# Design Feedback Response and Object-Oriented Design Patterns Analysis

## Actions Undertaken from the Design Feedback

The cards have been completely overhauled. An ICard interface has been introduced, with a standard card implementation that matches the existing functionality. Additionally, there are decorators available that can be added on top of the cards to adjust their strengths and weaknesses. This design aligns with the solution suggested in my design feedback presentation.

### Consolidation in PlayerCards Class

The `PlayerCards` class now encapsulates all the cards a player possesses. This class maintains a list of stacks that reference the cards, allowing all cards owned by a player to be quickly and easily searched within a unified container.

The methods required to present the game to the user are abstracted through the `Display` class, which contains abstract methods suitable for various types of output. The current implementation enhances the previous console-based methods. Nonetheless, it can be effortlessly extended to support other display types, such as WPF or web-based interfaces, for presenting the game to users.

In the main startup class, an instance of the `Display` class is created and dependency-injected into the game instance, allowing the game to invoke display methods as needed.

### Code Reusability Enhancement

To avoid code repetition, "shuffleable" and "sortable" stacks are defined as types of stack with only the extra methods added to the class.

### Single Responsibility Principle

The class responsible for controlling the overall game has had the code for displaying the game removed from it. Additionally, the software has been reorganised to make it more readable.

# Completeness of the Solution

**Positive Aspects:**

The solution is streamlined and avoids code repetition, adhering to robust object-oriented principles.

**Negative Aspects:**

However, the number of design patterns implemented in the PocketBeast project is insufficient. Specifically, the factory pattern should have been employed in the card creation process. This would have mitigated the significant responsibility currently placed on the game setup. Additionally, there have been no extra enhancements to the software. Given more time, I would have created a WPF game display class along with some simple UI.

# Advantages and Limitations of Object-Oriented Design Patterns

## Advantages of Object-Oriented Design Patterns

**Enhanced Code Reusability:**

One of the most significant advantages of object-oriented design patterns is their ability to promote code reusability. By encapsulating commonly used solutions in reusable patterns, developers can avoid writing repetitive code, thereby reducing redundancy. These design patterns provide well-established solutions for common problems, which can be adapted and reused across various projects. This not only accelerates the development process but also results in more maintainable and scalable codebases.

**Improved Communication Among Developers:**

Design patterns serve as a common language among developers, facilitating better communication and understanding. When a pattern is named and documented, it provides a shorthand for describing complex architectural concepts. For instance, mentioning the "Observer Pattern" immediately conveys a particular design and behaviour to all proficient developers. This shared vocabulary helps in streamlining discussions, making it easier for teams to collaborate, brainstorm, and resolve issues efficiently, which is particularly valuable in large teams or cross-functional collaborations.

**Increased Code Robustness and Reliability:**

Utilising well-defined design patterns can lead to more robust and reliable code. These patterns often incorporate best practices and have been tested extensively in various scenarios, reducing the likelihood of encountering unforeseen bugs or issues. By adhering to these tried-and-true patterns, developers can achieve a more stable and predictable codebase. Moreover, many design patterns encourage modularity, which makes it easier to isolate and fix bugs without affecting other parts of the system, thereby enhancing the overall reliability and stability of the software application.

## Limitations of Object-Oriented Design Patterns

**Complexity for Beginners:**

One notable limitation of object-oriented design patterns is that they can be difficult for beginners to grasp. These patterns often involve abstract concepts and require a strong understanding of object-oriented principles, making them somewhat inaccessible to novice programmers. The complexity can lead to confusion and misuse, resulting in code that is harder to understand and maintain. This steep learning curve may deter new developers from effectively implementing design patterns, potentially negating their benefits.

**Overhead and Boilerplate Code:**

While design patterns facilitate code reuse and scalability, they can also introduce additional overhead and boilerplate code. Some design patterns necessitate the creation of numerous classes and interfaces to achieve the desired level of abstraction and decoupling. This added complexity can lead to a more cumbersome and bloated codebase, which might be overkill for simpler projects or smaller codebases. In such cases, the additional effort to implement and maintain these patterns may not justify the benefits they provide, leading to inefficiencies.

**Risk of Misapplication:**

Another significant limitation is the risk of misapplication of design patterns. Developers might be tempted to apply design patterns in situations where they are not necessary or appropriate, simply because they are familiar with a particular pattern. This can lead to suboptimal solutions that are over-engineered and difficult to maintain. Misusing design patterns can result in increased complexity and decreased performance, counteracting the advantages they are supposed to provide. It’s crucial to thoroughly understand the problem at hand and carefully consider whether a design pattern is the right fit before implementation.