

# Contenido

<b>Metodologías de Desarrollo.....</b>	<b>4</b>
<b>Concepto de Diseño.....</b>	<b>4</b>
Definición.....	4
Según Ralph y Wand.....	4
Componentes del Diseño.....	4
Ejemplo de Relación.....	4
<b>Artesanía vs Ingeniería.....</b>	<b>4</b>
Artesanía: ANTES.....	4
Ingeniería: HOY.....	5
<b>Clasificación de metodología.....</b>	<b>5</b>
Orientado a función/dato.....	5
Orientado a objetos.....	5
<b>El Ciclo de Vida del Desarrollo de Software.....</b>	<b>5</b>
Definición.....	5
Objetivos.....	5
Actividades comunes del ciclo de vida de desarrollo.....	5
Estilos de gestión.....	5
Características deseables de una Metodología.....	6
<b>Ingeniería de Software.....</b>	<b>6</b>
Según la IEEE.....	6
Terminologías Relacionadas.....	6
<b>Proceso de Desarrollo de Software.....</b>	<b>7</b>
Definiciones.....	7
Según IEEE 🍌.....	7
Según Somerville 🍌.....	7
Según Pfleeger 🍌.....	7
Según Pressman 🍌.....	7
<b>Metodología Object-Oriented.....</b>	<b>8</b>
Beneficios de la OO.....	8
¿En qué mejora?.....	8
Tres métodos para comprender el mundo (COAD/YOURDON).....	8
¿Qué es un Sistema OO?.....	8
Definiciones.....	8
Gaona 🍌.....	8
Martin/Odell 🍌.....	8
Booch 🍌.....	8
Concepto de Mensaje.....	8
Concepto de Responsabilidad.....	9
Concepto de Colaboración.....	9
Concepto de Clase.....	9
Principios Fundamentales de la OO.....	9
Abstracción.....	9

Encapsulamiento.....	9
Herencia.....	10
Polimorfismo.....	10
Diseño Orientado a Objetos.....	10
👉 Según Booch.....	10
🤖 Fase Exploratoria Inicial de Objetos.....	10
Identificar Clases.....	10
Identificar Responsabilidades.....	10
Identificar Colaboraciones.....	10
Ejemplo de tarjeta CRC.....	11
Estereotipos básicos de Objetos.....	11
Interfaz.....	11
Entidad.....	11
Control.....	11
UML.....	12
¿Qué es?.....	12
¿Qué significa unificado?.....	12
¿Qué es un modelo?.....	12
Propósito de los modelos.....	12
Modelo Conceptual.....	12
Vistas.....	13
Área de clasificación estructural.....	13
Área de comportamiento dinámico.....	14
Área física.....	14
Área de Gestión del Modelo.....	14
Cuadro de Áreas - Vistas - Diagramas.....	14
Área Estructural.....	15
Vista Estática.....	15
Propósito.....	15
Clasificador.....	15
Clases y Objetos.....	15
Clases: Notación Gráfica.....	16
Relaciones.....	16
Asociación.....	16
Generalización.....	17
Dependencia.....	17
Clases: Niveles de Visibilidad.....	17
Diagrama de Clases y Objetos.....	17
Tipos de Datos.....	18
Clases: Estereotipos.....	18
Asociación: navegabilidad.....	19
Asociación: visibilidad.....	19
Asociación: casos especiales.....	19
Características estáticas o de instancia.....	21
Conjunto de generalización.....	21

Principio de sustitución de Barbara Liskov.....	21
Clases abstractas.....	21
Interfaces.....	22
Interfaz vs Clase Abstracta.....	22
Clase Abstracta.....	22
Interfaz.....	23
Vista de Casos de Uso.....	23
Caso de Uso.....	23
Especificación de Casos de Uso.....	23
Descripción de Casos de Uso.....	23
Actor.....	24
Identificación.....	24
Relaciones.....	24
Diagrama de Casos de Uso.....	25
Vista de diseño.....	25
Estructura Interna.....	25
Clasificador estructurado.....	25
Conector.....	25
Puertos.....	25
Diagrama de Colaboración.....	26
Colaboración.....	26
Diagrama de Componentes.....	26
Componentes.....	26
Área dinámica.....	27
Vista de interacción.....	27
Colaboración.....	27
Diagramas de secuencia.....	27
Operadores de interacción.....	28
Diagrama de comunicación.....	28
Vista de Máquina de Estados.....	28
Diagrama de Estados.....	29
Estado.....	29
Transiciones.....	29
Eventos.....	29
Condición de Guarda.....	30
Acciones (eventos).....	30
Estados Compuestos (subestados).....	30
Estado de Submáquina.....	31
Vista de Actividades.....	32
Área Física.....	33
Vista de Despliegue.....	33
Diagrama de despliegue.....	33
Área Gestión de Modelo.....	34
Vista de Gestión de Modelo.....	34
Dependencias de importación.....	34

<b>SOLID</b> .....	35
<b>SRP - Single Responsibility Principle</b> .....	35
<b>OCP - Open-Closed Principle</b> .....	36
<b>LSP - Liskov Substitution Principle</b> .....	36
<b>ISP - Interface Segregation Principle</b> .....	36
<b>DIP - Dependency Inversion Principle</b> .....	36
<b>GRASP</b> .....	36
<b>General Responsibility Assignment Software Patterns</b> .....	36
🏠 <b>Creador - Creator</b> .....	36
🧐⚠️ <b>Experto en información - Information Expert</b> .....	36
<b>Cohesión y acoplamiento</b> .....	37
<b>Bajo acoplamiento - Low Coupling</b> .....	37
<b>Alta cohesión - High Cohesion</b> .....	37
👮 <b>Controlador - Controller</b> .....	37
♻️ <b>Polimorfismo - Polymorphism</b> .....	38
👩 <b>Fabricación pura - Pure Fabrication</b> .....	38
🔄 <b>Indirección - Indirection</b> .....	38
<b>Variaciones protegidas. - Protected Variations</b> .....	38
<b>Beneficios</b> .....	39
<b>Ejemplo</b> .....	39
<b>Conclusiones</b> .....	39
<b>Diferencias</b> .....	39

# Metodologías de Desarrollo

## Concepto de Diseño

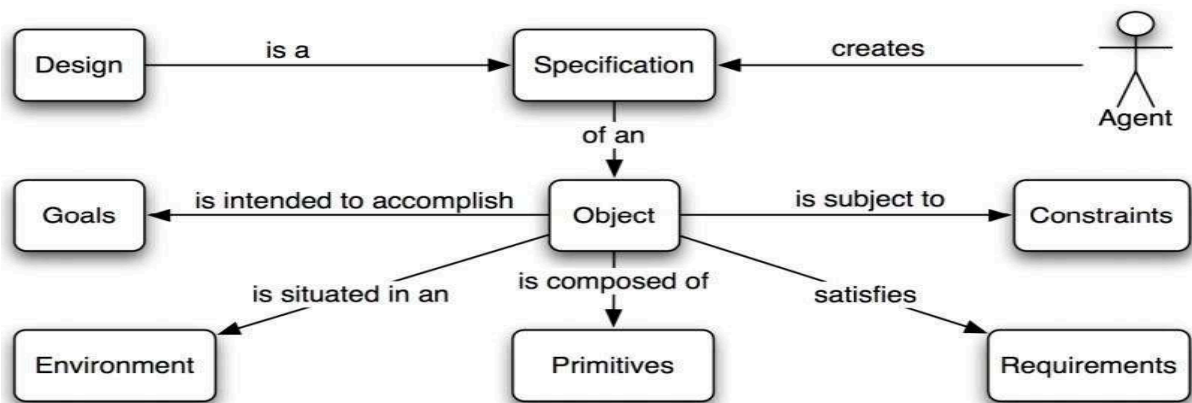
### Definición

#### Según Ralph y Wand

El **diseño** se define como la **especificación** de un **objeto** por parte de un **agente**, destinado a lograr **objetivos** en un **entorno particular**, utilizando un conjunto de **componentes primitivos**, satisfaciendo un conjunto de **requisitos** y sujeto a algunas **restricciones**

- **Análisis:** Responde al qué se va a diseñar, identificando requisitos y restricciones.
- **Diseño:** Responde al cómo se va a implementar, definiendo la estructura y componentes del sistema para alcanzar los objetivos establecidos.

Este enfoque destaca la importancia del diseño en el proceso de desarrollo de software, donde se busca especificar de manera clara y precisa cómo se va a construir un sistema para cumplir con los requisitos definidos previamente durante el análisis.



### Componentes del Diseño

- **Especificaciones:** Describir con detalle lo que se desea desarrollar.
- **Objeto:** El elemento que se va a diseñar.
- **Agente:** Los diseñadores involucrados en el proceso.
- **Objetivos:** Las metas que se pretenden alcanzar.
- **Entorno particular:** El contexto donde se desarrolla el objeto.
- **Componente Primitivo:** Recursos mínimos como controles, ventanas o cajas de texto.
- **Requisitos:** Las acciones que el objeto debe realizar (requisitos funcionales).
- **Restricciones:** Limitaciones que afectan al diseño (requisitos no funcionales).

### Ejemplo de Relación

- **Componentes Primitivas:** Palabras en inglés utilizadas en documentos legales.
- **Restricciones:** Cumplimiento de la constitución y leyes internacionales.
- **Entorno:** Marco legal nacional.
- **Requerimientos:** Definir crímenes y castigos de forma clara y no ambigua.
- **Objetivo:** Establecer un marco legal para abordar los crímenes.
- **Objeto:** Código criminal.
- **Agente:** Expertos legales.
- **Requisitos Funcionales:** Relativos a la función principal del sistema.
- **Requisitos No Funcionales:** Aspectos adicionales que no afectan la función principal.

## Artesanía vs Ingeniería

### Artesanía: ANTES

#### Problemas:

- Subjetividad: Basada en la experiencia del desarrollador
- Resultados no reproducibles

- Resultados finales impredecibles, sin punto de comparación
- Falta de control sobre el proyecto, carecía de estructura de equipos

Ingeniería: HOY

- Objetivos, Restricciones, Decisiones y Metodologías.
- Sistematización de Decisiones.
- Resultados reproducibles basados en plantillas predefinidas.

Clasificación de metodología

Orientado a función/dato	Orientado a objetos
<ul style="list-style-type: none"><li>• Énfasis en la transformación de datos</li><li>• Funciones y datos tratados como entidades separadas</li><li>• Difícil de entender y modificar</li><li>• Funciones, usualmente, dependientes de la estructura de los datos</li></ul>	<ul style="list-style-type: none"><li>• Énfasis en la abstracción de datos</li><li>• Funciones y datos encapsulados en entidades fuertemente relacionadas</li><li>• Facilidad de mantenimiento</li><li>• Mapeo directo a entidades del mundo real</li></ul>

Orientado a función/dato

dsfsdf

Orientado a objetos

Dfsdfds

El Ciclo de Vida del Desarrollo de Software

Definición

Conjunto de actividades que se llevan a cabo durante un proyecto de software.  
Se suele utilizar como sinónimo al término metodología, pero este último es más abarcativo ya que el ciclo de vida indica qué es lo que hay que obtener a lo largo del desarrollo del proyecto, pero no cómo. Esto lo debe indicar la metodología.

- **Ciclo de vida:** organiza, especifica qué hay que hacer pero no como
- **Metodología:** define como implemento o desarrollo esa organización

Objetivos

- 1) Definir las actividades a llevarse a cabo durante un proyecto de sistemas.
- 2) Lograr congruencia entre múltiples proyectos de SW.
- 3) Proporcionar puntos de control administrativo de las decisiones de continuar o no con el proyecto.

Actividades comunes del ciclo de vida de desarrollo

Independientemente del enfoque, el ciclo de desarrollo de software abarca las siguientes actividades:

- **Especificación de requerimientos:** Se realizan entrevistas con el usuario identificando los requerimientos y necesidades del usuario.
- **Análisis:** Modela los requerimientos del usuario.
- **Diseño:** Se modela la solución del sistema, teniendo en cuenta el ambiente de implementación a utilizar, por ejemplo, si el sistema es centralizado o distribuido, la base de datos a utilizar, lenguaje de programación, performance deseada, etc.
- **Implementación:** Dado el lenguaje de programación elegido se implementa el sistema.
- **Testeo:** En esta etapa se verifica y valida el sistema teniendo en cuenta algunos criterios determinados por el grupo correspondiente.
- **Mantenimiento:** Es la etapa más difícil de desarrollo del sistema, actualiza y modifica el sistema si surgen nuevos requerimientos.

Estilos de gestión

No es aconsejable elegir un modelo y seguirlo al detalle sino que se debe adaptar a las características del proyecto que está siendo desarrollado.  
{completar}

Modelo	Características	Ventajas	Desventajas
<b>Cascada</b>	- Actividades secuenciales claras. - Etapas bien definidas: requisitos, diseño, implementación, pruebas, mantenimiento.	- Claridad y simplicidad en la gestión del proyecto.	- Dificultad para gestionar cambios y retrocesos. - Rigidez en la adaptación a requisitos cambiantes.
<b>Prototipos</b>	- Iterativo, con prototipos presentados al cliente. - Combina bien con otros modelos de desarrollo.	- Reducción del rechazo del producto por parte del cliente. - Mejora en la comunicación de requisitos.	- Posible pérdida de trabajo en prototipos desechables. - Dificultad en la planificación precisa del proyecto.
<b>Espiral</b>	- Dirigido por riesgos. - Iterativo, con prototipos en cada vuelta del espiral.	- Reducción de riesgos en el desarrollo. - Flexibilidad y adaptabilidad.	- Complejidad en la gestión del proyecto. - Requiere experiencia en la identificación y gestión de riesgos.
<b>Iterativo e Incremental</b>	- Incremental: Construcción de subsistemas que se integran en el sistema completo. - Iterativo: Mejora y refinamiento continuo de la funcionalidad. - Iterativo e Incremental: Combina ambos enfoques, permitiendo la entrega de versiones con nuevas funcionalidades y mejoras.	- Adaptabilidad a cambios. - Entregas frecuentes y manejables.	- Requiere una gestión eficiente para coordinar iteraciones e incrementos. - Puede complicar la planificación y estimación de recursos.

⚠ 3 min read 0 entrante

## Características deseables de una Metodología

- Existencia de reglas predefinidas
- Cobertura total del ciclo de desarrollo
- Verificaciones intermedias
- Planificación y control
- Comunicación efectiva
- Utilización sobre un abanico amplio de proyectos
- Fácil formación
- Herramientas **CASE**<sup>1</sup>
- Actividades que mejoren el proceso de desarrollo
- Soporte al mantenimiento
- Soporte de la reutilización de software

## Ingeniería de Software

### Según la IEEE

- Enfoque sistemático y cuantificable para el desarrollo de software.
- Disciplina relacionada al desarrollo, operación y mantenimiento del sistema.
- Resultados medibles y cuantificables.

### Terminologías Relacionadas

- **Método:**
  - Según Pfleeger, procedimiento formal para producir un resultado.
  - Ejemplo: el método de un chef al preparar una salsa.
- **Metodología:**

<sup>1</sup> Las herramientas CASE (Computer Aided Software Engineering, Ingeniería de Software Asistida por Computadora) son diversas aplicaciones informáticas o programas informáticos destinadas a aumentar el balance en el desarrollo de software reduciendo el costo de las mismas en términos de tiempo y de dinero.

- o Definida por Pfleeger como un enfoque sistemático, disciplinado y cuantificable para el desarrollo de software, similar al concepto de ingeniería de software.
- o Según Pressman, un marco de proceso de desarrollo de software, incluyendo etapas como planificación, análisis de requisitos, diseño, implementación, pruebas y mantenimiento.
- **Proceso:**
  - o Combinación de métodos y técnicas.
  - o Según Pfleeger, es un marco para actividades necesarias en la construcción de software, incluyendo tecnologías, métodos técnicos y herramientas automatizadas.
  - o Según Pressman, es un conjunto de actividades que llevan a la producción de un producto de software, flexible y adaptable a las necesidades del proyecto y del cliente.

En resumen, la ingeniería de software representa un enfoque más estructurado y sistemático en comparación con la artesanía, que se basa en la experiencia individual del desarrollador. Este cambio hacia la ingeniería ha llevado a la adopción de metodologías y procesos formales para mejorar la reproducibilidad y la calidad de los resultados en el desarrollo de software.

Nuevamente Pressman hace referencias a las etapas, pero la idea es la misma

## Proceso de Desarrollo de Software

### Definiciones

#### Según IEEE 🍌

- Traducción de las necesidades del usuario en requisitos de software
- Transformación de los requisitos de software en diseño.
- Implementación del diseño en código.
- Prueba del código y, ocasionalmente, instalación y verificación del software para uso operativo.

#### Según Somerville 🍌

- Secuencia de actividades que lleva a la producción de un producto de software. Estas pueden tener productos intermedios
- Cuatro actividades comunes en cualquier definición:
  1. **Especificación:** Definir detalladamente lo que el sistema debe hacer.
  2. **Desarrollo:** Convertir las especificaciones en un sistema operativo.
  3. **Validación:** Asegurar que el sistema cumple con los requisitos y expectativas del cliente.
  4. **Evolución del Software:** Adaptar y mejorar el software para responder a cambios en los requisitos y el entorno operativo

🍌 También se suelen indicar: productos, roles y pre y postcondiciones, que preparan y verifican cada etapa

#### Según Pfleeger 🍌

- **Proceso:** Serie de pasos que implican actividades, restricciones y recursos que producen un resultado previsto.
- **Proceso de Desarrollo de SW:** También conocido como ciclo de vida del software, describe la vida de un producto de software desde su concepción hasta su implementación, entrega, uso y mantenimiento. Impone consistencia y estructura mediante una colección de procedimientos y una unidad anatómica del proceso. Permite capturar y transferir experiencias y generalmente involucra etapas como análisis y definición de requisitos, diseño de sistemas, diseño de programas, escritura de programas, revisión de unidad, pruebas de integración, prueba del sistema, entrega del sistema y mantenimiento.
  - o **Análisis y definición de requisitos:** Comprender y documentar lo que el usuario necesita
  - o **Diseño de sistemas y programas:** Planificar cómo cumplir esos requisitos.
  - o **Escritura de programas:** Implementar el diseño en código.
  - o **Revisión de unidad y pruebas de integración:** Verificar que las partes individuales y combinadas del software funcionen correctamente.
  - o **Prueba del sistema:** Asegurar que el sistema completo funcione como se espera.
  - o **Entrega y mantenimiento del sistema:** Implementar el software y mantenerlo operativo y actualizado.

#### Según Pressman 🍌

- **Proceso de desarrollo Software:** Marco para las actividades, acciones y tareas requeridas para construir software de alta calidad. Se compone de actividades, acciones y tareas:
  - o **Actividad:** Objetivo amplio.
  - o **Acción:** Conjunto de tareas que producen un producto de trabajo principal.
  - o **Tarea:** Enfoque en un objetivo pequeño pero bien definido.
- Cinco actividades comunes: **comunicación, planificación, modelado, construcción y despliegue**
  - o **Comunicación:** Interactuar con las partes interesadas para entender y definir los requisitos.
  - o **Planificación:** Establecer un plan detallado para el desarrollo del proyecto.
  - o **Modelado:** Crear representaciones abstractas del sistema para entender y comunicar su estructura y comportamiento.



- **Construcción:** Implementar y probar el software.
- **Despliegue:** Entregar el software al cliente y asegurar su funcionamiento en el entorno operativo.

27/03

# Metodología Object-Oriented

## Beneficios de la OO

### ¿En qué mejora?

- Reusabilidad del SW
- Mejora en la Mantenibilidad y Modificabilidad del SW
- Disminución del GAP Semántico. Representación consistente en todo el Ciclo de Vida
- Mejor interacción Usuario - Analista - Diseñador
- Más apropiado para resolver problemas complejos (GUIs, patterns, etc.)

### Tres métodos para comprender el mundo (COAD/YOURDON)

- Diferenciación de la experiencia en Objetos y Atributos
- Distinción entre el Todo y las partes de este
- Formación y distinción de Clases de Objetos

## ¿Qué es un Sistema OO?

### Definiciones

#### Gaona

Un sistema orientado a objetos es un sistema donde sus elementos son **entidades** a las que denominaremos **objetos** que interactúan entre sí por medio del envío y recepción de **mensajes**, todo esto con el fin de cumplir un **objetivo** claro y definido para el sistema. Cada uno de estos objetos **colabora** con otros objetos del sistema interactuando con los mismos por medio de estos mensajes, y cada objeto realiza tareas según un **rol** específico en el sistema acorde a la **responsabilidad** que tenga asignada

#### Martin/Odell

Un objeto es algo real o abstracto acerca del cual almacenamos datos y operaciones que manipulan dichos datos

#### Booch

Un objeto es una entidad que tiene **estado**, **comportamiento** e **identidad**. La estructura y el comportamiento de objetos similares se definen en su clase común. Los términos “instancia” y “objeto” son intercambiables

- Los objetos poseen métodos llevados a cabo por funciones o procedimientos
- Un objeto tiene estado, comportamiento e identidad.
  - El estado son los valores de los atributos que tiene un objeto en un determinado momento. Es el estado de los enlaces. Un estado depende de los atributos y sus enlaces
  - Un objeto no necesita de un atributo para identificarse, no se definen por un id abstracto. Se definen por su estado, comportamiento e identidad
  - Identidad refiere a la **clave primaria**

**¿Qué puede ser un objeto?** Un objeto puede representar algo físico, algo abstracto, algo del dominio del problema o algo del dominio de la solución

## Concepto de Mensaje

- Los mensajes son el mecanismo por el cual se comunican los objetos para poder interactuar y colaborar unos con otros.
- Siempre el **receptor** es la clase que contiene el método de ese mensaje.
- Se pueden enviar mensajes a un objeto con diferentes propósitos:
  - para indagar el valor de alguno de sus atributos (getter)
  - para asignar valor a alguno de sus atributos (setter)
  - para solicitar la realización de una operación (custom)
- existen tres tipos de mensajes:
  - **síncronicos:** cuando existe un tiempo de espera
  - **asíncronicos:** cuando no existe una espera de procesamiento

- **de tiempo libre:** es una mezcla de los anteriores, es sincrónico hasta que llega a un timer, luego pasa a ser asincrónico

## Concepto de Responsabilidad

- UML define una responsabilidad como “un contrato u obligación de un clasificador”
- Son cosas que un objeto **hace** o **conoce** (Larman)
- **Hacer**
  - hacer algo en sí mismo, como crear un objeto o hacer un cálculo
  - iniciar una acción en otros objetos
  - controlar y coordinar actividades en otros objetos
- **Conocer**
  - conocer datos privados encapsulados
  - saber sobre objetos relacionados
  - saber sobre cosas que puede derivar o calcular
- Las responsabilidades van primero, primero pienso en estas y luego en los atributos, métodos y relaciones
- **Rol** es el conjunto **relacionado** de responsabilidades que puede usarse de manera intercambiable. Sinónimo de propósito
- **Estereotipo de roles:** al simplificar y caracterizar en exceso podemos reflexionar sobre la naturaleza de un objeto más fácilmente. Siempre una clase cae en alguno de estos. Caracterización de las funciones que necesita una aplicación
  - **Information holder:** conoce y proporciona información
  - **Structurer:** mantiene las relaciones entre los objetos y la información
  - **Service provider:** realiza trabajos y ofrece servicios informáticos
  - **Coordinator:** reacciona a los eventos delegando tareas a otros
  - **Controller:** toma decisiones y dirige estrechamente las acciones de los demás
  - **Interfacer:** transforma la información y solicitudes entre distintas partes de nuestro sistema
- El rol es como el **propósito** de la clase, muy idéntico a su **intención**. Por lo que, primero se define el propósito de una clase, luego se deriva las responsabilidades y luego recién los atributos y métodos

## Concepto de Colaboración

- Una colaboración se produce cuando dos o más objetos interactúan entre sí para realizar una tarea. ya que un solo objeto no puede realizar todas las tareas del sistema, estas deben distribuirse entre los distintos objetos que componen el sistema
- Esta interacción se lleva a cabo a través del envío de mensajes
- Cuando un objeto no “sabe” cómo realizar una tarea por estar fuera del alcance de su responsabilidad “delega” en otro objeto la realización de la misma, solicitando esto por medio del envío de un mensaje

## Concepto de Clase

- Conceptualmente una clase es un **grupo de objetos con propiedades (atributos) similares, comportamiento común (operaciones), relaciones comunes entre objetos**, y semántica común. En este sentido, una clase es un poderoso mecanismo conceptual para agrupar objetos similares en conceptos finitos.
- Una clase es una matriz, un molde que se utiliza para “crear” objetos del mismo tipo. Esto es fundamentalmente válido en el caso de los lenguajes orientados a objetos basados en clases, como Smalltalk y Java.

## Principios Fundamentales de la OO

Para que un Ambiente o Lenguaje pueda soportar el paradigma orientado a objetos debe cumplir con estos cuatro principios fundamentales

### Abstracción

- Se utiliza para tratar la complejidad, detona características esenciales.
- Se enfoca en la vista externa de un objeto y así, sirve para separar el **comportamiento esencial** de un objeto **de su implementación**
- Los objetos asociados a conceptos útiles del dominio del problema o de la solución son las mejores abstracciones
- un objeto formado por un conjunto de métodos no relacionados es un ejemplo de mala abstracción

### Encapsulamiento

- mecanismo que permite ocultar los detalles de implementación de un objeto.
- se complementa con la abstracción
  - **Abstracción** → comportamiento observable
  - **Encapsulamiento** → implementación de ese comportamiento
- empaqueta, por lo que permite un control en la complejidad y un mejor mantenimiento

## Herencia

- Se enfoca de relacionar al dominio del problema o al dominio de la solución
- Por más de que haya herencia, si no cambio la visibilidad, no puedo acceder de la clase hija al atributo oculto de la clase padre

## Polimorfismo

- es el principio por el cual una misma operación es resuelta de diferente forma según el objeto que recibe el mensaje
- Sobrecarga: es definir más de un método por cada mensaje, los tipos de argumentos ayudan a decidir qué mensaje invoca
- la definición del método reside en la clase base
- la implementación del método reside en la clase derivada
- la invocación es resuelta al momento de ejecución
  - Early binding
  - Late binding

## Diseño Orientado a Objetos

### 🔔 Según Booch

- “El diseño orientado a objetos es un método de diseño que abarca el proceso de descomposición orientada a objetos y una notación para representar modelos tanto lógicos y físicos como estáticos y dinámicos del sistema bajo diseño”
- La descomposición OO es la diferencia central con el diseño estructurado

### 🧐 Fase Exploratoria Inicial de Objetos

#### Identificar Clases

- Algunos de los métodos más habituales para descubrir clases de objetos se basan en:
  - El conocimiento general del dominio del problema que el desarrollador posea
  - sistemas anteriores similares de los que haya participado el desarrollador
  - modelos de negocios analizables y que se relacionen con el dominio en estudio
  - sesiones de tarjetas CRC
  - glosario de términos
  - análisis gramatical de enunciados del dominio del problema
- Se identifica básicamente la clase sin enfocarse en los métodos. luego se elige el nombre cuidadosamente y se describe su propósito.
- es importante definir el propósito de la clase porque me impone un límite, y esto es conveniente para asignar sus responsabilidades
- no siempre es conocido a lo que se dedica una clase

#### Identificar Responsabilidades

- Una vez identificadas las clases se deben determinar sus responsabilidades
- Para encontrar responsabilidades
  - generalmente son verbos y pueden proceder de diferentes fuentes
  - hay responsabilidades declaradas o implícitas en los casos de uso
  - se identifican responsabilidades para cubrir los huecos en los casos de uso y otras descripciones del sistema con responsabilidades adicionales de nivel inferior
  - se identifican responsabilidades para apoyar las relaciones y dependencias entre los candidatos
- Las responsabilidades identificadas **deben asignarse a la clase a la cual pertenecen para:**
  - **mantener el comportamiento** relacionado a cierta información en la misma clase. Si un objeto mantiene determinada información, las operaciones que se realizan sobre dicha información deben agregarse a la misma clase
  - mantener la información acerca de una misma cosa en una misma clase o clases muy relacionadas
  - distribuir responsabilidades compuestas entre clases relacionadas
- Los patrones **GRASP** sirven para la asignación de responsabilidad, que yo tenga una responsabilidad no significa que esa clase tenga que cumplir por sí sola con esa responsabilidad ya que un programa OO es una red colaborativa de objetos. Por lo tanto, para responsabilidades que son un poco más complejas se deben relacionar varias clases.

#### Identificar Colaboraciones

- Las clases pueden cumplir sus responsabilidades realizando las operaciones por sí solas o colaborando con otras clases
- Para encontrar colaboraciones es conveniente realizar preguntas por clase, como pueden ser ¿esta clase es capaz de cumplir sus responsabilidades por sí misma? si no ¿de qué otra clase necesita colaboración?
- Las clases que no participan en ninguna colaboración son sospechosas y normalmente se descartan

- idealmente cada responsabilidad debe tener asociada sus colaboraciones
- se deben ver esas responsabilidades de una forma específica. No son solo “métodos”

### Ejemplo de tarjeta CRC

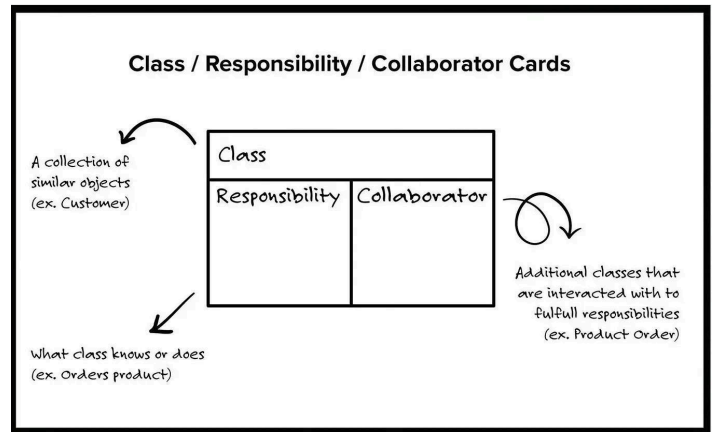
El documento representa un archivo, guarda texto, gráficos, tabla o texto enriquecido. -Con esta definición uno reconoce las posibles responsabilidades.

La idea es que la clase actúa de una forma distinta que un rol. Ambos con el mismo nombre

**Primera columna:** responsabilidades

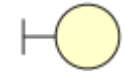
**Segunda columna:** colaboraciones

**Nombre de tarjeta:** clase



## Estereotipos básicos de Objetos

### Interfaz



Estas clases representan las interfaces del sistema con los usuarios y con otros sistemas con los que interactúa. Deberían contener la menor cantidad de Lógica del Negocio y limitarse a la lógica correspondiente al diálogo con los actores. Estas clases de objetos se corresponden con la Vista y el Controlador del patrón MVC.

En los puntos donde los actores inician o se comunican con los casos de uso, existe a menudo intercambio de datos. El actor suministra datos al sistema, o el sistema brinda datos al actor.

Es importante identificar qué datos son pasados. Estos datos pueden ser descritos utilizando clases de objetos interfase (que se corresponde con pantallas, ventanas, reportes, etc.). Es necesario identificar clases de objetos interfaces cuyos datos son suministrados a los actores o recibidos desde ellos. Como heurística general se creará:

Durante el análisis, se creará una clase interfaz por cada vínculo caso de uso- actor dispositivo. Esta clase representa la interfaz principal del vínculo del actor con el dispositivo. Por ejemplo, un supervisor de almacén puede utilizar una interfaz GUI para controlar movimientos de stock y una impresora para imprimir detalles de movimientos. Para estas actividades se pueden identificar dos objetos interfaz.

La identificación y diseño de los componentes de esta interfaz, que también son objetos interfaz (ej. botones, cajas de texto, etc.), se difiere hasta el diseño detallado de la aplicación.

También, siguiendo el patrón MVC, se puede asignar una clase interfaz (view) para cada entidad (modelo) que participa.

Los objetos interfaz pueden modelar protocolos de comunicación por capas como el modelo ISO. Un objeto interfaz se define por cada capa, el cual se comunica solo con objetos de la capa superior e inferior inmediata.

Se sugiere la siguiente nomenclatura para denominar estas clases: **UINombreClase**

### Entidad



Estas clases se corresponden con el Modelo del patrón MVC. Estos objetos representan generalmente la “memoria”. Encapsulan datos sobre entidades y conceptos del negocio que deben recordarse, y las reglas del negocio que se aplican a los mismos (ej. Alumno, Cuenta, Libro, etc.).

En virtud de esto, estas clases requieren de mecanismos de persistencia para almacenar los datos en algún sistema de almacenamiento, típicamente mecanismos de mapeo objeto-relacional para almacenar en bases de datos relacionales. En la realización de los prácticos, se asumirá una simplificación, para lo cual el alumno se limitará a crear estas clases y simular con constantes los datos que requiriera para demostrar el programa funcionando.

La nomenclatura utilizada para denominar estas clases será: **ENNombreClase**

### Control



Estas clases tienen la función primaria de coordinación del flujo de eventos entre objetos. No debe confundirse estas clases con la clase Controlador del patrón MVC.

A estas clases también se asignan responsabilidades que no son inherentes a alguna clase entidad o interfaz en particular. Los objetos de control se vinculan con los actores a través de objetos interfaz. Los objetos de control a menudo actúan como un conjunto de buffers entre los objetos entidad y los de control.

Debe tenerse la precaución de no utilizar objetos de control para separar funciones de datos, ya que esto va en contra del principio de encapsulamiento de la orientación a objetos, y se corre el riesgo de caer en un enfoque procedural clásico. Como lineamiento general, cuando se diseñe una realización de caso de uso, se asigna una clase de

control como coordinadora primaria del flujo de eventos de dicha realización. Esto no es estrictamente necesario, pero es una práctica recomendable.

La nomenclatura utilizada para identificar estas clases será: **CTNombreClase**.

10/04

# UML

## ¿Qué es?

- Aprender UML no es aprender el paradigma de objetos
- No es una metodología de desarrollo y es solo para modelos de objetos
- Es un lenguaje “unificado” de modelado para visualizar, especificar, construir y documentar los artefactos de un sistema intensivo en software
- **visualizar** ayuda a plasmar las ideas de alguna forma. Es ser capaz de plasmar en algún modelo la idea que yo quiero llevar a cabo
- **especificar** significa exactamente “que” elementos tendrá mi modelo, con que detalle voy a especificar
- **construir**, puede ser con ingeniería directa o inversa, esto es, a partir del modelo genero el código o a partir del código genero el modelo (Rational Software Architect - RSA). Los modelos nos proporcionan plantillas que nos guían en la construcción de un sistema
- **documentar** es pasar a UML y plasmarlo en un documento, decisiones de diseño que fui tomando y distintos diagramas, tiene el modelo.
- UML proporciona una forma estándar de representar los planos de un sistema, y comprende tanto elementos conceptuales, como los procesos del negocio y las funciones del sistema.

## ¿Qué significa unificado?

- Lenguaje = sintaxis + semántica
- Unificado a través de:
  - métodos y notaciones históricas
  - etapas de ciclo de desarrollo
  - dominios de aplicación
  - lenguajes y plataformas de implementación
  - procesos de desarrollo

## ¿Qué es un modelo?

- Es una representación de la realidad
- es una **representación** en algún medio que captura los aspectos importantes del sistema desde un determinado punto de vista
- un modelo de un sistema de software es realizado en un **lenguaje de modelado**
- no es lo mismo un modelo que un diagrama
- El diagrama es **una forma** de **representar** un modelo y el modelo es el conjunto de **todas** las ideas **representadas** en diagramas y otras cosas

## Propósito de los modelos

- Construimos modelos para comprender mejor el sistema que estamos desarrollando ya que no podemos comprender el sistema en su totalidad.
- **capturar y precisar requerimientos** de un dominio de conocimiento, que sea comprensible por todos los **stakeholders** del proyecto.
- “capturar y enumerar exhaustivamente los requisitos y el dominio de conocimiento, de forma que todos los implicados puedan entenderlos y estar de acuerdo con ellos”
- pensar sobre un diseño de un sistema (explorar alternativas)
- **capturar decisiones** de diseño de un sistema (separado de los requerimientos)
- **generar productos** de trabajo usables
- **documentar**

## Modelo Conceptual

El propio UML tiene modelos de cómo debe ser un modelo y eso se llama “meta-modelo”. Estos poseen elementos y los tres tipos principales (de elementos) son

- **bloques básicos**
  - **elementos (cosas)**: abstracciones que constituyen los ciudadanos de primera clase en un modelo

- **estructurales (clasificadores)**
    - **Classes:** Represent real-world entities with attributes, methods, and relationships.
    - **Interfaces:** Define a set of behaviors that classes can implement.
    - **datatype**
    - **casos de uso**
    - ?? clase activa
    - **Active Class:** An active class is a class that represents an entity that can initiate actions or respond to events. Active classes are typically represented by classes with a "do" or "handle" method.
    - **actor**
    - **Components:** Modular units of code that can be assembled to form larger systems.
    - **Artifacts:** Physical or conceptual elements that are used by the system, such as documents, files, or libraries.
    - **Nodes:** Computational units that execute behavior and interact with each other
  - **comportamiento**
    - **Interactions:** Collaborations between elements to achieve a goal.
    - **States:** Represent different conditions of an element.
    - **Activities:** Units of behavior that represent a series of steps.
  - **agrupamiento: paquetes**
    - **Packages:** Logical groupings of elements to organize and manage complexity
  - **anotaciones: notitas**
    - **Notes:** Textual descriptions that provide additional information about elements.
  - **relaciones**
    - **Dependency:** A unidirectional relationship where one element depends on another.
    - **Association:** A bidirectional relationship between elements that have knowledge of each other.
    - **Generalization:** A relationship where a subclass inherits attributes and methods from a superclass.
    - **Realization:** A relationship where a class implements the behavior of an interface.
  - **diagramas**
    - **Diagramas de clases (Class Diagrams):** Visualize classes, their attributes, methods, and relationships.
    - **Diagramas de casos de uso (Use Case Diagrams):** Model the interactions between actors and use cases.
    - **Diagrama de secuencia (Sequence Diagrams):** Illustrate the sequence of interactions between objects in a scenario.
  - **reglas:**
    - **Well-formedness Rules:** Define the constraints for creating valid UML models.
    - **Non-well-formedness Rules:** Identify invalid patterns in UML models.
  - **mecanismos**
    - **Adornos:** Mechanisms for customizing the UML metamodel to meet specific needs.
    - **Divisiones:** Logical groupings of elements within a package
    - **Extensiones:** Additional information attached to elements, such as stereotypes or visibility modifiers.
- Debe haber un conjunto de reglas para crear un conjunto de modelos bien formados y completos

## Vistas

- Una vista es un subconjunto de construcciones de modelado que se enfocan en un aspecto particular del sistema
- Las vistas se organizan en cuatro áreas:
  - **Estructural**
  - **De comportamiento dinámico**
  - **De distribución física**
  - **De gestión del modelo**

### Área de clasificación estructural

- Describe los elementos del sistema (clasificadores) y sus relaciones
- Los clasificadores más comunes son
  - Clases
  - Casos de uso
  - Componentes
  - Nodos
- Las vistas pueden ser
  - Estática → diagrama de clases
  - De Casos de uso → diagrama de casos de uso
  - De Diseño → diagrama de componentes

- Esta área se dedica a englobar cual es la estructura del sistema que yo quiero construir
- El concepto que yo puedo encontrar en esta área es el clasificador

### Área de comportamiento dinámico

- Describe el comportamiento del sistema a través del tiempo
- Sus vistas pueden ser
  - o De interacción
    - Diagrama de comunicación, antes llamado diagrama de colaboración
    - Diagrama de secuencia
  - o De máquina de estados
    - Diagrama de estados
  - o De actividades
    - Diagrama de actividades

### Área física

- Describe los recursos computacionales en el sistema y los artefactos desplegados en el
- Los clasificadores más comunes son los nodos y los artefactos
- Posee una vista denominada vista de despliegue □ diagrama de despliegue

### Área de Gestión del Modelo

- **Vista de Gestión del Modelo**
  - o Describe la organización de los modelos en unidades jerárquicas
  - o Diagrama de paquetes
  - o Permite organizar el sistema en paquetes, subsistemas y modelos
- **Perfil**
  - o Permiten adaptar los elementos de modelado asignándole una semántica particular
    - Estereotipos
    - Valores etiquetados
    - Restricciones (OCL)

## Cuadro de Áreas - Vistas - Diagramas

Área principal	Vista	Diagrama	Conceptos principales
estructural	<a href="#">Vista estatica</a>	Diagrama de clases	Asociacion, clase, dependencia, generalización, interfaz, realización
	<a href="#">Vista de diseño</a>	Estructura interna	Conector, interfaz, interfaz obligatoria, interfaz proporcionada, parte, puerto
		Diagrama de colaboración	Colaboración, conector, rol, uso de la colaboración
		Diagrama de componentes	Proporcionalidad, interfaz obligatoria, puerto, realización, subsistema
	<a href="#">Vista de casos de uso</a>	Diagrama de casos de uso	Actor, asociacion, caso de uso, extensión, generalización, de casos de uso inclusión
Dinámica	<a href="#">Vista de máquina de estado</a>	Diagrama de máquina de estados	Actividad do, disparador, efecto, estado, evento, región, transición, transición de finalización
	<a href="#">Vista de actividad</a>	Diagrama de actividades	Acción, actividad, control de flujo, división, excepción, flujo de datos, nodo de control, nodo objeto, pin, región de expansión, unión
	<a href="#">Vista de interacción</a>	Diagrama de secuencia	Especificación de la ejecución, especificación del suceso, fragmento de la interacción, línea de vida, mensaje, operando de la interacción, señal
		Diagrama de comunicación	Colaboración, condición de guarda, mensaje, rol, número de secuencia
<a href="#">Física</a>	Vista de despliegue	Diagrama de despliegue	Artefacto, dependencia, manifestación, nodo
<a href="#">Gestión del modelo</a>	Vista de gestión del proyecto	Diagrama de paquetes	Importar, modelo, paquete
	perfiles	Diagrama de paquetes	Estereotipo, perfil, restricción, valor etiquetado



# Área Estructural

## Vista Estática

- Desde esta vista parten las demás vistas
- Es un modelo incremental

### Propósito








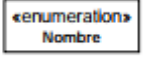
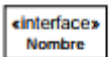

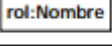
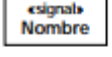

- Es la base sobre la que se construyen las otras vistas
- Captura la estructura de los objetos (datos + comportamiento)
- Es un modelo incremental

### Clasificador

- Es un concepto discreto que describe cosas que tienen identidad, estado, comportamiento y relaciones, y que puede o no poseer una estructura interna
- Tipos:
  - **Elementos del Sistema**
    - Clase
    - Interfaz
    - Tipos de datos
  - **Conceptos de comportamiento**
    - Casos de uso
    - Clasificador estructurado
  - **Cosas del entorno**
    - Actor
  - **Estructuras de implementación**
    - Componente
    - Nodo
    - Subsistema
- El concepto de clasificador representa **todas** las cosas que existen en UML

## Clases y Objetos

- **Clase:**
  - Representa un concepto discreto dentro de una aplicación modelado, que representa cosas de una determinada especie
  - Es un descriptor de un conjunto de objetos con estructura, comportamiento, relaciones, y semántica común
  - La clase tiene visibilidad con respecto a su contenedor. La visibilidad específica como puede ser utilizada por otras clases externas al contenedor.
  - Una clase tiene multiplicidad que especifica cuántas instancias de ella pueden existir
  - Pueden ser
    - Algo físico
    - Algo del negocio
    - Un concepto lógico
    - Algo de la aplicación
    - Algo del comportamiento
- **Objeto:**
  - Es una entidad discreta con identidad, estado y comportamiento invocable
  - Es una instancia de una clase
  - Objeto = estructura + operaciones + estado interno + identidad
  - El estado es descrito por atributos y asociaciones.

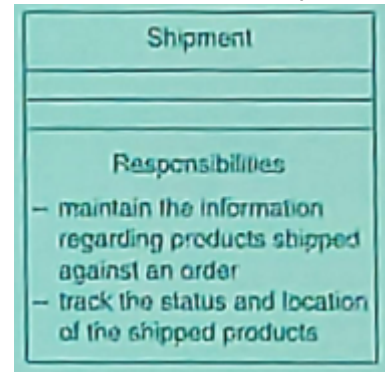
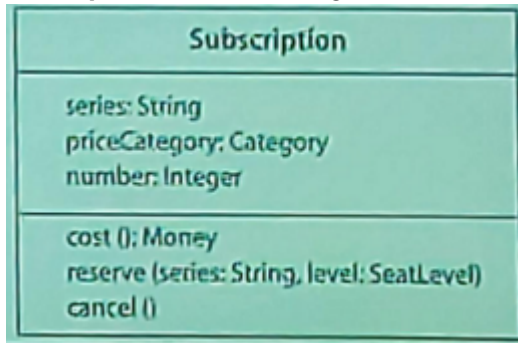
Clasificador	Función	Notación
actor	Un usuario externo al sistema	
artefacto	Una pieza física del sistema de información	
caso de uso	Una especificación del comportamiento de una entidad en su interacción con los agentes externos	
clase	Un concepto del sistema modelado	
clasificador estructurado	Un clasificador con estructura interna	
colaboración	Una relación contextual entre objetos que desempeñan roles	
componente	Una parte modular de un sistema con interfaces bien definidas	
enumeración	Un tipo de datos con valores literales predefinidos	
interfaz	Un conjunto con nombre de operaciones que caracterizan un comportamiento	
nodo	Un recurso computacional	
rol	Una parte interna en el contexto de una colaboración o clasificador estructurado	
señal	Una comunicación asíncrona entre objetos	
tipo primitivo	Un descriptor de un conjunto de valores primitivos que carecen de identidad	



- Los **atributos** son valores puros de datos sin identidad. Las **asociaciones** son conexiones entre objetos con identidad
- Las piezas individuales de comportamiento se describen mediante **operaciones**. Un **método** es la implementación de una operación.

### Clases: Notación Gráfica

- Cada clase se representa en un rectángulo con tres compartimentos
  - Nombre:** único en el paquete
  - Atributos:** propiedad con nombre, valor que describe un objeto. NO tienen identidad
  - Operaciones:** implementación de un servicio. **Método:** implementación de una operación
  - Responsabilidades:** obligación de la clase. Puede no estar en la definición de una clase, es opcional



### Relaciones

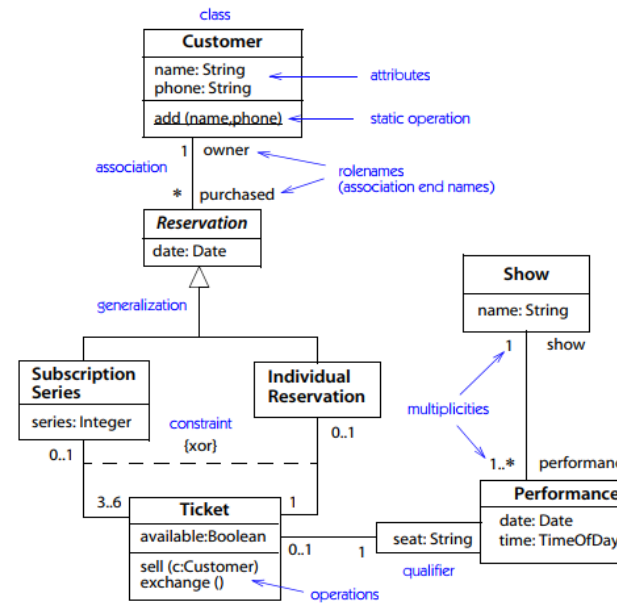
- Las relaciones entre clasificadores son
  - Asociación**
  - Generalización**
  - Dependencias**
    - Uso
    - Realización
- Si bien la asociación y las dependencias son muy parecidas, son muy distintas...

Relación	Función	Notación
asociación	Una descripción de una conexión entre instancias de clases	—
dependencia	Una relación entre dos elementos del modelo	— —>
generalización	Una relación entre una descripción más específica y una más general, utilizada para la herencia y para declaraciones de tipo polimórfico	— —>
realización	Relación entre una especificación y su implementación	— —>
uso	Una situación en la cual un elemento necesita de otro para su correcto funcionamiento	«kind» — —>

### Asociación

- Conexión semántica entre instancias de clases
- Relación estructural entre instancias
- Proporciona una conexión entre los objetos para el envío de mensajes
- Describe conexiones discretas entre objetos u otras instancias de un sistema
- Llevar la información sobre las relaciones entre los objetos del sistema
- “Da sentido a esa relación entre dos clases. Lo que me representa es una conexión entre objetos de esa clase. Es una relación estructural de las instancias. La instancia de una relación “asociación” se llama “enlace”.”
- Un objeto solo se puede asociar a sí mismo si la clase aparece más de una vez en la asociación, pero si esto se da, las dos instancias no tienen porqué ser el mismo objeto.
- Cada conexión de una asociación con una clase se denomina extremo de la asociación
- Pueden tener nombres de rol, visibilidad, navegabilidad y multiplicidad, estos se denominan adornos
- Enlace:**
  - Instancia de una asociación
  - Lista ordenada de referencias a objetos
- Adornos:** se llaman adornos ya que son opcionales, pero si no se ponen pierde el sentido el enlace
  - Nombre**
    - Puede tener un nombre en cada dirección
    - Puede omitirse en caso de tener nombres de rol
    - Es preferible ponerle nombre
  - Rol**
    - Va pegado a la clase que se quiere describir
  - Multiplicidad**
    - Especificación de multiplicidad (mínima ... máxima)
    - La multiplicidad mínima  $\geq 1$  establece una restricción de existencia

- o **Visibilidad**
- o **Agregación/Composición**
  - Representa una relación todo-partes entre objetos
  - Variante de la asociación con mayor fuerza semántica
  - **Agregación**
    - Pertenencia débil de las partes con el todo
    - Una parte puede tener varios todos
    - Cuando hablamos de una agregación es una asociación todo-parte
  - **Composición**
    - **Es una relación todo parte donde la parte es única a ese todo y no puede pertenecer a otros todos**
    - Forma de asociación más fuerte en la cual el compuesto es responsable de gestionar sus partes, por ejemplo, asignación y desasignación
    - Implica tres cosas
      - o Dependencia existencial
      - o Una pertenencia fuerte
      - o Los objetos contenidos no son compartidos
      - o No hay problema en compartir la clase, pero el objeto de esa clase no se comparte



### Generalización

- Relación taxonómica entre una descripción general y otra más específica que la extiende
- Relación “es un tipo de”, entre clases, es una abstracción de una abstracción
- **Herencia:** mecanismo a través del cual los atributos, operaciones y restricciones definidas para una clase, denominada superclase (o clase padre), pueden ser heredados (reutilizados) por otras clases denominadas subclases (o clases hijas)
- Aquí no intervienen los objetos, es una relación entre clases, o también lo podemos ver como una abstracción de una abstracción

### Dependencia

Es una relación entre dos elementos en la que un cambio en un elemento (el proveedor) puede afectar o suministrar información necesaria para el otro elemento (el cliente)

Agrupar varios tipos diferentes de relaciones, pueden tener estereotipo para establecer la naturaleza precisa de la dependencia. La dependencia puede ser concreta



### Clases: Niveles de Visibilidad

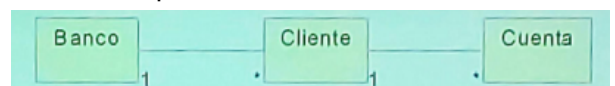
- El nivel de visibilidad especifica si el elemento de una clase puede ser usado por otra clase
- Ayuda a definir el nivel de encapsulamiento
- pueden ser:
  - o **Public:** sin restricción. Se representa con un +
  - o **Package:** es una especie de carpeta. Se representa con un ~
    - si un atributo está declarado como package todas las clases de ese paquete tendrán acceso, pero no las clases de otros paquetes
  - o **Protected:** mecanismo de herencia. Se representa con #
  - o **Private:** solo la misma clase. Se representa con un -

buscar en el libro para entender

### Diagrama de Clases y Objetos

El Diagrama de Clases y el Diagrama de Objetos pertenecen a dos vistas complementarias del modelo

- Un **Diagrama de Clases** muestra la abstracción de una parte del dominio



- Un **Diagrama de Objetos** representa una situación concreta del dominio. En este diagrama solo pueden haber objetos que pudiesen existir en el diagrama de clases original

En un diagrama de objetos no hay cardinalidad, y puede no existir el nombre del objeto

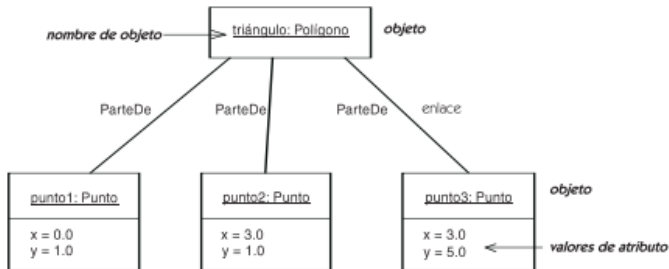
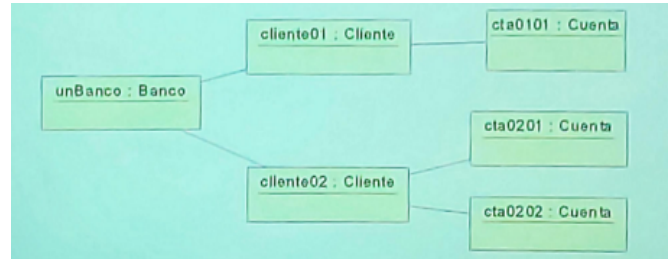


Figura 4.13 Diagrama de objetos



Se escribe  
[nombre del objeto]:[nombre de clase]  
y se subraya

24/4

## Tipos de Datos

- DataType** es un clasificador cuyas instancias son identificadas por sus valores
- Instancia = valor**
- Pueden tener operaciones (que no modifican valores)
- PrimitiveType**
  - Predefinidos en UML
  - Integer, Boolean, String, UnlimitedNatural y Real
- EnumerationType**
  - Definidos por el usuario
  - Está compuesto por un **conjunto** de valores específico
- Los utilizamos para describir los atributos
- Ese tipo de dato pueden ser tanto clases, como interfaces o algún tipo de clasificador**
- Un tipo de dato deriva de un clasificador cuyas instancias son identificadas por sus valores
- La identidad de un objeto es inherente a su existencia y la identidad de un tipo de dato viene dada por sus valores

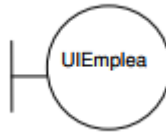
## Clases: Estereotipos

- Son un mecanismo de extensibilidad para definir meta-clases
- Pueden ampliar la semántica, pero no la estructura de las clases de metamodelos preexistentes
- Predefinidos en UML
  - Derive, executable, etc. (perfil estándar)
- Definidos por el usuario
- Los estereotipos son esas “etiquetas” entre los <<>>
- Se enriquecen los significados de los clasificadores
- Amplia el significado de la clase
- Se pueden tener estereotipos definidos por los usuarios
- Entidad - Control - Interfaz son ejemplos de estereotipos
- Están agrupados en algo llamado **perfil**
- PUDS - Objectory - Análisis

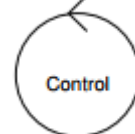
- Entidad <<entity>>**



- Interface o Frontera <<boundary>>**



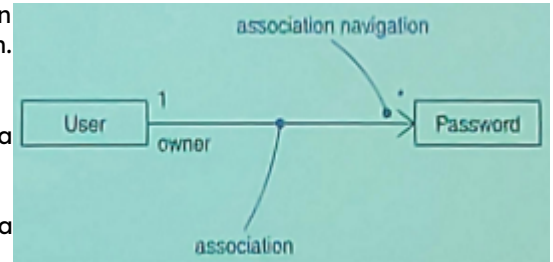
- Control <<control>>**



- Estos últimos no forman parte de UML. Forman parte del proceso unificado del desarrollo de software y son parte de la disciplina de análisis
- Es una forma rápida de reconocer de lo que se trata una clase

### Asociación: navegabilidad

- Indica la dirección en que los objetos de una clase pueden acceder a los objetos de otra clase a través de una asociación. Indica la dirección de conocimiento
- Adorno de un extremo de asociación (association end)
- El contexto de una asociación me indica si los objetos de una clase pueden ver a los objetos pertenecientes a la otra clase
- Esto forma parte del diseño
- Se utiliza exclusivamente para asociaciones binarias p/ la eficiencia de la lectura y el mantenimiento
- En este caso es unidireccional
- La visibilidad es un adorno

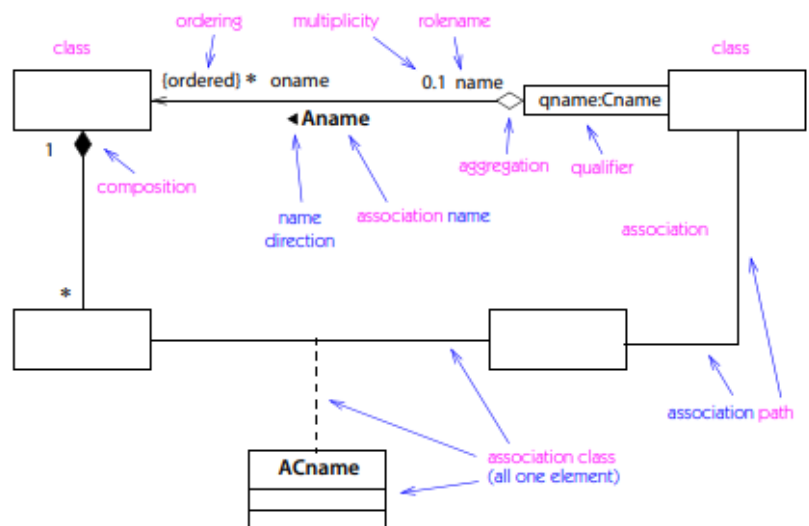


### Diferencia entre Navegabilidad (Navigability) y Posesión (Ownership)

- Navigability means that instances participating in links at runtime can be accessed efficiently from instances at the other end of the Association
  - The precise mechanism by which such efficient access is achieved is implementation specific. If an end is not navigable, access from the other ends may it may not be possible, and if it is, it might not be efficient"
  - Often used in the past according to an informal convention
    - Whereby non-navigable ends were assumed to be owned by the Association
    - Whereas navigable ends were assumed to be owned by the Classifier at the opposite end.
 This convention is now deprecated
  - Aggregation type, navigability, and end ownership are separate concepts, each with their own explicit notation
  - Association ends owned by classes are always navigable, while those owned by associations may be navigable or not
- La navegabilidad tiene que ver con que un objeto pueda acceder eficientemente a otro. Pero no implica que al no tener flecha, que no se pueda acceder, sino que no se encuentra una manera eficiencia
- Ownership of Association ends by an associated Classifier may be indicated graphically by a small filled circle (dot)
- The dot shows that the model includes a **property** of the type represented by the Classifier touched by the dot. This property is owned by the Classifier at the other end. In such a case it is normal to suppress the Property from the attributes compartment of the owning Classifier
- May be used in combination with the other graphic line-path notations for Properties of Associations and Association ends. These include aggregation type and navigability.
- Explicit end-ownership notation is not mandatory.
- Where the dot notation is used, it shall be applied consistently throughout each diagram, so the absence of the dot signifies ownership by the Association.
- La pertenencia se refiere a que un extremo no navegable es perteneciente a la asociación y que el extremo navegable es perteneciente al clasificador
- La pertenencia implica navegabilidad, pero la navegabilidad no implica pertenencia
- Convención de la cátedra
  - Usamos flecha para implicar tanto navegabilidad como propiedad
  - No usamos la cruz ni el punto salvo que se indique lo contrario

### Asociación: visibilidad

- Limita la visibilidad a través de esa asociación en relación con los objetos fuera de la asociación. Puede ser público, privado y protegido
- Privada indica que los objetos en ese extremo no son accesibles por ningún objeto fuera de la asociación
- Protegido solo accesible para objetos de clases derivadas
- Por defecto son todos públicos

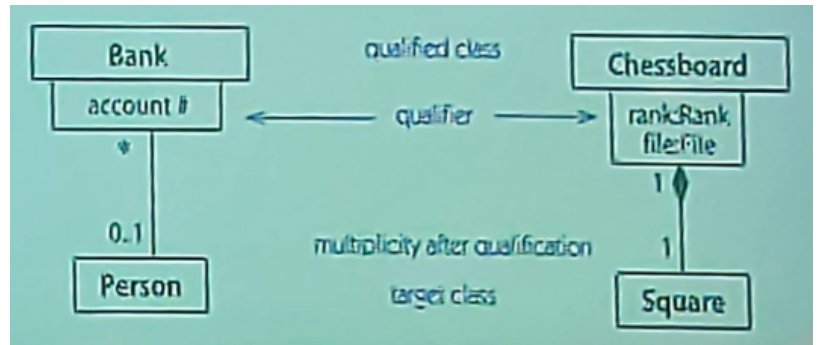


### Asociación: casos especiales

- Clase de asociación**
  - Útil cuando se necesita asignar atributos y operaciones a una asociación
  - Nombre de la clase = nombre de la asociación

Figure B-3. Association adornments within a class diagram

- o Recorro a las clases asociación cuando necesito almacenar atributos o métodos de una asociación en particular
- o O sea que la asociación pasa a tener identidad (la tiene siempre pero ahora está más en manifiesto)
- o La instancia de esa clase asociación va a estar pegada a ese enlace. Haciéndolo así no permito que haya dos presentaciones de la misma obra y en el mismo teatro.
- o El enlace tiene identidad y viene dada por los extremos
- o La existencia de la clase está vinculada a la existencia del enlace. Es muy débil. Su existencia es muy dependiente
- o De una clase asociación puedo sacar asociaciones a otras clases



#### • Asociación n-aria

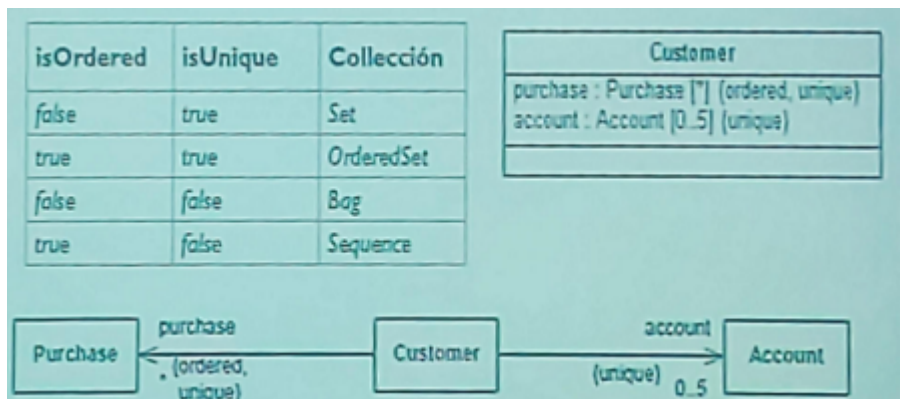
- o No existe navegabilidad

#### • Asociación calificada

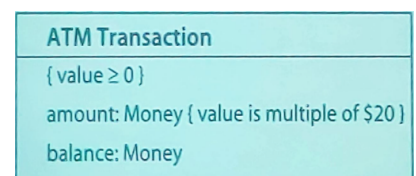
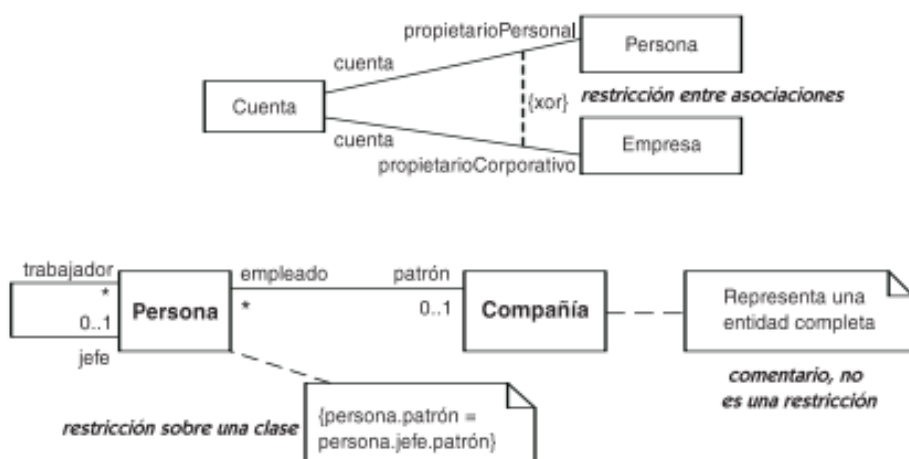
- o Calificador: es una ranura para un atributo o lista de atributos en una asociación binaria
- o Los valores de los atributos seleccionan uno o varios (conjunto) objetos relacionados
- o Generalmente implementado con algún tipo de colección indizada
- o Cuidado con las multiplicidades
- o Esto se implementa con hashtables o diccionario
- o Las multiplicidades son de muchos a muchos, pero cuando ponemos un clasificador debemos colocar la multiplicidad de este

#### • Asociación ordenada

- o Se indica mediante un elemento de multiplicidad entre llaves (modificador de propiedad)
- o No van a existir combinaciones repetidas
- o Por el lado negativo, en este tipo de asociación pueden existir dos instancias de un mismo enlace
- o ¿Porque ordenada? porque para las clases hay que aclarar el criterio de orden
- o Hay que aclarar el criterio de orden cuando tengo objeto



- **Restricción:** Una condición o restricción semántica representada como texto en lenguaje natural o un lenguaje formal específico



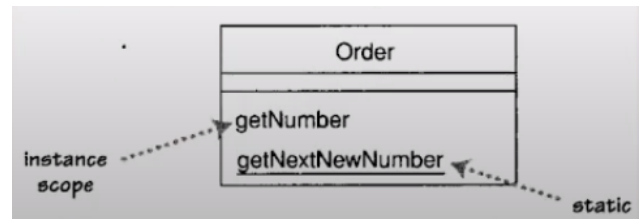
La restricción se la puede plasmar como en el ejemplo con un XOR, con una nota, o dentro de un atributo. Esta última se denomina running constraint

Figura 4.12 Restricciones



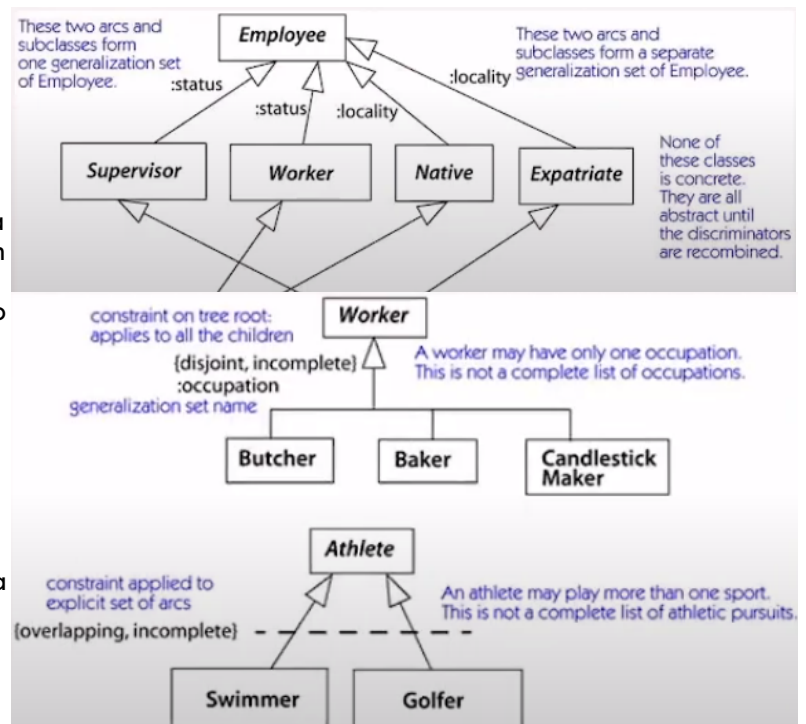
## Características estáticas o de instancia

- **Instance scope:** Cada instancia del clasificador tiene su propio valor para la característica (por defecto)
- **Static:** Sólo hay un valor de la característica para todas las instancias del clasificador (alcance de clase)
- Para los métodos de instancia se requiere instanciar un objeto, en cambio, para los estáticos se puede utilizar el método sin requerir instanciar la clase



## Conjunto de generalización

- Un conjunto de generalizaciones que componen una dimensión o aspecto de la especialización de un clasificador dado
- El conjunto generalización me genera una dimensión dentro del mismo nivel de generalización
- Para generalizar se utiliza un criterio, y al ser un aspecto de la especialización me aporta ese “criterio” que se tuvo al generalizar
- Tiene sentido cuando se está utilizando una relación de especialización y generalización
- **Restricciones**
  - **Disjoint:** Los clasificadores del conjunto son mutuamente excluyentes
  - **Overlapping:** Los clasificadores del conjunto no son mutuamente excluyentes
  - **Complete:** Los clasificadores en el conjunto cubren completamente una dimensión de especialización
  - **Incomplete:** Los clasificadores en el conjunto no cubren completamente una dimensión de especialización
  - Por defecto {incomplete, disjoint}

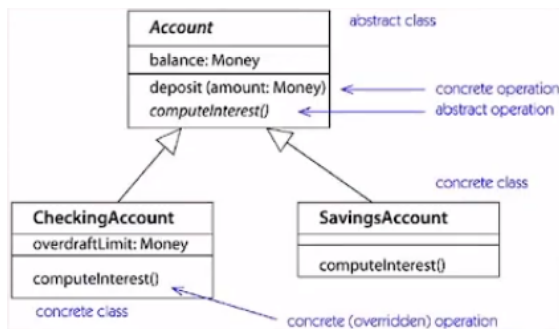


## Principio de sustitución de Barbara Liskov

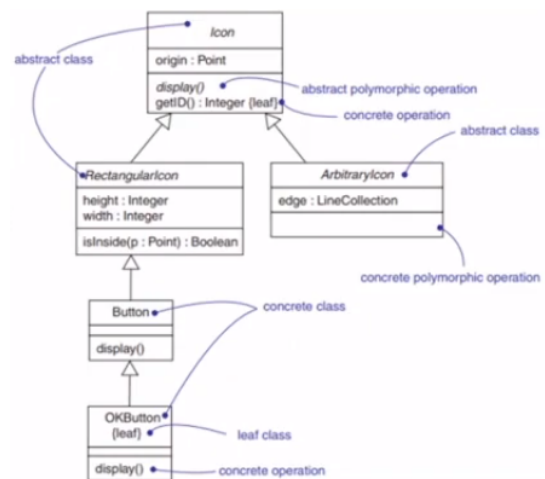
- Una instancia de un descendiente se puede utilizar donde quiera que esté declarado el antecesor
- Es decir, toda referencia a una instancia de una clase puede ser reemplazada por una instancia de cualquier subclase de esta.
- Cualquier objeto de una clase especializada puede reemplazar a un objeto de una clase padre
- **Type Hierarchy**
  - A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: If for each object  $O_1$  of type S there is an object  $O_2$  of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when  $O_1$  is substituted for  $O_2$ , then S is a subtype of T.

## Clases abstractas

- Una clase abstracta es una clase que no es instanciable
- Una operación que carece de una implementación es abstracta
- Se representa colocando en *itálicas* el nombre de la clase
- Una operación de una clase abstracta no tiene implementación y las clases hijas deben implementarlas



- Account [] cuenta;
- Cuenta[2].computeInterest();
- Leaf: no puede ser redefinido
- Operaciones polimórficas



## Interfaces

- se definen como un contrato al que las clases en relación se deben adherir
- describe un protocolo de comportamiento sin especificar su implementación
- se encarga de definir “qué puedo ofrecer” y no el “cómo lo hago” ni “quién lo hará”
- solo tiene operaciones, no atributos
- las interfaces pueden tener generalización y se la llama “realización”
- la interfaz no tiene una implementación
- una interfaz puede ser implementada por varias clases
- entre dos interfaces tengo una realización
- las dependencias no se suelen mostrar en un diagrama uml excepto entre clases e interfaces o entre paquetes
- si una interfaz es implementada por una clase utilizamos las flechas (rellena para proveer interfaz y vacía para representar la interfaz requerida
- si una interfaz es implementada por la conexión con otra interfaz entonces tenemos “**socket and ball**”
- Esa dependencia representaría un “criterio” de implementación, si una interfaz es proporcionada por una clase significa que esta clase la “realiza”, si en cambio la clase “requiere” de esta interfaz, significa que tiene una dependencia de uso, y es obligatorio que esa interfaz sea proporcionada para el correcto desempeño del componente

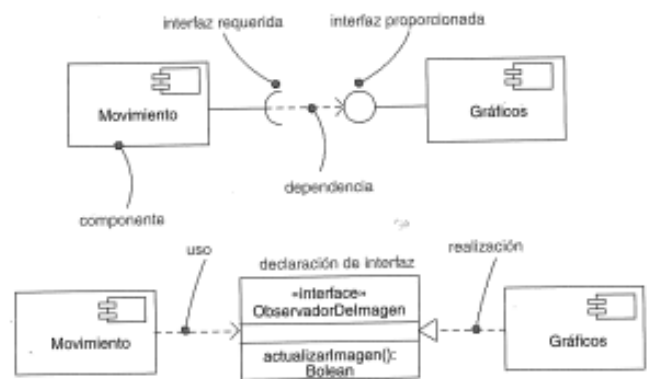


Figura 15.1: Componentes e interfaces.

## Interfaz vs Clase Abstracta

### Clase Abstracta

Una clase abstracta no puede tener instancias directas, es una clase no instanciable- es decir, que no puede tener instancias directas, bien porque su descripción está incompleta (Como cuando faltan métodos para una o más operaciones), o bien porque no se tiene la intención de instanciarla aunque su descripción esté completa. Una clase abstracta está pensada para la especialización. Para que sea útil, una clase abstracta debe tener descendientes que puedan tener instancias; una clase abstracta hoja carece de utilidad. Una clase abstracta puede tener operaciones concretas. Las operaciones concretas son aquellas que se pueden implementar una vez y utilizadas en todas las subclases.

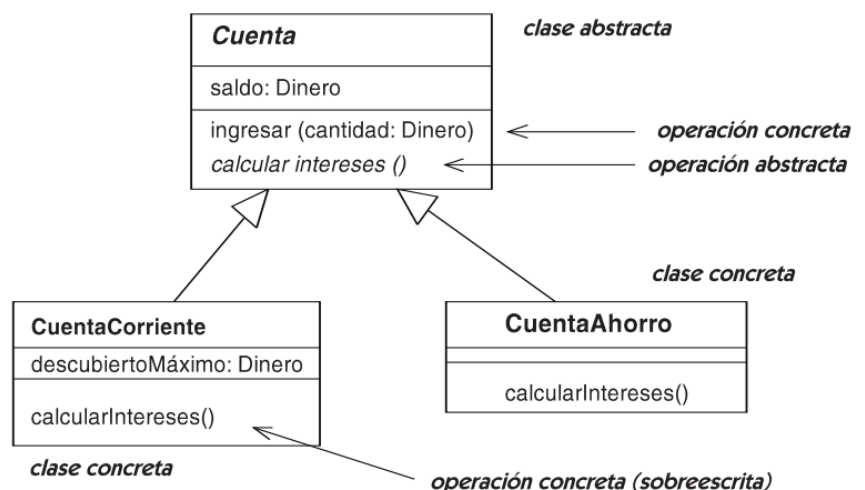


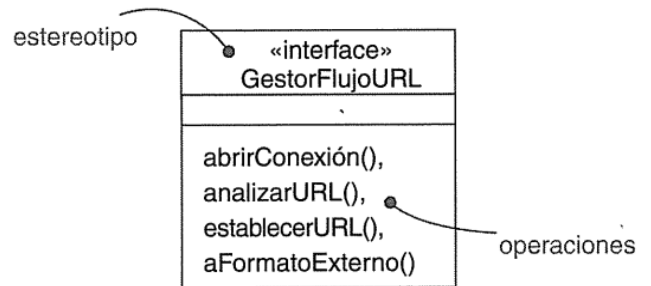
Figura 14.1 Clases abstractas y concretas

## Interfaz

Es una colección de operaciones que especifican un servicio de una clase o componente. Una interfaz bien estructurada proporciona una clara separación entre vistas externa e interna de una abstracción, haciendo posible comprender y abordar una abstracción sin tener que sumergirse en los detalles de su implementación. En UML, las interfaces se emplean para modelar las líneas de separación de un sistema. Una interfaz es una colección de operaciones que sirven para especificar un servicio de una clase o un componente. Al declarar una interfaz, se puede enunciar el comportamiento deseado de una abstracción independientemente de una implementación de ella. Los clientes pueden trabajar con esa interfaz.

Un tipo es un estereotipo de una clase utilizado para especificar el dominio de objetos, junto con las operaciones (pero no los métodos) aplicables al objeto. Un rol es el comportamiento de una entidad participante en un contexto particular. Una interfaz puede dibujarse mostrando únicamente su nombre, estos nombres pueden ser Simples o calificados, el simple es una cadena de texto, el calificado consta del nombre del interfaz precedido por el nombre del paquete en el que se encuentra.

Cuando se declara una interfaz, se representa como una clase estereotipada, listando sus operaciones en el comportamiento apropiado. Las operaciones se pueden representar sólo con su nombre o pueden extenderse para mostrar la Signatura completa y otras propiedades.



22/05

¿En qué consiste la vista de casos de uso? ¿Cuáles son los conceptos involucrados? Explicar diagrama de casos de uso. ¿Qué es la inclusión? ¿Qué es la extensión? ¿Qué plantillas existen para la descripción de los casos de uso?

## Vista de Casos de Uso

- Capturan los requerimientos funcionales del sistemas
- Describen la forma de usar el sistema tal y como se ve desde el exterior
- Visión de caja negra del sistema
- No es un modelo orientado a objetos
- Particiona la funcionalidad del sistema en unidades discretas: los casos de uso
- Diagrama de Casos de Uso: Actores + Casos de Uso

## Caso de Uso

- Secuencia de transacciones realizadas por el sistema que brinda un resultado de valor a un actor.
- Describe una “forma” de utilizar el sistema.
- Estructuran los modelos de objetos en vistas manejables.
- Pueden tener varios caminos de acción o “escenarios”.
- Sirven como hilo conductor del proceso de desarrollo.
- capturan requerimientos funcionales del sistema.

## Especificación de Casos de Uso

- descripción textual del larman, de uml no aprendemos a describir casos de uso
- los casos de uso son documentos de texto, no diagramas
- las plantillas se hacen en la primera etapa del inicio del proyecto
- tenemos plantillas intermedias con los caminos alternativos y estándar. De estas, que describen los caminos alternativos solo necesitamos saber los títulos, es parte por eso que tenemos diferentes niveles de abstracción
- esto se hace especificando de manera iterativa e incremental
- a las excepciones llamamos caminos alternativos, ya que son flujos que no necesariamente están mal
- no utilizamos el esquema que usa condicionales
- **los casos de uso pueden ser escritos en varios formatos y niveles de abstracción/formalidad**
  - **Brief:** Terse one-paragraph summary, usually of the main success scenario. The prior Process Sale example was brief.
  - **Casual:** Informal paragraph format. Multiple paragraphs that cover various scenarios. The prior Handle Returns example was casual.
  - **Fully dressed:** All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees
- generalmente, utilizamos un formato similar a la descripción completa o “fully dressed”

## Descripción de Casos de Uso

Plantilla estándar “fully-dressed” según Larman



Item	Description
Use Case Name	A brief description of the use case. Start with a verb
Scope	The system or subsystem that is being described (under design).
Level	Whether the use case is a user goal or a subfunction.
Primary Actor	The actor who initiates the use case. Calls on the system to deliver it services
Stakeholders and Interests	The people or organizations who are affected by the use case. Who cares about this use case, and what do they want?
Preconditions	The conditions that must be met before the use case can begin.
Success Guarantee	The conditions that must be met for the use case to be considered successful.
Main Success Scenario	A typical, successful execution of the use case.
Extensions	Alternative scenarios that may occur during the use case.
Special Requirements	Any non-functional requirements that apply to the UseC. Related non-func. req
Technology and Data Variations List	A list of the different ways that the use case can be implemented. Varying I/O methods and data formats
Frequency of Occurrence	How often the use case is expected to be executed. Influences investigation, testing and timing of implementation
Miscellaneous	Any other relevant information about the use case.

[https://www.craigharman.com/wiki/downloads/applying\\_uml/larman-ch6-applying-evolutionary-use-cases.pdf](https://www.craigharman.com/wiki/downloads/applying_uml/larman-ch6-applying-evolutionary-use-cases.pdf)

- Para describir los caminos alternativos, primero indicamos la parte numérica en donde se va a dar la alternativa, y luego a qué alternativa corresponde, indicando el título del camino alternativo
- Larman no divide explícitamente a los casos de uso del sistema y del negocio, los mezcla, por lo que las interacciones fuera del sistema no se agregan a la descripción de casos de uso. Nada de esto es UML, UML son los diagramas, óvalos y stickmans...
- las vistas de casos de uso capturan los...

## Actor

Representa algo que interactúa con el sistema, describe un rol que asume el usuario. Reside fuera del sistema y describe su entorno

## Identificación

- Identificación a través de actores: identificar los actores que comunicaran con el sistema
- Identificación a través de eventos: Eventos temporales o externos que el sistema debe ser capaz de responder

## Relaciones

- **Generalización:** distintas variantes de un caso de uso ("es un tipo de"). Tiene una flecha de herencia, agrupa casos de uso con funcionalidades similares
- **Inclusión:** los CU **no están completos** sin los CU extendidos (incluidos). La flecha va del lado incluido y parte del caso base.
  - ¿Cómo represento la inclusión en un caso de uso? se representa con un "incluye nombre\_casodeuso"
  - hay casos de usos abstractos y casos de usos concretos, los abstractos no pueden instanciarse por sí mismo, generalmente los bases son concretos y los incluidos abstractos
  - es la que más se usa, se aconseja su uso cuando tenemos pasos o secuencias comunes a varios casos de uso "algo que podemos factorizar"
- **Extensión:** son utilizados de forma condicional u opcional, los CU están completos sin los CU extendidos
  - la flecha va del lado del caso base y parte de las "partes extendidas"
  - Es factible hacer un extend como un camino alternativo, conviene trabajarlo como caso de uso cuando es una recopilación de pasos bastante grande, ya tuvimos una abstracción pero queremos congelar el resultado???

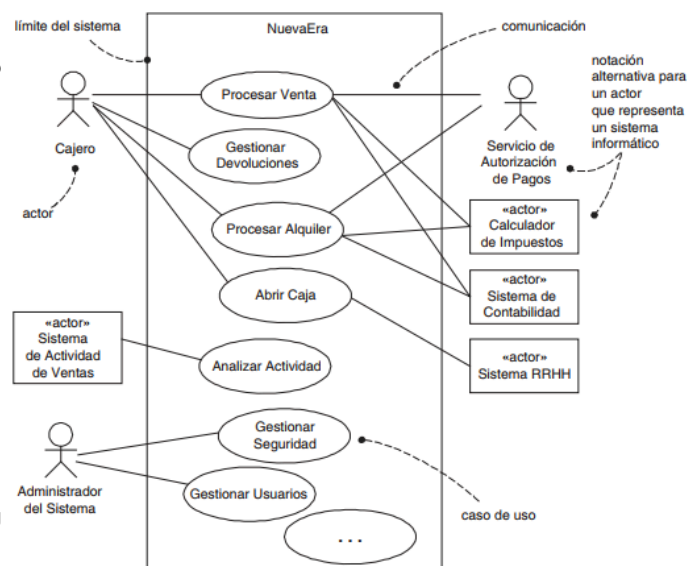


Figura 6.2. Diagrama de contexto de casos de uso parcial.

- se utiliza mas q nada para agregar una funcionalidad, no se comparte con otros casos, justamente se la agrega pq no está

## Diagrama de Casos de Uso

En el diagrama podemos observar actores principales, de los cuales solo uno podrá iniciar a la vez un caso de uso, este diagrama permite que más de un actor pueda iniciar un caso de uso, pero un caso de uso será instanciado por solo un actor

## Vista de diseño

- Muestra la descomposición del sistema en unidades modulares con límites de encapsulación y sus interfaces externas
- En el diseño se puede requerir conocer las partes constitutivas de un clasificador
- Es una vista más próxima a la implementación
  - Clases (clasificador) estructuradas
  - Colaboraciones
  - Componentes
- Son diagramas que permiten especificar que un clasificador contiene una estructura interna
- Define “cómo” está estructurado a nivel interno un clasificador

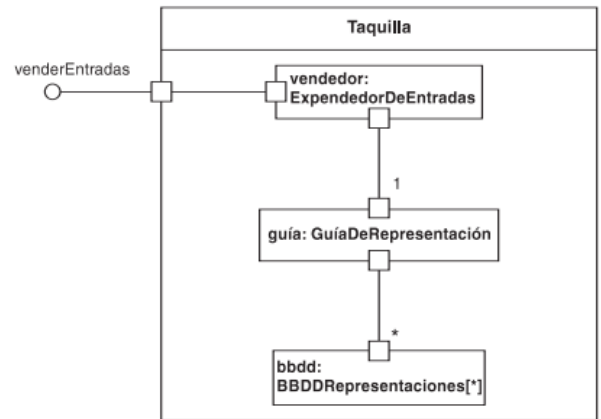


Figura 3.2 Diagrama de la estructura interna

## Estructura Interna

**Conceptos principales:** Conector, interfaz, interfaz obligatoria, interfaz proporcionada, parte, puerto

### Clasificador estructurado

- Es un clasificador con estructura interna que comprende una red de elementos conectados, cada uno de los cuales desempeña un rol en el comportamiento general modelado por el CE
- Los roles pueden estar relacionados por conectores. Cada rol tiene un nombre, un tipo y una multiplicidad dentro de su contenedor. Los roles que posee un CE a través de composición se denominan partes. Un objeto que es una parte puede pertenecer sólo a un objeto estructurado. Los roles en el diagrama de interacción son básicamente objetos
- Los roles que son **parte** en una clase estructurada se representan con línea continua, los roles que existen por afuera del CE se representan con línea punteada

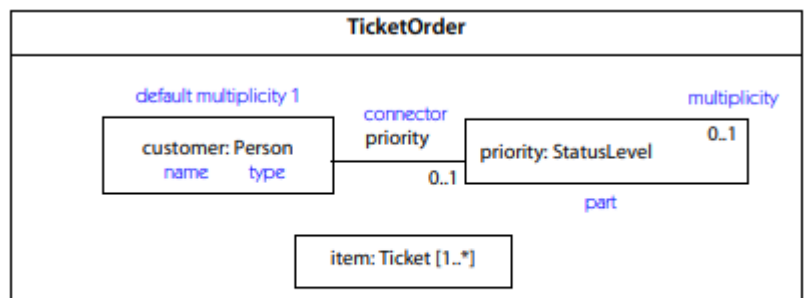


Figure 5-1. Structured class

### Conector

- Es una relación contextual entre dos roles/partes (dentro del mismo contexto en tiempo de ejecución, es decir, objeto compuesto, y pertenecen a la misma instancia compuesta)
- A diferencia de una asociación, cada asociación a una clase es independiente y no garantiza que los objetos de las partes conectadas estén contenidos dentro del mismo objeto compuesto

### Puertos

- Punto de interacción con una interfaz bien definida. Característica estructural de un clasificador que encapsula la interacción entre los componentes del clasificador y su entorno
- Se usa en clasificadores encapsulados (un tipo de CE)
- Tienen un conjunto de interfaces provistas e interfaces requeridas que definen sus interacciones externas
- Permiten modificar la estructura interna de un clasificador sin afectar a los clientes externos
- Permiten que las partes (piezas) encapsuladas se “conecten entre sí”

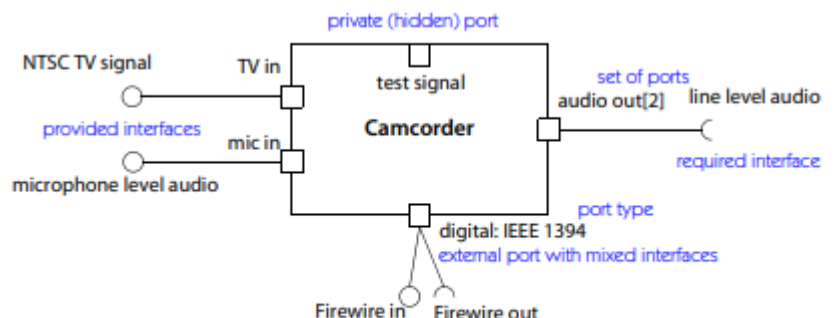


Figure 5-2. Structured class with ports

para formar la implementación de un clasificador

- **Se pueden declarar lo siguiente:**
- **De servicio (service port):** Si se declara como “false” el puerto va a ser utilizado solo en la implementación interna del clasificador y no es necesario para el entorno., por lo tanto puede ser borrado o alterado sin afectar el uso del clasificador. Por default se define como “true”, es decir, que es necesario para el entorno
- **De comportamiento (behavior port):** Si se declara como “true” las solicitudes se implementan mediante el comportamiento declarado del clasificador (como una máquina de estado o procedimiento). Si se declara como “false” se lo llama puerto de delegación
- **De delegación (delegation port):** Está conectado a un puerto de una parte interna. Una solicitud externa se transmitirá a la parte interna para su implementación.
- Un puerto puede implementarse o no, y en este caso representaría simplemente un concepto virtual
- Un puerto puede ser público, privado o protegido

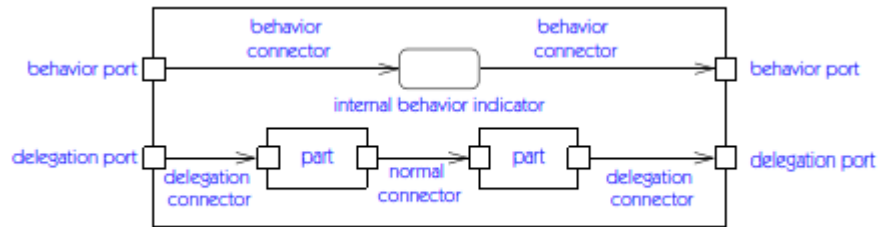


Figure 14-216. Internal ports in a structured classifier

## Diagrama de Colaboración

**Conceptos principales:** Colaboración, conector, rol, uso de la colaboración

### Colaboración

- Es un tipo de clasificador estructurado. Es una descripción de una colección de objetos que interactúan para implementar algún comportamiento en un contexto
- Contiene roles que son ocupados por objetos en tiempo de ejecución
- Un objeto puede participar en más de una colaboración con el mismo o distintos roles
- Un conector representa una descripción de las asociaciones entre los roles de la colaboración. Las relaciones entre roles y conectores dentro de una colaboración sólo son significativas en ese contexto
- Una colaboración es una forma de implementar un CU
- Las colaboraciones están íntimamente relacionadas con las interacciones

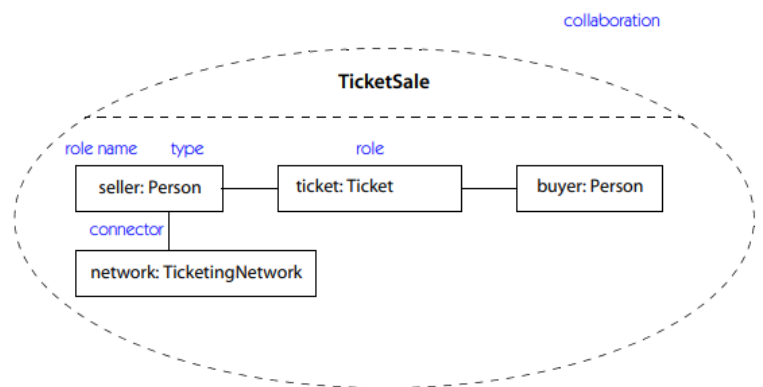


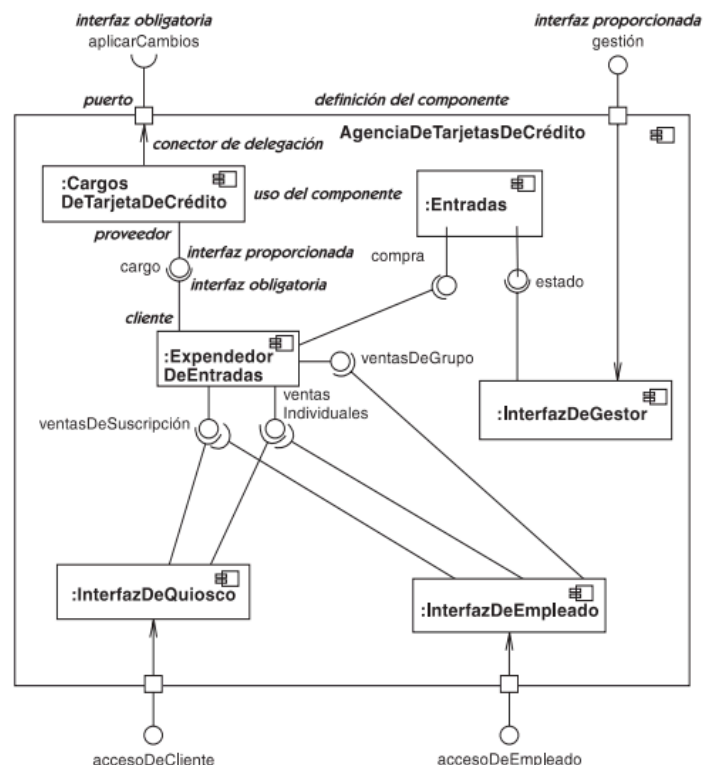
Figure 5-3. Collaboration definition

## Diagrama de Componentes

**Conceptos principales:** Proporcionalidad, interfaz obligatoria, puerto, realización, subsistema

### Componentes

- Representan una pieza modular de un sistema lógico o físico
- No dependen directamente de otros componentes, sino de las interfaces compatibles con los componentes
- Son sustituibles y autocontenidos
- Tienen interfaces que soportan (interfaces provistas) e interfaces que requieren de otros componentes (interfaces requeridas)



**REPASO DE CUESTIONARIO:**

- DIFERENCIA ENTRE UN ACTOR PRIMARIO Y UN ACTOR SECUNDARIO EN UN CASO DE USO**

Tiene que haber uno y solo un actor primario en un caso de uso. Son secundarios los que participan, pero no dan inicio al CU

- El **actor primario** es aquel que inicia el caso de uso y tiene un objetivo principal que cumplir dentro del mismo. Es el motor que impulsa el caso de uso y participa activamente en cada paso del proceso.
- El **actor secundario** es aquel que participa en el caso de uso de manera indirecta o auxiliar, proporcionando información o recursos al sistema o al actor primario. Su participación no es esencial para el flujo principal del caso de uso, pero puede contribuir a su éxito o eficiencia
- Ej: En un caso de uso de "Retirar efectivo de un cajero automático", el **actor primario** sería el **cliente que desea retirar dinero**. El cliente inicia el caso de uso insertando su tarjeta, ingresando su PIN, seleccionando la cantidad de dinero y retirando el efectivo. El **actor secundario** podría ser el **banco que administra la cuenta del cliente**. El banco no inicia el caso de uso ni tiene un objetivo directo, pero proporciona la información de la cuenta y el dinero al cliente a través del cajero automático

Característica	Actor primario	Actor secundario
Papel en el caso de uso	Inicia y participa activamente	Participa de manera indirecta o auxiliar
Objetivo	Tiene un objetivo principal que cumplir	No tiene un objetivo principal
Interacción con el sistema	Interactúa directamente con el sistema	Puede o no interactuar directamente con el sistema
Importancia para el caso de uso	Es fundamental para la ejecución exitosa	Su participación no es imprescindible, pero puede ser útil

## Área dinámica

### Vista de interacción

Es la vista de un modelo que muestra el **intercambio de mensajes** entre objetos (partes) para lograr un objetivo. Esta vista se modela mediante **interacciones** que actúan sobre **clasificadores estructurados** (incluidas las **colaboraciones**). Una interacción es un conjunto de mensajes que se intercambian dentro del contexto de una colaboración por instancias de clases (objetos) a través de enlaces (instancias de asociación).

Es mostrado a través de diagramas de secuencia y de comunicación (antes colaboración)

La Interacción me explica la forma dinámica de una colaboración. Para llevar a la práctica un Caso de Uso necesito una colaboración de un conjunto de clases; cada interacción de un caso de uso se realiza dentro de un diagrama de secuencia

Hay dos diagramas más en uml en la vista de interacción, pero no están en el manual de referencia

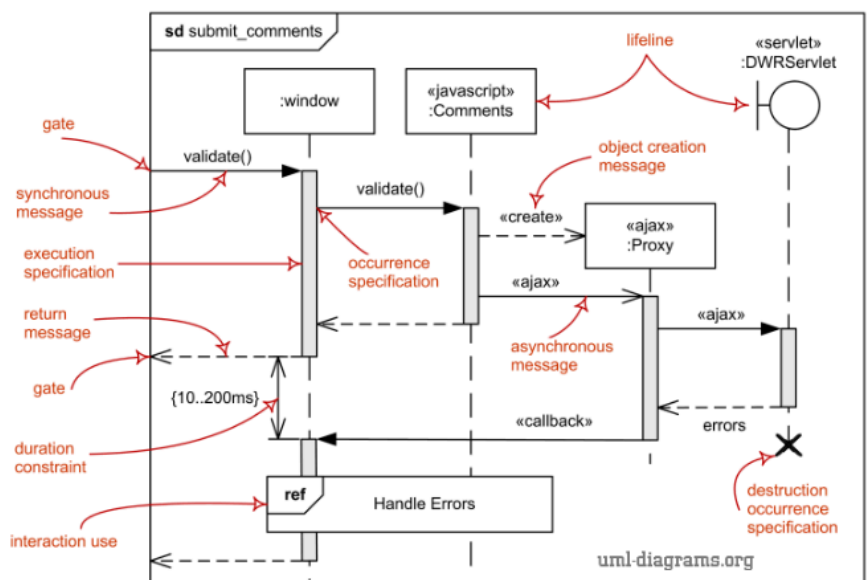
- diagrama de tiempo**
- diagrama de overview**

### Colaboración

- El aspecto estático es similar a la vista estática, contiene un conjunto de roles y sus relaciones que definen el contexto para el aspecto dinámico
- El aspecto dinámico es el conjunto de mensajes intercambiados entre los objetos que ocupan los roles. Tal intercambio de mensajes en una colaboración es llamado una interacción
- Una colaboración puede incluir una o más interacciones

### Diagramas de secuencia

- Es un diagrama que muestra las interacciones ordenadas por



secuencia temporal

- Muestra roles (representando objetos) en una interacción y su secuencia de mensajes
- Línea de vida: rol + línea de guiones
- Es una especificación de ejecución, muestra a un objeto activo durante la ejecución de un método
- Se intercambian mensajes usando flechas
- Al trabajar con un diagrama de secuencia se lo envuelve en un cuadro que se llama SD y habla de la interacción de manera detallada
- La forma genérica de llamar al cuadrado es el rol que para nosotros es el objeto pero no necesariamente siempre. Pueden ser actores cuando se realizan los casos de uso
- Argumento es el valor que utilizo cuando llamo al método, parámetro es cuando defino el método

### Operadores de interacción

- Alt – alternatives
- Opt – option
- Loop – iteration
- Break – break
- Par – parallel
- Ref – interaction use

### Diagrama de comunicación

- Hace énfasis en la distribución física y las relaciones entre los objetos
- En este diagrama se omite la duración de los mensajes, son isomorfo

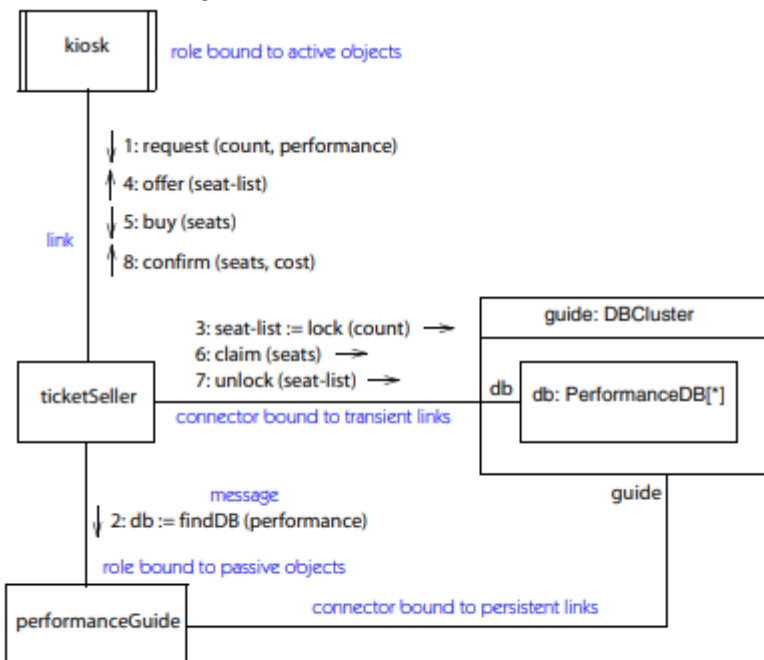


Figure 3-10. Communication diagram

## Vista de Máquina de Estados

- Describe el comportamiento dinámico de los objetos, modelando su ciclo de vida
- Autómatas finitos con estados y transiciones
- Cada objeto se trata en forma aislada, al que se comunica con el resto del mundo detectando eventos y respondiendo a ellos
- Es útil modelar solo para objetos con comportamiento estado-dependiente
- Uso de diagramas de estado

¿Por qué se necesitan máquinas de estados? Cada objeto es como un ente que tiene un ciclo de vida y que responde a ciertos estímulos.

¿En qué casos conviene modelar a través de una máquina de estados? Cuando vemos que hay una dependencia grande del comportamiento. Todos los objetos tienen estados, pero muchas veces no los tenemos bien definidos y muchas veces no nos interesa tener el control. Es necesario cuando el objeto cambia su comportamiento dependiendo si está en un estado particular. Tratamos de modelar modelos bien específicos. Se centra en el objeto y en una clase en particular, no en las relaciones de estos.

### Beneficios de usar diagramas de máquina de estados:

- **Visualización del comportamiento:** Los diagramas de máquina de estados proporcionan una forma visual clara y concisa de representar el comportamiento de un sistema.
- **Mejora de la comprensión:** Al visualizar el comportamiento del sistema, los diagramas de máquina de estados ayudan a los desarrolladores y usuarios a comprender mejor cómo funciona el sistema.
- **Detección de errores:** Los diagramas de máquina de estados pueden ayudar a detectar errores en el diseño del sistema, como estados inaccesibles o transiciones no deseadas.
- **Facilidad de mantenimiento:** Los diagramas de máquina de estados pueden facilitar el mantenimiento del sistema, ya que proporcionan una representación clara de su comportamiento.

### Diagrama de Estados

[dibujo de la transición o arco desde un estado A a un estado B una vez que se cumpla un evento o condición]

Se puede no tener un evento o condición/acción, pero conviene tenerlo

Los estados se llaman “vértices” y los eventos “arcos”

### Estado

- Es una condición o situación en un periodo de tiempo de la vida de un objeto durante el cual:
  - Un conjunto de valores de atributos y enlaces del objeto satisface alguna condición
  - Se espera que ocurra un evento
  - El objeto realiza una actividad
- El estado en el que se encuentra un objeto determina su comportamiento
- El comportamiento del objeto está reflejado en el diagrama de estado asociado a su clase
- Puede tener nombre, aunque a menudo son anónimos y se describen simplemente mediante sus efectos y relaciones
- Son unidades de control

### Transiciones

- Una transición que deja un estado define la respuesta de un objeto en ese estado a una ocurrencia de un evento
- En general, una transición tiene un evento que la dispara, una condición de guarda, un efecto y un estado destino.
- Arcos
- No es una asociación
- Caracterizado por:
  - **Estado de origen:** El estado en el que se encuentra el objeto antes de la transición.
  - **Evento disparador:** El evento que desencadena la transición.
    - Un evento puede ser un evento externo (como la entrada del usuario) o un evento interno (como la finalización de un proceso).
  - **Condición de guarda:** Una condición que debe cumplirse para que la transición se active.
  - **Acción (efecto):** Las acciones que se ejecutan cuando la transición se activa.
  - **Estado destino:** El estado en el que se encuentra el objeto después de la transición.
- Tipos de transiciones
  - **Externa:** cuando se pasa de un estado a otro. Incluye la auto-transición
  - **Interna:** se escuchan eventos, no hay cambios. No se modelan con “flechitas” pero se escriben, son solo texto
  - **Local:** no cambia el estado contenedor. Solo en estados compuestos, produce un cambio entre los estados internos del estado compuesto
  - **De finalización:** carece de evento disparador explícito. Es accionada por la finalización de la actividad del estado del que se vale.
- Las transiciones no son instantáneas, lo instantáneo es el evento
- Las transiciones internas no se ven en un diagrama, las transiciones locales se ven, pero es algo MUY específico
- En un evento la condición de guarda se especifica con corchete, los eventos
- En un diagrama de estado UML, una transición se representa por una flecha que va desde el estado de origen al estado de destino. La flecha puede estar etiquetada con el evento disparador, la condición de guarda y la acción.

### Eventos

- Los eventos representan el tipo de cambios que un objeto puede detectar
- Acontecimiento significativo que tiene localización en tiempo y espacio
- Cualquier cosa que pueda influir en un objeto se puede caracterizar como un evento

Tabla 7.1 Tipos de eventos

Tipo de evento	Descripción	Sintaxis
evento de cambio	Un cambio en el valor de una expresión Booleana	cuando (exp)
evento de llamada	Recepción, por un objeto, de una petición explícita sincrónica	op (a:T)
evento de señal	Recepción de una comunicación asincrónica, explícita y con nombre, entre objetos	nombreS (a:T)
evento de tiempo	La llegada de un tiempo absoluto o el transcurso de una cantidad relativa de tiempo	tras (tiempo)



- Es un tipo de ocurrencia significativa que tiene una localización en tiempo y espacio
- Ocurre en un punto de tiempo y no tiene duración. Es instantáneo
- Se modela como evento si su ocurrencia tiene consecuencias (cambio en un objeto)
- Los eventos pueden tener parámetros que caractericen cada ocurrencia de un evento individual, de la misma forma que las clases tienen atributos que caracterizan cada objeto.
- Se pueden modelar con clases y jerarquías
- Pueden ser:

- **De Señal:** tipo de clasificador que está pensado explícitamente como un **vehículo de comunicación**. Es un evento **asíncrono**. El objeto emisor, crea e inicializa explícitamente una instancia de la señal. La recepción de una señal es un evento para el objeto receptor. Además de ser mensajes se pueden modelar como clases. Esto lo hace de manera asíncrona y **unidireccional**. Se pueden utilizar múltiples señales para modelar comunicaciones bidireccionales. El emisor y el receptor pueden ser el **mismo objeto**. Puedo tener señales internas y externas, dependiendo de si la hace un actor o si se produce dentro del sistema y dentro de un objeto particular

- **De Llamada:** es la recepción de una llamada de operación por un objeto. La clase receptora elige si una operación se implementará como método o como disparador de un evento de llamada en una máquina de estados. Una vez que el objeto receptor procesa el evento tomando una decisión, devuelve el control al objeto que realizó la llamada. A diferencia de un método, una transición de una máquina de estados disparada por un evento de llamada puede continuar su propia ejecución en paralelo con el que realizó la llamada
- **De Cambio:** es una manera declarativa de esperar hasta que se satisfaga una condición, representa un cómputo continuo y potencialmente no local. Solo se deben utilizar cuando no es natural una forma más explícita de comunicación. A diferencia de una condición de guarda, el evento de cambio se evalúa continuamente hasta que se convierte en verdadero, momento en el cual se dispara la transición; en cambio, en la condición de guarda, la condición se evalúa **una vez** cuando ocurre el evento disparador.
- **De Tiempo:** este puede ser especificado de un modo absoluto (hora) o de un modo relativo (tiempo transcurrido desde un evento dado). Representan el paso del tiempo

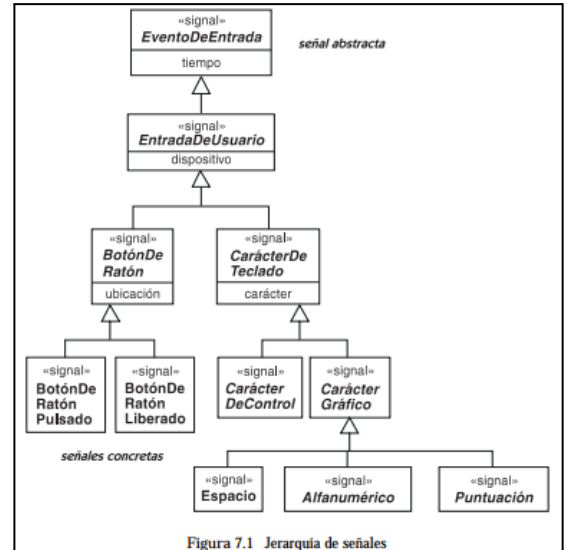


Figura 7.1 Jerarquía de señales

### Condición de Guarda

{completar}

Es la condición que tiene una transición para ejecutarse (?)

### Acciones (eventos)

- Una acción es un cómputo atómico y breve
- Acciones específicas de entrada y salida que se pueden dar durante un evento o **por** un evento

Do es una acción que se va a ejecutar durante toda la vida de ese estado, es decir, mientras que ese estado esté activo, ejemplo, el create() en un diagrama de actividades, tiene un solo estado, el de creación, en el ejemplo que dio del diagrama de secuencias

On evento: acción, internal transition

Evento diferido o pospuesto, voy apilando los eventos, cuando pase este estado le voy a mandar a la próxima etapa, y si la próxima etapa escucha esos eventos los va a empezar a desapilar y a ejecutar, si no son de interés para esa etapa se pierden

### Estados Compuestos (subestados)

Estados compuestos con estados, los estados dentro del estado compuesto se llaman subestados, mientras que el estado compuesto está activo se pueden ir ejecutando los subestados, eso va a tener un cambio del subestado, pero no del estado compuesto, por lo tanto, va a ser una transición **LOCAL** no interna, por lo tanto, se modela **DENTRO** del estado compuesto (estado contenedor de subestados)

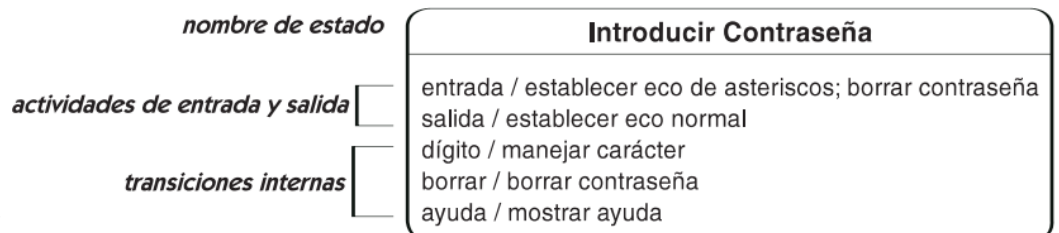


Figura 7.4 Transiciones internas y acciones de entrada y salida

Estados NO ORTOGONALES significa que un objeto puede tener un estado a la vez

### Estado Compuesto Ortogonal

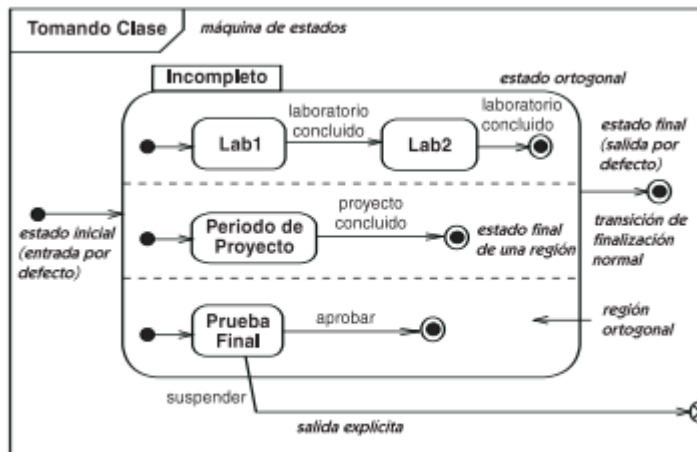


Figura 7.6 Máquina de estados con un estado compuesto ortogonal

### Estado Compuesto No Ortogonal

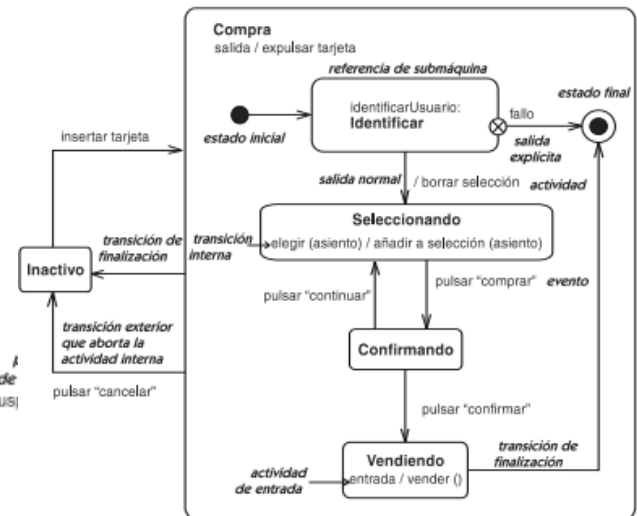


Figura 7.5 Máquina de estados

### Estado de Submáquina

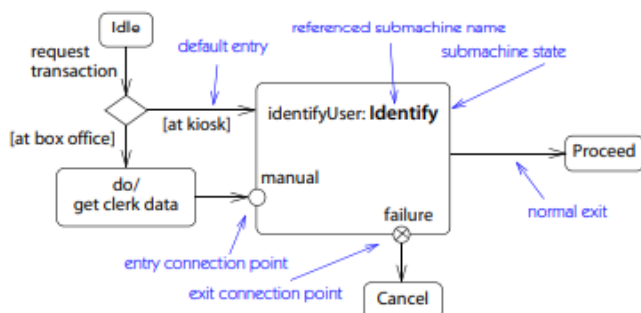


Figure 14-265. Submachine state

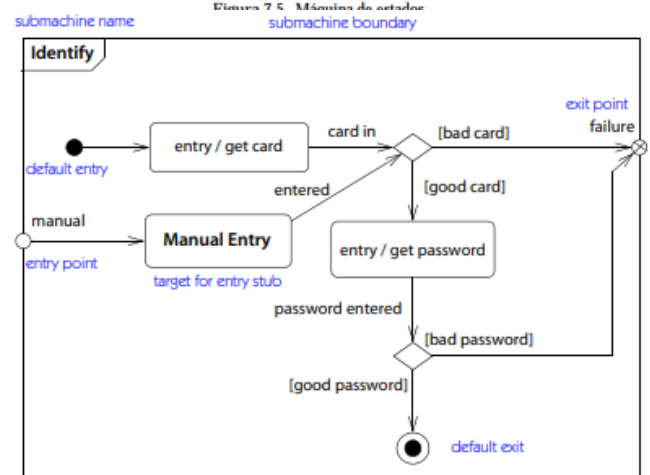


Figure 14-266. Submachine definition

Identify es una submaquina, identifyUser va a llamar a identify para "copiar" su funcionamiento

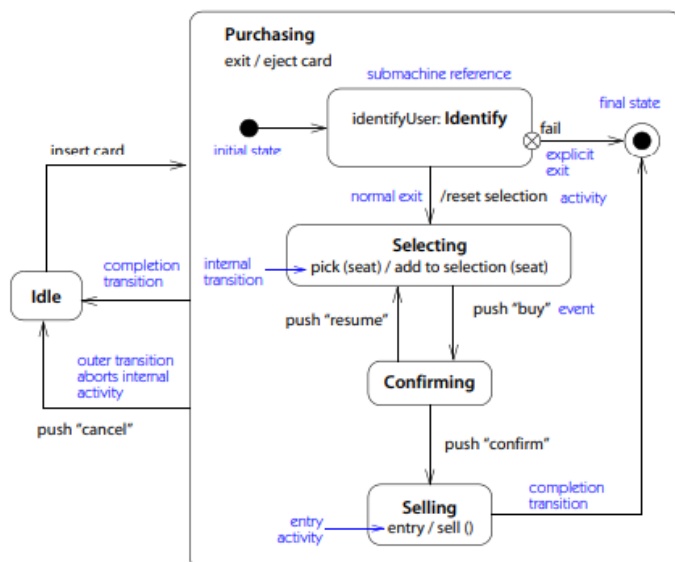


Figure 7-5. State machine

### Temas de la Clase:

# Modelado de Flujos de Actividades. Actividades, transiciones, control del flujo. Diagrama de Actividades. Modelado de Interacción de Objetos. Colaboración de objetos. Aspecto estático y dinámico de la colaboración. Diagramas de colaboración y secuencia.

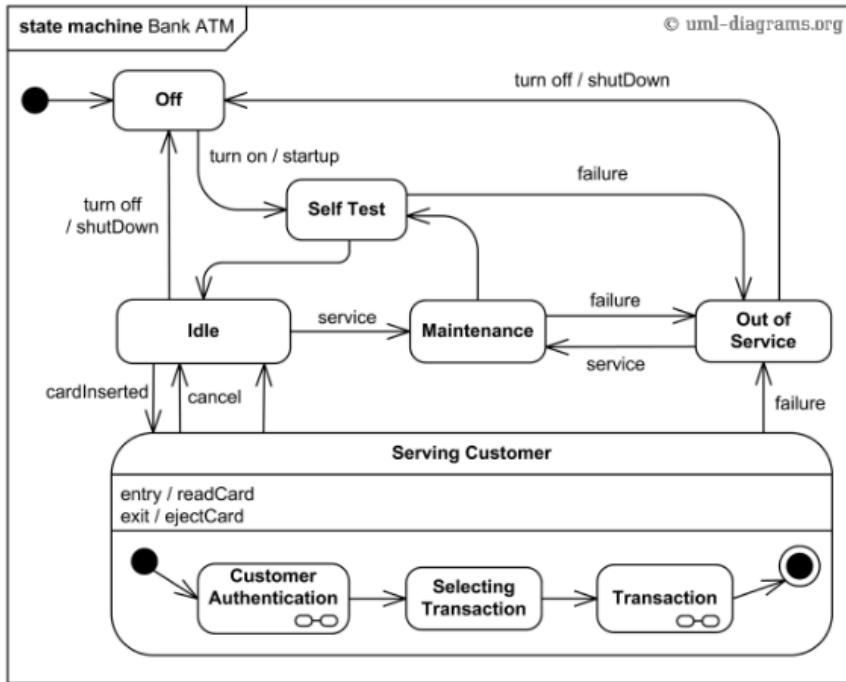
# Modelado físico del sistema. Modelo de Implementación: Componentes. Modelo de despliegue: Nodos. Enlaces.



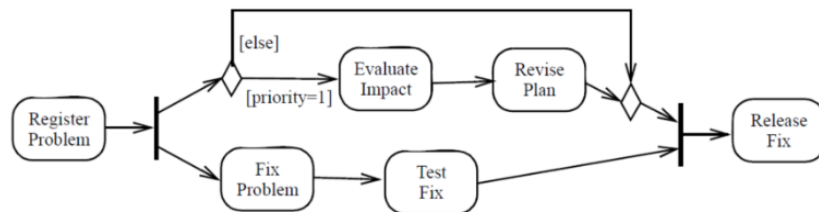
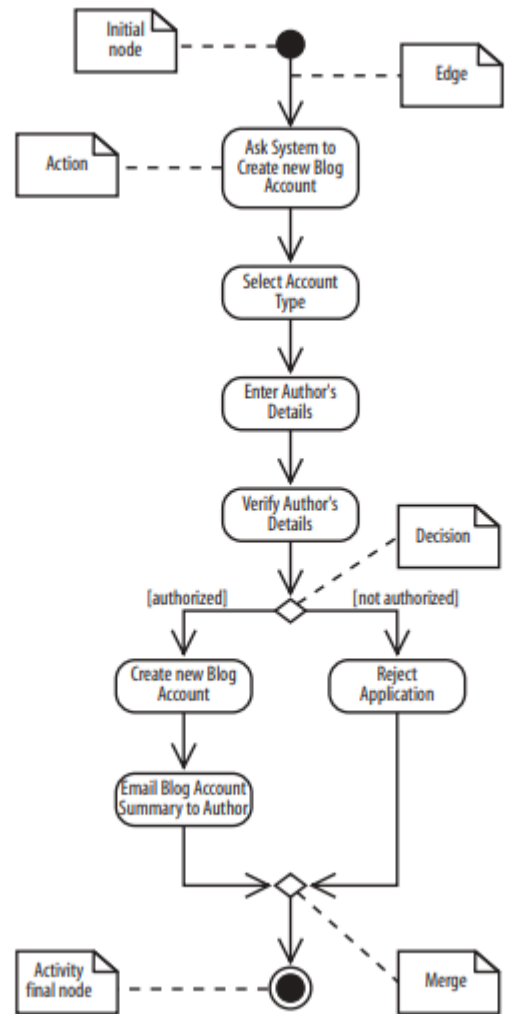
## # Gestión del Modelo: Paquetes. Dependencias. Modelo y Subsistema.

**Repaso con las preguntas del Cuestionario:**

- La flecha que va de un estado a otro se llama flujo de control
- Se llama estado oculto al estado compuesto en el cual no veo los estados internos



1. Serving customer es un estado compuesto
2. Transaction es un estado compuesto
3. Serving customer es un estado de submáquina
4. readCard se ejecuta sólo la primera que se activa el estado
5. turn off podría interpretarse como una señal externa
6. service es el nombre de la transición
7. La transición entre Idle y Self Test se efectuará únicamente cuando Self Test culmine exitosamente

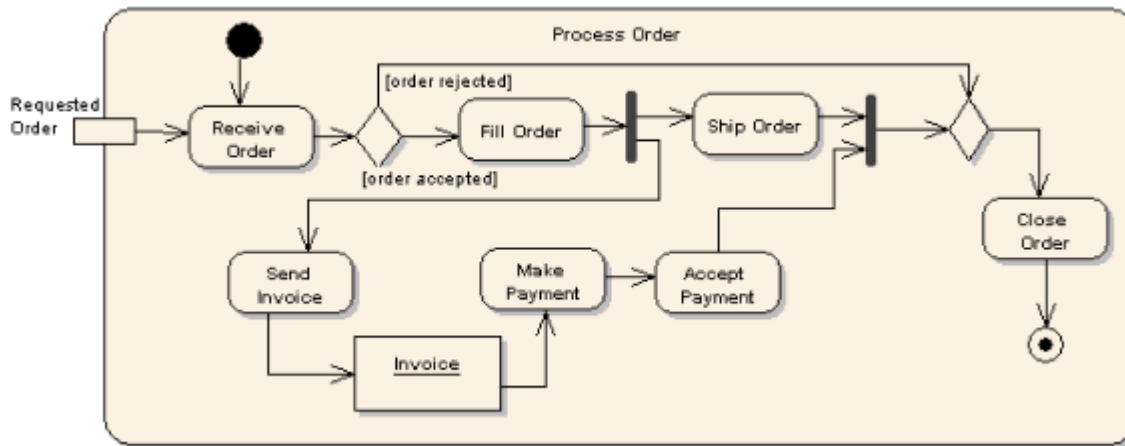


1. Un nodo del tipo merge (rombo) puede recibir varios flujos al mismo tiempo.
2. Las flechas existentes entre nodos de actividad son denominadas flujo de actividad.
3. La barra vertical anterior a Release Fix indica una sincronización de todos los flujos, por lo que se deben terminar todas las actividades previas antes de que se ejecute Release Fix

## Vista de Actividades

- Variante de la máquina de estados para modelar flujos de trabajo
- Utiliza diagramas de actividad, que es, esencialmente, un diagrama de flujo
- Es un caso particular de los diagramas de estado
- Los nodos representan estados de actividad, no de un objeto
- Contienen
  - o Acciones
  - o Nodos de actividad
  - o Flujos de control
  - o Valores de objetos

no sé de qué clase es esto 😞



## Área Física

### Vista de Despliegue

#### Diagrama de despliegue

- ¿Cuándo un nodo puede ser invisible? Cuando tengo una representación virtual
- El componente no necesariamente es físico
- El artefacto implementa un componente a nivel de diseño
- El artefacto es un archivo físico, que podemos ver
- Hay artefactos definidos en UML en el apartado de perfiles que amplían la especificación de estos, se los puede omitir si tenemos nombres que sean representativos a la actividad que se quiere lograr con ese artefacto, es decir el propósito del mismo 9-1, 9-2, 9-3-\*

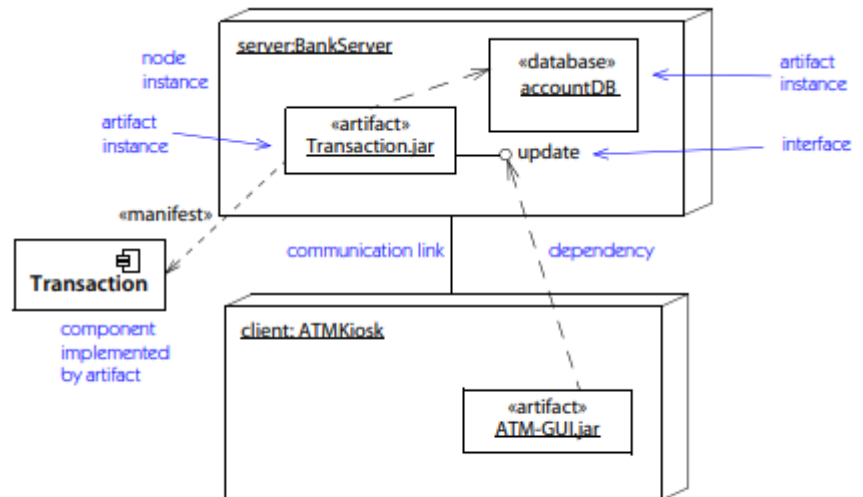


Figure 10-1. Deployment diagram

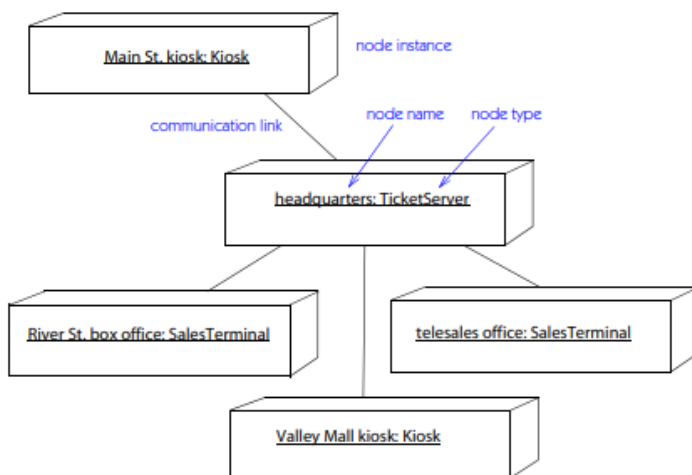


Figure 3-12. Deployment diagram (instance level)

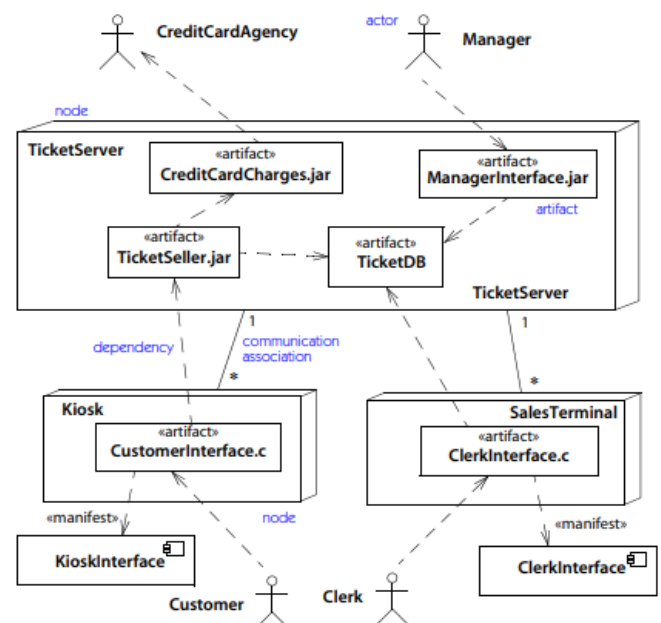


Figure 3-11. Deployment diagram (descriptor level)

# Área Gestión de Modelo

## Vista de Gestión de Modelo

### Dependencias de importación

El paquete da un contexto a la clase

Nombre totalmente calificado, cuando digo que clase y de donde lo estamos trayendo, vamos a ir teniendo el :: por cada paquete al que ingrese

Con <<import>> no importo la referencia si no que voy a traer ese espacio de nombre dentro de mi carpeta y cuando quiera utilizarlo no voy a necesitar llamar a las clases como si fueran externas, sino que las voy a usar como si fueran propias de la clase que realiza el import

La herencia del paquete representa clases abstractas, es decir, la herencia de las clases

Modelo y subsistema

Cuándo modelo los paquetes no modelo en sí diferentes funcionalidades, si no diferente nivel de abstracción. Generalmente habla de una vista particular

Cuando hablamos de subsistema podemos hablar de que tienen diferentes funcionalidades

C#, o Java pero adaptarlo a Py

{completar} con los diagramas y explicaciones

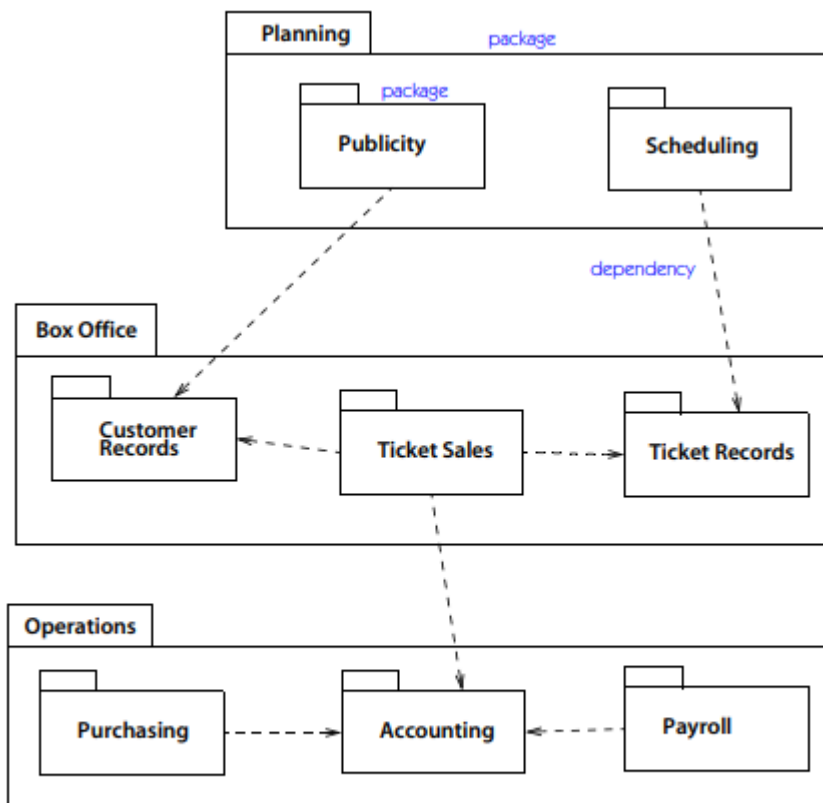
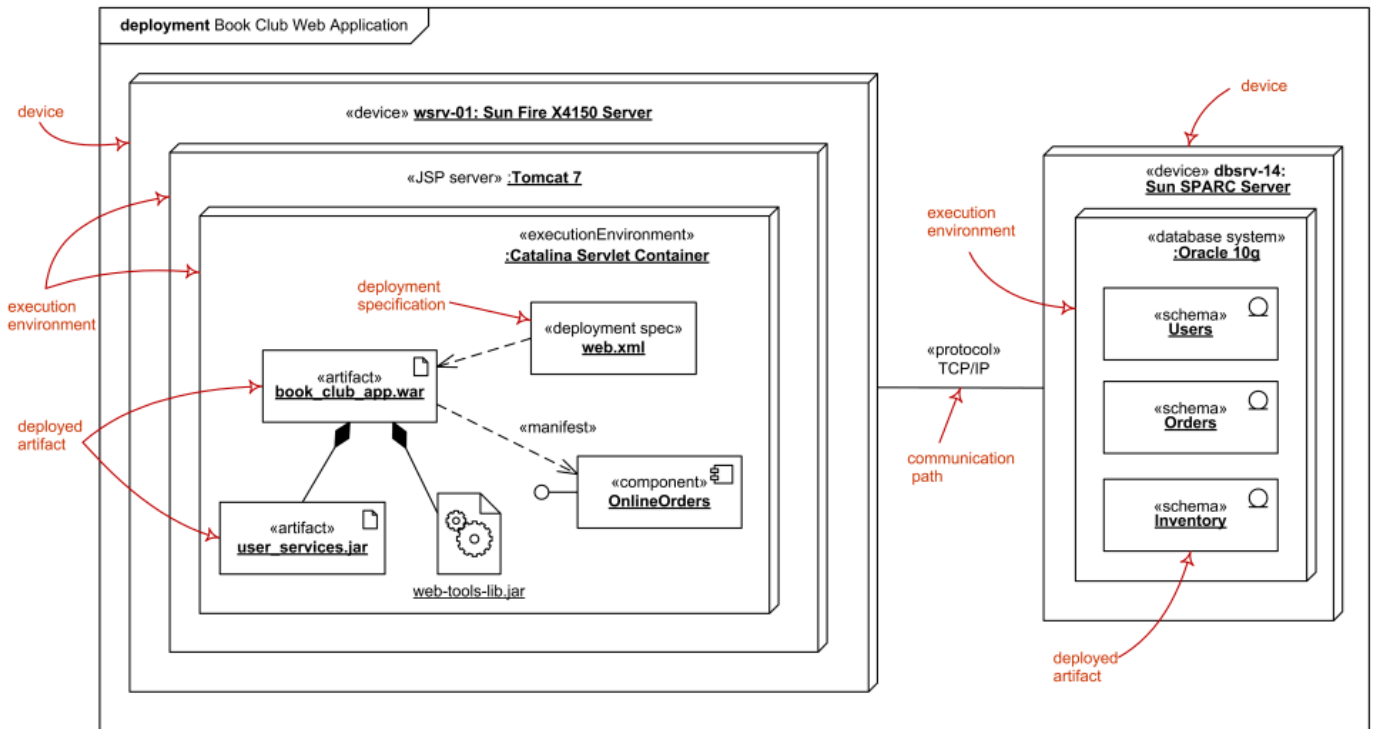


Figure 3-13. Package diagram

Cuestionario de clase (close 12/6)

Pregunta 1:

Pregunta 2:



Pregunta 3:

**Respuesta:**

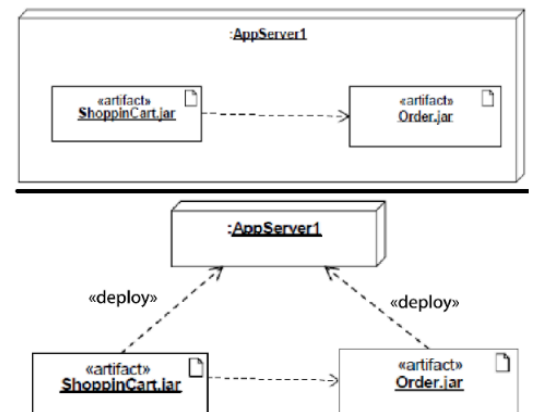
De las opciones las correctas son

**a. AppServer1 es un nodo.**

**b. Ambos diagramas son equivalentes.**

La incorrecta es

**c. ShoppinCart.jar manifiesta Order.jar.**



## PRINCIPIOS SOLID

### SOLID

**SOLID** stands for five key principles in object-oriented design that promote well-structured, maintainable, and flexible code (y escalable).

Se podría resumir diciendo que son cinco principios que hay que tener siempre presentes si queremos desarrollar un software de calidad, legible, entendible y fácilmente testeable. **¿Por qué es fácilmente testeable?** Principalmente porque nos va a obligar a sacar muchas interfaces, cosa que favorecerá la utilización de Mocking, y hará que el trabajo de testear una clase sea conciso, además favorece el TDD porque se prepara el código para los sucesivos cambios propios del TDD

#### SRP - Single Responsibility Principle:

- A class should have one, and only one, reason to change.
- This means a class should focus on a single functionality or responsibility.
- Having multiple unrelated functionalities within a single class makes it harder to understand, maintain, and modify.

- Se serializa para separar responsabilidades, y de esta forma sea más fácil programar, de cierta forma se modulariza más la implementación

### OCP - Open-Closed Principle:

- Software entities (classes, modules, functions) should be open for extension, but closed for modification.
- This means we should be able to extend the functionality of a system without changing existing code. This is achieved through techniques like inheritance and interfaces.

### LSP - Liskov Substitution Principle:

- Objects of a superclass should be replaceable with objects of its subclasses without altering the program's correctness.
- In simpler terms, if a program expects an object of a certain type (superclass), it should work correctly when given an object of a subclass, as long as the subclass adheres to the behavior of the superclass.

### ISP - Interface Segregation Principle:

- Clients shouldn't be forced to depend on methods they don't use.
- This principle suggests creating smaller, more specific interfaces instead of one large interface containing functionalities not used by all clients.

### DIP - Dependency Inversion Principle:

- High-level modules shouldn't depend on low-level modules. Both should depend on abstractions. Abstractions shouldn't depend on details. Details should depend on abstractions.
- This principle encourages writing code that relies on abstractions (interfaces) rather than concrete implementations. This allows for easier switching of implementations without modifying the dependent code.

By following these SOLID principles, developers can create more robust, adaptable, and easier-to-maintain software systems.

12/06

## GRASP

## General Responsibility Assignment Software Patterns

**GRASP** stands for **General Responsibility Assignment Software Patterns**. It's a collection of guidelines that help developers assign responsibilities to classes and objects in object-oriented design. Here's a breakdown of some core GRASP principles:



### Creador - Creator

**This principle focuses on assigning the responsibility for creating objects of a class. It can be done by the class itself (factory method) or a dedicated creator class.**

El patrón creador nos ayuda a identificar quién debe ser el responsable de la creación o instanciación de nuevos objetos o clases. Éste patrón nos dice que la nueva instancia podrá ser creada por una clase si:

- Contiene o agrega la clase.
- Almacena la instancia en algún sitio (por ejemplo una base de datos)
- Tiene la información necesaria para realizar la creación del objeto (es 'Experta')
- Usa directamente las instancias creadas del objeto

Una de las consecuencias de usar este patrón es la visibilidad entre la clase creada y la clase creadora. Una ventaja es el bajo acoplamiento, lo cual supone facilidad de mantenimiento y reutilización.



### Experto en información - Information Expert:

**This principle emphasizes assigning responsibilities to the class that has the most information required to fulfill them. This ensures efficient data access and reduces redundancy.**

Éste es el principio básico de asignación de responsabilidades, la S de SOLID. Experto en información nos dice que la responsabilidad de la creación de un objeto o la implementación de un método, debe recaer sobre la clase que conoce toda la información necesaria para crearlo o ejecutarlo, de este modo obtendremos un diseño con mayor cohesión y

cuya información se mantiene encapsulada, es decir, disminuye el acoplamiento. Éste es el principio que se debería aplicar más a menudo, el que hay que tener más en cuenta. La suma de "Experto en información" y SRP es que una clase sólo debería tener un motivo para cambiar y debería ser ella misma la encargada de crear los objetos e implementar los métodos sobre los que es experta.

## Cohesión y acoplamiento

### Bajo acoplamiento - Low Coupling

**This principle aims for minimal dependencies between classes. Loosely coupled classes are easier to understand, maintain, and reuse independently.**

El grado de acoplamiento indica lo vinculadas que están unas clases con otras, es decir, la medida en la que afecta un cambio en una clase a las demás y por tanto lo **dependientes** que son unas clases de otras.

La idea es tener las clases lo menos ligadas entre sí que se pueda, de tal forma que, en caso de producirse una modificación en alguna de ellas, tenga la mínima repercusión posible en el resto de clases, potenciando la reutilización, y disminuyendo la dependencia entre las clases. También hay varios tipos de acoplamiento.

1. Acoplamiento de contenido: cuando un módulo referencia directamente el contenido de otro módulo. (hoy en día es difícil verlo, quizá es más fácil verlo en entornos de programación funcional)
2. Acoplamiento común: cuando dos módulos acceden (y afectan) a un mismo valor global.
3. Acoplamiento de control: cuando un módulo le envía a otro un elemento de control que determina la lógica de ejecución del mismo.

¿cómo reducimos el impacto del cambio? Asignando responsabilidades para que el acoplamiento (innecesario) permanezca bajo. Utilizo este principio para evaluar alternativas

### Alta cohesión - High Cohesion

**This principle promotes designing classes with a clear and focused purpose. A highly cohesive class has responsibilities that are well-related and contribute to a single functionality.**

El grado de cohesión mide la coherencia de una clase, esto es, lo coherente que es la información que almacena una clase con las responsabilidades y relaciones que ésta tiene con otras clases.

Nos dice que la información que almacena una clase debe de ser coherente y debe estar, en la medida de lo posible, relacionada con la clase.

La idea es que tenga bajas responsabilidades y clases **bien enfocadas**, y por eso se relaciona con el principio de responsabilidad única de SOLID. Los puritanos y teóricos diferencian 7 tipos de cohesión:

1. Cohesión coincidente: el módulo realiza múltiples tareas pero sin ninguna relación entre ellas.
2. Cohesión lógica: el módulo realiza múltiples tareas relacionadas pero en tiempo de ejecución sólo una de ellas será llevada a cabo.
3. Cohesión temporal: las tareas llevadas a cabo por un módulo tienen, como única relación el deber ser ejecutadas al mismo tiempo.
4. Cohesión de procedimiento: la única relación que guardan las tareas de un módulo es que corresponden a una secuencia de pasos propia del producto.
5. Cohesión de comunicación: las tareas corresponden a una secuencia de pasos propia del producto y todas afectan a los mismos datos.
6. Cohesión de información: las tareas llevadas a cabo por un módulo tienen su propio punto de arranque, su codificación independiente y trabajan sobre los mismos datos. El ejemplo típico: OBJETOS
7. Cohesión funcional: cuando el módulo ejecuta una y sólo una tarea, teniendo un único objetivo a cumplir.

**Responde al problema** de: ¿cómo mantenemos las clases enfocadas, comprensibles y manejables y, como efecto secundario, lograr un acoplamiento bajo?

Como se ve claramente, los conceptos de **cohesión** y **acoplamiento** están íntimamente relacionados. Un mayor grado de cohesión implica uno menor de acoplamiento. Maximizar el nivel de cohesión intramodular en todo el sistema resulta en una minimización del acoplamiento intermodular.



### Controlador - Controller

**This principle assigns the responsibility of handling system events or coordinating activities to a controller class. This centralizes control flow and simplifies interactions between other objects.**

El patrón controlador es un patrón que sirve como intermediario entre una determinada interfaz y el algoritmo que la implementa, de tal forma que es el controlador quien recibe los datos del usuario y quien los envía a las distintas clases según el método llamado. Este patrón sugiere que la lógica de negocio debe estar separada de la capa de presentación, lo que aumenta la reutilización de código y permite a la vez tener un mayor control.

La pregunta a realizarse para este patrón es: ¿cuál es el primer objeto más allá de la capa de UI en recibir y coordinar ("controlar") una operación del sistema?

**Un Controlador:**

- Representa el sistema general, un objeto raíz, un dispositivo en el que se ejecuta el SW
- Representa un escenario de caso de uso en el que se produce una operación del sistema
- Debería ser el primer objeto llamado después de un cambio en la interfaz de usuario.

- Controla/ejecuta un caso de uso. No hace demasiado por si solo, controla, coordina.
- Pertenece a la capa de aplicación o a la de servicios

Se recomienda dividir los eventos del sistema en el mayor número de controladores para poder **aumentar así la cohesión y disminuir el acoplamiento**.



## Polimorfismo - Polymorphism

**This principle encourages leveraging polymorphism (ability to treat objects of different classes in a similar way) to achieve flexible and adaptable code. It can be implemented through inheritance and interfaces.**

**Problema:** ¿cómo manejar las alternativas en función del tipo? ¿cómo crear componentes de software conectables?

**Solución:** cuando las alternativas o los comportamientos relacionados varían según el tipo (clase), asigne la responsabilidad del comportamiento mediante operaciones polimorfismo a los tipos para los que varía el comportamiento

**Corolario:** no pruebe el tipo de un objeto y sue la lógica condicional para realizar diversas alternativas según el tipo. Este principio fundamental de la orientación a objeto permite que una misma función sea implementada de diferente forma según la clase que la implementa

La idea es evitar el uso de condicionales cuando pueden ser reemplazadas por polimorfismo, redefiniendo el método en las clases que lo necesiten



## Fabricación pura - Pure Fabrication

**This principle allows introducing new classes to fulfill specific responsibilities without violating cohesion or coupling principles. It helps maintain clear class boundaries.**

La fabricación pura se da en las clases que no representan un ente u objeto real del dominio del problema sino que se han creado intencionalmente para disminuir el acoplamiento, aumentar la cohesión y/o potenciar la reutilización del código. La fabricación pura es la solución que surge cuando el diseñador se encuentra con una clase poco cohesiva y que no tiene otra clase en la que implementar algunos métodos. Es decir que se crea una clase "inventada" o que no existe en el problema como tal, pero que, añadiéndola, logra mejorar estructuralmente el sistema. Como contraindicación deberemos mencionar que al abusar de este patrón suelen aparecer clases función o algoritmo, esto es, clases que tienen un solo método.

Entonces ahora podemos decir que las arquitecturas de "n capas" se basan en el patrón de "fabricación pura" ya que cada capa le aporta abstracción al proyecto

Su **objetivo** es la alta cohesión



## Indirección - Indirection

**This principle suggests introducing an intermediate layer of indirection (like an interface or abstract class) between closely coupled classes. This allows for easier changes and decoupling.**

El patrón de indirección nos permite mejorar el bajo acoplamiento entre dos clases asignando la responsabilidad de la mediación entre ellos a una clase intermedia.

**Responde al problema:** ¿dónde debo asignar responsabilidades para evitar o reducir el acoplamiento directo entre elementos y mejorar la reutilización? Es decir, si sabemos que un objeto utiliza otro que muy posiblemente va a cambiar, ¿cómo protegemos a un objeto frente a cambios previsibles? ¿cómo desacoplar objetos para que se admita un acoplamiento bajo y el potencial de reutilización siga siendo mayor?

**Solución:** asignar la responsabilidad a un objeto que medie entre los elementos para proteger al primer objeto de los posibles cambios del segundo.

Esto se hace con el propósito de separar responsabilidades. La idea es que no se acoplen directamente, y que los cambios exteriores no incidan dentro del sistema.

Este patrón es fundamental para crear abstracciones ya que nos permite introducir API's externas en la aplicación sin que tengan demasiada influencia en el código que tendría que cambiar cuando cambia la API. Además, la primera clase podría cambiarse fácilmente para que en vez de ésta API utilizara cualquier otra.

Su **objetivo** es el bajo acoplamiento

## Variaciones protegidas. - Protected Variations

**This principle addresses situations where subclasses might need to modify inherited behavior. It suggests using techniques like abstract methods or hooks to allow for controlled variations within a defined framework.**

Variaciones protegidas, es el principio fundamental de protegerse frente al cambio. Esto quiere decir que lo que veamos en un análisis previo que es susceptible de modificaciones lo envolvemos en una interfaz y utilicemos el polimorfismo para crear varias implementaciones y posibilitar implementaciones futuras de manera que quede lo menos ligado posible a nuestro sistema. De ésta forma, cuando se produzca la variación o el cambio que esperamos, dicho cambio nos repercuta lo mínimo. Este principio está muy relacionado con el polimorfismo y la indirección.



## Beneficios:

- **Flexibilidad:** Se pueden agregar nuevos tipos de jugadores sin modificar el código principal.
- **Reutilización:** La lógica básica de las acciones se define en la clase abstracta `Jugador` y se reutiliza en las clases derivadas.
- **Modularidad:** El código está organizado en clases con responsabilidades bien definidas, lo que facilita su comprensión y mantenimiento.
- **Legibilidad:** El código es más claro y fácil de entender al evitar la duplicación de código y el uso de lógica condicional para determinar el tipo de jugador.

## Ejemplo

¡Claro que sí! Imaginemos un juego de Monopoly con "variaciones protegidas":

El tablero:

- Casillas de propiedades: Cada casilla representa una propiedad con características únicas, como su valor, tipo (residencial, comercial, etc.) y alquiler.
- Casillas de acción: Estas casillas desencadenan eventos que afectan a los jugadores, como ir a la cárcel, pagar impuestos o recibir dinero.

Los jugadores:

- Jugador estándar: Un jugador típico que compra propiedades, construye casas y hoteles, y cobra alquiler a otros jugadores.
- Jugador inversor: Un jugador que se enfoca en comprar propiedades estratégicas y negociar acuerdos con otros jugadores para obtener beneficios.
- Jugador constructor: Un jugador que se centra en construir casas y hoteles para aumentar el valor de sus propiedades y maximizar el alquiler.

Las acciones:

- Comprar una propiedad: El jugador paga el precio de la propiedad al banco y pasa a ser su dueño.
- Construir casas y hoteles: El jugador paga una cantidad de dinero para construir casas o hoteles en sus propiedades, lo que aumenta el alquiler.
- Cobrar alquiler: Cuando un jugador cae en una propiedad de otro jugador, debe pagar el alquiler correspondiente.

Implementación con "variaciones protegidas":

- Clase abstracta `Jugador`: Define las acciones básicas que puede realizar cualquier jugador, como comprar propiedades, construir y cobrar alquiler.
- Clases derivadas `JugadorEstandar`, `JugadorInversor` y `JugadorConstructor`: Heredan de la clase `Jugador` y sobrescriben el método `cobrarAlquiler()` para implementar la lógica específica de cada tipo de jugador.

Ejemplo:

- `JugadorEstandar.cobrarAlquiler()`: Calcula el alquiler base de la propiedad y lo cobra al jugador que cayó en ella.
- `JugadorInversor.cobrarAlquiler()`: Además del alquiler base, el jugador inversor puede aplicar un recargo adicional a la renta según la estrategia de inversión.
- `JugadorConstructor.cobrarAlquiler()`: El jugador constructor aumenta el alquiler base en función del número de casas y hoteles que haya construido en la propiedad..

En resumen:

Las "variaciones protegidas" permiten organizar el código del juego de Monopoly de manera eficiente y modular, aprovechando el polimorfismo para implementar comportamientos diferentes para cada tipo de jugador. Esto hace que el código sea más flexible, reutilizable, legible y fácil de mantener.

¡Espero que este ejemplo te ayude a comprender mejor el concepto de "variaciones protegidas" en el contexto de la programación orientada a objetos!

## Conclusiones

These are some of the key GRASP principles. Following them helps developers create well-structured, maintainable, and reusable object-oriented software.

## Diferencias

**Indirección:** Útil para reducir el acoplamiento entre clases.



**Fabricación Pura:** Para mejorar la cohesión de las clases existentes.

**Patrón de Controlador:** Para centralizar la lógica de negocio y facilitar la separación de preocupaciones.

**Variaciones Protegidas:** Para implementar comportamientos diferentes en clases derivadas.

Característica	Indirección	Fabricación Pura	Patrón de Controlador	Variaciones Protegidas	Experto en Información	Creador
Objetivo	Acoplamiento	Cohesión	Lógica de negocio	Polimorfismo	Cálculos/Información	Creación de instancias
Beneficios	Flexibilidad, testabilidad, reusabilidad	Cohesión, legibilidad, mantenibilidad	Separación de preocupaciones, mantenibilidad, testabilidad	Flexibilidad, reusabilidad, modularidad, legibilidad	Cohesión, mantenibilidad, legibilidad	Encapsulación, modularidad, legibilidad
Aplicación	Dependencias directas	Lógica específica	Coordinación entre clases	Comportamientos diferentes	Cálculos/Acceso a información	Creación frecuente
Ejemplo en Monopoly	Casilla, Tablero	CalculadorAlquiler, Jugador	Juego	JugadorEstandar, JugadorInversor, JugadorConstructor	Casilla	Tablero

Ejemplo de Relaciones de artefactos en el proceso unificado

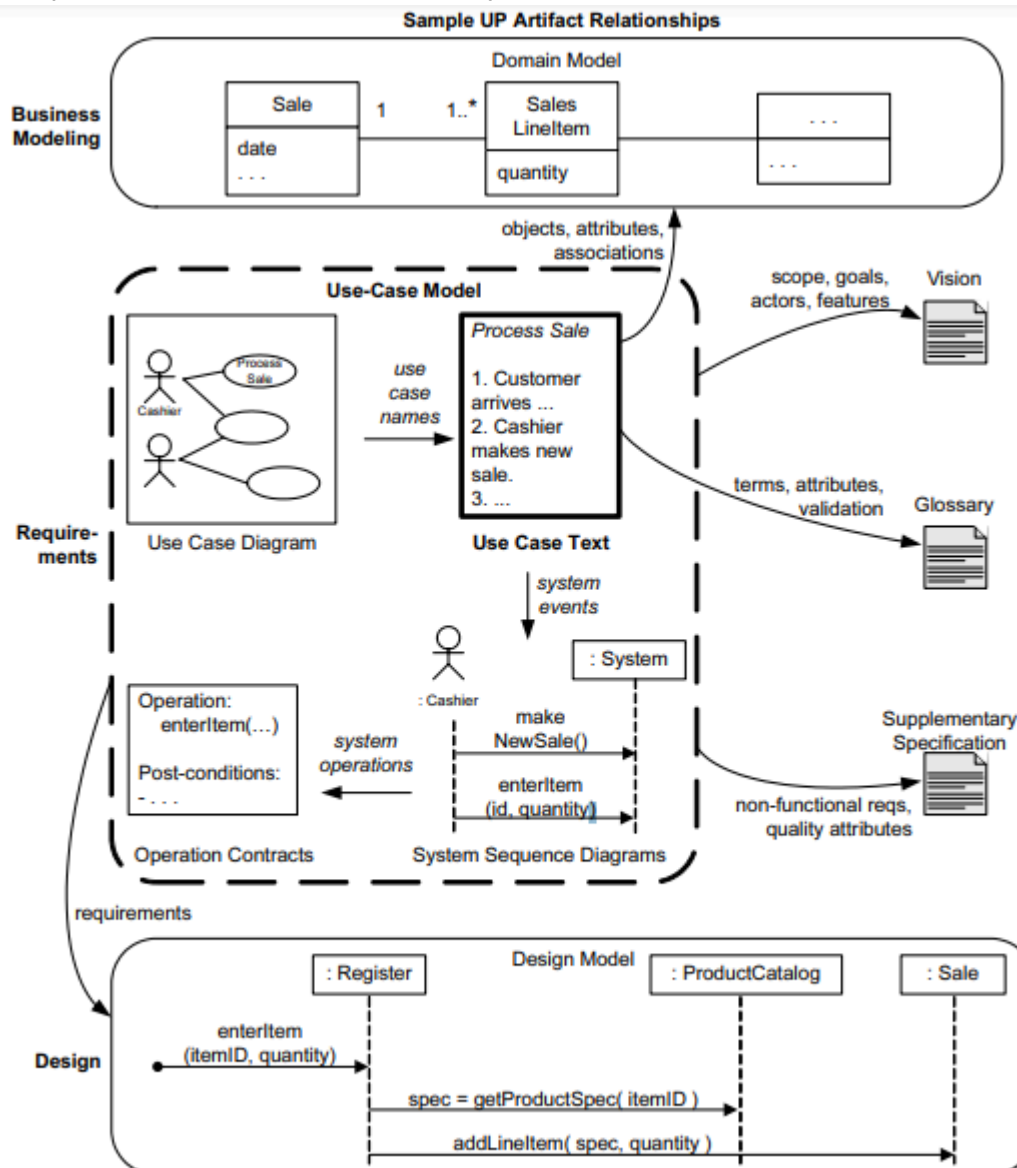


Figure 6.1 Sample UP artifact influence.

