Software construido con objetos 🡪 Un conjunto de objetos que colaboran enviándose mensajes. Todo cómputo ocurre “dentro” de los objetos.

La clave es poder agregar nueva funcionalidad (no pre-vista originalmente), reemplazar objetos o modificarlos y que el sistema “no se entere”, ni se rompa.

Los Sistemas están compuestos (solamente) por un conjunto de objetos que colaboran para llevar a cabo sus responsabilidades.

Responsabilidades de los objetos:

* Conocer sus propiedades
* Conocer otros objetos
* Llevar a cabo ciertas acciones

Aspectos a tener en cuenta:

* NO hay un objeto “main”.
* Cuando codificamos, describimos (programamos) clases.
* Una jerarquía de clases no indica lo mismo que la jerarquía top-down.
* Cuando se ejecuta el programa lo que tenemos son objetos que cooperan y que se crean dinámicamente durante la ejecución del programa.
* Podemos pensar la interacción usuario/software de la misma manera.
* Este mismo modelo nos permite entender otros modelos de computación.
* Este mismo modelo no asume objetos localizados en el mismo espacio de memoria (pueden estar distribuidos).

Impacto en como “pensamos” el software:

* La estructura general cambia: en vez de una jerarquía (Main/procedures/sub-procedures), tenemos una red de “cosas” que se comunican.
* Pensamos en qué cosas hay en nuestro software (los objetos) y como se comunican entre sí.
* Hay un “shift” mental crítico en forma en la cual pensamos el software como objetos.
* - Mientras que la estructura sintáctica es “lineal” el programa en ejecución NO lo es.

OBJETO:

* Es una abstracción de una entidad del dominio del problema. (Ej.: persona)
* Puede representar también conceptos del espacio de la solución. (Ej.: archivos)

Propiedades:

* + 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: determina su conocimiento.

* Está dado por:
  + Propiedades básicas (intrínsecas) del objeto.
  + Otros objetos con los cuales colabora para llevar a cabo sus responsabilidades.
* Se mantiene en las variables de instancia (v.i.) del objeto.
* Es privado del objeto. Ningún otro objeto puede accederlo.

Variables de Instancia (V.I.):

* En general 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.
* El comportamiento indica qué sabe hacer el objeto. Cuáles son sus responsabilidades.
* Se especifica a través del conjunto de mensajes que el objeto sabe responder (protocolo).
* IMPLEMENTACIÓN:
  + 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 al objeto.

Envío de mensaje:

* 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 es mensaje (siempre y cuando exista).
* Como resultado del envío de un mensaje puede retornarse un objeto.

Especificación de un mensaje:

* Nombre: correspondiente al protocolo del objeto receptor.
* Parámetros: información necesaria para resolver el mensaje.
* Cada lenguaje de programación propone una sintaxis particular para indicar el envío de un mensaje.

MÉTODO:

* Es la contraparte funcional del mensaje.
* Expresa la forma de llevar a cabo la semántica propia de un mensaje particular (el cómo).
* Puede realizar 3 cosas:
  + Modificar el estado interno del objeto.
  + Colaborar con otros objetos (enviándoles mensajes).
  + Retornar y terminar.
* Entrada/Salida:
  + En un sistema diseñado correctamente, un objeto no debería realizar ninguna operación vinculada a la interfaz o a la interacción.
  + En la mayoría de los entornos de desarrollo es hasta imposible hacerlo, y en nuestro caso lo será.
  + ¿Qué ganamos? 🡪 Poder cambiar el estilo o el dispositivo de interacción sin necesidad de tocar el código que pasa a ser independiente de la interfaz.

Formas de Conocimiento:

* Para que un objeto conozca a otro lo debe poder “nombrar”. Decimos que se establece una ligadura (binding) entre un nombre y un objeto.
* Formas de conocimiento o tipos de relaciones entre objetos.
  + Conocimiento Interno: Variables de instancia.
  + Conocimiento Externo: Parámetros.
  + Conocimiento Temporal: Variables temporales.
* Además, existe una cuarta forma de conocimiento especial: las pseudo-variables (como “this” o “self”).

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

* Características:
  + 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. Es el objeto quien decide qué se publica.
  + Facilita modularidad y reutilización.

CLASES 🡪 Es una descripción abstracta de un conjunto de objetos.

* Roles:
  + Agrupan el comportamiento común a sus instancias.
  + Definen la forma de sus instancias.
  + Crean objetos que son instancia de ellas.
* En consecuencia, todas las instancias de una clase se comportan de la misma manera.
* Cada instancia mantendrá su propio estado interno.
* Especificación 🡪 Por medio de un nombre, el estado o estructura interna que tendrán sus instancias y los métodos asociados que definen el comportamiento.

Creación de Objetos:

* Comúnmente se utiliza la palabra reservada NEW para instanciar nuevos objetos.
* ¿Quién crea objetos? Otros objetos.
* ¿Cuándo los crea? Cuando sea necesario.

Instanciación de Objetos:

* Es el mecanismo de creación de objetos.
* Los objetos se instancian a partir de un molde.
* La clase funciona como molde.
* Un nuevo objeto es una instancia de una clase.
* Todas las instancias de una misma clase:
  + Tendrán la misma estructura interna.
  + Responderán al mismo protocolo (los mismos mensajes) de la misma manera (los mismos métodos).

\*Los objetos se parecen a “registros”, pero:

- Están encapsulados

- Contienen comportamiento

- Conocen a otros objetos mediante referencias y no mediante datos

\*

Sub-clasificación (subclases) 🡪 Comportamiento común entre las clases.

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

Objetivos de jerarquías:

* Entender mejor el dominio.
* Reducir algo de código y descripciones redundantes.
* Aprovechar mejor el polimorfismo.

Herencia 🡪 Mecanismo por el cual las subclases reutilizan el comportamiento y estructura de su superclase. Pueden:

* Crear una nueva clase como extensión de otra.
* Diseñar e implementar sólo la diferencia que presenta la nueva clase.
* Factorizar similitudes entre clases.
* Tipos:
  + Herencia de Comportamiento
    - Una subclase hereda todos los métodos definidos en su superclase.
    - Las subclases pueden redefinir el comportamiento de su superclase.
  + Herencia de Estructura
    - No hay forma de restringirla.
    - No es posible redefinir el nombre de un atributo que se hereda.

Método Lookup:

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

Clases Abstractas 🡪 Es una clase que no puede ser instanciada.

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

Polimorfismo:

* Dos o más objetos son polimórficos con respecto a un mensaje, si todos pueden entender ese mensaje, aún cuando cada uno lo haga de un modo diferente.
* El overloading es una forma particular de polimorfismo 🡪 Cuando dentro de una misma clase hay métodos con igual nombre, pero con argumentos de distintos tipos.
* Ventajas:
  + Economía de nombres
  + Código genérico
  + Objetos desacoplados
  + Objetos intercambiables
  + Programar por protocolo, no por implementación
  + Bien usado, implica menos código.

Binding 🡪 Asociación de operadores y operandos.

* Dinámico: En el momento de la ejecución.

Bug (Error):

* Un programa que no hace lo que se espera que haga,
* que lo hace mal o
* que directamente falla.

Testear 🡪 Asegurarme tanto como sea posible que el programa está libre de errores.

* ¿Para qué? 🡪 Como programadores queremos encontrar los errores antes de que lo haga el usuario.
* ¿Quién lo hace? 🡪 El principal responsable es el programador. A veces necesitamos ayuda del usuario o de expertos en escribir test.
* ¿Cuándo lo hacemos? 🡪 Lo antes posible.

¿Por qué no testeamos (o lo hacemos mal)?

* Lo dejamos para el final
* Hay muchas combinaciones que considerar
* Requiere planificación, preparación y recursos adicionales
* Es una tarea repetitiva, y nos parece poco interesante.
* Creemos que es tarea de otro
* Creemos que alcanza con “programar bien”
* El objetivo de testear es encontrar bugs (de nuestro programa)

Tipos de Test:

* Funcionales
* No funcionales
* De unidad:
  + Asegura que la unidad mínima 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 el valor esperado en cada caso.
* De integración
* De regresión
* Punta a punta
* Automatizados:
  + Se utiliza software para guiar la ejecución de los test y controlar los resultados.
  + Requiere que diseñemos, programemos y mantengamos programas “test” (para nosotros serán objetos).
  + Suele basarse en herramientas que resuelven gran parte del trabajo.
  + Una vez escritos, los puedo reproducir a costo mínimo, cuando quiera.
  + Los test son “parte del software” (y un indicador de su calidad).
* De carga
* De performance
* De aceptación
* De UI
* De accesibilidad
* Alpha y Beta tests
* A/B
* …

jUnit 🡪 Es un framework, en Java, para automatizar la ejecución de test de unidad.

* Ayuda a escribir test útiles.
* Cada test se ejecuta independientemente de otros (aislados).
* jUnit detecta, recolecta, y reporta errores y problemas.
* xUnit es un nombre genérico, lo que aprendemos podemos llevarlo a otros lenguajes.

Anatomía de un test suite jUnit:

* Una clase de test por cada clase a testear
* Importamos las partes de JUnit que necesitamos
* Un método que prepara lo que necesitan los test, y quedan en v.i. (@BeforeEach / setUp())
* Uno o varios métodos de test por cada método a testear (@Test)
* Un método que limpia lo que se preparo (si es necesario)

Independencia entre test:

* No puedo asumir que otro test se ejecutó antes o se ejecutará después del que estoy escribiendo.
* Por cada método de test:
  + Se crea una nueva instancia de nuestra clase de test
  + Se prepara (método @BeforeEach)
  + Se ejecuta el test y se registran errores y fallas

Test runner:

* Se encarga de verificar cuáles son las clases que tienen test (busca clases con anotaciones (@..))
* Cuáles son los test
* Instancia los objetos y le envía los mensajes necesarios a los test para que se ejecuten y devuelvan los resultados.

¿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

Estrategia general:

1. Pensar qué podría variar (qué valores puede tomar) y que pueda causar un error o falla.
2. Elegir valores de prueba para maximizar las chances de encontrar errores haciendo la menor cantidad de pruebas posibles.
3. Nos vamos a enfocar en dos estrategias:
   1. Particiones equivalentes: 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).
      1. Si se trata de valores en un rango, tomo un caso dentro y uno por fuera en cada lado del rango.
      2. Si se trata de casos en un conjunto, tomo un caso que pertenezca al conjunto y uno que no.
   2. Valores de borde: Los errores ocurren con frecuencia en los límites y ahí es donde los vamos a buscar.
      1. Intentamos identificar bordes en nuestras particiones de equivalencia y elegimos esos valores.
      2. Buscamos los bordes en propiedades del estil: velocidad, cantidad, posición, tamaño, etc..
      3. Buscar valores como: primero/último, máximo/mínimo, arriba/abajo, principio/fin, vació/lleno, etc..

Testing en OO1:

* Testear es asegurarnos de que nuestros objetos hacen lo que se espera, como se espera.
* Escribir test de unidad (JUnit) es parte de “programar”.
* Escribir test nos ayuda a entender que se espera de nuestros objetos.
* Podemos hacer TDD o no, pero siempre escribimos test.

**Colecciones**: Para representar relaciones 1 a muchos (un objeto que conoce a muchos objetos)

Librerías/framework de colecciones:

* Todos los lenguajes OO ofrecen librerías de colecciones
  + Buscan abstracción, interoperabilidad, performance, reúso, productividad
* Las colecciones admiten, generalmente, contenido heterogéneo en términos de clase, pero homogéneo en términos de comportamiento.
* La librería de colecciones de Java se organiza en términos de:
  + Interfaces: Representa la esencia de distintos tipos de colecciones.
  + Clases abstractas: Capturan aspectos comunes de implementación.
  + Clases concretas: Implementaciones concretas de las interfaces.
  + Algoritmos útiles (implementados como métodos estáticos).

Algunos tipos populares (interfaces)

* List (java.util.List)
  + Sus elementos están indexados por enteros de 0 en adelante
* Set (java.util.Set)
  + No admite duplicados, sus elementos NO están 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 en que se recuperan los objetos (LIFO, FIFO, por prioridad, etc.)

Generics y polimorfismo

* Las colecciones admiten cualquier objeto en su contenido
* 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

Colecciones en OO1:

* El rol principal de las colecciones es mantener relaciones entre objetos.
* También vamos a utilizarlas como nuestros repositorios de objetos.

Operaciones frecuentes:

* Ordenar respeto a algún criterio
  + En Java, para ordenar nos valemos de un comparador.
    - Comparador = objeto que encapsula la lógica de comparación de elementos en la colección. Es un objeto de una clase que implementa la interfaz Comparator<>.
  + Los TreeSet usan un comparador para mantenerse ordenados.
  + Para ordenar List, le enviamos el mensaje sort, con un comparador como parámetro.
* Recorrer y hacer algo con todos sus elementos
  + Es algo frecuente.
  + El loop de control es un lugar más donde cometer errores.
  + El código es repetitivo y queda atado a la estructura/tipo de la colección
  + Para solucionar esto utilizamos un iterador
    - Todas las colecciones entienden el mensaje iterator()
    - Un Iterator encapsula:
      * Como recorrer una colección particular
      * El estado de un recorrido
    - No nos interesa la clase del iterador (son polimórficos)
    - El loop for-each esconce la existencia del iterador
* Encontrar un elemento (max, min, dni=xxx, …)
* Filtrar para quedarme solo con algunos elementos
* Reducir (promedio, suma, …)

\*Nos interesa escribir código que sea independiente (tanto como sea posible) del tipo de colección que utilizamos.\*

Precauciones:

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

Expresiones Lambda (clausuras)

* + - * Son métodos anónimos (no tienen nombre, no pertenecen a ninguna clase)
      * Útiles para:
  + Parametrizar lo que otros objetos deben hacer
  + Decirles a otros objetos que me avisen cuando pase algo (callbacks)
    - * Sintáxis: (p1, p2) -> {p1.mensaje1(), p2.mensaje2()}

forEach(lambda) 🡪 recorre la colección y para cada uno de sus elementos hace lo que se indica en la expresión lambda pasada como parámetro.

Streams 🡪 Objetos que permiten procesamiento funcional de colecciones. Las operaciones de combinan para formar pipelines (tuberías). Para obtenerlos le envío el mensaje stream() a una colección.

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

Pipelines:

* Para construir un pipeline encadeno envíos de mansajes
  + Una fuente, de la que se obtienen lo elementos.
  + Cero o más operaciones intermedias, que devuelven un nuevo stream.
  + Operaciones terminales, que retornan un resultado.
* La operación terminal guía el proceso

Algunos mensajes que entienden los streams:

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 u elemento y resulta en true o false.

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.

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.
* Recibe como parámetro un objeto Collector
  + Podemos programar uno, pero solemos utilizar los que “fabrica” Collectors (Collectors.toList(), Collectors.counting(), …).

**Análisis y Diseño O.O.**

Análisis 🡪 El análisis pone énfasis en una investigación del problema y los requisitos, en lugar de ponerlo en la solución.

Diseño 🡪 El diseño pone énfasis en una solución conceptual que satisface los requisitos, en lugar de ponerlo en la implementación.

Casos de Uso: Se definen para satisfacer los objetivos de usuario o actores principales.

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

\*El actor principal y los objetivos de usuario dependen del límite del Sistema.\*

* Diagrama: Proporciona información visual concisa del sistema, los actores extremos y cómo lo utilizan.
* Los casos de uso y los actores deben tener “buenos” nombres
  + Casos de Uso: Verbo sustantivo
  + Actor: rol – subsistema – dispositivo

Tipos de formalidad – Casos de Uso:

* Breve: Es un resumen conciso que no ocupa más de un párrafo. Se describe el escenario principal con éxito (curdo 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 alternativos). Cuenta con otras secciones como pre y post condiciones, curso de error, etc..

Diagramas de Secuencia del Sistema (DSS): Se derivan de los casos de uso.

* Muestran, para un escenario de un caso de uso, los eventos que los actores generan. Estos eventos de entrada del sistema invocan operaciones del sistema.

*Modelo del Dominio:*

* 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.\*

* Consejos:
  + Usar nombres del dominio del problema, no de la solución.
  + Omitir detalles irrelevantes.
  + No inventar nuevos conceptos (evitar sinónimos).
  + Descubrir conceptos del mundo real.
* Estrategias.
  + Identificación de frases nominales:
    - Encontrar conceptos (y sus atributos) mediante la identificación de los sustantivos en la descripción textual del dominio del problema.
  + Utilización de una lista de categorías de clases conceptuales:

|  |  |
| --- | --- |
| **Categoría de Clase Conceptual** | **Ejemplos** |
| Objeto físico o tangible |  |
| 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:

1. Listar los conceptos candidatos.
2. Graficarlos en un Modelo del Dominio.
3. Agregar atributos a los conceptos: Se identifican los atributos que son necesarios para satisfacer los requerimientos de información de los casos de uso en desarrollo. (Deberían ser atributos simples o tipos de datos primitivos).

¡¡Relacionar las clases conceptuales con asociaciones, no con atributos!!

Represente lo que inicialmente podría considerarse un tipo de dato primitivo como una clase conceptual, si:

* + - 1. Está compuesto de secciones separadas
      2. Tiene operaciones asociadas
      3. Tiene otros atributos
      4. Es una cantidad con una unidad
      5. Es una abstracción de uno o más tipos con esas cualidades

1. Agregar asociaciones entre conceptos:

|  |  |
| --- | --- |
| **Categoría** | **Ejemplo** |
| 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 |  |

* + - Focalizar las asociaciones que necesitan se preservadas por un lapso de tiempo.
    - Evitar mostrar asociaciones redundantes o derivadas.
    - Es más importante identificar clases conceptuales que asociaciones conceptuales.
    - Demasiadas asociaciones pueden oscurecer el Modelo el Dominio.
    - Recuerde agregar multiplicidades.
    - Recuerde agregar roles.
  + Agregue clases conceptuales de especificación cuando necesite la descripción de un artículo o servicio, o si al eliminar las instancias que describen, se pierde información.
* Revisar: Sinónimos, conceptos no considerados, relaciones, multiplicidades, roles.

Contratos de las Operaciones: 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:
  + Operación: nombre de la operación y parámetros.
  + Referencias Cruzadas (opcional): Casos de uso en los que puede tener lugar la operación.
  + Precondiciones: Suposiciones relevantes sobre el estado del sistema o de los objetos del Modelo de Dominio, antes de la ejecución de la operación.
    - No se validarán dentro de la operación, sino que se asumirán como verdaderas.
    - Son suposiciones no triviales que el lector debe saber que se hicieron.
    - Son declarativas 🡪 Ej.: El cliente *está* registrado como usuario.
  + 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:
      * Creación y eliminación de instancias
      * Modificación de atributos
      * Creación o ruptura de asociaciones
    - Son declarativas 🡪 Ej.: Se *creó* un nuevo cliente.

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 representando las clases que serán implementadas.

🡪 Crear diagramas de interacción requiere la aplicación de Principios o Heurísticas para la Asignación de Responsabilidades.

**Heurísticas para Asignación de Responsabilidades (HAR)**

Responsabilidades de los objetos:

* Conocer:
  + Sus datos privados encapsulados.
  + Sus objetos relacionados.
  + Cosas derivables o calculables
* Hacer
  + Algo por si mismos.
  + Iniciar una acción en otros objetos.
  + Controlar o coordinar actividades de otros objetos.
    - * La habilidad para asignar responsabilidades es extremadamente importante en el diseño.
      * La asignación de responsabilidades generalmente ocurre durante la creación de diagramas de interacción.
      * Heurísticas:
  + 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 sus responsabilidades, un objeto puede requerir de información que se encuentra dispersa en diferentes clases 🡪 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 objetos A en forma exclusiva.

La intención del Creador es encontrar una clase que necesite conectarse al objeto creado en alguna situación. Eligiéndolo como el creador se favorece el bajo acoplamiento.

* + Controlador: Asignar la responsabilidad de manejar eventos del sistema a una clase que representa:
    - El sistema global, dispositivo o subsistema.

La intención del Controlador es encontrar manejadores de los eventos del sistema, sin recargar de responsabilidad a un solo objeto y manteniendo alta cohesión.

* + Bajo Acoplamiento: Asignar responsabilidades de manera que el acoplamiento permanezca lo más bajo posible.
    - El Acoplamiento es una medida de dependencia de un objeto con otros. Es bajo si mantiene pocas relaciones con otros objetos.
      * 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, sino que debe incluirse como principio de diseño que influye en la elección de la asignación de responsabilidad.
  + Alta Cohesión: Asignar responsabilidades de manera que la cohesión permanezca lo más fuerte posible.
    - La Cohesión es una medida de la fuerza con la que se relacionan las responsabilidades de un objeto, y la cantidad de ellas.
      * Ventaja: clases más fáciles de mantener, entender y reutilizar.
      * El nivel de cohesión no se puede considerar de manera aislada a otras responsabilidades y otras heurísticas, como Experto y Bajo Acoplamiento.

Realización de casos de uso:

* Los casos de uso sugieren los eventos del sistema que se muestran en los diagramas de secuencia del sistema.
* En los contratos de las operaciones utilizando conceptos del Modelo del Dominio, se describen los efectos que dichos eventos (operaciones) producen en el sistema.
* Los eventos del sistema representan los mensajes que dan inicio a Diagramas de Secuencia del Diseño, mostrando las interacciones entre los objetos del sistema.
* Los objetos con sus métodos y relaciones se muestran en el Diagrama de Clases del Diseño (basado en el Modelo del Dominio).

Diagramas de Interacción:

* Cree un diagrama de secuencia por cada operación del sistema en desarrollo (la operación es el mensaje de partida en el diagrama).
* Si el diagrama queda complejo, sepárelo en diagramas menos complejos (uno por cada escenario).
* Use el contrato de la operación como punto de partida; piense en objetos que colaboran para cumplir la tarea (la mayoría de estos objetos están definidos en el Modelo del Dominio).
* Aplique las HAR para obtener un mejor diseño.

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 nombre 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.

🡪 Agregar al Modelo Conceptual, los mensajes/métodos que surjan en los diagramas de interacción de cada operación 🡨

\*Transformación de los diseños en código 🡪 Mapear los artefactos de diseño a código orientado a objetos.\*

Clases 🡪 Clases

Atributos 🡪 Variables simples

Asociaciones 🡪 Variables de referencia

Métodos 🡪 Métodos

Multiobjetos 🡪 Colecciones

Extendiendo el Modelo Conceptual:

* Nuevos conceptos pueden ser identificados y agregados al modelo.
* Descubriendo nuevas clases para evitar preguntar por el tipo o valor de un atributo 🡪 Polimorfismo.

Clases Conceptuales:

* Entidades (Entity): 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.
  + No viven por si mismo, necesitan una entidad base, son intercambiables. Persisten adjunto a su base, no separadamente.
  + Inmutables (NO le defino setters).

Polimorfismo 🡪 Cuando el comportamiento varía según el tipo, asigne la responsabilidad a los tipos/las clases para las que varía el comportamiento.

* + - * No permite sustituir objetos que tienen idéntica interfaz.

Principios SOLID 🡪 Relacionados a las HAR, para un buen estilo de DOO. Promueven Alta Cohesión y Bajo Acoplamiento.

* S (SRP: The Single-Responsability Principle) Principio de Responsabilidad Única 🡪 Una clase debería cambiar por una sola razón. Debería ser responsable de únicamente una tarea, y ser modificada por una sola razón (alta cohesión).
* O (OCP: The Open-Closed Principle) 🡪 Entidades de software deberían ser:
  + “Abiertas” para extensión: ser capaz de añadir nuevas funcionalidades.
  + “Cerradas” para modificación: al añadir la nueva funcionalidad no se debe cambiar el diseño existente.
* L (LSP: The Liskov Substitution Principle) 🡪 Los objetos de un programa deben ser intercambiables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa.
* I (ISP: The Interface-Segregation Principle) 🡪 Las clases que tienen interfaces “voluminosas” son clases cuyas interfaces no son cohesivas.
  + Las clases no deberían verse forzadas a depender de interfaces que no utilizan. Cuando creamos interfaces (protocolos) para definir comportamientos, las clases que las implementan, no deber estar forzadas a incluir métodos que no va a utilizar.
* D (DIP: The Dependency-Inversion Principle) 🡪
  + Los módulos de alto nivel de abstracción no deben depender de los de bajo nivel
    - Módulo de alto nivel: Se refieren a los objetos que definen qué es y qué hace el sistema.
    - Módulo de bajo nivel: No están directamente relacionados con la lógica de negocio del programa (no definen dominio).
  + Las abstracciones no deben depender de detalles, sino al revés.
    - Abstracciones: Se refieren a protocolos (interfaces) o clases abstractas.
    - Detalles: Son las implementaciones concretas.

\*Ser capaz de invertir una dependencia es lo mismo que ser capaz de intercambiar una implementación concreta por otra implementación concreta cualquiera, respecto a la misma abstracción.\*

Reúso de Código:

* Herencia:
  + Herencia Total: debo conocer todo el código que se hereda 🡪 Reutilización de Caja Blanca.
  + Usualmente debemos redefinir o anular métodos heredados.
  + Los cambios en la superclase se propagan automáticamente a las subclases.
  + Herencia de Estructura vs. Herencia de Comportamiento.
  + Es útil para extender la funcionalidad del dominio de aplicación
* Composición:
  + 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.

Herencia vs. Composición de Objetos:

* + - * Las clases y los objetos creados mediante herencia están estrechamente acoplados ya que cambiar algo en la superclase afecta directamente a la/ñas 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.

JavaScript (ECMAScript):

* Naturaleza basada en prototipos.
* Características:
  + Lenguaje de propósito general
  + Dinámico
  + Basado en objetos (con base en prototipos en lugar de clases)
  + Multiparadigma
  + Se adapta a una amplia variedad de estilos de programación
  + Pensado originalmente para scripting de páginas web
  + Con una fuerte adopción en el lado del servidor (NodeJS)

Prototipos:

* En JS no tengo clases
* La forma más simple de crear un objeto es mediante la notación literal
* Cada objeto puede tener su propio comportamiento (métodos)
* Los objetos pueden heredar comportamiento, estructura y estado de otros (sus prototipos)
* Cualquier objeto puede servir como prototipo (del cual crear otros)
* Una vez que tengo un objeto (prototipo) puedo crear otros a partir de él con Object.create(…)
* Termino armando cadenas de delegación

Smalltalk:

* OO de pies a cabeza.
* Ambiente que invita a un enfoque exploratorio de desarrollo.
* Fuente de muchas de las ideas que hoy vemos en otros lenguajes y ambientes.
* Características:
  + Lenguaje OO puro – todo es un objeto (Incluso las clases)
  + Tipado dinámico
  + Propone una estrategia exploratoria (construccionista) al desarrollo de software
  + El ambiente es tan importante como el lenguaje
    - Está implementado en Smalltalk
    - Ricas librerías de clases
    - Todo su código fuente disponible y modificable
    - Tiene su propio compilador, debugger, editor, inspector, etc.
    - Es extensible
  + Sintaxis minimalista (con sustento en su foco educativo)
  + Fuente de inspiración de casi todo lo que vino después