Class Hierarchy

The project is centered around the `Player` abstract class, which is extended by `AIPlayer` and `HumanPlayer`. This enables the program to handle both AI and human players uniformly while allowing each to define their own behavior for methods such as `takeTurn()` and `selectCharacter()`. The `Game` class manages the overall gameplay and interacts with players through the `Player` superclass, promoting generalization. Other important classes include `CharacterCard`, `DistrictCard`, `Deck<T>`, and utility classes like `CommandHandler`, `GameState`, and `PurpleCardEffects`.

Object-Oriented Design

The design uses well-defined packages (`citadels.card`, `citadels.player`, etc.) and separates responsibilities across classes. `Game` coordinates the game loop, while `CommandHandler` handles user input, and `GameState` is responsible for saving and loading. Each class is focused and modular, improving maintainability and testing. Collections such as `List<Player>` and `Deck<DistrictCard>` are used to manage game components. The `Deck<T>` class is generic and reusable across different card types. The design supports scalability by isolating domain logic, UI input, and data persistence in different classes.

OO Principles

The code applies object-oriented principles effectively:

- **Encapsulation:** Fields such as `gold`, `hand`, and `city` are private or protected, and can only be accessed or modified through public methods like `getGold()` or `build()`. This keeps object state consistent and secure.
- **Inheritance:** `AIPlayer` and `HumanPlayer` inherit from `Player`, enabling code reuse and simplifying turn logic.
- **Polymorphism:** The `Game` class handles all players using the `Player` reference type, and behavior like character selection or building is resolved at runtime based on the actual subclass.
- **Abstraction:** Abstract classes (like `Player`) define a common interface while hiding implementation details, supporting clean, extensible design.
- **Single Responsibility:** Each class handles one primary task. `Game` handles game logic, `Deck<T>` manages card storage, `GameState` saves state, and `PurpleCardEffects` handles special logic.

Extension

A natural extension would be to add a new character with a unique ability, such as a `"Spy"` that reveals another player's hand. To implement this:

1. Add a new `CharacterCard` instance named `"Spy"` with a distinct rank.
2. Modify the character selection logic in `Game` to include this card.
3. In the `Game` turn logic, add a condition for `if (character.getName().equals("Spy"))`, and define the special effect, possibly invoking a new method in `Player` like `revealHand()`.
4. Optionally, update `AIPlayer` logic to decide when to pick or counter the `"Spy"`.

Because of the modular class design and use of polymorphism, this feature could be added with minimal disruption to existing code.

J-Unit Testing

citadels

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
citadels		89%		76%	135	397	107	1,099	10	77	0	5
citadels.player		97%		86%	37	196	10	417	1	54	0	3
citadels.card		98%		100%	0	29	3	70	0	15	0	3
citadels.effect		99%		96%	2	45	1	82	0	13	0	1
citadels.util		100%		100%	0	11	0	17	0	10	0	1
Total	624 of 8,122	92%	184 of 992	81%	174	678	121	1,685	11	169	0	13

Figure 1: J-Unit Tests