Apuntes para final de Teoría Cátedra

Inicio

Introducción

- **POO:** La programación orientada a objetos (POO) es un paradigma de programación que se basa en el concepto de "objetos". Cada objeto representa un elemento del "mundo" con sus relaciones y sus atributos.

- **Clase:** Es la plantilla con la que se crean objetos, definiendo métodos y atributos.

    - Es responsable de la creación de objetos que la instancien.
    - Para crear un objeto, necesitamos enviar un mensaje a la clase (`new`).
    - Podemos usar `new` (deja las variables de instancia con valores nulos), o implementar un mensaje en la clase que inicialice los datos (`inicializar`) para luego pedir algunos datos iniciales (`inicializar(...)`), luego usamos `new inicializar(...)`.
    - También podemos tener un mensaje en la clase personalizado que oculte el inicializar (`crear(...)`) donde le paso los datos iniciales.
    - Se organizan en una jerarquía de clases.
        - Ocultan la estructura y las operaciones propias al resto del programa.
        - Encapsulan todas las operaciones del mismo tipo.
        - Describen la interfaz de los objetos (qué mensajes pueden responder).

- **Objeto:** Es una instancia de una clase.

    - Una vez que es instanciado comienza su tiempo de vida.
    - Identidad: Cada objeto puede ser identificado como una entidad única.
    - Estado: Comprende todas las variables de instancia del objeto y sus valores en un momento determinado del tiempo.
    - Comportamiento: Cómo actúa y reacciona en términos de sus cambios de estado y pasaje de mensajes.
    - Relaciones: Se relacionan con otros objetos mediante el envío de mensajes.

- **Mensaje:** Los objetos interactúan entre sí mediante mensajes. Tienen que llevar la información necesaria para resolverlo. Puede ayudarse de otros objetos para resolver lo que se le pide. Proceso dinámico de pedir a un objeto que lleve a cabo una acción específica. Un mensaje siempre se envía de un objeto a otro. La acción realizada en respuesta al mensaje no es fija, puede variar dependiendo de la clase del objeto receptor.

    - Un único método contiene el código que implementa el mensaje. El mismo código se aplica a todas las instancias de la clase. Es posible acceder y actualizar atributos.
    - `self` y `this` permiten hacer referencia a la instancia actual de la clase.
    - Un objeto que no existe no puede recibir mensajes.
    - Forma parte de la interfaz de un objeto.

- **Responsabilidad:** Las responsabilidades están relacionadas con las obligaciones de un objeto en términos de su comportamiento.

    - Conocimiento: Sobre sus atributos y asociaciones.
    - Acción: Hacer algo por sí mismo, iniciar acción en otros objetos, controlar y coordinar actividades en otros objetos.
    - La responsabilidad no es un método, pero se implementa utilizando métodos, que actúan solos o con otros métodos y sobre otros objetos.

- **Encapsulamiento:** Sabemos que un objeto tiene que realizar una acción pero no nos importa cómo.
- **Identidad:** Cada objeto tiene su identidad y es tratado de forma individual.
- **Estado:** Abarca a todos los atributos y sus valores actuales.
- **Comportamiento:** De un objeto es cómo actúa y reacciona en términos de estado y mensajes.
- **Abstracción:** Considerar la resolución de un problema sin tener en cuenta los detalles por debajo de un cierto nivel, filtramos los aspectos relevantes obteniendo soluciones más generales.
- **MDO:** Es la forma que tenemos de modelar las relaciones entre objetos.
    - UML para describir las clases y sus relaciones (las que tendrán que respetar los objetos).
    - Un sistema de software es usualmente complejo.
    - Es necesario descomponerlo en partes manejables y de complejidad comprensible.
    - Cada parte puede ser representada como un modelo que describe y abstrae los aspectos esenciales del sistema.
    - Los modelos están compuestos por diagramas y documentos que describen cosas.
    - Los modelos enfatizan información estática o dinámica del sistema.
    - Cada relación entre clases puede tener: nombre, multiplicidad (cuántas instancias de una clase se pueden relacionar), y navegabilidad (dirección de recorrido).
    - Una clase se puede relacionar a sí misma.
    - En la navegabilidad se determina quién puede enviar mensajes a la otra clase y a qué clases se lo puede enviar.
    - Para modelar la lógica de los mensajes se usa un diagrama de secuencias.
    - Donde marcamos qué pasa cuando a una instancia se le solicita una acción por medio.
    - También se detallan los mensajes a nivel lógico.
- **Arquitectura en tres capas:** Es una forma de separar una solución en tres capas: interfaz de usuario, la lógica de aplicación, y el acceso a los datos.
    - Interfaz de usuario: Es lo que el usuario verá del sistema y con el que interactuará.
    - Lógica de aplicación: Objetos del dominio, que cumplen con los requerimientos de la aplicación.
    - Acceso a datos: Mecanismos de almacenamiento persistente.
- **Herencia:** Es un mecanismo para poder repetir atributos y métodos en otras clases sin tener que repetir código.
    - Una clase toma el comportamiento y la estructura interna de otra y agrega su propia conducta.
    - El comportamiento y datos asociados con la clase hija (subclase) son siempre una extensión de las propiedades asociadas con la clase padre (superclase).

    - Cuando recibe un mensaje lo busca en la clase que lo recibe, en caso de no encontrarlo lo buscará en la superclase.
    - Todos los objetos en la mayoría de los lenguajes son subclase de la clase `Object`.
    - Una clase abstracta es aquella que no puede tener instancias, describe atributos y comportamientos de subclases, tiene operaciones abstractas.

- **Polimorfismo:** Los objetos de distintas clases y de la misma familia entienden los mismos mensajes.

    - Igual semántica, diferentes implementaciones, un mismo mensaje puede provocar la invocación de métodos diferentes.
    - **Sobrecarga:** Cambian la cantidad de parámetros en un mismo mensaje.
    - **Refinamiento:** Hace uso del método heredado pero cambia luego el resultado.
    - **Reemplazo:** Reemplaza el método heredado. Suele ser usado cuando una clase hereda de una clase abstracta.
    - **Puro:** La variable polimorfica tiene un metodo auxiliar que es redefinido o remplazado de manera que el comportamiento de un metodo varia de una subclase a otra.
    - Visibilidad: Pública, privada y protected (solo accesible desde subclases).

- **Ligadura:** Es la forma en la que el lenguaje asocia las llamadas a funciones y referencias a variables con las definiciones correspondientes (sus clases).

    - **Ligamento Estático:** En el momento de la declaracino de una varible esta solo puede acceder a sus metodos y los heredados de la subclase.
        - Ej: tenemos la super clase A con los metodos (M1, M2) y la subclase B con los metodos (M1, M2, M3) si hacemos lo siguiente `A objeto = new B();` ahora la instancia objeto solo podra acceder a (M1, M2) de A y a (M3) de B. por lo que el remplazo o el refinamiento no tendran efecto.
        - cada lenguajes con ligamento estatico es diferente por lo que existen formas de indicar que el ligamente tiene que ser dinamico.
    - **Dinámica:** Ocurre en tiempo de ejecución, por lo que la evaluación de qué tipo de función o método llamar ocurre mientras se ejecuta la aplicación, de forma que se buscaran los metodos en la jerarquia de la clase.

- **Mecanismos de reutilización:**

    - **Tipos de herencia:**

        - **Especialización/Extensión:** Las subclases agregan nuevas funcionalidades sin alterar el comportamiento heredado. La subclase es un subtipo de la superclase, respetando su interfaz y agregando nuevas funcionalidades. Ejemplo: una clase `Ventana` con métodos como mover, cambiar tamaño, y minimizar, y una subclase `VentanaDeTexto` que añade la capacidad de editar texto.

        - **Especificación:** Las superclases tienen métodos abstractos y concretos, proporcionando una interfaz común. Las subclases implementan los métodos abstractos definidos por la superclase. Ejemplo: una clase `Figura` con un método abstracto `dibujar()` y subclases como `Circulo` y `Rectangulo` que implementan `dibujar()`.

        - **Generalización:** Factorización de funcionalidades comunes en una superclase basada en datos y no en comportamiento. Creación de una superclase que factoriza

funcionalidades comunes de varias subclases. Ejemplo: clases con una interfaz común basadas en valores de datos.

    - **Limitación:** Reutilización de código donde la subclase es más restrictiva que la superclase, no cumpliendo con el principio de sustitución. Ejemplos problemáticos: `Pila` como subclase de `ColaDoble` o `Set` como subclase de `Lista`.

- **Mecanismos de reuso:**
    - La herencia puede no siempre ser adecuada, ya que puede llevar una carga innecesaria de métodos y datos en las subclases. A veces, introduce métodos y datos que no son relevantes para las subclases, lo que lleva a una carga innecesaria.
    - **Colaborador Interno (Delegación):** Se utiliza cuando solo se requiere parte de la funcionalidad de una superclase, proporcionando un control más granular sobre qué métodos y datos reutilizar. Una clase utiliza otra clase para implementar algunas de sus funcionalidades, seleccionando qué partes de la interfaz se utilizan y cuáles se ignoran.

- **Herencia vs colaborador interno:**
    - **Herencia:** Relación "Es un" donde un concepto es una instancia especializada de otro. Uso del principio "ES UN", facilitando la reutilización de código y la consistencia de interfaces.
    - **Colaborador interno:** Relación de uso, donde un concepto utiliza a otro para implementar ciertas funcionalidades. Ofrece más control sobre qué operaciones se aplican a una estructura de datos particular, mejorando la mantenibilidad y eficiencia del código.

- **Herencia Múltiple:**
    - Ventajas y problemas de la herencia múltiple, como la ambigüedad de nombre y la herencia de ancestros comunes.
    - Soluciones propuestas en lenguajes que soportan herencia múltiple.
    - Permite a una clase heredar de múltiples superclases combinando diferentes características y comportamientos.
    - Determinar cómo manejar características heredadas de un ancestro común, resolviendo si deben tener copias duplicadas o compartidas.

- **Excepciones:**
    - Las excepciones son mecanismos que permiten gestionar errores de ejecución de una forma controlada.
    - Permiten separar la lógica de manejo de errores del código principal.
    - Ejemplos comunes incluyen manejo de archivos no encontrados, errores de red, etc.
    - La estructura básica incluye el lanzamiento de una excepción (`throw`) y su captura (`try-catch`).