Técnico en

DESARROLLO DE SOFTWARE

Diseño e Implementación del Software

Instituto Von Neumann

Diseño e Implementación del Software

< Unidad I >

Principios de todo diseño de software

Existen ciertos principios que te ayudarán al momento de desarrollar algún sistema ya sea para la reducción de tiempo, costos o esfuerzo, dichos principios son lineamientos generales aplicados a la programación orientada a objetos.

Como con todo en la vida, utilizar estos principios de forma descuidada puede hacer más mal que bien. El costo de aplicar estos principios en la arquitectura de un programa es que puede hacerlo más complicado de lo que debería. Dudo que exista un producto de software exitoso en el que se apliquen todos estos principios al mismo tiempo. Aspirar a estos principios es bueno, pero intenta siempre ser pragmático y no tomes todo lo escrito aquí como un dogma.

Al momento de construir un buen diseño de software se deberán tomar en cuenta algunas características siendo las principales las siguientes:

- **Reutilización de código (Reducción de costos y tiempo)**

Al momento de reutilizar código existen 3 niveles de reutilización, los cuales enumeramos a continuación:

Nivel bajo: Reutilización de clases, bibliotecas de clases, contenedores.

Nivel medio: Patrones de diseño, los patrones de diseño son más pequeños y abstractos que los frameworks, en realidad, son una descripción sobre cómo pueden relacionarse un par de clases e interactuar entre sí. El nivel de reutilización aumenta cuando pasas de clases a patrones y por último a frameworks. Lo bueno de esta capa intermedia es qué, a menudo, los patrones ofrecen la reutilización de

un modo menos arriesgado que los frameworks ya que los patrones te permiten reutilizar ideas y conceptos de diseño con independencia del código concreto.

Nivel superior: A este nivel se intentan destilar las decisiones de diseño, identificar las abstracciones clave para resolver un problema, comúnmente a este nivel se representan con clases y definen relaciones entre ellas. Normalmente, vamos a encontrar a este nivel a un framework que es más tosco que una única clase ya que utilizan el llamado principio de Hollywood de "no nos llames, nosotros te llamamos a ti". Solo considera que crear un framework comprende un alto riesgo y una inversión considerable.

- **Extensibilidad ("El cambio es lo único constante en la vida de un programador")**

El concepto de extensibilidad hace referencia a las distintas situaciones donde es necesario seguir agregando nuevas funcionalidades a un sistema, tomando en cuenta siempre todas las características necesarias para evolucionar correctamente, con frecuencia se puede extender las funcionalidades cuando:

Comúnmente como desarrolladores comprendemos mejor el problema una vez que comenzamos a resolverlo.

Algo fuera de tu control ha cambiado (alguna tecnología, framework, práctica, por ejemplo, flash).

Los postes de la portería se mueven: "si alguien te pide cambiar algo de tu aplicación, eso significa que todavía le importa a alguien".

Por estos motivos, todos los desarrolladores experimentados se preparan para posibles cambios futuros cuando diseñan la arquitectura de una aplicación.

Principios básicos

- **Simple**

Este principio es muy importante ya que nos permitirá el poder garantizar el fácil entendimiento, efectividad y mantenimiento del diseño del sistema, esto puede contribuir a su permanencia en el negocio.

- **Mantenible**

Al momento de diseñar un nuevo sistema se deben de considerar diferentes aspectos para poder garantizar su continuidad en el negocio, uno de ellos es que debe ser posible darle mantenimiento para resguardar su confiabilidad y poder garantizar un buen rendimiento en cuanto a las necesidades del negocio, evitando fallas futuras, así como también prever toda situación de riesgo que pueda afectarle en algún momento futuro.

- **Debe poder probarse**

Todo software debe de estar diseñado para poder probarse, recordemos que algo que no se puede probar simplemente no sirve ya que no cuenta con las características necesarias para poder validar si su funcionamiento es correcto o no, esta es una característica importante ya que todo diseño debe de estar acompañado de su plan de pruebas.

- **Encapsula lo que varía**

Para aplicar este principio es necesario identificar los aspectos de tu aplicación que varían y sepáralos de los que se mantienen inalterables. Recordemos que el objetivo principal de este principio es minimizar el efecto provocado por los cambios.

Puedes aislar las partes del programa que varían, en módulos independientes, protegiendo el resto del código frente a efectos adversos. Al hacerlo, dedicarás menos tiempo a lograr que el programa vuelva a funcionar, implementando y probando los cambios. Cuanto menos tiempo dediques a realizar cambios, más tiempo tendrás para implementar funciones.

- **Encapsulación a nivel del método**

Digamos que estás creando un sitio web de comercio electrónico. En alguna parte de tu código, hay un método obtenerTotaldelPedido que calcula un total del pedido, impuestos incluidos.

Podemos anticipar que el código relacionado con los impuestos tendrá que cambiar en el futuro. La tasa impositiva dependerá de cada país, estado, o incluso ciudad en la que resida el cliente, y la fórmula puede variar a lo largo del tiempo con base a nuevas leyes o regulaciones. Esto hará que tengas que cambiar el método obtenerTotaldelPedido bastante a menudo. Pero incluso el nombre del método sugiere que no le importa cómo se calcula el impuesto.

Lo que sugiere la encapsulación a nivel del método es separar la lógica del total del pedido y la del cálculo de los impuestos en métodos distintos y que únicamente el obtenerTotalddelPedido convoque al método para calcular los impuestos y así se logra delegar una responsabilidad unica sin la necesidad de cambiar el funcionamiento de un método que no tiene relación directa con el otro.

- **Encapsulación a nivel de la clase**

Con el tiempo puedes añadir más y más responsabilidades a un método que solía hacer algo sencillo. Estos comportamientos añadidos suelen venir con sus propios campos y métodos de ayuda que acaba nublando la responsabilidad principal de la clase contenedora. Si se extrae todo a una nueva clase se puede conseguir mayor claridad y sencillez.

Programa a una interfaz, no a una implementación

Este principio depende de abstracciones, no de clases concretas.

Sabrás que el diseño es lo suficientemente flexible si puedes extenderlo con facilidad y sin descomponer el código existente. Vamos a asegurarnos de que esta afirmación es correcta viendo un nuevo ejemplo con gatos. Un Gato que puede comer cualquier comida es más flexible que uno que sólo puede comer salchichas. Al primer gato le puedes dar salchichas porque éstas son un subgrupo de “cualquier comida”, pero puedes extender el menú de ese gato con cualquier otra comida.

La forma más flexible de establecer la colaboración entre objetos es la siguiente:

- Determina lo que necesita exactamente un objeto del otro: ¿qué métodos ejecuta?
- Describe esos métodos en una nueva interfaz o clase abstracta.
- Haz que la clase que es una dependencia implemente esta interfaz.
- Ahora, haz la segunda clase dependiente de esta interfaz en lugar de la clase concreta. Todavía puedes hacerlo funcionar con objetos de la clase original, pero ahora la conexión es mucho más flexible.

Favorece la composición sobre la herencia

La herencia es probablemente la forma más obvia y sencilla de reutilizar código entre clases, por ejemplo: Consideremos que tienes dos clases con el mismo código. Creas una clase base común para estas dos clases y colocas dentro el código similar. ¡Pan comido! lamentablemente, la herencia tiene sus contras:

- ✔ **Una subclase no puede reducir la interfaz de la superclase:** Tienes que implementar todos los métodos abstractos de la clase padre, incluso aunque no vayas a usarlos.
- ✔ **Al sobrescribir métodos debes asegurarte de que el nuevo comportamiento sea compatible con el de la base.** Es importante porque los objetos de la subclase pueden pasarse a cualquier código que espere objetos de la superclase y no quieres que ese código se rompa.
- ✔ **La herencia rompe la encapsulación de la superclase** porque los detalles internos de la clase padre se hacen disponibles para la subclase.
- ✔ **Las subclases están fuertemente acopladas a superclases.** Cualquier cambio en una superclase puede descomponer la funcionalidad de las subclases.
- ✔ **Intentar reutilizar código mediante la herencia puede conducir a la creación de jerarquías de herencia paralelas.** Normalmente, la herencia sucede en una única dimensión. Pero cuando hay dos o más dimensiones, debes crear muchas combinaciones de clases, hinchando la jerarquía de clases hasta un tamaño ridículo.

Principios SOLID

SOLID es una regla mnemotécnica para cinco principios y de buenas prácticas de diseño ideados para hacer que los diseños de software sean más comprensibles, flexibles y fáciles de mantener siendo estos.

S – Principio de responsabilidad única (Single Responsability Principle)

Este principio establece que un componente o clase debe tener una responsabilidad única, sencilla y concreta. Esto simplifica el código al evitar que existan clases que cumplan con múltiples funciones, las cuales son difíciles de memorizar y muchas veces significan una pérdida de tiempo buscando qué parte del código hace qué función.

O – Principio abierto/cerrado (Open/Closed Principle)

Este principio establece que los componentes del software deben estar abiertos para extender a partir de ellos, pero cerrados para evitar que se modifiquen.

L – Principio de sustitución de Liskov (Liskov Substitution Principle)

Este principio establece que una subclase puede ser sustituida por su superclase. Es decir, podemos crear una subclase llamada Auto, la cual deriva de la superclase Vehículo. Si al usar la superclase el programa falla, este principio no se cumple.

I – Principio de segregación de interfaz (Interface Segretation Principle)

Este principio establece que los clientes no deben ser forzados a depender de interfaces que no utilizan. Es importante que cada clase implemente las interfaces que va a utilizar. De este modo, agregar nuevas funcionalidades o modificar las existentes será más fácil.

D – Principio de inversión de dependencias (Dependency Inversion Principle)

Este principio establece que los módulos de alto nivel no deben de depender de los de bajo nivel. En ambos casos deben depender de las abstracciones. Alto nivel se

refiere a operaciones cuya naturaleza es más amplia o abarca un contexto más general y bajo nivel son componentes individuales más específicos.

UML

El Lenguaje Unificado de Modelado (UML) desempeña un papel importante en el diseño y desarrollo de software, aunque también es de gran apoyo para todos aquellos procesos o sistemas que no tienen software en muchas industrias ya que nos permite ilustrar de forma visual el comportamiento de un sistema o proceso.

Gracias a UML podemos observar y detectar errores potenciales en las estructuras de aplicaciones, poder comprender el comportamiento del sistema de una forma más efectiva, así como también de otros procesos empresariales.

Historia

UML se implementó por primera vez en la década de los 90 gracias a tres ingenieros de software: Grady Booch, Ivar Jacobson y James Rumbaugh. El objetivo que buscaban era desarrollar una forma más sencilla de representar diseños complejos de software, a la vez que separaban la metodología del proceso. Hoy, el UML sigue siendo la indicación estándar para los desarrolladores, así como para gestores de proyectos, propietarios de negocios, empresarios tecnológicos y profesionales de distintos sectores.

Ventajas

- ✔ Hace más simples diseños o ideas complejas.
- ✔ Abre distintas líneas de comunicación.
- ✔ Contribuye a la automatización de producción de software y procesos.
- ✔ Contribuye a la solución de problemas en la arquitectura de software.
- ✔ Contribuye a mejorar la calidad de trabajo.
- ✔ Reduce costos.
- ✔ Mejora el time to market.

Tipos de diagramas

En UML podemos encontrar principalmente dos tipos de diagrama: diagramas de estructuras y diagramas de comportamiento, los cuales podemos utilizar para representar diferentes situaciones o escenarios.

El uso que le podemos dar a los distintos diagramas dependerá el rol que se cumple (diseñador, Project Manager, desarrollador) y dependerá de las necesidades que se tengan, recordando siempre que el objetivo que cumple UML es expresar visualmente diagramas que sean fáciles de entender para todos.

Diagramas estructurales

Este tipo de diagramas comúnmente los utilizamos cuando deseamos representar la estructura estática de un software o sistema, este tipo de diagramas comúnmente es utilizado para mostrar la implementación o niveles de abstracción que el sistema pueda tener.

Un diagrama estructural también puede representar la estructura jerárquica de los distintos componentes o módulos y las interacciones entre sí. Dentro de los diagramas UML estructurales podemos encontrar:

1. **Diagramas de clases**

Este diagrama es utilizado para representar el diseño lógico y físico de un sistema al más bajo nivel, podemos ver que dentro de UML es el diagrama más utilizado, este diagrama provee una imagen de diferentes estructuras o clases y la forma en que estas se relacionan entre sí, este tipo de diagrama está conformado por tres partes las cuales son:

- ✔ Nombre de clase
- ✔ Atributos de clase
- ✔ Métodos o funciones.

2. **Diagrama de objetos**

Este diagrama es utilizado para representar las distintas relaciones, así como también una representación de las instancias obtenidas a partir de los diagramas de clases. Evidenciando de mejor forma los defectos que pueda tener un diseño.

3. **Diagrama de componentes**

Gracias a este tipo de diagrama podemos visualizar las distintas agrupaciones lógicas de elementos y sus relaciones. En otras palabras, ofrece una vista más simplificada de un sistema complejo al desglosarlo en componentes más pequeños. Cada una de las piezas se muestra con una caja rectangular, que tiene su nombre escrito dentro. Los conectores definen la relación/las dependencias entre los diferentes componentes.

4. **Diagrama de estructura compuesta**

Este tipo de diagrama comúnmente es utilizado por personas externas al campo de desarrollo de software, es similar a un diagrama de clases, adopta un enfoque más profundo, que describe la estructura interna de múltiples clases y muestra las interacciones entre ellas.

5. **Diagrama de despliegue**

El diagrama de despliegue es útil para poder identificar en donde se implementarán los distintos componentes de un sistema haciéndose valer de los distintos nodos y artefactos que lo conforman y sus relaciones.

6. **Diagrama de paquetes**

Es utilizado para representar las dependencias entre los paquetes que componen un modelo. Su objetivo principal es mostrar la relación entre los diversos componentes grandes que forman un sistema complejo.

7. **Diagrama de perfiles**

Este tipo de diagrama es más similar a un lenguaje que a un diagrama. Comúnmente nos ayuda a crear nuevas propiedades y semántica para los diagramas UML al definir estereotipos personalizados, valores marcados y restricciones. Estos perfiles le permiten personalizar un metamodelo de UML para diferentes plataformas (por ejemplo, Java Platform, Enterprise Edition (Java EE) o Microsoft .NET Framework) y dominios (por ejemplo modelado de proceso empresarial, arquitectura orientada a servicios, aplicaciones médicas y más).

Diagramas UML de comportamiento

Este tipo de diagramas son utilizados para poder mostrar la funcionalidad de un sistema y nos ayuda a poder visualizar todo lo que debe de ocurrir dentro de un software.

Algunos diagramas de comportamiento son:

1. **Diagrama de Actividades**

Este tipo de diagramas nos muestran el conjunto de actividades que deben de realizarse para lograr un objetivo, muestran prácticamente como cada actividad conduce a la siguiente y como todas estas se relacionan entre sí, este tipo de diagrama también se conoce como asignación o modelado de proceso empresarial.

2. **Diagrama de casos de uso**

Este diagrama describe lo que un sistema hace las cosas, pero no la forma en que las hace. Un caso de uso es un conjunto de eventos que ocurren cuando un "actor" usa un sistema para completar un proceso. Un actor se define como cualquier persona o cualquier cosa que interactúa con el sistema (persona, organización o aplicación) desde fuera del sistema. Por lo tanto, un diagrama de casos de uso describe visualmente ese conjunto de secuencias y representa los requisitos funcionales del sistema.

3. **Diagrama de descripción general de interacción**

Este diagrama es complejo y similar al diagrama de actividad, ya que muestran una secuencia paso a paso de las actividades. Se diferencian en que un diagrama de descripción general de interacción es un diagrama de actividad que se compone de diferentes diagramas de interacción. Usan la misma composición que un diagrama de actividad (nodos iniciales, finales, decisión, unión, fork y join) e incorpora elementos como la interacción, el uso de la interacción, restricción de tiempo y restricción de la duración.

4. **Diagrama de tiempos**

Este tipo de diagramas también es conocido como diagrama de eventos, ya que muestra como los objetos o actores se desempeñan en una línea de tiempo, su enfoque principal es identificar la duración de los eventos y los cambios que producen en función de las restricciones de duración. Este tipo de diagramas está conformado por los siguientes elementos:

- ✔ Línea de vida: participante individual
- ✔ Línea de tiempo de estado: estados diferentes por los que pasa la línea de vida dentro de una canalización
- ✔ Restricción de duración: tiempo necesario para que se cumpla una restricción
- ✔ Restricción de tiempo: un periodo en el que el participante debe completar una acción
- ✔ Destrucción: cuando finaliza la línea de vida de un objeto. Después de que se realiza la destrucción en una línea de tiempo, no se produce otra ocurrencia.

5. **Diagrama de máquina de estados**

Este tipo de diagrama es utilizado para poder representar comportamientos complejos y en situaciones donde es esencial representar un buen nivel de detalle, este tipo de diagramas también es conocido como grafico de estados. Un diagrama de máquina de estados nos puede ayudar a describir el comportamiento de un objeto y la forma en que cambia según los eventos externos o internos.

6. **Diagrama de secuencia**

Este es uno de los diagramas UML más populares que podemos utilizar, este diagrama es bueno para mostrar todo tipo de procesos empresariales. Simplemente revela la estructura de un sistema, mostrando la secuencia de mensajes e interacciones entre actores y objetos cronológicamente. Los diagramas de secuencia muestran iteraciones y ramificaciones simples. Es favorable al realizar múltiples tareas.

7. Diagrama de comunicación

Diagrama de comunicación o colaboración es muy parecido a un diagrama de secuencia. Este tipo de diagrama enfatiza la comunicación entre objetos y muestra su organización que participa en una interacción y presenta iteraciones y ramificaciones más complejas.

<Bibliografía>

Referencias de contenido:

http://www.uml.org/what-is-uml.htm

Recursos adicionales

Enlaces:

- ✔ http://catarina.udlap.mx/u_dl_a/tales/documentos/lis/fuentes_k_jf/capitulo2.pdf
- ✔ http://www.inf.ed.ac.uk/teaching/courses/cs2/LectureNotes/CS2Ah/SoftEng/se02.pdf
- ✔ https://msdn.microsoft.com/es-es/library/bb972214.aspx
- ✔ https://es.scribd.com/document/54251038/Especificaciones-de-Los-Requisitos-Del-Software
- ✔ http://www.omg.org/spec/UML/2.2/
- ✔ https://profeuttec.yolasite.com/resources/Patrones%20de%20dise%C3%B1o%20-%20Erich%20Gamma.pdf

Libros:

- ✔ James A. Senn. (2001). Análisis y Diseño de Sistemas de Información. México: McGraw-Hill.
- ✔ Joseph Schmuller. (2000). Aprendiendo UML en 24 Horas. México: Pearson Education
- ✔ Shvets A. Sumergete en los patrones de diseño
- ✔ Robert C. Martin, Clean Code- A Handbook of Agile Software Craftsmanship.