# Дополнение задач с legacy кодом

Конкретные примеры legacy кода с типичными проблемами, которые предстоит решить.

## Задачи с Legacy кодом (дополненные)

## Задача 2: "Система квестов с унаследованным кодом"

**Сложность:** 2/5

## Исходный Legacy код:

```
public class LegacyQuestManager
{
   private static List<Quest> _quests = new List<Quest>();
   private static int _nextId = 1;

   // Проблема 1: Статическая зависимость от базы данных
   public void CompleteQuest(int questId, string playerName)
   {
      // Проблема 2: Прямое создание зависимостей
      var database = new QuestDatabase();
      var logger = new ConsoleLogger();

      var quest = _quests.FirstOrDefault(q => q.Id == questId);
      if (quest != null)
      {
         // Проблема 3: Смесь логики и побочных эффектов
         quest.IsCompleted = true;
         quest.CompletedBy = playerName;
         quest.CompletionDate = DateTime.Now;

         database.Save(quest); // Может упасть!
         logger.Write($"Quest {questId} completed by {playerName}");

         // Проблема 4: Жесткая связь с email системой
         var emailService = new EmailService();
         emailService.SendRewardEmail(playerName, quest.Reward);
```

```
        }
    }

    // Проблема 5: Метод делает слишком много
    public void AddQuest(string title, string description, int reward)
    {
        var quest = new Quest
        {
            Id = _nextId++,
            Title = title,
            Description = description,
            Reward = reward,
            IsCompleted = false
        };

        _quests.Add(quest);

        // Проблема 6: Неявные зависимости
        File.WriteAllText($"quest_{quest.Id}.json",
            JsonSerializer.Serialize(quest));
    }

    // Проблема 7: Непотокобезопасный код
    public List<Quest> GetActiveQuests()
    {
        return _quests.Where(q => !q.IsCompleted).ToList();
    }
}

// Вспомогательные классы с проблемами
public class QuestDatabase
{
    public void Save(Quest quest)
    {
        // Симуляция проблем с БД
```

```
        if (DateTime.Now.Second % 3 == 0)
            throw new Exception("Database connection failed!");

        Thread.Sleep(100); // Блокирующий вызов
    }
}

public class EmailService
{
    public void SendRewardEmail(string player, int reward)
    {
        Thread.Sleep(200); // Долгая операция
        Console.WriteLine($"Email sent to {player} about reward {reward}");
    }
}
```

### Задачи для студентов:

1. Проанализировать 7 проблем в коде (помечены комментариями)
2. Применить Sprout Method для добавления системы уведомлений
3. Использовать Wrap Method для добавления логирования
4. Сделать код тестируемым и потокобезопасным

### Задача 7: "Рефакторинг боевой системы с тестами"

**Сложность:** 4/5

### Исходный Legacy код:

```
public class CombatSystem
{
    private static readonly Random _random = new Random();
    private List<BattleLog> _logs = new List<BattleLog>();

    // ПРОБЛЕМА 1: Слишком много ответственности в одном методе
    public BattleResult ResolveCombat(Player attacker, Player defender)
    {
        // ПРОБЛЕМА 2: Смесь вычислений и побочных эффектов
        Console.WriteLine($"Combat: {attacker.Name} vs {defender.Name}");
```

```
        int attackerDamage = CalculateDamage(attacker, defender);
        int defenderDamage = CalculateDamage(defender, attacker);

        // ПРОБЛЕМА 3: Состояние гонки при многопоточном доступе
        attacker.Health -= defenderDamage;
        defender.Health -= attackerDamage;

        // ПРОБЛЕМА 4: Жесткая привязка к UI (Console)
        if (attacker.Health <= 0)
        {
            Console.WriteLine($"{attacker.Name} died!");
            attacker.IsAlive = false;
        }

        if (defender.Health <= 0)
        {
            Console.WriteLine($"{defender.Name} died!");
            defender.IsAlive = false;
        }

        // ПРОБЛЕМА 5: Сложно тестировать из-за побочных эффектов
        var result = new BattleResult
        {
            Winner = attacker.Health > defender.Health ? attacker : defender,
            DamageDealt = attackerDamage,
            DamageReceived = defenderDamage
        };

        // ПРОБЛЕМА 6: Блокирующие операции
        SaveBattleLog(result);
        SendCombatNotification(attacker, defender, result);

        return result;
    }
```

```
    // ПРОБЛЕМА 7: Непотокобезопасные вычисления
    private int CalculateDamage(Player attacker, Player defender)
    {
        // Гонка данных при использовании Random
        int baseDamage = _random.Next(attacker.MinDamage, attacker.MaxDamage);
        int defense = _random.Next(defender.MinDefense, defender.MaxDefense);

        return Math.Max(1, baseDamage - defense);
    }

    // ПРОБЛЕМА 8: Долгие синхронные операции
    private void SaveBattleLog(BattleResult result)
    {
        Thread.Sleep(50); // Имитация работы с БД
        _logs.Add(new BattleLog { Result = result, Timestamp = DateTime.Now });
    }

    private void SendCombatNotification(Player p1, Player p2, BattleResult result)
    {
        Thread.Sleep(30); // Имитация отправки уведомления
        Console.WriteLine($"Notification: {result.Winner.Name} won!");
    }

    // ПРОБЛЕМА 9: Отсутствие асинхронных методов
    public List<BattleLog> GetCombatHistory()
    {
        return _logs;
    }
}

// Классы данных с проблемами
public class Player
{
    public string Name { get; set; }
```

```
    public int Health { get; set; } // Публичное поле - проблема!
    public bool IsAlive { get; set; }
    public int MinDamage { get; set; }
    public int MaxDamage { get; set; }
    public int MinDefense { get; set; }
    public int MaxDefense { get; set; }
}
```

**Задачи для студентов:**

1. Выделить чистые методы для вычисления урона (Sprout Method)
2. Оборачивать старые методы для добавления функциональности (Wrap Method)
3. Добавить потокобезопасность с использованием lock и ConcurrentCollection
4. Создать асинхронные версии методов
5. Написать модульные тесты для ключевых компонентов

**Задача 10: "Полная RPG система с Legacy рефакторингом"**

**Сложность:** 5/5

**Исходный Legacy код (сильно упрощенный):**

```
public class RPGGameEngine
{
    private static RPGGameEngine _instance;
    private List<Player> _players = new List<Player>();
    private List<Item> _items = new List<Item>();

    // ПРОБЛЕМА: Singleton антипаттерн
    public static RPGGameEngine Instance
    {
        get
        {
            if (_instance == null)
                _instance = new RPGGameEngine();
            return _instance;
        }
```

```
}

// ПРОБЛЕМА: God Object - делает всё
public void ProcessGameTick()
{
    // Обновляем игроков
    foreach (var player in _players)
    {
        UpdatePlayer(player);
        CheckQuests(player);
        RegenerateHealth(player);
    }

    // Обновляем мир
    RespawnItems();
    ProcessCombat();
    SaveGameState(); // Блокирующая операция!
}

// ПРОБЛЕМА: Длинные методы со смешанной логикой
private void UpdatePlayer(Player player)
{
    if (player.Health <= 0)
    {
        HandlePlayerDeath(player);
    }

    // Логика восстановления
    if (player.Health < player.MaxHealth)
    {
        player.Health += player.HealthRegen;
        Console.WriteLine($"{player.Name} regenerated health");
    }

    // Проверка квестов
```

```
    foreach (var quest in player.ActiveQuests)
    {
        if (quest.IsCompleted)
        {
            GiveReward(player, quest.Reward);
            player.CompletedQuests.Add(quest);
            player.ActiveQuests.Remove(quest);
        }
    }

    // Сохранение состояния
    if (player.Level > player.LastSavedLevel)
    {
        SavePlayerProgress(player);
    }
}

// ПРОБЛЕМА: Прямые зависимости
private void SavePlayerProgress(Player player)
{
    var database = new PlayerDatabase();
    var backup = new BackupService();

    try
    {
        database.Save(player);
        backup.CreateBackup(player);
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error saving player: {ex.Message}");
        // ПРОБЛЕМА: Проглатывание исключений
    }
}
```

```
    // ПРОБЛЕМА: Отсутствие асинхронности
    private void SaveGameState()
    {
        // Долгая блокирующая операция
        Thread.Sleep(1000);
        File.WriteAllText("gamesave.json",
            JsonSerializer.Serialize(new { Players = _players, Items = _items }));
    }

    // ПРОБЛЕМА: Непотокобезопасные операции
    public void AddPlayer(Player player)
    {
        _players.Add(player);
        Console.WriteLine($"Player {player.Name} added");
    }

    public void AddItem(Item item)
    {
        _items.Add(item);
        Console.WriteLine($"Item {item.Name} added");
    }
}

// Вспомогательные классы с проблемами
public class PlayerDatabase
{
    public void Save(Player player)
    {
        // Имитация проблем с БД
        if (DateTime.Now.Millisecond < 100)
            throw new Exception("Database timeout");

        Thread.Sleep(100);
    }
}
```

```
public class BackupService
{
   public void CreateBackup(Player player)
   {
      Thread.Sleep(200);
      // Может упасть в 10% случаев
      if (new Random().Next(0, 10) == 0)
         throw new Exception("Backup failed");
   }
}
```

**Конкретные задачи по рефакторингу:**

1. **Разбить God Object:**
   - Создать отдельные классы: PlayerManager, CombatSystem, QuestSystem, InventorySystem
   - Внедрить dependency injection
2. **Добавить тестируемость:**

```
// Было:
var database = new PlayerDatabase();

// Стало:
public class PlayerService
{
   private readonly IPlayerRepository _repository;
   public PlayerService(IPlayerRepository repository)
   {
      _repository = repository;
   }
}
```

3. **Добавить многопоточность:**
   - Асинхронное сохранение игроков
   - Параллельная обработка разных систем
   - Потокобезопасные коллекции

4. **Решить проблемы синхронизации:**
    - Заменить Singleton на нормальное управление зависимостями
    - Добавить правильные примитивы синхронизации

**Дополнительные указания для студентов:**

**Шаблон решения Legacy задач:**

1. **Анализ (30 минут):**
    - Составьте список всех проблем в коде
    - Определите приоритеты исправления
    - Наметьте план рефакторинга
2. **Создание тестов (45 минут):**
    - Напишите тесты ДО рефакторинга
    - Убедитесь, что тесты падают ожидаемым образом
    - Создайте тестовые двойники (mock objects)
3. **Рефакторинг (60+ минут):**
    - Применяйте изменения небольшими шагами
    - Постоянно запускайте тесты
    - Используйте техники из модуля 1
4. **Добавление новой функциональности (30 минут):**
    - Sprout Method для новых фич
    - Wrap Method для улучшения существующего кода

**Пример применения Sprout Method:**

```
// Было в Legacy коде:
public void CompleteQuest(int questId, string playerName)
{
   // ... много логики ...
   emailService.SendRewardEmail(playerName, quest.Reward);
}

// Стало после рефакторинга:
```

```
public void CompleteQuest(int questId, string playerName)
{
    // Существующая логика
    var quest = FindQuest(questId);
    if (quest != null)
    {
        MarkQuestCompleted(quest, playerName);

        // Sprout Method - новая функциональность
        NotifyQuestCompletion(quest, playerName);
    }
}

// Новый чистый метод
private void NotifyQuestCompletion(Quest quest, string playerName)
{
    var notification = new QuestNotification(quest, playerName);
    _notificationService.Send(notification);

    if (quest.Reward > 1000)
    {
        _achievementService.UnlockAchievement(playerName, "RichReward");
    }
}
```

Здесь собраны конкретные проблемы и примеры legacy (унаследованного) кода. Данные примеры помогут понять с какими испытаниями можно столкнуться при работе с таким кодом.