Assignment 1: The Object-Oriented Blueprint

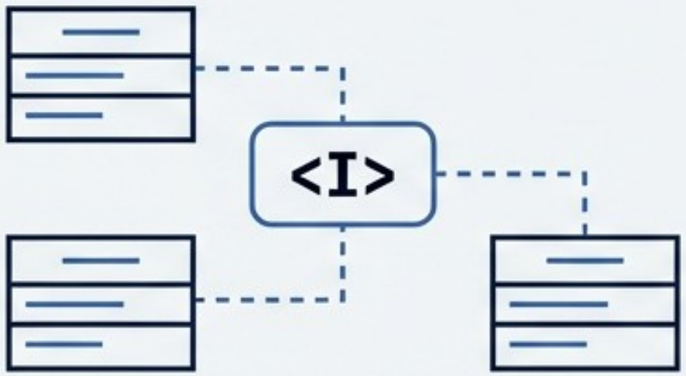
Designing a Console-Based Crazy 8's Game

Generated from the assignment file by Notebook LM

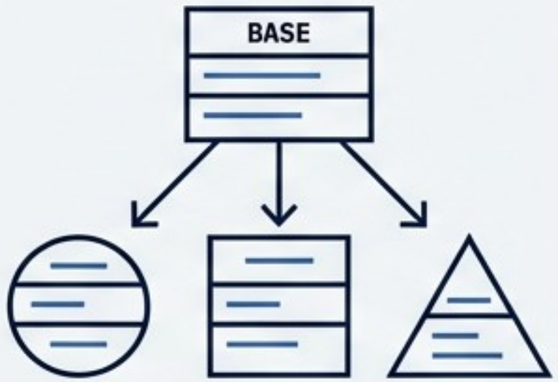
KSU SWE 4743: Object-Oriented Design

Your Mission: Design with Intent, Not for Sophistication

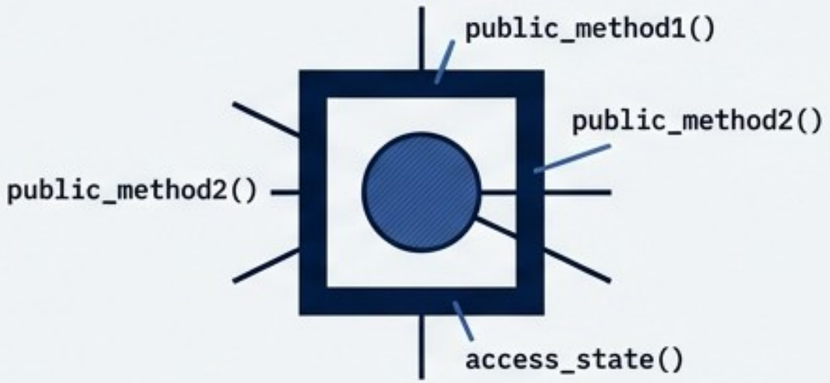
This assignment is not about creating a complex game. It is an explicit exercise in demonstrating correct and intentional use of foundational **object**-oriented **design principles.**



Interfaces & Abstract Classes



Polymorphism & Dynamic Dispatch



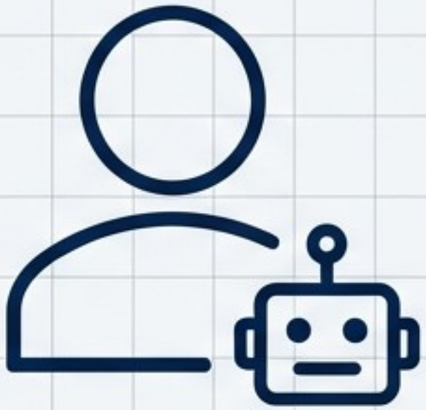
True Encapsulation

Mastering the Architect’s Toolkit

By completing this assignment, you will demonstrate that you can:

- Design software around **abstractions**, not concrete implementations.
- Define contracts using **interfaces** and share behavior with **abstract classes**.
- Apply **polymorphism** using collections of base types.
- Rely on **dynamic dispatch** instead of conditional logic (`if/else`, `switch`).
- Protect internal state with strict **encapsulation**.
- Organize code into a cohesive, non-trivial class structure.
- Manage and document your work professionally using **Git** and a **README.md**.

The Arena: A Simplified Game of Crazy 8's



Players: 1 Human vs. 1 Computer

Deck: Standard 52-card deck

Deal: 7 cards to each player

Objective: Be the first player to empty your hand.

The Rules of Engagement

On Your Turn

- Play **one card** that matches the top discard's **rank** OR **suit**.

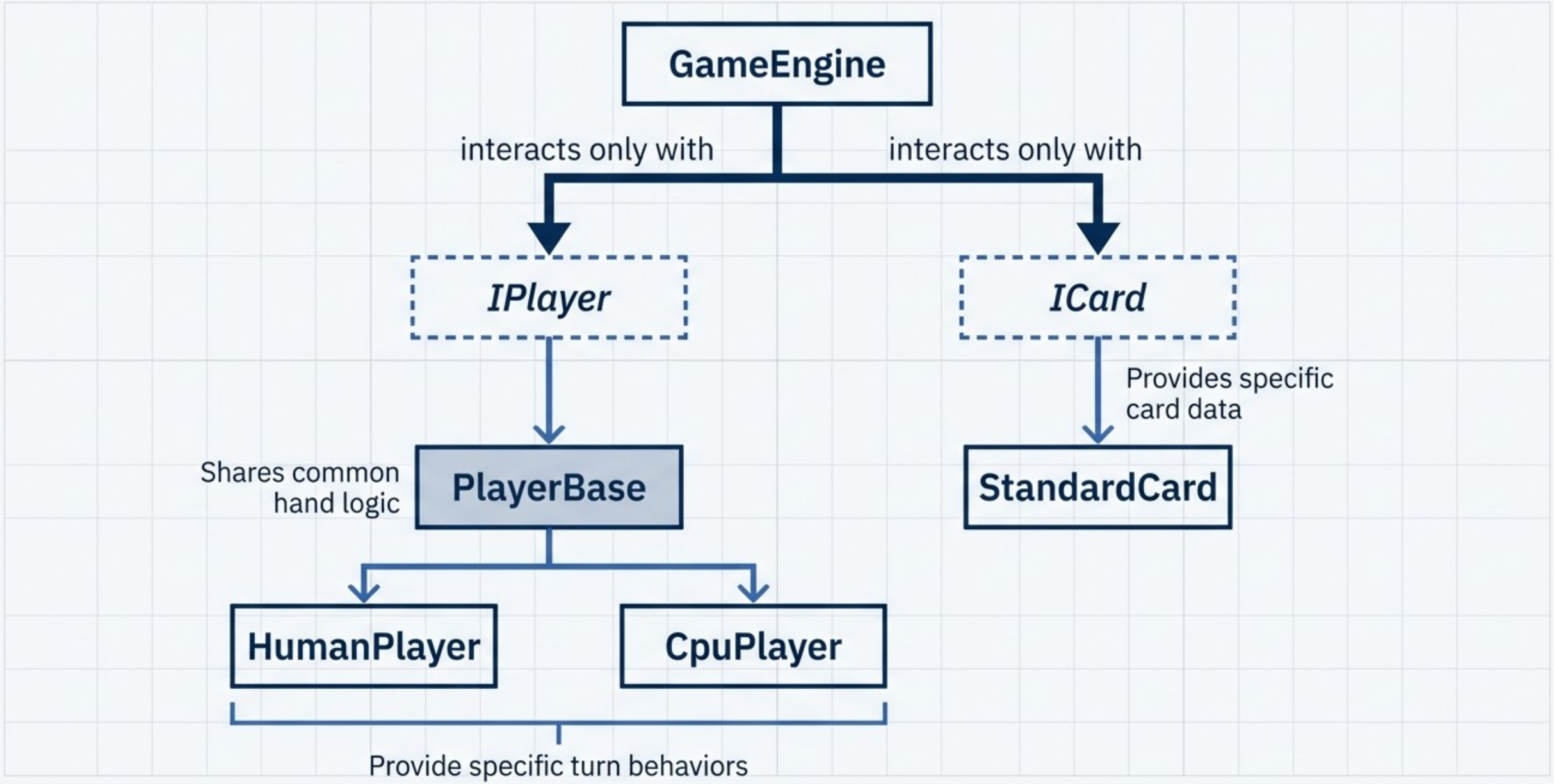
- **Eights are Wild**: An `8` can be played at any time. The player then declares a new suit for the next player to match.

If You Cannot Play

- Draw **one card** from the deck.
- If the new card is playable, you may play it immediately.
- Otherwise, your turn ends.

Please do not implement additional rules (e.g., stacking, draw-twos, skips). The focus is on the design, not game features.

The Core Blueprint: Depend on Abstractions, Not Concretions



Designing with Contracts: The Rules for Interfaces

The game engine must interact with cards and players **only through these interfaces**.

- ☑ **Minimalist:** Include only what the game engine truly needs.
- ☑ **Behavior-focused:** Define *what* an object can do, not *how*.
- ☑ **No UI Logic:** No printing or input handling.
- ☑ **No State Exposure:** No public fields or mutable collections.
- ☑ **Enable Polymorphism:** The engine must never need to know the concrete type (`is`, `instanceof`).

The Payoff: Polymorphism & Dynamic Dispatch

The game engine iterates over a **List<IPlayer>**, treating all players identically. The specific turn logic is dispatched to the correct object at runtime.

DO THIS

```
// Engine doesn't know or care who is playing.
currentPlayer.TakeTurn(context);
```

The concrete object (HumanPlayer or CpuPlayer) provides its own implementation. This is dynamic dispatch.

DO NOT DO THIS

```
if (player is HumanPlayer) {
    // Human-specific logic here...
} else if (player is CpuPlayer) {
    // CPU-specific logic here...
}
```

This defeats the purpose of polymorphism and creates brittle, hard-to-maintain code.

Protecting Your State: The Rules of Encapsulation

If another object can directly mutate your internal collection, your class does **not** own its state.

A Player's Hand

INCORRECT (Violates Encapsulation)

```
public List<ICard> Hand;
```

Allows any other class to freely add or remove cards, bypassing game rules.

CORRECT (Encapsulated)

```
private readonly List<ICard> _hand;
public void Draw(ICard card) { ... }
public IReadOnlyList<ICard> PeekHand()
{ ... }
```

The `Player` class fully controls its state. Other objects can observe (via `PeekHand`), but not mutate.

The Composition Root: Your 'Short and Boring' Main

Think of `Main` as the switch that turns the game on.
It has one job: **build the object graph** and **start the application**.

✔ What `Main` SHOULD do

- ✔ Create top-level objects (players, deck, game engine).
- ✔ 'Wire' them together (pass dependencies).
- ✔ Start the game (e.g., `game.Run()`).

✘ What `Main` MUST NOT do

- ✘ Contain game rules or turn logic.
- ✘ Use large conditionals on types.
- ✘ Be more than ~20-30 lines long.

Example: Proper `Main` in C#

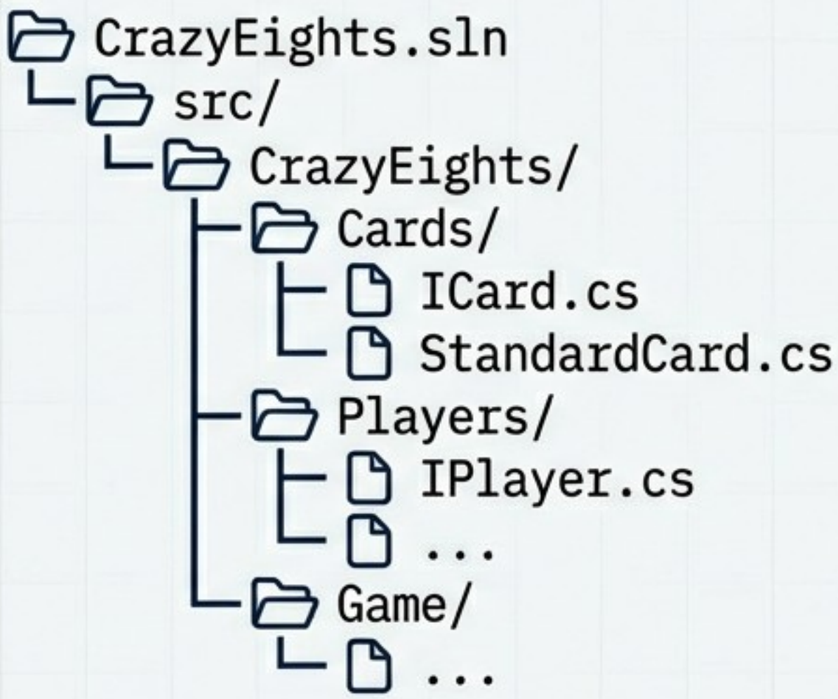
```
IPlayer human = new HumanPlayer("You");
IPlayer cpu = new CpuPlayer("CPU");
Deck deck = Deck.CreateStandardShuffled();

CrazyEightsGame game = new CrazyEightsGame(deck, human, cpu);
game.Run();
```

Project Constraints: The Forbidden Patterns

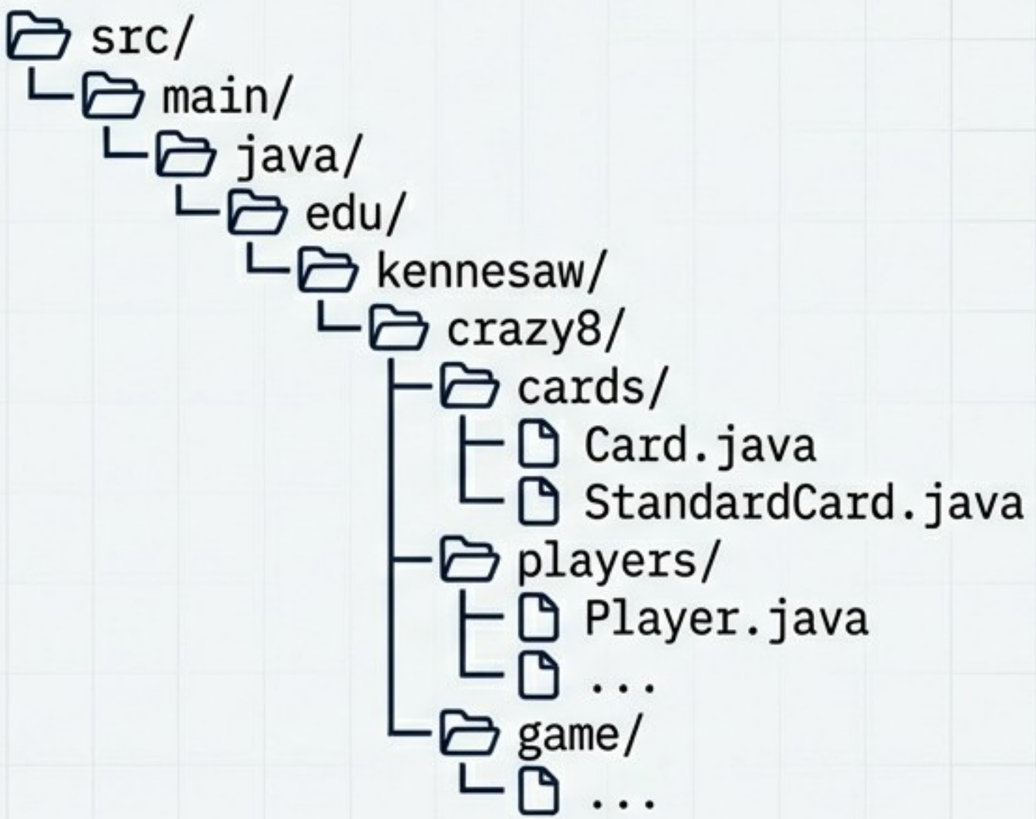
- 🚫 Use large `switch` statements on card rank or suit.
- 🚫 Use `instanceof` or `is` checks to control behavior.
- 🚫 Put game logic in `Main`.
- 🚫 Create “God Classes" or "God Methods."
- 🚫 Place more than one class in a single file.
- 🚫 Use any external libraries or frameworks.
- 🚫 Add unnecessary game features.

Organizing Your Blueprint: Required Project Structure

C# Structure



Java Structure



Key Mandates:

- Each class must be in its own file.
- Use logical namespaces (C#) or packages (Java). You **must not** use the default package in Java.

The Deliverable: A Professional README.md

Your repository is incomplete without a clear, professional `README.md`. It must contain:

1. **Project Description:** Briefly explain the game and the OO concepts you demonstrated.
2. **How to Run (from the console):** Provide exact command-line instructions. **Not from an IDE.**

   Example C#:

   ```
   dotnet run
   ```

   Example Java:

   ```
   javac *.java
   java edu.kennesaw.crazy8.Main
   ```

3. **Screenshot**: Include at least one inline screenshot of the running application using Markdown syntax:

   ```
   ![Game Screenshot](screenshot.png)
   ```

A Simple, Well-Designed Solution is the Goal

“You will be evaluated on how you design, not how clever the game is.”

If your solution is easy to read, easy to extend, and clearly object-oriented, you are doing it right.

Good engineers balance sound design with practical decisions to build and ship working software. This is your first step.

Submitting for Feedback

The goal is learning and feedback, not points. A detailed review is available upon request.

1. Complete the assignment and push your work to a public GitHub repository.
2. Email the repository URL to JAdkiss1@Kennesaw.edu.

Pre-Flight Checklist (Crucial)

Before requesting a review, ensure that:

- ✓ Your code compiles and runs.
- ✓ Your `README.md` includes clear console execution instructions.

Reviews will not be provided for code in zip files. I will not run your application from an IDE.