**Clase 1**

El paradigma de OO aporta

* Poder de abstracción
  + Utilizando el dominio como guía
  + Juntando datos y comportamiento
* Buena organización de nuestro programa
  + Alta cohesión: cada unidad tiene un foco claro
  + Bajo acoplamiento: minimizando la dependencia entre unidades
* Localidad de cambios

**Clase 2**

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 sistemas están compuestos (solamente) por un conjunto de objetos que colaboran para llevar a cabo sus responsabilidades.
* Los objetos son responsables de:
  + conocer sus propiedades,
  + conocer otros objetos (con los que colaboran) y
  + llevar a cabo ciertas acciones.

¿Qué es un 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 de los Objetos

Un objeto tiene:

* 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

El 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. (¿Cuál es el impacto de esto?)
* El estado interno está dado por:
  + Propiedades básicas (intrínsecas) del objeto.
  + Otros objetos con los cuales colabora para llevar a cabo sus responsabilidades.

Variables de instancia

* En general las variables son REFERENCIAS (punteros) a otros objetos con los cuales el objeto colabora.
* Algunas pueden ser atributos básicos

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.
* La realización de cada mensaje (es decir, la manera en que un objeto responde a un mensaje) se especifica a través de un método.
* Cuando un objeto recibe un mensaje responde activando el método asociado.
* El que envía el mensaje delega en el receptor la manera de resolverlo, que es privada del 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

Envío de un mensaje

* 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.

Especificación de un Mensaje

¿Cómo se especifica un mensaje?

* Nombre: correspondiente al protocolo del objeto receptor.
* Parámetros: información necesaria para resolver el mensaje.
* Cada lenguaje de programación propone una sintaxis particular para indicar el envío de un mensaje.

Métodos

¿Qué es un método?

* 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 de 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”)
* Si no puedo hacer I/O, ¿cómo pruebo mis objetos?
* Con tests unitarios (próximamente )

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”)

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.

This

* this (o en algunos lenguajes self) es una “pseudo-variable”
  + no puedo asignarle valor
  + Toma valor automáticamente cuando un objeto comienza a ejecutar un método
* this hace referencia al objeto que ejecuta el método (al receptor del mensaje que resultó en la ejecución del método)
* Se utiliza para:
  + Descomponer métodos largos (top down)
  + Reutilizar comportamiento repetido en varios métodos
  + Aprovechar comportamiento heredado (proximamente ...)
* En algunos lenguajes (p.e. Java):
  + Puede obviarse (es implícito), aunque en OO1 preferimos no hacerlo
  + Para desambiguar referencia a las variables de instancia del objeto

**Clase 3**

Relaciones entre objetos

* Un objeto conoce a otro porque
  + Es su responsabilidad mantener a ese otro objeto en el sistema (tiene un, conoce a)
  + Necesita delegarle trabajo (enviarle mensajes)
* Un objeto conoce a otro cuando
  + Tiene una referencia en una variable de instancia (rel. duradera)
  + Le llega una referencia como parámetro (re. temporal)
  + Lo crea (rel. temporal/duradera)
  + Lo obtiene enviando mensajes a otros que conoce (rel. temporal)

Objetos y el chequeo de tipos

* Java es un lenguaje, estáticamente, fuertemente tipado
  + Debemos indicar el tipo de todas las variables (relaciones entre objetos)
  + El compilador chequea la correctitud de nuestro programa respecto a tipos
* Se asegura de que no enviamos mensajes a objetos que no los entienden
* Cuando declaramos el tipo de una variable, el compilador controla que solo “enviemos a esa variable” mensajes acordes a ese tipo
* Cuando asignamos un objeto a una variable, el compilador controla que el tipo del objeto sea compatible con el de la variable

Tipos en lenguajes OO

* Tipo = Conjunto de firmas de operaciones/métodos (nombre, orden y tipos de los argumentos)
* Decimos que un objeto (instancia de una clase) “es de un tipo” si ofrece el conjunto de operaciones definido por el tipo
* Con eso en mente:
  + Cada clase en Java define “explícitamente” un tipo (es un conjunto de firmas de operaciones)
  + Cada instancia de una clase A “es del tipo” definido por esa clase
* Pero las clases no son la única forma de definir tipos

Interfaces

* Una clase define un tipo, y también implementa los métodos correspondientes
* Una variable tipada con una clase solo “acepta” instancias de esa clase (\*)
* Una interfaz nos permite declarar tipos sin tener que ofrecer implementación (desacopla tipo e implementación)
* Puedo utilizar Interfaces como tipos de variables
* Las clases deben declarar explícitamente que interfaces implementan
* Una clase puede implementar varias interfaces
* El compilador chequea que la clase implemente las interfaces que declara (salvo que sea una clase abstracta)

Polimorfismo

* Objetos de distintas clases son polimórficos con respecto a un mensaje, si todos lo entienden, aun cuando cada uno lo implemente de un modo diferente
* Polimorfismo implica:
  + Un mismo mensaje se puede enviar a objetos de distinta clase
  + Objetos de distinta clase “podrían” ejecutar métodos diferentes en respuesta a un mismo mensaje
* Cuando dos clases Java implementan una interfaz, se vuelven
* polimórficas respecto a los métodos de la interfaz

Polimorfismo bien aplicado

* Permite repartir mejor las responsabilidades (delegar)
* Desacopla objetos y mejora la cohesión (cada cual hace lo suyo)
* Concentra cambios (reduce el impacto de los cambios)
* Permite extender sin modificar (agregando nuevos objetos)
* Lleva a código más genérico y objetos reusables
* Nos permite programar por protocolo, no por implementación

Clase 5

Method Lookup con herencia

* Cuando un objeto recibe un mensaje, se busca en su clase un método cuya firma se corresponda con el mensaje. Si no lo encuentra, sigue buscando en la superclase de su clase, y en la superclase de esta, etc.

Super

* Podemos pensar a super como una “pseudo-variable” (como this)
  + no puedo asignarle valor
  + Toma valor automáticamente cuando un objeto comienza a ejecutar un método
* En un método, “super y this” hacen referencia al objeto que lo ejecuta (al receptor del mensaje)
* Utilizar super en lugar de this “solo cambia la forma en la que se hace el method lookup”
* Se utiliza para:
  + Solamente para extender comportamiento heredado (reimplementar un método e incluir el comportamiento que se heredaba para él).

Clase abstracta

* Una clase abstracta captura comportamiento y estructura que será común a otras clases
* Una clase abstracta no tiene instancias (no modela algo completo)
* Seguramente será especializada
* Puede declarar comportamiento abstracto y utilizarlo para implementar comportamiento concreto

Clase 6

Librería/framework de colecciones

* Todos los lenguajes OO ofrecen librerías de colecciones
  + Buscan abstracción, interoperabilidad, performance, reúso, productividad
* Las colecciones admiten, generalmente, contenido heterogéneo en términos de clase, pero homogéneo en términos de comportamiento
* La librería de colecciones de Java se organiza en términos de:
  + Interfaces: representa la esencia de distintos tipos de colecciones
  + Clases abstractas: capturan aspectos comunes de implementación
  + Clases concretas: implementaciones concretas de las interfaces
  + Algoritmos útiles (implementados como métodos estáticos)

Ordenando colecciones

* En Java, para ordenar nos valemos de un Comparador
* Los TreeSet usan un comparador para mantenerse ordenados
* Para ordenar List , le enviamos el mensaje sort, con un comparador como parámetro

Iterator (iterador externo)

* Todas las colecciones entienden iterator()
* Un Iterator encapsula:
  + Como recorrer una colección particular
  + El estado de un recorrido
* No nos interesa la clase del iterador (son polimórficos)
* El loop for-each esconde la existencia del iterador

Expresiones Lambda (clausuras / closures)

* Son métodos anónimos (no tienen nombre, no pertenecen a ninguna clase)
* Útiles para:
  + parametrizar lo que otros objetos deben hacer
  + decirles a otros objetos que me avisen cuando pase algo (callbacks)

Streams

* Objetos que permiten procesamiento funcional de colecciones
  + Las operaciones se combinan para formar pipelines (tuberías)
* Los streams:
  + No almacenan los datos, sino que proveen acceso a una fuente de datos subyacente (colección, canal I/O, etc.)
  + Cada operación produce un resultado, pero no modifica la fuente
  + Potencialmente sin final
  + Consumibles: Los elementos se procesan de forma secuencial y se descartan después de ser consumidos
  + La forma más frecuente de obtenerlos es vía el mensaje stream() a una colección

Stream Pipelines

* Para construir un pipeline encadeno envíos de mensajes
  + Una fuente, de la que se obtienen los elementos
  + Cero o más operaciones intermedias, que devuelven un nuevo stream
  + Operaciones terminales, que retornan un resultado
* La operación terminal guía el proceso. Las operaciones intermedias son Lazy: se calculan y procesan solo cuando es necesario, es decir, cuando se realiza una operación terminal que requiere el resultado.

Optional

* se utiliza para representar un valor que podría estar presente o ausente en un resultado.
* son una forma de manejar la posibilidad de valores nulos de manera más segura y explícita.
* Algunos métodos en Streams, como findFirst() o max(), devuelven un Optional para representar el resultado. Luego, se puede utilizar métodos de Optional como ifPresent(), orElse(), orElseGet(), entre otros, para manipular y obtener elvalor de manera segura.

Cuando no usar streams

Los lenguajes de programación proporcionan nuevas características que están disponibles para ser utilizadas.

Es esencial estar dispuesto a explorar y aprender estas funcionalidades

* saber cuándo y cómo aplicarlas de manera efectiva
* reconocer cuándo no son apropiadas.

Siempre que me sea posible voy a usar alguna construcción de más alto nivel

* Son más concisas
* Están optimizadas y probadas

Sin embargo, es importante reconocer que, en algunos casos, no es la elección óptima, y es necesario considerar otras soluciones más adecuadas a la situación específica

**Clase 7**

* Nuestras clases salen del lenguaje de los requerimientos (lenguaje del dominio)
* No todos los objetos aparecen en nuestra primera versión del modelo
* El diseño evoluciona a medida que agregamos funcionalidad y entendemos el dominio
* “delegar” es el criterio principal para un buen diseño
  + Nuestro objeto SISTEMA (el servicio/sistema) quedó con bastante comportamiento porque nuestros requerimientos tenían que ver mucho con buscar en las colecciones (no siempre es el caso)
  + No todo pasa por ese objeto SISTEMA
* En un par de clases vamos a aprender heurísticas que nos apoyan en el proceso de diseño

**Clase 8**

Lenguaje UML

* Es un lenguaje de modelado visual que se usa para especificar, visualizar, construir y documentar artefactos de un sistema de software.
* Permite capturar decisiones y conocimientos.

Diagramas de estructura

* Diagrama de Clases
* Diagrama de Paquetes
* Diagrama de Componentes
* Diagrama de Objetos
* Diagrama de Despliegue

Diagramas de Comportamiento

* Diagrama de Casos de Uso
* Diagramas de Interacción
  + Diagrama de Secuencia
  + Diagrama de Colaboración
* Diagrama de Máquinas de estado
* Diagrama de Actividades

Diagramas de Casos de Uso

* Un caso de uso es una representación del comportamiento de un sistema tal como se percibe por un usuario externo.
* Describe una interacción específica entre los actores y el sistema, proporcionando un proceso completo de cómo se utiliza el sistema en situaciones reales.
* El término 'actor' engloba a personas, así como a otros sistemas que interactúan con el sistema
* Los elementos de un modelo de casos de uso son:
  + Actores
  + Casos de uso
  + Relaciones

Diagrama de Clases

(El que usamos en UML)

Una clase es una descripción de conjunto de objetos que comparten los mismos atributos, objetos que comparten los mismos atributos, operaciones, métodos, relaciones y semántica.

Una clase es representada gráficamente por cajas con tres compartimientos



Diagrama de paquetes

* Permiten la agrupación de clases
* Se utiliza para organizar los elementos.
* Son útiles para mostrar la organización de un sistema y cómo los elementos se agrupan y relacionan entre sí.
* Se quiere
  + una alta cohesión dentro de un paquete. Los elementos dentro de un paquete están relacionados
  + poco acoplamiento entre ellos (exportando sólo aquellos elementos necesarios e importando solo lo necesario)

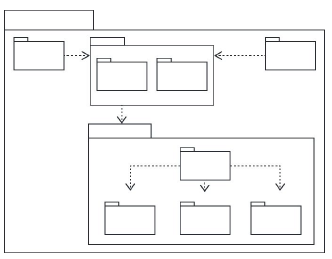


Diagrama de secuencia

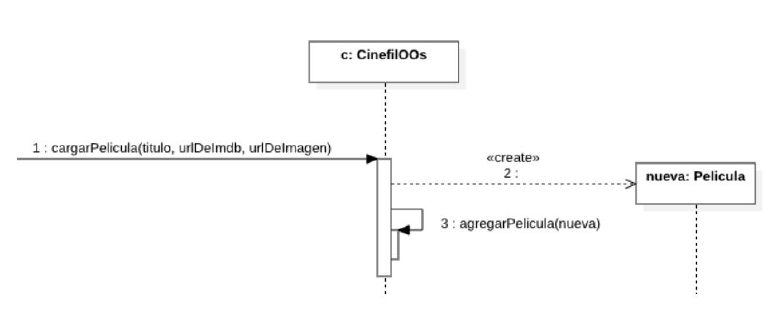
Un diagrama de secuencia es un tipo de diagrama de interacción porque describe cómo —y en qué orden— colabora un grupo de objetos.

- Muestra claramente cómo interactúan distintos objetos en un sistema a lo largo del tiempo.

En un diagrama de secuencia

- los objetos se representan en la parte superior del diagrama

- el tiempo avanza de arriba hacia abajo.



Un fragmento combinado es un elemento que se utiliza para representar la lógica de control y las estructuras condicionales en una secuencia de interacción entre objetos. A través de ellos se pueden especificar bloques repetitivos, opcionales y alternativos, entre otros.

Fragmentos más utilizados:

* opt
* alt
* loop
* alt (Alternativa): Este tipo de fragmento se utiliza para modelar una elección entre diferentes opciones de interacción. En cada opción se evalúa una condición booleana para determinar cuál de las opciones se ejecutará.
* opt (Opcional): Representa una parte de la secuencia de interacción que puede o no ejecutarse, dependiendo de una condición booleana. Si la condición es verdadera, se ejecuta la parte opcional; de lo contrario, se omite.
* loop (Bucle): Se utiliza para modelar repeticiones de una secuencia de interacción. Puede especificar el número de repeticiones o utilizar una condición para controlar la terminación del bucle.

¿Cómo se define UML?

Documento de especificación de UML

Metamodelo de UML

* es una especificación que define las construcciones y elementos básicos que pueden utilizarse para crear diagramas en UML.
* es esencialmente una descripción formal de cómo se estructuran y relacionan los elementos

Herramientas de modelado

Las herramientas de modelado UML pueden ser tanto gráficas como basadas en el metamodelo UML

Herramienta gráfica: permite dibujar elementos con la misma imagen que UML

Herramientas basadas en el metamodelo UML: Estas herramientas se centran más en la manipulación directa de los elementos del metamodelo UML.

**Clase 9**

Tipos de test

* **Tests funcionales**
* Test no funcionales
* **Tests de unidad**
* Tests de integración
* Tests de regresión
* Test punta a punta
* **Tests automatizados**
* Test de carga
* Test de performance
* Test de aceptación
* Test de UI
* Test de accesibilidad
* Alpha y beta tests
* Test A/B

Test de unidad

* Test que asegura que la unidad mínima de nuestro programa funciona correctamente, y aislada de otras unidades
  + En nuestro caso, la unidad de test es el método
* Testear un método es confirmar que el mismo acepta el rango esperado de entradas, y que retorna le valor esperado en cada caso
  + tengo en cuenta parámetros, estado del objeto antes de ejecutar el método, objeto que retorna el método, y estado del objeto al concluir la ejecución del método

Tests automatizados

* Se utiliza software para guiar la ejecución de los tests y controlar los resultados
* Requiere que diseñemos, programemos y mantengamos programas “tests”
  + En nuestro casos serán objetos
* Suele basarse en herramientas que resuelven gran parte del trabajo
* Una vez escritos, los puedo reproducir a costo mínimo, cuando quiera
* Los tests son “parte del software” (y un indicador de su calidad)

jUnit

* jUnit es un framework, en Java, para automatizar la ejecución de tests de unidad
* Ayuda a escribir tests útiles
* Cada test se ejecuta independientemente de otros (aislados)
* jUnit detecta, recolecta, y reporta errores y problemas
* xUnit es su nombre genérico; lo que aprendamos podemos llevarlo a otros lenguajes

Anatomía de un test suite jUnit

* Una clase de test por cada clase a testear
* Un método que prepara lo que necesitan los tests
* Y queda en variables de instancia
* Uno o varios métodos de test por cada método a testear
* Un método que limpia lo que se preparó (si es necesario)

Independencia entre tests

* No puedo asumir que otro test se ejecutó antes o se ejecutará después del que estoy escribiendo
* Por cada método de test:
  + Se crea una nueva instancia de nuestra clase de test
  + Se prepara (método marcado como @BeforeEach)
  + Se ejecuta el test y se registran errores y fallas

¿Por qué, cuándo, y como testear? (revisado)

* Testeamos para encontrar bugs
* Testeamos con un propósito (buscamos algo)
* Pensamos por qué testear algo y con qué nivel queremos hacerlo
* Testeamos temprano y frecuentemente
* Testeo tanto como sea el riesgo del artefacto

Estrategia general

* Pensar que podría variar (que valores puede tomar) y que pueda causar un error o falla
* Elegir valores de prueba para maximizar las chances de encontrar errores haciendo la menor cantidad de pruebas posibles
* Nos vamos a enfocar en dos estrategias:
  + Particiones equivalentes
  + Valores de borde

Tests de particiones equivalentes

* Partición de equivalencia: conjunto de casos que prueban lo mismo o revelan el mismo bug
  + Asumo que, si un ejemplo de una partición pasa el test, los otros también lo harán. Elijo uno.
* Si se trata de valores en un rango, tomo un caso dentro y uno por fuera en cada lado del rango
  + Ej., la temperatura debe estar entre 0 y 100 - > casos: -50, 50 , 150.
* Si se trata de casos en un conjunto, tomo un caso que pertenezca al conjunto y uno que no
  + Ej., la temperatura debe ser un valor positivo- > Casos: -50, 50

Tests con valores de borde

* Los errores ocurren con frecuencia en los límites y ahí es donde los vamos a buscar
* Intentamos identificar bordes en nuestras particiones de equivalencia y elegimos esos valores
* Buscar los bordes en propiedades del etilo: velocidad, cantidad, posición, tamaño , duración, edad, etc.
* Y buscar valores como: primero/último, máximo/mínimo, arriba/abajo, principio/fin, vacío/lleno, antes/después, junto a, alejado de , etc.

Testing en OO1

* En el marco de OO1, testear es asegurarnos de que nuestros objetos hacen lo que se espera, como se espera
* Escribir tests de unidad (con JUnit) es parte “programar”
* Escribir tests nos ayuda a entender que se espera de nuestros objetos
* Podemos hacer TDD o no, pero siempre escribimos tests

**Clase 10**

Análisis y Diseño O.O.

* Análisis

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

* Diseño

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

Modelo del Dominio

Un Modelo del Dominio es una representación visual de las clases conceptuales del mundo real en un dominio de interés.

Construyendo el Modelo del Dominio

Pasos a seguir:

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

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.

Encontrar conceptos (y sus atributos) mediante la identificación de los sustantivos en la descripción textual del dominio del problema.

Ejemplo:

1. El Cliente selecciona un libro para agregar al carrito.
2. El sistema agrega el libro al carrito.
3. El carrito agrega el libro y se presenta la descripción, precio y suma parcial.

Agregue clases conceptuales de especificación cuando necesite la descripción de un artículo o servicio, o si al eliminar las instancias que describen, se pierde información.

Agregar atributos

Se identifican los atributos que son necesarios para satisfacer los requerimientos de información de los casos de uso en desarrollo.

Los atributos en un modelo deberían ser, preferiblemente, atributos simples o tipos de datos primitivos.

Los tipos de datos de los atributos más comunes incluyen:

* Boolean
* Número
* String (texto)
* Temporales (Hora / Fecha)

Recuerde relacionar las clases conceptuales con asociaciones, no con atributos

Represente lo que inicialmente podría considerarse un tipo de dato primitivo como una clase conceptual, si:

* Está compuesto de secciones separadas
* Tiene operaciones asociadas
* Tiene otros atributos
* Es una cantidad con una unidad
* Es una abstracción de uno o más tipos con esas cualidades

La mejor manera de expresar que un concepto utiliza a otro es, nuevamente, con una asociación, no con un atributo de clave ajena

**Clase 11**

Responsabilidades de los objetos

* Conocer
  + Conocer sus datos privados encapsulados
  + Conocer sus objetos relacionados
  + Conocer cosas derivables o calculables
* Hacer
  + Hacer algo él mismo
  + Iniciar una acción en otros objetos
  + Controlar o coordinar actividades de otros objetos

Heurísticas para Asignación de Responsabilidades

* La habilidad para asignar responsabilidades es extremadamente importante en el diseño.
* La asignación de responsabilidades generalmente ocurre durante la creación de diagramas de interacción.
* Experto en Información
* Creador
* Controlador
* Bajo Acoplamiento
* Alta Cohesión

Experto en Información (Experto)

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”.

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.

Controlador

Descripción: asignar la responsabilidad de manejar eventos del sistema a una clase que representa:

* El sistema global, dispositivo o subsistema

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 mantiene 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.

Clases Conceptuales: “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).
* ¡Si el Value Object no es inmutable, entonces NO es un Value Object !
* ¿Como identificar un Value Object en el modelo?

Cuando necesitamos por ej. modelar Moneda, Fecha, Dirección, que puedan tener cierto comportamiento (getters..)

¿Como representar “Value Objects” en el modelo?

Principios S O L I D

Relacionados a las HAR(Heurísticas para Asignación de responsabilidades), para un buen estilo de DOO.

Promueven Alta Cohesión y Bajo Acoplamiento.

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) 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

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 de Clases

* Herencia total: debo conocer todo el código que se hereda -> Reutilización de Caja Blanca
* Usualmente debemos redefinir o anular métodos heredados
* Los cambios en la superclase se propagan automáticamente a las subclases
* Herencia de Estructura vs. Herencia de comportamiento
* Es útil para extender la funcionalidad del dominio de aplicación

Composición de Objetos

* Los objetos se componen en forma Dinámica -> Reutilización de Caja Negra
* Los objetos pueden reutilizarse a través de su interfaz (sin conocer el código)
* A través de las relaciones de composición se pueden delegar responsabilidades entre los objetos

Herencia vs. Composición de Objetos

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

Cómo solucionar un mal uso de la herencia

Composición, no herencia…

* Mecánicamente, heredar de ArrayList no cumple con el encapsulamiento; en cambio,
* componer con ArrayList para contener la colección de objetos de la pila es una opción de implementación que permite ocultarla públicamente.
* En este caso, en vez de heredar, ¡¡el uso o composición permite reúso y mantiene el encapsulamiento!!