Apunte Algoritmos y Programación III - FIUBA

lcondoriz

Agosto 2023

Índice

1. Introducción

Apunte de la Materia Algoritmos y programación III.

2. Programación Orientada a Objetos

Definición 2.1. Paradigmas de Programación

- **Imperativos:** (énfasis en la ejecución de instrucciones)
  - Programación Procedimental (p. ej. Pascal).
  - Programación Orientada a Objetos (p. ej. Smalltalk, Java = multi-paradigma).
- **Declarativos:** (énfasis en la evaluación de expresiones)
  - Programación Funcional (p. ej. Haskell).
  - Programación Lógica (p. ej. Prolog).

2.1. Sistemas Orientados a Objetos

Definición 2.2. (Entidad) Una entidad es un objeto del mundo real que tiene un identificador único, un estado y un comportamiento [2].

Tipos de entidades:

- **Entidades físicas:** (p. ej. un auto, una persona, un libro).
- **Entidades conceptuales:** (p. ej. un viaje, una reserva, un préstamo).
- **Entidades fuertes:** son entidades que pueden sobrevivir por sí solas.
- **Entidades debil:** no pueden existir sin una entidad fuerte y se representan con un cuadrado con doble línea

Definición 2.3. (Clase) Una clase es un conjunto de objetos que comparten características y comportamientos comunes, así como propiedades y atributos comunes. Es un prototipo o plano definido por el usuario a partir del cual se crean objetos

Definición 2.4. (Objeto) Un objeto es una entidad que puede recibir mensajes, responder a los mismos y enviar mensajes a otros objetos. Un objeto es una entidad que tiene comportamiento.

Todo objeto tiene tres características:

- **Identidad:** Es un identificador único que lo diferencia de los demás objetos. La identidad es lo que distingue a un objeto de otro.

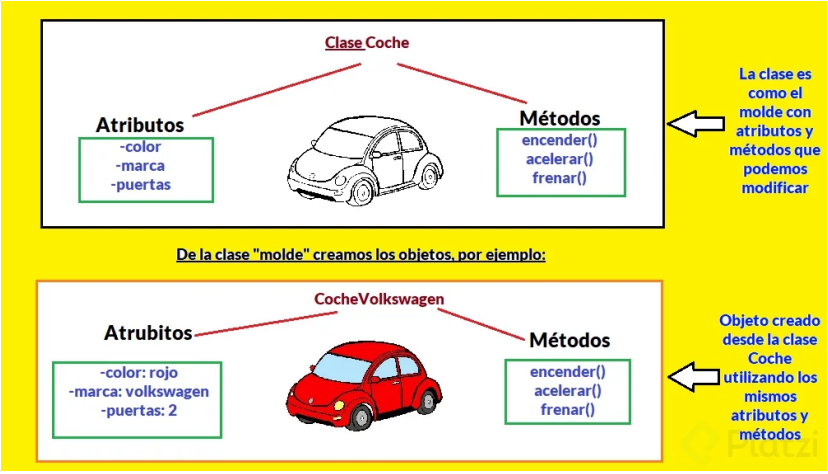

Figura 1: Clase

- **Estado:** El estado es la situación en que un objeto se encuentra. Un objeto puede cambiar su estado a través del tiempo.
- **Comportamiento:** Es el conjunto de posibles respuestas de un objeto ante los mensajes que recibe. El comportamiento de un objeto está compuesto por las respuestas a los mensajes que recibe un objeto, que a su vez pueden provocar:
  - Un cambio de estado en el objeto receptor del mensaje.
  - La devolución del estado de un objeto, en su totalidad o parcialmente
  - El envío de un mensaje desde el objeto receptor a otro objeto (delegación)

Cuando un objeto es creado a partir de una clase, se dice que el objeto es una **instancia** de la clase, esto se puede ver en la Figura 1, creación del objeto CocheVolkswagen.

Definición 2.5. (Atributo (o variable de instancia)) En POO, llamamos atributo a una variable interna del objeto que sirve para almacenar parte del estado del mismo

Definición 2.6. (Método) Llamamos método a la implementación de la respuesta de un objeto a un mensaje. En términos de implementación, se asemeja a funciones o procedimientos de programación en otros paradigmas.

Definición 2.7. (Interfaz) Al conjunto de las firmas de los métodos se lo suele llamar interfaz o protocolo del objeto. La interfaz de un objeto es el conjunto de mensajes a los que puede responder.

Definición 2.8. (Mensaje, cliente y receptor) Un mensaje es la interacción entre un objeto que pide un servicio y otro que lo brinda.

El objeto que envía el mensaje se llama objeto cliente y quien recibe el mensaje se llama objeto receptor.

Definición 2.9. (Delegación) Cuando un objeto, para responder un mensaje, envía mensajes a otros objetos, decimos que delega ese comportamiento en otros objetos. También es la colaboración de otros objetos para poder responder un mensaje.

Definición 2.10. (Encapsulamiento) Cada objeto es responsable de responder a los mensajes que recibe, sin que quien le envía el mensaje tenga que saber cómo lo hace. Esto es lo que llamamos encapsulamiento.

"Tell, don't ask", implica que los objetos deben manejar su propio comportamiento, sin que nosotros manipulemos su estado desde afuera.

Definición 2.11. (Polimorfismo) El polimorfismo es la capacidad de respuesta que tienen distintos objetos de responder de maneras diferentes a un mismo mensaje.

2.1.1. Pasos para resolver un problema

1. **Encontrar objetos:** (entidades del dominio del problema) Comenzar con el diseño del modelo.
2. **Determinar los mensajes:** (cómo deben interactuar los objetos)
   a) Diagrama de secuencia UML, como colaboran los objetos.
3. **implementar el comportamiento de los objetos:**

2.1.2. Relaciones estáticas

Son los niveles de visibilidad que controlan el acceso a los atributos y métodos de una clase. min 46 Youtube

- **Público:** (public) se puede acceder desde cualquier clase.
- **Privado:** (private) solo se puede acceder desde la misma clase.
- **Protegido:** (protected) se puede acceder desde la misma clase y desde las subclases.

2.2. Diseño por contrato y un procedimiento constructivo

La idea primigenia del diseño por contrato es, entonces, que un objeto servidor brinda servicios a objetos clientes sobre la base de un contrato que ambos se comprometen a cumplir.

Definición 2.12. (Modelo) Un modelo es una representación simplificada de la realidad, que nos permite entenderla y manipularla.

Definición 2.13. (Abstracción) Es una simplificación que incluye solo aquellos detalles relevantes para determinado propósito y descarta los demás.

2.2.1. Precondiciones

Las precondiciones expresan en qué estado debe estar el medio ambiente antes de que un objeto cliente le envíe un mensaje a un receptor. En general, el medio ambiente está compuesto por el objeto receptor, el objeto cliente y los parámetros del mensaje, pero hay ocasiones en que hay que tener en cuenta el estado de otros objetos.

Ante el incuplimiento de una precondición se lanza una excepción.

Definición 2.14. (Excepción) Una excepción es un objeto que el receptor de un mensaje envía a su cliente como aviso de que el propio cliente no está cumpliendo con alguna precondición de ese mensaje.

2.2.2. Postcondiciones

El conjunto de postcondiciones expresa el estado en que debe quedar el medio como consecuencia de la ejecución de un método. En términos operativos, es la respuesta ante la recepción del mensaje.

El cumplimiento de las postcondiciones es responsabilidad del receptor. Si una postcondición no se cumple se debe a que el método está mal programado por quien deba implementar el objeto receptor.

Definición 2.15. (Prueba unitaria) Una prueba unitaria es aquélla prueba que comprueba la corrección de una única responsabilidad de un método.

Corolario: Deberíamos tener al menos una prueba unitaria por cada postcondición.

2.2.3. Invariantes

Los invariantes son condiciones que debe cumplir un objeto durante toda su existencia.

Los invariantes suelen estar presentes a través de precondiciones o postcondiciones.

2.3. Pruebas unitarias

2.3.1. Desarrollo empezando por las pruebas

2.4. colaboración entre objetos

Veremos delegación y programación por diferencia, cuestiones de comportamiento que se apoyan en los aspectos estructurales llamados asociación y herencia.

2.5. Pilares de la POO

1. **Abstracción:** Es la capacidad de representar un objeto del mundo real en un programa. Es el proceso de identificar las características esenciales de un objeto y eliminar las características no esenciales.

2. **Encapsulamiento:** Es la capacidad de ocultar los detalles de implementación de un objeto.

3. **Polimorfismo:** Es la capacidad de enviar mensajes sintácticamente iguales a objetos de distintas clases y que estos respondan de manera diferente.

4. **Herencia:** Es la capacidad de definir nuevas clases a partir de otras ya existentes.

2.6. Herencia

Definición 2.16. (Herencia) La herencia es un mecanismo que permite definir nuevas clases a partir de otras ya existentes. La herencia permite definir una clase a partir de otra, reutilizando sus atributos y comportamientos. La clase de la que se hereda se llama *superclase* y la clase que hereda se llama *subclase*.

La herencia es una relación entre clases, por la cual se define que una clase puede ser un caso particular de otra. A la clase más general la llamamos madre y a la más patricular hija.

Corolario: Cuando hay herencia, todas las instancias de la clase hija son también instancias de la clase madre.

Definición 2.17. (Programación por diferencia) La programación por diferencia es un procedimiento constructivo que consiste en definir una clase a partir de otra, agregando o modificando atributos y comportamientos.

Programamos por diferencia cuando indicamos que parte de la implementación de un objeto está definida en otro objeto, y por lo tanto sólo implementamos las diferencias específicas.

2.7. Clases

Tipos de clases:

1. **Clases abstractas:** Son clases que no pueden ser instanciadas, es decir, no pueden tener objetos. Son clases que sirven para definir un comportamiento común a varias clases.

2. **Clases concreta:** Son clases que pueden ser instanciadas, es decir, pueden tener objetos.

Listing 1: Clase abstracta

```
1 public class HelloWorld {
2     public static void main(String[] args) {
3         System.out.println("Hello, World!");
4     }
5 }
```

3. Los objetos colaboran

En POO lo mejor ir encontrando relaciones a partir del comportamiento esperado en los escenarios.

Lo más importante en POO no es tanto la estructura de relacionamiento de los objetos, sino el comportamiento. Por lo tanto, deberíamos ocuparnos de, siguiendo el diseño por contrato, determinar los contratos de cada objeto a implementar (precondicionesa y postcondiciones, invariantes, etc.)

3.1. Relaciones entre objetos

3.1.1. Los objetos interactúan

Todo mensaje tiene un objeto **cliente** y uno **receptor**. Pero los objetos no tienen por qué ser siempre servidores o clientes, sino que su condición cambia según el mensaje. Por ejemplo, un objeto puede ser el receptor de un mensaje y, para poder responder ese mensaje envía a su vez una solicitud a objeto, en cuyo caso se convierte en cliente.

3.1.2. Yendo a lo práctico... o casi

En POO lo mejor no es pensar en relaciones entre objetos per se, sino más bien ir encontrando estas relaciones a partir del **comportamiento** esperado en los escenarios. De esa manera, no tenemos relaciones tan complejas sin una razón para ello.

3.1.3. ¿Cómo implementamos esto?

En POO lo importante no es tanto la estructura de relacionamiento de los objetos, sino el comportamiento.

Por lo tanto, deberíamos ocuparnos de, siguiendo el diseño por contrato, determinar los contratos de cada objeto a implementar.

3.2. Conceptualizando

3.2.1. Dependencia y asociación

Definición 3.1. Dependencia Un objeto depende de otro cuando debe conocerlo para poder enviarle un mensaje. Corolario: Todo objeto cliente depende de su servidor.

Esta dependencia puede venir dada de tres maneras:

- Porque el objeto servidor se envía como argumento.
- Porque el objeto servidor se obtiene como respuesta al envío de un mensaje a otro objeto.
- Porque el objeto cliente tiene una referencia al servidor. (Esto es asociación)

Definición 3.2. Asociación Una asociación es una forma de dependencia en la que el objeto cliente tiene almacenada una referencia al objeto servidor.

Tanto las dependencias más débiles como las asociaciones tienen un sentido: decimos que el cliente depende del servidor, y no al revés. Para un mismo mensaje, lo habitual es que las dependencias sean unidireccionales.

Ejemplo 3.1. Ejemplos de dependencia y asociación (preguntar si es correcto)

- Si tengo una clase A donde los atributos usan una clase B. Esto sería asociación por que el objeto cliente tiene almacenado una referencia al objeto servidor?
- Si tengo una clase A donde sus métodos usan una clase B. Esto sería dependencia por que el objeto cliente, depende objeto servidor?

Sí, tienes razón en ambos casos.

Si tienes una clase A donde los atributos usan una clase B, esto sería una asociación porque el objeto cliente (clase A) tiene almacenada una referencia al objeto servidor (clase B). Esto significa que el objeto cliente puede interactuar con el objeto servidor a través de esta referencia.

Si tienes una clase A donde sus métodos usan una clase B, esto sería una dependencia porque el objeto cliente (clase A) depende del objeto servidor (clase B). Esto significa que cualquier cambio en la definición de la clase B podría afectar a la clase A. Sin embargo, esta dependencia puede ser débil si se hace que el código dependa de interfaces o clases abstractas en lugar de clases concretas.

3.2.2. Interludio metodológico: ¿en qué orden probar cuando tenemos objetos que deben construirse luego?

Definición 3.3. (Digresión) Conceptualmente, si una prueba está probando el comportamiento de más de una clase, o incluso más de una responsabilidad de un objeto, es una prueba de integración, no una prueba unitaria.

3.2.3. Relaciones entre clases

3.2.4. Colaboración por delegación

Uno de los objetivos centrales de la POO era permitir la programación en base a componentes.

La manera más sencilla de vinculación es la simple **asociación** entre objetos. Esta vinculación nos sirve para la delegación de comportamiento.

3.2.5. Programación por diferencia: herencia

Una forma de vinculación de componentes es la herencia, o también relación de generalización-especialización.

Definición 3.4. (Herencia) La herencia es una relación entre clases, por la cual se define que una clase puede ser un caso particular de otra. A la clase más general la llamamos madre y a la más patricular hija.

Corolario: Cuando hay herencia, todas las instancias de la clase hija son también instancias de la clase madre.

Definición 3.5. (programación por diferencia) Programamos por diferencia cuando indicamos que parte de la implementación de un objeto está definida en otro objeto, y por lo tanto sólo implementamos las diferencias específicas.

Como regla, entonces, conviene hacer siempre este test a una relación para ver si aplicar o no la herencia: *¿necesitamos reutilizar la interfaz de una clase tal como está en otra clase, sin que nada me sobre?* Si es así, puedo usar herencia, haciendo que la clase que va a reutilizar sea hija de la clase que provee el código que nos interesa. Si no, conviene reutilizar por delegación.

Dejar la herencia solamente para aquellos casos en que la clase madre tenga una interfaz contenida en la interfaz de la clase hija. Es decir, que la clase madre no tenga métodos que sobren en la clase hija.

NOTA: La herencia se puede usar en cualquier tipo de clase pero comumente se nota más, su uso, en las clases abstractas y en las interfaces.

3.2.6. Programación por diferencia: delegación de comportamiento

Definición 3.6. Delegación de comportamiento Cuando deseamos programar por diferencia en JavaScript recurrimos a la delegación de comportamiento, por la cual un objeto delega su comportamiento en otro objeto prototípico sin necesidad de repetir las definiciones de los métodos.

3.2.7. Redefinición

Definición 3.7. (Redefinición) La redefinición existe para definir un mismo comportamiento en una clase derivada, para el mismo mensaje de la clase base.

Por lo tanto, la semántica o significado del mensaje se debe mantener. Si así no fuera, conviene definir un método diferente, con nombre diferente.

En la refinición es importante que se mantenga la firma del método, las precondiciones y postcondiciones pueden variar ligeralmente.

3.2.8. Clases abstractas

Definición 3.8. (Clase abstracta) Una clase es abstracta cuando no puede tener instancias en forma directa, habitualmente debido a que sus clases descendientes cubren todos los casos posibles.

Cuando queramos indicar una clase que no es abstracta (que puede tener instancias) la llamaremos clase concreta.

3.2.9. Métodos abstractos

Definición 3.9. (Método abstracto) Un método es abstracto cuando no lo implementamos en una clase, pero sí deseamos que todas las clases descendientes puedan entender el mensaje.

Cuando queramos indicar un método que no es abstracto (que tiene implementación) lo llamaremos método concreto.

Como corolario, un método abstracto no va a tener implementación, sino que sólo se define su firma, para que las clases descendientes lo implementen en base a ella.

3.3. Cuestiones de implementación

3.3.1. Delegación y herencia en Smalltalk y Java

La delegación tiene una implementación muy directa en ambos lenguajes: basta recibir al objeto servidor como parámetro o como respuesta a un mensaje, o simplemente tener una referencia a él, para poder delegar.

3.3.2. Visibilidad

Hay tres niveles de visibilidad:

- **Pública**: el método, atributo o clase en cuestión se puede utilizar en cualquier parte del programa.
- **Privada**: el método, atributo o clase en cuestión se puede utilizar solamente dentro de su clase.
- **Protegida**: el método, atributo o clase en cuestión se puede utilizar solamente dentro de su clase o una clase descendiente de ella.
- **Paquete**: el método, atributo o clase en cuestión se puede utilizar solamente dentro de su paquete.

En Smalltalk hay menos control de la visibilidad por parte del programador: todos las clases y los métodos son públicos y los atributos son todos protegidos.

- **Smalltalk**: Todos los métodos son publicos, todos los atributos son protegidos (aunque se recomienda considerarlos privados).
- **Java**: Los métodos y atributos pueden ser publicos, privados o protegidos.

3.3.3. Métodos y clases abstractos

Smalltalk, las clases abstractas son aquellas que tienen algún método abstracto, sin que tengamos que declararlo en forma explícit

Smalltalk, como recién dijimos, maneja la cuestión en tiempo de ejecución, lanzando excepciones. En Java, en cambio, es el compilador el que chequea que no se llame un método abstracto o se intente instanciar una clase abstracta.

3.3.4. Constructores en situaciones de herencia y asociación

3.3.5. Herencia y compatibilidad en lenguajes con comprobación de tipos estática

3.4. Modularización

Principios de diseño:

- clases
- Paquetes
- Métodos

3.4.1. Cohesión y acoplamiento

Definición 3.10. (Cohesión) La cohesión es un principio de diseño que indica que los elementos de un módulo deben estar relacionados entre sí, y que el módulo debe tener una única responsabilidad.

Para mantener compresibles, mantenibles y reutilizables a los módulos.

3.5. Pricipios SOLID

3.5.1. Única responsabilidad (SRP)

Tiene que ver con la cohesión.

Cada clase o método debe tener una única responsabilidad, y por lo tanto una única razón para cambiar.

Por ejemplo: si tenemos una clase que se encarga de la lógica de negocio y de la persistencia (guardar), esta clase tiene dos responsabilidades y por lo tanto

dos razones para cambiar.

3.5.2. Abierto-cerrado (OCP)

"Las clases tienen que estar cerradas para modificación, pero abiertas para reutilización".

- No modificar las clases existentes
- Extenderlas o adaptarlas por herencia o delegación.

3.5.3. Sustitución de Liskov (LSP)

En los clientes, debemos poder sustituir un tipo por sus subtipos.
Relación es .[es] un"La subclase sea un subconjunto de la clase. Las clase base no deben tener comportamiento que dependan de las clases derivadas.

3.5.4. Segregación de interfaces (ISP)

3.5.5. Inversión de dependencias (DIP)

4. frameworks xUnit

4.1. La necesidad de herramientas

Definición 4.1. (framework) Un entorno de trabajo (inglés framework) o marco de trabajo es un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.

4.2. Primer intento: SUnit

1. Para escribir una clase de pruebas, debemos hacer que la misma derive de la clase TestCase.
2. Cada método de prueba debe poder tener cualquier nombre, a condición de que empiece con la palabra test.
3. Al escribir cada método, nos podemos ayudar con el comportamiento provisto en TestCase, que incluye algunos métodos de nombres *assert, deny, should, etc.*
4. SUnit se ocupa de que todos los métodos test escritos hasta el momento en una clase derivada de TestCase se puedan ejecutar a la vez.

5. Al correr un conjunto de pruebas en SUnit, una interfaz gráfica nos indica con color verde o rojo qué pruebas pasaron y cuáles no.

Definición 4.2. (Digresión: SUnit es un framework. Case) La diferencia entre biblioteca y framework es muy sutil para muchos profesionales. Sin embargo, una biblioteca brinda servicios a través de clases y funciones listas para usar, mientras que un framework brinda servicios a través de comportamientos por defecto que se pueden cambiar o ampliar a través de clases provistas por el programador.

Dicho de otra manera, una biblioteca está lista para ser usada mediante la invocación de sus servicios, mientras que un framework es un programa que invoca a los comportamientos definidos por el programador.

4.3. Herramientas de cobertura

Herramienta que mostrase si el código era cubierto o no por las pruebas

5. Polimorfismo: los objetos se comportan a su manera

5.1. ¿Qué es polimorfismo?

El polimorfismo es la capacidad que tienen distintos objetos de responder de maneras diferentes a un mismo mensaje

Definición 5.1. Polimorfismo Llamamos polimorfismo a la posibilidad de que distintos objetos respondan de manera diferente ante la llegada del mismo mensaje.

El polimorfismo es la capacidad que tienen distintos objetos de responder de maneras diferentes a un mismo mensaje.

Definición 5.2. (mensaje polimorfo) Un mensaje es polimorfo cuando la respuesta al mismo puede ser diferente en función del objeto receptor.

5.2. Polimorfismo y herencia: ¿realmente deben ir juntos?

El vínculo entre polimorfismo y herencia es, simplemente, una cuestión de implementación

5.3. Métodos abstractos y comprobación estática

En definitiva, los métodos abstractos sirven, en los lenguajes de comprobación dinámica, como un medio para obligar a las clases descendientes a implementar ese comportamiento. En los lenguajes de comprobación estática, hay ocasiones en que no podemos sino definir ciertos métodos abstractos si queremos que funcione el polimorfismo.

5.4. Polimorfismo sin herencia en lenguajes de comprobación estática: interfaces

Polimorfismo sin herencia. Java es uno de ellos, y el mecanismo que utiliza se denomina interfaces.

Otra manera de ver a una interfaz es simplemente como un conjunto de métodos abstractos

6. Refactorización

El problema es que, con cada cambio que le hacemos al software, éste se degrada en cuanto a calidad del código.

Definición 6.1. (Entropía) Entropía del software: una degradación de la calidad inexorablemente continua y creciente.

6.1. Refactorización al rescate

Definición 6.2. (refactorización como técnica) Refactorización es una técnica que busca mejorar la calidad del código con vistas a facilitar su mantenimiento, pero sin alterar el comportamiento observable del mismo.

Definición 6.3. (refactorización como un cambio) Una refactorización es un cambio realizado sobre el código de un sistema de software de modo tal que no se afecte su comportamiento observable y con el sólo fin de mejorar la legibilidad, la facilidad de comprensión, la extensibilidad u otros atributos de calidad interna con vistas al mantenimiento.

Asegurar la preservacipon preservación del comportamiento luego de una refactorización Los métodos analíticos, y en especial los de análisis estático, son los que utilizan las herramientas de refactorización automática.

Pero no toda refactorización es posible de resolver por métodos analíticos ni automatizable. Por eso se suelen usar pruebas para asegurar la preservación del comportamiento. En definitiva, para permitir refactorizaciones más seguras y confiables, una buena táctica es trabajar con pruebas unitarias automatizadas, escritas antes de refactorizar, y correrlas después para asegurarnos que nuestro programa sigue funcionando tal como lo hacía antes del cambio.

6.2. A qué llamamos observable

El conjunto de entradas y el conjunto de salidas debería ser el mismo antes y después de una refactorización.

Lo que busca una refactorización es que se preserve lo que un usuario espera del sistema, que es lo que habitualmente llamamos requerimientos.

Definición 6.4. (refactorización (basada en requerimientos)) Una refactorización es un cambio realizado sobre el código de un sistema de software, con

el sólo fin de mejorar la calidad interna con vistas al mantenimiento, de modo tal que, luego de esa transformación, se sigan cumpliendo los requerimientos de la aplicación

6.3. Refactorización como parte de TD

La práctica de refactorización permanente nos ayuda a que el diseño vaya evolucionando conforme los cambios van surgiendo. ver libro fontenla pag 135.

7. Java

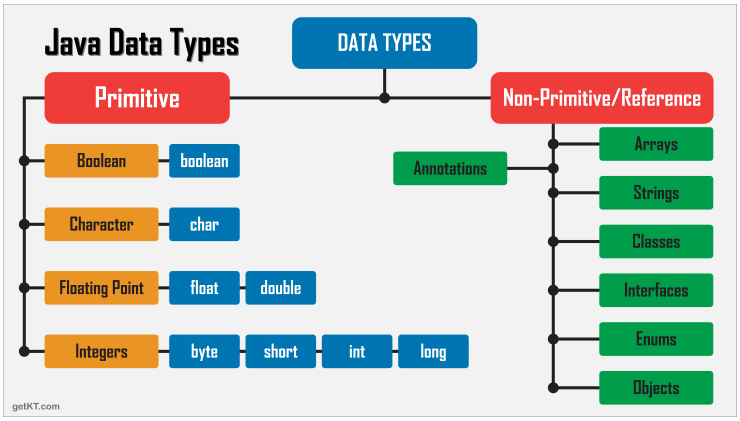

Figura 2: Tipos de datos en Java [4]

7.1. Tipos de clases en JAVA

- **Clases**: Son las clases comunes y corrientes, que se pueden instanciar.
- **Clases Abstractas**: Son clases que no se pueden instanciar, pero que sirven para heredar.
- **Interfaces**: Son clases que no se pueden instanciar, pero que sirven para heredar.
- **Clases Anónimas**: Son clases que no tienen nombre, y se usan para sobreescribir métodos.
- **Clases finales**: Son clases que no se pueden heredar.

7.2. Relaciones entre clases

Relaciones en UML: [1]

- **Asociación**
  - **Agregación**
  - **Composición**
- **Herencia/Generalización**
- **Dependencia**
- **Realización/Implementación**

1. **Asociación:** Indica que una propiedad de una clase contiene una referencia a una instancia (o instancias) de otra clase.

   La asociación es la relación más utilizada entre una clase y otra clase, lo que significa que existe una conexión entre un tipo de objeto y otro tipo de objeto. Las combinaciones y agregaciones también pertenecen a las relaciones asociativas , pero las relaciones entre clases de afiliaciones son más débiles que las otras dos.

   Hay cuatro tipos de asociaciones : asociaciones bidireccionales , asociaciones unidireccionales , autoasociación y asociaciones de números múltiples.

   Por ejemplo: coches y conductores, un coche corresponde a un conductor en particular y un conductor puede conducir varios coches. El * en al fugura significa 0 o más.

Figura 3: Asociación

   a) **Agregación:** La relación entre el todo y la parte, y el todo y la parte se pueden separar. Las relaciones agregadas también representan la relación entre el todo y una parte de la clase, los objetos miembros son parte del objeto general, pero el objeto miembro puede existir independientemente del objeto general.Tiempo de vida *independiente*.

   Por ejemplo, los conductores de autobús y la ropa y los sombreros de trabajo son parte de la relación general, pero se pueden separar. La

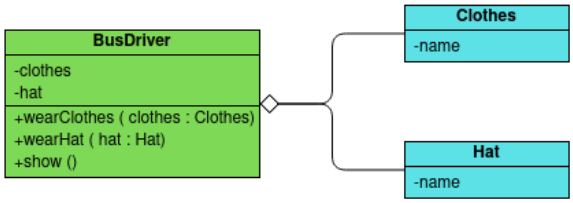

Figura 4: Agregación

ropa de trabajo y los sombreros se pueden usar en otros conductores. Los conductores de autobuses también pueden usar otra ropa de trabajo y sombreros.

Otro ejemplo puede ser el de un auto y sus ruedas, el auto depende si o si de tener cuatro ruedas.

b) **Composición:** La relación entre el todo y la parte, pero el todo y la parte no se pueden separar.

La relación de combinación representa la relación entre el todo y la parte de la clase, y el total y la parte tienen una duración constante. Una vez que el objeto general no existe, algunos de los objetos no existirán y todos morirán en la misma vida. Tiempo de vida *dependiente*.

Por ejemplo, una persona está compuesta por una cabeza y un cuerpo. Los dos son inseparables y coexisten.

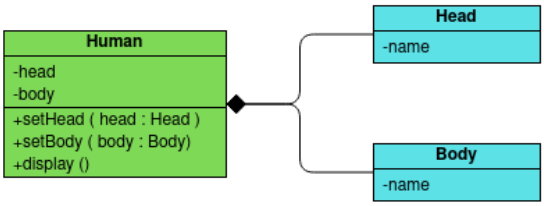

Figura 5: Composición

2. **Herencia/Generalización:** La herencia también se denomina generalización y se utiliza para describir la relación entre las clases padre e hijo. Una clase principal también se denomina clase base y una subclase también se denomina clase derivada.

En la relación de herencia, la subclase hereda todas las funciones de la clase principal y la clase principal tiene todos los atributos, métodos y subclases. Las subclases contienen información adicional además de la misma información que la clase principal.
a

Por ejemplo: autobuses, taxis y automóviles son automóviles, todos tienen nombres y todos pueden estar en la carretera.

3. **Dependencia:** Suponga que un cambio en la clase A provoca un cambio en la clase B, luego diga que la clase B depende de la clase A. En la mayoría de los casos, las dependencias se reflejan en los métodos de una clase que utilizan el objeto de otra clase como parámetro . Una relación de dependencia es una relación de "uso". Un cambio en una cosa en particular puede afectar a otras cosas que la usan, y usar una dependencia cuando es necesario indicar que una cosa usa otra.

   Por ejemplo: El auto depende de la gasolina. Si no hay gasolina, el automóvil no podrá conducir.

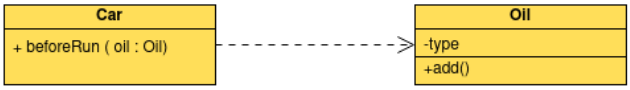

Figura 6: Dependencia

4. **Realización/Implementación:** La implementación (Implementación) se utiliza principalmente para especificar la relación entre las interfaces y las clases de implementación.

   Una interfaz (incluida una clase abstracta) es una colección de métodos. En una relación de implementación, una clase implementa una interfaz y los métodos de la clase implementan todos los métodos de la declaración de la interfaz.

   Por ejemplo: los automóviles y los barcos son vehículos, y el vehículo es solo un concepto abstracto de una herramienta móvil, y el barco y el vehículo realizan las funciones móviles específicas.

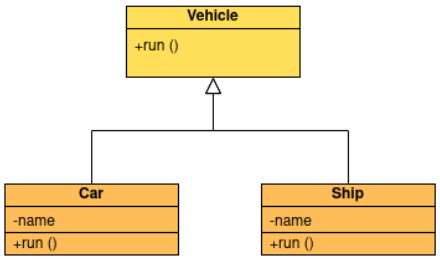

Figura 7: Realización/Implementación

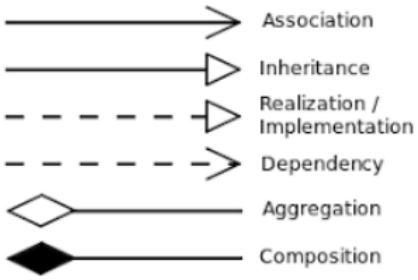

Figura 8: UML Relaciones entre clases

- **Asociación:** Es una relación de tipo **"tiene un"** entre clases, donde una clase tiene una referencia a otra clase. Por ejemplo, si tenemos una clase Persona y una clase Coche, podríamos decir que una persona tiene un coche, y representar esta relación con una asociación entre las clases Persona y Coche. Esto se prodria notar en el código como una variable de instancia de tipo Coche en la clase Persona o como un atributo.
  - **Agregación:** Es un tipo especial de asociación que representa una relación de tipo **"parte de" ("contine" o "hace referencia")** entre clases, donde una clase es parte de otra clase. A diferencia de la asociación, en la agregación las partes pueden existir independientemente del todo. Por ejemplo, si tenemos una clase Equipo y una clase Jugador, podríamos decir que un jugador es parte de un equipo, y representar esta relación con una agregación entre las clases Equipo y Jugador.
  - **Composición:** Es otro tipo especial de asociación que también representa una relación de tipo **"parte de" ("contine" o "hace referencia")** entre clases, pero en este caso las partes no pueden existir independientemente del todo. Por ejemplo, si tenemos una clase Casa y una clase Habitacion, podríamos decir que una habitación es parte de una casa, y representar esta relación con una composición entre

las clases Casa y Habitacion.

- **Herencia:** Es una relación de tipo **"es un"** entre clases, donde una subclase hereda las propiedades y comportamientos de su superclase y puede agregar o modificar propiedades y comportamientos propios. Por ejemplo, si tenemos una clase Vehiculo con propiedades como marca, modelo y color, y comportamientos como acelerar y frenar, podríamos crear una subclase Coche que herede estas propiedades y comportamientos de la clase Vehiculo, y agregar propiedades específicas como numeroDePuertas y comportamientos específicos como abrirTechoSolar.
- **Dependencia:** Es una relación entre clases donde una clase depende de otra clase para su funcionamiento. Por ejemplo, si tenemos una clase Calculadora y una clase Operacion, podríamos decir que la calculadora depende de la operación para realizar cálculos, y representar esta relación con una dependencia entre las clases Calculadora y Operacion.
- **Realización/Implementación:** Es una relación de tipo **"se comporta como"** entre clases, donde una clase implementa una interfaz y debe proporcionar implementaciones para todos los métodos definidos en la interfaz. Por ejemplo, si tenemos una interfaz Volador con métodos como despegar y aterrizar, podríamos tener clases como Avion y Pajaro que implementen esta interfaz y proporcionen implementaciones para los métodos despegar y aterrizar.

7.3. Interface

Interfaces: [3].

- Son una **colección de métodos abstractos** con propiedades (atributos) **constantes**.
- Una interfaz **solamente puede extender o implemtar otras interfaces** (la cabtidad que quiera).
- Da a conocer qué se debe hacer (métodos) **pero sin mostrar su implemtación** (solo puede tener métodos abstractos).
- Solo puede tener **métodos** con **métodos públicos** (no pueden ser protected o private).
- Solo puede tener "variables"public static final (o sea constantes).
- La palabra tener **abstract** en la definición de métodos no es obligatoria.
- Generalmente las interfaces indican el **PUEDE HACER** de un objeto.

Figura 9: Tipos de lenguajes

7.4. Tipos de lenguajes

- **JAVA:** es un lenguaje con combinación estática, lleva a que una gran cantidad de error en el código surjan en **tiempo de compilación**.
- **SMALLTALK:** es un lenguaje con combinación dinámica, lleva a que una gran cantidad de error en el código surjan en **tiempo de ejecución**.

La comprobación estática de tipos trata de asegurar que los programas no produzcan errores durante su ejecución.link

7.5. Conceptos de la clases

Ocultamiento de información: Para evitar que el objeto se torne inconsistente.

Encapsulamiento: Mantener el estado consistente del objeto.

- Atributo de instancia: Son los atributos que tiene cada objeto.
- Atributo de clase: Son los atributos que tienen todas las instancias de la clase.

Definición 7.1. (Desnormalización) Es cuando se repite información en distintas clases. Desnormalización del estado: Es cuando se repite información en distintos atributos de una clase. O también tener atributos que se pueden calcular a partir de otros atributos. Por ejemplo, tener el atributo edad y fecha de nacimiento.

Definición 7.2. (Lazy loading) Es cuando se carga la información de un objeto cuando se la necesita. Por ejemplo, cuando se tiene un atributo que es una lista de objetos, y se quiere acceder a un objeto de esa lista, se carga la lista completa.

Referencias

[1] relación de clases. En: 1 (). URL: https://blog.visual-paradigm.com/es/what-are-the-six-types-of-relationships-in-uml-class-diagrams/#:~:text=Hay%20seis%20tipos%20principales%20de,%2C%20agregaci%C3%B3n%2C%20asociaci%C3%B3n%20y%20dependencia..

[2] Definicion de Entidad. En: 1 (). URL: https://platzi.com/clases/1566-bd/20197-entidades-y-atributos/.

[3] Interfacez. En: 1 (). URL: https://www.youtube.com/watch?v=hfwtzjOhvKk.

[4] Tipos de datos primitivos JAVA. En: 1 (). URL: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html.