architecture-reviewer

Agent Test Scenario: Architecture Reviewer

This document contains test scenarios for validating the Architecture Reviewer agent.

Scenario 1: Clean Architecture Compliance Check

Input

Prompt:

```
Review the Task domain model and its dependencies for Clean Architecture
compliance.
```

Context:

- TaskManager.Domain/Tasks/Task.cs open
- TaskManager.Application/Services/TaskService.cs open
- TaskManager.Infrastructure/Repositories/TaskRepository.cs open

Sample Code Issues to Identify:

```
// In TaskManager.Domain/Tasks/Task.cs
public class Task
{
    // ❌ Should not have public setters
    public string Title { get; set; }
    public TaskPriority Priority { get; set; }

    // ❌ Should not reference Infrastructure
    public void Notify()
    {
        var emailService = new EmailService(); // Infrastructure concern
        emailService.Send(...);
    }
}
```

Expected Output

Format:

```
# Architecture Review: Task Domain Model

## Summary
[2-3 sentence assessment]

## Layer Analysis

### Domain Layer
- ✅ [Positive findings]
- ⚠️ [Concerns]
- ❌ [Violations]

### Application Layer
- ✅ [Positive findings]
- ⚠️ [Concerns]

### Infrastructure Layer
- ✅ [Positive findings]
- ⚠️ [Concerns]

## Dependency Analysis
[Specific dependency violations with file references]

## DDD Pattern Review
[Assessment of aggregates, entities, value objects]

## Recommendations
1. [Critical priority]
2. [High priority]
3. [Medium priority]
```

Content Expectations:

- Identifies domain layer exposing public setters
- Flags domain referencing infrastructure (EmailService)
- Notes violation of dependency rule (Domain → Infrastructure)
- Suggests factory methods instead of public setters
- Recommends moving notifications to Application/Infrastructure
- References specific files and line numbers

Success Criteria

- [ ] Identifies all dependency violations
- [ ] References specific files (e.g., "Task.cs line 45")
- [ ] Explains why violations matter
- [ ] Provides actionable recommendations
- [ ] Follows defined output format
- [ ] Professional, constructive tone

Scenario 2: DDD Pattern Validation

Input

Prompt:

```
Review the Order aggregate for DDD best practices. Focus on:
- Aggregate boundaries
- Invariant enforcement
- Entity vs Value Object usage
```

Context:

- TaskManager.Domain/Orders/Order.cs open (hypothetical)

Sample Code:

```
// TaskManager.Domain/Orders/Order.cs
public class Order
{
    public OrderId Id { get; private set; }
    public List<OrderItem> Items { get; private set; } = new();
    public Address ShippingAddress { get; set; } // ❌ Should be private
          setter

    // ✅ Good: Factory method
```

```
    public static Order Create(OrderId id, Address shippingAddress)
    {
        return new Order
        {
            Id = id,
            ShippingAddress = shippingAddress
        };
    }

    // ❌ Bad: Public constructor still accessible
    public Order() { }

    // ⚠️ Concern: Doesn't validate invariants
    public void AddItem(Product product, int quantity)
    {
        Items.Add(new OrderItem(product, quantity));
    }

    // ✅ Good: Encapsulation
    public decimal GetTotal() => Items.Sum(i => i.SubTotal);
}

// ❌ Bad: OrderItem should be value object, not entity
public class OrderItem
{
    public Guid Id { get; set; }
    public Product Product { get; set; }
    public int Quantity { get; set; }
    public decimal SubTotal => Product.Price * Quantity;
}

// ⚠️ Concern: Address as entity instead of value object
public class Address
{
    public Guid Id { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
}
```

Expected Output

Format: Standard architecture review format

Content Expectations:

Positive Findings:

- Factory method (Order.Create) is correct pattern
- Private setters on Id and Items
- GetTotal encapsulation is good

Violations:

- Public parameterless constructor defeats factory pattern purpose
- ShippingAddress has public setter
- OrderItem is entity (has Id) when it should be value object
- Address is entity when it should be value object
- AddItem doesn't validate quantity > 0

Recommendations:

1. **Critical:** Make parameterless constructor private
2. **Critical:** Convert OrderItem to value object (remove Id, make immutable)
3. **Critical:** Convert Address to value object (record type, value equality)
4. **High:** Make ShippingAddress setter private
5. **High:** Add invariant validation in AddItem (quantity > 0)
6. **Medium:** Consider domain events for OrderItemAdded

Success Criteria

- [ ] Identifies public constructor alongside factory method
- [ ] Distinguishes entities from value objects correctly
- [ ] Recognizes missing invariant validation
- [ ] Explains entity vs value object criteria
- [ ] Provides specific code examples in recommendations
- [ ] Prioritizes by impact

Scenario 3: Repository Pattern Review

Input

Prompt:

```
Review the repository implementations for adherence to DDD repository
patterns.
```

Context:

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

Sample Code:

```
// TaskManager.Domain/Repositories/ITaskRepository.cs
public interface ITaskRepository
{
```

```
    // ❌ Bad: Generic CRUD naming
    Task<TaskEntity> GetById(Guid id);
    void Create(TaskEntity task);
    void Update(TaskEntity task);
    void Delete(Guid id);

    // ✅ Good: Business-intent naming
    Task<IEnumerable<TaskEntity>> FindOverdueTasks();
}

// TaskManager.Infrastructure/Repositories/TaskRepository.cs
public class TaskRepository : ITaskRepository
{
    private readonly DbContext _context;

    public async Task<TaskEntity> GetById(Guid id)
    {
        // ⚠️ Concern: Returning null instead of Option or throwing
        return await _context.Tasks.FindAsync(id);
    }

    // ❌ Bad: Repository knows about SaveChanges
    public void Create(TaskEntity task)
    {
        _context.Tasks.Add(task);
        _context.SaveChanges(); // Should be in Unit of Work
    }
}
```

Expected Output

Content Expectations:

Violations:

- Generic CRUD method names (GetById, Create, Update, Delete) instead of business language
- SaveChanges called in repository (should be Unit of Work responsibility)
- Returns null instead of Option type or throwing NotFound exception

Recommendations:

1. **High:** Rename repository methods to use ubiquitous language:
   - `GetById` → `FindTask` or `LoadTask`
   - `Create` → `Add` (acceptable) or domain-specific verb
   - `Delete` → `Remove`
2. **High:** Remove SaveChanges from repository, use Unit of Work pattern
3. **Medium:** Return `Task<TaskEntity?>` with null handling or throw `TaskNotFoundException`
4. **Medium:** Add business-intent query methods like `FindOverdueTasks`

Success Criteria

- [ ] Identifies CRUD naming anti-pattern
- [ ] Recognizes SaveChanges in repository violates SRP
- [ ] Suggests business-intent naming
- [ ] References DDD repository pattern principles
- [ ] Explains why these patterns matter

Scenario 4: Cross-Layer Dependency Violation

Input

Prompt:

```
Review this code for architectural issues.
```

Context:

- TaskManager.Application/Services/NotificationService.cs open

Sample Code:

```
// TaskManager.Application/Services/NotificationService.cs
using TaskManager.Infrastructure.Email; // ❌ Application → Infrastructure

namespace TaskManager.Application.Services;

public class NotificationService
{
    // ❌ Application directly depends on Infrastructure
    private readonly SmtpEmailSender _emailSender;

    public NotificationService()
    {
        // ❌ Newing up infrastructure
        _emailSender = new SmtpEmailSender("smtp.example.com");
    }

    public void NotifyTaskAssigned(Task task, User user)
    {
        _emailSender.SendEmail(user.Email, "Task Assigned", ...);
    }
}
```

Expected Output

Content Expectations:

Violations:

- Application layer directly references Infrastructure namespace
- Application creates concrete Infrastructure class (SmtpEmailSender)
- Violates Dependency Inversion Principle
- Tight coupling to email implementation

Recommendations:

1. **Critical:** Define `IEmailSender` interface in Application layer
2. **Critical:** Move `SmtpEmailSender` implementation to Infrastructure
3. **Critical:** Inject `IEmailSender` via constructor (DI)
4. **High:** Remove `new SmtpEmailSender()` - use dependency injection

Example:

```
// TaskManager.Application/Ports/IEmailSender.cs
public interface IEmailSender
{
    void SendEmail(string to, string subject, string body);
}

// TaskManager.Infrastructure/Email/SmtpEmailSender.cs
public class SmtpEmailSender : IEmailSender { ... }

// TaskManager.Application/Services/NotificationService.cs
public class NotificationService
{
    private readonly IEmailSender _emailSender;

    public NotificationService(IEmailSender emailSender)
    {
        _emailSender = emailSender;
    }
}
```

Success Criteria

- [ ] Identifies dependency direction violation
- [ ] Explains Dependency Inversion Principle
- [ ] Provides concrete refactoring example
- [ ] Shows interface in Application, implementation in Infrastructure

Mentions dependency injection

Scenario 5: Comprehensive Review

Input

Prompt:

```
Perform a comprehensive architecture review of the TaskManager solution.
Check all layers and identify the top 5 most critical issues.
```

Context:

- Entire TaskManager solution open

Expected Output

Format: Full architecture review covering all layers

Content Expectations:

Summary:

- Overall assessment of architecture health
- Major themes (e.g., "generally follows Clean Architecture but has several dependency violations")

Layer Analysis:

- Each layer (Domain, Application, Infrastructure, Api) analyzed
- Mix of ✅, ⚠️, ❌ findings per layer

Top 5 Critical Issues:

1. [Most critical architectural violation]
2. [Second most critical]
3. [Third]
4. [Fourth]
5. [Fifth]

Dependency Graph:

- Visual or textual representation of current dependencies
- Highlight violations

Recommendations:

- Prioritized by impact and effort
- Quick wins identified
- Long-term refactoring suggestions

Success Criteria

- [ ] Covers all architectural layers
- [ ] Prioritizes findings effectively
- [ ] Provides roadmap for improvement
- [ ] Balances quick wins with long-term goals
- [ ] References specific files throughout
- [ ] Executive summary suitable for stakeholders

Scenario 6: Positive Review

Input

Prompt:

```
Review the User aggregate for architectural compliance.
```

Context:

- TaskManager.Domain/Users/User.cs open

Sample Code (Well-Designed):

```
// TaskManager.Domain/Users/User.cs
public sealed class User  // ✅ Sealed
{
    public UserId Id { get; private set; }  // ✅ Private setter
    private string _email;  // ✅ Backing field

    // ✅ Private constructor
    private User(UserId id, string email)
    {
        Id = id;
        _email = email;
    }

    // ✅ Factory method with validation
    public static Result<User> Create(UserId id, string email)
    {
        if (string.IsNullOrWhiteSpace(email))
            return Result.Failure<User>("Email is required");

        if (!email.Contains("@"))
            return Result.Failure<User>("Invalid email format");
```

```
        return Result.Success(new User(id, email));
    }

    // ✅ Business method with validation
    public Result ChangeEmail(string newEmail)
    {
        if (string.IsNullOrWhiteSpace(newEmail))
            return Result.Failure("Email is required");

        _email = newEmail;
        // ✅ Could raise domain event here
        return Result.Success();
    }
}
```

Expected Output

Content Expectations:

Positive Findings:

- Sealed class (prevents inheritance where not intended)
- Private setters protect invariants
- Private constructor prevents invalid state
- Factory method with validation
- Business methods validate before mutation
- Result type for error handling
- No dependencies on other layers
- Aggregate root properly encapsulated

Potential Enhancements (not violations):

- Consider domain event for EmailChanged
- Could add value object for Email with validation
- May want to store email history for audit

Tone:

- Acknowledges good practices
- Constructive enhancement suggestions
- Educational about why patterns are well-applied

Success Criteria

- [ ] Recognizes and acknowledges good architecture
- [ ] Explains why patterns are correct
- [ ] Suggests enhancements, not just violations

- [ ] Educational and encouraging tone
- [ ] Distinguishes "nice to have" from "must fix"

Testing Checklist

When testing the Architecture Reviewer agent with these scenarios:

- [ ] All scenarios produce structured output in expected format
- [ ] ✅ ⚠️ ❌ symbols used appropriately
- [ ] Specific file references included (file name, ideally line numbers)
- [ ] Dependency violations correctly identified
- [ ] DDD patterns assessed accurately (aggregate, entity, value object)
- [ ] Recommendations prioritized by impact
- [ ] Code examples provided in recommendations
- [ ] Tone is professional and constructive
- [ ] Explanations include "why" not just "what"
- [ ] Positive findings acknowledged, not just violations
- [ ] Distinguishes critical issues from nice-to-haves
- [ ] Quick wins identified where applicable

See Also

- Architecture Reviewer Agent
- Custom Agent Catalog
- Lab 07: Workflow Agents
- Architecture Design
- ADR 0001: Use Clean Architecture