RESUMEN OBJETOS

En un software de objetos, un conjunto de objetos que colaboran enviándose **mensajes**. Todo computo ocurre “dentro” de los objetos.

El éxito se obtiene cuando se agrega funcionalidad y modificaciones sin que el sistema se “entere”

En un programa construido con objetos:

- No hay “main”.
- Se describen clases
- Los objetos se crean dinámicamente durante la ejecución del programa
- Una jerarquia de clases no indica lo mismo que la jerarquia top-down
- No hay un objeto más importante que otros. El comienzo de una aplicación depende del flujo de control, de decisiones del desarrollador, del tipo de interacción, etc

Sistema compuesto por **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.

Cuando en el objeto a de la clase A ejecutamos el método m que tiene el siguiente código:

e = new Estudiante ()

Creamos un objeto de la clase Estudiante, que será conocido por a mediante la variable e

Un **objetos** es una abstracción de una entidad del dominio del problema. Tiene:

- Identidad.
    - para distinguir un objeto de otro
- Conocimiento.
    - En base a sus relaciones con otros objetos y su estado interno
- Comportamiento.
    - Conjunto de mensajes que un objeto sabe responder

Los objetos poseen **estructura** y **comportamiento**. El comportamiento indica que mensajes pueden entender y la estructura está especificada por variables de instancia

El estado interno de un objeto determina su conocimiento. El **estado interno** se mantiene en las variables de instancia ,anteriormente mencionadas, del objeto (son privadas). En general las variables son REFERENCIAS (punteros) a otros objetos con los cuales el objeto colabora. Algunas pueden ser atributos básicos.

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**.

Las variables de instancia están encapsuladas y solo son accesibles por los métodos que ejecuta el objeto. Para acceder una variable del objeto A, necesitamos enviarle un mensaje. Cuando un objeto recibe un **mensaje**, ejecuta el **método** que corresponde al mensaje recibido. El que envía el mensaje **delega** en el receptor la manera de resolverlo, que es privada del objeto.

Un **método** puede:

- modificar el estado interno del objeto
- colabora con otros objetos
- retornar y terminar

Para poder enviarle un mensaje a un objeto, hay que conocerlo. Se establece una ligadura (**binding**) entre un nombre y un objeto.

Se puede identificar tres formas de conocimiento o tipos de relaciones entre objetos.

1. Conocimiento Interno: Variables de instancia.
2. Conocimiento Externo: Parámetros.
3. Conocimiento Temporal: Variables temporales.

Además existe una cuarta forma de conocimiento especial: las pseudo-variables (como “this” o “self”).

El **encapsulamiento** es la cualidad de los objetos de ocultar los detalles de implementación y su estado interno del mundo exterior.

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.

Las clases se especifican por medio de un **nombre**, el **estado** o estructura interna que tendrán sus instancias y los **métodos** asociados que definen el comportamiento.

Se usa **new** para instanciar nuevos objetos. Para que un objeto esté listo para llevar a cabo sus responsabilidades hace falta **inicializarlo** (darles valor a sus variables). Esto se puede hacer mediante los constructores.

Una clase representa un concepto en el dominio de problema (o de la solución). Cuando tienen comportamiento en común hay una **Subclasificación** (subclases)

- Se reúne el comportamiento y la estructura común en una clase, la cual cumplirá el rol de **superclase**.
- Se conforma una jerarquía de clases
    - Luego otras clases pueden cumplir el rol de subclases, heredando ese comportamiento y estructura en común.
- Debe cumplir la relación es-un (y a veces algo mas).

La **herencia** es el mecanismo por el cual las subclases reutilizan el comportamiento y estructura de su superclase.

Toda relación de herencia implica:

- Herencia de comportamiento
    - Una subclase hereda todos los métodos definidos en su superclase.
    - Las subclases pueden redefinir el comportamiento de su superclase.
- Herencia de estructura
    - No hay forma de restringirla.
    - No es posible redefinir el nombre de un atributo que se hereda.

Al enviarse un mensaje a un objeto:

1) Se determina cuál es la clase del objeto.
2) Se busca el método para responder al envío del mensaje en la jerarquía, comenzando por la clase del objeto, y subiendo por las superclases hasta llegar a la clase raíz (Object)

Este proceso se denomina **method lookup.**

Una **clase abstracta** es una clase que no puede ser instanciada.

- Se diseña sólo como clase padre de la cual derivan subclases.
- Representan conceptos o entidades abstractas.
- Sirven para factorizar comportamiento común.
- Usualmente, tiene partes incompletas.
    - Las subclases completan las piezas faltantes, o agregan variaciones a las partes existentes.

Dos o más objetos son **polimórficos** con respecto a un mensaje, si todos pueden entender ese mensaje, aún cuando cada uno lo haga de un modo diferente. Los objetos pueden ser de distintas clases. El polimorfismo permite que haya un código mas compacto.

Se consigue:

- Código genérico
- Objetos desacoplados
- Objetos intercambiables
- Objetos reutilizables
- Programar por protocolo, no por implementación

Importa el QUE HACER y no QUIEN LO HACE.

El **binding dinamico** es un mecanismo por el cual se decide que método responderá al mensaje recién cuando se lo recibe, en tiempo de ejecución.

Basicamente nos permite evitar el chequeo explicito de el tipo al que pertenece un objeto apuntado por una variable dejando que dicho objeto se encargue de decidir que método se ejecuta (usando el algoritmo usual de "lookup").

THIS siempre apunta al objeto que recibe el mensaje.

Un **tipo** en un lenguaje orientado a objetos es un conjunto de firmas de métodos. Una interfaz es similar a una clase abstracta, solo que permite definir tipos desacoplándolos de sus implementaciones. Sirve cuando distintos objetos saben responder al mismo mensaje. Las **interfaces** proveen un mayor nivel de abstracción.

El **testing** forma parte del programa. Uno escribe test para encontrar errores. Estos errores pueden ser:

- El programa no hace lo que tiene que hacer.
- Hace algo mal.
- El programa "explota".

Hacer testeo de unidad en objetos es asegurarse de que el programa hace lo que se espera, lo hace como se espera, y no falla. Es importante testear temprano, y tanto como sea el riesgo del artefacto a testear.

Hay distintos tipos de test pero ahora nos importan:

- Tests funcionales:
    - Verifican que el programa hace lo que tiene que hacer.
- Tests de unidad:
    - 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.
- Tests automatizados
    - Software para guiar la ejecución de los tests y controlar los resultados. Básicamente programas que testean programas.
    - Son "parte del software" (y un indicador de su calidad).

Los tests que escribimos con JUnit son tests funcionales, automatizados, de unidad.

- Hay una clase de test por cada clase a testear
- Un método que prepara lo que necesitan los tests (BeforeEach) 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)

Hay independencia entre test. No puedo asumir que otro test se ejecutó antes o se ejecutará después del que estoy escribiendo.

La **estrategia general** para testear es pensar que podría variar (que valores puede tomar) y que pueda causar un error o falla. Para eso usamos dos estrategias:

- Escribo tests de **particiones equivalentes** en donde identifico particiones, y elijo valores representativos dentro y fuera de cada partición para usarlos en los tests.
    - Conjunto de casos que prueban lo mismo o revelan el mismo bug.
    - Si se trata de casos en un conjunto, tomo un caso que pertenezca al conjunto y uno que no.

    - Si se trata de valores en un rango, tomo un caso dentro y uno por fuera en cada lado del rango
- Escribo tests con **valores de borde** en donde identifico los bordes en las particiones de equivalencia y uso valores en esos bordes para mis tests.
    - Los errores ocurren con frecuencia en los limites 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.

Escribir tests de unidad (con JUnit) es parte “programar” y nos ayuda a entender que se espera de nuestros objetos.

Las **colecciones** tienen punteros a los objetos. Si modifico un objeto, alcanza a todos los que tenia puntero. Se usan cuando las relaciones son 1 a muchos.

Todos los lenguajes OO ofrecen librerías de colecciones. Las colecciones admiten, generalmente, contenido heterogéneo en términos de clase, pero homogéneo en términos de comportamiento, es decir, en las colecciones puede haber objetos de distintas clases, pero tienen que ser polimórficos (mismo tipo), para que puedan entender el mismo mensaje.

Siembre (o casi) que tenemos colecciones repetimos las **mismas operaciones.** Un ejemplo de esto, es recorrer colecciones. En JAVA el mensaje iterator() que entienden las colecciones retorna un objeto que abstrae a forma en la que se recorre la colección. Para iterar una colección con un iterador, le envío el mensaje iterator() sin importar la clase de colección con la que trato.

- 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

<u>IMPORTANTE</u>

- Nunca modifico una colección que obtuve de otro objeto.
- Cada objeto es responsable de mantener los invariantes de sus colecciones.
- Solo el dueño de la colección puede modificarla.
- Recordar que una colección puede cambiar luego de que la obtengo

Las **Expresiones Lambda** son métodos anónimos.

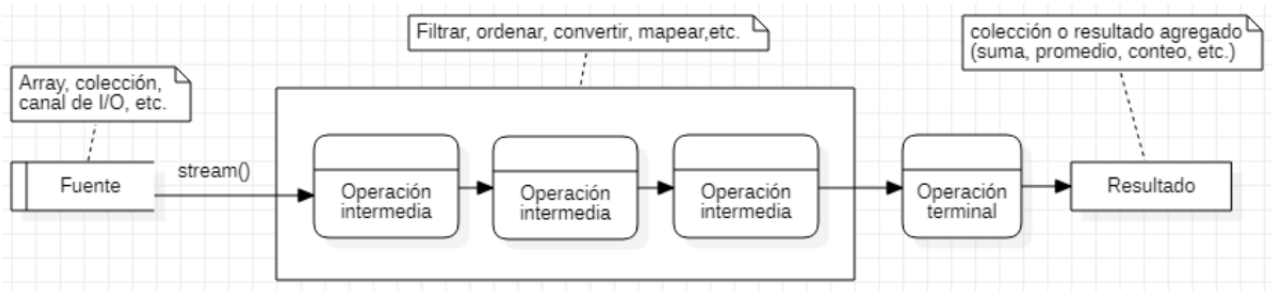
Los **streams** permiten procesamiento funcional de colecciones, las operaciones se combinan para formar pipelines (tuberías)

Los streams:

- No almacenan sino que proveen acceso a una fuente (colección, canal I/O, etc.)
- Cada operación produce un resultado, pero no modifica la fuente
- Potencialmente sin final
- Consumibles: cada elemento se visita una sola vez

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



Para filtrar una colección en Java, es recomendable utilizar el protocolo de streams mediante **filter**(), que solo "deja pasar" los elementos que cumplen cierto predicado.

map() nos da un stream que transforma cada elemento de entrada aplicando una función que indiquemos. La función de transformación (de mapeo) recibe un elemento del stream y devuelve un objeto

collect() es una operación terminal que nos permite obtener un objeto o colección de objetos a partir de los elementos del stream.

Para llevarse

- Ojo con las colecciones de otro … son una invitación a romper el encapsulamiento.
- Observar la estrategia de diseño de encapsular lo que varia/molesta.
- No queremos reinventar la rueda

Un buen Diseño OO debe cumplir con **alta cohesion** y **bajo acoplamiento**.

Analisis -> investigación del problema y requisitos.

Diseño -> solucion conceptual que satisface los requisitos.

Los **casos de uso** se definen para satisfacer los objetivos de usuario o actores principales. Sirven para encontrar objetivos que requieren los actores principales.

En resumen hay que:

- Identificar los actores principales (y actores secundarios)
- Identificar los objetivos de usuario de cada actor
- Definir los casos de uso

- El diagrama de casos de Uso proporciona información visual concisa del sistema, los actores externos y cómo lo utilizan.

Los DSS (Representación de los Diagramas de Secuencia del Sistema) se derivan de los casos de uso. Permite determinar quien le envia a quien que mensaje y la respuesta que se espera. Aparecen objetos, instancias.

El **modelo del dominio** permite encontrar relaciones entre las clases y determinar atributos. Es útil para representar el dominio del problema en la etapa de Análisis.

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

Consejo:

- Representar problema, no solución.
- Omitir detalles irrelevantes.
- Evitar sinónimos.
- Descubrir conceptos del mundo real.

Estrategias:

- Identificación de frases nominales: encontrar conceptos (y sus atributos) mediante la identificación de los sustantivos en la descripción textual del dominio del problema.
- Utilización de una lista de categorías de clases conceptuales:

Categoría de Clase Conceptual	Ejemplos
Objeto físico o tangible	Libro impreso
Especificación de una cosa	Especificación del producto, descripción
Lugar	--
Transacción	Compra, pago, cancelación
Roles de la gente	cliente
Contenedor de cosas	Catálogo de libros, carrito
Cosas en un contenedor	Libro,
Otros sistemas	--
Hechos	cancelación, venta, pago
Reglas y políticas	Política de cancelación
Registros financieros/laborales	Factura/ Recibo de compra
Manuales, documentos	Reglas de cancelación, cambios de categoría de cliente

Para construir el Modelo del Dominio:

1- Listar los conceptos candidatos
2- Graficarlos en un Modelo del Dominio: Un Modelo del Dominio es una representación visual de las clases conceptuales del mundo real en un dominio de interés.

3- Agregar atributos a los conceptos: 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. (Recuerde relacionar las clases conceptuales con asociaciones, no con atributos)
4- Agregar asociaciones entre conceptos.

En UML, la relación de conocimiento entre objetos o instancias de clases: se modela con una asociación hacia el/los objetos que se conocen, agregando en el final de la asociación nombre (rol) y multiplicidad.

Si algo esta poco tiempo en el sistema, conviene no representar.

El **Modelo del Dominio** debe incluir todas las clases candidatas que identifiquemos a partir de los Casos de Uso y de la Lista de Categorías, luego pueden revisarse en el Diagrama de Clases.

Los **contratos** son una de las formas de describir comportamiento del sistema en forma detallada. Describen pre y post condiciones para las operaciones.

Las secciones de un contrato son:

- 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.
- Postcondiciones: el estado del sistema o de los objetos del Modelo del Dominio, después de que se complete la ejecución de la operación.
    - Describen el estado y cambios del sistema después de ejecutarse la operación, utilizando conceptos del modelo conceptual o del Dominio

Las Heurísticas para Asignación de Responsabilidades (HAR) promueven ambos Alta Cohesión y Bajo Acoplamiento.

La habilidad para asignar responsabilidades es extremadamente importante en el diseño. La asignación de responsabilidades generalmente ocurre durante la creación de diagramas de interacción.

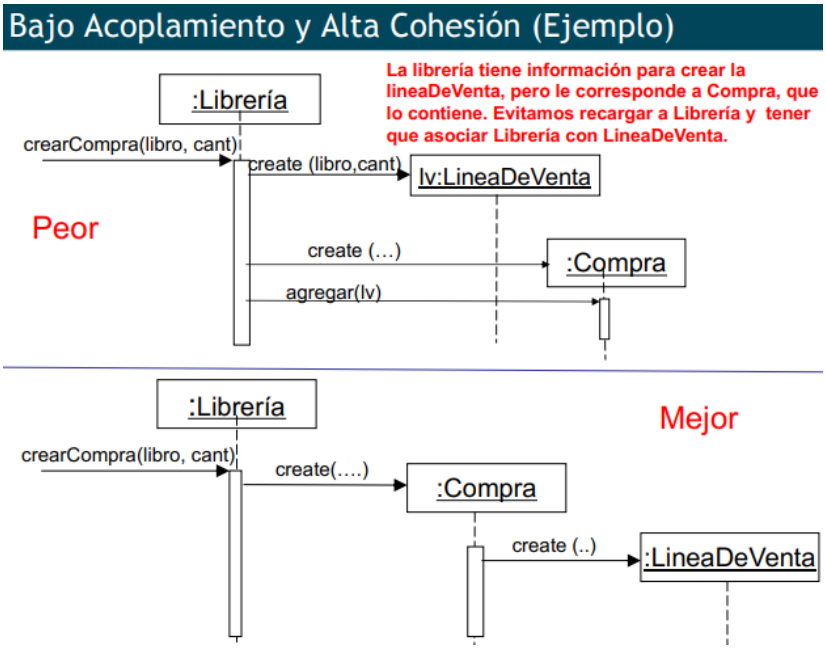
- Experto en Información: es el objeto que sabe unir responsabilidades parciales para lograr un todo. Es la clase que tiene la información necesaria para realizar la responsabilidad.
- Creador: asignar a la clase B la responsabilidad de crear una instancia de la clase A si:
    - B contiene objetos A (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 encontrar una clase que necesite conectarse al objeto creado en alguna situación. Eligiéndolo como el creador se favorece el bajo acoplamiento.

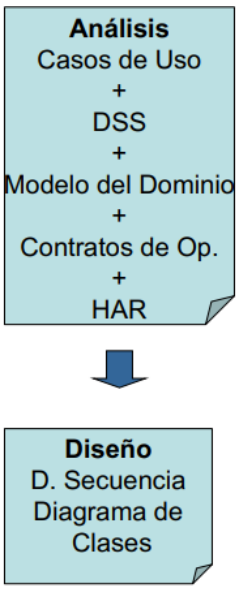
- Controlador: asignar la responsabilidad de manejar eventos del sistema a una clase que representa: El sistema global, dispositivo o subsistema
  La intención del Controlador es encontrar manejadores de los eventos del sistema, sin recargar de responsabilidad a un solo objeto y manteniendo alta cohesión.

El **Bajo Acoplamiento** es 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.

La **Alta Cohesión** es 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.



Del análisis al 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.

Agregando Heurísticas para Asignación de responsabilidades (HAR)

Polimorfismo: 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.

"No hables con extraños": 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)

Heurísticas para Diseño "ágil" Orientado a Objetos (Principios S O L I D)

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 vs. Composición

- Las clases y los objetos creados mediante herencia están **estrechamente acoplados** ya que cambiar algo en la superclase afecta directamente a la/las subclases.
    - La herencia es total. **Caja blanca**.
    - Usualmente debemos redefinir o anular métodos heredados.
    - Los cambios en la superclase se propagan automáticamente a las subclase.
    - Herencia de Estructura vs. Herencia de comportamiento. Una superclase puede tener igual estructura, peo comportamiento distinto.
    - Es útil para extender la funcionalidad del dominio de aplicación.
- 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.
    - Un objeto se compone de forma dinámica. Uno puede cambiar las instancias de los objetos con los que se colabora en momento de ejecución. **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

Si en una **herencia** se niega comportamiento que se hereda/se hereda más de lo que se usa, es mejor que se use **composición**.

Smalltalk

- Lenguaje OO puro – todo es un objeto.
    - Las clases también son objetos, tienen sus propios atributos y métodos. El new, por ejemplo, es un mensaje de clase.
        - Las clases son instancias de una clase también (su metaclase). Por cada clase hay una metaclase (se crean juntas).
    - Está abierto a modificación. Incluso lo que comúnmente conocemos como estructuras de control (como el if, while, etc.) se implementa como envíos de mensajes a objetos.
- Es un lenguaje dinámico, en el que no se indica explícitamente el tipo de las variables.
- Propone una estrategia exploratoria (construccionista) al desarrollo de software.
- El ambiente es tan importante como el lenguaje.
    - Está implementado en Smalltalk.
    - Ricas librerías de clases (fuentes de inspiración y ejemplos).
    - Todo su código fuente disponible y modificable.
    - Tiene su propio compilador, debugger, editor, inspector, perfilador, etc.
    - Es extensible.
- Sintaxis minimalista (con sustento en su foco educativo).
- Fuente de inspiración de casi todo lo que vino después (en OO).

JavaScript (ECMAScript)

- Lenguaje de propósito general.
- Es un lenguaje dinámico, en el que no se indica explícitamente el tipo de las variables.
- Multiparadigma.
- Se adapta a una amplia variedad de estilos de programación.
- Pensado originalmente para scripting de páginas web.

- Con una fuerte adopción en el lado del servidor (NodeJS).
- Basado en objetos (con base en prototipos en lugar de clases).
    - En Javascript no tengo clases.
    - La forma más simple de crear un objeto es mediante la notación literal (estilo JSON).
    - Cada objeto puede tener su propio comportamiento (métodos).
    - Cada objeto hereda comportamiento y estado de su prototipo.
    - Cualquier objeto puede servir como prototipo de otro.
    - Puedo cambiar el prototipo de un objeto (y así su comportamiento y estado).
    - Termino armando cadenas de delegación.
    - Puedo asignar funciones como constructores.