He elegido el patrón **Observer** y aquí te explico las razones clave:

1. **Desacoplamiento entre sujeto y observadores**: El patrón **Observer** permite que la clase `Stock` no tenga que saber nada acerca de los detalles de los observadores (los diferentes tipos de clientes). Esto es crucial porque tenemos diferentes tipos de clientes que quieren recibir diferentes cantidades de datos de una acción
2. **Facilidad para añadir nuevos tipos de clientes**: Este patrón permite que los nuevos tipos de clientes se integren fácilmente sin modificar el código del sujeto. Esto reduce el impacto de los cambios y facilita la evolución del sistema.
3. **Actualización automática**: Cuando los datos de la acción cambian (precio de cierre, máximo, mínimo, etc.), el patrón **Observer** garantiza que todos los clientes suscritos (observadores) se actualicen automáticamente. No es necesario que cada cliente realice consultas periódicas al servidor para obtener la información actualizada.
4. **Flexibilidad en la suscripción**: Con este patrón, cada cliente puede suscribirse para recibir solo la información que le interesa. Esto es ideal para el caso que nos planteas, donde algunos clientes solo están interesados en el precio de cierre, otros en toda la información y algunos en una acción específica (por ejemplo, "AAPL").
5. **Escalabilidad y adaptabilidad**: Si en el futuro decidimos agregar más tipos de acciones o clientes, el patrón **Observer** lo facilita mucho.

PRINCIPIOS SOLID:

- **SRP (Responsabilidad Única)**: Cada clase tiene una única responsabilidad. `Stock` gestiona los datos de la acción, y los clientes (`SimpleClient`, `DetailedClient`, `FocusedClient`) solo se encargan de mostrar la información que necesitan.
- **OCP (Abierto/Cerrado)**: El sistema está diseñado de tal manera que se pueden agregar nuevos tipos de clientes sin modificar el código de la clase `Stock` ni los clientes existentes.
- **LSP (Sustitución de Liskov)**: Cualquier tipo de cliente que implemente la interfaz `Observer` puede ser utilizado de forma intercambiable en la clase `Stock`.
- **ISP (Segregación de Interfaces)**: Las interfaces son específicas y los clientes solo implementan las que realmente necesitan (es decir, `Observer` para recibir notificaciones).

- **DIP (Inversión de Dependencias)**: La clase `Stock` depende de la interfaz `Observer` y no de implementaciones concretas de los clientes, lo que promueve la flexibilidad y facilidad de extensión.