Gemini

¡Entendido! Tienes razón, el ejemplo de la Misión es muy claro para el patrón State.

Aquí está el resumen teórico corregido, usando el ejemplo de la `Mission` para el patrón State.

---

1. Patrón Composite (Composición) 🌳

¿Qué hace?

El patrón **Composite** te permite componer objetos en **estructuras de árbol** para representar jerarquías de "parte-todo". Lo más importante es que permite a los clientes (el código que usa la estructura) tratar a los **objetos individuales** (hojas) y a las **composiciones de objetos** (grupos) de manera **uniforme**.

El cliente no necesita saber si está hablando con un solo objeto o con un grupo entero de ellos.

Ejemplo que vimos: El Inventario

- **La Interfaz Común (`BaseItem`):** Era la "idea" abstracta de un ítem. Definía la operación común `getValue()`.
- **La Hoja (`Item`):** Era el objeto individual, el final del árbol (ej: `blade`, `coins`). Su `getValue()` simplemente devolvía su propio valor.
- **El Compuesto (`Bag`):** Era el grupo (ej: `inventory`, `bagCoins`). Su `getValue()` era más complejo: sumaba su propio valor (si tuviera) y luego **recursivamente** llamaba a `getValue()` en todos sus "hijos" (los `BaseItem` dentro de su lista).
- **Resultado:** Tu `main` solo tuvo que llamar a `inventory.getValue()` una vez. No le importó que "inventory" fuera una bolsa llena de otras bolsas e ítems; el patrón se encargó de sumar todo automáticamente.

---

2. Patrón State (Estado) 🤷

¿Qué hace?

El patrón **State** permite a un objeto alterar su **comportamiento** cuando su **estado interno** cambia. Visto desde fuera, parece que el objeto cambia de clase.

Su objetivo principal es evitar código desordenado con sentencias `if-else` o `switch` gigantescas. En lugar de eso, encapsula cada estado (y su comportamiento) en su propia clase.

Ejemplo que vimos: La Misión

- **El Contexto (`Mission`):** Era el objeto que cambiaba de comportamiento. No sabía qué estado venía después, simplemente tenía una referencia a su estado actual.
- **La Interfaz de Estado (`MissionState`):** Definía el método que cambiaba (ej: `next()`).
- **Los Estados Concretos (`Blocked`, `Unblocked`, `Progress`, `Failed`, `Completed`):** Eran las clases que implementaban la lógica real de la transición.
- **Resultado:** Cuando tu `main` llamaba a `mission.next()`, la clase `Mission` no tenía un `if` o `switch`. Simplemente delegaba la llamada a su estado actual (ej: `Blocked`). Era la clase `Blocked` la que tenía la lógica de cambiar el estado de la misión a `Unblocked` (llamando a `mission.setState(new Unblocked())`). El estado `Progress` incluso tenía lógica condicional (podía pasar a `Completed` o `Failed`), pero toda esa complejidad estaba fuera de la clase `Mission`, haciendo que todo fuera limpio y fácil de extender.

---

3. Patrón Observer (Observador) 📣

¿Qué hace?

El patrón **Observer** define una dependencia de **uno-a-muchos** (como un "publicador" y sus "suscriptores"). Cuando el objeto "publicador" (el Sujeto) cambia de estado, todos sus suscriptores (los Observadores) son notificados y actualizados automáticamente.

Esto crea un **bajo acoplamiento**: el publicador no sabe quiénes son sus suscriptores, solo sabe que tiene una lista de ellos y que debe llamarles al método `update()`.

Ejemplo que vimos: El Vigilante y los Enemigos

- **El Sujeto/Observable (`Vigilant`):** Era el "publicador". Tenía una lista de `MyObserver` y los métodos `subscribe()` y `unsubscribe()`.
- **El Observador (`Mage`, `Warrior`):** Eran los "suscriptores". Implementaban la interfaz `MyObserver` y su método `update()`.
- **Resultado:** Tu `main` "suscribió" a los magos y guerreros al vigilante. Cuando se llamó a `vigilant.activate()`, éste no supo que existían "magos" o "guerreros". Simplemente recorrió su lista y llamó a `update()` en cada `MyObserver`. Esto hizo que cada mago y guerrero reaccionara (activándose y atacando) sin que el `Vigilant` estuviera fuertemente acoplado a ellos.

(Es la misma lógica de la analogía del profesor que sube notas al sistema y el sistema avisa a los alumnos suscritos).

---

4. Patrón Adapter (Adaptador) 🔌

¿Qué hace?

El patrón **Adapter** actúa como un **traductor** o "enchufe" intermediario. Permite que dos interfaces **incompatibles** trabajen juntas.

Se usa para hacer que una clase existente (que no puedes o no quieres modificar) "encaje" en una interfaz que tu sistema espera.

Ejemplo que vimos: El Zombie

- **La Interfaz Objetivo (`Enemy`):** Era lo que tu sistema esperaba. Todas tus clases de enemigo (como `Mage` y `Warrior`) usaban los métodos `attack()` y `damage()`.
- **La Clase Incompatible (`Zombie`):** Era la clase "vieja" o "externa". Tenía la funcionalidad, pero con nombres diferentes (`bite()`, `hit()`, `headDamage()`).
- **El Adaptador (`ZombieAdapter`):** Esta fue la clase "traductora". Por fuera, **implementaba** `Enemy` (para que tu sistema estuviera feliz). Por dentro, **contenía una instancia de** `Zombie`.
- **Resultado:** Cuando tu `main` llamó a `zombieAdapter.attack()`, el adaptador recibió esa llamada y la "tradujo" a lo que el `Zombie` entendía, llamando a `zombie.bite()` o `zombie.hit()`. Esto te permitió integrar al `Zombie` en tu sistema sin tener que modificar la clase `Zombie` original.