Resumen Teórico Promoción OO1

Contenido

Clase 1 y 2 "Objetos, Clases, Instancias y Mensajes"

Programa o Sistema Orientado a Objetos

- Los sistemas están compuestos (solamente) por un conjunto de **objetos** que **colaboran** para llevar a cabo sus responsabilidades mediante el envío de **mensajes**.
- Algoritmos y datos ya no se piensan por separado.
- Cuando se ejecuta el programa lo que tenemos son objetos que cooperan y que se crean dinámicamente durante la ejecución del programa.
- Cuando codificamos, describimos clases.
- No hay un objeto "main".
- La estructura general cambia:
  - ❖ en vez de una jerarquía: **Main/procedures/sub-procedures** tenemos una red de "cosas" que se **comunican entre sí.**
- **Mientras que la estructura sintáctica es "lineal" el programa en ejecución no lo es.**

¿Qué es un objeto?

- Abstracción de una entidad del dominio del problema.
- Puede representar también conceptos del espacio de la solución (estructuras de datos, tipos "básicos", archivos, ventanas, conexiones, iconos, adaptadores, ...)

Características de los Objetos

- **Identidad**
  - ❖ Para distinguir un objeto de otro (independiente de sus propiedades).
- **Conocimiento**
  - ❖ En base a sus relaciones con otros objetos y su estado interno.
- **Comportamiento**
  - ❖ Conjunto de mensajes que un objeto sabe responder.

Estado Interno de un Objeto

- El estado interno de un objeto determina su conocimiento.
- El estado interno se mantiene en las **variables de instancia** del objeto.
- Es **privado** del objeto. Ningún otro objeto puede accederlo.
- El estado interno está dado por:
  - ❖ Propiedades básicas (intrínsecas) del objeto.
  - ❖ Otros objetos con los cuales colabora para llevar a cabo sus responsabilidades.

Responsabilidades de un Objeto

- Conocer sus propiedades.
- Conocer otros objetos (con los que colaboran).
- Llevar a cabo ciertas acciones.

Variables de Instancia

- Son **REFERENCIAS** (punteros) a otros objetos con los cuales el objeto colabora.
- Algunas pueden ser atributos básicos.

Comportamiento

- Un objeto se define en términos de su comportamiento.
- indica qué sabe hacer el objeto. Cuáles son sus **responsabilidades**.
- Se especifica a través del conjunto de **mensajes** que el objeto sabe responder: **protocolo**.

Implementación del Comportamiento

- La realización de cada **mensaje** se especifica a través de un **método**.
- Cuando un **objeto** recibe un **mensaje** responde activando el **método** asociado.

- El que envía el mensaje **delega** en el receptor la manera de resolverlo, que es **privada** del objeto.

Encapsulamiento

- Cualidad de los objetos de ocultar los detalles de implementación y su estado interno del mundo exterior.

Características del Encapsulamiento

- Esconde detalles de implementación.
- Protege el estado interno de los objetos.
- Un objeto sólo muestra su "cara visible" por medio de su protocolo.
- Los métodos y su estado quedan escondidos para cualquier otro objeto.
- Reduce el **acoplamiento**, facilita **modularidad** y **reutilización**.

Envío de Mensajes

- Para poder enviarle un mensaje a un objeto**, hay que conocerlo**.
- Al enviarle un mensaje a un objeto, éste responde activando el método asociado a ese mensaje (siempre y cuando exista).

Especificación de un Mensaje

- **Nombre**
  - ❖ Correspondiente al protocolo del objeto receptor.
- **Parámetros**
  - ❖ Información necesaria para resolver el mensaje.

Métodos

- Contraparte funcional del mensaje.
- forma de llevar a cabo la semántica propia de un mensaje particular (el cómo).

Cosas que puede realizar un Método

- Modificar el estado interno del objeto.
- Colaborar con otros objetos.
- Retornar y terminar.

Entrada y Salida con Objetos

- Se establecen conceptos como **lógica de dominio** y **lógica de interfaz** (son separados el uno del otro).
- En un sistema diseñado correctamente, un objeto de dominio no debería realizar ninguna operación vinculada a la **interfaz** (mostrar algo) o a la **interacción** (esperar un "input").
- Si un objeto no puede realizar E/S la forma de probar sus funcionalidades es mediante **test unitarios**.

Formas de Conocimiento

- Para que un Objeto conozca a otros debe de establecer una **ligadura o binding** entre un **nombre** y un **objeto.**

Tipos de relaciones entre Objetos

- **Conocimiento Interno**
  - ❖ Variables de instancia.
- **Conocimiento Externo**
  - ❖ Parámetros.
- **Conocimiento Temporal**
  - ❖ Variables temporales.
- **Pseudo-variables**
  - ❖ "this" o "self" y "super".

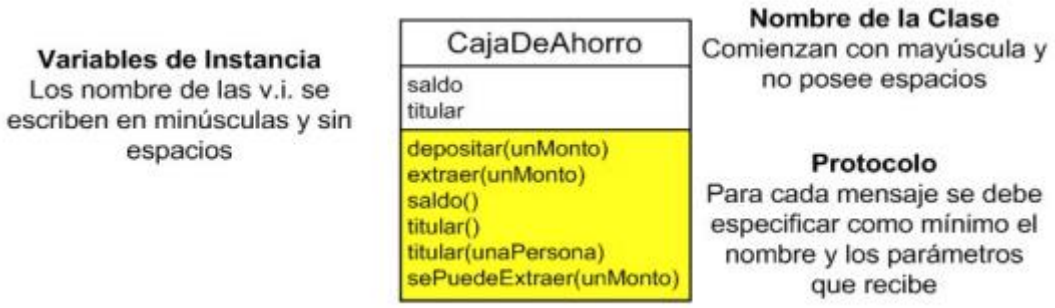
Clases

- Descripción abstracta de un conjunto de objetos.
- Todas las instancias de una clase se comportan de la misma manera.
- Cada instancia mantendrá su propio estado interno.

Roles que cumplen las Clases

- Agrupan el comportamiento común a sus instancias.
- Definen la **forma** de sus instancias.
- Crean objetos que son instancia de ellas.

Especificación de una Clase



Method lookup

- Cuando un **objeto** recibe un mensaje, se busca un método con la **firma correspondiente** (nombre y parámetros) en la clase de la cual es instancia.
- **Escenarios posibles:**
  - ❖ Lo encuentra y lo ejecuta “en el contexto del objeto”
  - ❖ No lo encuentra, vuelve a buscar en las **superclases** de ese objeto, y si no lo encuentra entonces tendremos un error o excepción ya sea en tiempo de ejecución o en compilación.
- **Es la clave para la Orientación a Objetos.**

Instanciación e Inicialización

- Mecanismo de creación de objetos.

- Los objetos se instancian a partir de un **molde**, en nuestro caso una **clase**.
- Un nuevo **objeto** es una **instancia** de una clase.
- Todas las instancias de una clase tendrán la misma **estructura interna** y responderán al mismo **protocolo** de la misma **manera**.
- Para que un objeto esté listo para responder a sus **responsabilidades** hace falta **inicializarlo** con **valores iniciales**.

Identidad

- Las variables al ser **punteros** puede ocurrir que más de una apunte al mismo **objeto**.
- En JAVA para saber si dos variables son **idénticas** se usa "==".

Igualdad

- Dos objetos pueden ser iguales en cuanto a sus valores.
- La igualdad se define en función del dominio.
- En JAVA para saber si dos variables son **iguales** se usa "equals()".

Pseudo-variable "this" o "self"

- No se le puede dar valor.
- Toma valor automáticamente cuando un objeto comienza a ejecutar un método.
- hace referencia al objeto que ejecuta el método.

Uso de la Pseudo-variable "this" o "self"

- Descomponer métodos largos (top down).
- Reutilizar comportamiento repetido en varios métodos.
- Aprovechar comportamiento heredado.

Clase 3 "Polimorfismo, Interfaces y Delegación"

Relaciones entre objetos

¿Por qué un objeto conoce a otro?

- Es su responsabilidad mantener a ese otro objeto en el sistema.
- Necesita delegarle trabajo.

¿Cuándo un objeto conoce a otro?

- Tiene una referencia en una variable de instancia.
- Le llega una referencia como parámetro.
- Lo crea.
- Lo obtiene enviando mensajes a otros que conoce.

Un objeto que conoce a muchos…

- Las relaciones de un objeto a muchos se implementan con **colecciones**.
- Decimos que un objeto conoce a muchos, pero en realidad conoce a una **colección**, que tiene **referencias** a esos muchos.

Tipos en leguajes Orientados a Objetos

- Conjunto de firmas de operaciones/métodos (nombre, orden y tipos de los argumentos).
- Decimos que un objeto **"es de un tipo"** si ofrece el conjunto de operaciones definido por el tipo.

Chequeo de Tipos

- Java es un lenguaje, estáticamente, fuertemente tipado, es decir, se debe indicar el tipo de todas las variables.
- El compilador se encarga del Chequeo de Tipos.
- Se asegura de que no enviamos mensajes a objetos que no los entienden.

- Cuando asignamos un objeto a una variable controla que el tipo del objeto sea compatible con el de la variable.

Interfaces

- Nos permite declarar **tipos** sin tener que ofrecer **implementación** (desacopla tipo e implementación).
- Elemento que define un conjunto de operaciones que una clase o componente debe implementar.
- Puedo utilizar Interfaces como **tipos de variables**.
- Las clases deben declarar explícitamente que interfaces implementan.
- Una clase puede implementar varias **interfaces**.
- El compilador chequea que la clase implemente las interfaces que declara.

Envidia

- Una clase puede ser **envidiosa y egoísta** de otra queriendo hacer todo el trabajo posible.
- Responsabilidades poco repartidas.
- Clases más **acopladas** y poco **cohesivas**, lo contrario a lo que buscamos.

Delegación

- Mecanismo que permite que una clase **delegue** en otra una determinada funcionalidad.
- Se puede aplicar como una sustitución a la **herencia.**
- Clases más **desacopladas** y más **cohesivas**.

Polimorfismo

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

- Si dos clases implementan una interfaz, se vuelven polimórficas respecto a los métodos de la interfaz.

Cosas que implica el Polimorfismo

- Un mismo mensaje se puede enviar a objetos de distinta clase.
- Objetos de distinta clase “podrían” ejecutar métodos diferentes en respuesta a un mismo mensaje.

¿Qué nos permite el Polimorfismo bien aplicado?

- Delegación de **responsabilidades**.
- Desacopla **objetos** y mejora la **cohesión**.
- Reduce el impacto de los **cambios**.
- Permite la **extensión** sin **modificación**.
- Código más **genérico** y **reusable**.
- Programar por **protocolo** y no por **implementación**.

Clase 4 “Nuestra Arquitectura de Referencia”

Arquitectura Monolítica

- Una aplicación o sistema que utiliza una Arquitectura Monolítica es aquel donde existe una unidad cohesiva de código que contiene toda la funcionalidad necesaria para realizar la tarea para la cual fue diseñada.
- Modelamos esta unidad mediante un Objeto.
- Asumimos que se mantiene en el tiempo.
- Persiste colecciones.
- Actúa como punto de entrada de datos en nuestro grafo de un sistema.
- Hace lo menos posible.

Clase 5 “Herencia”

Herencia

- Mecanismo que permite a una clase “heredar” estructura y comportamiento de otra clase.
- Estrategia de reúso de código.
- Estrategia para reúso de conceptos
- Es una característica transitiva, es decir, si un objeto A hereda de un objeto B, y a su vez, un objeto B hereda de un objeto C, entonces el objeto A también hereda de manera indirecta las características de C. En otras palabras, la relación de herencia se transmite a lo largo de la cadena de clases.

¿Cómo se si es adecuado usar Herencia?

- Preguntarse “es-un” es la regla para identificar usos adecuados de herencia.
    - ❖ Si suena bien en el lenguaje del dominio, es probable que sea un uso adecuado.

Overriding de Métodos

- La búsqueda en la cadena de superclases termina tan pronto encuentro un método cuya firma coincide con la que busco.
- Si heredaba un método con la misma firma, el mismo queda “oculto”.
- No es una práctica que ocurra con frecuencia.

Pseudo-variable “super”

- Toma las mismas propiedades que **“this” o “self”** en cuanto a la toma de valor y el hecho de que no se le puede asignar uno.
- Utilizar “**super**” cambia la forma en la que se hace el **method lookup.**
- Se usa para extender comportamiento heredado.

Super y el method lookup

- Cuando **super** recibe un mensaje, la búsqueda de métodos comienza en la clase inmediata superior a aquella donde está definido el método que envía el mensaje.

Super() en los constructores

- Los constructores en Java son subrutinas que se ejecutan en la creación de objetos - no se heredan. Por lo tanto, si quiero reutilizar comportamiento de otro constructor debo invocarlo explícitamente usando **super(...)** al principio.

Especializar y Generalizar

- **Especializar**
  - ❖ Crear una subclase especializando una clase existente.
- **Generalizar**
  - ❖ Introducir una superclase que abstrae aspectos comunes a otras, suele resultar en una clase abstracta.

Clase abstracta

- Captura comportamiento y estructura que será común a otras clases.
- No puede tener instancias.
- Normalmente son especializadas por otras clases.
- Puede declarar comportamiento abstracto y utilizarlo para implementar comportamiento concreto.
- Puede poseer comportamiento concreto.

Clase 6 "Colecciones"

Colecciones en OO1

- Su rol principal es mantener relaciones entre objetos.
- En esta materia se usan como repositorios de datos.
- Buscan abstracción, interoperabilidad, performance, reuso, productividad.

- Admiten, generalmente, contenido heterogéneo en términos de clase, pero homogéneo en términos de comportamiento.
- En JAVA son populares los siguientes tipos de colecciones: List, Set, Map y Queue.

Generics y Polimorfismo en Colecciones

- Cuanto más sepa el compilador respecto al contenido de la colección, mejor podrá chequear lo que hacemos.
- Contenido homogéneo da lugar a **polimorfismo**.
- Al definir y al instanciar una colección indico el tipo de su contenido.

Operaciones Frecuentes en Colecciones

- Ordenar respecto a algún criterio.
- Recorrer y hacer algo con todos sus elementos.
- Encontrar un elemento (max, min, DNI = xxx, etc.).
- Filtrar para quedarme solo con algunos elementos.
- Recolectar algo de todos los elementos.
- Reducir (promedio, suma, etc.).
- Nos interesa escribir código que sea independiente del tipo de colección que utilizamos, promover el uso de **streams** y **métodos genéricos** en las interfaces de las **colecciones**.

Iterator

- Encapsula como recorrer una colección y el estado de un recorrido.
- Son polimórficos.

Precauciones a tener en cuenta al usar Colecciones

- Nunca modificar una colección obtenida de otro objeto. Podemos romper el **encapsulamiento**.
- Cada objeto es responsable de mantener los invariantes de sus colecciones.

Expresiones Lambda

- Son métodos anónimos.
- Útiles para:
  - ❖ Parametrizar lo que otros objetos deben hacer.
  - ❖ Decirle a otros objetos que me avisen cuando pase algo (callbacks).
- Sintaxis:
  - ❖ **(parámetros, separados, por, coma) -> {cuerpo lambda}**

Streams

- Objetos que permiten procesamiento funcional de colecciones combinando operaciones para formar **pipelines**.
- Están en un alto nivel por lo tanto son más **concisas** y están **optimizadas** y **probadas**.
- No almacenan los datos, sino que proveen acceso a una fuente de datos subyacente.
- Cada operación produce un **resultado**, pero no modifica la **fuente**.
- Potencialmente sin final.
- **Consumibles**:
  - ❖ Los elementos se procesan de forma secuencial y se descartan después de ser consumidos.
- Normalmente obtenibles con el mensaje "**stream**()".

Stream Pipelines

- Se construyen mediante el encadenamiento de **mensajes**.
- Elementos necesarios:
  - ❖ Una **fuente** de elementos.
  - ❖ Cero o más **operaciones intermedias**, que devuelven otro **stream**.
  - ❖ **Operaciones terminales**, que retornan un resultado.
- Las **operaciones terminales** guían el proceso.
- Las **operaciones intermedias** son **Lazy**:

- ❖ Se calculan y procesan solo cuando es necesario, es decir, cuando se realiza una operación terminal que requiere el **resultado**.

Stream Pipelines - Algunos ejemplos

Operaciones intermedias	Operaciones terminales
filter	count | sum
map	average
limit	findAny | findFirst
sorted	collect
	anyMatch | allMatch | noneMatch
	min | max

Optional

- Utilizado para representar un valor que podría estar presente o no en un **resultado**.
- Forma de manejar la posibilidad de valores **nulos** de manera **segura** y **explícita**.
- Es retornado por ciertos métodos de Stream (findFirst, max, etc.).

Operaciones frecuentes en Stream

- **Filter:**
  - ❖ Operación Intermedia.
  - ❖ Retorna un nuevo stream que solo contiene los elementos que cumplen cierto predicado.
  - ❖ Los predicados son expresiones lambdas que resultan en valores booleanos.
- **Map**:
  - ❖ Operación Intermedia.

- ❖ Retorna un stream que transforma cada elemento de entrada aplicándoles una función de mapeo.
- ❖ La función de mapeo recibe un elemento del stream y devuelve un objeto.

- **Sorted**:
  - ❖ Operación Intermedia.
  - ❖ Usado para ordenar elementos en un orden específico.
  - ❖ Puede requerir un comparador personalizado.
- **Collect**:
  - ❖ Operación Terminal.
  - ❖ Reductor que nos permite obtener un objeto o colección de objetos a partir de los elementos de un stream.
  - ❖ Recibe como parámetro un objeto Collector.
- **Find FIrst**:
  - ❖ Operación Terminal.
  - ❖ Devuelve un Optional con el primer elemento del Stream si existe.

Clase 7 "UML"

- Es un lenguaje de modelado visual que nos permite…
  - ❖ Especificar.
  - ❖ Visualizar.
  - ❖ Construir.
  - ❖ Documentar.

  …artefactos de un sistema de software.
- Permite capturar decisiones y conocimientos.

Diagramas de Estructura

- Grupo conformado por:
  - ❖ **Diagrama de Clases.**
  - ❖ Diagrama de Paquetes.
  - ❖ Diagrama de Componentes.
  - ❖ **Diagrama de Objetos.**
  - ❖ Diagrama de Despliegue.

Diagramas de Comportamiento

- Grupo conformado por:
  - ❖ **Diagrama de Casos de Uso.**
  - ❖ **Diagramas de Interacción.**
    - ➢ **Diagrama de Secuencia.**
    - ➢ Diagrama de Colaboración.
  - ❖ Diagrama de Máquinas de estado.
  - ❖ Diagrama de Actividades.

Diagrama de Casos de Uso

- Representación del comportamiento de un sistema tal como se percibe por un usuario externo.
- Describe una **interacción** específica entre los **actores** y el **sistema**, brindando un proceso completo de cómo se utiliza el sistema en situaciones reales.
- Se definen para satisfacer **los objetivos de usuarios o actores principales.**
- Los elementos de un modelo de casos de uso son:
  - ❖ **Actores**:
    - ➢ El término "**Actor**" engloba tanto personas como sistemas o entidades externas.
    - ➢ Son los que interactúan con el **sistema**.
  - ❖ **Caso de Uso**:
    - ➢ Representación de un **requerimiento** o **funcionalidad específica.**
  - ❖ **Relaciones**

Tipos o Grados de formalidad de un Diagrama de Casos de Uso

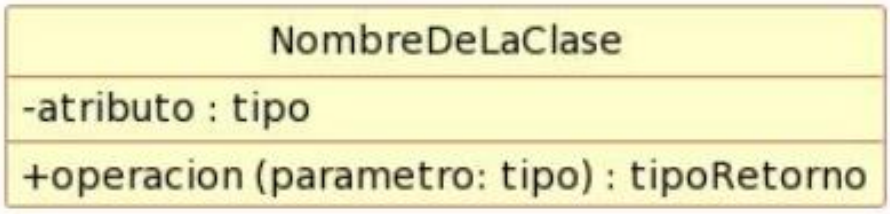
- **Breve**
  - ❖ resumen conciso que no ocupa más de un párrafo. Se describe el escenario principal con éxito (curso normal).
- **Informal**

- ❖ La descripción puede abarcar varios párrafos, pero no demasiados, especificando varios escenarios. Se caracteriza por un estilo informal de escritura.

- **Completo**
  - ❖ Es el formato más elaborado, ya que se describen con detalle todos los pasos y variaciones (curso normal y alternativo). Cuenta con otras secciones como pre y post condiciones, etc.

Diagrama de Clases

- Descripción de conjunto de objetos que comparten los mismos atributos, operaciones, métodos, relaciones y semántica.
- Una clase es representada gráficamente por cajas con tres compartimientos



Nombre de la Clase

- En singular.
- Debe comenzar con Mayúscula.
- Estilo CamelCase.
- Si la clase es abstracta:
  - ❖ Cursiva o estereotipo <<abstract>>.

Atributos de la Clase

- **Visibilidad**:
  - ❖ Privada (-).
  - ❖ Protegida (#).
  - ❖ Pública (+).
  - ❖ Paquete (~).
- **Nombre**:
  - ❖ Estilo CamelCase.

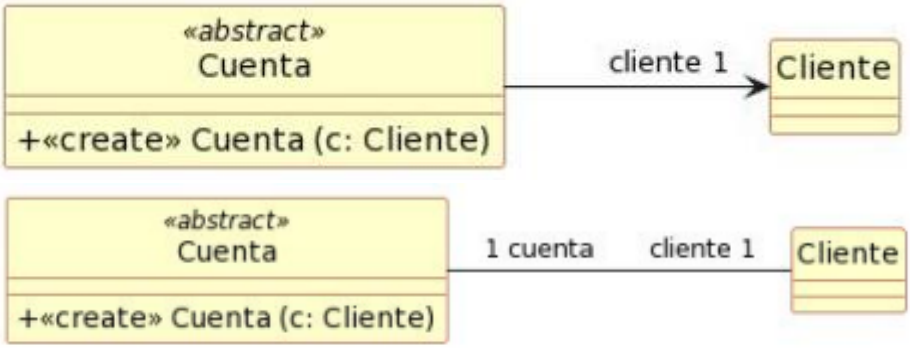
- ❖ Comienza con minúscula.
- **Tipos**:
  - ❖ Integer, Real, Boolean, String.

Operaciones

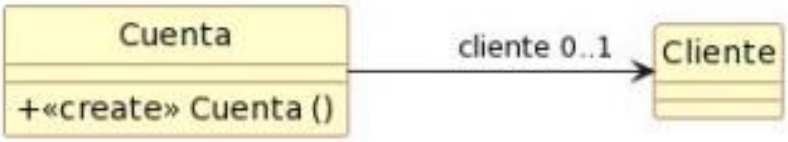
- **Visibilidad**:
  - ❖ Privada (-).
  - ❖ Protegida (#).
  - ❖ Pública (+).
  - ❖ Paquete (~).
- **Nombre**:
  - ❖ Estilo CamelCase.
  - ❖ Comienza con minúscula.
- **Parámetros**:
  - ❖ Nombre: estilo CamelCase.
- **Tipo de Retorno**:
  - ❖ Si no retorna nada, no se especifica.
  - ❖ SI retorna un objeto, se indica de qué clase.
  - ❖ Si retorna una colección, se indica el nombre de la clase [*].
- **Si el método es abstracto**:
  - ❖ Cursiva o estereotipo <<abstract>>.
- **Si el método es un constructor**:
  - ❖ Con estereotipo <<create>>.

Asociaciones entre Clases

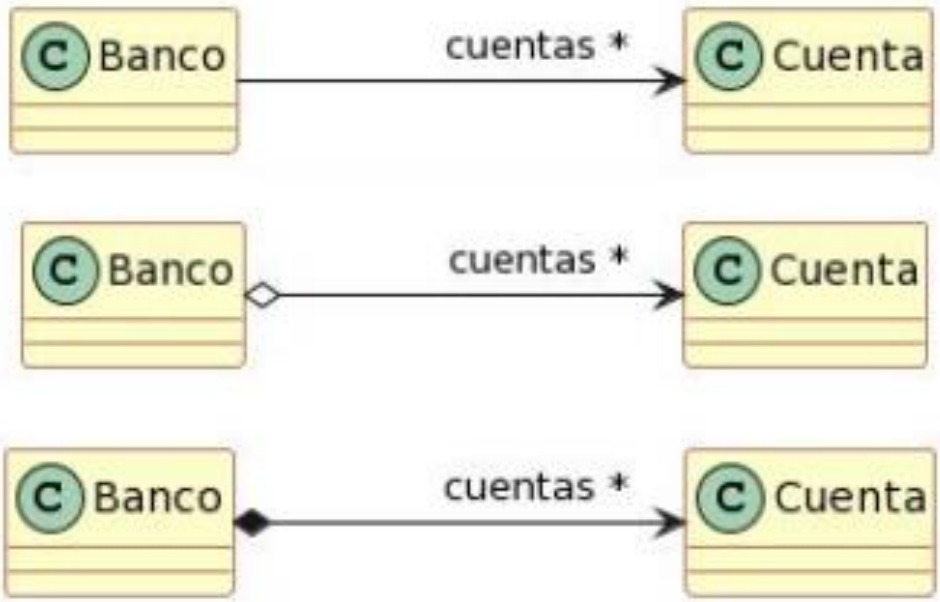
- **Navegabilidad**



- **Multiplicidad**



- **Nombre de Rol.**
- **Tipos**
  - ❖ Simple.
  - ❖ Agregación.
  - ❖ Composición.



Interfaces

- Se representan mediante una caja con el nombre de la interfaz y una lista de operaciones o métodos que deben ser implementados por las clases que la utilicen. No se especifica la implementación de las operaciones en la interfaz.
- Usar estereotipo <<interface>>.

Interfaces y Herencia

- Una clase concreta **implementa** una interface. Esto significa que la clase proporciona una implementación concreta para todos los métodos definidos en esa interfaz.
- Una clase concreta **extiende** a una clase abstracta, significa que debe proveer las implementaciones de los métodos abstractos.

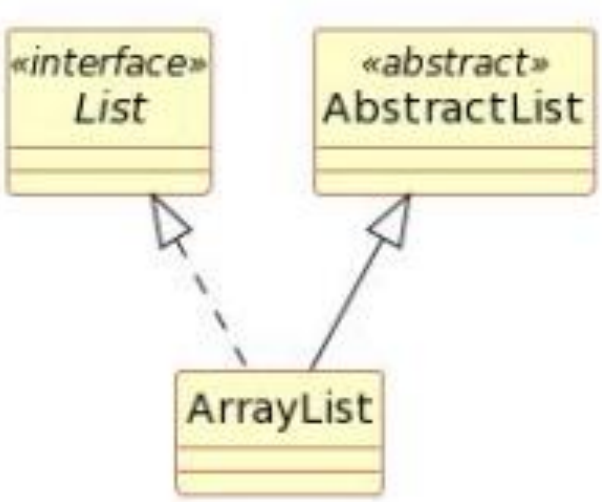


Diagrama de Objetos

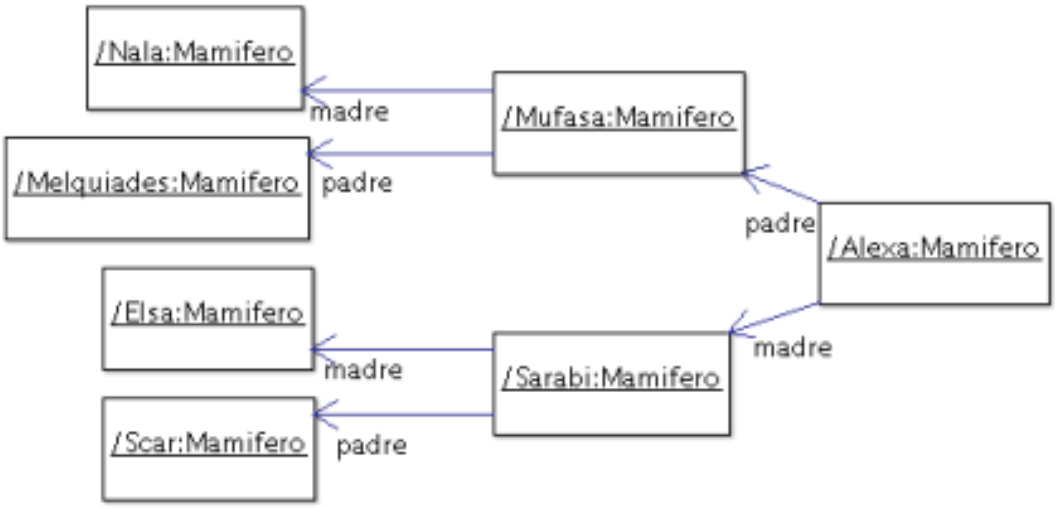


Diagrama de Paquetes

- Permiten la agrupación de clases.

- Se usa para organizar los elementos.
- Útiles para mostrar la organización de un sistema y cómo los elementos se agrupan y relacionan entre sí.

¿Qué buscamos con el uso de estos Diagramas?

- Alta cohesión dentro de un paquete.
- Los elementos dentro de un paquete están relacionados.
- Poco acoplamiento entre ellos.

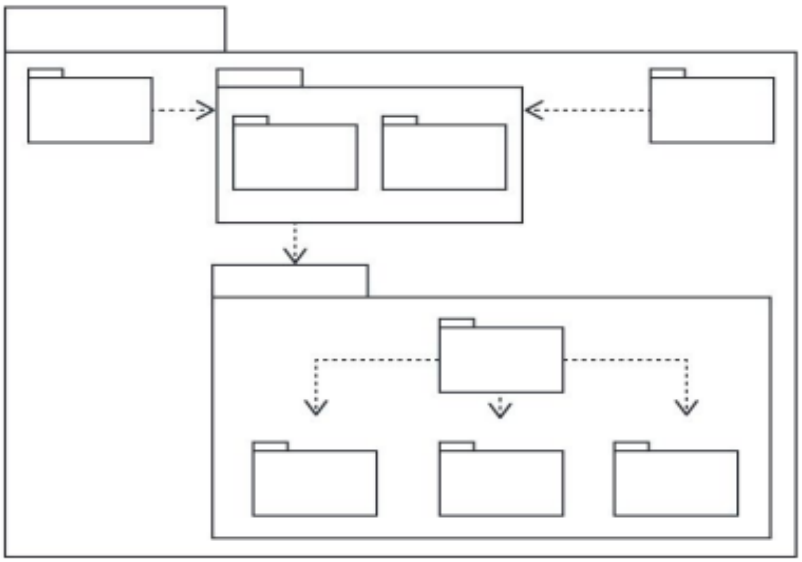
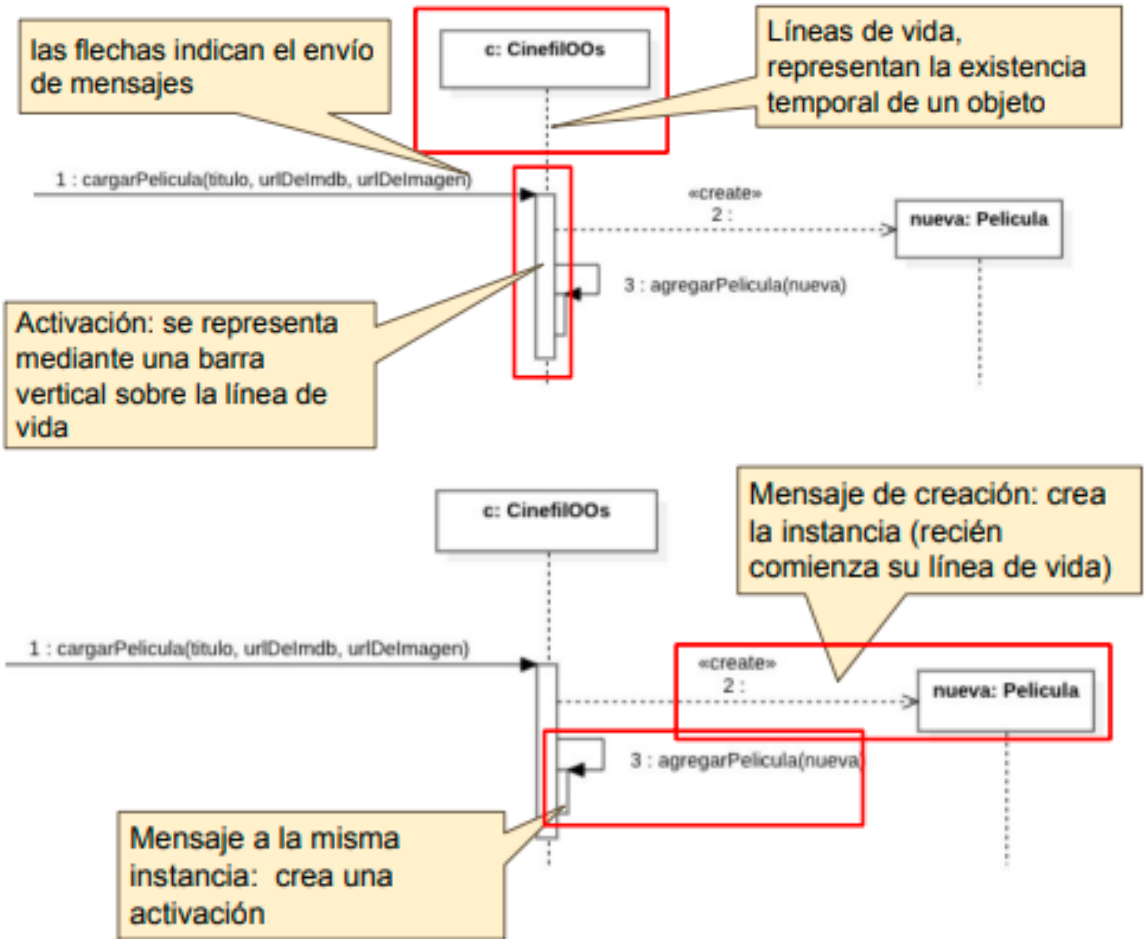


Diagrama de Secuencia

- Diagrama de interacción que describe cómo, y en qué orden, colabora un grupo de objetos.
- Muestra cómo interactúan distintos objetos en un sistema a lo largo del tiempo.
- Los objetos se representan en la parte superior del diagrama.
- El tiempo avanza de arriba hacia abajo.



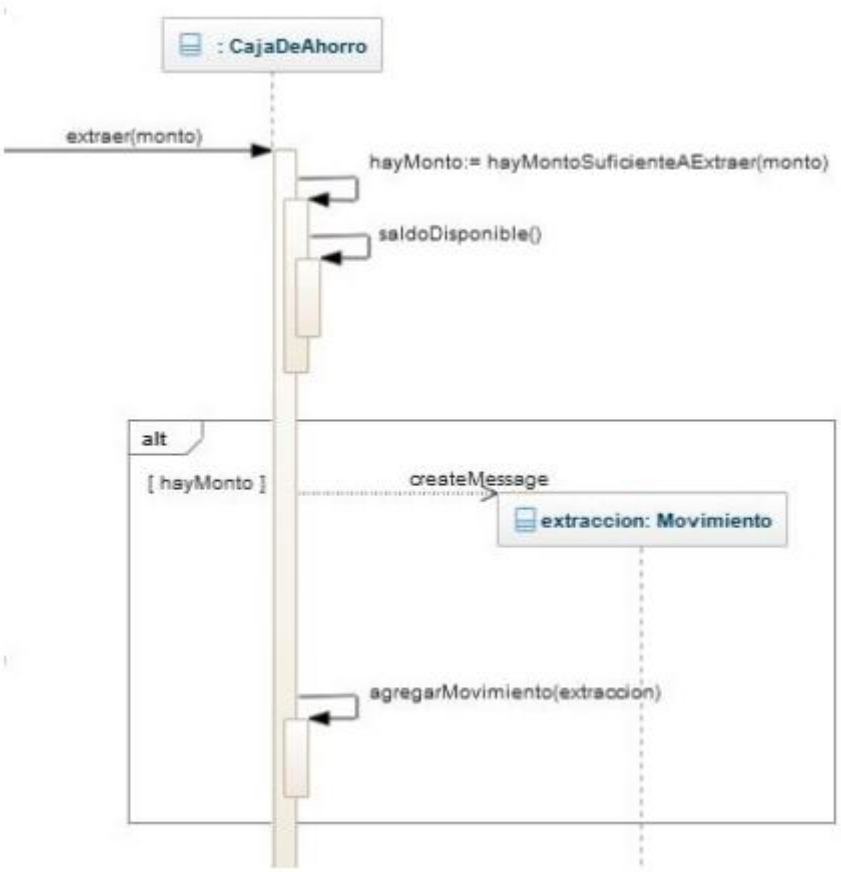
- Sintaxis del mensaje
  - ❖ **[valor de retorno :=] nombre del mensaje (parámetros)**
  - ❖ Los corchetes indican que es opcional.

Diagrama de Secuencia – CombinedFragment

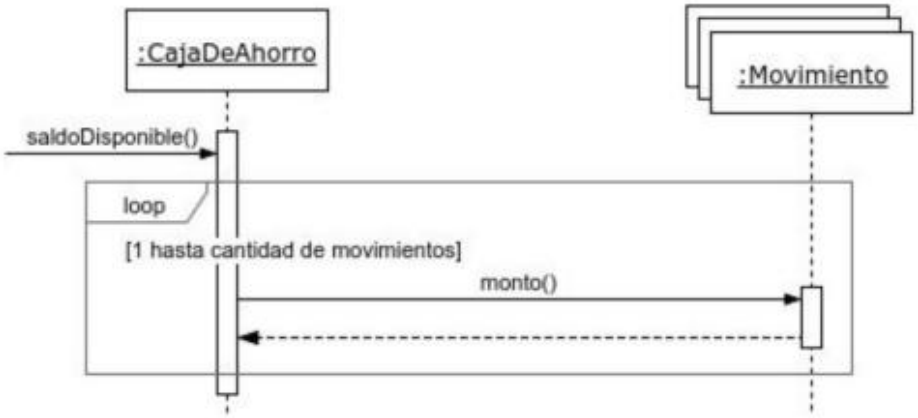
- Elemento que se utiliza para representar la lógica de control y las estructuras condicionales en una secuencia de interacción entre objetos. A través de ellos se pueden especificar bloques repetitivos, opcionales y alternativos, entre otros.

Fragmentos más utilizados

- **alt (Alternativa)**
  - ❖ se utiliza para modelar una elección entre diferentes opciones de interacción. En cada opción se evalúa una condición booleana para determinar cuál de las opciones se ejecutará.



- **opt (Opcional)**
  - ❖ Representa una parte de la secuencia de interacción que puede o no ejecutarse, dependiendo de una condición booleana.
- **loop (Bucle)**
  - ❖ Se utiliza para modelar repeticiones de una secuencia de interacción. Puede especificar el número de repeticiones o utilizar una condición para controlar la terminación del bucle.



Clase 8 "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.**
- Test de integración.
- Tests de regresión.
- Test punta a punta.
- **Tests automatizados.**
- Etc.

Test de Unidad

- Test que asegura que la **unidad mínima** (en nuestro caso los métodos) de nuestro programa funciona correctamente, y aislada de otras unidades.
- Testear un método es confirmar que el mismo acepta el rango esperado de entradas, y que retorna le valor esperado en cada caso. Teniendo en cuenta:
  - ❖ Parámetros.
  - ❖ Estado del objeto pre-ejecución del método.
  - ❖ Objeto que es retornado.
  - ❖ Estado del objeto post-ejecución del método.

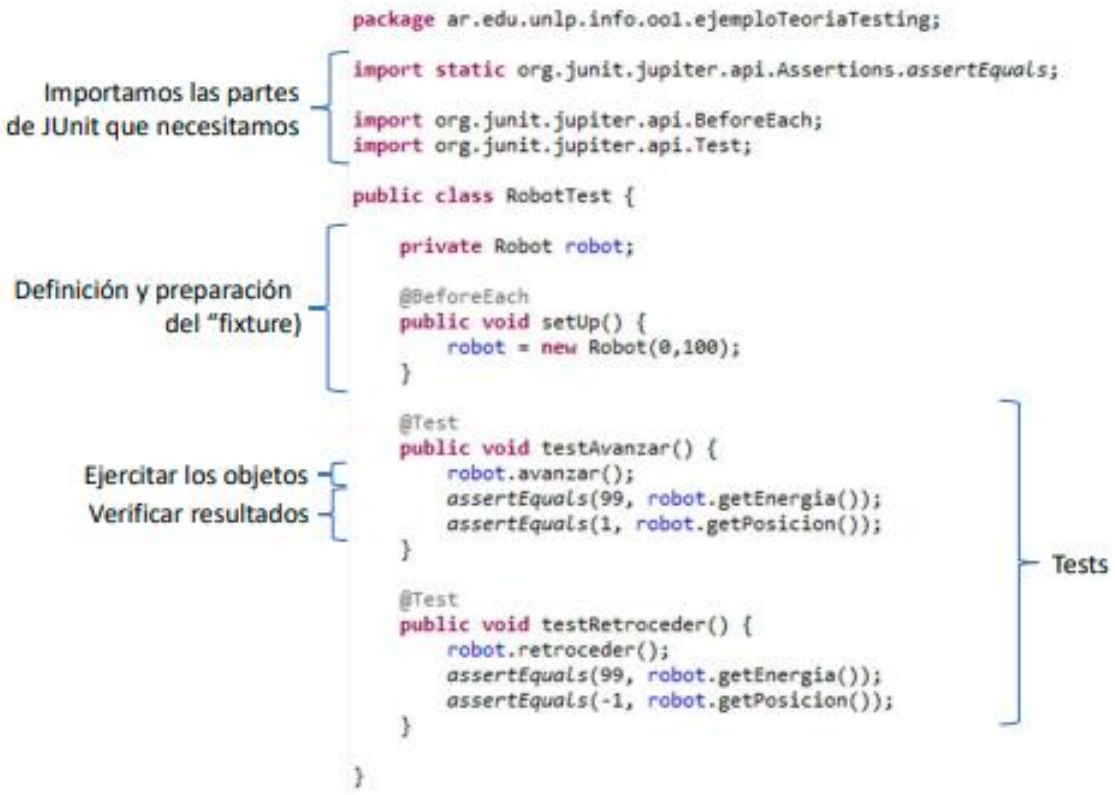
Tests Automatizados

- Se hace uso de software para guiar la ejecución de los tests y controlar los resultados.

- Requiere el diseño, programación y mantenimiento de programas “tests”.
- Suelen basarse en herramientas que resuelven gran parte del trabajo.
- Una vez creados, se pueden reproducir a costo mínimo y cuando quiera.
- Los tests se vuelven parte del software y un indicador de su calidad.

jUnit

- Framework de Java para la automatización de la ejecución de tests de unidad.
- Los tests tienen una ejecución aislada a otros tests.
  - ❖ No se puede asumir que otro test se ejecutó antes o se ejecutará después del que estoy escribiendo.
- jUnit se vuelve el encargado de la detección, recolección y reporte de errores y problemas.




```
package ar.edu.unlp.info.oo1.ejemploTeoriaTesting;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class RobotTest {

    private Robot robot;

    @BeforeEach
    public void setUp() {
        robot = new Robot(0,100);
    }

    @Test
    public void testAvanzar() {
        robot.avanzar();
        assertEquals(99, robot.getEnergia());
        assertEquals(1, robot.getPosicion());
    }

    @Test
    public void testRetroceder() {
        robot.retroceder();
        assertEquals(99, robot.getEnergia());
        assertEquals(-1, robot.getPosicion());
    }

}
```

¿Por qué, cuándo y cómo testear?

- Testeamos para encontrar bugs.
- Testeamos con un propósito.
- Pensamos por qué testear algo y con qué nivel hacerlo.
- Testeamos temprano en el desarrollo y frecuentemente.
- Testeo tanto como sea el riesgo del artefacto.

Estrategias de Testing

- **Ambas Estrategias son complementarias entre ellas.**

Test de Particiones Equivalentes

- Partición de Equivalencia
  - ❖ Conjunto de casos que prueban lo mismo o revelan el mismo bug.
- Si se trata de valores en un rango, tomo un caso dentro y uno por fuera en cada lado del rango.
- Si se trata de casos en un conjunto, tomo un caso que pertenezca al conjunto y uno que no.

Test con Valores de Borde

- Los errores ocurren con frecuencia en los límites por eso los buscamos ahí.
- Identificamos bordes en nuestras particiones de equivalencia y elegimos esos valores.
- Normalmente son valores del estilo: primero/último, máximo/mínimo, arriba/abajo, principio/fin, vacío/lleno, antes/después, junto a, alejado de, etc.

Clase 9 y 10 "Análisis y Diseño Orientado a Objetos"

Análisis

- El **análisis** pone énfasis en una **investigación del problema** y los **requisitos**, en lugar de ponerlo en la **solución**.
- Conformado por:
  - ❖ Casos de Uso.
  - ❖ DSS (Diagramas de Secuencia del Sistema).
  - ❖ Modelo del Dominio.
  - ❖ Contratos de Operación.
  - ❖ HAR (Heurísticas para la Asignación de Responsabilidades).

Diseño

- El **diseño** pone énfasis en una **solución conceptual** que **satisface los requisitos**, en lugar de ponerlo en la **implementación**.
- Conformado por:
  - ❖ Diagramas de Secuencia.
  - ❖ Diagramas de Clase.

Identificación de clases conceptuales

- La tarea central es identificar las clases conceptuales relacionadas con el escenario que se está diseñando.
- Es mejor especificar en exceso un modelo del dominio con muchas clases conceptuales de grano fino que especificar por defecto.

Estrategias de Identificación de clases conceptuales

- **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
Objeto físico o tangible
Especificación de una cosa
Lugar
Transacción
Roles de la gente
Contenedor de cosas
Cosas en un contenedor
Otros sistemas
Hechos
Reglas y políticas
Registros financieros/laborales
Manuales, documentos

Construcción de un Modelo de Dominio

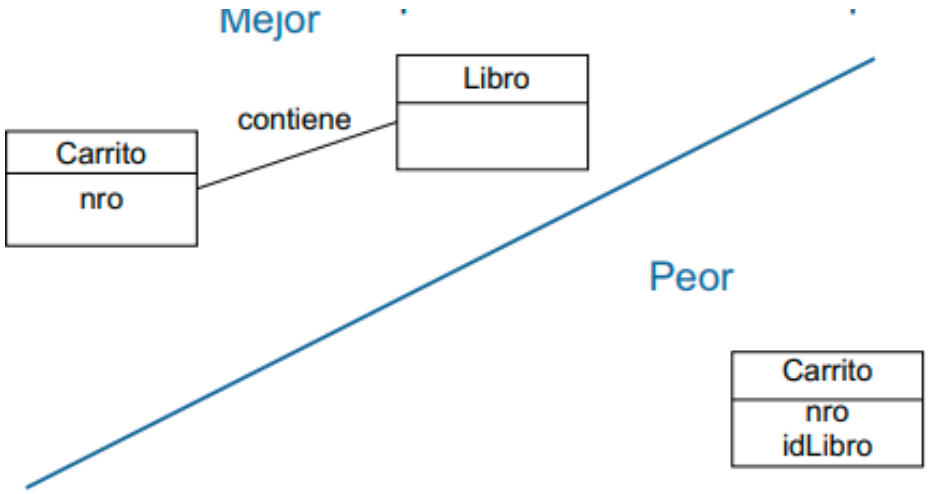
- **Pasos a seguir:**

1) **Listar los conceptos candidatos (Pueden ser clases o atributos).**

2) **Graficarlos en un Modelo del Dominio.**
   - ❖ Representación visual de las clases conceptuales del mundo real en un dominio de interés.

3) **Agregar Atributos**
   - ❖ Se identifican los atributos que son necesarios para satisfacer los requerimientos de información de los casos de uso en desarrollo.

- Los atributos en un modelo deberían ser, preferiblemente, atributos simples o tipos de datos primitivos.
- Representar lo que puede ser considerado tipo de dato primitivo como una clase conceptual si:
  - Está compuesto de secciones separadas.
  - Tiene operaciones asociadas.
  - Tiene otros atributos.
  - Es una cantidad con una unidad.
  - Es una abstracción de uno o más tipos con esas cualidades.
- La mejor manera de expresar que un concepto utiliza a otro es con una asociación.



4) Agregar asociaciones entre conceptos

- Focalizar las asociaciones que necesitan ser preservadas por un lapso de tiempo.
- Evitar mostrar asociaciones redundantes o derivadas.
- Más importante identificar clases conceptuales que asociaciones conceptuales.
- Demasiadas asociaciones pueden oscurecer el Modelo del Dominio.
- Agregar multiplicidades.
- Agregar roles.
- Lista de Asociaciones comunes:

Categoría
A es una parte física de B
A es una parte lógica de B
A está físicamente contenido en B
A está lógicamente contenido en B
A es una descripción para B
A es un miembro de B
A usa o maneja a B
A se comunica con B
A está relacionado con la transacción B
A es una transacción relacionada con otra transacción B
A es dueño de B

Contratos: Describiendo casos de uso

- Son una de las formas de describir comportamiento del sistema en forma detallada. Describen pre y post condiciones para las operaciones.

Secciones del Contrato

- **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 validan en la operación, se asumen como verdaderas.
- **Postcondiciones**

- ❖ El estado del sistema o de los objetos del Modelo del Dominio, después de que se complete la ejecución de la operación.

Operación: checkout pedido (c: Cliente)

Precondiciones:
- El cliente está registrado en el Sistema Gloovo.
- Existe un carrito, con productos, asociado al cliente.

Postcondiciones:
- Se creó un nuevo pedido con el carrito, la dirección de entrega y forma de pago.
- Se agregó el pedido a la colección de pedidos del cliente.
- Se agregó el pedido a la colección de pedidos de la EmpresaDePedidos.
- Se vació el carrito del cliente.

Transición del Análisis al Diseño

- Crear diagramas de interacción que muestran cómo los objetos se comunican con el objetivo de cumplir con los requerimientos capturados en la etapa de análisis.
- A partir de los diagramas de interacción, diseñar diagramas de clases.
- Crear diagramas de interacción requiere la aplicación de **Principios o Heurísticas para la Asignación de Responsabilidades**.
- Una vez hechos los **Diagramas de Interacción**, tener en cuenta las **clases** que participan allí y también en el **Modelo del Dominio o Conceptual** para poder graficarlas en un **Diagrama de Clases**.
- Para **Agregar Atributos y Asociaciones** vemos el **Modelo Conceptual** y para **Agregar Métodos** vemos los **Diagramas de Interacción**

Heurísticas para la Asignación de Responsabilidades

- La habilidad para asignar las responsabilidades es extremadamente importante en el diseño.

- Generalmente ocurre durante la creación de diagramas de interacción.

Experto en Información

- Asignar una **responsabilidad** al **experto en información** (la clase que tiene la información necesaria para realizar la responsabilidad). Expresa la intuición de que los objetos **hacen cosas relacionadas con la información que tienen.**
- Para cumplir con su responsabilidad, un objeto puede requerir de información que se encuentra dispersa en diferentes clases, esos son **expertos en información "parcial".**

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 determinar 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.

Bajo Acoplamiento

- Medida de **dependencia** de un objeto con otros. Es **bajo** si mantiene pocas relaciones con otros objetos.
- El **alto acoplamiento** dificulta el **entendimiento** y complica la **propagación de cambios en el diseño.**
- No se puede considerar de manera aislada a otras heurísticas, debe incluirse como principio de diseño que influye en la asignación de responsabilidades.
- Asignar responsabilidades de manera que el acoplamiento permanezca lo más **bajo** posible.

Alta Cohesión

- Medida de la **fuerza** con la que se relacionan las **responsabilidades** de un objeto, y la cantidad de ellas.
- El nivel de cohesión no se puede considerar de manera aislada a otras responsabilidades y otras heurísticas, como **Experto** y **Bajo Acoplamiento**.
- Asignar responsabilidades de manera que la cohesión permanezca lo más fuerte posible.
- Ventaja:
    - ❖ Clases fáciles de mantener, entender y reutilizar.

Polimorfismo como HAR

- 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 a objetos distantes o indirectos.
- Dentro de un método solo se puede enviar 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.

Clases Conceptuales

Entitys

- Tienen un **identificador**, son **modificables** y **comparables** por **Identidad**.

Value Object

- Comparables por **contenido** (igualdad estructural), **no tienen identificador.**
- No viven por sí mismos, **necesitan una entidad base**, son. Persisten adjunto a su base, no separadamente.
- **Inmutables** (no se le definen setters).
- En un Diagrama de Clases se representan con el estereotipo <<value object>>.

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

- Promueven Alta Cohesión y Bajo Acoplamiento.

"S" SRP: The Single-Responsibility Principle (Principio de Responsabilidad Única)

- Una clase debería cambiar por una sola razón.
- **Debería ser responsable de únicamente una tarea**, y ser modificada por una sola razón **(alta cohesión**).
- Es el principio más difícil de aplicar, pero el más fácil de entender.
- **Responsabilidad**

- ❖ En el contexto de **SRP**, e define como una **razón de cambio**. Si existe más de una razón que motive el cambio de una clase, entonces esa clase posee más de una **responsabilidad**.

"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. No se permiten alteraciones del código fuente.

"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, hacer un uso correcto de **herencia** y **polimorfismo**.

"I" ISP: The Interface-Segregation Principle

- Las clases que tienen interfaces "**voluminosas**" son clases cuyas interfaces **no son cohesivas.**
- Las clases no deben verse forzadas a depender de **interfaces o métodos** que no utilizan.

"D" DIP: The Dependency-Inversion Principle

- Los **módulos de alto nivel** de **abstracción** no deben depender de los de **bajo nivel**.
- 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).
- **Abstracciones:**
    - ❖ Se refieren a protocolos (o interfaces) o clases abstractas.
- **Detalles:**
    - ❖ Son las implementaciones concretas.

Herencia VS Composición de Objetos

Herencia de Clases

- **Herencia total:**
    - ❖ Debo **conocer todo el código** que se hereda -> **Reutilización de Caja Blanca.**
- Herencia de Estructura vs. Herencia de comportamiento.
- Útil para extender la funcionalidad del dominio de aplicación.
- Las clases y los objetos creados mediante herencia están **estrechamente acoplados.**

Composición de Objetos

- Los objetos se componen en forma Dinámica -> **Reutilización de Caja Negra**
- **Los objetos pueden reutilizarse a través de su interfaz (sin conocer el código).**
- A través de las relaciones de composición se pueden delegar responsabilidades entre los objetos.
- Las clases y los objetos creados a través de la composición están **débilmente acoplados**.

Clase 11 "Javascript y Smalltalk"

Smalltalk

- Lenguaje orientado a objetos puro, es decir, **todo es un objeto**.
- Tipado dinámico.

- Sintaxis minimalista.
- Propone una estrategia exploratoria) al desarrollo de software.
- Tiene un ambiente igual de cuidado e importante que el lenguaje en sí.
- **Posee 2 tipos de Objetos:**
  - ❖ Los que pueden crear instancias de sí mismos y describir su estructura y comportamiento, **Clases**.
  - ❖ Los que no pueden crear instancias.
- Todo **objeto** es instancia de una **clase** y estas son instancias de sus **metaclases**.
  - ❖ Por cada clase hay una **metaclase**, estas dos **se crean juntas**.
  - ❖ Las **metaclases** son instancia de la clase **Metaclass.**

Javascript

- Lenguaje de propósito general.
- Dinámico.
- Basado en objetos.
- Multiparadigma.
- Se adapta a muchos estilos de programación.
- Pensado originalmente para scripting de páginas web.
- Con una fuerte adopción en el lado del servidor (NodeJS).

Prototipos

- **No existen las clases.**
- La forma más simple de crear objetos es mediante **notación literal** (estilo JSON).
- **Cada objeto puede:**
  - ❖ Tener su propio **comportamiento**.
  - ❖ Heredar **comportamiento** y **estado** de otros (sus **prototipos**).
  - ❖ Servir como **prototipo** de otro.
- Es posible cambiar el **prototipo** de un objeto, haciendo variar su **estado** y **comportamiento**.
- Generan **cadenas de delegación**.