RESUMEN PARCIAL TEORICO OO1

- Un software construido con objetos consta de un conjunto de objetos que colaboran enviándose mensajes para poder llevar a cabo sus responsabilidades.
- Los objetos son responsables de:
  - Conocer sus propiedades.
  - Conocer otros objetos.
  - Llevar a cabo ciertas acciones.

Objeto

- Un objeto es una abstracción de una entidad del dominio del problema (persona, producto, etc).
- Puede representar también conceptos del espacio de la solución (estructuras de datos, archivos, etc).

Características de un objeto

- 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.

Estado interno

- El estado interno de un objeto determina su conocimiento.
- Este se mantiene en las variables de instancia del objeto.
- Es privado del objeto. Ningún otro objeto puede accederlo.
- Esta dado por:

- Propiedades básicas del objeto.
- Otros objetos con los cuales colabora para llevar a cabo sus responsabilidades.

Variables de instancia

- En general las variables son referencias a otros objetos con los cuales el objeto colabora.
- Algunas pueden ser atributos básicos.

Comportamiento

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

Implementación del comportamiento

- La manera en que un objeto sabe responder a un mensaje se especifica a través de un método.
- Cuando un objeto recibe un mensaje responde activando el método asociado.

Encapsulamiento

- Es la cualidad de los objetos de ocultar los detalles de implementación y su estado interno.
- Características:
  - Esconde detalles de implementación.

- Protege el estado interno de los objetos.
- Los métodos y su estado quedan escondidos para cualquier otro objeto. El objeto decide que se ve.
- **Reduce acoplamiento, facilita modularidad y reutilización.**

Envió de mensaje

- Para poder enviar un mensaje a un objeto hay que conocerlo.
- Como resultado del envío de un mensaje puede retornarse un objeto.
- Se especifica de la siguiente manera:

- *Nombre*: correspondiente al protocolo del objeto receptor.
- *Parámetros*: info necesaria para resolver el mensaje.

Métodos

- Un método es la contraparte funcional del mensaje, expresa la forma de llevar a cabo la semántica propia de un mensaje particular.
- Un método realiza basicamente 3 cosas:
  - Modifica el estado interno del objeto.
  - Colabora con otros objetos.
  - Retorna un valor y termina.

Formas de conocimiento

- Un objeto solo puede enviar mensajes a otros que conoce.
- Podemos identificar 3 formas de conocimiento:
  - Conocimiento interno: variables de instancia.
  - Conocimiento externo: parámetros.
  - Conocimiento temporal: variables temporales.

Clases e instancias

- Una clase es una descripción abstracta de un conjunto de objetos.
- Las clases cumplen 3 roles:
  - Agrupan el comportamiento común a sus instancias.

- Definen la forma de sus instancias.
- Crean objetos que son instancia de ellas.

Method lookup (búsqueda de métodos)

- Cuando un objeto recibe un mensaje, se busca un método con la firma correspondiente en la clase de la cual es instancia.
- Si se lo encuentra, se lo ejecuta, sino habrá un error en tiempo de ejecución.
- Esto se conoce como Binding Dinámico.

Instanciacion

- Es el mecanismo de creación de objetos.
- Un nuevo objeto es una instancia de una clase.
- Todas las instancias de una misma clase:

- Tendrán la misma estructura interna.
- Responderán a los mismos mensajes de la misma manera.

Identidad

- Las variables son punteros a objetos.
- Mas de una variable puede apuntar a un mismo objeto.

Igualdad

- Dos objetos pueden ser iguales.
- La igualdad se define en función del dominio.

This

- This es una pseudo variable a la cual no se puede asignarle un valor.
- Hace referencia al objeto que ejecuta el método.

RELACIONES ENTRE OBJETOS

- Un objeto conoce a otro cuando:

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

Interfaces

- Una interfaz nos permite declarar tipos sin tener que ofrecer implementación.
- Una clase puede implementar varias interfaces.

Polimorfismo

- Un objeto de distintas clases son polimorficos con respecto a un mensaje si todos lo entienden, aun cuando cada uno lo implemente de un modo diferente.
- Polimorfismo implica:
  - 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.

- *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 mas genérico y objetos reusables.
- Nos permite programar por protocolo, no por implementación.

Herencia

- Mecanismo por el cual una se le permite a una clase “heredar” estructura y comportamiento de otra clase.
  - Es una estrategia de reuso de código.

- Es una estrategia para reuso de conceptos.

- Una clase que hereda de otra se dice que es una *subclase* de …, o que *hereda* de …, o que *extiende* … .
- Una clase de la que heredan otras clases, se dice que es *superclase*.

Method lookup con herencia

- Cuando un objeto 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 así sucesivamente.

Super

- Super es una "pseudo-variable" como this.
- No se puede asignarle valor.
- En un método, super y this hacen referencia al objeto que lo ejecuta.
- Se utiliza para extender comportamiento heredado (reimplementar un método e incluir el comportamiento que se heredaba para el).
- Cuando super recibe un mensaje, la búsqueda de métodos comienza en la clase inmediata superior a aquella donde esta definido el método que envía el mensaje.

Clase abstracta

- Una clase abstracta captura comportamiento y estructura que sera común a otras clases.
- Una clase abstracta no puede ser instanciada.
- Seguramente sera especializada (se crea una subclase especializando la superclase).
- Puede declarar comportamiento abstracto y utilizarlo para implementar comportamiento concreto.

Generalizar

- Implica introducir una superclase que abstrae aspectos comunes a otras, esto suele resultar en una clase abstracta.
- (Cuando dos clases comparten aspectos comunes se crea una clase abstracta que contenga los mismos).

COLECCIONES

- Las colecciones buscan abstracción, interoperabilidad, performance, reuso, productividad.
- Admiten contenido heterogéneo en términos de clase, pero homogéneo enterminos de comportamiento.
- EL rol principal de las colecciones es mantener relaciones entre objetos.
- Algunos tipos de colecciones:
  - List:
    - Admite duplicados.
    - Sus elementos se están indexados por enteros desde 0 en adelante.
  - Set:
    - No admite duplicados.
    - Sus elementos no están indexados, ideal para chequear pertenencia.
  - Map:
    - Asocia objetos que actúan como claves, a otros que actúan como valores.

- Queue:
    - Maneja el orden en que se recuperan los objetos.

- Las colecciones admiten cualquier objeto en su contenido.
- Cuanto mas sepa el compilador respecto al contenido de la colección, mejor podrá chequear lo que hacemos.
- Al definir e instanciar una colección indico el tipo de su contenido.

Operaciones en colecciones.

- Ordenar respecto a algún criterio.
- Recorrer y hacer algo con todos sus elementos.
- Encontrar un elemento (max, min, 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.

Iterator

- Todas las colecciones entienden iterator().
- Un iterator encapsula:
    - Como recorrer una colección particular.
    - El estado de un recorrido.

- Nunca debo modificar 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.

Expresiones Lambda

- Son métodos anónimos (no tienen nombre ni pertenecen a ninguna clase).
- Útiles para:
    - Parametrizar lo que otros objetos deben hacer.
    - Decirle a otros objetos que me avisen cuando pase algo.

Streams

- Objetos que permiten procesamiento funcional de colecciones.
- Las operaciones se combinan para formar pipelines.

- No almacenan los datos, sino que proveen acceso a una fuente de datos subyacente.
- Cada operación produce un resultado, pero no modifica la fuente.
- Consumibles: los elementos se procesan de forma secuencial y se descartan después de ser consumidos.
- La forma mas frecuente de obtenerlos es vía el mensaje stream() a una colección.

Streams Pipelines

- Para construir un pipeline encadeno envíos de mensajes.

- Debe contener:
  - Una fuente, de la que se obtienen los datos.
  - Cero o mas operaciones intermedias, que devuelven un nuevo stream.
  - Operaciones terminales, que retornan un resultado.
- La operación termina guía el proceso. Las intermedias son Lazy: se calculan y procesan solo cuando es necesario es decir, cuando se realiza una operación terminal que requiere el resultado.

Optional

- Se utiliza para representar un valor que podría estar presente o ausente en un resultado.
- Son una forma de manejar la posibilidad de valores nulos de manera mas segura y explicita.

*Operación **filter()***

- El mensaje filter retorna un nuevo stream que solo “deja pasar” los elementos que cumplen cierto predicado.
- El predicado es una expresión lambda que toma un elemento y resulta en true o false.

*Operación: **map()***

- El mensaje 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.

Operación: sorted()

- Se usa para ordenar los elementos de la secuencia en un orden especifico.
- Se puede usar para ordenar elementos en orden natural (si son comparables) o se debe proporcionar un comparador personalizado para especificar como se debe realizar la ordenación.

Operación: collect()

- El mensaje collect() es una operación terminal.
- Es un “reductor” que nos permite obtener un objeto o colección de objetos a partir de los elementos del stream.

Operación: findFirst()

- El mensaje findFirst() es una operación terminal.
- Devuelve un Optional con el primer elemento del Stream si existe.
- Luego puedo usar:
  - orElse() que devuelve el valor contenido en el Optional si esta presente. Si este esta vacío, entonces devuelve el valor predeterminado proporcionado como argumento.

UML (Lenguaje Unificado de Modelado)

- Es un lenguaje de modelado visual que se usa para especificar, visualizar, construir y documentar artefactos de un sistema de software.
- Permite capturar decisiones y conocimientos.

Diagramas de Casos de Uso

- Un caso de uso es una representación del comportamiento de un sistema tal como se percibe por un usuario externo.
- Describe la interacción especifica entre los actores y el sistema, proporcionando un proceso completo de como se utiliza el sistema en situaciones reales.

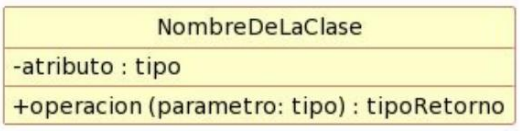
- Los elementos de un modelo de casos de uso son:
  - Actores.
  - Casos de uso.
  - Relaciones.

  Actor: representa a un usuario externo, un sistema o una entidad que interactuá con el sistema que se esta modelando.
  Caso de uso: es una representación de una funcionalidad especifica.
  Relaciones: son las relaciones entre los casos de uso (extensión, inclusión).

- Una relación A incluye a B significa que una instancia de A también incorporara el comportamiento especificado en B.

Diagrama de clases

- Una clase es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, métodos, relaciones y semántica.
- Una clase se representa gráficamente por cajas con 3 compartimientos:



- Nombre de clase: singular
- debe comenzar con mayúscula.
- estilo CamelCase.

- Si la clase es abstracta, debe ponerse en cursiva con la inscripción "*«abstract»*".

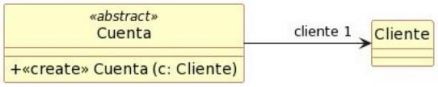
- Atributos:
  - Visibilidad: privada (-), protegida (#).
  - Nombre: estilo camelCase, comienza con minúscula.

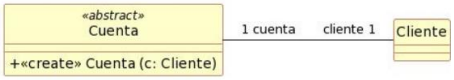
- Tipo:
- Integer.
- Real.
- Boolean.
- String.

- Métodos:

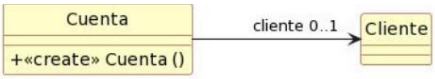
- Si es abstracto:
 - cursiva.
 - con la inscripción “*«abstract»*”.

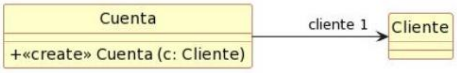
- Operaciones:
  - El nombre debe seguir el estilo CamelCase, comenzando en minúscula.
  - Los parámetros deben tener nombre y tipo, y deben separarse por comas.
  - Se indica el tipo de retorno, si devuelve algún valor. Si no retorna nada, no se especifica el retorno.
- Asociación:
  - Navegabilidad:





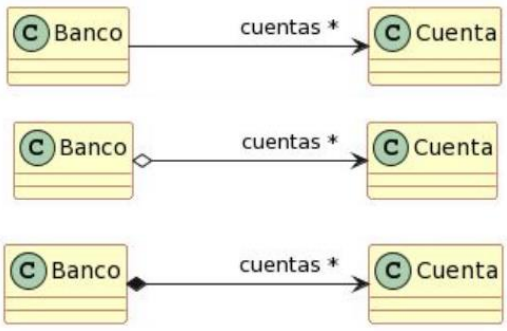
  - Multiplicidad:





  - Tipo:
    - 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 implementación de las operaciones en la interfaz, simplemente se define que operaciones deben estar disponibles en las clases que la implementan.

- Cuando una *clase* implementa una *interfaz*, se representa mediante una linea punteada desde la clase a la interfaz.

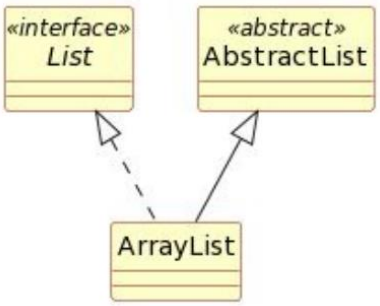


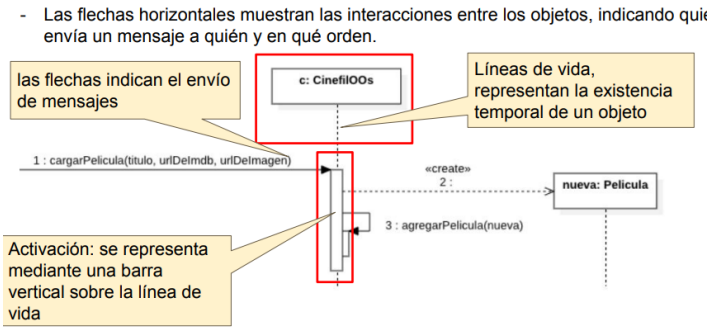
Diagrama de paquetes

- Permite la agrupación de clases.
- Se utiliza para organizar los elementos.
- Son útiles para mostrar la organización de un sistema y como los elementos se agrupan y relacionan entre si.

- Se quiere:
  - Una alta cohesión dentro de un paquete, es decir, los elementos dentro de un paquete están relacionados.
  - Poco acoplamiento entre ellos.

Diagrama de secuencia

- Un diagrama de secuencia es un tipo de diagrama de interacción porque describe como y en que orden colabora un grupo de objetos.
- Muestra claramente como interactúan distintos objetos en un sistema a lo largo del tiempo.
- En un diagrama de secuencia:
  - Los objetos se representan en la parte superior del diagrama.
  - El tiempo avanza de arriba hacia abajo



TESTING

- Testear es asegurarse de que el programa:
  - Hace lo que se espera.
  - Lo hace como se espera y no falla.

- Tipos de test:
  - Test funcionales.
  - Test de unidad.
  - Test automatizados.
  - Etc.

Test de Unidad

- Asegura que la unidad mínima de nuestro programa funciona correctamente, y aislada de otras unidades.
  - En este 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 el valor esperado en cada caso.
  - Tener en cuenta:
    - Parámetros.
    - Estado del objeto antes de ejecutar el método.
    - Objeto que retorna el método.
    - Estado del objeto al concluir la ejecución del método.

Test automatizados

- Se utiliza software para guiar la ejecución de los tests y controlar los resultados.
- Requiere que diseñemos, programemos y mantengamos programas “tests”.
  - En este caso serán objetos.
- Suele basarse en herramientas que resuelven gran parte del trabajo.
- Una vez escritos, los puedo reproducir a costo mínimo, cuando quiera.
- Los tests son “parte del software”.

Junit

- Framework en Java para automatizar la ejecución de tests de unidad.
- Cada test se ejecuta independientemente de otros.
- Detecta, recolecta y reporta errores y problemas.

Anatomía de un test jUnit

- Una clase de test por cada clase a testear.
- Un método que prepara lo que necesitan los tests.
- Uno o varios métodos de test por cada método a testear.
- Un método que limpia lo que se preparo (si es necesario).

Independencia entre tests

- No puedo asumir que otro test se ejecuto antes o se ejecutara después del que estoy escribiendo.

Por que, cuando y como testear?

- Testeamos para encontrar bugs.
- Testeamos con un propósito (buscando algo).
- Testeamos temprano y frecuentemente.
- Testeo tanto como sea el riesgo del artefacto.

Estrategia general

- Pensar que podría variar y que pueda causar un error o falla.
- Elegir valores de prueba para maximizar las chances de encontrar errores haciendo la menor cantidad de pruebas posibles.
- Dos estrategias:
  - Particiones equivalentes.

- Valores de borde.

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 de cada lado del rango.
  - Ej: la temperatura debe estar entre 0 y 100 → casos: -50, 50, 150.
- Si se trata de casos en un conjunto, tomo un caso que pertenezca al conjunto y uno que no.
  - Ej: la temperatura debe ser un valor positivo → casos: -50, 50.

Tests con valores de borde

- 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 estilo: velocidad, cantidad, posición, tamaño, duración, edad, etc.
- Buscar valores como: primero/ultimo, máximo/mínimo, arriba/abajo, etc.

ANÁLISIS Y DISEÑO OO

- **Análisis**: pone énfasis en una investigación del problema y los requisitos, en lugar de ponerlo en la solución.
- **Diseño**: pone énfasis en una solución conceptual que satisface los requisitos, en lugar de ponerlo en la implementación.

Actores y casos de uso

- Los casos de uso se definen para satisfacer los objetivos de usuario o actores principales.
- El diagrama de casos de uso proporciona información visual concisa del sistema, los actores externos y como lo utilizan.
- Los nombres de los casos de uso deben tener el formato verbo – sustantivo.
- Los nombres de los actores deben describir su rol, o identificar un subsistema o dispositivo.

Casos de uso – Tipos de formalidad
Existen 3 tipos o grados de formalidad:

- *Breve*:
  Es un resumen conciso que no ocupa mas 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 mas elaborado, ya que se describen con detalle todos los pasos y variaciones (curso normal y alternativo). Tiene secciones como pre y post condiciones, curso de error, etc.

Modelo del dominio

Identificación de clases conceptuales

- La tarea central es identificar las clases conceptuales relacionadas con el escenario que se esta diseñando.
- Consejos:
  - Usar nombres del dominio del problema, no de la solución.
  - Omitir detalles irrelevantes.
  - No inventar nuevos conceptos.
  - Descubrir conceptos del mundo real.

- Estrategias:
  - Identificación de frases nominales.
  - Utilización de una lista de categorías de clases conceptuales.

Frases nominales

- Encontrar conceptos mediante la identificación de los sustantivos en la descripción textual del dominio del problema.

Lista de categorías

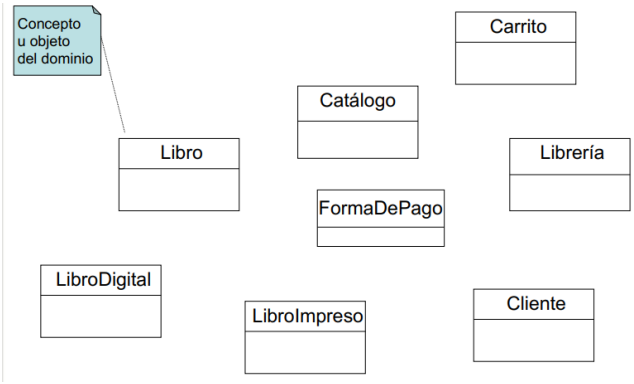
Categoría de Clase Conceptual	Ejemplos
Objeto físico o tangible	Libro impreso
Especificación de una cosa	Especificación del producto, descripción
Lugar	--
Transacción	Compra, pago, cancelación
Roles de la gente	cliente
Contenedor de cosas	Catálogo de libros, carrito
Cosas en un contenedor	Libro,
Otros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

del

Construyendo el modelo dominio

- Pasos a seguir:

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



- Se identifican los atributos que son necesarios para satisfacer los requerimientos de información de los casos de uso en desarrollo (preferiblemente atributos simples o tipos de datos primitivos).

3. Agregar atributos a los conceptos:

4. Agregar asociaciones entre conceptos.
- Lista de asociaciones comunes:

Categoría	Ejemplo
A es una parte física de B	No aplicable
A es una parte lógica de B	Detalle-Carrito
A está físicamente contenido en B	No aplicable
A está lógicamente contenido en B	Libro-Catálogo
A es una descripción para B	EspecificaciónDeProducto - Libro
A es un miembro de B	Cliente-Librería
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

Algunos tips:
- Focalizar las asociaciones que necesitan ser preservadas por un lapso de tiempo.
- Evitar mostrar asociaciones redundantes o derivadas.
- Es mas importante identificar clases conceptuales que asociaciones conceptuales.
- Demasiadas asociaciones pueden oscurecer el Modelo del Dominio.
- Recuerde agregar multiplicidades.
- Recuerde agregar roles.

Contratos de las operaciones

- Son una de las formas de describir comportamiento del sistema en forma detallada. Describen pre y post condiciones para las operaciones.
- Secciones de un 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. 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 cambios en el estado de los objetos del modelo del dominio.

HEURISTICAS PARA ASIGNACIÓN RESPONSABILIDADES (HAR)

- Responsabilidades de los objetos:

- Conocer sus datos privados encapsulados.
- Conocer sus objetos relacionados.
- Conocer cosas derivables o calculables.

-Hacer algo el mismo.
- Iniciar una acción en otros objetos.
- Controlar o coordinar actividades de otros objetos.

- La asignación de responsabilidades generalmente ocurre durante la creación de diagramas de interacción.

Heuristicas:

- *Experto en información.*
- *Creador*.
- *Controlador*: asignar la responsabilidad de manejar eventos del sistema a una clase que representa el sistema global, dispositivo o subsistema.
- *Bajo acoplamiento*: asignar responsabilidades de manera que el acoplamiento permanezca lo mas bajo posible.
  El acoplamiento es una medida de dependencia de un objeto con otros, es bajo si mantiene pocas relaciones con otros objetos.
- *Alta cohesión*: asignar responsabilidades de manera que la cohesión permanezca lo mas 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.
- *Polimorfismo*: cuando el comportamiento varia según el tipo, asigne responsabilidad a los tipos / las clases para las que varia el comportamiento.
- *"No hables con extraños"*: evitar diseñar objetos que recorren largos caminos de estructura y envían mensajes (hablan) a objetos distantes o indirectos (extraños). Dentro de un método solo pueden enviarse mensajes a objetos conocidos.

Modelo de Diseño

- Los casos de uso sugieren los eventos del sistema que se muestran en los diagramas de secuencia del 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.

Creación de los diagramas de clases de diseño

- Identificar las clases que participan en los diagramas de interacción y en el modelo del dominio o conceptual.
- Graficarlas en un diagrama de clases.
- Colocar los atributos presentes en el modelo conceptual.
- Agregar nombres de métodos analizando los diagramas de interacción.
- Agregar tipos y visibilidad de atributos y métodos.
- Agregar las asociaciones necesarias.

- Agregar roles, navegabilidad, nombre y multiplicidad a las asociaciones.

Clases Conceptuales: "Entity vs Value Object"

- Las entidades o clases del dominio de mi problema tienen un identificador, son modificables y comparables por *identidad*.
  Value Object:
- Son comparables por contenido (igualdad estructural), no tienen identificador.
- Son inmutables.
- Para representar un value object en el modelo, se antepone "<<value object>>" al nombre.

PRINCIPIOS SOLID

- **S**: Single responsibility: Una clase debe tener una única responsabilidad.
- **O**: Open-closed: Una clase debe estar abierta para la extensión, y cerrada para los cambios de código.
- **L**: Liskow substitution: Objetos de un programa deberían poder ser cambiados por instancias de subclases sin alterar el correcto funcionamiento de ese programa.
- **I**: Interface segregation: Varias interfaces bien especificas son mejores que pocas generales.
- **D**: Dependency inversion: Depender de abstracciones y no de clases concretas.

El principio de única responsabilidad

- Responsabilidad: se define como una razón de cambio. Si pensamos que existe mas de una razón que motive el cambio de una clase, entonces esa clase posee mas de una responsabilidad.
- El principio de Única Responsabilidad (SRP) es uno de los principios mas simples de comprender pero el mas difícil de aplicar. Juntar responsabilidades es algo que realizamos naturalmente. Encontrar y separar esas responsabilidades es algo mas relacionado a lo que el diseño de software en verdad realiza. El resto de los principios que discutiremos vuelven de una forma a este.

SMALTALK

- En este lenguaje todo es un objeto.
- Tipado dinamicamente.
- Hay dos tipos de objetos: los que pueden crear instancias de si mismos, y los que no. Los primeros se llaman clases.
- Las clases son objetos capaces de crear instancias y describir su estructura y comportamiento.
- Todo objeto es una instancia de una clase. Las clases son instancias de una clase también (su metaclase).
- Las metaclases son instancias de la clase Metaclass.

JAVASCRIPT (ECMAScript)

- Lenguaje de propósito general.
- Dinámico.
- Basado en objetos (con base en **prototipos** en lugar de clases).
- Multiparadigma.

Prototipos:

- En Javascript no hay clases.
- Cada objeto puede tener su propio comportamiento (métodos).
- Los objetos heredan comportamiento y estado de otros (sus prototipos).
- Cualquier objeto puede servir como prototipo de otro.
- Puedo cambiar el prototipo de un objeto (y así también su comportamiento y estado).
- Se terminan armando cadenas de delegación.