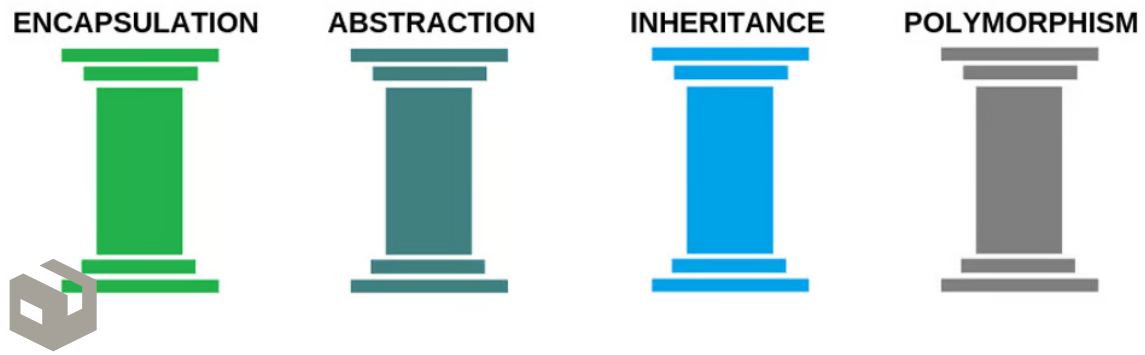


Orientación a objetos 1

- Clases
  - Clase 1 - Intro a objetos

    Esta clase es un repaso de lo visto en taller de programación (nada nuevo)

    - Cómo es un software construido con objetos?

      Un conjunto de objetos que colaboran enviandose mensajes.

      - No hay un objeto “main”
      - Cuando codificamos, programamos clases
      - Jerarquía de clases <> Jerarquía top-down
      - La estructura cambia: en vez de una jerarquía: main, procedures, sub-procedures tenemos una red de “cosas” que se comunican
      - Mientras que la estructura sintáctica es “lineal” el programa en ejecución no lo es.
    - Qué es un objeto?

      Es una abstracción de una entidad del dominio del problema. Ejemplos: persona, producto cuenta bancaria.

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

Un objeto tiene:

- Identidad
- Conocimiento
- Comportamiento

▼ Formas de conocimiento entre objetos

1. Conocimiento interno: variables de instancia
2. Conocimiento externo: parámetros
3. Conocimiento temporal: variables temporales
4. Las pseudo-variables (this o self)

▼ Encapsulamiento

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

▼ Clase 2 - Relaciones objetosas

▼ Relaciones entre objetos

- Un objeto conoce a otro porque:
  - Es su responsabilidad mantener a ese otro objeto en el sistema (por ej, cada cuenta conoce a su titular)
  - Necesita delegarle trabajo (enviarle mensajes)
- Un objeto conoce a otro cuando:
  - Tiene una referencia de ese objeto como variable de instancia
  - Le llega como parámetro
  - Lo crea
  - Lo obtiene enviando mensajes a otros objetos que conoce

▼ This: un objeto que habla solo

- Es una “pseudo-variable”

- Hace referencia al objeto que ejecuta el método. Solo tiene sentido dentro de un método.
- Sirve para descomponer métodos más largos
- En algunos lenguajes (por ej, java)
  - Puede obviarse (en oo1 no)
  - Para desambiguar referencia a las variables de instancia del objeto
- Siempre lo pongo cuando tengo que enviar mensajes a mi mismo

▼ Reutilizar comportamiento repetido

- No repetir código. Si es en el mismo objeto, se hace un método privado para reutilizar el comportamiento repetido. Ahí se usa el this

▼ Igualdad e identidad

▼ Identidad / el operador ==

- Las variables son punteros a objetos
- Más de una variable puede apuntar a un mismo objeto
- Para saber si apuntan al mismo objeto uso ==

▼ Igualdad / el método equals ( )

- Dos objetos pueden ser iguales

```
if  (unAuto.equals(otroAuto)  {
    // los autos son iguales
}
```

Son iguales, pero no son el mismo objeto

▼ Tipos en lenguajes OO

▼ Chequeo de tipos

Java es un lenguaje fuertemente tipado. El compilador chequea la correctitud de nuestro programa respecto a tipos

- Tipo: conjunto de firmas de operaciones / métodos

- Tipos primitivos y tipos de referencias (objetos)
- Las clases definen explícitamente un tipo
- Pero... las clases no son la única forma de definir tipos

▼ Interfaces

- Las interfaces son TIPOS. No son clases, ni clases abstractas, etc.
- Permiten separar tipo e implementación.
- Puedo utilizar interfaces como tipos de variables.
- Las clases deben declarar explicitamente que interfaces utilizan (pueden implementar varias).

▼ Método Lookup

▼ Polimorfismo

- Objetos de distintas clases son polimórficos con respecto a un mensaje, si todos lo entienden, aun cuando cada uno lo implemente de modo diferente

▼ 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
- Cuando dos clases Java implementan una interfaz, se vuelven polimórficas respecto a los métodos de la interfaz

▼ Clase 3 - Herencia

Herencia: mecanismo que permite a una clase “heredar” estructura y comportamiento de otra clase.

▼ Método lookup con herencia

Cuando un método 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...

- Sobre escribir métodos = override

▼ Super

- Pseudo-variable
- Si queremos aprovechar (extender) un método heredado, usamos el super
- “Cambia la forma en que se hace el método lookup”
- Al escribir super.algo() se busca primero en la clase inmediata superior.

▼ Clase abstracta

- Captura comportamiento y estructura que será común a otras clases

▼ Situaciones de uso de herencia

- Implemento métodos abstractos
- Extiendo métodos heredados
- Agrego nuevos métodos

▼ Clases abstractas VS interfaces

- Una clase abstracta “es una clase”
  - Puedo usarla como tipo
  - Puede o no tener métodos abstractos
  - Ofrece implementación de algunos métodos
  - Sus métodos concretos pueden depender de sus métodos abstractos
- Una interfaz define un tipo
  - Sirve como contrato
  - Puede extender a otras

- Se pueden implementar muchas interfaces pero solo se puede heredar de una clase

▼ Clase 4 - Colecciones y UML

▼ Colecciones

Colecciones ≠ listas !!!!, Lista = un tipo de colección

▼ Colecciones como objetos

- La librería de colecciones de java se organiza en términos de:
  - Interfaces
  - Clases abstractas
  - Clases concretas
  - Algoritmos útiles

▼ Algunos tipos de colecciones (interfaces)

- List (java.util.List)
  - Admite duplicados
  - Elementos indexados
- Set (java.util.Set)
  - No admite duplicados
  - Elementos no indexados (ideal para chequear pertenencia)
- Map (java.util.Map)
  - Asocia objetos que actúan como claves a otros que actúan como valores
- Queue (java.util.Queue)
  - Maneja el orden con el que se recuperan los objetos (LIFO, FIFO, por prioridad, etc)

▼ Operaciones sobre colecciones

Siempre (o casi) que tenemos colecciones repetimos las mismas operaciones:

▼ Ordenar respecto algún criterio

- Para esto usamos un objeto comparador.
- Es un objeto de una clase que implementa la interfaz comparator.
- Utilizan un método principal “compare”. que retorna un int

  negativo → el primero es menor
  0 → son iguales
  positivo → el primero es mayor
- Los TreeSet utilizan un comparador para mantenerse ordenados (se espera como parámetro en el constructor)
- Para ordenar List, le enviamos el mensaje sort, con un comparador como parámetro.

```
import java.util.Comparator;
public class ComparadorProductoPrecio implemen
    public int compare (Producto p1, Producto
        return Double.compare(p1.getPrecio(),
    }
}

productos.sort (new ComparadorProductoPrecio (
```

▼ Recorrer y hacer algo con todos los elementos

Utilizan la interfaz iterator.

resumido:

```
for (Producto P: productos)
para cada P, de tipo Producto, sacado de la co
// recorremos una lista de productos
{
    // hacemos algo con cada P
}
```

- ▼ Encontrar un elemento (min,max, DNI = xxx, etc)
- ▼ Filtrar para quedarme con solo algunos elementos
- ▼ Reducir (promedio, suma, etc)

▼ Precaución

- 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 la colección puede cambiar luego de que la obtengo

▼ Streams

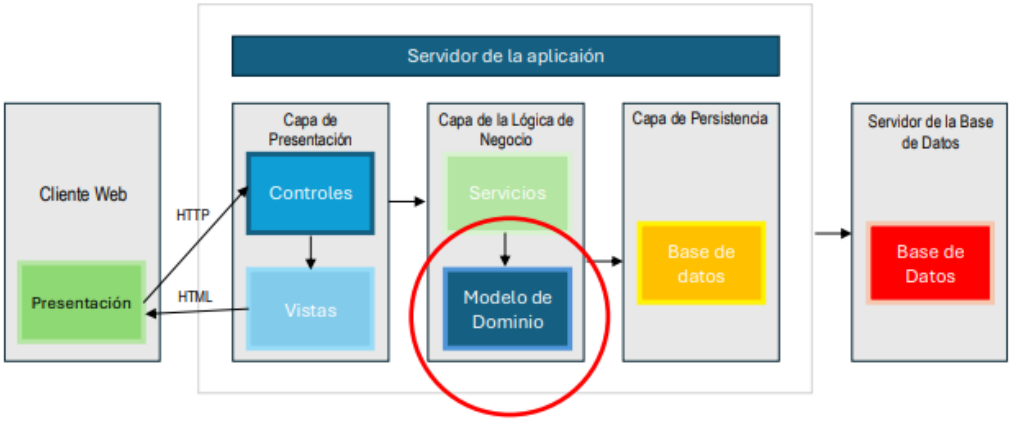
Una banda de data, ver los apuntes en recursos

▼ UML

Una banda de data tmb. Ver PP

▼ Clase 5 - Introducción al análisis y diseño OO parte 1

▼ Modelo del dominio



▼ Clases conceptuales y asociaciones

▼ Identificación de clases conceptuales

Categoría de Clase Conceptual	Ejemplos
Objeto físico o tangible	Libro impreso
Especificación de una cosa	Especificación del producto, descripción
Lugar	Tienda
Transacción	Compra, pago, cancelación
Roles de la gente	cliente
Contenedor de cosas	Catálogo de libros, carrito, Avion
Cosas en un contenedor	Libro, Artículo, Pasajero
Otros sistemas	Sistema de facturación de AFIP
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

▼ Frases nominales

- El cliente selecciona un libro para agregarlo al carrito.
- El carrito agrega el libro y se presenta el precio y la suma del carrito.

▼ Construyendo el modelo del dominio

1. Listar los conceptos (clases y atributos) candidatos
2. Graficarlos en el modelo del dominio
3. Agregar atributos a los conceptos
4. Agregar asociaciones entre los conceptos

▼ Agregar atributos

- Los atributos en un modelo deberían ser, preferiblemente, atributos simples o tipos de datos primitivos. (Boolean, string, números, etc)
- ¿Y si un atributo es una clase conceptual? : En ese caso hablamos de asociaciones.

▼ Asociaciones

LAS RELACIONES ENTRE CLASES CONCEPTUALES DEBEN MODELARSE CON ASOCIACIONES

¿Cómo se que un atributo en una clase?

- 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.

Categoría	Ejemplo
A es una parte física de B	Ala - Avión
A es una parte lógica de B	Etapa de vuelo - Vuelo
A está físicamente contenido en B	Pasajero - Avión
A está lógicamente contenido en B	Libro-Catálogo
A es una descripción para B	EspecificaciónDeProducto - Libro
A es un miembro de B	Piloto - Aerolínea
A usa o maneja a B	Cliente Carrito
A se comunica con B	Cliente- Librería
A está relacionado con la transacción B	Cliente- Pago Cliente- Agregar al carrito
A es una transacción relacionada con otra transacción B	Pago- Compra
A es dueño de B	Cliente- Carrito

▼ Contratos

▼ Secciones en un contrato

- Operación: se detalla el nombre de la operación y los parámetros.
- Pre - condiciones: Suposiciones relevantes del estado del sistema antes o de los objetos del Modelo Del Dominio, antes de ejecución de la operación.
- Post - condiciones: el estado del sistema o de los objetos del Modelo del Dominio, después de que se complete la ejecución de la operación.
    - Describen cambios que deben ocurrir. Por ejemplo: Creación de elementos, creación de asociaciones, destrucción de elementos o asociaciones.
    - Son declarativas: afirmaciones que deben ser verdaderas luego de la

ejecución de la operación.

- Son una forma de describir el comportamiento en un sistema en forma detallada.
  - Ejemplo: contratar servicio por única vez (c: Cliente, fecha: Date, s: Servicio)
    - Pre-condición:
      - la fecha es una fecha válida para el contexto.
    - Post-condición:
      - el cliente posee una contratación por única vez para el servicio s y la fecha indicada.

▼ Clase 6 - Introducción al análisis y diseño OO parte 2

▼ Heurísticas para asignación de responsabilidades

Responsabilidades de los objetos:

- Hacer
  - hacer algo por si mismo
  - iniciar una acción en otros objetos
  - controlar o coordinar actividades de otros objetos
- Conocer (para hacer)
  - conocer sus datos privados encapsulados
  - conocer sus objetos relacionados
  - conocer cosas derivables o calculables

- Buscamos alta cohesión (que cada clase sepa bien sus responsabilidades) y bajo acoplamiento (baja dependencia de una clase de otra)

▼ Principios SOLID

- Single-Responsability Principle: Una clase deberá ser responsable de únicamente una tarea, y ser modificada por una razón. (alta cohesión).
- 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.

- Liskov Substitution Principle: Los objetos de un programa deben ser intercambiables por instancias de sus subtipos (subtipos = los objetos que extienden esa clase) sin alterar el correcto funcionamento del programa. Uso correcto de herencia y polimorfismo.
- Inteface-Segregation Principle: Cuando creamos interfaces (protocolos), las clases que la implementan no deben estar forzadas a incluir métodos que no van a utilizar.
- Dependency-Inversion Principle:
    - Los módulos de alto nivel de abstracción no deben depender de los de bajo nivel.
        - 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.
    - Las abstracciones no deben depender de detalles. Los detalles deben depender de las abstracciones.
        - Abstracciones: se refieren a protocolos (o interfaces) o clases abstractas.
        - Detalles: son las implementaciones concretas.

▼ Clase 7 - Testing

- ¿Qué es testear?

  Asegurarse de que el programa hace lo que se espera, como se espera, y no falla.

▼ Tipos de test

- Tests funcionales
- Test no funcionales
- Tests de unidad
- Tests de integración
- Tests de regresión
- Test punta a punta
- Tests automatizados

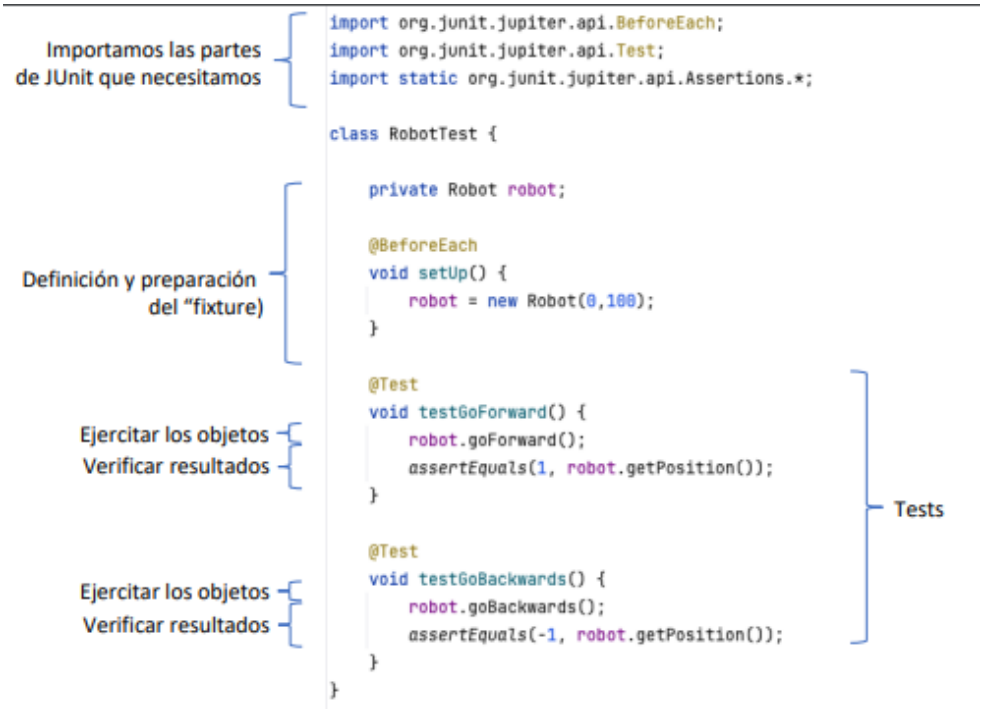
- Test de carga
- Test de performance
- Test de aceptación
- Test de UI
- Test de accesibilidad
- Alpha y beta tests
- Test A/B
- ...

▼ Test de unidad

Testea metodos: 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.

▼ jUnit

- herramienta para simplificar la creación de tests de unidad y automatizar su ejecución y reporte
- Ayuda a escribir/programar tests útiles
- Cada test se ejecuta independientemente de otros (aislados)
- jUnit detecta, recolecta, y reporta errores y problemas



VARIACIONES DEL ASSERT

```
@Test
void assertExamples() {
    assertEquals(5, "Hello".length());
    assertNotEquals("Hello","Bye");
    assertNotNull(myList);
    assertSame(myList, someList);
    assertTrue(myList.isEmpty());
    assertFalse(someList.isEmpty());
    assertThrows(IndexOutOfBoundsException.class, () -> {
        myList.remove("Hello");
    });
}
```

▼ ¿Por qué, cuándo y cómo testear?

- 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

- No necesario testear código de base que otros ya testearon (por ejemplo, partes del SDK, etc.)

- Por cada clase, creo una clase test.

▼ Test 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 casos en un conjunto, tomo un caso que pertenezca al
  conjunto y uno que no.
  - Ej., debe tener entrada → Casos: una persona con entrada, una sin.
- 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.

▼ Test de 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 estados/parámetros: velocidad, cantidad, posición, tamaño , duración, edad, etc.
- También podemos buscar en relaciones entre ellas (diferencia entre saldo y monto a extraer)
- Y buscar valores como: primero/último, máximo/mínimo, arriba/abajo, principio/fin, vacío/lleno, antes/después, junto a, alejado de , etc.

▼ Test 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.

▼ Clase 8 - Reuso de código, 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.
- 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.

▼ Clase 9 - JavaScript y Smalltalk

▼ JavaScript

- Lenguaje de propósito general
- Tipado dinámicamente
- Basado en objetos (con base en prototipos en lugar de clases)
  - No tengo clases
  - La forma más simple de crear un objeto es mediante la notación general
  - Cada objeto puede tener su propio comportamiento (métodos)
  - Los objetos heredan comportamiento y estados de otros (sus prototipos)
  - 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

▼ SmallTalk

- Lenguaje OO puro, donde todo es un objeto, incluso las clases
- Tipado dinámicamente
- Tiene dos tipos de objetos: los que pueden crear instancias (de si mismos) y los que no.
  - A los primeros los llamamos clases
- Entonces, hay objetos capaces de crear instancias y describir su estructura de comportamiento: las clases (p.e, Smallinteger)
- Todo objeto es instancia de una clase (p.e, 1 de SmallInteger)
- Las clases son instancias de la clase Metaclass
  - Por cada clase hay una metaclase (se crean juntas)
  - “SmallInteger es instancia de Metaclass
- Las metaclases son instancias de la clase Metaclass
  - “SmallInteger” class es intancia de Metaclass

▼ Clase 10 - Persistencia

▼ Preguntas de práctica para el teórico

- En un programa construido con objetos...
  - No hay un objeto más importante que otro.
- Para qué creamos clases?
  - Para representar la estructura y comportamiento de todos los objetos que son instancia de la clase.
- Para poder funcionar los objetos conocen...
  - A otros objetos a los que pueden enviarle mensajes usando el protocolo que dichos objetos exhiben.
- Cuando desarrollamos software con el paradigma de objetos, nuestros programas se pueden ver como...
  - Un conjunto de objetos que colaboran entre sí enviándose mensajes.

- Qué es un tipo en un lenguaje orientado a objetos?
    - Un tipo es conjunto de firmas de métodos.
- El binding dinámico nos permite...
    - Evitar el chequeo explicito de el tipo al que pertenece el objeto, dejando que dicho
      objeto se encargue de decidir que método se ejecuta (usando el algoritmo usual de "lookup")
- Cuando un objeto o recibe un mensaje m()...
    - Si encuentra el método m() correspondiente en su clase, lo ejecuta.
- Decimos que en un lenguaje de POO existe polimorfismo cuando...
    - puedo elegir diferentes implementaciones del método m() según me interese.
- Qué es recomendable usar para filtrar una colección en java?
    - El protocolo streams.
- Cuándo y tanto es importante testar?
    - Temprano, y cuanto sea el riesgo del artefacto a testear.
- Qué es recomendable para una colección en java?
    - Que todos los objetos de una colección compartan un tipo.
- ¿Por qué decimos que solo el objeto que es “dueño” de una colección puede modificarla?
    - Porque si no lo hacemos así, estamos rompiendo encapsulamiento.
- En el Modelo Conceptual o del Dominio, para lograr un mejor diseño OO, es aconsejable:
    - Incluir todas las clases candidatas que identifiquemos a partir de los Use Cases y de la Lista de Categorías, luego pueden revisarse en el Diagrama de Clases.
- En Reuso de código (Herencia Vs. Composición):
    - El reuso por composición, permite usar al objeto a través de su protocolo, sin necesidad de tener que conocer su implementación.
- En Reuso de código (Herencia):

    - Extendemos pensando en "Herencia de comportamiento", para cumplir con "Is-a”.
- Los Diagramas de Secuencia del Sistema (DSS) y los Diagramas de Secuencia del Diseño (DSD):
    - Son ambos Diagramas de Secuencia UML que expresan interacción entre objetos, en distintas etapas del proceso de desarrollo OO.
- En los Contratos de Operaciones, las postcondiciones:
    - describen el estado y cambios del sistema después de ejecutarse la operación, utilizando conceptos del modelo conceptual o del Dominio.
- ECMAScript está basado en clases?
    - No, está basado en prototipos.
- Qué se hereda en ECMAScript?
    - Comportamiento y prototipo.
- Qué se implementa con objetos en SmallTalk?
    - Todo se implementa con objetos y 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.
- SmallTalk y ECMAScript están tipados...
    - Dinámicamente.