Design Patterns

Decorator

The decorator pattern is a structural design pattern. It allows behaviour to be added dynamically to objects without modifying the original code. This makes it especially suitable for scenarios where flexibility and extensibility are important, for example styling.

- Wrapping Objects: To wrap an object with another to modify how the system interacts with it or how it behaves.
- Object Composition: Instead of creating new types of objects, decorators give us the ability to change behaviour by grouping objects together.
- Dynamic Behaviour Modification: : Change how objects behave while the program is running
- Maintaining System Scalability: The approach keeps code modular, making it easier to extend and maintain without introducing unnecessary complexity.

As was hinted at before, the decorator pattern is well suited for handling the styling. By wrapping the objects with decorators, additional styling properties are able to be applied dynamically without modifying the original/core implementation.

Comparison with Alternatives:

Unlike Inheritance, the decorator pattern allows styling features to be added without changing the existing code, making it more modular and invites changeability into the system. Additionally, comparable to the composite pattern, decorator is more suitable when functionality needs to be layered rather than structured hierarchically.

Why decorator fits

We chose Decorator over inheritance to avoid a rigid hierarchy of style classes. This lets us combine font, colour, indent, and other styles flexibly and independently at runtime.

Command

The command pattern is a design pattern that turns commands/actions into objects. This allows us to handle different user commands without directly tying them to specific parts of the program. In JabberPoint, the command pattern would be used to handle the following: Next, Previous, Exit, New, Save, GoTo, Help and Open.
Because each action is wrapped inside a command object, it becomes easier for us the manage and expand the system. Due to Loose Coupling, the system doesn't need the specifics on how each command works, making it more easy to add, change or delete commands later. The only downside is that the code can become more complicated since we introduce extra layers between the action sender and the action receiver.

Comparison with Alternatives

The strategy pattern encapsulates algorithms, whereas the command pattern encapsulates actions. As for the Observer pattern, it could handle event-based execution but lacks the encapsulation and reusability that the Command pattern provides.

Why Command fits

We chose Command for its ability to queue, log, and undo actions if needed. It cleanly separates the triggering of an action (like a key press) from the logic of the action itself.

Builder

The builder pattern helps create complex objects step by step. The builder pattern lets us construct objects in a clear and flexible way without making a huge constructor with many parameters.
In our project, the Builder Pattern will be used to create slides. It will act as an interface for different builders:

- AnimationBuilder – Handles animations in slides.
- BaseLineBuilder – Manages basic slide content.
- VideoBuilder – Adds video elements to slides.

A Director will oversee the process, ensuring slides are built correctly with all necessary components.
Benefits that will be gained when using builder:

- Step-by-Step Object Creation – We build slides piece by piece instead of handling everything at once.
- Reusable Code – The same builder logic can be used to create different types of slides.
- Cleaner Code – The complex logic of building a slide is separated from the rest of the program, making it easier to manage.

By using the Builder Pattern, we make slide creation more structured, flexible, and easier to expand in the future.

Comparison with Alternatives

Factory creates single step objects, Builder pattern handles complex, step-by-step construction. Additionally, Abstract Factory could manage families of builders but would overcomplicate our use case.

Why Builder Fits

Slide content can vary greatly (text, images, future animation), so we needed step by step flexibility. Builder allows us to separate construction logic from presentation logic entirely.