Introduction

This comprehensive refactoring document outlines six key improvements to enhance the application's design, maintainability, and functionality. The refactoring operations cover:

1. **Extended State Pattern** for Tournament
2. **Strategy Pattern** for player behaviors
3. **Enhanced startup phase** with bot players
4. **Adapter Pattern** for file handling
5.

Each section includes identified targets, rationale, before/after depictions, and testing requirements.

1. Adding Tournament Phase to State Pattern

Targets Identified

The application currently lacks support for tournament mode, which would allow multiple game instances to run with different configurations. This requires extending our existing State Pattern implementation.

Refactoring Operation

Target 1: Tournament Phase Implementation

- **Before**: No tournament capability exists in the state machine
- **After**:
    - Create `TournamentPlay` for configuration
    - Implement method `tournamentGame()` to manage multiple games
- **Tests**:
    - Test tournament configuration validation
    - Verify proper game instance creation
    - Check results aggregation
    - Test state transitions

Before/After Depiction

Before State Pattern

```
class GameEngine {
   void start() {
      // without tournament phase
   }
}
```

After Tournament Implementation

```
class TournamentPlay implements Phase {
    Public TournamentPlay(GameEngine p_ge) {
        super(p_ge);
    }
    void tournamentGame(GameEngine engine) {
      // Configure and start tournament
     }
}
```

2. Strategy Pattern for Player's issue_order()

Targets Identified

The current implementation handles all order issuance through a single `issue_order()` method with conditional logic for different player types. This makes it difficult to add new player behaviors or modify existing ones.

Refactoring Operation

Target 2: Player Behavior Strategies

- **Before**: Monolithic `issue_order()` method with conditional logic
- **After**:
    - Create `PlayerBehavior` interface with `issue_order()` method
    - Implement concrete strategies (`HumanBehavior`, `AggressiveBehavior`, `RandomBehavior`, `BenevolentBehavior`, `CheaterBehavior` etc.)
    - Player class delegates order issuance to strategy object
- **Tests**:

        - Test each strategy's order issuance behavior
        - Test strategy initiation at runtime
        - Verify order validation for each behavior strategy type
- **Why Necessary**: Enables flexible player behavior modification and extension without changing core player class

Before/After Depiction

Before

```
class Player {
   void issue_order() {
      if (isHuman) {
         // Manual order input
      } else if (isBot) {
         // Bot logic
      }
   }
}
```

After

```
interface PlayerBehavior {
   Order issue_order();
}

class AggressiveBehavior implements PlayerBehavior {
   Order issue_order() {
      // Aggressive bot logic
   }
}

class Player {
   private PlayerBehavior strategy;

   void setStrategy(PlayerBehavior s) {
      this.strategy = s;
   }

   void issue_order() {
      Order order = strategy.issue_order();
```

```
        // Add order to the queue
    }
}
```

3. Enhanced Startup Phase with Bot Players

Targets Identified

The startup phase currently only supports manual players, limiting gameplay options. Adding bot players with different behaviors will make the game more versatile.

Refactoring Operation

Target 3: Bot Player Integration

- **Before**: Only manual player creation during startup
- **After**:
    - Extend `StartupState` to support bot player configuration
    - Add bot behavior options (random, benevolent, aggressive, cheater)
    - Implement player method for creating different player types
- **Tests**:
    - Test bot player creation during startup
    - Verify behavior consistency for each behavior type
    - Test mixed human/bot gameplay scenarios
- **Why Necessary**: Enables single-player mode and provides more gameplay variety

Before/After Depiction

Before

```
class Startup {
    void startUp() {
        //Only creates human players
       //Manual play
    }
}
```

After

```
class Startup {
   void startUp() {
      //options to create different players with different behaviors
     //Ability to auto play
   }
}
```

4. Adapter Pattern for File Handling

Targets Identified

The application needs to support both "Domination" and "Conquest" map file formats while maintaining a consistent interface for map operations.

Refactoring Operation

Target 4: File Format Adapters

- **Before**: Separate methods for each file format
- **After**:
    - Create `FileManager` interface
    - Implement `DominationMapAdapter` and `ConquestMapAdapter`
    - Use adapter pattern to unify file operations
- **Tests**:
    - Test reading/writing both formats
    - Verify format conversion
    - Test invalid file handling
- **Why Necessary**: Provides consistent interface while supporting multiple formats

Before/After Depiction

Before

```
class MapFileHandler {
   void writeDomination(String file) { /* ... */ }
   Void readDomination(String file) { /* ... */ }
}
```

After

```
interface FileManager {
```

```
    void read(String file, Map map);
    void write(String file, Map map);
}

class DominationMapAdapter implements FileManager {
    // Implements Domination format
}

class MapFileHandler {
    private FileManager adapter;

    void setAdapter(FileManager a) {
        this.adapter = a;
    }

    void load(Map map, String file) { }
    void save(Map map, String file) { }
}
```