test-strategist

Agent Test Scenario: Test Strategist

This document contains test scenarios for validating the Test Strategist agent.

Scenario 1: Unit Test Strategy for Domain Aggregate

Input

Prompt:

```
Propose a test strategy for the Task aggregate. Focus on unit tests for
business logic and invariant enforcement.
```

Context:

- TaskManager.Domain/Tasks/Task.cs open

Sample Domain Code:

```
public sealed class Task
{
    public TaskId Id { get; private set; }
    private string _title;
    private string _description;
    private TaskStatus _status;
    private DateTime? _dueDate;

    private Task(TaskId id, string title)
    {
        Id = id;
        _title = title;
        _status = TaskStatus.NotStarted;
    }

    public static Result<Task> Create(TaskId id, string title, string
          description)
    {
        if (string.IsNullOrWhiteSpace(title))
            return Result.Failure<Task>("Title is required");
        if (title.Length > 200)
            return Result.Failure<Task>("Title must be 200 characters or
          less");

        var task = new Task(id, title);
        task._description = description;
        return Result.Success(task);
    }

    public Result Complete()
    {
        if (_status == TaskStatus.Completed)
            return Result.Failure("Task is already completed");

        _status = TaskStatus.Completed;
        return Result.Success();
    }

    public Result SetDueDate(DateTime dueDate)
    {
        if (dueDate < DateTime.UtcNow)
            return Result.Failure("Due date must be in the future");

        _dueDate = dueDate;
        return Result.Success();
    }
}
```

Expected Output

Format:

```
# Test Strategy: Task Aggregate

## Component: TaskManager.Domain.Tasks.Task

## Test Pyramid Distribution
- Unit Tests: 90%
- Integration Tests: 10%
- E2E Tests: 0%

## Unit Tests

### Task.Create()
#### Scenario 1: Create with valid data
- **Type:** Happy path
- **Given:** Valid TaskId, title, and description
- **When:** Task.Create() is called
- **Then:** Returns Success with Task instance

[Additional scenarios...]

## Test Data & Fixtures
[Suggested test data]

## Recommendations
[Prioritized testing suggestions]
```

Content Expectations:

Unit Test Scenarios:

Create Method:

1. ✅ Happy path: Valid data returns Success
2. ❌ Error case: Null title returns Failure
3. ❌ Error case: Empty title returns Failure
4. ❌ Error case: Title > 200 chars returns Failure
5. ✅ Edge case: Title exactly 200 chars succeeds
6. ✅ Edge case: Null description is allowed

Complete Method:

1. ✅ Happy path: NotStarted → Completed succeeds
2. ✅ Happy path: InProgress → Completed succeeds
3. ❌ Error case: Already Completed returns Failure

SetDueDate Method:

1. ✅ Happy path: Future date succeeds

2. ❌ Error case: Past date returns Failure
3. ⚠️ Edge case: Date exactly at DateTime.UtcNow (boundary)
4. ✅ Edge case: Far future date succeeds

Test Data Suggestions:

- Valid TaskId: `new TaskId(Guid.NewGuid())`
- Valid title: "Implement user authentication"
- Long title: 200-char string for boundary testing
- Invalid title: empty string, whitespace, 201-char string
- Future date: `DateTime.UtcNow.AddDays(7)`
- Past date: `DateTime.UtcNow.AddDays(-1)`

Recommendations:

1. Use xUnit theory with InlineData for boundary cases
2. Create TaskBuilder test fixture for valid defaults
3. Test each invariant violation separately
4. Consider property-based testing for title length validation

Success Criteria

- [ ] Covers all public methods
- [ ] Happy path identified for each method
- [ ] Error cases for each invariant
- [ ] Edge cases at boundaries
- [ ] Test data suggestions provided
- [ ] Mocking strategy (none needed for pure domain)
- [ ] Test organization suggested (e.g., class per method)

Scenario 2: Integration Test Strategy for Repository

Input

Prompt:

```
What integration tests do we need for TaskRepository? We're using
Entity Framework Core with SQL Server.
```

Context:

- TaskManager.Infrastructure/Repositories/TaskRepository.cs open
- TaskManager.Domain/Repositories/ITaskRepository.cs open

Sample Repository:

```
public interface ITaskRepository
{
    Task<TaskEntity?> FindById(TaskId id);
    Task Add(TaskEntity task);
    Task<IEnumerable<TaskEntity>> FindOverdueTasks();
    Task<IEnumerable<TaskEntity>> FindByAssignee(UserId userId);
}
```

Expected Output

Content Expectations:

Test Pyramid:

- Unit Tests: 10% (repository interface tests with fakes)
- Integration Tests: 90% (database interaction)
- E2E Tests: 0%

Integration Test Scenarios:

FindById:

1. ✅ Happy path: Existing task is found
2. ❌ Not found: Non-existent ID returns null
3. ✅ Edge case: Task with all optional fields populated
4. ✅ Edge case: Task with minimal data

Add:

1. ✅ Happy path: Task is persisted to database
2. ✅ Edge case: Task with complex data (collections, value objects)
3. ⚠️ Error case: Duplicate ID throws/handled appropriately

FindOverdueTasks:

1. ✅ Happy path: Returns tasks where DueDate < Now
2. ✅ Edge case: Empty result when no overdue tasks
3. ✅ Edge case: Multiple overdue tasks ordered correctly
4. ⚠️ Boundary: Task due exactly now

FindByAssignee:

1. ✅ Happy path: Returns user's assigned tasks
2. ✅ Edge case: User with no tasks returns empty
3. ✅ Edge case: User with many tasks (pagination if applicable)

Test Infrastructure:

- **Database:** Use Testcontainers with SQL Server
- **Setup:** Arrange fresh database per test
- **Cleanup:** Database disposed after each test
- **Data:** Use TaskBuilder fixtures

Example Test Structure:

```
public class TaskRepositoryTests : IAsyncLifetime
{
    private readonly SqlServerContainer _container;
    private TaskDbContext _context;
    private TaskRepository _repository;

    // Testcontainers setup
    public async Task InitializeAsync() { ... }
    public async Task DisposeAsync() { ... }

    [Fact]
    public async Task FindById_ExistingTask_ReturnsTask() { ... }
}
```

Recommendations:

1. Use Testcontainers for real SQL Server instance
2. Test each query method with real database
3. Verify EF mappings (value objects, owned entities)
4. Test transaction behavior if using Unit of Work
5. Performance test with larger datasets (100+ tasks)

Success Criteria

- [ ] Recommends Testcontainers or similar
- [ ] Covers all repository methods
- [ ] Addresses database-specific concerns (EF mappings)
- [ ] Suggests test data strategy
- [ ] Mentions performance testing
- [ ] Test setup/teardown approach described

Scenario 3: E2E Test Strategy for API

Input

Prompt:

```
What end-to-end tests should we have for the Task API endpoints?
We have:
- POST /tasks (create task)
- GET /tasks/{id} (get task)
```

```
- PUT /tasks/{id}/complete (mark complete)
- GET /tasks (list tasks with filtering)
```

Context:

- TaskManager.Api/Program.cs open (Minimal API)

Expected Output

Content Expectations:

Test Pyramid:

- Unit Tests: 60% (domain + application logic)
- Integration Tests: 30% (repository + infrastructure)
- E2E Tests: 10% (critical user journeys)

E2E Test Scenarios:

Critical User Journeys:

Journey 1: Create and Complete Task

1. POST /tasks with valid data → 201 Created
2. GET /tasks/{id} → 200 OK, task exists
3. PUT /tasks/{id}/complete → 200 OK
4. GET /tasks/{id} → 200 OK, status = Completed

Journey 2: Validation Errors

1. POST /tasks with missing title → 400 Bad Request
2. POST /tasks with title too long → 400 Bad Request
3. PUT /invalid-id/complete → 404 Not Found
4. PUT /tasks/{completed-task-id}/complete → 409 Conflict

Journey 3: Filtering and Querying

1. POST multiple tasks with different statuses
2. GET /tasks?status=NotStarted → Returns only NotStarted
3. GET /tasks?assignee={userId} → Returns user's tasks
4. GET /tasks?overdue=true → Returns overdue tasks

Test Framework:

- **Tool:** WebApplicationFactory with xUnit
- **Database:** Testcontainers SQL Server
- **Auth:** Mock auth if needed or test auth flow
- **Client:** HttpClient from WebApplicationFactory

Example Test:

```
public class TaskApiTests : IClassFixture<WebApplicationFactory<Program>>
{
    private readonly HttpClient _client;
```

```
    [Fact]
    public async Task CreateAndCompleteTask_FullJourney_Succeeds()
    {
        // Arrange
        var createRequest = new { title = "Test Task", description = "Test" };

        // Act - Create
        var createResponse = await _client.PostAsJsonAsync("/tasks",
          createRequest);
        createResponse.StatusCode.Should().Be(HttpStatusCode.Created);
        var taskId = await GetTaskIdFromResponse(createResponse);

        // Act - Complete
        var completeResponse = await _client.PutAsync($"/tasks/{taskId}/
          complete", null);
        completeResponse.StatusCode.Should().Be(HttpStatusCode.OK);

        // Assert
        var getResponse = await _client.GetAsync($"/tasks/{taskId}");
        var task = await getResponse.Content.ReadFromJsonAsync<TaskDto>();
        task.Status.Should().Be("Completed");
    }
}
```

Recommendations:

1. Focus E2E on critical happy paths and error cases
2. Use WebApplicationFactory for in-memory testing
3. Test authentication/authorization if implemented
4. Test content negotiation (JSON, problem details)
5. Performance test under load if API is high-traffic

Success Criteria

- [ ] Identifies critical user journeys
- [ ] Balances happy path and error scenarios
- [ ] Recommends appropriate E2E framework
- [ ] Suggests realistic test data
- [ ] Mentions auth/authz testing if applicable
- [ ] Focuses on integration, not exhaustive coverage

Scenario 4: Test Coverage Analysis

Input

Prompt:

```
Review our current test coverage for TaskService and identify gaps.
```

Context:

- TaskManager.Application/Services/TaskService.cs open
- tests/TaskManager.UnitTests/Services/TaskServiceTests.cs open

Current Tests:

```
public class TaskServiceTests
{
    [Fact]
    public async Task CreateTask_ValidData_CreatesTask()
    {
        // Arrange
        var repository = A.Fake<ITaskRepository>();
        var service = new TaskService(repository);

        // Act
        var result = await service.CreateTask("Title", "Description", null);

        // Assert
        result.IsSuccess.Should().BeTrue();
        A.CallTo(() => repository.Add(A<TaskEntity>._)).MustHaveHappened();
    }

    // Only one test exists!
}
```

TaskService Code:

```
public class TaskService
{
    private readonly ITaskRepository _repository;
    private readonly INotificationService _notificationService;

    public async Task<Result<TaskId>> CreateTask(string title, string
          description, UserId? assignee)
    {
        var taskId = new TaskId(Guid.NewGuid());
        var taskResult = Task.Create(taskId, title, description);

        if (taskResult.IsFailure)
            return Result.Failure<TaskId>(taskResult.Error);
```

```
        await _repository.Add(taskResult.Value);

        if (assignee.HasValue)
            await _notificationService.NotifyTaskAssigned(taskResult.Value,
          assignee.Value);

        return Result.Success(taskId);
    }

    public async Task<Result> CompleteTask(TaskId taskId)
    {
        var task = await _repository.FindById(taskId);
        if (task == null)
            return Result.Failure("Task not found");

        var completeResult = task.Complete();
        if (completeResult.IsFailure)
            return completeResult;

        await _repository.Update(task);
        await _notificationService.NotifyTaskCompleted(task);

        return Result.Success();
    }
}
```

Expected Output

Content Expectations:

Coverage Analysis:

Existing Coverage:

- ✅ CreateTask happy path

Missing Coverage:

CreateTask:

- ❌ Missing: Invalid title (validation failure)
- ❌ Missing: Notification sent when assignee provided
- ❌ Missing: No notification when assignee is null
- ⚠️ Missing: Error handling if repository throws

CompleteTask:

- ❌ Missing: Happy path (task found and completed)
- ❌ Missing: Task not found scenario
- ❌ Missing: Task already completed (business rule violation)

- ❌ Missing: Notification sent on completion
- ❌ Missing: Repository update called

Recommended Test Scenarios:

```
## CreateTask - Missing Tests

### Scenario: Invalid title validation
- **Given:** Title is null or empty
- **When:** CreateTask is called
- **Then:** Returns Failure with error message
- **Verify:** Repository.Add NOT called

### Scenario: Notification when assigned
- **Given:** Valid task data with assignee
- **When:** CreateTask is called
- **Then:** NotificationService.NotifyTaskAssigned called once

### Scenario: No notification when unassigned
- **Given:** Valid task data without assignee
- **When:** CreateTask is called
- **Then:** NotificationService NOT called

## CompleteTask - Missing Tests

### Scenario: Complete existing task
- **Given:** Task exists and is NotStarted
- **When:** CompleteTask is called
- **Then:** Returns Success, Repository.Update called, Notification sent

### Scenario: Task not found
- **Given:** TaskId doesn't exist in repository
- **When:** CompleteTask is called
- **Then:** Returns Failure with "Task not found"

### Scenario: Task already completed
- **Given:** Task exists but is already Completed
- **When:** CompleteTask is called
- **Then:** Returns Failure (from domain rule)
```

Recommendations:

1. Aim for 80%+ coverage on Application services
2. Test all error paths, not just happy path
3. Verify collaborator interactions (repository, notifications)
4. Use FakeItEasy for mocking dependencies
5. Consider parameterized tests for validation scenarios
6. Add tests before fixing bugs (TDD for bug fixes)

Success Criteria

- [ ] Identifies specific missing test cases
- [ ] Categorizes by method/scenario
- [ ] Explains what each test should verify
- [ ] Provides coverage % guidance
- [ ] Suggests test organization improvements
- [ ] Prioritizes critical gaps

Scenario 5: Property-Based Testing Recommendation

Input

Prompt:

```
Should we use property-based testing for the Task aggregate?
If so, what properties should we test?
```

Context:

- TaskManager.Domain/Tasks/Task.cs open

Expected Output

Content Expectations:

Recommendation: Yes, property-based testing is valuable for:

- Invariant testing across wide input ranges
- Finding edge cases you didn't anticipate
- Validating business rules hold for all inputs

Properties to Test:

Property 1: Title Length Invariant

```
[Property]
public Property TitleLength_AlwaysEnforced_Regardless
OfInput()
{
    return Prop.ForAll(
        Arb.Generate<string>(),
        title =>
        {
            var result = Task.Create(TaskId.New(), title, "desc");
```

```
            return result.IsSuccess
                ? title.Length <= 200 && !string.IsNullOrWhiteSpace(title)
                : title == null || title.Trim() == "" || title.Length > 200;
        });
}
```

Property 2: Status Transitions

```
Property: Once completed, task cannot transition to other states
```

Property 3: Due Date Validation

```
Property: Setting due date always rejects past dates
```

Property 4: Immutability of Id

```
Property: TaskId never changes after creation
```

Framework: FsCheck or CsCheck for C#

Recommendations:

1. Use property-based tests for invariants and business rules
2. Complement, don't replace, example-based tests
3. Start with critical invariants (title length, status transitions)
4. Use custom generators for domain types (TaskId, TaskStatus)
5. Property tests are great for regression testing

Success Criteria

- [ ] Assesses appropriateness of property-based testing
- [ ] Identifies specific properties to test
- [ ] Provides concrete examples
- [ ] Recommends frameworks
- [ ] Explains benefits and trade-offs
- [ ] Suggests integration with existing test suite

Scenario 6: Performance Testing Guidance

Input

Prompt:

```
What performance tests should we have for the task listing API?
We expect 100,000+ tasks in production.
```

Context:

- GET /tasks?status=X&assignee=Y&page=1&pageSize=20

Expected Output

Content Expectations:

Performance Test Scenarios:

Load Testing:

1. **Concurrent Users:** 50-100 concurrent requests
2. **Response Time:** P95 < 200ms, P99 < 500ms
3. **Throughput:** 100+ requests/second

Data Volume Testing:

1. **Large Dataset:** 100,000 tasks in database
2. **Pagination:** Test at page 1, page 1000, last page
3. **Filtering:** Test with various filter combinations
4. **Sorting:** Test sort performance on large datasets

Specific Scenarios:

Scenario 1: Baseline Performance

- **Given:** 100,000 tasks in database
- **When:** GET /tasks?page=1&pageSize=20
- **Then:** Response time < 200ms

Scenario 2: Complex Filtering

- **Given:** 100,000 tasks in database
- **When:** GET /tasks?status=InProgress&assignee={userId}&overdue=true
- **Then:** Response time < 300ms

Scenario 3: Deep Pagination

- **Given:** 100,000 tasks
- **When:** GET /tasks?page=5000&pageSize=20
- **Then:** Response time < 500ms (or use cursor-based pagination)

Tools:

- NBomber or K6 for load testing
- BenchmarkDotNet for micro-benchmarks
- SQL Server profiling for query optimization

Database Considerations:

- Index on status, assignee, dueDate
- Consider cursor-based pagination for deep pages
- Query plan analysis for complex filters

Recommendations:

1. Establish performance baselines early
2. Test with production-like data volumes
3. Index critical query columns
4. Consider caching for frequent queries
5. Monitor query execution plans
6. Set up performance regression testing in CI

Success Criteria

- [ ] Identifies relevant performance metrics
- [ ] Proposes realistic load scenarios
- [ ] Considers data volume impact
- [ ] Recommends appropriate tools
- [ ] Addresses database optimization
- [ ] Provides acceptance criteria (response times)

Testing Checklist

When testing the Test Strategist agent with these scenarios:

- [ ] Output follows structured format
- [ ] Test pyramid distribution provided
- [ ] Happy path tests identified
- [ ] Error cases covered
- [ ] Edge cases at boundaries noted
- [ ] Test categorization clear (unit/integration/e2e)
- [ ] Test data and fixtures suggested
- [ ] Mocking strategy explained
- [ ] Framework recommendations provided
- [ ] Specific test examples included

- [ ] Coverage gaps identified when analyzing existing tests
- [ ] Prioritization by risk/value
- [ ] Practical and actionable advice
- [ ] Tone is educational and helpful

See Also

- Test Strategist Agent
- Custom Agent Catalog
- Testing Strategy
- Lab 07: Workflow Agents