Parte Teórica

Tomás Felipe Melli

July 2, 2025

Índice

1 2do parcial 2C2024

1.1 ¿Qué métrica relaciona el personaje del artículo "Pattern Abuser" con la aplicación de patrones? Explique la métrica y justifique si sirve o no.

> I used the number of patterns as a **reusability metric**, and assured success using this metric. I was able to guarantee during my period with the company that the number of patterns used in their code would grow dramatically.

Como vemos en esta línea, él consideraba que aumentando el uso de patrones podía hacer el código más reutilizable. La conclusión nos cuenta que si bien es reutilizable, no lo entiende nadie, hasta él mismo dice que no sabe ya qué hacer y quiere meter más patrones. Por tanto, la conclusión es que la métrica no es buena, porque al final, el código es tan general que nadie lo entiende, por tanto no lo pueden usar porque no saben cómo.

1.2 Diferencia entre las dos métricas propuestas en *Modern Software Engineering* para evaluar la efectividad de un equipo de software. Explique además las medidas que se rastrean para *Throughput*.

David Farley dice que `no mejoramos, porque no medimos correctamente`. Él reclama que la velocidad o la cantidad de líneas de código, no es una métrica que sirva. Propone dos diferentes :

1. **Stability**, que se mide de dos formas, con la **tasa de change failure** que es la métrica que nos dice a qué ritmo un cambio produce un defecto en un punto del proceso y el **tiempo de recuperación** , es decir, cunto le toma al equipo luego retomar el proceso desde ese fallo. La idea de esta métrica (`stability`) es dar una idea de calidad, qué tan efectivo es el equipo en *delivering software*.
2. **Throughput** se mide de dos formas, con el **lead time** que es el tiempo que toma de la idea a la línea de código funcional (working software) y la **frecuencia** de despliegue, cunto toma ya poner esa línea en producción. Esta métrica (`throughput`) es más bien de eficiencia y velocidad.

1.3 ¿Qué dos patrones o mecanismos subyacen en toda solución modelada con un *Visitor*? Explique brevemente en qué parte se utiliza cada uno.

Como bien se menciona en la **motivation** del `visitor`, cuando queremos recorrer una estructura que contiene elementos heterogéneos de manera uniforme, sin duplicar la lógica, necesitamos el `visitor`. Esta estructura per se, es una jerarquía que como vimos en el **composite** (componemos objetos en una estructura en forma de árbol para representar jerarquías) y por tanto subyace este patrón. A su vez, el `visitor` para su funcionamiento depende del **double-dispatch** ya que él con el mensaje `visit` visitará a cierto tipo de objeto heterogéneo de la jerarquía y luego, este último lo aceptara con `accept`.

2 2do recu 2C2024

2.1 David Farley dice que la ingeniería de software se diferencia de otras ingenierías ya que la producción en masa de nuestros diseños no resulta un gran problema. ¿Por qué dice esto?

David habla sobre la diferencia entre la ingeniería de producción y la de diseño. Hace hincapié en esto ya que para nosotros, programadores, producir el código es prácticamente gratis e instantáneo. Es hacer el *build* como dice. También habla sobre que nuestros problemas de producción no tienen los problemas como el puente con simulaciones para producir un modelo y ver qué onda, ya que nuestro modelo, es nuestro producto.

2.2 Decida si las siguientes afirmaciones son verdaderas o falsas. Justifique en ambos casos:

1. El "Handler" en el patrón de Object Recursion siempre actúa como un "Terminator" al manejar solicitudes.
2. El patrón de Object Recursion requiere que todos los objetos en la estructura sean del mismo tipo para implementar el manejo de solicitudes.

1. La respuesta es **falso**. El `handler` define el tipo de objetos que pueden manejar las solicitudes iniciadas por el cliente. Actúa como interfaz común para los objetos que participan en la recursión. Los responsables de la terminación son el `terminator` o el `recurser`(en algunos casos puede. El `recurser` es un tipo específico de Handler que mantiene una referencia a uno o más sucesores. Maneja la solicitud delegándola a sus sucesores, pudiendo ejecutar lógica adicional antes o después de dicha delegación).

2. También es **falso**. En este patrón como requerimiento mínimo e indispensable es que los objetos tengan una interfaz común para poder handlear la solicitud. Pero sólo eso.

2.3 Explique el dilema "Safety" vs "Transparency" del patrón Composite, y cuál es la postura de la materia en cuanto al mismo.

El dilema "Safety" vs "Transparency" del patrón Composite es el tradeoff de diseño que ocurre cuando tenemos dos objetos de naturaleza diferente pero muy parecidos, como una carpeta y un archivo de texto. Si armamos una jerarquía de la cuál ambos son hijos, vamos a querer definir el mensaje `agregar: unArchivo` pero el problema es que un archivo de texto no puede agregar un archivo ni una carpeta, como sí la carpeta. Tener todo el protocolo definido en esa clase abstracta que puede ser `directorio` lo hace transparente, ya que el cliente utiliza los componentes del sistema de manera uniforme. El problema es que podría ocurrir que un archivo de texto haga inapropiadamente algo con ese agregar. Esto introduce problemas de seguridad. En contrapartida, distinguirlos nos permitirá un entorno más seguro pero menos uniforme.