Orientación a Objetos 1

Resumen teorías

Clase 24/08

Programa o sistema orientado a objetos

¿Cómo es un software construido con objetos?

Un conjunto de objetos que colaboran enviándose mensajes. Todo cómputo ocurre “dentro” de los objetos

La clave del éxito es poder agregar nueva funcionalidad (no prevista originalmente), reemplazar objetos o modificar objetos y que el sistema “no se entere”, ni se rompa.

Los objetos son responsables de:

- Conocer sus propiedades
- Conocer otros objetos (con los que colaboran)
- Llevar a cabo ciertas acciones

Características

- No hay un objeto “main”
- Algoritmos y datos ya no se piensan por separado
- Cuando codificamos, describimos clases
- Cuando se ejecuta el programa lo que tenemos son objetos que cooperan y que se crean dinámicamente durante la ejecución del programa
- Una jerarquía de clases no indica lo mismo que la jerarquía top-down
- La estructura general cambia: en vez de una jerarquía: Main/procedures/sub-procedures tenemos una red de “cosas” que se comunican
- Pensamos en qué “cosas” hay en nuestro software (los objetos) y cómo se comunican entre sí.
- Hay un “shift” mental crítico en forma en la cuál pensamos el software como objetos
- Mientras que la estructura sintáctica es “lineal” el programa en ejecución no lo es

Objeto

Es una abstracción de una entidad del dominio del problema. Ejemplos: Persona, Producto, Cuenta Bancaria, Auto, Plan de Estudios,…

Puede representar también conceptos del espacio de la solución (estructuras de datos, tipos “básicos”, archivos, ventanas, conexiones, iconos, adaptadores, ...)

Características

- Identidad: Para distinguir un objeto de otro (independiente de sus propiedades)
- Conocimiento: En base a sus relaciones con otros objetos y su estado interno
- Comportamiento: Conjunto de mensajes que un objeto sabe responder

Estado interno

- El estado interno de un objeto determina su conocimiento.
- El estado interno se mantiene en las variables de instancia (v.i.) del objeto.
- Es privado del objeto. Ningún otro objeto puede accederlo.
- El estado interno está dado por:
    - Propiedades básicas (intrínsecas) del objeto.
    - Otros objetos con los cuales colabora para llevar a cabo sus responsabilidades.

Comportamiento

- Un objeto se define en términos de su comportamiento.
- El comportamiento indica qué sabe hacer el objeto. Cuáles son sus responsabilidades.
- Se especifica a través del conjunto de mensajes que el objeto sabe responder: protocolo.
- Para poder enviarle un mensaje a un objeto, hay que conocerlo.
- Al enviarle un mensaje a un objeto, éste responde activando el método asociado a ese mensaje (siempre y cuando exista).
- Como resultado del envío de un mensaje puede retornarse un objeto.

Encapsulamiento

“Es la cualidad de los objetos de ocultar los detalles de implementación y su estado interno del mundo exterior”

Características:

- Esconde detalles de implementación.
- Protege el estado interno de los objetos.
- Un objeto sólo muestra su “cara visible” por medio de su protocolo.
- Los métodos y su estado quedan escondidos para cualquier otro objeto. Es el objeto quien decide qué se publica.
- Reduce el acoplamiento, facilita modularidad y reutilización

Métodos

Es la contraparte funcional del mensaje.

Expresa la forma de llevar a cabo la semántica propia de un mensaje particular (el cómo).

Un método puede realizar básicamente 3 cosas:

- Modificar el estado interno del objeto
- Colaborar con otros objetos (enviándoles mensajes)
- Retornar y terminar

Entrada/Salida con objetos

En la actualidad se habla de “lógica de dominio” y “lógica de interfaz” como asuntos separados

En un sistema diseñado correctamente, un objeto de dominio no debería realizar ninguna operación vinculada a la interfaz(mostrar algo) o a la interacción (esperar un “input”)

Formas de conocimiento

- Un objeto solo puede enviar mensajes a otros que conoce
- Para que un objeto conozca a otro lo debe poder “nombrar”. Decimos que se establece una ligadura (binding) entre un nombre y un objeto.
- Podemos identificar tres formas de conocimiento o tipos de relaciones entre objetos:
    - Conocimiento Interno: Variables de instancia.
    - Conocimiento Externo: Parámetros.
    - Conocimiento Temporal: Variables temporales.
    - Existe una cuarta forma de conocimiento especial: las pseudo-variables (como “this” o “self” y “super” o “parent”)

Clases e instancias

Una clase es una descripción abstracta de un conjunto de objetos.

Las clases cumplen tres roles:

- Agrupan el comportamiento común a sus instancias.
- Definen la forma de sus instancias.
- Crean objetos que son instancia de ellas

En consecuencia todas las instancias de una clase se comportan de la misma manera.

Cada instancia mantendrá su propio estado interno.

Method lookup

Cuando un objeto recibe un mensaje, se busca un método con la firma correspondiente (nombre y parámetros) en la clase de la cual es instancia.

Si se lo encuentra, se lo ejecuta “en el contexto del objeto”

Si no se lo encuentra, tendremos un error (o excepción) en tiempo de ejecución

Lenguajes estáticamente tipados, como Java, encuentran estos problemas al compilar

Esto se conoce como “Dynamic Binding” y es clave para la OO

Instanciación

Es el mecanismo de creación de objetos.
Los objetos se instancian a partir de un molde.
La clase funciona como molde.
Un nuevo objeto es una instancia de una clase.
Todas las instancias de una misma clase:

- Tendrán la misma estructura interna.
- Responderán al mismo protocolo (los mismos mensajes) de la misma manera (los mismos métodos).

Identidad

Las variables (de instancia, temporales, parámetros) son apuntadores a objetos
En cualquier momento, más de una variable puede apuntar a un mismo objeto
Para saber si apuntan al mismo objeto, utilizo “==“

Igualdad

Dos objetos (por ende, no idénticos) pueden ser iguales
La igualdad se define en función del dominio
Para saber si dos objetos son iguales, uso “equals()”

This

Hace referencia al objeto receptor del mensaje que resultó en la ejecución del método

Super

Cuando super recibe un mensaje, la búsqueda de métodos comienza en la clase inmediata superior a aquella donde está definido el método que envía el mensaje (sin importar la clase del receptor)

Introducción al Análisis y Diseño Orientado a Objetos

El análisis pone énfasis en una investigación del problema y los requisitos, en lugar de ponerlo en la solución.

El diseño pone énfasis en una solución conceptual que satisface los requisitos, en lugar de ponerlo en la implementación.

Modelo del dominio

Identificación de clases conceptuales

La tarea central es identificar las clases conceptuales relacionadas con el escenario que se está diseñando

Es mejor especificar en exceso un modelo del dominio con muchas clases conceptuales de grano fino que especificar por defecto

Consejos:

- Usar nombres del dominio del problema, no de la solución
- Omitir detalles irrelevantes
- No inventar nuevos conceptos (evitar sinónimos)
- Descubrir conceptos del mundo real

Estrategias:

- Identificación de frases nominales
- Utilización de una lista de categorías de clases conceptuales

Pasos para construirlo

1. Listar los conceptos candidatos
2. Graficarlos en un Modelo del Dominio
3. Agregar atributos a los conceptos
4. Agregar asociaciones entre conceptos.

Contratos

Son una de las formas de describir el comportamiento del sistema en forma detallada. Describen pre y post condiciones para las operaciones.

Secciones:

- Operación: nombre de la operación y parámetros.
- Precondiciones:
    - Suposiciones relevantes sobre el estado del sistema o de los objetos del Modelo del Dominio, antes de la ejecución de la operación.
    - No se validarán dentro de la operación, sino que se asumirán como verdaderas.
    - Son suposiciones no triviales que el lector debe saber que se hicieron.
    - Son declarativas (expréselas así) Ejemplo: El cliente está registrado como usuario del Sistema.
- Postcondiciones:
    - El estado del sistema o de los objetos del Modelo del Dominio, después de que se complete la ejecución de la operación.
    - Describen cambios en el estado de los objetos del Modelo del Dominio:
        - Creación y eliminación de instancias
        - Modificación de atributos
        - Creación o ruptura de asociaciones
    - Son declarativas (expréselas así) Ejemplo: Se creó un nuevo Cliente / El cliente fue incorporado al Sistema

Heurísticas para Asignación de Responsabilidades

Tipos de HAR

Experto en información

Descripción

Asignar una responsabilidad al experto en información (la clase que tiene la información necesaria para realizar la responsabilidad). Expresa la intuición de que los objetos hacen cosas relacionadas con la información que tienen.

Para cumplir con su responsabilidad, un objeto puede requerir de información que se encuentra dispersa en diferentes clases: expertos en información “parcial”. Ejemplo: ¿Quién tiene la responsabilidad de conocer el monto total de una compra? … La compra. Entonces: LineaDeVenta es responsable de conocer el subtotal por cada ítem EspecificaciónDelProducto es responsable de conocer el precio

Creador

Descripción

Asignar a la clase B la responsabilidad de crear una instancia de la clase A si:

- B contiene objetos A (agregación, composición).
- B registra instancias de A.
- B tiene los datos para inicializar objetos A.
- B usa a objetos A en forma exclusiva

La intención del Creador es determinar una clase que necesite conectarse al objeto creado en alguna situación. Eligiéndolo como el creador se favorece el bajo acoplamiento.

Ejemplo

¿Quién debe ser responsable de crear una LineaDeVenta? … La compra.

Controlador

Descripción

Asignar la responsabilidad de manejar eventos del sistema a una clase que representa el sistema global, dispositivo o subsistema.

La intención del Controlador es encontrar manejadores de los eventos del sistema, sin recargar de responsabilidad a un solo objeto y manteniendo alta cohesión

Ejemplo

¿Quién debe ser el controlador de los eventos ingresarLibro o finalizarCompra? … ManejadorCompras, Librería

Bajo Acoplamiento

Descripción

Asignar responsabilidades de manera que el acoplamiento permanezca lo más bajo posible.

El acoplamiento es una medida de dependencia de un objeto con otros. Es bajo si se mantienen pocas relaciones con otros objetos.

El alto acoplamiento dificulta el entendimiento y complica la propagación de cambios en el diseño.

No se puede considerar de manera aislada a otras heurísticas, sino que debe incluirse como principio de diseño que influye en la elección de la asignación de responsabilidad.

Alta Cohesión

Descripción

Asignar responsabilidades de manera que la cohesión permanezca lo más fuerte posible.

La cohesión es una medida de la fuerza con la que se relacionan las responsabilidades de un objeto, y la cantidad de ellas.

Ventaja: clases más fáciles de mantener, entender y reutilizar.

El nivel de cohesión no se puede considerar de manera aislada a otras responsabilidades y otras heurísticas, como Experto y Bajo acoplamiento.

Polimorfismo

Descripción

Cuando el comportamiento varía según el tipo, asigne la responsabilidad a los tipos/las clases para las que varía el comportamiento.

Nos permite sustituir objetos que tienen idéntica interfaz.

Ejemplo

El sistema de venta de libros debe soportar distintas bonificaciones de pago con tarjeta de crédito. (ya visto previamente) … Como la bonificación del pago varía según el tipo de tarjeta, deberíamos asignarle la responsabilidad de la bonificación a los distintos tipos de tarjeta.

No hables con extraños

Descripción

Evite diseñar objetos que recorren largos caminos de estructura y envían mensajes (hablan) a objetos distantes o indirectos (extraños).

Dentro de un método sólo pueden enviarse mensajes a objetos conocidos:

- Self/this
- un parámetro del método
- un objeto que esté asociado a self/this
- un miembro de una colección que sea atributo de self/this
- un objeto creado dentro del método

Los demás objetos son extraños (strangers)

Modelo de diseño

Los casos de uso sugieren los eventos del sistema que se muestran en los diagramas de secuencia del sistema.

En los contratos de las operaciones, utilizando conceptos del Modelo del Dominio, se describen los efectos que dichos eventos (operaciones) producen en el sistema.

Los eventos del sistema representan los mensajes que dan inicio a Diagramas de Secuencia del Diseño, mostrando las interacciones entre los objetos del sistema.

Los objetos con sus métodos y relaciones se muestran en el Diagrama de Clases del Diseño (basado en el Modelo del Dominio).

Pasos

1. Cree un diagrama de secuencia por cada operación del sistema en desarrollo (la operación es el mensaje de partida en el diagrama).
2. Si el diagrama queda complejo, sepárelo en diagramas menos complejos (uno por cada escenario).
3. Use el contrato de la operación como punto de partida; piense en objetos que colaboran para cumplir la tarea (la mayoría de estos objetos están definidos en el modelo del dominio).
4. Aplique las Heurísticas para Asignación de Responsabilidades (HAR) para obtener un mejor diseño.

Creación diagrama de clases de diseño

Identificar las clases que participan en los diagramas de interacción y en el Modelo del Dominio o Conceptual.

Graficarlas en un diagrama de clases.

Colocar los atributos presentes en el Modelo Conceptual.

Agregar nombres de métodos analizando los diagramas de interacción.

Agregar tipos y visibilidad de atributos y métodos.

Agregar las asociaciones necesarias.

Agregar roles, navegabilidad, nombre y multiplicidad a las asociaciones.

Entity vs Value Object

Las Entidades o clases del dominio de mi problema tienen un identificador, son modificables y comparables por Identidad.

Value Object:

- Son comparables por contenido (igualdad estructural), no tienen identificador.
- No viven por sí mismos, necesitan una entidad base, son intercambiables (un billete de 100 Pesos AR lo puedo cambiar por otro). Persisten adjunto a su base, no separadamente.
- Inmutables (No le defino setters).

SOLID

S

SRP: The Single-Responsibility Principle

Principio de Responsabilidad única.

Una clase debería cambiar por una sola razón.

Debería ser responsable de únicamente una tarea, y ser modificada por una sola razón (alta cohesión).

O

OCP: The Open-Closed Principle

Entidades de software (clases, módulos, funciones, etc.) deberían ser “abiertas” para extensión, y “cerradas” para modificación.

Abierto a extensión: ser capaz de añadir nuevas funcionalidades.

Cerrado a modificación: al añadir la nueva funcionalidad no se debe cambiar el diseño existente.

L

LSP: The Liskov Substitution Principle

Los objetos de un programa deben ser intercambiables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa.

Es decir, que si el programa utiliza una clase (clase A), y ésta es extendida (clases B, C, D, etc…) el programa tiene que poder utilizar cualquiera de sus subclases y seguir siendo válido. Uso correcto de herencia (Is a/es un) y polimorfismo.

I

ISP: The Interface-Segregation Principle

Las clases que tienen interfaces “voluminosas” son clases cuyas interfaces no son cohesivas.
Las clases no deberían verse forzadas a depender de interfaces que no utilizan. Cuando creamos interfaces (protocolos) para definir comportamientos, las clases que las implementan, no deben estar forzadas a incluir métodos que no va a utilizar.

D

DIP: The Dependency-Inversion Principle

A. Los módulos de alto nivel de abstracción no deben depender de los de bajo nivel.
B. Las abstracciones no deben depender de detalles. Los detalles deben depender de las abstracciones.

    - Módulos de alto nivel: se refieren a los objetos que definen qué es y qué hace el sistema.
    - Módulos de bajo nivel: no están directamente relacionados con la lógica de negocio del programa (no definen el dominio). Por ejemplo, el mecanismo de persistencia o el acceso a la red.
    - Abstracciones: se refieren a protocolos (o interfaces) o clases abstractas.
    - Detalles: son las implementaciones concretas, (cuál mecanismo de persistencia, etc).

Ser capaz de «invertir» una dependencia es lo mismo que ser capaz de «intercambiar» una implementación concreta por otra implementación concreta cualquiera, respecto a la misma abstracción.

Herencia vs. Composición

Herencia

Herencia total: debo conocer todo el código que se hereda -> Reutilización de Caja Blanca (conozco todo lo que heredo y puedo reutilizar todo)

Las clases y los objetos creados mediante herencia están estrechamente acoplados ya que cambiar algo en la superclase afecta directamente a la/las subclases.

Composición

Los objetos se componen en forma Dinámica -> Reutilización de Caja Negra
Los objetos pueden reutilizarse a través de su interfaz (sin conocer el código)

Las clases y los objetos creados a través de la composición están débilmente acoplados, lo que significa que se pueden cambiar más fácilmente los componentes sin afectar el objeto contenedor.