Design Patterns in TypeScript Project - Explanation and Guide

# 1. Project Overview

This project demonstrates six different use cases of common software design patterns in TypeScript using Node.js. The goal is to implement creational, structural, and behavioral design patterns effectively while adhering to best coding practices, such as logging, exception handling, and performance optimization.

# 2. Technologies Used

- TypeScript: A statically typed superset of JavaScript for writing scalable and maintainable code.  
- Node.js: A JavaScript runtime used for building the application backend.  
- ts-node: Used to run TypeScript directly without needing to compile the code manually.  
- Other libraries: `prompt-sync` for user input in the terminal.

# 3. Installation Guide

Follow these steps to set up the project:

1. Clone the GitHub repository:

```bash  
git clone <your-repository-link>  
```

2. Navigate to the project directory:

```bash  
cd <project-directory>  
```

3. Install the dependencies using npm:

```bash  
npm install  
```

4. Run the application in development mode:

```bash  
npm run dev  
```

5. For production, build and run the app:

```bash  
npm run build  
node dist/index.js  
```

# 4. Project Structure Explanation

The project is structured as follows:

```  
src/  
 └── design-patterns/  
 └── behavioral/  
 └── observer-pattern.ts  
 └── command-pattern.ts  
 └── creational/  
 └── singleton-pattern.ts  
 └── factory-pattern.ts  
 └── structural/  
 └── adapter-pattern.ts  
 └── decorator-pattern.ts  
 └── services/  
 └── logging-service.ts  
 └── models/  
 └── index.ts  
```

Each design pattern is implemented in its respective folder based on its category (behavioral, creational, or structural). The `services/` folder contains utility services like logging, and the `index.ts` serves as the main entry point for running the application.

# 5. Detailed Explanation of Each Design Pattern

## 5.1 Singleton Pattern

The Singleton pattern ensures that a class has only one instance while providing a global access point. This is useful in scenarios like configuration management, where you want a single instance managing settings.  
In this project, the Singleton is demonstrated with the `ConfigurationManager` class, which stores and retrieves application settings.

## 5.2 Observer Pattern

The Observer pattern defines a one-to-many relationship between objects. When the state of one object changes, its dependents (observers) are notified automatically.  
In the project, the Observer pattern is used to simulate a subscription system where multiple observers react to state changes.

## 5.3 Factory Pattern

The Factory pattern abstracts the process of creating objects. Instead of instantiating objects directly, a factory method is responsible for creating the object based on given input.  
In the project, the Factory pattern is used to create different types of `User` objects depending on their role (e.g., Admin, Guest).

## 5.4 Adapter Pattern

The Adapter pattern allows incompatible interfaces to work together. It acts as a bridge between two incompatible systems.  
In the project, the Adapter pattern is used to integrate a third-party API with the internal system by adapting its interface.

## 5.5 Decorator Pattern

The Decorator pattern dynamically adds responsibilities to objects. This is useful for enhancing functionality without altering the base class.  
In the project, the Decorator pattern is used to add logging functionality to objects dynamically.

## 5.6 Command Pattern

The Command pattern encapsulates a request as an object, allowing for parameterization and queuing of requests.  
In this project, the Command pattern is demonstrated by encapsulating user actions into command objects.

# 6. Logging, Exception Handling, and Input Handling

Logging is handled through the `Logger` class, which provides methods for logging different levels of information (INFO, ERROR). The system ensures that all important events and errors are logged.  
Exception handling is implemented using try-catch blocks across the code to ensure that the application fails gracefully without unexpected crashes.  
Input from users is gathered dynamically using `prompt-sync`, and no infinite loops with hardcoded flags are used.

# 7. Running the Application

To run the application, you can use the following commands:  
- Development mode: `npm run dev`  
- Production mode: `npm run build` followed by `node dist/index.js`  
The application will prompt the user to input commands and interact with the design patterns implemented.

# 8. Key Decisions Taken

The primary focus of this project was to demonstrate the correct implementation of design patterns while ensuring the following:  
- \*\*Separation of Concerns\*\*: Each design pattern is implemented in a separate module to enhance readability and maintainability.  
- \*\*Defensive Programming\*\*: Input validation, exception handling, and logging are implemented to make the application robust.  
- \*\*Best Practices\*\*: Code follows global best practices, including modularity, consistent naming conventions, and optimized performance.