Pipes & Filters

Problema

Necesito implementar una serie de transformaciones (conversion, enrichment, filtering, batching, consolidation) que quiero poder combinar y reusar de forma independiente en distintos contextos.

Solución

Implementar cada transformación en un componente (filtro), que recibe un mensaje, realiza la transformación que le corresponde y retorna el mensaje resultante. Cada filtro se conecta con el siguiente a través de pipes, los cuales pueden además hacer buffering de mensajes. Un filtro no puede asumir nada acerca del contexto en el que se utiliza (por ej, con qué otros filtros se comunica).

Beneficios

- **Reuso** de transformaciones en distintos contextos.
- Reduce el costo del cambio. Si hay que cambiar algo, probablemente el cambio impacte a un solo componente de la cadena.
- Favorece la **testeabilidad**.
- Beneficia la **modificabilidad**. Permite **combinar transformaciones** de distintas maneras para generar distintas funcionalidades, incuso en forma dinámica.
- Permite la **distribución** de las transformaciones en distintos nodos con distintas capacidades computacionales. Por ej: puedo tener la transformación de formatos XML en un nodo dedicado, para que esa operación no sea un cuello de botella.

Dificultades

- Identificar las transformaciones que se requieren y encapsularlas en componentes puede ser una tarea compleja.
- Impacto en la **performance**, debido al overhead de las comunicaciones entre los filtros.
- Los filtros no saben en qué contexto están siendo utilizados y no pueden asumir nada del contexto. Esto dificulta el **manejo de errores**.
- Afecta la **mantenibilidad** (sistemas más difíciles de mantener).

Consideraciones

- Cuanto más granulares sean las transformaciones, más fácil será probar cada transformación y reusarla en otro contexto. Por otro lado, si tengo demasiadas transformaciones se degrada la performance.
- Para poder conectar dos filtros, la salida de uno debe ser compatible con la entrada del otro. Es importante la correcta definición de interfaces.

Layers

Problema

Cómo asegurar que se cumplen los atributos de calidad de mantenibilidad, reusabilidad, escalabilidad, robustez y seguridad, si estoy desarrollando una aplicación con numerosos componentes ubicados en distintos niveles de abstracción.

Solución

Agrupo los componentes en distintas capas lógicas (layers), siguiendo algunas reglas:

- debe haber cohesión entre los componentes de una capa.
- los componentes de una capa deben pertenecer al mismo nivel de abstracción. A veces se tienen grupos de componentes cohesivos entre sí en un mismo nivel de abstracción; en ese caso se pueden crear varios subsistemas en esa capa de abstracción.
- cada capa puede acoplarse con capas inferiores (de más bajo nivel de abstracción).
- la comunicación entre capas se realiza a través de interfaces bien definidas (no acoplarse a implementaciones).
- se debe evitar que capas inferiores dependan de capas superiores. Las capas inferiores pueden comunicarse con las superiores a través de eventos, delegados y callbacks, lo cual brinda un nivel de indirección.

2 modelos de capas:

- Estricto: cada capa puede comunicarse sólo con la capa inmediatamente inferior.
- Relajado: cada capa puede comunicarse con cualquiera de las capas inferiores a ella y con la capa superior.

Manejo de errores:

- manejar errores en el nivel más bajo posible.
- si se debe lanzar una excepción de bajo nivel hacia una capa superior, hacer que esté encapsulada por una excepción de alto nivel que tenga algún significado para el invocador.

Beneficios

- beneficia la **modificabilidad**.
- minimiza el impacto de los cambios.
- favorece el **reuso**, ya que las capas inferiores no dependen de las superiores y por lo tanto pueden utilizarse en otros contextos.
- las capas inferiores pueden ser reemplazadas por otras con bajo impacto (idealmente nulo). Esto facilita el **testing** y favorece la modificabilidad.

Dificultades

- el pasaje entre varias capas para realizar una operación tiene impacto sobre la **performance**. Esto se soluciona usando un modelo relajado, pero al hacerlo se tienen otras dificultades, por ej. cambios en capas inferiores pueden tener impacto en varias capas superiores.
- el uso de capas puede agregar **complejidad** innecesaria en aplicaciones pequeñas.

Model-View-Controller

Problema

Cómo se puede modularizar el sistema teniendo en cuenta que:

- la UI es lo que más cambia (depende mucho de los dispositivos, se puede querer mostrar la misma información de distinta manera, se quieren hacer cambios para mejorar la usabilidad para un grupo de usuarios, etc). Conociendo este factor, se busca que los cambios en la UI tengan un costo mínimo y no impacten sobre otras partes del sistema (como la lógica de negocio) que son más estables. O sea, se busca favorecer la modificabilidad en este contexto.
- se pueden tener distintas vistas de la misma información, y los cambios en una vista deben reflejarse en las demás.
- diseñar una UI requiere distintas habilidades que diseñar lógica de negocio.
- generalmente, es más fácil generar pruebas automatizadas para la lógica de negocio que para la interfaz, por lo tanto si están mezcladas se ve afectada la testeabilidad.

Solución

La idea es separar la lógica de negocio, la presentación y las acciones basadas en input del usuario en 3 clases:

- Model: lógica de negocio.
- View: consulta el modelo y muestra la información. Cada vista tiene asociada un controlador.
- Controller: interpreta los inputs del usuario y actualiza el modelo y la(s) vista(s) en base a esos eventos.

Beneficios

- Múltiples vistas del mismo modelo, que se sincronizan cada vez que ocurre un cambio.
- Reemplazo dinámico de vistas.
- Un cambio en una o más vistas no afecta el modelo: **modificabilidad** y **testeabilidad**.

Dificultades

- **complejidad** (indirección, manejo de eventos)
- dificultades similares a las del patrón Observer: costo de los updates si se realizan con mucha frecuencia (**performance**), se puede estar actualizando información que no ha cambiado.

Publisher/Subscriber

Problema

Cómo lograr la integración de aplicaciones de modo de que una aplicación (publisher) envíe mensajes únicamente a las aplicaciones interesadas en esos mensajes (subscribers) sin conocer dichas aplicaciones.

Solución

Se extiende la infraestructura de comunicación y se provee un mecanismo en el cual los publicadores colocan los mensajes en la infraestructura y los subscriptores se suscriben a ciertos mensajes. Existen 2 posibilidades:

- crear tópicos. Cuando un publicador envía un mensaje, el mismo publicador o un intermediario le agrega información acerca del tópico al que pertenece. Los subscriptores se suscriben a los tópicos que les interesan. El mecanismo de envío y recepción puede variar:
  - Listas (similar a un Observer): el publicador (o un mediador) provee una interfaz para suscribirse a un tópico y mantiene una lista de los suscriptores. Los mensajes que pertenecen a ese tópico se envían a todos los subscriptores de la lista.
  - Broadcast: el mensaje se coloca en la red y cada subscriptor decide si procesarlo o no de acuerdo al tópico del mensaje.
- inspección dinámica de mensajes.

Beneficios

- bajo acoplamiento entre publicadores y subscriptores.
- seguridad.

Dificultades

- impacto en performance, en especial cuando se inspeccionan los paquetes.
- complejidad (identificación y mantenimiento de tópicos, implementación de un mecanismo de subscripción, uso de APIs especializadas).