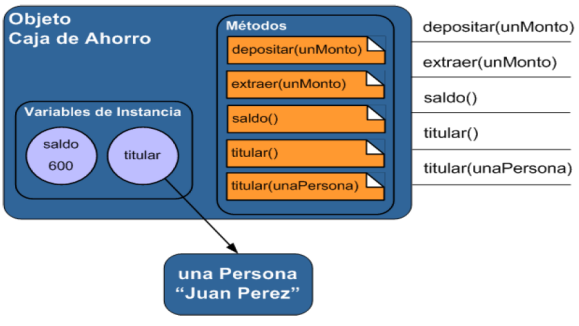
Software construido con objetos → es un conjunto de objetos que colaboran enviándose mensajes para llevar a cabo sus responsabilidades

- Se puede agregar nueva funcionalidad sin que el sistema se "entere" o se rompa
- Los objetos son *responsables* de
  - Conocer sus propiedades,
  - Conocer otros objetos
  - Llevar a cabo ciertas acciones
- Aspectos de interés:
  - No hay un objeto "main"
    - Más allá de que en el código haya un "cuerpo" de código semejante al main procedural (incluso con ese nombre), no existe el equivalente "estructural" al main que "conduce" el flujo en cada ejecución
  - Cuando codificamos , describimos 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 se crean dinámicamente durante su ejecución
  - Mientras que la estructura sintáctica es "lineal", el programa en ejecución no lo és

Objeto → abstracción de una entidad del dominio del problema (Persona, Producto, Auto, etc.). Puede representar también conceptos del espacio de la solución (estructuras de datos, etc)

- Características:
  - Un objeto tiene:
    - **Identidad** → para distinguir un objeto de otro
    - **Conocimiento** → en base a sus relaciones con otros objetos y su estado interno
      - Para que un objeto conozca a otro lo debe poder "nombrar". Se establece una ligadura (binding) entre un nombre y un objeto
      - Hay 3 formas de conocimiento:
        - 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")
    - **Comportamiento** → conjunto de mensajes que un objeto sabe responder. Básicamente, qué sabe hacer el objeto
      - Se especifica a través del conjunto de mensajes que el objeto sabe responder→*protocolo*
      - La realización (manera en que un objeto responde) 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.
      - Ejemplo:



Observese la variable "titular" apuntando a un objeto Persona

    - El estado interno:
        - Determina su conocimiento
        - Está dado por:
            - Propiedades básicas del objeto
            - Otros objetos con los cuales colabora para llevar a cabo sus responsabilidades
        - Se mantiene en las variables de instancia del objeto.
        - Es privado del objeto. Ningún otro objeto puede accederlo
    - ¿Cómo se crean los objetos? Instanciación → mecanismo de creación de objetos
        - Palabra reservada *new*
        - 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 de la misma manera
    - Para que un objeto esté listo para llevar a cabo sus responsabilidades hay que inicializarlo→darle valor a sus variables
        - Se hace mediante → **constructores**
- Responsabilidades de los objetos
    - **Conocer**
        - Sus datos privados encapsulados
        - Sus objetos relacionados
        - Cosas derivables o calculables
    - **Hacer**
        - Hacer algo él mismo
        - Iniciar una acción en otros objetos
        - Coordinar actividades de otros objetos

Variables de instancia → referencias (punteros) a otros objetos con los cuales el objeto colabora

- Algunas pueden ser atributos basicos
- Un objeto O1 puede mandarle mensajes a otros objetos O2 y O3 porque “los conoce ”, o sea hay una variable en O1 que APUNTA a O2 y otra a O3 (o la misma variable que cambia de valor en diferentes momentos)

Métodos → Es la contraparte funcional del mensaje

- Expresa la forma de llevar a cabo la semántica propia de un mensaje particular (el cómo)
- ¿Qué hace un método? 3 cosas:
    - Modificar el estado interno del objeto
    - Colaborar con otros objetos (enviándoles mensajes)
    - Retornar y terminar

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

- 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 e instancias → descripción abstracta de un conjunto de objetos

- Las clases cumplen tres roles:
  - Agrupan el comportamiento común a sus instancias.
  - Definen la forma de sus instancias
  - Crean objetos que son instancia de ellas
    - En consecuencia todas las instancias de una clase se comportan de la misma manera
      - Pero cada instancia mantendrá su propio estado interno
- Las clases se especifican por medio de un nombre, el estado o estructura interna que tendrán sus instancias y los métodos asociados que definen el comportamiento

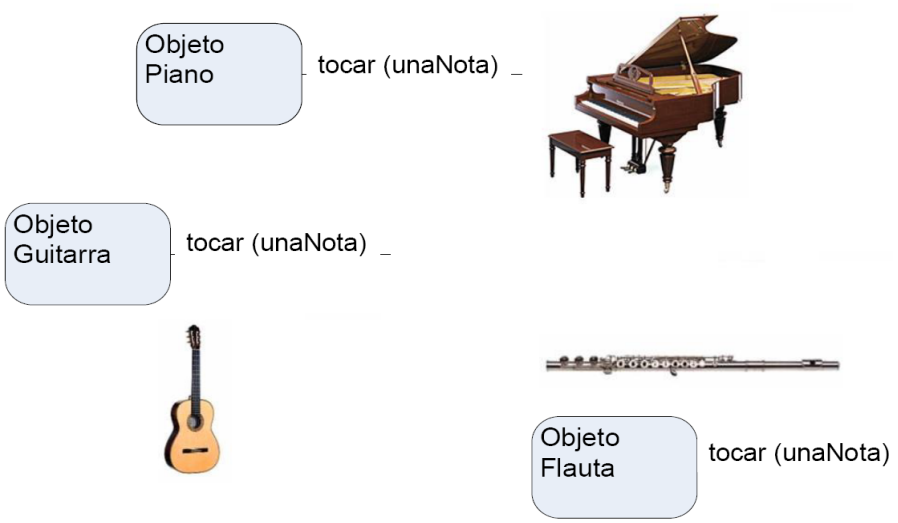
Sobre la E/S de información:

- En un sistema diseñado correctamente , un objeto no debería realizar ninguna operación vinculada a la interfaz o a la interacción (esperar un "input")
- En la mayoría de los entornos de desarrollo es hasta imposible hacerlo
- Ventaja → Poder cambiar el estilo o el dispositivo de interacción sin necesidad de tocar el código, que pasa a ser independiente de la interfaz

Cuando las clases tienen comportamiento común → **Subclasificación**

- 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"
- ¿Para qué sirven las jerarquías?
  - Entender mejor el dominio
  - Reducir algo de codigo y descripciones redundantes
  - Aprovechar mejor el polimorfismo
- **Herencia** → mecanismo por el cual las subclases reutilizan el comportamiento y estructura de su superclase.
  - Permite:
    - Crear una nueva clase como refinamiento de otra.
    - Diseñar e implementar sólo la diferencia que presenta la nueva clase.
    - Factorizar similitudes entre clases.
  - Implica:
    - Herencia de comportamiento
      - Una subclase hereda todos los métodos definidos en su superclase.
      - Las subclases pueden redefinir el comportamiento de su superclase.
    - Herencia de estructura
      - No hay forma de restringirla.
      - No es posible redefinir el nombre de un atributo que se hereda.
  - Cuando se le envía un mensaje a una subclase de una herencia → **Method Lookup**
    - 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** → no pueden ser instanciadas
    - ¿Y para qué carajo sirven si no las puedo instanciar?
      - 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 (métodos abstractos)
        - Las subclases completan las piezas faltantes, o agregan variaciones a las partes existentes
    - **Polimorfismo** → varios objetos pueden entender un mismo mensaje, aún cuando cada uno responde de manera distinta
        - Ventajas:
            - Código genérico
            - Objetos desacoplados
            - Objetos intercambiables
            - Objetos reutilizables
            - Programar por protocolo, no por implementación
            - Bien usado, implica menos código



    - **Binding dinámico** → mecanismo por el cual se escoge, en tiempo de ejecución, el método que responderá a un determinado mensaje
        - Útil cuando este no puede ser determinado en tiempo de compilación
        - Ejemplo → Figura figuras[10];
            - Figuras es un arreglo que dentro podía tener triángulos, cuadrados, etc.
            - En lugar de hacer:

```
for (i=0; i<10; i++) {
    if (figura[i].getClass().equals("Cuadrado")
            dibujarCuadrado();
    if (figura[i].getClass().equals("Triángulo")
            dibujarTriángulo();
}
```

            - Directamente se hace:

```
for (i=0; i<10; i++)
    figura[i].dibujar();
```

            - Y cada figura sabe ejecutar su propio método, sin tener que chequearlo en el código

Testing

- ¿*Por qué* testear?
    - Encontrar bugs
        - Porque no alcanza con "programar bien"
- ¿*Cómo* testear?
    - Con un propósito (buscamos algo)
    - Pensamos por qué testear algo y con qué nivel queremos hacerlo

- ¿*Cuándo* testear?
  - Temprano y frecuentemente
  - Y se testea tanto como sea el riesgo del artefacto
- Requiere planificación, preparación y recursos adicionales
- Tipos de tests
  - Tests de unidad
    - Asegura que la unidad mínima de nuestro programa funciona correctamente, y aislada de otras unidades
      - En nuestro caso, la unidad de test es el método
    - Testear un método es confirmar que el mismo acepta el rango esperado de entradas, y que retorna el valor esperado en cada caso
  - Tests automatizados
    - Se utiliza software para guiar la ejecución de los tests y controlar los resultados
    - Requiere que diseñemos, programemos y mantengamos programas “tests”
      - En nuestro 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” (y un indicador de su calidad)
  - Tests funcionales
  - Test no funcionales
  - Tests de integración
  - Tests de regresión
  - Test punta a punta
  - Test de carga
  - Test de performance
  - Test de aceptación
  - Test de UI
  - Test de accesibilidad
  - Alpha y beta tests
  - Test A/B
  - …
- Tests de Unidad Automatizados → **jUnit**
  - Framework
  - Ayuda a escribir tests útiles
  - Cada test se ejecuta independientemente de otros (aislados)
  - jUnit detecta, recolecta, y reporta errores y problemas
  - Formato:
    - Una clase de test por cada clase a testear
    - Un método que prepara lo que necesitan los tests → @BeforeEach → setUp()
      - Y queda en variables de instancia
    - Uno o varios métodos de test por cada método a testear
    - Un método que limpia lo que se preparó (si es necesario)
- Los tests deben tener **independencia** entre sí
  - 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 marcado como @BeforeEach)
    - Se ejecuta el test y se registran errores y fallas
- Estrategia:
  - 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

- Enfoque en 2 estrategias:
  - **Particiones equivalentes** → conjunto de casos que prueban lo mismo o revelan el mismo bug → si un ejemplo de una partición pasa el test, el resto también
    - Si se trata de valores en un rango, tomo un caso dentro y uno por fuera en cada lado del rango
      - Ej → la temperatura debe estar entre 0 y 100 → casos: -50, 50 , 150.
    - 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
  - **Valores de borde**
    - Los errores ocurren con frecuencia en los límites y ahí es donde los vamos a buscar
    - Intentamos identificar bordes en nuestras particiones de equivalencia y elegimos esos valores
    - Buscar los bordes en propiedades del estilo: velocidad, cantidad, posición, tamaño , duración, edad, etc.
    - Y buscar valores como: primero/último, máximo/mínimo, arriba/abajo, principio/fin, vacío/lleno, antes/después, junto a, alejado de , etc.

Collections → conjunto de punteros que apuntan a objetos, agrupándolos

- Todos los lenguajes OO ofrecen librerías de colecciones
  - Buscan:
    - Abstracción
    - Interoperabilidad
    - Performance
    - Reuso
      - Las collections permiten escribir código independiente
        - Si uso una lista, el código es igual que si uso un set, lo que permite modificar el programa sin tener que reescribir todo
        - Así no reinventamos la rueda
    - Productividad
- *Objetivo* → mantener relaciones entre objetos
- Las colecciones admiten, generalmente, contenido heterogéneo en términos de clase, pero homogéneo en términos de comportamiento
- *Collections populares:*
  - List (java.util.List) → sus elementos se 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 → ¿lo usa alguien siquiera? Si lo usás hay tabla
  - Queue (java.util.Queue) → maneja el orden en que se recuperan los objetos (LIFO, FIFO, por prioridad, etc.)
- 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
- NO se modifican las collections de otro objeto
  - Solo el dueño las modifica
    - ticket.getItems().add(producto) → hay tabla
- Expresiones Lambda → métodos anónimos (no tienen nombre, no pertenecen a ninguna clase)
  - Útiles para:
    - Parametrizar lo que otros objetos deben hacer

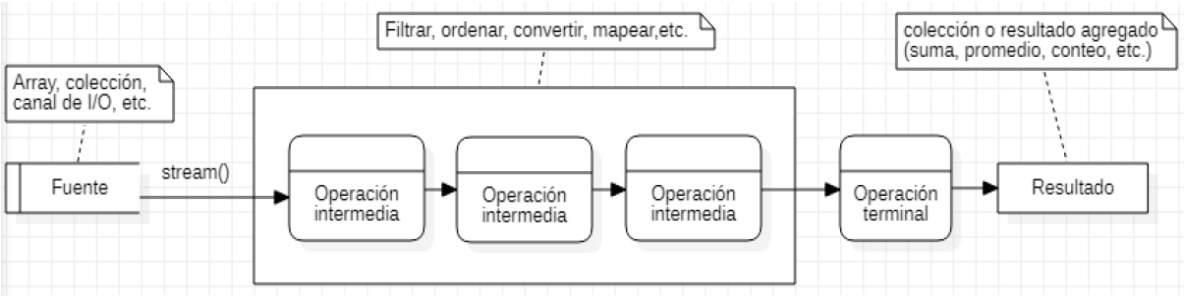
        - Decirle a otros objetos que avisen cuando pase algo (callbacks)

```
clientes.iterator().forEachRemaining(c -> c.pagarLasCuentas());

clientes.forEach(c -> c.pagarLasCuentas());

JButton button = new JButton("Click Me!");
button.addActionListener(e -> this.handleButtonAction(e));
```

- **Streams** → Objetos que permiten procesamiento funcional de colecciones
    - Las operaciones se combinan para formar pipelines (tuberías)
        - Para construir un pipeline encadeno envíos de mensajes
            - **Una fuente** → de la que se obtienen los elementos → list, set, etc.
            - **Cero o más operaciones intermedias** → que devuelven un nuevo stream → map, filter, etc.
            - **Operaciones terminales** → que retornan un resultado → collect, findFirst, etc.
        - La operación terminal guía el proceso



    - Características:
        - 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
        - La forma más frecuente de obtenerlos es vía el mensaje stream() a una colección

Análisis y diseño

- **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
- Se utilizan **Casos de Uso (CU)** → para satisfacer los objetivos de los usuarios o actores principales.
    - Hay que:
        - Identificar los actores principales (y actores secundarios)
        - Identificar los objetivos de usuario de cada actor
        - Definir los casos de uso
    - El diagrama proporciona información visual concisa del sistema, los actores externos y cómo lo utilizan.
    - Los casos de uso y los actores deben tener “buenos” nombres:
        - Casos de uso → Verbo y sustantivo → “Registrar usuario”
        - Actor → rol / subsistema / dispositivo → Usuario, Servidor, etc.
    - Existen tres tipos o *grados de formalidad*:
        - 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

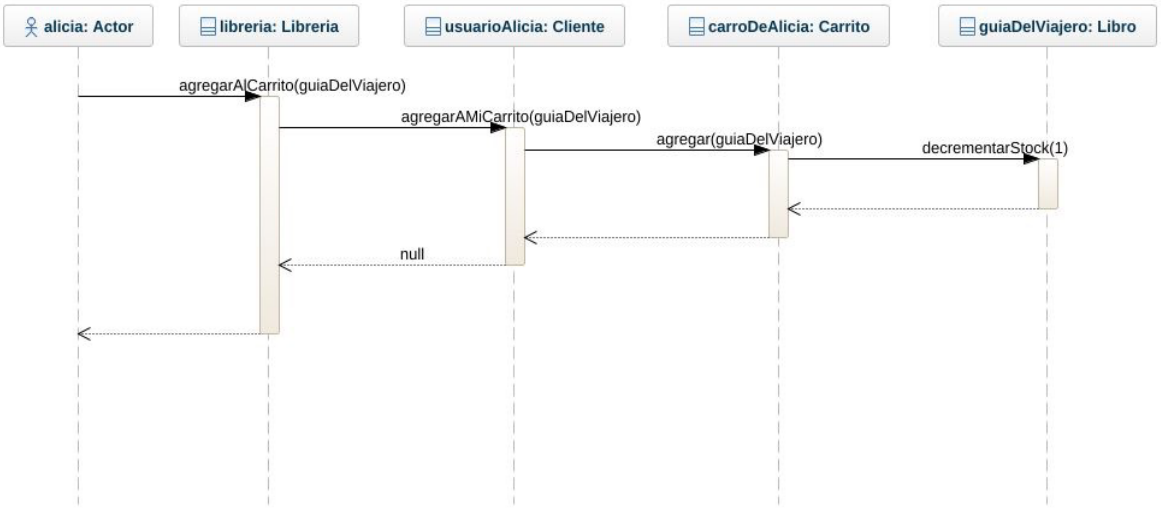
        - Se caracteriza por un estilo informal de escritura.
    - Completo → es el formato más elaborado, ya que se describen con detalle todos los pasos y variaciones (curso normal y alternativos)
        - Cuenta con otras secciones como pre y post condiciones, curso de error, etc.
- **Contratos** → Son una de las formas de describir el comportamiento del sistema en forma detallada. Describen pre y post condiciones para las operaciones.
    - Secciones:
        - 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
            - Se asumen verdaderas, por lo que no se validan
        - Postcondiciones → describen cambios en el estado de los objetos del Modelo del Dominio
    - Ejemplo:

        Operación: checkout pedido (c: Cliente)

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

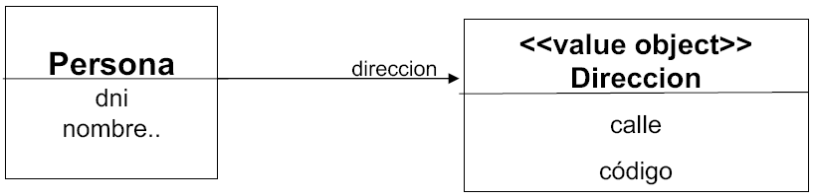
        Postcondiciones:
        - Se creó un nuevo pedido con el carrito, la dirección de entrega y forma de pago.
        - Se agregó el pedido a la colección de pedidos del cliente.
        - Se agregó el pedido a la colección de pedidos de la EmpresaDePedidos.
        - Se vació el carrito del cliente.

- Se utilizan **Diagramas de Secuencia del Sistema (DSS)** → derivan de los CU y describen cómo una acción de un actor genera acciones a lo largo de los objetos del sistema.
    - Ej → el actor genera el evento del sistema “agregar al carrito”:



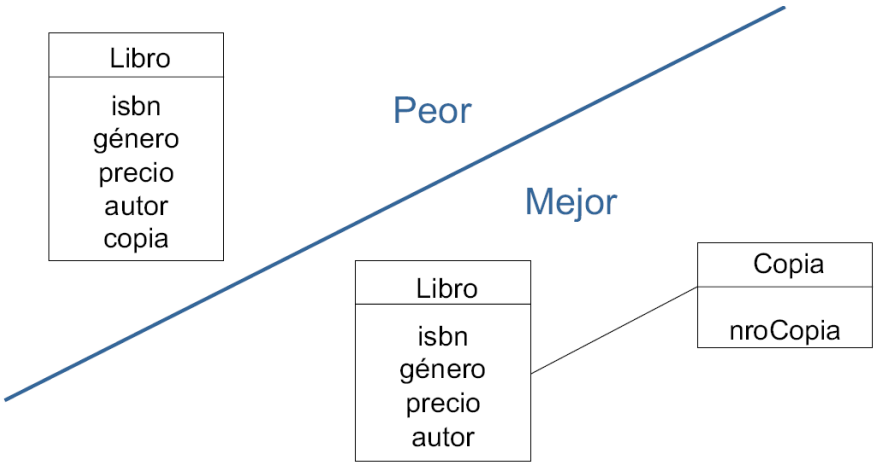
- **Modelo de dominio** → representación visual de las clases conceptuales del mundo real en un dominio de interés
    - **Identificación de clases conceptuales**
        - Es mejor especificar en exceso un modelo del dominio con muchas clases conceptuales de grano fino que especificar por defecto
        - 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.
    - Ejemplo:
      1. El Cliente selecciona un libro para agregar al carrito.
      2. El sistema agrega el libro al carrito.
      3. El carrito agrega al libro y presenta la descripción, precio y suma parcial.
  - **Utilización de una lista de categorías** de clases conceptuales
- Pasos a seguir:
  1. Listar los conceptos candidatos → pueden ser clases o atributos
  2. Graficarlos en un Modelo del Dominio
  3. Agregar atributos a los conceptos
  4. Agregar asociaciones entre conceptos
- **Entidad o Value Object**
  - Los Value Object:
    - Son comparables pro contenido
    - Son inmutables (sin setters)
    - No viven por sí mismos → dependen y persisten con otra entidad
    - Son intercambiables
  - Cómo identificar Value Object → cuando se modela una moneda, fecha, dirección, etc.
  - Cómo representar un Value Object → <<Value Object>> delante del nombre



    - Nótese que dirección no tiene sentido si no está junto a persona. Entonces es un Value Object

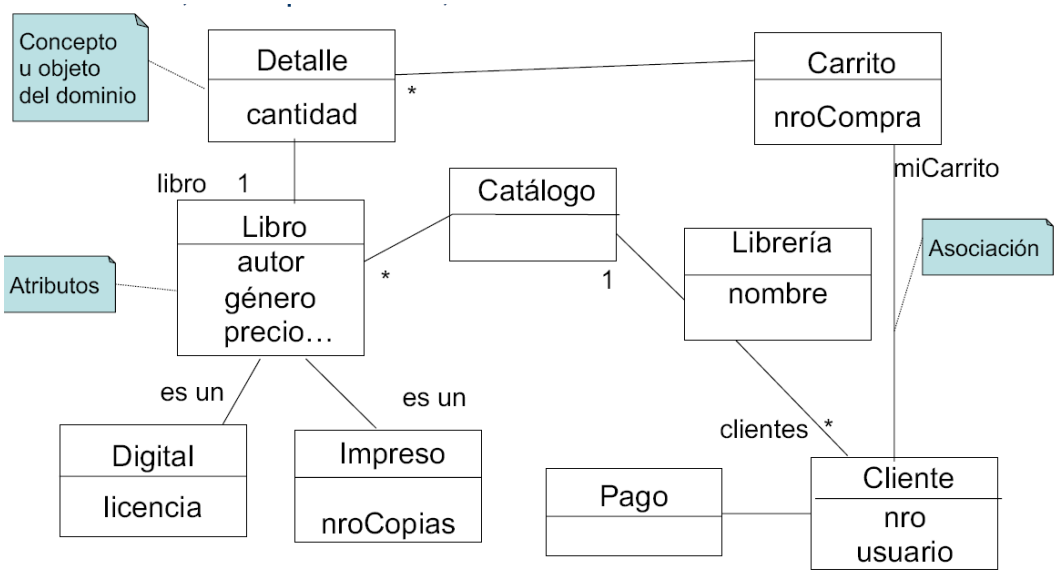
- **Agregar atributos** → para satisfacer los requerimientos de información de los CU
  - Preferiblemente atributos primitivos
  - Si no es primitivo es preferible considerarlo una clase conceptual
- **Agregar asociaciones**
  - ¿Es mejor todo en un concepto o un concepto con especificación?



  - Tips:
    - Focalizar las asociaciones que necesitan ser 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 del Dominio
- Agregar multiplicidades
- Agregar roles

○ Ejemplo (modelo del dominio parcial):

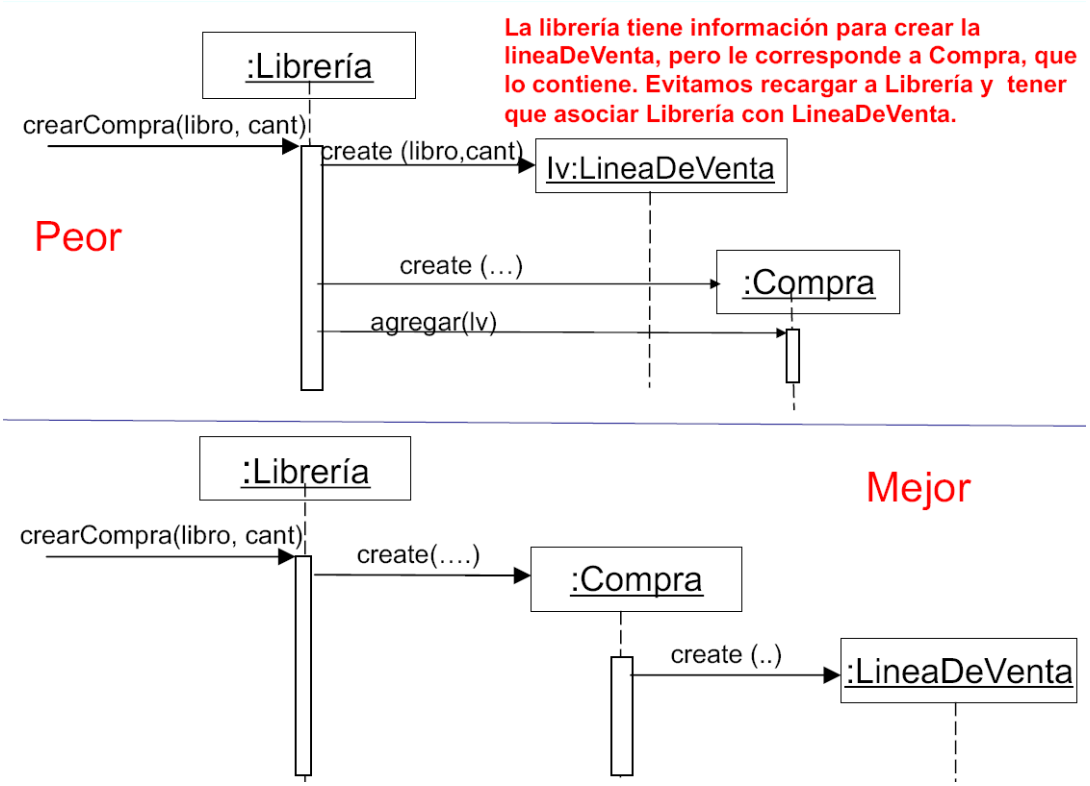


- **Heurísticas para Asignación de Responsabilidades (HAR)**
  - La habilidad para asignar responsabilidades es extremadamente importante en el diseño.
    - La asignación de responsabilidades generalmente ocurre durante la creación de diagramas de interacción.
  - Son:
    - **Experto en Información (Experto)** → asignar una responsabilidad al experto en información
      - Para cumplir con su responsabilidad, un objeto puede requerir de información que se encuentra dispersa en diferentes clases → *expertos en información "parcial"*
      - Ej → ¿Quién tiene la responsabilidad de conocer el monto total de una compra? → La compra.
        - Entonces:
          - LineaDeVenta es responsable de conocer el subtotal por cada ítem
          - EspecificaciónDelProducto es responsable de conocer el precio del item.
      - *Objetivo* → expresar la intuición de que los objeto hacen cosas relacionadas con la información que tienen
    - **Creador** → asignar a la clase B la responsabilidad de crear una instancia de la clase A
      - Esto 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.
      - Ej → ¿Quién debe ser responsable de crear una LineaDeVenta? → La compra
      - *Objetivo* → 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: al sistema global, dispositivo o subsistema
      - Ej → ¿Quién debe ser el controlador de los eventos ingresarLibro o finalizarCompra? → ManejadorCompras, Librería

- *Objetivo* → 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 los cambios en el diseño.
- 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 responsabilidades
- *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
- Ejemplo de bajo acoplamiento y alta cohesión:



■ **Polimorfismo** → cuando el comportamiento varía según el tipo, hay que asignar la responsabilidad a las clases para las que varía el comportamiento.
- Nos permite sustituir objetos que tienen idéntica interfaz.
- Ej → el sistema de venta de libros debe soportar distintas bonificaciones de pago con tarjeta de crédito. → Como la bonificación del pago varía según el tipo de tarjeta → deberíamos asignarle la responsabilidad de la bonificación a los distintos tipos de tarjeta.

■ **“No hables con extraños”** → Hay que evitar diseñar objetos que recorren largos caminos de estructura y envían mensajes (hablan) a objetos distantes o indirectos (extraños).
- Dentro de un método sólo pueden enviarse mensajes a objetos conocidos:
  - Self/this
  - Un parámetro del método
  - Un objeto que esté asociado a self/this
  - Un miembro de una colección que sea atributo de self/this
  - Un objeto creado dentro del método
- Los demás objetos son extraños (strangers)

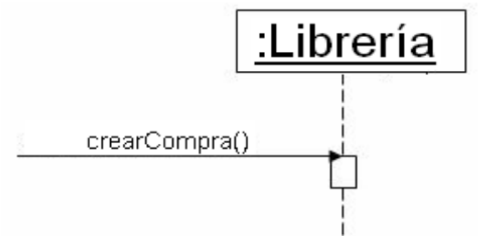
- **Diagramas de interacción**
  - Consejos:
    - Se crea un diagrama por cada operación del sistema en desarrollo
    - Si el diagrama queda complejo, conviene separarlo en diagramas menos complejos (uno por cada escenario).
    - Usar el contrato de la operación como punto de partida
      - Pensar en objetos que colaboran para cumplir la tarea
    - Aplicar HAR para obtener un mejor diseño.
  - Ejemplo → crear una nueva compra:

... Comenzamos con el contrato de la operación.

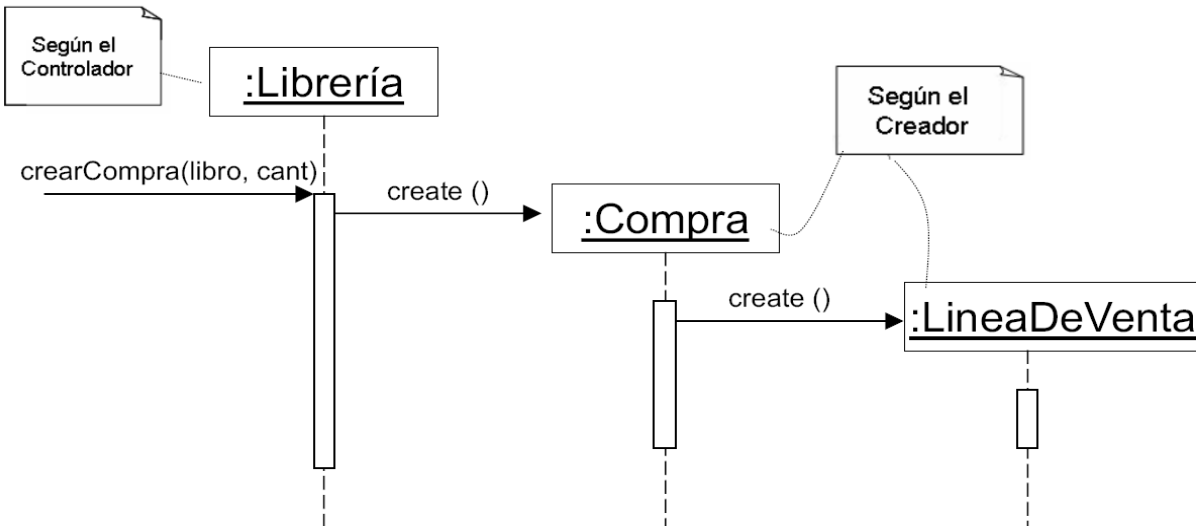
Contrato: crear compra

Operación:	crearCompra ()
Referencias cruzadas:	Caso de Uso: Comprar libro
Precondiciones:	ninguna
Postcondiciones:	- Se creó una instancia de Compra compra (creación de inst.) - compra se asoció con la Librería (formación de asoc.) - Se inicializaron los atributos de compra

... Elección de la clase controlador

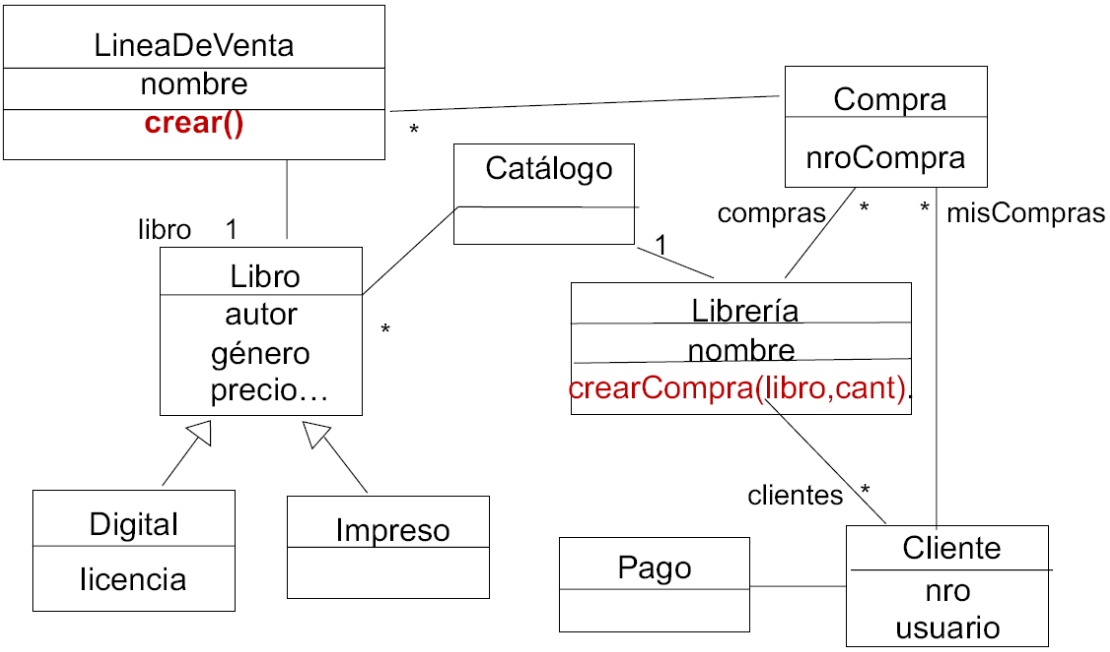


... Aplicando más HAR



- **Diagramas de clases de diseño**
  - *Pasos*
    1. Identificar las clases que participan en los diagramas de interacción y en el Modelo del Dominio o Conceptual.
    2. Graficarlas en un diagrama de clases.
    3. Colocar los atributos presentes en el Modelo Conceptual.
    4. Agregar nombres de métodos analizando los diagramas de interacción.
    5. Agregar tipos y visibilidad de atributos y métodos.
    6. Agregar las asociaciones necesarias.
    7. Agregar roles, navegabilidad, nombre y multiplicidad a las asociaciones.

- Ej (diagrama parcial



Herencia vs. Composición vs. Interfaces

- **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
  - Las clases y objetos que heredan están estrechamente acoplados, ya que cambiar algo en la superclase afecta directamente a las subclases
- **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 objetos creados por composición están débilmente acoplados, por lo que se pueden cambiar más fácilmente componentes sin afectar al objeto contenedor
  - Ejemplo
    - Mal uso de la herencia

```
import java.util.ArrayList;

public class Stack<T> extends ArrayList<T> {

    public void push(T object) {
        this.add(object);
    }
    public T pop() {
        return this.remove(this.size() - 1);
    }
```

- Stack funcionará como una pila, pero su interfaz es voluminosa → está formada por mensajes que hay que anular o redefinir

- La interfaz pública de esta clase no es solo push y pop, también incluye:
  - add en cualquier posición por índice,
  - remove de una posición a otra,
  - Muchos otros mensajes heredados de ArrayList, que son inapropiados para una Pila.
- "Una pila NO ES UN ArrayList"
- Heredar ArrayList viola el encapsulamiento → es una opción de implementación que debe ocultarse

■ Mejor usar composición en este caso

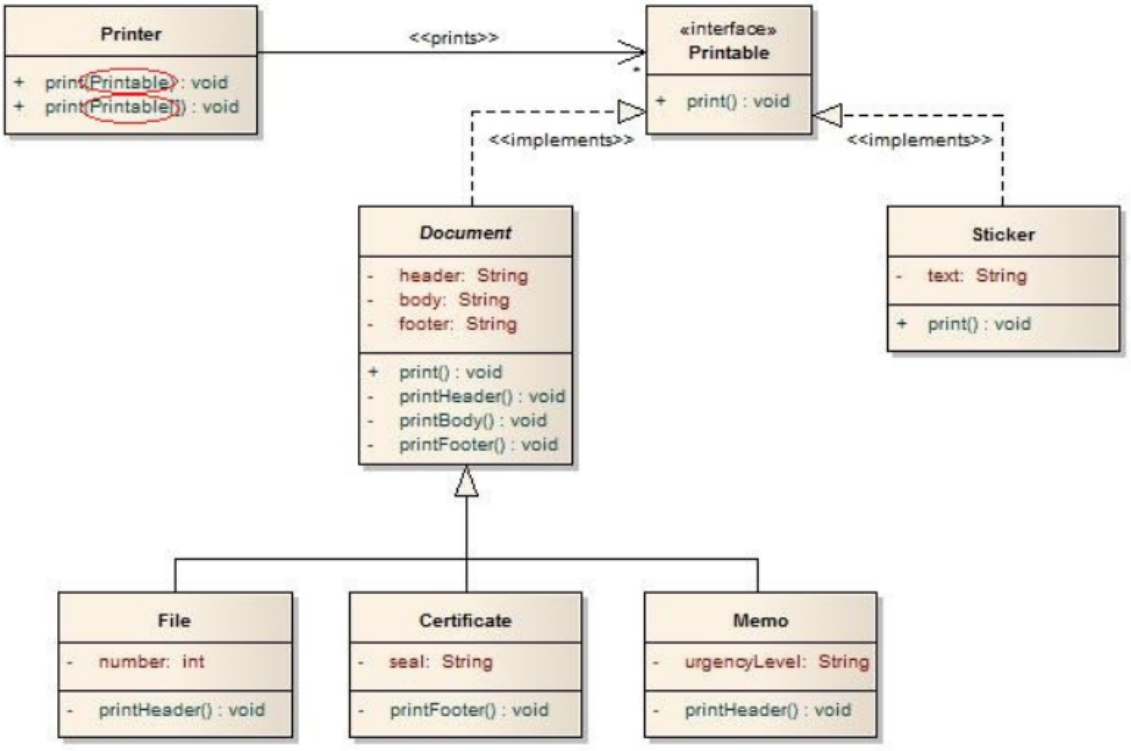
```
import java.util.ArrayList;

public class Stack<T> {
    private ArrayList<T> elementos

    public void push(T object) {
        elementos.add(object);
    }
    public T pop() {
        return elementos.remove(elementos.size() - 1);
    }
```

- Componer con ArrayList para contener la colección de objetos de la pila es una opción de implementación que permite ocultarla públicamente
- En este caso, el uso o composición permite reuso y mantiene el encapsulamiento

- **Interfaces**
  - Las interfaces proveen un mayor nivel de <u>abstracción</u>
    - Si estos puntos de abstracción se detectan oportunamente, podremos mantenernos dentro de un escenario de Extensión no Intrusiva.
  - Son una buena idea para relacionar objetos que hacen lo mismo, pero de forma distinta y con distintos atributos
  - Ejemplo:



Smalltalk

- Lenguaje OO puro → todo es un objeto (hasta las clases)
    - Hay dos tipos de objetos:
        - Los que pueden crear instancias (de si mismos) → **Clases**
        - Los que no pueden crear instancias
    - Si las clases entienden mensajes, tienen su propio conocimiento y comportamiento → ¿Dónde se especifica su estructura y comportamiento? ¿En otra clase?
        - Todo objeto es instancia de una clase → las clases son instancia de una clase también (su **metaclase**)
            - Por cada clase hay una metaclase (se crean juntas).
                - SmallInteger es instancia de “SmallInteger class”
            - Las metaclases son instancias de la clase Metaclass
                - “SmallInteger class” es instancia de Metaclass
- Tipado **dinámicamente**
- Propone estrategia exploratoria 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, perfilador, etc.
    - Es extensible
- Sintaxis minimalista
- Fuente de inspiración de casi todo lo que vino después (en OO)

Javascript (ECMAScript)

- Lenguaje de propósito general
- **Dinámico**
- Basado en objetos → con base en **prototipos** en lugar de clases
    - La forma más simple de crear un objeto es mediante la notación literal (estilo JSON)
    - 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 su comportamiento y estado)
    - Termino armando cadenas de delegación
    - __proto__ → define prototipo del objeto
        - robin.__proto__ = batman
            - El prototipo de Robin será Batman
- 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)