage Copilot Edits for multi-file refactoring

Overview

In this lab, you'll work with both new and existing code:

- **Part 1**: Generate new API endpoints efficiently using Copilot's context awareness
- **Part 2**: Refactor legacy code (`LegacyTaskProcessor`) to modern standards
- **Part 3**: Apply advanced refactoring patterns (Object Calisthenics)

Agent Mode Challenge: Go Beyond Ask/Edit

For this lab, try using **Agent Mode** for at least one major task (such as refactoring `LegacyTaskProcessor` or generating all CRUD endpoints at once). Agent Mode lets Copilot plan and execute multi-step, multi-file changes, and can invoke advanced tools (like MCP evaluation or tracing) automatically.

How to try it:

- Switch Copilot Chat to "Agent" mode (dropdown in chat panel)
- Describe your goal in natural language (e.g., "Refactor LegacyTaskProcessor to use async/await, add logging, and follow Object Calisthenics")
- Review the plan and results, iterate as needed
- For advanced users: reference MCP tools directly (e.g., "Evaluate my API endpoints using aitk-evaluation_planner")

Compare:

- What did Agent Mode do differently than Ask/Edit?
- Did it propose a plan, use multiple tools, or make changes across files?
- Was the result more complete or did it need more review?

Facilitator Tip: Encourage participants to share their Agent Mode results and discuss when this approach is most effective.

Prerequisites

- Completed Lab 1 (TDD) and Lab 2 (Requirements to Code)
- Familiar with Copilot Chat, Inline Chat, and slash commands
- Understanding of Clean Architecture layers
- Repository at clean state

Part 1: Generate API Endpoints (20 minutes)

Scenario: Complete CRUD Operations

You have the POST /tasks endpoint from Lab 2. Now complete the REST API with GET, PUT, and DELETE operations.

1.1 Understand Existing Structure with @workspace

Before generating new code, understand what exists:

```
@workspace Show me the API endpoint structure. Where are endpoints defined and how are they
```

Copilot should identify:

- `src/TaskManager.Api/Extensions/EndpointExtensions.cs` - Endpoint definitions
- Minimal API pattern with extension methods
- Existing POST /tasks endpoint
- DI container registration in `Program.cs`

1.2 Generate Query: GET /tasks (List All)

Step 1: Design Query Handler Ask Copilot Chat:

```
Create a GetTasksQuery handler in the Application layer following CQRS pattern.
It should:
- Return all tasks from ITaskRepository
- Support optional filtering by TaskStatus (enum: Todo, InProgress, Done)
- Order results by CreatedAt descending
Include unit tests using xUnit and FakeItEasy
```

Expected Output:

- `src/TaskManager.Application/Queries/GetTasksQuery.cs`
- `src/TaskManager.Application/Queries/GetTasksQueryHandler.cs`
- `tests/TaskManager.UnitTests/Application/Queries/GetTasksQueryHandlerTests.cs`

Note: The domain model uses `TaskStatus` enum (Todo/InProgress/Done) rather than a boolean `IsCompleted` field.

Step 2: Implement Endpoint Use `#file` context variable:

```
Add a GET /tasks endpoint in #file:src/TaskManager.Api/Extensions/EndpointExtensions.cs that
- Accepts optional query parameter: status (string: "Todo", "InProgress", or "Done")
- Calls GetTasksQueryHandler
- Returns 200 OK with array of TaskResponse
- Uses async/await and proper error handling
Follow the existing endpoint pattern
```

Expected Addition:

```
public static void MapTaskEndpoints(this IEndpointRouteBuilder app)
{
    // ... existing POST /tasks endpoint ...

    // GET /tasks
    app.MapGet("/tasks", async (
        [FromQuery] string? status,
        GetTasksQueryHandler handler,
        CancellationToken cancellationToken) =>
    {
        try
        {
            // Parse status string to TaskStatus enum if provided
            TaskStatus? taskStatus = null;
            if (!string.IsNullOrEmpty(status) &&
                Enum.TryParse<TaskStatus>(status, true, out var parsed))
            {
                taskStatus = parsed;
            }

            var query = new GetTasksQuery { Status = taskStatus };
            var tasks = await handler.HandleAsync(query, cancellationToken);
            var response = tasks.Select(t => new TaskResponse
            {
                Id = t.Id.Value,
                Title = t.Title,
                Description = t.Description,
                Priority = t.Priority.ToString(),
                Status = t.Status.ToString(),
                DueDate = t.DueDate,
                CreatedAt = t.CreatedAt
            });

            return Results.Ok(response);
        }
        catch (Exception ex)
```

```
        {
            return Results.Problem(
                detail: ex.Message,
                statusCode: 500);
        }
    })
    .WithName("GetTasks")
    .WithTags("Tasks")
    .Produces<IEnumerable<TaskResponse>>(200)
    .Produces<ProblemDetails>(500);
}
```

1.3 Generate Query: GET /tasks/{id} (Get by ID)

Ask Copilot:

```
Create a GetTaskByIdQuery handler in Application layer that:
- Accepts a Guid taskId
- Returns single task from repository or null
- Throws ArgumentException if taskId is empty
Include unit tests with FakeItEasy

Then add GET /tasks/{id} endpoint that returns 200 OK or 404 Not Found
```

Key Learning: Notice how Copilot reuses patterns from existing code (error handling, response mapping, validation).

1.4 Generate Command: PUT /tasks/{id} (Update)

Use Inline Chat (`Ctrl+I` / `Cmd+I`):

1. Open `EndpointExtensions.cs`
2. Position cursor after the GET endpoints
3. Press `Ctrl+I` / `Cmd+I`
4. Enter:

```
Add PUT /tasks/{id} endpoint that:
- Accepts UpdateTaskRequest (title, description, priority, dueDate)
- Creates UpdateTaskCommand
- Calls UpdateTaskCommandHandler
- Returns 200 OK with updated task or 404 if not found
Include command handler in Application layer with tests
```

1.5 Generate Command: DELETE /tasks/{id}

Ask Copilot Chat:

```
Create DeleteTaskCommand and handler that:
- Accepts taskId
- Removes task from repository
- Returns success (void)
- Throws if task not found

Add DELETE /tasks/{id} endpoint returning 204 No Content or 404 Not Found
Include unit tests for handler
```

1.6 Run and Test

```
dotnet build
dotnet test
cd src/TaskManager.Api
dotnet run
```

Test the full API:

```
# Create a task
curl -X POST http://localhost:5000/tasks \
  -H "Content-Type: application/json" \
  -d '{"title": "Test Task", "priority": "Medium", "dueDate": "2025-10-30T12:00:00Z"}'

# List all tasks
curl http://localhost:5000/tasks

# Get specific task (use ID from create response)
curl http://localhost:5000/tasks/{id}

# Update task
curl -X PUT http://localhost:5000/tasks/{id} \
  -H "Content-Type: application/json" \
  -d '{"title": "Updated Task", "priority": "High", "dueDate": "2025-11-01T12:00:00Z"}'

# Delete task
curl -X DELETE http://localhost:5000/tasks/{id}
```

Part 2: Refactor Legacy Code (15 minutes)

Scenario: Legacy Task Processor

The repository contains `LegacyTaskProcessor.ProcessTask` - poorly written code that needs refactoring.

2.1 Find the Legacy Code

Use `@workspace`:

```
@workspace Find the LegacyTaskProcessor class
```

Location: `src/TaskManager.Infrastructure/Legacy/LegacyTaskProcessor.cs`

2.2 Analyze Current Issues

Use `/explain` on the problematic method:

1. Navigate to the `ProcessTask` method (not `ProcessTaskBatch` - that's a typo in earlier drafts)
2. Select the entire method
3. Use Inline Chat (`Ctrl+I` or `Cmd+I`): `/explain`

Copilot should identify issues:

- Nested if statements (6+ indentation levels)
- Synchronous blocking code (`Thread.Sleep`)
- Poor error handling (exceptions swallowed with empty catch)
- No logging
- Magic numbers (1, 2, 50) and strings
- Long method (80+ lines with multiple responsibilities)
- Poor naming (`data`, `flag`, `type`, `i`)
- String concatenation in loops (inefficient)
- Mixed concerns (file I/O in processing logic)
- Not following guard clause pattern

2.3 Refactor with /refactor Command

Select the entire `ProcessTask` method and use Copilot Chat:

```
/refactor this method to follow Clean Code principles:
1. Use guard clauses (fail fast, no nested ifs)
2. Convert to async/await
3. Add structured logging with ILogger<LegacyTaskProcessor>
4. Extract smaller methods for single responsibilities
5. Use proper exception handling (don't swallow exceptions)
6. Replace magic numbers with constants or enums
7. Use meaningful parameter and variable names
8. Use StringBuilder for string operations in loops
9. Separate concerns: extract file I/O to an interface (ITaskOutputWriter)
10. Follow Object Calisthenics: max 2 levels of indentation per method
Follow .github/copilot-instructions.md conventions and make the class sealed
```

Expected Improvements:

- Strongly-typed `ProcessingType` enum instead of `int type`

- Guard clauses for null/empty input (fail fast)
- Private helper methods: `ProcessFormatting()`, `ProcessCapitalization()`, `TruncateIfNeeded()`
- Async signature: `Task<string> ProcessTaskAsync(...)`
- Constructor injection: `ILogger<LegacyTaskProcessor>`, `ITaskOutputWriter?`
- Proper error handling with logging
- `StringBuilder` for efficient string building
- Meaningful names: `taskIdentifier`, `inputText`, `processingType`, `shouldInvertCase`

Expected Refactored Code:

```
public async Task<ProcessingResult> ProcessTaskBatchAsync(
    IEnumerable<TaskItem> tasks,
    CancellationToken cancellationToken = default)
{
    if (tasks == null)
        throw new ArgumentNullException(nameof(tasks));

    _logger.LogInformation("Starting batch processing of tasks");

    var taskList = tasks.ToList();
    if (taskList.Count == 0)
    {
        _logger.LogInformation("No tasks to process");
        return ProcessingResult.Empty;
    }

    var result = new ProcessingResult();

    foreach (var task in taskList)
    {
        await ProcessSingleTaskAsync(task, result, cancellationToken);
    }

    _logger.LogInformation(
        "Batch processing completed: {SuccessCount} succeeded, {FailureCount} failed",
        result.SuccessCount,
        result.FailureCount);

    return result;
}

private async Task ProcessSingleTaskAsync(
    TaskItem task,
    ProcessingResult result,
```

```
    CancellationToken cancellationToken)
{
    if (!IsTaskValid(task))
    {
        _logger.LogWarning("Invalid task {TaskId} skipped", task.Id);
        result.AddFailure(task.Id, "Invalid task data");
        return;
    }

    try
    {
        await ExecuteTaskProcessingAsync(task, cancellationToken);
        result.AddSuccess(task.Id);

        _logger.LogInformation(
            "Task {TaskId} processed successfully",
            task.Id);
    }
    catch (Exception ex)
    {
        _logger.LogError(
            ex,
            "Failed to process task {TaskId}",
            task.Id);
        result.AddFailure(task.Id, ex.Message);
    }
}

private static bool IsTaskValid(TaskItem task)
{
    if (task == null) return false;
    if (string.IsNullOrWhiteSpace(task.Title)) return false;
    if (task.Priority < 0 || task.Priority > 3) return false;

    return true;
}

private async Task ExecuteTaskProcessingAsync(
    TaskItem task,
    CancellationToken cancellationToken)
{
    // Update task status
    task.Status = TaskStatus.Processing;
    await _repository.UpdateAsync(task, cancellationToken);
```

```
    // Simulate processing
    await Task.Delay(100, cancellationToken);

    // Complete task
    task.Status = TaskStatus.Completed;
    task.CompletedAt = DateTime.UtcNow;
    await _repository.UpdateAsync(task, cancellationToken);
}
```

2.4 Generate Tests for Refactored Code

Select the refactored method and use `/tests`:

```
/tests
```

Verify generated tests cover:

- Null input throws ArgumentNullException
- Empty collection returns empty result
- Valid tasks are processed successfully
- Invalid tasks are logged and skipped
- Processing exceptions are caught and logged
- Result contains correct success/failure counts

Run tests:

```
dotnet test
```

Part 3: Apply Object Calisthenics (10 minutes)

Scenario: Further Code Quality Improvements

Apply Object Calisthenics rules from `.github/copilot-instructions.md` Section 7.

3.1 Review Object Calisthenics Rules

Ask Copilot:

```
What are the Object Calisthenics rules from #file:.github/copilot-instructions.md?
```

Key rules:

1. Only one level of indentation per method
2. Don't use 'else' keyword (guard clauses)
3. Wrap all primitives and strings
4. First-class collections
5. One dot per line (avoid call chains)

6. Don't abbreviate names
7. Keep all entities small
8. No classes with more than two instance variables
9. No getters/setters/properties (for domain entities)

3.2 Apply: Wrap Primitives

Find places where primitive types are used directly for domain concepts.

Ask Copilot:

```
Review the TaskItem class. Are there primitive types that should be wrapped in value objects
```

Before:

```
public class TaskItem
{
    public Guid Id { get; set; }
    public string Status { get; set; } // Primitive obsession
    public int Priority { get; set; } // Magic numbers
}
```

After (with Copilot assistance):

```
public sealed class TaskItem
{
    public TaskId Id { get; private set; }
    public TaskStatus Status { get; private set; }
    public Priority Priority { get; private set; }
}
```

3.3 Apply: First-Class Collections

Find collections that are exposed directly and wrap them.

Ask Copilot:

```
If we have a class with a List<Task> property, how should we wrap it following Object Calist
```

Before:

```
public class TaskList
{
    public List<TaskItem> Tasks { get; set; }
}
```

After:

```
public sealed class TaskCollection
{
    private readonly List<TaskItem> _tasks;
```

```
    public TaskCollection(IEnumerable<TaskItem> tasks)
    {
        _tasks = tasks?.ToList() ?? new List<TaskItem>();
    }

    public int Count => _tasks.Count;

    public IReadOnlyList<TaskItem> Items => _tasks.AsReadOnly();

    public void Add(TaskItem task)
    {
        if (task == null)
            throw new ArgumentNullException(nameof(task));

        _tasks.Add(task);
    }

    public TaskItem? FindById(TaskId id) =>
        _tasks.FirstOrDefault(t => t.Id == id);
}
```

3.4 Apply: No Abbreviations

Use Inline Chat to expand abbreviated names:

1. Find abbreviated variable names (e.g., `var t`, `var res`, `int cnt`)
2. Select the code
3. Inline Chat: "Expand all abbreviated variable names to be fully descriptive"

Before:

```
var res = await _repo.GetAsync(id);
if (res != null)
{
    var cnt = res.Items.Count();
    // ...
}
```

After:

```
var result = await _repository.GetAsync(id);
if (result != null)
{
    var itemCount = result.Items.Count();
    // ...
}
```

Part 4: Multi-File Refactoring with Copilot Edits (Optional, if time)

Scenario: Rename Across Multiple Files

Use Copilot Edits for cross-cutting changes.

4.1 Open Copilot Edits

1. Open Command Palette (`Ctrl+Shift+P` / `Cmd+Shift+P`)
2. Search for "Copilot Edits: Open"
3. Or use dedicated Copilot Edits panel in sidebar

4.2 Add Files to Working Set

Add related files:

- `src/TaskManager.Domain/Entities/Task.cs`
- `src/TaskManager.Application/Commands/CreateTaskCommand.cs`
- `src/TaskManager.Application/Commands/CreateTaskCommandHandler.cs`
- `tests/TaskManager.UnitTests/Commands/CreateTaskCommandHandlerTests.cs`

4.3 Describe Change

In the Copilot Edits panel:

```
Rename the "Title" property to "Name" across all files in the working set. Update:
- Entity property
- Command property
- All references in handlers
- All test assertions
Ensure consistency across the entire codebase
```

4.4 Review Proposed Changes

Copilot will show:

- All files that will be modified
- Exact changes in each file
- Side-by-side diff view

4.5 Accept or Reject

- Review each change carefully
- Accept all if changes look correct
- Or accept/reject individual file changes
- Run tests after applying: `dotnet test`

Key Learning Points

Context-Aware Code Generation

1. **@workspace**: Understanding existing structure before generating
2. **#file**: Referencing specific files for consistent patterns
3. **#selection**: Refactoring specific code sections
4. **Pattern Reuse**: Copilot learned patterns from existing endpoints

Effective Refactoring Workflow

1. **/explain**: Understand code before changing it
2. **/refactor**: Automated refactoring with specific goals
3. **/tests**: Generate tests for refactored code
4. **Iterative**: Refactor in small steps, run tests frequently

Code Quality Improvements

1. **Guard Clauses**: Early returns reduce indentation
2. **Async/Await**: Modern patterns for I/O operations
3. **Logging**: Structured logging provides observability
4. **Single Responsibility**: Extracted methods with clear purposes
5. **Object Calisthenics**: Advanced quality constraints

Multi-File Editing

1. **Copilot Edits**: Consistent changes across multiple files
2. **Working Set**: Explicitly define scope of changes
3. **Review Process**: Always review AI-proposed changes
4. **Safe Refactoring**: Tests validate behavior preservation

---

Extension Exercises (If Time Permits)

Exercise 1: Add Pagination

Refactor GET /tasks to support pagination (page, pageSize query parameters).
Use Copilot to:

1. Add pagination to repository
2. Update query handler
3. Modify endpoint
4. Update tests

Exercise 2: Add Sorting

Add sorting support to GET /tasks (sortBy, sortOrder parameters). Valid sort fields: title, priority, dueDate, createdAt

Exercise 3: Extract API Response Builder

Create a dedicated class for building TaskResponse from Task entity. Use Copilot Edits to update all endpoints to use the builder.

Success Criteria

You've completed this lab successfully when:

- Full CRUD API endpoints implemented (POST, GET, GET by ID, PUT, DELETE)
- All endpoints follow consistent patterns
- LegacyTaskProcessor refactored to modern standards
- Refactored code follows Object Calisthenics principles
- Guard clauses used instead of nested ifs
- Async/await pattern applied throughout
- Structured logging added
- All tests passing
- Code is clean, readable, and maintainable

Troubleshooting

Copilot Generates Inconsistent Patterns

Problem: New endpoints don't match existing style
Solution: Use `#file` to reference existing endpoint file, explicitly state "Follow the existing pattern"

Refactoring Breaks Tests

Problem: Tests fail after refactoring
Solution: This is OK! Update tests to match new behavior. Use `/tests` to regenerate tests.

Too Many Changes at Once

Problem: Copilot suggests massive refactoring
Solution: Break into smaller steps. Refactor one method at a time. Run tests after each change.

Multi-File Edit Misses Files

Problem: Copilot Edits doesn't update all references
Solution: Use VS Code's built-in "Rename Symbol" (F2) for simple renames. Use Copilot Edits for semantic changes.

Next Steps

Move on to **Lab 4: Testing, Documentation & Workflow** where you'll:

- Generate comprehensive test suites with `/tests`
- Create documentation with `/doc`
- Write Conventional Commit messages
- Draft PR descriptions with `@workspace`

Additional Resources

- Object Calisthenics
- Refactoring Techniques
- Clean Code Principles
- Minimal APIs in .NET