Design Alternatives and Their Pros & Cons

Many approaches are explored to build the system modular, scalable, and maintainable, when the system is designed. Below are three different design patterns, compared to the chosen Controller Pattern, and their pros and cons.

1. Mediator Pattern

- Mediator Pattern centralizes communication between components through mediator (Mediator Pattern uses: GameManager as an example). They don’t communicate directly, sending requests to the GameManager, who coordinates.

Pros:

- Low Coupling: The dependencies between components are reduced to only interacting with the mediator.
- Encapsulation of Interactions: Interaction management logic (e.g. hero attacks a monster) are narrowed down to the GameManager which makes it easy to modify or extend.
- Flexibility: Interactions are centralized, and adding new components or changing existing ones is trivial.

Cons:

- Coordination Overhead: As the system gets bigger, you add complexity on the mediator side as you have many interactions taking place.
- Risk of Bottleneck: If the mediator has to do high frequency communication, it might slow down the system.

Comparison to Controller Pattern:

- Similar benefits come from the Controller Pattern but in a more explicit manner, delegating responsibilities explicitly to more specialized managers (e.g. MonsterManager, EnchantmentManager) to reduce the risk of a bottleneck.
- As opposed to the Mediator Pattern the Controller Pattern deals with system management rather than dealing with individual interactions.

2. Observer Pattern

Components in this design communicate with each other indirectly through the events. If we take a more specific example, the Hero will initiate a "MonsterDefeated" event and the MonsterManager itself will be listening for this event to do something.

Pros:

- High Decoupling: Components don’t depend on each other and don’t call each other directly; instead, they coordinate their actions through events.

- Real-Time Responsiveness: This is where event driven systems are useful – This means handling dynamic interactions like updating the UI when a monster is defeated.
- Extensibility: Adding new features is trivial: new events can be introduced, along with new listeners, on existing components without change.

Cons:

- Debugging Complexity: The tracing of flow of events and identifying the source of issues can be difficult.
- Performance Concerns: In real time systems it can add latency due to the fact you're handling so many events.
- Setup Overhead: The events need an infrastructure to be managed that brings in complexity to development.

Comparison to Controller Pattern:

The Controller Pattern keeps things nice and clean whereas the Event Driven Architecture can go mad because with no central point of control there will be no order.
In use of the Controller Pattern it's simpler to debug and manage since responsibilities are explicitly defined inside the GameManager.

3. Layered Architecture

This is approaches in which system organized into layers (e.g. presentation, logic and data). Each layer interacts with its immediate neighbors only, and these neighbors belong to the same layer, and thus we maintain separation of concerns.

Pros:

- Separation of Concerns: Maintainability is broken into layer, each layer focuses on responsibility to improve maintainability.
- Scalability: The system is easier to extend as layers are independently modifiable or replaceable.
- Testing: It can be tested in layers, reducing scope of errors per layer.

Cons:

- Increased Complexity: The strict boundaries between the layers make the system very complex to implement.
- Performance Overhead: Latency in real time systems adds up from the communication between layers.
- Rigid Structure: Dynamic interactions are more common in games, and are less adaptable to layered designs.

Comparison to Controller Pattern:

Compared to the Controller pattern, it's more flexible because it centralizes system level operations without strict time layers.

The complexity in Layered Architecture will add complexity that wouldn't be necessary for a comparatively small scale game.

Pros:

- Centralized Control: The GameManager has responsibility for coordinating system level operation while keeping system state management consistent.
- Simplified Communication: Hero and MonsterManager interact with GameManager to reduce interclass dependencies allowing for clean Simplicity.
- Delegation: The GameManager delegates some tasks to managers (e.g. MonsterManager and EnchantmentManager) so it doesn't become a God Object.

Cons:

- Potential Overloading: If the GameManager isn't properly modularized it will end up having too many responsibilities.
- Single Point of Failure: If there are any problems with the GameManager, the entire system will break.
- Moderate Coupling: By decoupling components from each other, we still have a central dependency to the GameManager.