Revised IDF Operation First Strike - Solo Development Plan

This revised plan addresses the Single Responsibility Principle (SRP) violations in the original design by properly separating concerns and ensuring each class has only one reason to change.

Project Structure with Improved SRP

```
IDFOperationFirstStrike/
├── src/
│   ├── Core/                  // Core interfaces and domain models
│   │   ├── Interfaces/        // All interfaces
│   │   │   ├── IStrikeUnit.cs
│   │   │   ├── IFuelConsuming.cs
│   │   │   ├── IIntelligenceProvider.cs
│   │   │   ├── IIntelligenceAnalyzer.cs
│   │   │   └── IStrikeExecutor.cs
│   │   ├── Models/            // Domain models (data only)
│   │   │   ├── StrikeOption.cs
│   │   │   ├── Terrorist.cs
│   │   │   ├── IntelligenceMessage.cs
│   │   │   └── StrikeReport.cs
│   ├── StrikeUnits/           // Strike unit implementations (behavior only)
│   │   ├── F16FighterJet.cs
│   │   ├── HermesDrone.cs
│   │   └── M109Artillery.cs
│   ├── Organizations/         // Core organization entities
│   │   ├── IDF.cs             // Simplified IDF model
│   │   └── Hamas.cs           // Simplified Hamas model
│   ├── Managers/              // Business logic managers (separated
responsibilities)
│   │   ├── StrikeUnitManager.cs    // Manages strike units
│   │   ├── IntelligenceManager.cs  // Manages intelligence data
│   │   ├── TerroristManager.cs     // Manages terrorist data
│   │   ├── StrikeCoordinator.cs    // Coordinates strikes
│   │   └── SimulationManager.cs    // Overall simulation logic
│   ├── Services/              // Support services
│   │   ├── IntelligenceGenerator.cs  // Generates intelligence
│   │   ├── StrikeExecutor.cs     // Executes strikes
│   │   └── StrikeHistorian.cs    // Records and queries strike history
│   ├── Utils/                 // Utility classes
│   │   ├── WeaponScoreRegistry.cs
│   │   └── LocationTargetTypeMapper.cs
│   ├── Presentation/          // UI concerns only
│   │   ├── ConsoleDisplayManager.cs  // Manages all console output
│   │   ├── IntelligenceDisplay.cs    // Intelligence display logic
│   │   ├── StrikeUnitDisplay.cs      // Strike unit display logic
│   │   ├── TerroristDisplay.cs       // Terrorist display logic
│   │   └── MenuController.cs         // Menu handling
│   └── Program.cs             // Main program entry (minimal code)
└── IDFOperationFirstStrike.csproj
```

Git Branch Strategy

Main Branches

- `main` - Stable codebase (only merge completed features here)
- `development` - Integration branch for completed features

Feature Branches

1. `feature/core-interfaces` - Core interfaces
2. `feature/core-models` - Domain models
3. `feature/strike-units` - Strike unit implementations
4. `feature/organizations` - IDF and Hamas organizations
5. `feature/managers` - Business logic managers
6. `feature/services` - Support services
7. `feature/utils` - Utility classes
8. `feature/presentation` - UI-related classes
9. `feature/program` - Main program and simulation coordination

Phased Development Plan (With Improved SRP)

Phase 1: Core Domain Models and Interfaces (Branches: feature/core-interfaces, feature/core-models)

- Set up core interfaces with focused responsibilities
- Create domain models that focus only on data, not behavior
- Ensure proper separation between data and operations

Phase 2: Organizations and Utils (Branches: feature/organizations, feature/utils)

- Implement Hamas and IDF as simple data containers
- Create utility classes that encapsulate specific calculations
- Focus on data storage, not behavior

Phase 3: Strike Units (Branch: feature/strike-units)

- Implement strike units focused on core strike behavior
- Remove display logic and reporting from strike units
- Ensure strike units don't have multiple responsibilities

Phase 4: Managers (Branch: feature/managers)

- Create separate manager classes for each domain concern
- Implement proper separation of concerns

- Ensure each manager has a single responsibility

Phase 5: Services (Branch: feature/services)

- Implement support services with focused responsibilities
- Create execution and reporting services
- Ensure clean boundaries between services

Phase 6: Presentation Layer (Branch: feature/presentation)

- Implement display classes separated from business logic
- Create menu controllers and UI handlers
- Ensure all console output is separated from business logic

Phase 7: Program Coordination (Branch: feature/program)

- Create minimal Program.cs that delegates to appropriate classes
- Implement SimulationManager for overall coordination
- Ensure proper dependency injection and configuration

Implementing Each SRP-Compliant Class

Core Interfaces Example

```csharp
// IStrikeUnit.cs - Focused on strike unit capabilities only
public interface IStrikeUnit
{
    string Name { get; }
    int Ammo { get; }
    int Fuel { get; }
    bool CanStrike(string targetType);
    void PerformStrike(Terrorist target, IntelligenceMessage intel); // No console out
}

// Separate interface for display concerns
public interface IStrikeDisplay
{
    void DisplayStrikeResults(IStrikeUnit unit, Terrorist target, IntelligenceMessage
}
```

Domain Models Example

csharp

```csharp
// Terrorist.cs - Focused on data only
public class Terrorist
{
    public string Name { get; set; }
    public int Rank { get; set; }
    public bool IsAlive { get; set; } = true;
    public List<string> Weapons { get; set; } = new List<string>();

    // No calculation methods, no display logic
}
```

Managers Example

csharp

```csharp
// TerroristManager.cs - Handles terrorist operations
public class TerroristManager
{
    // Calculates weapon score without exposing implementation details
    public int GetWeaponScore(Terrorist terrorist)
    {
        return terrorist.Weapons.Sum(w => WeaponScoreRegistry.GetScore(w)) * terrorist
    }

    public List<Terrorist> GetAliveTargets(List<Terrorist> terrorists)
    {
        return terrorists.Where(t => t.IsAlive).ToList();
    }

    public Terrorist GetMostDangerousTarget(List<Terrorist> terrorists)
    {
        return GetAliveTargets(terrorists)
            .OrderByDescending(t => GetWeaponScore(t))
            .FirstOrDefault();
    }

    // No display logic, no console output
}
```

Services Example

```csharp
// IntelligenceGenerator.cs - Only generates intelligence
public class IntelligenceGenerator
{
    private readonly Random _random = new Random();
    private readonly string[] _locations = { "home", "in a car", "outside" };

    public IntelligenceMessage Generate(Terrorist terrorist)
    {
        return new IntelligenceMessage
        {
            Target = terrorist,
            Location = _locations[_random.Next(_locations.Length)],
            Timestamp = DateTime.Now
        };
    }

    // No display logic, no console output
}
```

Presentation Example

```csharp
// IntelligenceDisplay.cs - Only handles display logic
public class IntelligenceDisplay
{
    public void ShowIntelligenceSummary(List<IntelligenceMessage> messages)
    {
        Console.WriteLine("\n[Intelligence Summary]");
        Console.WriteLine($"Total reports: {messages.Count}");

        var groups = messages.GroupBy(m => m.Target).ToList();
        foreach (var group in groups)
        {
            Console.WriteLine($"- {group.Key.Name}: {group.Count()} reports");
        }
    }

    // Only display concerns, no business logic
}
```

Implementation Strategy

1. **Start with interfaces and models**: Build a solid foundation with proper separation of concerns

2. **Use dependency injection**: Inject dependencies rather than creating them
3. **Create focused tests**: Test each component in isolation
4. **Implement UI last**: Ensure all business logic is independent of the presentation layer

Benefits of This Approach

1. **Improved maintainability**: Each class has a clear, single responsibility
2. **Better testability**: Business logic can be tested without UI dependencies
3. **Easier extension**: New features can be added without modifying existing code
4. **Reduced complexity**: Each class is simpler and more focused
5. **Better SOLID compliance**: Adheres to all SOLID principles, not just SRP

Example Program.cs with Proper SRP

csharp

```csharp
class Program
{
    static void Main(string[] args)
    {
        // Create services (could use dependency injection in a more complex app)
        var terroristManager = new TerroristManager();
        var intelManager = new IntelligenceManager();
        var strikeManager = new StrikeUnitManager();
        var strikeCoordinator = new StrikeCoordinator(strikeManager);
        var strikeHistorian = new StrikeHistorian();

        // Create display handlers
        var intelDisplay = new IntelligenceDisplay();
        var terroristDisplay = new TerroristDisplay();
        var strikeDisplay = new StrikeUnitDisplay();

        // Create menu controller
        var menuController = new MenuController(
            terroristManager,
            intelManager,
            strikeCoordinator,
            strikeHistorian,
            intelDisplay,
            terroristDisplay,
            strikeDisplay);

        // Create simulation manager that coordinates everything
        var simulationManager = new SimulationManager(
            menuController,
            terroristManager,
            intelManager,
            strikeCoordinator);

        // Run the simulation
        simulationManager.Initialize();
        simulationManager.Run();
    }
}
```