Padrões de Software – SkyControl

1. **Arquitetura Orientada a Eventos (Event-Driven Architecture - EDA)**

Aplicação no Projeto: A comunicação entre o **Simulador de Drones** e a **Central de Controle (Backend)** é inteiramente assíncrona, mediada pelo *Message Broker* RabbitMQ.

- **Produtor (Producer):** A classe SimulatedDrone atua como um produtor de eventos. A cada ciclo de 3 segundos, ela gera um objeto DroneTelemetry e o serializa para JSON, enviando-o para a exchange telemetry.exchange. O drone não "sabe" quem receberá essa mensagem, ele apenas a publica.
- **Consumidor (Consumer):** No backend, a classe TelemetryListener utiliza a anotação @RabbitListener para escutar a fila telemetry.queue. Assim que uma mensagem chega, o método handleTelemetryMessage é acionado automaticamente.
- **Benefício Prático:** Se o backend ficar indisponível (offline) para manutenção, os drones continuam voando e publicando dados (que ficam retidos na fila), garantindo que a simulação nunca trave por falha no servidor.

2. **Padrão Observer (Publish-Subscribe Interno)**

Aplicação no Projeto: Além do RabbitMQ (que é um Pub/Sub externo), o backend implementa um barramento de eventos interno para atualizar a interface do usuário em tempo real.

- **O Sujeito (Subject):** A classe InternalEventBus gerencia uma lista de "ouvintes" (listeners). Quando a telemetria chega do RabbitMQ, o TelemetryListener publica um evento TELEMETRY_UPDATED neste barramento.
- **O Observador (Observer):** O EventStreamController se registra neste barramento. Ao receber a notificação de nova telemetria, ele "empurra" esses dados para o navegador do usuário via **Server-Sent Events (SSE)**.
- **Benefício Prático:** Elimina a necessidade do Frontend ficar perguntando ao servidor "tem dados novos?" a todo segundo (polling), economizando recursos de rede e processamento.

3. **Padrão State (Máquina de Estados)**

Aplicação no Projeto: A lógica de voo de cada drone é governada por uma máquina de estados finita, implementada na classe SimulatedDrone.

- **Estados Definidos:** O enum State { IDLE, MOVING, PAUSED } define os modos de operação.
- **Comportamento Dinâmico:**
    - No estado **MOVING**, o método run() executa a lógica vetorial updateMovementToTarget() para deslocar a latitude/longitude.

    - No estado **PAUSED**, a thread entra em suspensão (wait()) via pauseLock, parando qualquer consumo de processamento de movimento até receber um comando de retomada.
    - No estado **IDLE**, o drone apenas consome bateria e flutua (varia altitude), sem alterar sua posição geográfica.
- **Benefício Prático:** Organiza a lógica complexa de simulação. Por exemplo, quando ocorre uma "Emergência" (checkEmergencyAlert), o drone transita forçadamente para PAUSED, garantindo que ele pare de voar imediatamente sem a necessidade de condicionais complexas (if/else) espalhadas por todo o código.

4. **Padrão MVC (Model-View-Controller)**

Aplicação no Projeto: Utilizado na estruturação do projeto Spring Boot (dronebackend) para expor a API REST.

- **Controller:** Classes como FormationController recebem as requisições HTTP do Frontend (ex: "Formar Linha"). Elas não contêm regra de negócio, apenas orquestram a chamada.
- **Service (Camada de Negócio):** O FormationService contém a "inteligência". Ele calcula as coordenadas de cada drone para formar a linha ou coluna e decide quais comandos enviar.
- **Model:** Classes como Drone e DroneTelemetry são POJOs (Plain Old Java Objects) puros que representam os dados sem comportamento atrelado.
- **Benefício Prático:** Facilita a manutenção. Se quisermos mudar a regra matemática da "Formação em V", alteramos apenas o FormationService, sem risco de quebrar a API (Controller) ou a estrutura de dados (Model).

5. **Padrão Singleton (Gerenciado pelo Spring)**

Aplicação no Projeto: Todas as classes anotadas com @Service, @Controller e @Component (como JsonDatabaseService) são instanciadas apenas uma vez pelo container do Spring Framework.

- **Benefício Prático:** Eficiência de memória. Não é necessário criar uma nova conexão com o banco de dados (arquivo JSON) ou um novo ouvinte RabbitMQ para cada requisição. Uma única instância processa todas as chamadas de forma compartilhada.