

Sesión 1

Unidad 1

Introducción al Análisis de Sistemas

Mg. Gustavo G. Delgado Ugarte

INTRODUCCIÓN A LA INGENIERÍA DE SOFTWARE

Ingeniería del Software

- La ingeniería del software es una disciplina de la ingeniería que comprende todos los aspectos de la producción de software desde las etapas iniciales de la especificación del sistema, hasta el mantenimiento de éste después de que se utiliza.

Ingeniería del Software

- Disciplina de la Ingeniería
 - Los ingenieros
 - Aplican teorías, métodos y herramientas
 - Trabajan con restricciones financieras y organizacionales
- Aspectos de la producción de Software
 - No sólo comprende los procesos técnicos, sino también
 - Gestión de proyectos de software
 - Desarrollo de herramientas
 - Métodos y teorías de apoyo a la producción de software

Proceso del Software

- Conjunto de actividades y resultados asociados que producen un producto de software
- Todos los procesos de software tienen 4 actividades comunes
 - Especificación del software
 - Desarrollo del software
 - Validación del software
 - Evolución del software

Proceso del Software

- **Especificación.**- Se define el software a producir y las restricciones sobre su operación
- **Desarrollo.**- Diseño y programación del software
- **Validación.**- se valida el software para asegurar que es lo que el cliente requiere
- **Evolución.**- el software se modifica para adaptarlo a los cambios requeridos por el cliente y el mercado

Modelo de Proceso

- Descripción simplificada de un proceso del software que presenta una visión de ese proceso
- La mayor parte de los modelos de procesos del software se basan en uno los siguientes modelos generales o paradigmas de desarrollo
 - Enfoque de la cascada
 - Desarrollo iterativo
 - Ingeniería del software basada en componentes (CBSE)

Modelo de Proceso

- Enfoque de la cascada
 - Las actividades se representan como fases
 - Especificación de requerimientos
 - Diseño del software
 - Implementación
 - Pruebas, etc.
 - Después que cada etapa queda definida “se firma” y el desarrollo continúa con la siguiente etapa

Modelo de Proceso

- Desarrollo iterativo
 - Entrelaza las actividades de especificación, desarrollo y validación
 - El sistema inicial se desarrolla rápidamente a partir de especificaciones muy abstractas
 - El sistema se refina en base a peticiones del cliente para producir un sistema que satisfaga sus necesidades
 - Luego, el sistema es entregado

Modelo de Proceso

- Ingeniería del software basada en componentes
 - La técnica supone que las partes del sistema existen
 - El proceso de desarrollo del sistema se enfoca en integración de estas partes

Métodos de Ingeniería del Software

- Enfoque estructurado para el desarrollo de software cuyo propósito es facilitar la producción de software de alta calidad de una forma costeable.
- Años 70
 - Métodos orientados a funciones
 - Análisis Estructurado
 - JSD

Métodos de Ingeniería del Software

- Años 80 y 90
 - Métodos orientados a objetos
 - Propuestas de Booch y Rumbaugh
 - UML (Lenguaje de Modelado Unificado).- enfoque que integra los otros enfoques.

INTRODUCCIÓN A LA INGENIERÍA DE REQUERIMIENTOS

Introducción

- El desarrollo de sistemas basados en computador ha estado plagado de problemas desde la década de 1960 (Sommerville, 1997)
 - Sistemas entregados fuera de tiempo
 - Sistemas fuera del presupuesto
 - Sistemas que no hacen lo que los usuarios quieren
 - Sistemas que nunca son usados completamente

Introducción

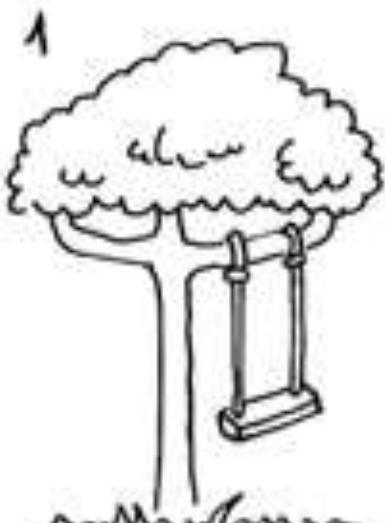
- Problemas comunes relacionados con los requerimientos de sistemas son (Sommerville,1997)
 - Los requerimientos no reflejan las necesidades reales de los clientes
 - Los requisitos son inconsistentes y/o incompletos.
 - Es muy caro hacer cambios a los requerimientos una vez que han sido acordados.
 - Existen malos entendidos entre los clientes, quienes desarrollan los requerimientos del sistema y los ingenieros de software que desarrollan o mantienen el sistema.

Introducción

- Lo más difícil en la construcción de un sistema de software es decidir precisamente qué construir... No existe tarea con mayor capacidad de lesionar al sistema, cuando se hace mal... Ninguna otra tarea es tan difícil de rectificar a posteriori (F.P. Brooks, 1987)

Introducción

- Boehm, 1975: 45% de los errores tienen su origen en los requisitos y en el diseño preliminar.
- DeMarco, 1984: 56% de los errores que tienen lugar en un proyecto de software, se deben a una mala especificación de requisitos.
- Chaos Report, 1995: Los factores principales que conducen al fracaso en los proyectos de software son:
 - Falta de comunicación con los usuarios
 - Requisitos incompletos
 - Cambios a los requisitos



LO QUE PIDE EL
CLIENTE



LO QUE QUIERE OFRECER
EL DPTO. COMERCIAL



LO QUE EXIGE
GARANTÍA DE CALIDAD



LO QUE DISEÑA
PROYECTOS



LO QUE IMPATI
PROBULLICA



LA DISPOSICIÓN DEL
DPTO. DE POSVENTA



LA ENTREGA FINAL
AL CLIENTE



EL INFORME DEL
Jefe de Programa

Ingeniería de Requerimientos

- El proceso de establecer los servicios que el cliente requiere de un sistema y las restricciones bajo las cuales opera y será desarrollado.
- Los requerimientos son descripciones de los servicios y restricciones generadas durante el proceso de ingeniería de requerimientos.

¿Qué es un requerimiento?

- Puede ser un conjunto de descripciones abstractas de alto nivel de un servicio o una restricción del sistema.
- Es una especificación formal matemáticamente detallada de una función del sistema.
- Por lo tanto, un requerimiento puede tener una función dual
 - Puede ser la base de un contrato, por lo tanto, debe tener una interpretación abierta a distintas propuestas de solución.
 - Puede ser el contrato por si solos, por lo tanto deben ser definidos en detalle.
 - Ambos se denominan documentos de requerimientos.

Tipos de Requerimientos

- Requerimientos de Usuario
 - Son declarados en lenguaje natural apoyado por diagramas de los servicios que el sistema proveerá y sus restricciones operacionales. Escrito para clientes.
- Requerimientos del Sistema
 - Se establece en detalle y con precisión los servicios y restricciones del sistemas. Se denomina también especificación funcional. Sirve como contrato entre cliente y desarrollador.
- Especificación de Software
 - Descripción abstracta del diseño del software base que implementa un requerimiento. Escrito para desarrolladores.

Requerimientos Funcionales y no Funcionales

- Requerimientos Funcionales
 - Declaraciones de los servicios del sistema, la manera como reaccionará a entradas límites o particulares. En algunos casos, declaran lo que el sistema no hará.
- Requerimientos No Funcionales
 - Son restricciones de los servicios o funciones ofrecidas por el sistema (restricciones de tiempos, restricciones sobre procesos de desarrollo, uso de estándares, etc.)
- Requerimientos del Dominio
 - Son requerimientos que provienen del dominio de aplicación del sistema y reflejan las características de este dominio.

Requerimientos Funcionales

- Describen la funcionalidad o servicios del sistema.
- Dependen del tipo de software, expectativas del usuario y el tipo de sistema donde el software será usado.
- Los requerimientos funcionales del usuario pueden ser establecidos en alto nivel, en términos de lo que el sistema debe hacer, pero los requerimientos de funcionalidad del sistema deben describir los servicios del sistema en detalle.

Ejemplos de requerimientos funcionales

- El usuario podrá buscar en todo la base de datos o en un subconjunto de ella.
- El sistema proveerá un visualizador apropiado para que el usuario lea los documentos almacenados en la base de datos.
- Toda orden se la asignará un identificador único (OrdenID) con el cuál el usuario será capaz de copiarla.

Imprecisiones en los Requerimientos

- Los problemas provienen de requerimientos no precisamente establecidos.
- Requerimientos ambiguos pueden ser interpretados de diferentes maneras por clientes, usuarios y desarrolladores.
- Considere el término “rendimiento óptimo”

Compleitud y Consistencias de Requerimientos

- Los requerimiento tienen que ser tanto completos como consistentes.
- Completos
 - Se debe incluir todos los servicios requeridos por el usuario.
- Consistentes
 - No hay conflictos o contradicciones en los requerimientos definidos.

Requerimientos No Funcionales

- Define propiedades y restricciones del sistema
- Ejemplo: Confiabilidad, tiempo de respuesta, seguridad, uso de herramientas y lenguajes, etc.
- Los requerimientos no funcionales pueden ser más críticos que los funcionales. Si estos no se cumplen, el sistema no se puede usar.

Clasificación de Requerimientos No Funcionales

- Requerimientos del Producto
 - Especifican que el producto entregado debe tener una determinada velocidad de ejecución, confiabilidad, etc.
- Requerimientos Organizacionales
 - Son consecuencia de políticas y procedimientos de la organización, ejemplo: uso de estándares, lenguajes y herramientas, etc.
- Requerimientos Externos
 - Provienen de factores externos al sistema y su proceso de desarrollo, ejemplo: interoperabilidad con otros sistemas, legislación.



Figura 3 Tipos de requerimientos no funcionales

Objetivos y Requerimientos

- Cumplimiento de Metas vs Verificabilidad
- Objetivo
 - Intención general de logro
- Verificabilidad
 - Declaración usando unidades de medida que pueden ser objetivamente verificadas
- Los objetivos son útiles para los desarrolladores ya que ellos expresan las intenciones de los usuarios del sistema.

Ejemplo

- Objetivo del Sistema
 - El sistema debe ser fácil de usar de tal manera que se minimice la ocurrencia de errores
- Requerimiento no funcional verificable
 - Todos los usuarios deben ser capaces de usar el sistema después de 2 horas de entrenamiento. El número promedio de errores de un usuario entrenado no debe ser superior a 2 por día.

Medidas de Requerimientos

Propiedad	Medida
Velocidad	Transacciones procesadas/segundo Tiempo de respuesta Usuario/Evento Tiempo de refresco de pantalla
Tamaño	K Bytes Número de chips RAM
Facilidad de Uso	Tiempo de entrenamiento Número de ayudas
Confiabilidad	Tiempo antes de falla Probabilidad de no disponibilidad Tasa de ocurrencia de fallas Disponibilidad
Robustez	Tiempo de recuperación ante fallas Porcentaje de eventos causantes de fallas Probabilidad de corrupción de datos cuando falla
Portabilidad	Porcentaje de líneas de código dependiente meta Número de sistemas meta

Interacción entre Requerimientos

- Existe la posibilidad de conflictos entre diferentes requerimientos en un sistema complejo
- Sistema de una Nave Espacial
 - Para minimizar el peso, el número de chips separados que se debe usar debe ser el mínimo.
 - Para minimizar el poder de consumo, se debe usar chips de bajo consumo.
 - **Conflicto!!** Usar chips de bajo consumo puede significar tener que usar más chips.
 - **Decisión!!** ¿Cuál es el requerimiento más crítico?

Requerimientos del Dominio

- Son derivados desde el dominio de la aplicación y describen características y atributos del sistema que reflejan los fundamentos del dominio de aplicación
- Pueden ser funcionales o no funcionales, nuevos o restringiendo los existentes o estableciendo cómo hacer cálculos específicos
- Si los requerimientos de dominio no se satisfacen, puede que sea imposible trabajar con el sistema.

Problemas en los Requerimientos del Dominio

- Entendibilidad
 - Los requerimientos son expresados en el lenguaje del dominio de la aplicación
 - Este lenguaje, a menudo no es entendido por los ingenieros de software y desarrolladores del sistema.
- Implicancias (Obviedad)
 - Los especialistas del dominio entienden el área, pero pueden dejar fuera alguna información sobre algún requerimiento porque para ellos resulta obvio.

Requerimientos de Usuario

- Describen los requerimientos funcionales y no funcionales, de tal forma que sean entendibles por los usuarios del sistema que no poseen conocimientos técnicos detallados.
- Los requerimientos de usuario son definidos usando lenguaje natural, representaciones y diagramas intuitivos sencillos.

Sesión 2

Tecnología Orientada a Objetos

Unidad 1

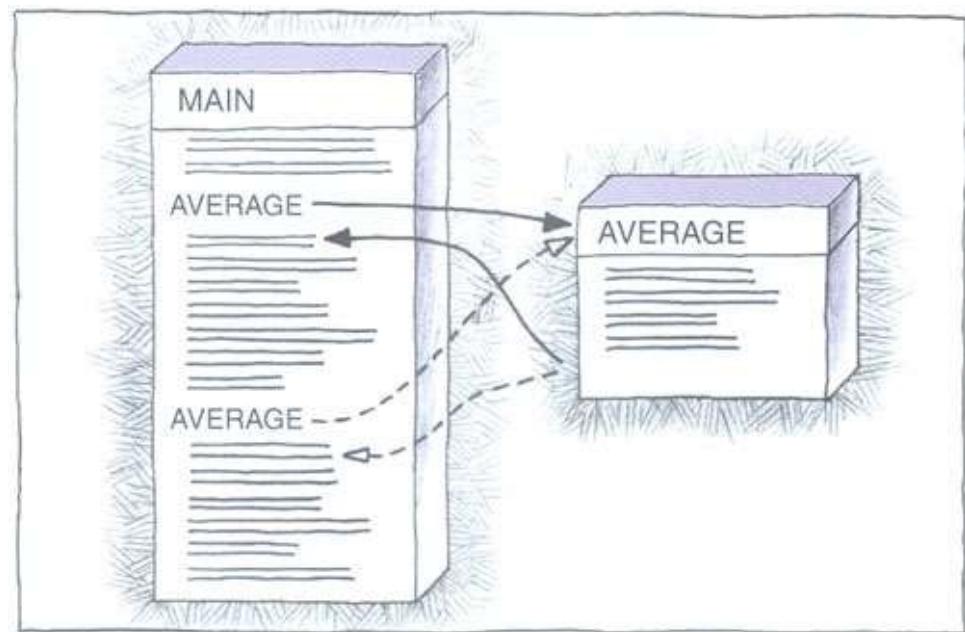
Mg. Gustavo G. Delgado Ugarte

Historia de la tecnología de objetos

- Programación Modular
- Programación Estructurada
- Ingeniería de Software Asistida por Computadora
- Lenguajes de 4ta Generación

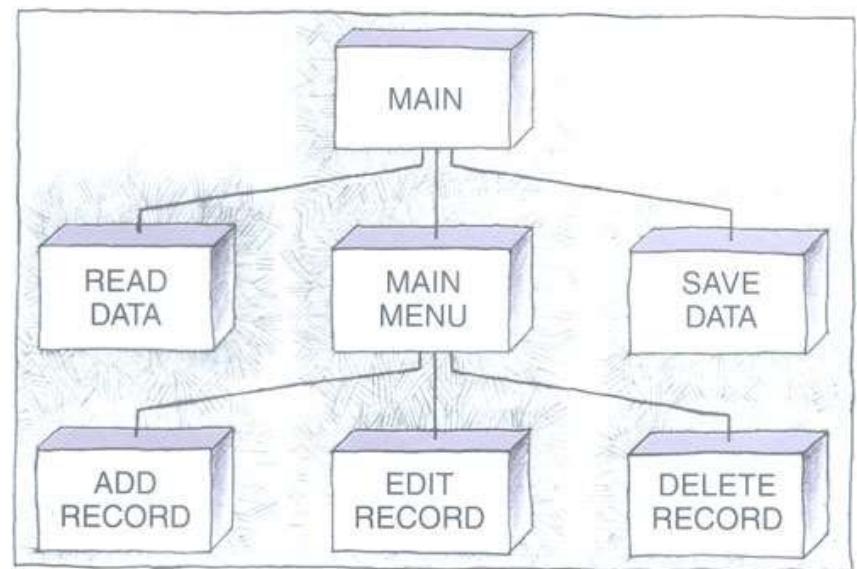
Programación Modular

- Consiste en construir un gran programa, dividiéndolo en pequeñas partes y combinándolas
- El soporte más elemental es la subrutina, que surgió a inicios de los 1950s



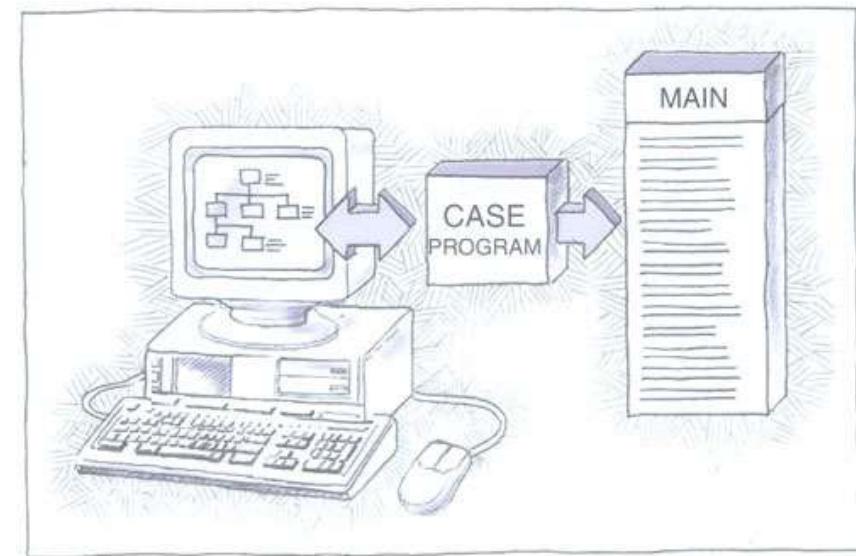
Programación Estructurada

- Surge en los 1960s
- Se basa en la descomposición funcional
- Un programa se divide en componentes, y estos en subcomponentes, y así sucesivamente



Ingeniería de Software Asistida por Computadora

- Principal innovación en la programación estructurada
- Permite a las computadoras manejar el proceso de descomposición funcional, gráficamente define subrutinas en diagramas anidados y verifica que todas las interacciones entre subrutinas sigan correctamente la forma especificada
- Sistemas CASE avanzados generan automáticamente estructuras de programas completos a partir de estos diagramas, una vez que toda la información de diseño se ha ingresado



Lenguajes de Cuarta Generación

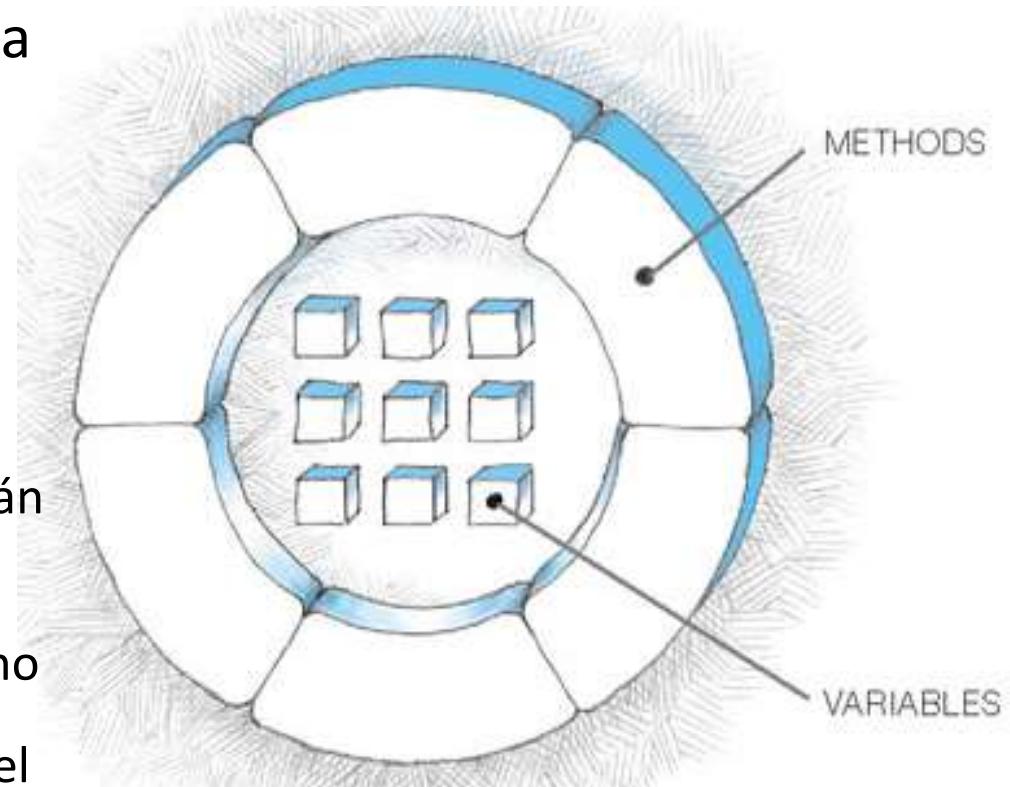
- Otro enfoque de programación automática
- Incluyen una amplia gama de herramientas para ayudar a automatizar la generación de aplicaciones de negocios de rutina, incluyendo la creación de formularios, informes y menús
- **Ventaja:** Personas que no son programadores los pueden utilizar
- **Desventaja:** Sólo puede generar programas muy simples y para problemas bien conocidos

¿Qué es Tecnología de Objetos?

- La definición estándar de la industria de Tecnología Orientada a Objetos, y puede resumirse en términos de tres conceptos clave
 - Los objetos proporcionan encapsulación de los procedimientos y datos
 - Los mensajes soportan el polimorfismo a través de objetos
 - Las clases implementan la herencia dentro de las jerarquías de clase

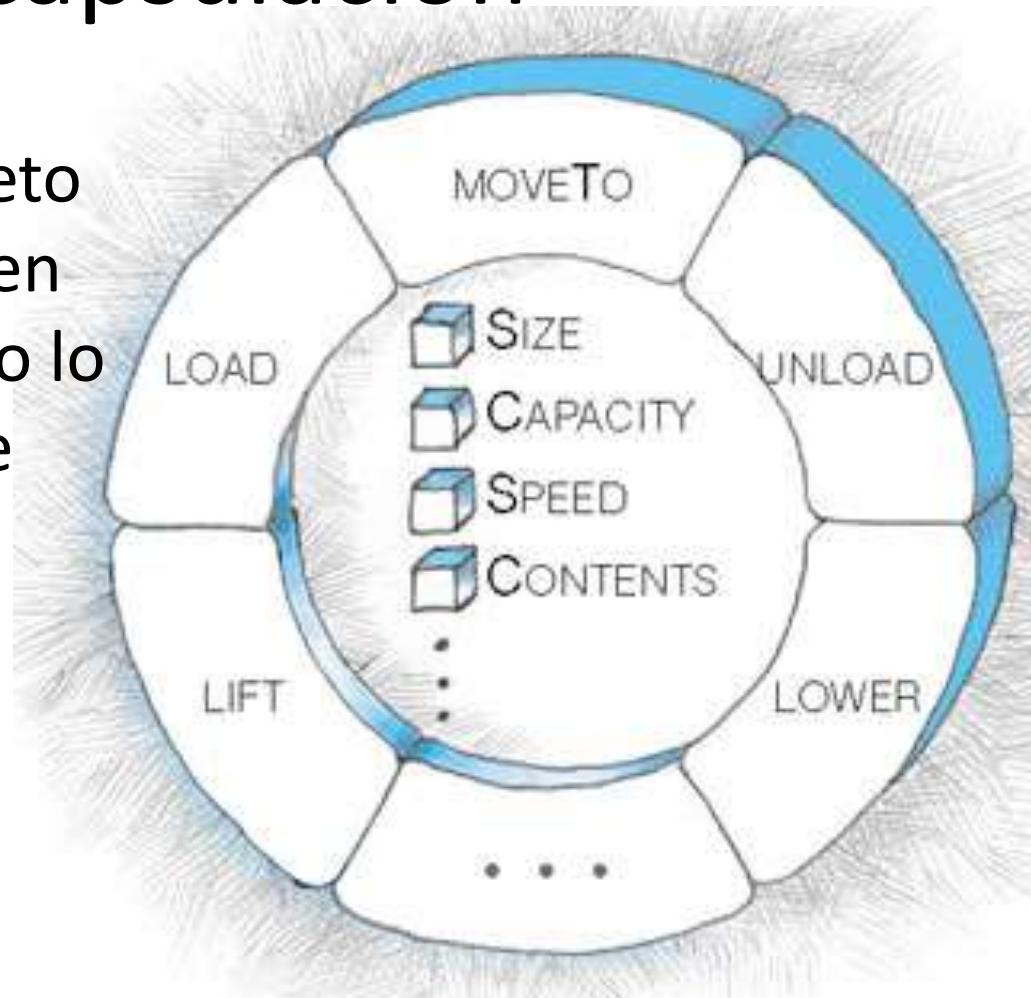
Encapsulación

- Un objeto es un paquete de software que contiene una colección de procedimientos y datos relacionados
 - Los procedimientos son llamados **métodos** para distinguirlos de los procedimientos convencionales que no están unidos a objetos
 - Los elementos de datos se refieren generalmente como **variables** debido a que sus valores pueden variar con el tiempo



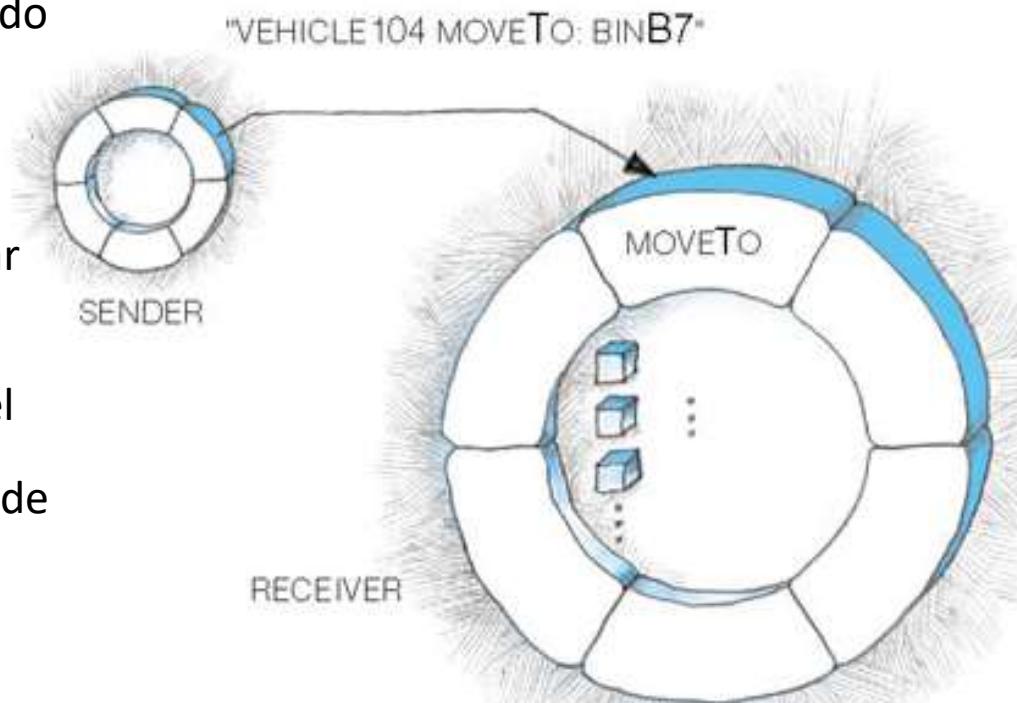
Encapsulación

- Todo lo que un objeto "sabe" se captura en sus variables, y todo lo que puede hacer se expresa en sus métodos.



Mensajes y Polimorfismo

- Los objetos interactúan unos con otros, enviándose mensajes pidiendo a los objetos llevar a cabo sus métodos
- Un **mensaje** es el nombre de un objeto seguido del nombre de un método que el objeto sabe ejecutar
- Si un método requiere cualquier información adicional con el fin de conocer con exactitud qué hacer, el mensaje incluye esa información como una colección de elementos de datos llamados **parámetros**
- El objeto que inicia un mensaje se llama el **remitente**, y el objeto que recibe el mensaje se llama el **receptor**.

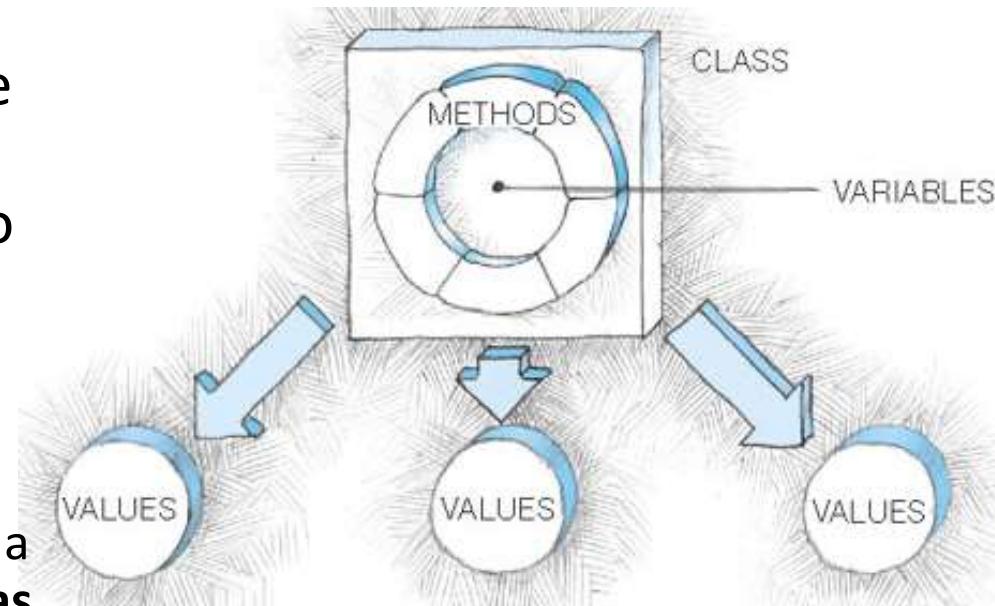


Mensajes y Polimorfismo

- Para que un mensaje tenga sentido, el remitente y el receptor deben ponerse de acuerdo sobre el formato del mensaje. Este formato se establece en **la firma del mensaje** que especifica el nombre del método a ejecutar y los parámetros que se incluirán.
- La capacidad de que diferentes objetos puedan responder al mismo mensaje, de diferentes maneras se llama **polimorfismo**

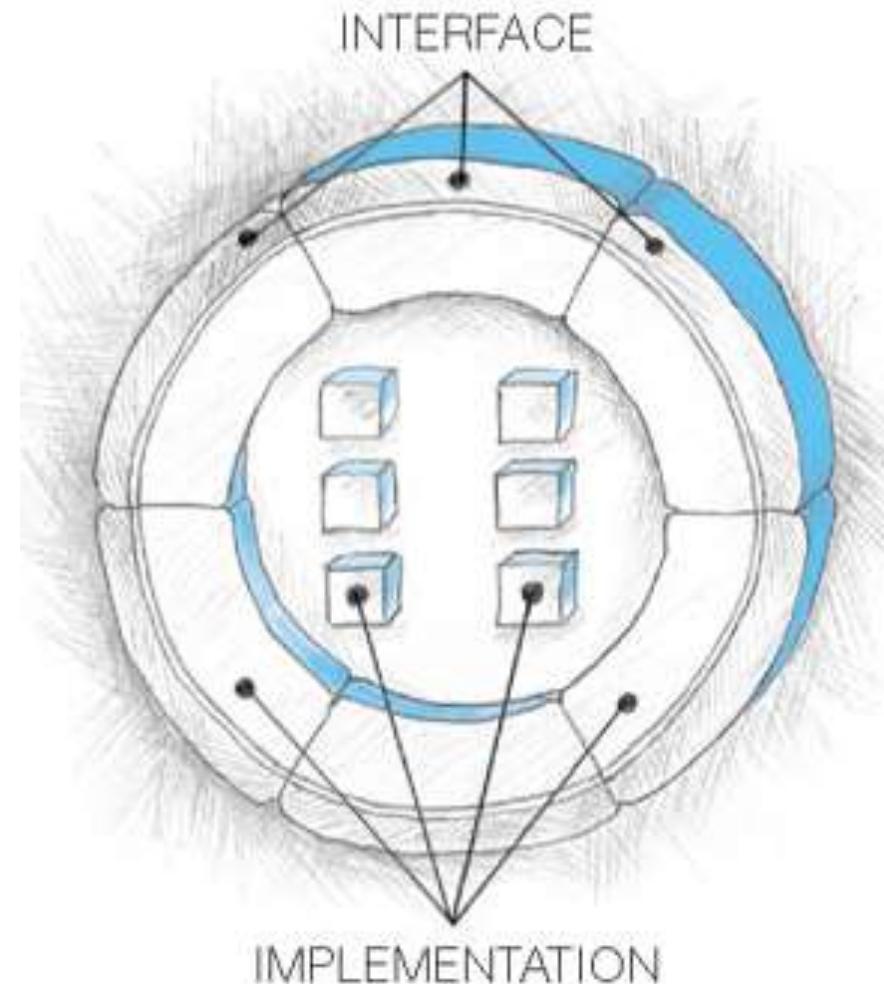
Clases y Herencia

- Una **clase** es una plantilla de software que define los métodos y las variables que se incluirán en un determinado tipo de objeto
 - Los métodos y las variables que componen el objeto se definen una sola vez, en la definición de la clase
 - Los objetos que pertenecen a una clase, llamados **instancias** de esa la clase, contienen sus propios valores particulares para las variables



Clases y Herencia

- El conjunto de mensajes que un objeto se compromete a responder se llama **interfaz del mensaje**.
- Esta interfaz se especifica como una colección de firmas de mensajes, cada uno de los cuales define el nombre y los parámetros para un mensaje concreto.
- El único requisito de diseño es que una clase proporcione un método para implementar cada mensaje especificado en su interfaz.
- Los detalles internos de la clase están totalmente ocultos detrás de esta interfaz y puede incluir cualquier número de variables, así como métodos "invisibles" que son utilizados sólo por el objeto en sí.



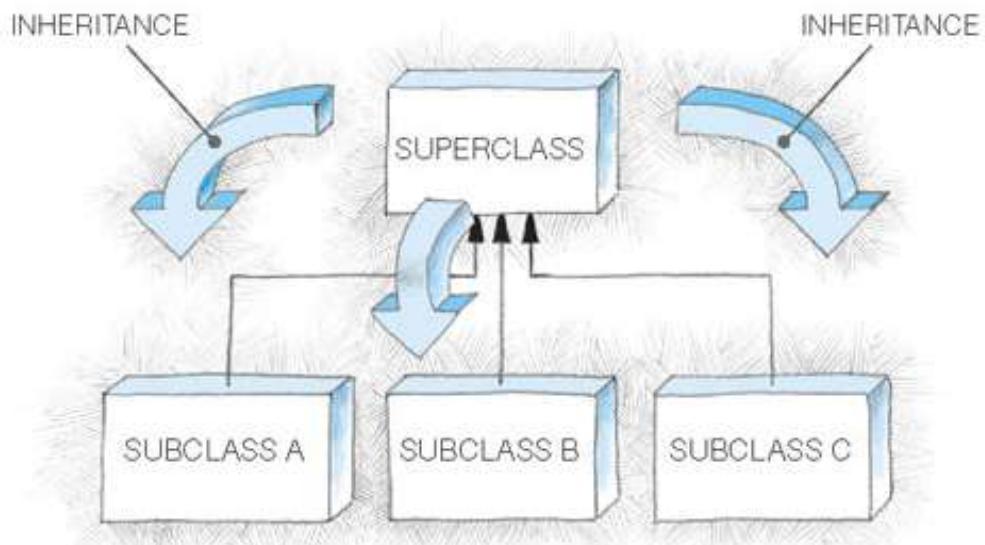
Clases y Herencia

- En resumen
 - Un **objeto** es una **instancia** de una **clase** determinada.
 - Los **métodos** y las **variables(Atributos)** se definen en la **clase**, y sus valores se almacenan en la **instancia**

Clases y Herencia

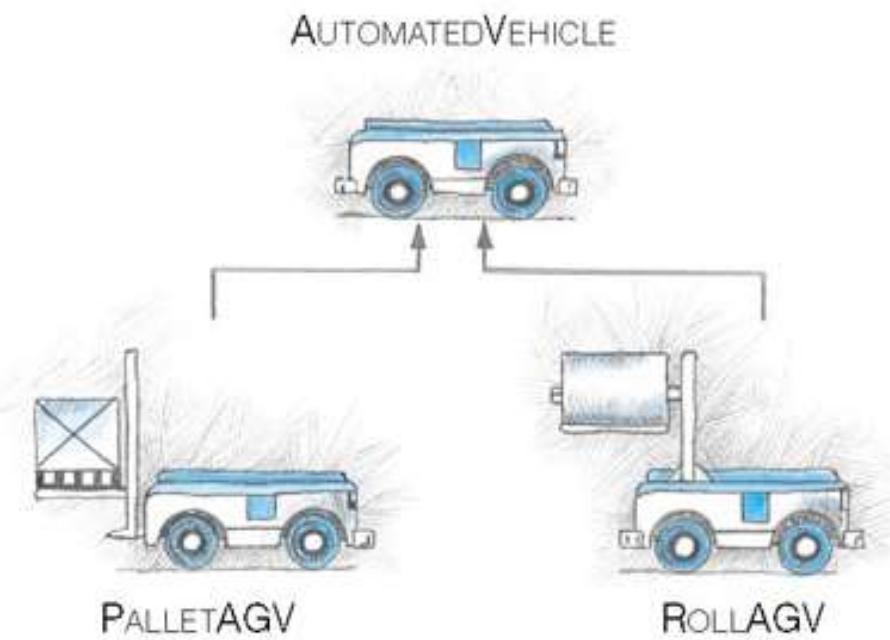
- **Herencia.**- Definición de una clase a partir de la definición de otra clase.

- **Subclases.**- clases heredan de una clase
- **Superclase.**- clase que hereda a otras clases



Clases y Herencia

- Además de los métodos y las variables que heredan, las **subclases** pueden definir sus propios métodos y variables.
- También pueden redefinir cualquiera de los métodos heredados, una técnica conocida como de **overriding (sobrescribir)**.

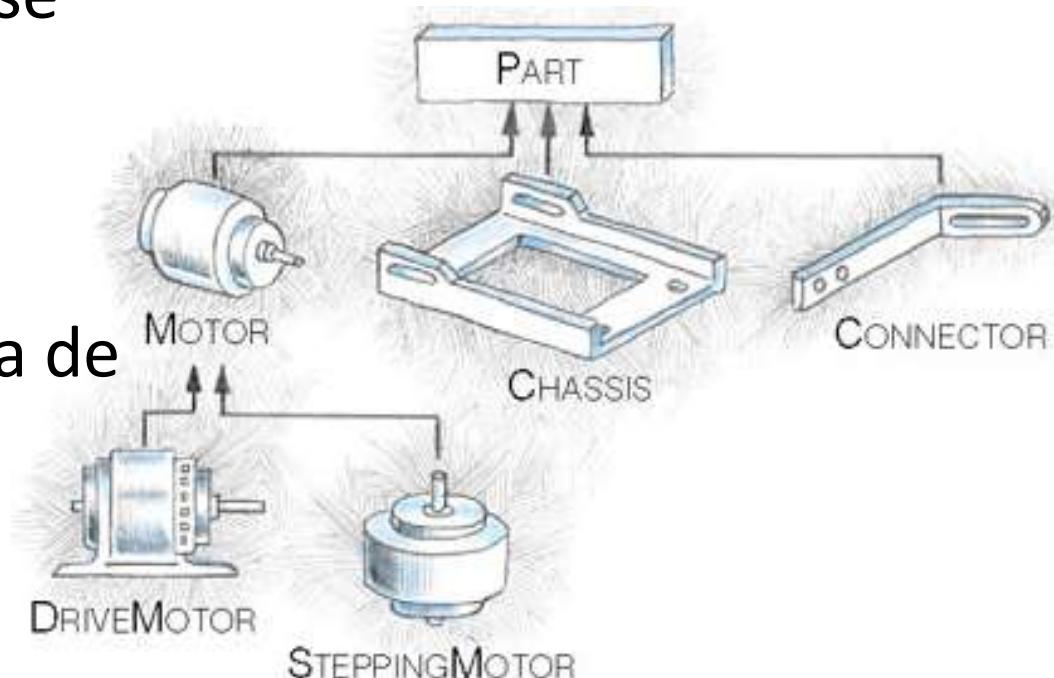


Clases y Herencia

- Dado que las clases definen interfaces de mensaje, estas interfaces son también heredados por sus subclases. Esto significa que todas las subclases de una clase dada están garantizados para responder a los mensajes que pueden ser manejados por la clase padre.

Clases y Herencia

- Las clases se pueden anidar en cualquier grado, y la herencia se acumulará automáticamente a través de todos los niveles. La estructura de árbol resultante se conoce como una **jerarquía de clases**.



Clases y Herencia

- Las jerarquías de clase aumentan la capacidad de los objetos para reflejar la forma en que vemos el mundo real.
 - Generalización y Especialización

Lenguajes Orientados a Objetos

- Muchos lenguajes de programación adic和平n las características del objeto, por lo que la lista de Lenguajes OO est和平 en crecimiento
- Lo importante no es si un lenguaje es "verdaderamente" orientados a objetos, sino lo f\'acil que es aplicar los principios de objetos en el entorno que proporciona el lenguaje.

Lenguajes Orientados a Objetos

- Smalltalk
- C++
- Java
- Eiffel
- Object COBOL
- Visual
- Ada
- C#
- VB.NET
- Clarion
- Delphi
- Lexico (en castellano)
- Objective-C
- OCAML
- Oz
- PHP
- PowerBuilder
- Python
- Ruby
- Object Pascal
- CLIPS
- Actionscript
- PERL
- Simula
- ABAP
- Gambas

Aplicaciones de la Tecnología Orientada a Objetos

- Arquitecturas basadas en Componentes
 - Common Object Request Broker Architecture (CORBA) de Object Management Group (OMG)
 - Component Object Model (COM) de Microsoft

Aplicaciones de la Tecnología Orientada a Objetos

- Bases de Datos
 - Prototipos Experimentales
 - ORION
 - OpenOODB
 - IRIS
 - ODE
 - proyecto ENCORE/ObServer
 - Sistemas Comerciales
 - GEMSTONE/OPAL de ServicLogic
 - ONTOS de Ontologic
 - Objectivity de Objectivity Inc.
 - Versant de Versant Technologies
 - ObjecStore de Object Design
 - O2 de O2 Technology

Aplicaciones de la Tecnología Orientada a Objetos

- Objetos Distribuidos
 - Common Object Request Broker Architecture (CORBA) de Object Management Group (OMG)
 - Distributed Component Object Model (DCOM) de Microsoft
 - Remote Method Invocation (RMI) de Java, Sun Microsystems

Aplicaciones de la Tecnología Orientada a Objetos

- Análisis y Diseño
 - Lenguaje de Modelado Unificado (UML) de la OMG
 - Proceso Unificado Racional (RUP)

Sesión 3

Principios del Modelamiento Visual

Unidad 1

Mg. Gustavo G. Delgado Ugarte

DEFINICIÓN DE MODELAMIENTO

Definición de modelamiento

- Una organización exitosa es aquella que constantemente implementa software de calidad del software que satisfaga las necesidades de sus usuarios
- Una organización que puede desarrollar software en forma oportuna y previsible, con uso eficiente y eficaz de los recursos, tanto humanos como materiales, es la que tiene un negocio sostenible
- Un buen software, es aquel que satisfaga las necesidades cambiantes de los usuarios y el negocio

Definición de modelamiento

- Para implementar un software que cumple con sus metas, tiene que conocer y comprometer a los usuarios de una forma disciplinada, para exponer las necesidades reales de su sistema
- Para desarrollar software de calidad perdurable, tiene que diseñar una arquitectura básica sólida que es resistente al cambio
- Para desarrollar software de forma rápida, eficiente y eficaz, con un mínimo de desechos y retrabajo de software, es necesario tener las personas adecuadas, las herramientas adecuadas y el enfoque correcto
- Para hacer todo esto de forma consistente y predecible, con una apreciación de los costos en la vida útil del sistema, debe tener un proceso de desarrollo robusto que puede adaptarse a las necesidades cambiantes de su negocio y la tecnología

Definición de modelamiento

- El modelamiento es parte central de todas las actividades que conducen a la implementación de un buen software
- Construimos modelos para:
 - Comunicar la estructura deseada y el comportamiento de nuestro sistema
 - Visualizar y controlar la arquitectura del sistema
 - Comprender mejor el sistema que estamos construyendo, con frecuencia la exposición de las oportunidades para la simplificación y la reutilización
 - Gestionar el riesgo

LOS CUATRO PRINCIPIOS DEL MODELAMIENTO VISUAL

Los cuatro principios del modelamiento visual

1. La elección de qué modelos crear tiene una profunda influencia sobre la forma en que es atacado un problema y la forma una solución
2. Cada modelo puede ser expresado en diferentes niveles de precisión
3. Los mejores modelos se conectan a la realidad
4. Un único modelo o punto de vista es insuficiente. Cada sistema no trivial es abordado mejor a través de un pequeño conjunto de modelos casi independientes con múltiples puntos de vista

La elección de qué modelos crear tiene una profunda influencia sobre la forma en que es atacado un problema y la forma una solución

- En otras palabras, elegir bien los modelos
- Los modelos correctos
 - Iluminan brillantemente los problemas de desarrollo más “preversos”
 - Ofrecen una visión que simplemente no podría obtenerse de otra manera
- Los modelos errados inducen al error, lo que nos enfoca en cuestiones irrelevantes

“La elección de qué modelos crear tiene una profunda influencia sobre la forma en que es atacado un problema y la forma una solución”

- Rigurosas y continuas pruebas de los modelos, nos proveerán de un nivel mucho más alto de confianza en que el sistema que se ha modelado se comportará como lo espera en el mundo real
 - Ej. Comportamiento de Vientos de un edificio, mediante un modelo probado en un túnel de viento

“La elección de qué modelos crear tiene una profunda influencia sobre la forma en que es atacado un problema y la forma una solución”

- En el software, los modelos que elija puede afectar su visión del mundo
 - Si construye un sistema a través de los ojos de un desarrollador de bases de datos, es probable que se centrará en los modelos de entidad-relación que impulsan el comportamiento en los procedimientos almacenados y disparadores(triggers)
 - Si construye un sistema a través de los ojos de un analista de estructura, es probable que terminará con los modelos centrados en algoritmia, con flujos de datos de un proceso a otro
 - Si construye un sistema a través de los ojos de un desarrollador orientado a objetos, que terminará con un sistema cuya arquitectura está centrada en un mar de clases y patrones de interacción que dirigen cómo las clases trabajan juntas
 - Modelos ejecutables puede ayudar mucho a la prueba
- Cada visión del mundo conduce a un tipo diferente de sistema, con diferentes costos y beneficios

“Cada modelo puede ser expresado en diferentes niveles de precisión”

- Si está construyendo un rascacielos
 - A veces se necesita un punto de vista de 30.000 pies, para ayudar a los inversionistas visualizar su apariencia
 - A veces se necesita bajar al nivel del suelo, cuando hay un tramo de tubería difíciles o un elemento estructural inusual

“Cada modelo puede ser expresado en diferentes niveles de precisión”

- Lo mismo ocurre con los modelos de software
 - A veces, un modelo ejecutable rápido y simple de la interfaz de usuario es exactamente lo que necesita
 - Otras veces, se tiene que ir a nivel de bits, como cuando se va a especificar las interfaces entre sistemas o cuando se está luchando con cuellos de botella en redes
- Los mejores tipos de modelos son los que permiten elegir el grado de detalle, dependiendo de quién está visualizando qué y por qué
 - Un analista o un usuario final tendrá que centrarse en las cuestiones del qué
 - Un desarrollador tendrá que centrarse en las cuestiones del cómo.
 - Ambos quieren visualizar el sistema en diferentes niveles de detalle en diferentes momentos

“Los mejores modelos se conectan a la realidad”

- Un modelo físico de un edificio que no responde de la misma manera como lo hacen los materiales reales tiene sólo un valor limitado
- Un modelo matemático de una aeronave que asume sólo las condiciones ideales y la fabricación perfecta puede enmascarar algunas de las características potencialmente fatal de la aeronave real
- Es mejor tener modelos que tienen una clara conexión con la realidad, y donde la conexión es débil, saber exactamente cómo estos modelos se han divorciado del mundo real.
- Todos los modelos simplifican la realidad, el truco es asegurarse de que sus simplificaciones no enmascarar los detalles importantes

“Los mejores modelos se conectan a la realidad”

- El talón de Aquiles de las técnicas de análisis estructurado es que el modelo de análisis y el modelo de diseño están desconectados
 - El sistema concebido y el sistema construido divergen con el tiempo
- En los sistemas orientados a objetos, es posible conectar casi todos los puntos de vista independientes de un sistema en un todo semántico

“Un único modelo o punto de vista es insuficiente”

- En la construcción de un edificio, no existe un solo conjunto de planos que muestre todos sus detalles.
- Se necesitará los planos de planta, elevaciones, planos eléctricos, planos de calefacción y planos de Sanitarios.
- **Dentro de cualquier tipo de modelo, necesita múltiples puntos de vista para captar la amplitud del sistema**

“Un único modelo o punto de vista es insuficiente”

- Para entender la arquitectura de un sistema, se necesita varios puntos de vista complementarios entre sí:
 - Una visión de casos de uso (la exposición de los requisitos del sistema)
 - Una vista de diseño (capturar el vocabulario del espacio del problema y el espacio de soluciones)
 - Una visión de la interacción (mostrando las interacciones entre las partes del sistema y entre el sistema y el medio ambiente)
 - Un punto de vista de aplicación (abordando la realización física del sistema)
 - Una vista de despliegue (centrándose en cuestiones de ingeniería de sistemas)
- Cada uno de estos puntos de vista pueden tener aspectos estructurales, así como de comportamiento
- En conjunto, estas vistas representan los planos de software

EL LENGUAJE UNIFICADO DE MODELADO (UML)

UML

- El UML es un lenguaje para la
 - Visualización
 - Especificación
 - Construcción
 - Documentación
- de los artefactos de un sistema de software

UML es un Lenguaje

- Un lenguaje proporciona un vocabulario y las reglas para combinar palabras en dicho vocabulario con el fin de comunicarnos
- Un lenguaje de modelado es un lenguaje cuyo vocabulario y reglas se centran en la representación conceptual y física de un sistema
- Un lenguaje de modelado como el UML es un lenguaje estándar para los modelos de software

UML es un Lenguaje para Visualización

- Problema 1
 - Para muchos programadores, la distancia entre el pensar una aplicación y luego codificarla es cercana a cero. (Lo piensas, lo codificas)
 - El programador se sigue haciendo modelos pero mentalmente
 - La comunicación de los modelos conceptuales a los demás está propenso a errores, a menos que todos los involucrados hablen el mismo idioma.
 - Los proyectos y las organizaciones desarrollan su propio lenguaje, y es difícil de entender lo que está pasando, si es usted un extraño o nuevo en el grupo

UML es un Lenguaje para Visualización

- Problema 2
 - Hay algunas cosas sobre un sistema de software que no se pueden entender, al menos que construyamos modelos que trasciendan al lenguaje de programación textual
 - El significado de una jerarquía de clases se puede deducir, pero no directamente, observando detenidamente el código para todas las clases en la jerarquía
 - La distribución física y la posible migración de los objetos en un sistema basado en la Web se puede deducir, pero no directamente, mediante el estudio de código del sistema.

UML es un Lenguaje para Visualización

- Problema 3
 - Si el desarrollador que cortar el código, nunca escribió los modelos que están en su cabeza, esta información se perderá para siempre o, en el mejor de los casos, sólo será parcialmente recreable a partir de la aplicación una vez que el desarrollador haya avanzado
- Escribir modelos en UML aborda el tercer problema: **Un modelo explícito facilita la comunicación**

UML es un Lenguaje para Visualización

- Algunas cosas son mejores en modelo textual, mientras que otros son mejores modelados de forma gráfica.
 - En todos los sistemas interesante, hay estructuras que trascienden lo que se pueda representar en un lenguaje de programación
- Esto aborda al segundo problema: **UML es un lenguaje gráfico**

UML es un Lenguaje para Visualización

- UML es algo más que un montón de símbolos gráficos; detrás de cada símbolo en la notación UML hay una semántica bien definida
 - Un programador puede escribir un modelo en UML, y otro desarrollador **puede interpretar ese modelo sin ambigüedad**
- Esto resuelve el primer problema

UML es un Lenguaje para Especificación

- **Especificación** significa construir **modelos precisos, sin ambigüedades y completos**
- UML aborda la especificación de todas las decisiones importantes de análisis, diseño e implementación que se deben tomar en el desarrollo y despliegue de un sistema de software

UML es un Lenguaje para Construcción

- UML no es un lenguaje de programación visual, pero sus modelos se pueden conectar directamente a una variedad de lenguajes de programación
 - Es posible mapear un modelo en UML a un lenguaje de programación como Java, C++ o Visual Basic, o incluso a las tablas de una base de datos relacional o una base de datos orientada a objetos

UML es un Lenguaje para Construcción

- Este mapeo permite a la ingeniería directa, la generación de código desde un modelo UML a un lenguaje de programación
- Es posible reconstruir un modelo a partir de una implementación de nuevo en UML (La ingeniería inversa)
- La combinación de estos dos caminos, la generación de código y la ingeniería inversa, permite tener la capacidad de trabajar ya sea en una vista gráfica o textual, mientras las herramientas mantienen coherentes las dos vistas

UML es un Lenguaje para Construcción

- UML es suficientemente expresivo y sin ambigüedades para permitir
 - La ejecución directa de modelos
 - La simulación de sistemas
 - La instrumentación de los sistemas que se ejecutan

UML es un Lenguaje para Documentación

- Una “sana” organización de software produce todo tipo de artefactos, además de código ejecutable. Estos artefactos incluyen (pero no limitados a)
 - Requisitos
 - Arquitectura
 - Diseño
 - El código fuente
 - Planes de Proyecto
 - Pruebas
 - Prototipos
 - Liberaciones

UML es un Lenguaje para Documentación

- Dependiendo de la cultura de desarrollo, algunos de estos artefactos son tratados más o menos formalmente que otros.
- **Tales artefactos** no son sólo los entregables de un proyecto, también **son críticos para controlar, medir y comunicar** acerca de un sistema durante su desarrollo y después de su implementación

UML es un Lenguaje para Documentación

- UML aborda la documentación de la arquitectura de un sistema y todos sus detalles
- UML proporciona un lenguaje para expresar los requisitos y las pruebas
- UML proporciona un lenguaje para el modelado de las actividades de la planificación del proyecto y gestión liberaciones

PROCESO DEL MODELADO VISUAL

Proceso del modelado visual

- El UML es en gran parte independiente del proceso, lo que significa que no está vinculado a ciclo de vida de desarrollo de software en particular.
- Sin embargo, para obtener el mayor beneficio de la UML, se debe considerar un proceso que:
 - Esté orientado a Casos de Uso
 - Esté centrado en la arquitectura
 - Sea Iterativo e incremental

Orientado a Casos de Uso

- Significa usar Casos de Uso como artefacto principal para establecer el comportamiento deseado del sistema, para comprobar y validar la arquitectura del sistema, para las pruebas, y para la comunicación entre los stakeholders del proyecto

Centrado en la arquitectura

- Significa que la arquitectura de un sistema se utiliza como un artefacto principal de la conceptualización, la construcción, gestión y evolución del sistema en desarrollo.

Iterativo e incremental

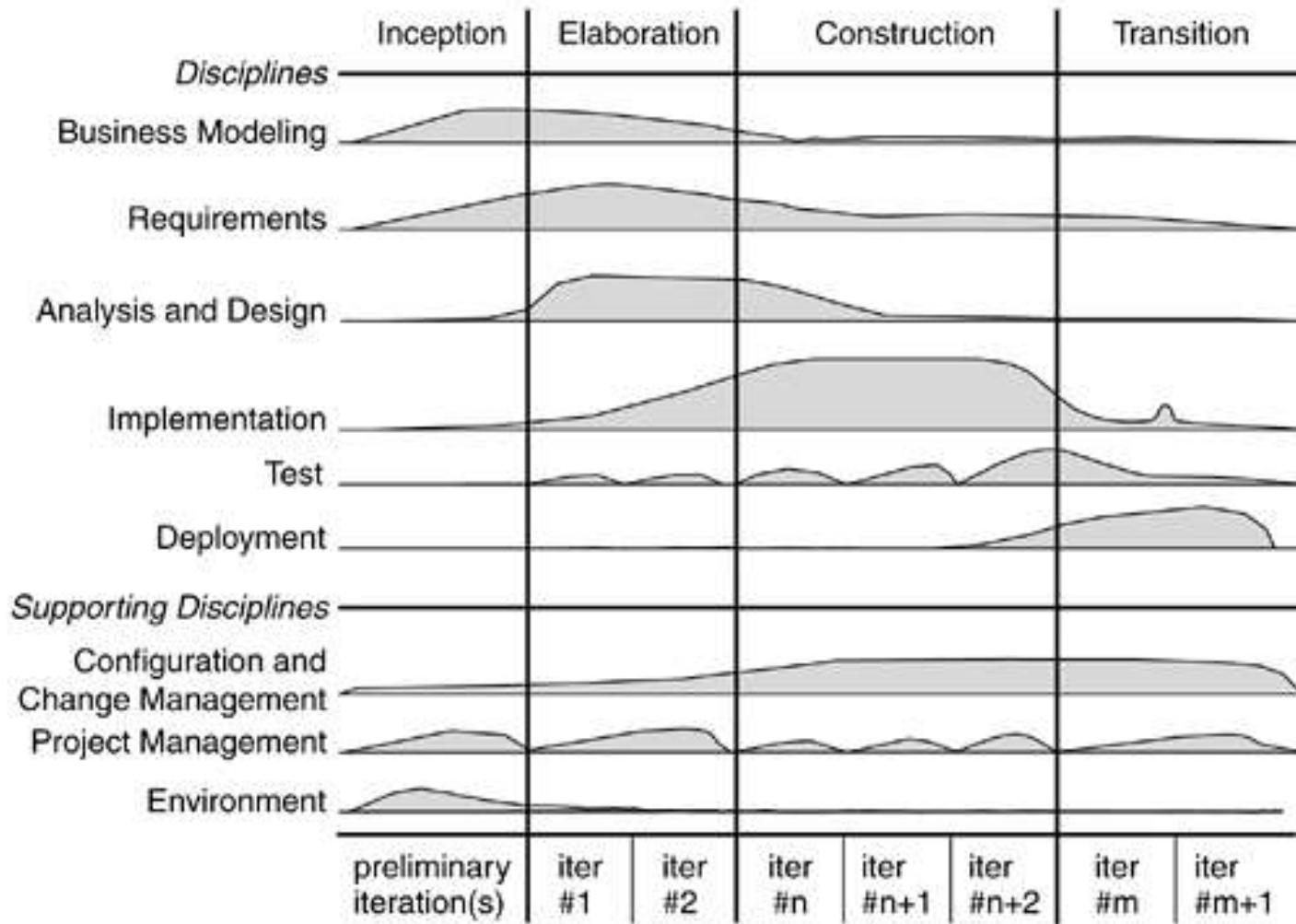
- Un proceso iterativo implica la gestión de una constante liberación de ejecutables
- Un proceso incremental implica la integración continua de la arquitectura del sistema para producir estas liberaciones, en cada nueva liberación se incorpora mejoras constantes de las otras
- En conjunto, un proceso iterativo e incremental es orientado a riesgos, lo que significa que cada nueva liberación se centra en atacar y reducir los riesgos más importantes para el éxito del proyecto

Fases

- Este proceso se puede dividir en fases
- Una fase es el lapso de tiempo entre dos hitos importantes del proceso, cuando un conjunto bien definido de los objetivos se cumplen, los artefactos se han completado, y se toman las decisiones si se debe pasar a la siguiente fase
- Hay cuatro fases en el ciclo de vida del software de desarrollo: inicio, elaboración, construcción y transición

Fases

Figure 2-24. Software Development Life Cycle



Fases

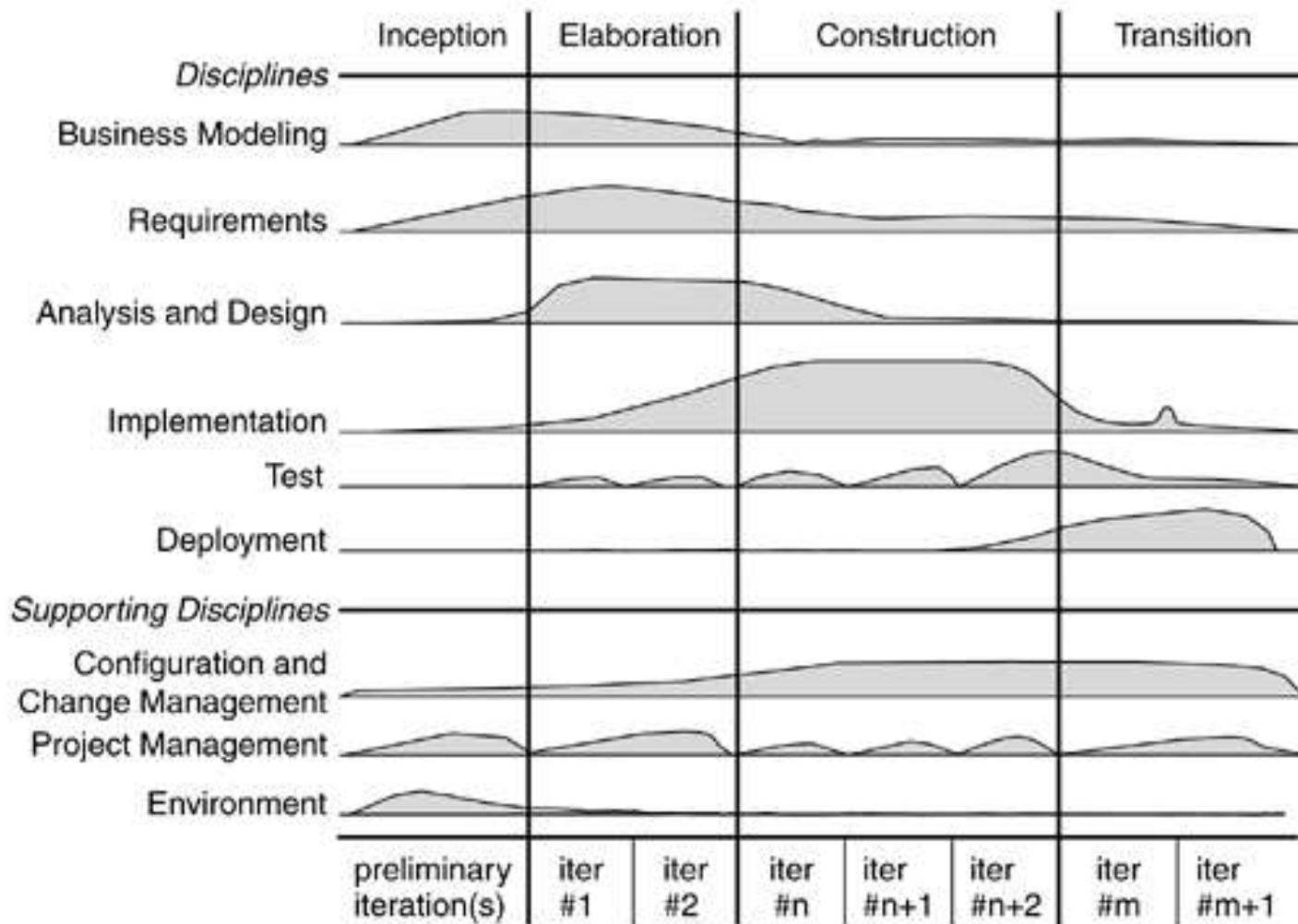
- **Inicio.**- Establece la visión, el alcance, y el plan inicial del proyecto
- **Elaboración.**- Diseña, implementa y prueba una arquitectura sólida y completa el plan de proyecto
- **Construcción.**- Construye la primera versión del sistema operativo
- **Transición.**- Entregar el sistema a sus usuarios finales

Fases

- Inicio y Elaboración se enfocan más en las actividades creativas y la ingeniería del ciclo de vida de desarrollo
- Construcción y Transición se enfocan más en las actividades de producción

Disciplinas

Figure 2-24. Software Development Life Cycle



Disciplinas

- El Rational Unified Process(RUP) consiste en nueve disciplinas
 - **Modelo de Negocio.**- Describe la estructura y la dinámica de la organización del cliente
 - **Requisitos.**- Obtiene requisitos usando una variedad de enfoques
 - **Análisis y diseño.**- Describe los múltiples puntos de vista arquitectónicos

Disciplinas

- **Aplicación.**- Toma en cuenta el desarrollo de software, pruebas unitarias y la integración
- **Prueba.**- Describe las secuencias de comandos, la ejecución de pruebas, y las métricas de seguimiento de defectos
- **Implementación.**- Incluye lista de materiales, notas de la versión, entrenamiento y otros aspectos de la entrega de una aplicación

Disciplinas

- **Gestión de configuración.**- Controla los cambios y mantiene la integridad de los artefactos de un proyecto y las actividades de gestión
- **Gestión de Proyectos.**- Describe las diversas estrategias de trabajo con un proceso iterativo
- **Medio Ambiente.**- Cubre la infraestructura necesaria para desarrollar un sistema

Sesión 4

Principios del Modelamiento Visual

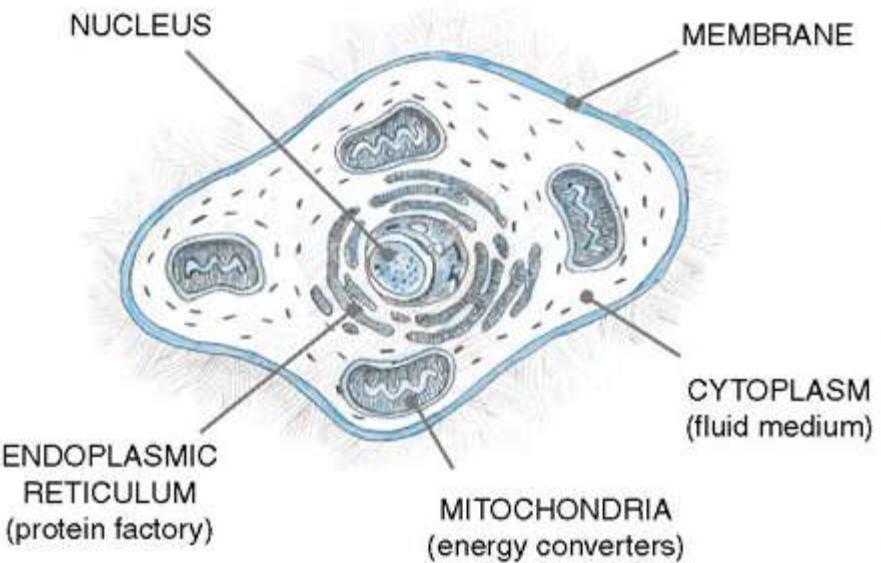
(Conceptos de Objetos)

Unidad 1

Mg. Gustavo G. Delgado Ugarte

Definición de Objeto

- Un **objeto** es una **instancia** de una **clase**
- Los objetos son “paquetes” que combinan la información y el comportamiento.
 - Ejemplo: Célula
 - Información.- contenida en las moléculas de proteína dentro del núcleo
 - Comportamiento.- Ej. la conversión de energía al movimiento, se lleva a cabo por las estructuras fuera del núcleo



Principios de la Orientación a Objetos

- **Abstracción**
 - Capacidad de eliminar los rasgos comunes de toda una categoría de objetos de datos y colocarlos en una definición por separado
 - Tal definición es abstracta, ya que define una clase de objetos en lugar de un objeto específico

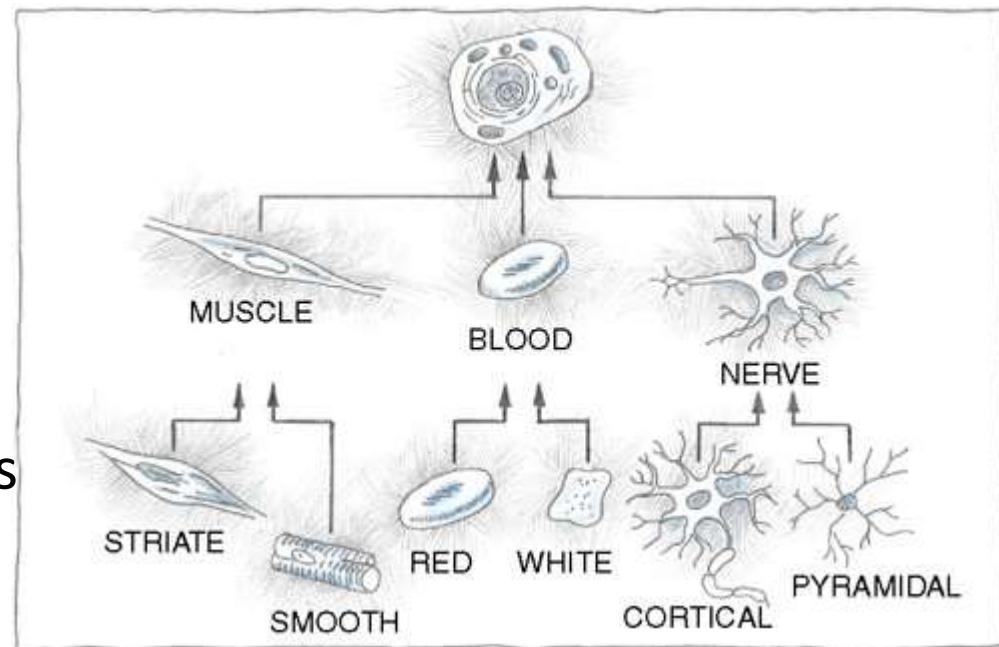
Principios de la Orientación a Objetos

- **Encapsulación**
 - Capacidad de envolver los datos y la lógica(comportamiento) de un objeto, permitiendo construir programas complejos a partir de objetos simples
 - La complejidad es literalmente escondida en los objetos, con importantes beneficios en cuanto a fiabilidad y facilidad de mantenimiento

Principios de la Orientación a Objetos

- **Herencia**

- Capacidad de definir un objeto como una variación de otro
- El nuevo objeto se dice que hereda las propiedades del objeto original, con excepción de aquellas propiedades que se sobreescreiben (overriding) o se asigna un valor diferente



Principios de la Orientación a Objetos

- **Polimorfismo**

- Capacidad de utilizar una variación (o "subclase", en la terminología orientada a objetos) donde se prevé una instancia de la clase original
- Capacidad de manipular objetos de distintas clases utilizando sólo el conocimiento de sus propiedades comunes sin tener en cuenta su clase exacta
- Se refiere a la capacidad para que varias clases derivadas de una antecesora utilicen un mismo método de forma diferente

Definición de Clase

- Una clase es una plantilla de software que define los métodos y las variables que se incluirán en un determinado tipo de objeto
- Las clases son abstracciones que representan a un conjunto de objetos con un comportamiento e interfaz común
- Podemos definir una clase como "un conjunto de cosas (físicas o abstractas) que tienen el mismo comportamiento y características... Es la implementación de un tipo de objeto (considerando los objetos como instancias de las clases)". (**Piattini 1996**).

Polimorfismo y generalización

- El polimorfismo adquiere su máxima expresión mediante la herencia
 - Una figura representa figuras genéricas
 - Una clase figura puede responder a mensajes *dibujar*, *borrar* y *mover*
 - Cualquier clase derivada de figura es un tipo de figura y puede recibir el *mismo* mensaje (Ej. Cuadrado, Triangulo, Elipse)
 - El polimorfismo permite que una misma función se comporte diferente según sea la clase sobre la que se aplica
 - Si se envía el mensaje dibujar, la respuesta será diferente según la clase (Ej. Un triangulo se dibuja diferente a un cuadrado)

Organizando los elementos de los modelos: Paquetes

- Un paquete es parte de un modelo
- Cada parte de un modelo debe pertenecer a un paquete
- El modelador puede asignar el contenido de un modelo a un conjunto de paquetes
 - La asignación debe seguir un principio racional como funcionalidad común, implementación estrechamente relacionada y un punto de vista común.

Organizando los elementos de los modelos: Paquetes

- Los paquetes contienen elementos de modelado al más alto nivel
 - Clases y relaciones
 - Máquinas de estado
 - Diagramas de casos de uso
 - Interacciones
 - Colaboraciones
- Los paquetes pueden contener otros paquetes

Organizando los elementos de los modelos: Paquetes

- Existen varias maneras de organizar los paquetes de un sistema
 - Por la vista
 - Por la funcionalidad
 - En base a cualquier otra cosa que elija el modelador

Organizando los elementos de los modelos: Paquetes

- Los paquetes son unidades de organización jerárquica de uso general en modelos de UML
 - Pueden ser usados para
 - Almacenamiento
 - Control de acceso
 - Gestión de la configuración
 - Construcción de bibliotecas que contengan fragmentos reutilizables del modelo

Organizando los elementos de los modelos: Paquetes

- Si se elige bien los paquetes, reflejan la arquitectura de alto nivel de un sistema
 - Descomposición en subsistemas y sus dependencias
 - La dependencia entre paquetes resume la dependencia entre los contenidos del paquete

Organizando los elementos de los modelos: Paquetes

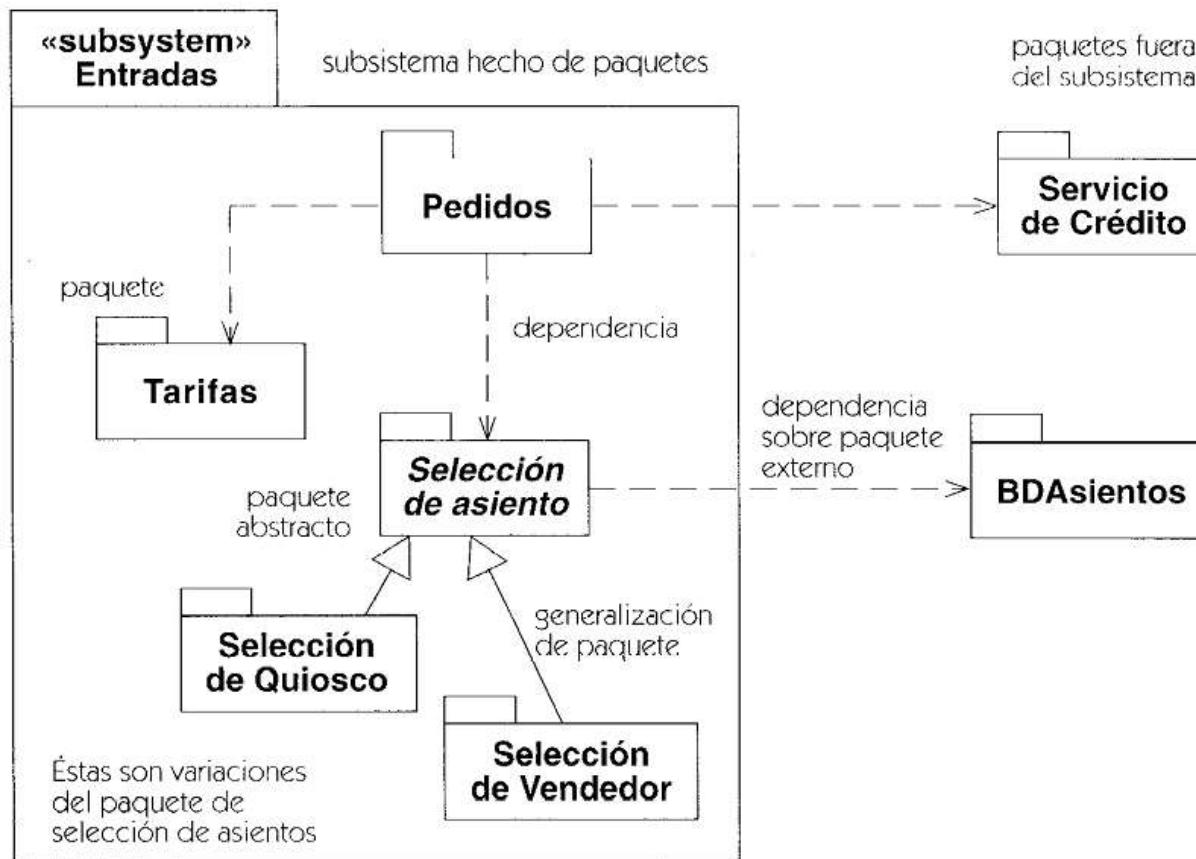


Figura 10.1 Paquetes y sus relaciones

Sesión 5

Introducción al Unified Model Language (UML)

Unidad 2

Introducción al Unified Model Language (UML)

Mg. Gustavo G. Delgado Ugarte

Historia del UML

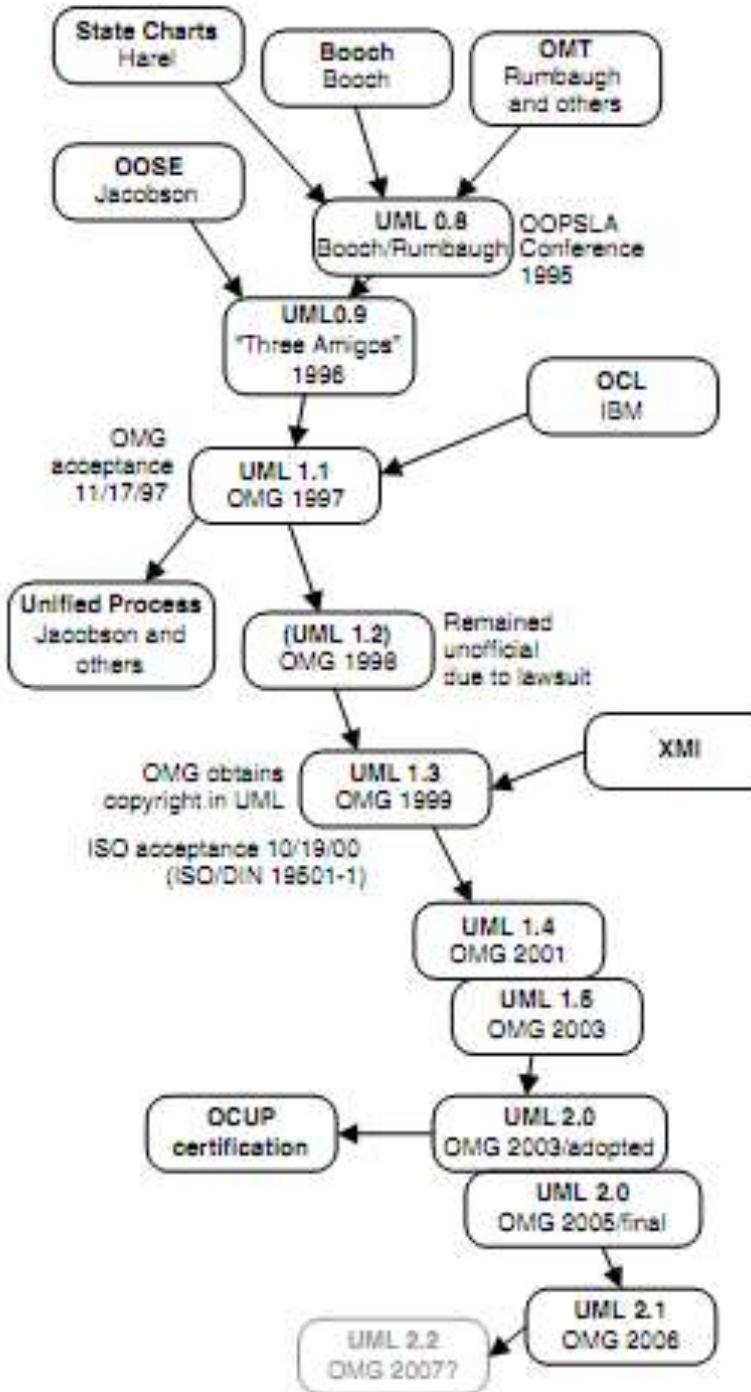
- El UML es la creación de Grady Booch, James Rumbaugh, e Ivar Jacobson (“Los Tres Amigos”)
 - Trabajaron en organizaciones separadas durante los años 1980 y principios de 1990, elaborando cada uno su propia metodología de análisis y diseño orientado a objetos
 - A mediados de la década de 1990, empezaron a pedirse prestadas ideas uno de otros, así que decidieron desarrollar su trabajo juntos
 - En 1994 Rumbaugh se unió a Rational Software Corporation, donde Booch ya estaba trabajando. Jacobson ingresó a Rational, un año después

Historia del UML

- Borradores de UML comenzaron a circular en toda la industria del software, y la respuesta resultante trajo cambios sustanciales
- Muchas empresas sintieron que UML serviría a sus fines estratégicos, surgió UML consortium (consorcio UML) surgido
 - Miembros: DEC, Hewlett-Packard, Intellicorp, Microsoft, Oracle, Texas Instruments, Rational, otros.
- **1997** el consorcio produjo la versión 1.0 de UML y lo presentó al Object Management Group (OMG) en respuesta a la petición de la OMG publicó una solicitud de propuesta de un lenguaje de modelado estándar

Historia del UML

- El consorcio, generó la versión 1.1, y lo presentó a los OMG, que lo aprobó a finales de **1997**.
- OMG se hizo cargo del mantenimiento de UML y produjo dos revisiones más en **1998**.
- UML se ha convertido en un estándar de facto en la industria del software, y sigue evolucionando
 - Versiones 1.3, 1.4, 1.5 ,2.0
- Además de las ideas de Booch, Rumbaugh y Jacobson, incluyen otras, por ejemplo, diagramas de estado de Harel (Harel, 1987)



Tecnologías base del UML

- Tecnología de Objetos
 - Métodos Orientado a Objetos
 - OMT (Object-modeling technique) de Rumbaugh
 - Método Booch de Grady Booch
 - Método OOSE (Object-Oriented Software Engineering) de Jacobson
 - Notación de Casos de Uso del Objectory
 - Notación de componentes de Booch
 - Diagramas de Estado de Harel
 - Conceptos de muchos otros métodos OO
 - Ej. Bran Selic, Conrad Bock, James Odell, etc.

OMG (Object Management Group)

- El Object Management Group, una organización internacional de la industria en la cual todas las empresas de TI importantes son miembros, además de los que desarrollaron y estandarizaron UML
- Las Empresas miembros del OMG cooperan en el mantenimiento y la aplicación del estándar UML
- Los miembros del OMG incluyen
 - Grandes empresas internacionales: IBM, Hewlett-Packard, Sun Microsystems, Telelogic, Boeing, Adobe y DaimlerChrysler
 - Empresas innovadoras de tamaño medio: las empresas alemanas Prostep AG, b + m Informatik, y OOSE GmbH

Sub especificaciones UML

- UML, versión 2.0 ha sido formalmente dividido en las sub especificaciones siguientes:
 - **Infraestructura:** Núcleo de la arquitectura, los perfiles, y los estereotipos
 - **Superestructura:** elementos de los modelos estáticos y dinámicos
 - **Object Constraint Language (OCL):** Un lenguaje formal utilizado para describir expresiones en los modelos UML
 - **Diagrama de intercambio:** El formato de intercambio de diagramas UML.

Metamodelo UML

- UML se define formalmente empleando un metamodelo – un modelo de las estructuras del UML
- El metamodelo está expresado en UML

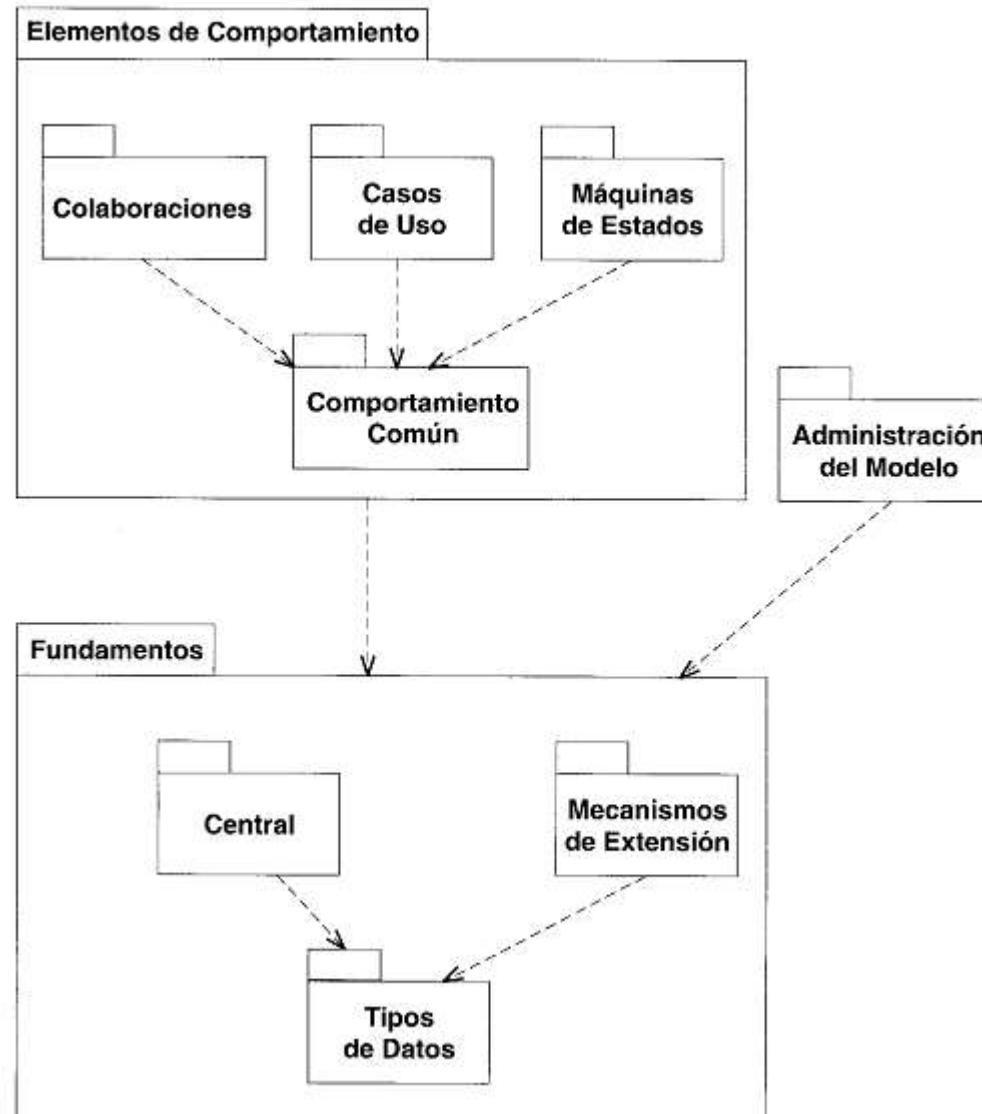
Metamodelo UML

- Cada sección de documento semántico contiene:
 - Un diagrama de clases que muestra una porción del metamodelo
 - Una descripción textual de las clases del metamodelo definidas en esa sección, con atributos y relaciones
 - Una lista de restricciones aplicables a los elementos del modelo expresado en lenguaje natural y en OCL
 - Una descripción de la semántica dinámica de las estructuras de UML definidas en la sección (Informal)

Estructura del Metamodelo

- El metamodelo está dividido en 3 paquetes principales
 - **El paquete fundamentos** define la estructura estática de UML
 - **El paquete de elementos de comportamiento** define la estructura dinámica del UML
 - **El paquete de administración del modelo** define la estructura organizativa de los modelos de UML

Estructura del Metamodelo



Paquete de Fundamentos

- Núcleo(Central)
 - Describe las principales estructuras estáticas del UML
 - Clasificadores
 - Contenido.- incluye atributo, operación, método y parámetro
 - Relaciones.- incluyen generalización, asociación y dependencia
 - Se definen varias metaclasses abstractas
 - Elemento generalizable
 - Espacio de nombres
 - Elemento del modelo
 - Define plantilla , tipos de subclases de dependencia, componente, nodo y comentario

Paquete de Fundamentos

- Tipo de Dato
 - Describe las clases de tipos de datos que utiliza el metamodelo
- Mecanismos de extensión
 - Describe los mecanismos
 - Restricción
 - Estereotipo
 - Valor etiquetado

Paquete Elementos de Comportamiento

- Tiene un subpaquete por cada vista principal
- Tiene un paquete para estructuras de comportamiento que comparten las tres vistas principales

Paquete Elementos de Comportamiento

- Comportamiento común
 - Describe señal, operación y acción
 - Describe instancias de clases correspondientes a diferentes descriptores
- Colaboraciones
 - Describe colaboración, interacción, mensaje, rol de clasificador y asociación

Paquete Elementos de Comportamiento

- Casos de uso
 - Describe actor y caso de uso
- Maquinas de estado
 - Describe la estructura de una máquina de estados
 - Estado
 - Pseudoestado
 - Evento
 - Señal
 - Transición
 - Condición de guarda
 - Describe estructuras adicionales para modelos de actividad
 - Estado de acción
 - Estado de actividad
 - Estado de flujo de objeto

Paquete de administración de modelos

- Describe los paquetes, modelos y subsistemas
- Describe las propiedades de propiedad y visibilidad de los espacios de nombre y de los paquetes
- No posee subpaquetes

Arquitectura 4+1

- Una de las herramientas para representar modelos de arquitectura anteriores al UML es la denominada 4+1 vistas propuesta por Kruchten
- El modelo describe la arquitectura de software del sistema a través de 5 vistas concurrentes

Arquitectura 4+1

- **La vista lógica**
 - Trata de clases y subsistemas, tiene las siguientes particularidades:
 - Soporta los requerimientos funcionales
 - Identifica mecanismos y diseña elementos comunes a través del sistema
 - Utiliza los diagramas de clases y la notación de Booch además de utilizar el estilo arquitectónico orientado a objetos

Arquitectura 4+1

- **La vista concurrente o de procesos**
 - Describe el diseño de concurrencia y aspectos de sincronización
 - Especifica las líneas de mando que ejecutan cada operación en cada una de las clases señaladas en la vista lógica
 - Los diseñadores realizan esta vista en varios niveles de abstracción, además de dividir el software en conjuntos independientes de tareas, es decir, se empaqueta en pequeños programas o librerías del subsistema
 - Los estilos arquitectónicos más usados son los de tuberías y filtros o el de cliente/servidor.

Arquitectura 4+1

- **La vista de componentes o de desarrollo**
 - Describe la organización estática de software en los ambientes de desarrollo
 - Es una importante característica de lógica en casos de vistas, autónomas, persistentes y de distribución, describe la participación en diferentes operaciones, determina si existe persistencia entre un objeto y otro, además determina el estado de los objetos y operaciones de accesibilidad por muchos nodos
 - Es recomendable usar el estilo de arquitectura por capas.

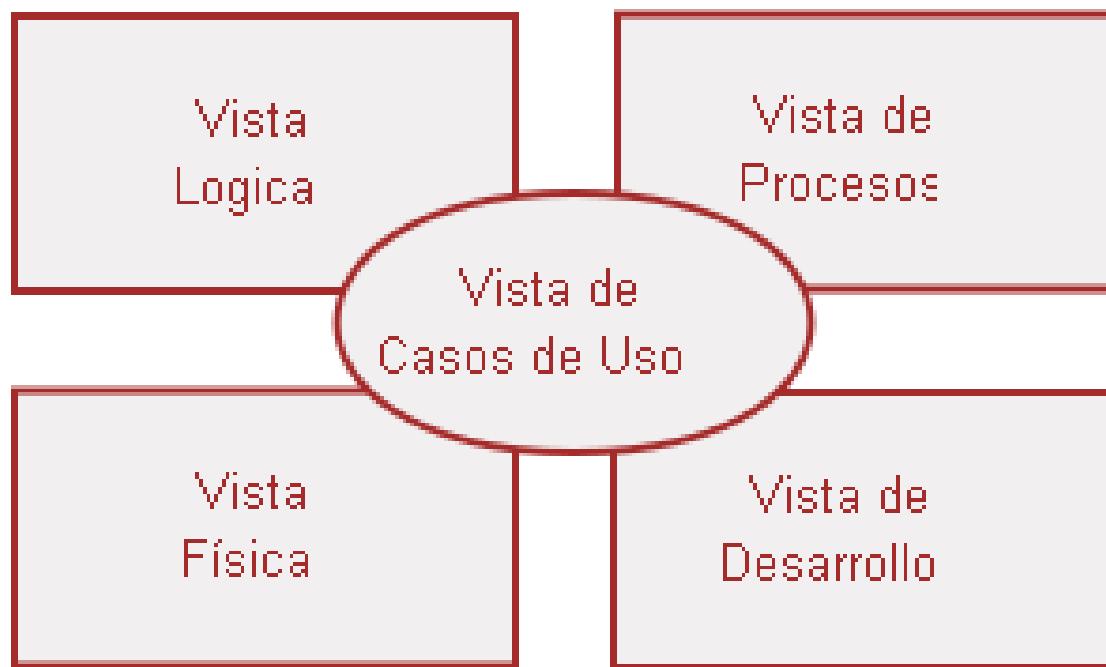
Arquitectura 4+1

- **La vista distribuida o física**
 - Se refiere a la implementación en módulos y fragmentación en muchas capas
 - Colecciona las categorías de clases y grupos
 - Describe el mapeo del software en el hardware y toma en cuenta los requerimientos funcionales del sistema, tales como confiabilidad, respuesta y escalabilidad

Arquitectura 4+1

- **La vista de casos de uso o escenarios**
 - Gobierna los requerimientos que son necesarios para el usuario final, y construye elementos comunes a través del sistema
 - Esta vista es redundante relacionado con el conjunto que forman las anteriores de ahí que se le denomina +1, pero su inclusión es vital ya que desempeña dos roles importantes
 - Actúa como indicador que ayuda al diseñador a descubrir los elementos de la arquitectura durante su diseño
 - Valida e ilustra el diseño de la misma
 - La notación es similar a la que se utiliza en la vista lógica, a excepción de que usa los conectores de la vista de procesos para indicar la interacción entre objetos

Arquitectura 4+1



Estándares relacionados con UML y con la orientación a objetos

- Entornos de desarrollo integrado
- Herramientas CASE
- Técnicas de Modelado a Objetos
- Programación orientada a objetos
- **XMI**, un formato estándar basado en XML para el intercambio de modelos UML.
- **OCL**, Lenguaje de especificación para los diferentes modelos en UML.
- **Webml**, Metodología para el diseño de Sistemas de Información Web.
- **Business Process Modeling Notation o BPMN**, notación gráfica estandarizada que permite el modelado de procesos de negocio, en un formato de flujo de trabajo (workflow), es actualmente mantenida por el OMG
- **CORBA** (*Common Object Request Broker Architecture* — arquitectura común de intermediarios en peticiones a objetos); es un estándar que establece una plataforma de desarrollo de sistemas distribuidos facilitando la invocación de métodos remotos bajo un paradigma orientado a objetos. Fue definido y está controlado por el OMG

Sesión 6

Principio de los diagramas de UML

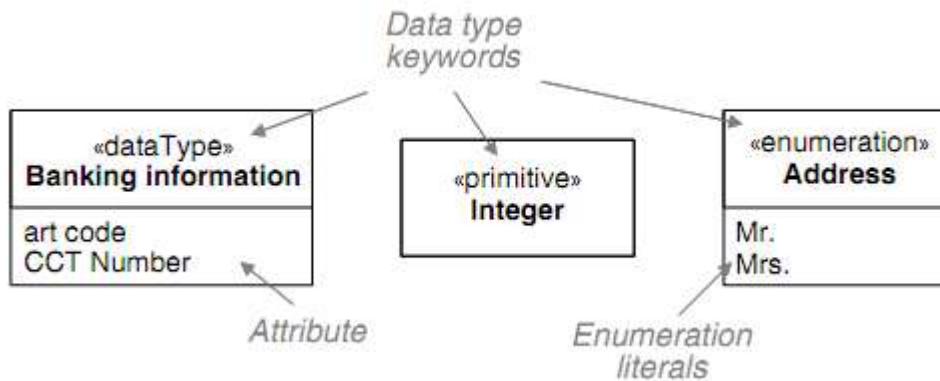
Unidad 2

Introducción al Unified Model
Language (UML)

Mg. Gustavo G. Delgado Ugarte

Tipos de datos en UML

- UML distingue los siguientes tipos de datos:
 - Tipos de datos simples (DataType)
 - Tipos de datos primitivos (PrimitiveType)
 - Tipos Enumeración



Tipos de datos en UML

- Tipos de datos simples (DataType)
 - Tipo con valores que no tienen una identidad
 - Ej. dinero, información bancaria
 - Dos instancias de un tipo de datos con los mismos valores de los atributos son indistinguibles

Tipos de datos en UML

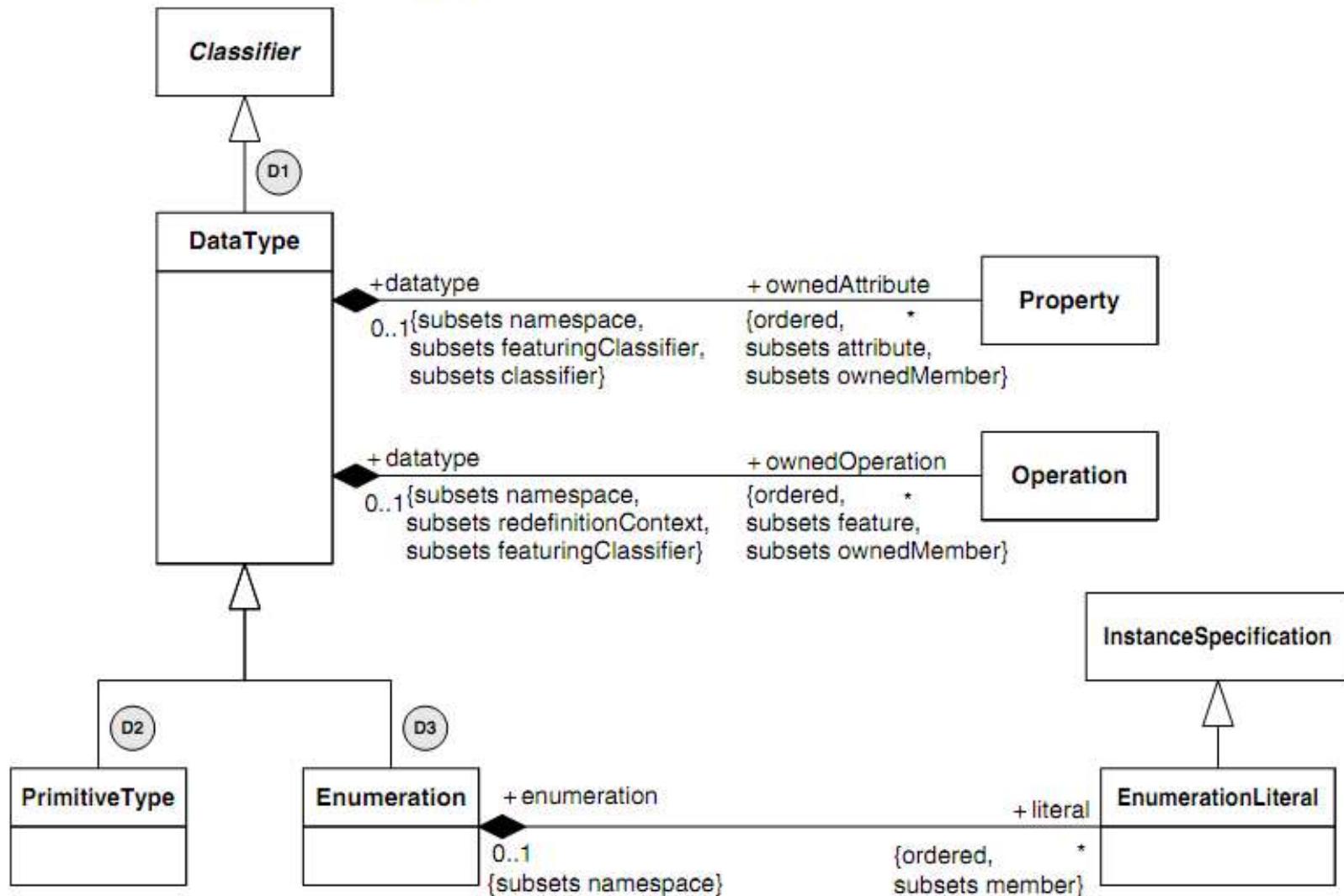
- Tipos de datos primitivos (PrimitiveType)
 - Tipo de datos simple sin estructuras
 - UML define los siguientes tipos de datos primitivos:
 - Entero: (infinito)
 - Conjunto de enteros: (...,-2,-1,0,1,2,...)
 - Boolean: verdadero, falso
 - Naturales Ilimitados (Infinito)
 - Conjunto de números naturales (0,1,2,...)
 - El símbolo de infinito es el asterisco (*)
 - Este tipo de datos se utiliza, por ejemplo, en el metamodelo para definir multiplicidades

Tipos de datos en UML

- Tipos Enumeración
 - Tipos de datos simples con valores que se originan de un conjunto limitado de literales enumerados
 - Ejemplo: Estado civil

Tipos de datos en UML

Metamodel



Clasificadores

- Aunque el metamodelo de la clase Clasificador es abstracto, lo que significa no se ve como un elemento de modelado, una notación se define en la especificación
- Esta notación es heredada por las subclases y utilizados por ellas
 - Algunas de la notaciones eventualmente se sobrescribirán

Clasificadores

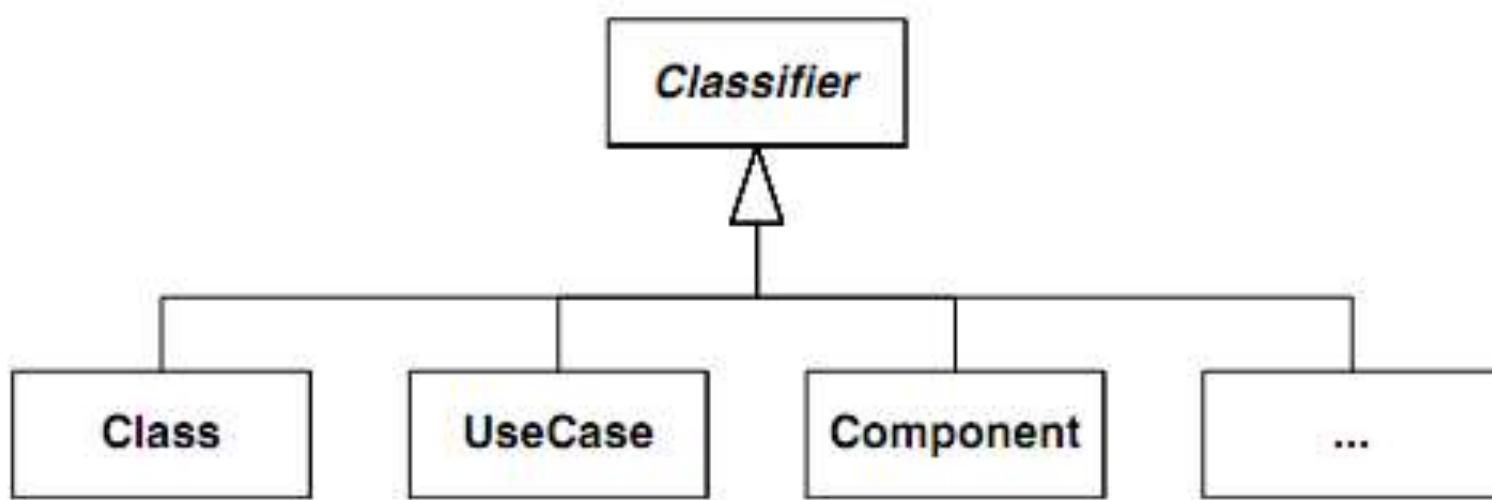


FIGURE 2.28 A classifier and a set of concrete subclasses (simplified).

Clasificadores

- Es una clase base abstracta que clasifica las instancias en lo que respecta a sus características
- Es un espacio de nombres (namespace) y un tipo, y puede ser generalizada
- Clasificadores concretos en el metamodelo UML incluyen, por ejemplo, clase, componente, y caso de uso
- Un clasificador asocia un conjunto de características
 - Características concretas son operaciones y atributos

Clasificadores

Metamodel

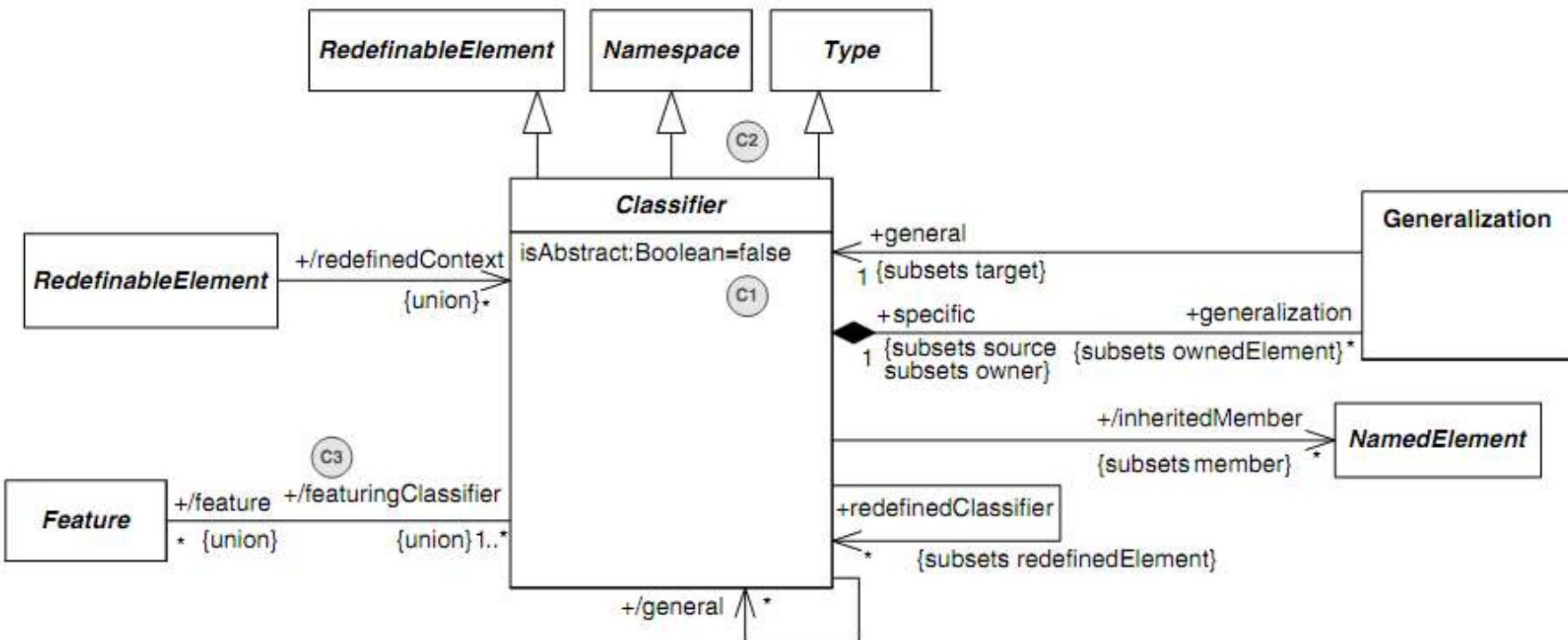


FIGURE 2.32 The classifier metamodel.

Clasificadores

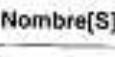
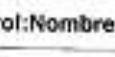
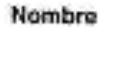
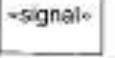
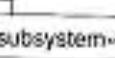
- La notación estándar para el clasificador es un rectángulo que contiene el nombre con el nombre de la subclase entre guillemets (<<, >>) por encima de ella
- Cada subclase introduce variantes de notación
 - El nombre de la clase concreto del metamodelo («clase») suele omitirse
 - Un caso de uso es generalmente representado en la notación por una elipse

Clasificadores

<i>Notation</i>	«class» aClass	«use case» aUseCase	«component» aComponent
<i>Alternative notation</i>	aClass	aUseCase 	 aComponent

FIGURE 2.29 Examples of the notation of Classifier subclasses.

Tabla 4.1 Tipos de Clasificadores

Clasificador	Función	Notación
actor	Un usuario externo al sistema	
clase	Un concepto del sistema modelado	
clase en un estado	Una clase restringida a estar en el estado dado	
rol	Clasificador restringido a un uso particular en una colaboración	
componente	Una pieza física de un sistema	
tipo de dato	Un descriptor de un conjunto de valores primitivos que carecen de identidad	
interfaz	Un conjunto de operaciones con nombre que caracteriza un comportamiento	
nodo	Un recurso computacional	
señal	Una comunicación asíncrona entre objetos	
subsistema	Un paquete que es tratado como una unidad con una especificación, implementación, e identidad	
caso de uso	Una especificación del comportamiento de una identidad y su interacción con los agentes externos	

Clasificadores

- Un clasificador es abstracto si su descripción es incompleta
 - Esto significa que no se pueden crear instancias
- La Abstracción es una característica de los clasificadores
 - Los nombres de los clasificadores abstractos se escriben en cursiva.
 - Si lo desea, puede agregar la cadena de propiedad {abstracta}.
 - Además de la zona de notación para el nombre (compartimentos en UML), puede haber otros, por ejemplo, para los atributos y las operaciones

Clasificadores

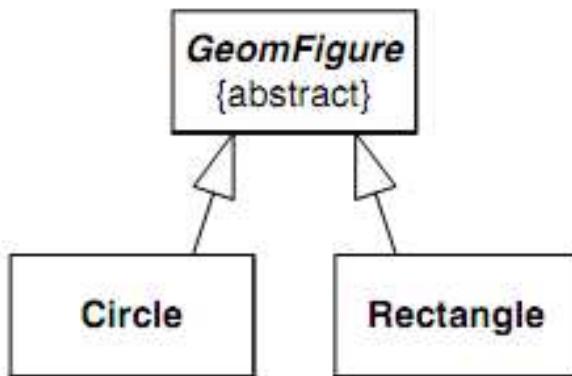


FIGURE 2.30 An abstract class called **GeomFigure**.

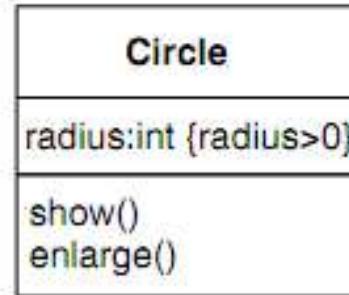


FIGURE 2.31 A class with compartments for name, attributes, and operations.

Introducción a los diagramas en UML

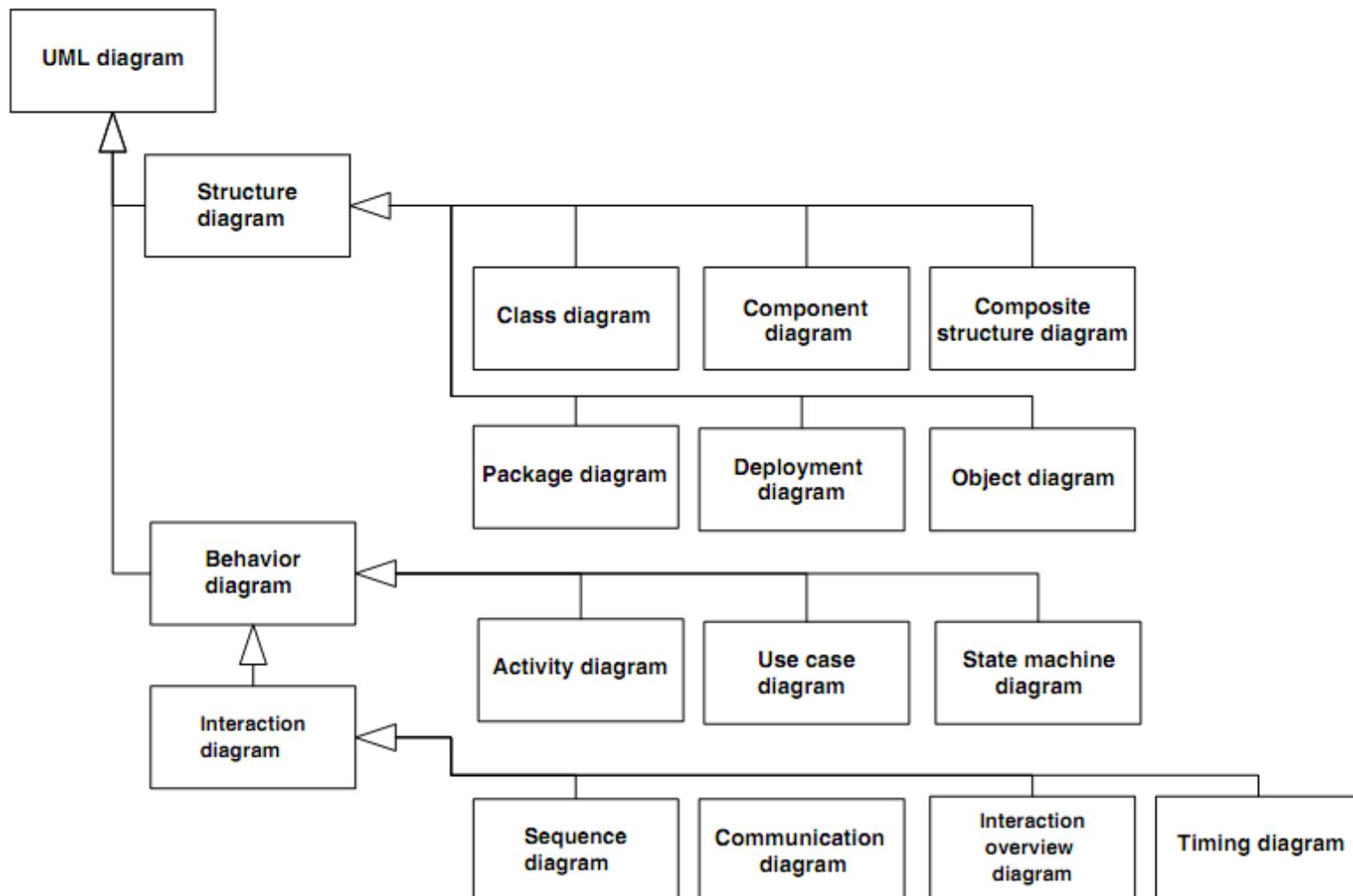


FIGURE 2.3 Overview of the UML diagrams.

Introducción a los diagramas en UML

- Un diagrama consta de un área dentro de un rectángulo y un encabezado en la esquina superior izquierda
- El encabezado diagrama muestra el tipo de diagrama (opcional), el nombre del diagrama (obligatorio), y los parámetros (opcional).

Introducción a los diagramas en UML

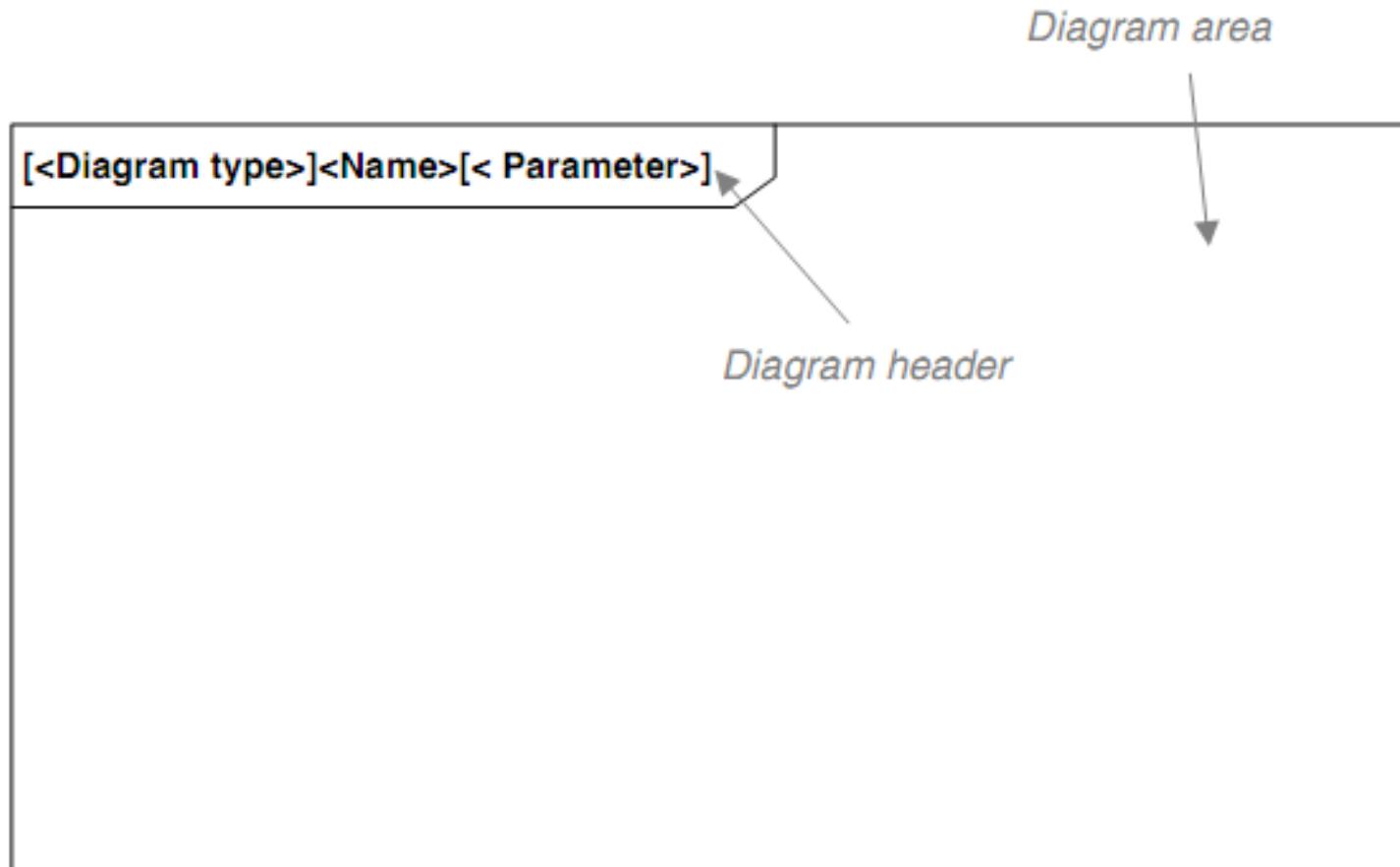


FIGURE 2.4 Basic notation for diagrams.

Introducción a los diagramas en UML

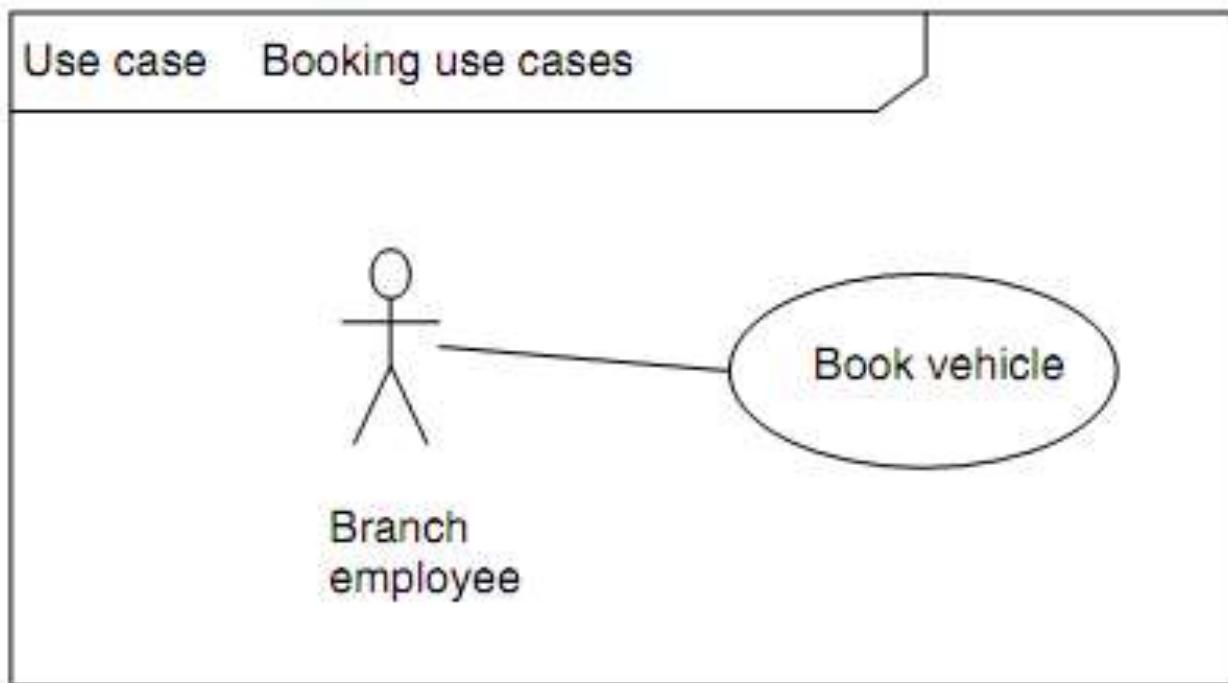


FIGURE 2.5 Example of a use case diagram.

Tipos de elementos del UML

Metamodel

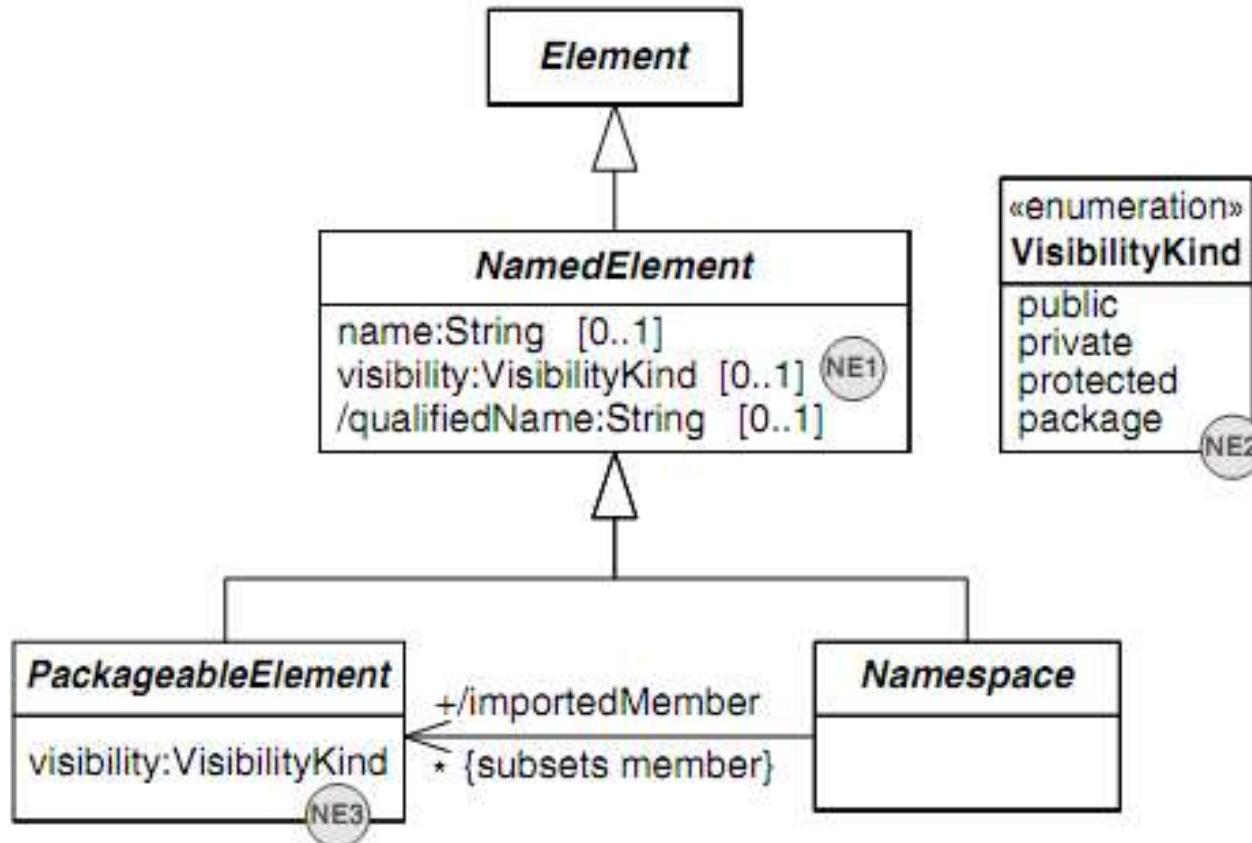


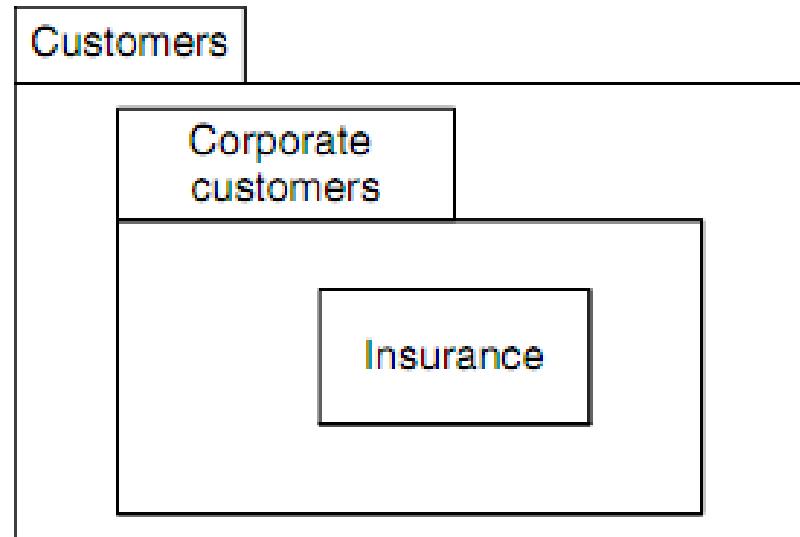
FIGURE 2.15 The metamodel for **NamedElement**.

Tipos de elementos del UML

- **Elemento nombrado** es un elemento que puede tener un nombre y una visibilidad definida (público, privado, protegido , empaquetado)

Tipos de elementos del UML

- **Espacio de nombres (namespace)** es un elemento nombrado que puede contener elementos nombrados
 - Dentro de un espacio de nombres, elementos nombrados se identifican por sus nombres
 - Tienen un nombre cualificado, resultando en espacios de nombres anidados, por ejemplo, la clase Customers:: CorporateCustomers:: Insurance



Tipos de elementos del UML

- **Elemento empaquetable** es un elemento nombrado que puede pertenecer directamente a un paquete.
 - La declaración de la visibilidad es obligatoria para un elemento empaquetable

+ = public

- = private

= protected

~ = package

Estereotipos

- Los estereotipos son extensiones formales de los elementos del modelo existente en el metamodelo UML, es decir, extensiones metamodelo
- El elemento de modelado está directamente influenciada por la semántica definida por la extensión
- En lugar de introducir un elemento nuevo del modelo al metamodelo, los estereotipos agregan semántica a un elemento actual del modelo

Estereotipos

- Los Estereotipos clasifican los posibles usos de un elemento del modelo. Esta clasificación no tiene nada que ver con el modelado de metaclases.
- Ciertas características comunes son atribuidos a uno o varios elementos del modelo

Estereotipos múltiples

- Diversos estereotipos se pueden utilizar para clasificar a un solo elemento de modelado
- La representación visual de un elemento puede estar influenciada por la asignación de estereotipos
- Los estereotipos se pueden añadir a los atributos, operaciones y relaciones
- Los estereotipos pueden tener atributos para almacenar información adicional

Estereotipos - Notación

- Un estereotipo se coloca antes o por encima del nombre del elemento (por ejemplo, nombre de la clase) y encerrado en guillemets («,»)
- Los caracteres «y», a veces conocidos como comillas en ángulo, no debe confundirse con los caracteres doble mayor que, <<, o doble menor que, >>
- No todas las ocurrencias de esta notación significa que este viendo un estereotipo. Palabras claves predefinidas en UML también se encierran en guillemets

Estereotipos – Símbolos Gráficos

- Como alternativa a esta notación puramente textual, símbolos especiales pueden ser utilizados
- Los estereotipos «entity» («entidad»), «boundary» («límite») y «control» son buenos ejemplos.
- Las herramientas son libres de uso de algún color especial u otro elemento de énfasis visual

Estereotipos – Símbolos Gráficos

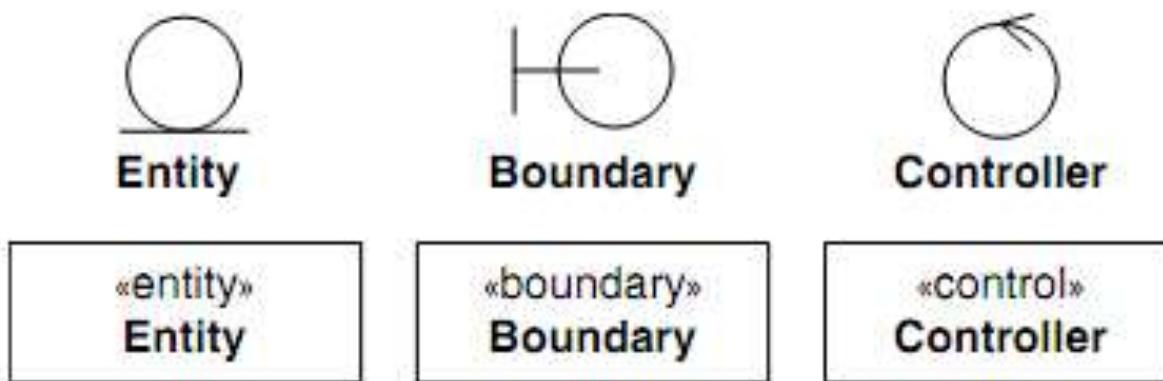


FIGURE 2.6 Visual and textual stereotypes.

UML Standard Stereotypes

Stereotype	UML Element	Description
«call»	Dependency (usage)	Call dependency between operations or classes.
«create»	Dependency (usage)	The source element creates instances of the target element.
«instantiate»	Dependency (usage)	The source element creates instances of the target element. Note: This description is identical to the one of «create».
«responsibility»	Dependency (usage)	The source element is responsible for the target element.
«send»	Dependency (usage)	The source element is an operation and the target element is a signal sent by that operation.
«derive»	Abstraction	The source element can, for instance, be derived from the target element by a calculation
«refine»	Abstraction	A refinement relationship (e.g., between a design element and a pertaining analysis element).
«trace»	Abstraction	Serves to trace of requirements.
«script»	Artifact	A script file (can be executed on a computer).
«auxiliary»	Class	Classes that support other classes («focus»).
«focus»	Class	Classes contain the primary logic. See «auxiliary».
«implementationClass»	Class	An implementation class specially designed for a programming language, where an object may belong to one class only.
«metaclass»	Class	A class with instances that are, in turn, classes.
«type»	Class	Types define a set of operations and attributes, and they are generally abstract.
«utility»	Class	Utility classes are collections of global variables and functions, which are grouped into a class, where they are defined as class attributes/operations.
«buildComponent»	Component	An organizationally motivated component.
«implement»	Component	A component that contains only implementation, no specification.

«buildComponent»	Component	An organizationally motivated component.
«implement»	Component	A component that contains only implementation, no specification.
«framework»	Package	A package that contains Framework elements.
«modelLibrary»	Package	A package that contains model elements, which are reused in other packages.
«create»	Behavioral feature	A property that creates instances of the class to which it belongs (e.g., constructor).
«destroy»	Behavioral feature	A property that destroys instances of the class to which it belongs (e.g., destructor).

FIGURE 2.7 UML standard stereotypes (selection).

Sesión 7 y 8

Casos de Uso

Unidad 3

**Modelado del comportamiento
dinámico del sistema**

Mg. Gustavo G. Delgado Ugarte

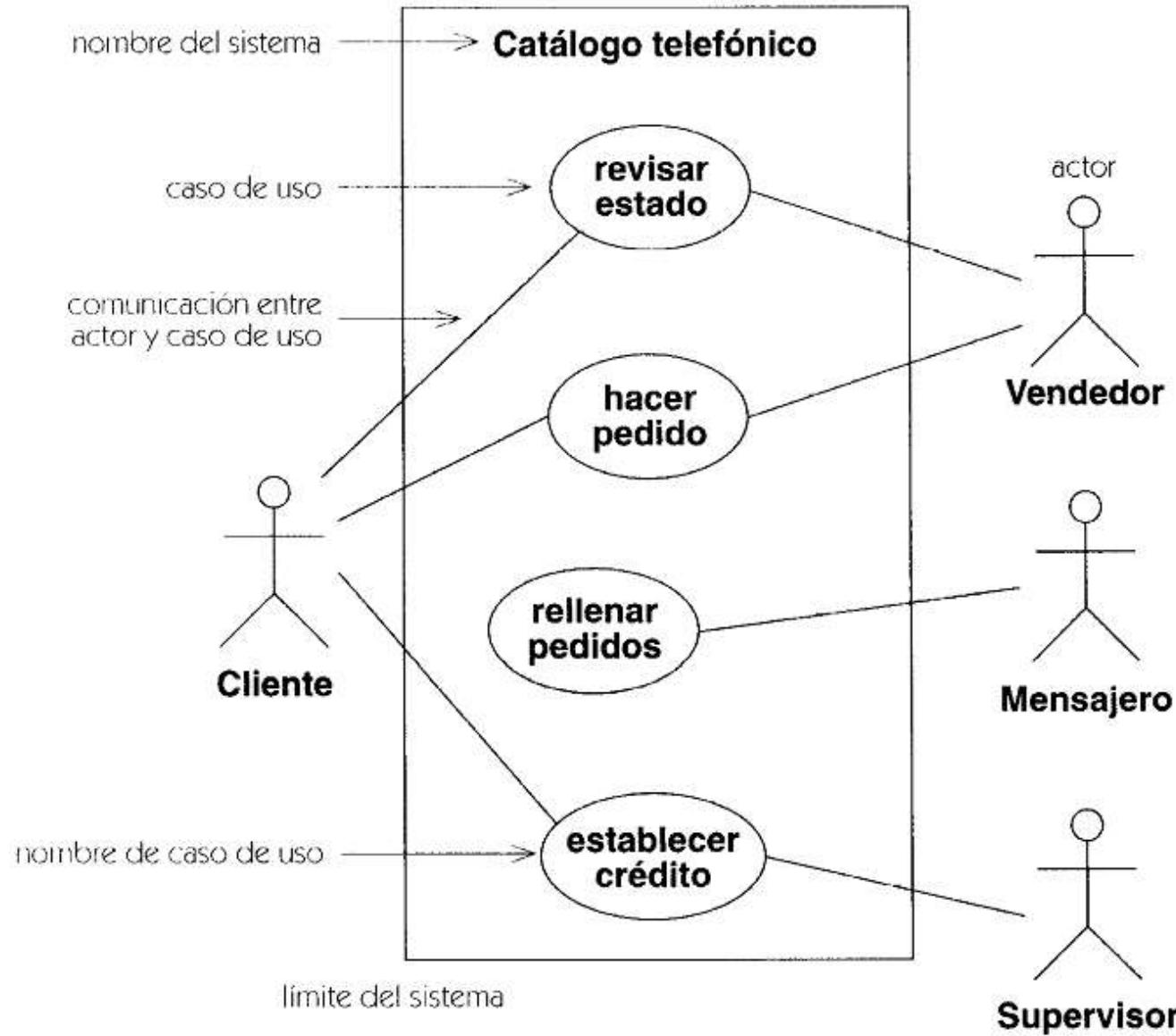
Diagramas de Casos de Uso

- Los diagramas de casos de uso capturan el comportamiento de un sistema, subsistema o clase, tal como se muestra a un usuario exterior
- Reparte la funcionalidad del sistema en transacciones significativas para los actores (usuarios ideales del sistema), llamadas casos de uso

Diagramas de Casos de Uso

- Un caso de uso describe una interacción con los actores como secuencia de mensajes entre el sistema y uno o más actores
- Los actores incluyen seres humanos, sistemas informáticos (software), procesos, hardware
- Ej. Aplicación telefónica de venta por catálogo

Diagramas de Casos de Uso



Actor

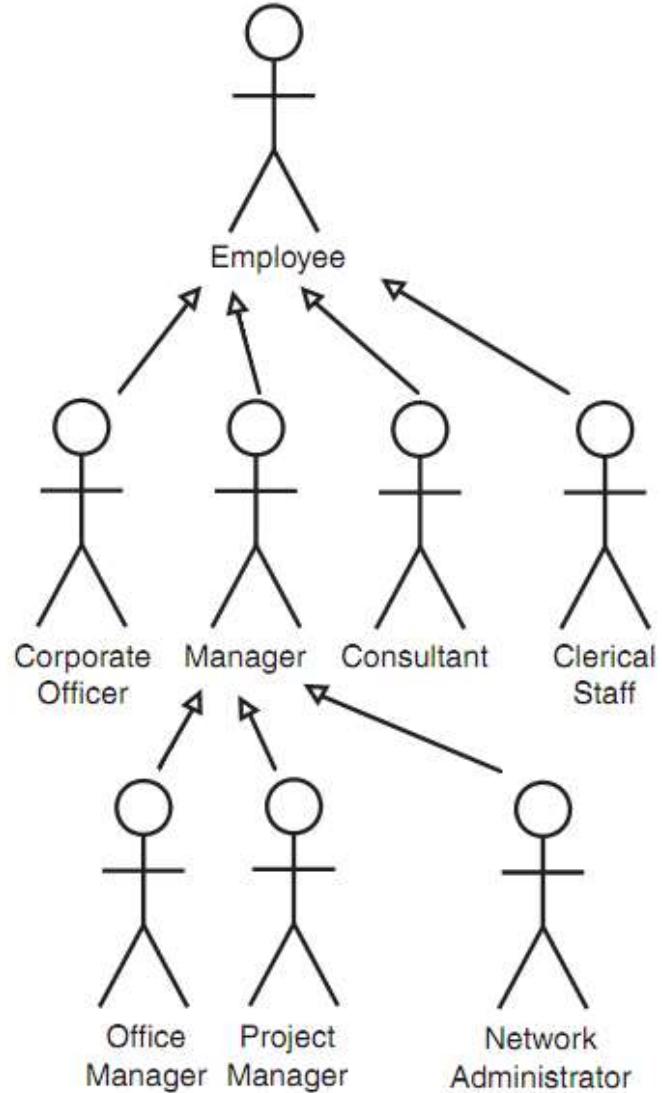
- Es una idealización de
 - Una persona externa (Rol)
 - Un proceso
 - Una cosa
- que interactúa con el sistema, subsistema o clase
- Caracteriza las interacciones que los usuarios exteriores pueden tener con el sistema

Actor

- Un usuario puede estar ligado a múltiples actores
- Diferentes usuarios pueden estar ligados al mismo actor
- Cada actor participa en uno o más casos de uso
 - Interactúa con el caso de uso intercambiando mensajes

Actor

- Los actores pueden ser definidos en jerarquías de generalización, en las cuales una descripción abstracta del actor es compartida y aumentada por una o más descripciones específicas del actor



Actor

- Un actor puede ser
 - Un ser humano (Rol)
 - Otro sistema informático (Software)
 - Certo proceso ejecutable (Software o Hardware)

Caso de Uso

- Es una unidad coherente de funcionalidad, externamente visible, proporcionada por una unidad del sistema y expresada por una secuencia de mensajes intercambiados por la unidad del sistema y uno o más actores
- El propósito es definir una pieza de comportamiento coherente, sin revelar la estructura interna del sistema
 - El qué pero no el cómo

Caso de Uso

- La definición del caso de uso incluye todo el comportamiento
 - Líneas principales (comportamiento normal)
 - Variaciones sobre el comportamiento normal y Condiciones excepcionales junto con la respuesta deseada
- La ejecución de cada caso de uso es independiente de las demás

Caso de Uso

- La dinámica de un caso de uso se puede especificar por
 - Diagramas de estado
 - Diagramas de secuencia
 - Diagramas de colaboración
 - Descripciones informales de texto
- Un caso de uso puede participar en varias relaciones

Relaciones

Tabla 5.1 Tipos de Relaciones de Casos de Uso

<i>Relación</i>	<i>Función</i>	<i>Notación</i>
asociación	La línea de comunicación entre un actor y un caso de uso en el que participa	_____
extensión	La inserción de comportamiento adicional en un caso de uso base que no tiene conocimiento sobre él	« extend » — — — ➤
generalización de casos de uso	Una relación entre un caso de uso general y un caso de uso más específico, que hereda y añade propiedades a aquél	→ ▶
inclusión	Inserción de comportamiento adicional en un caso de uso base, que describe explícitamente la inserción	« include » — — — ➤

Relación de Inclusión

- Un caso de uso puede incorporar el comportamiento de otros casos de uso como fragmentos de su propio comportamiento
- La relación de inclusión (<<include>>) es usada para integrar un caso de uso dentro de otro, convirtiéndose en parte lógica de ese caso de uso

Relación de Inclusión

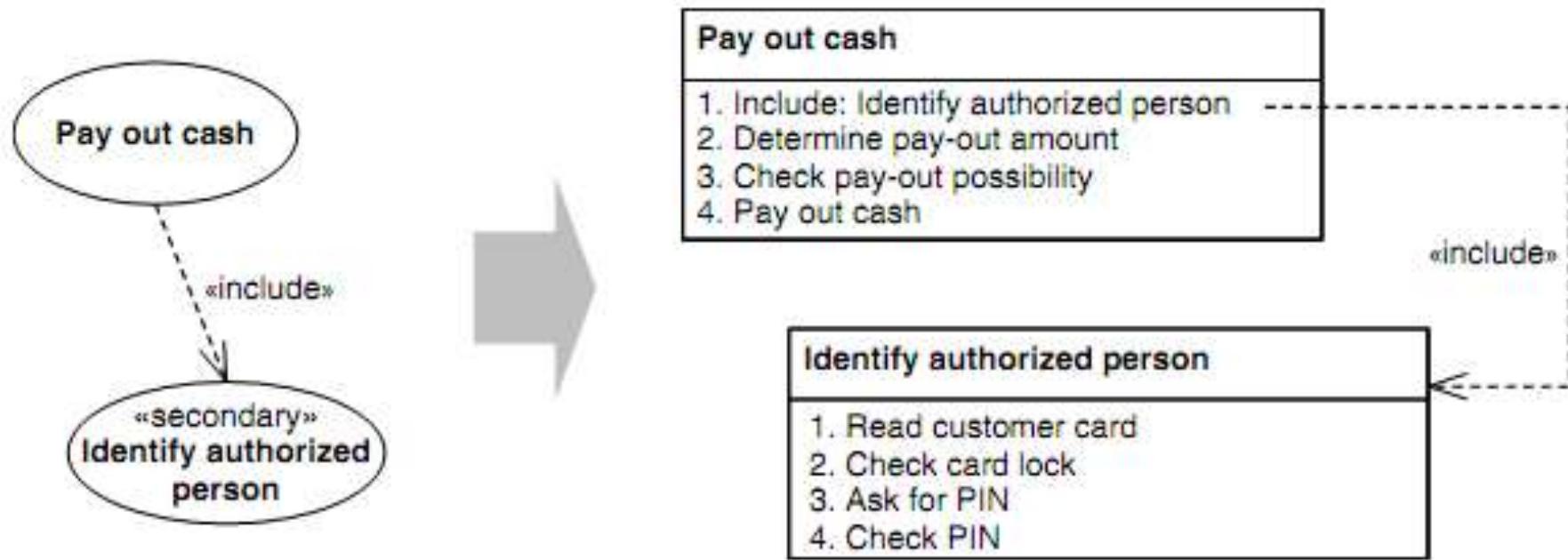


FIGURE 2.123 An include relationship in a use case diagram and its meaning in the use case description (Note: The right side is not UML).

Relación de Extensión

- Un caso de uso se puede también definir como una extensión incremental de un caso de uso base
- La relación Extensión (<<extend>>) es utilizada para expresar que un caso de uso será extendido por otro caso de uso en ciertas circunstancias y en cierto punto; el cuál es llamado el *punto de extensión*

Relación de Extensión

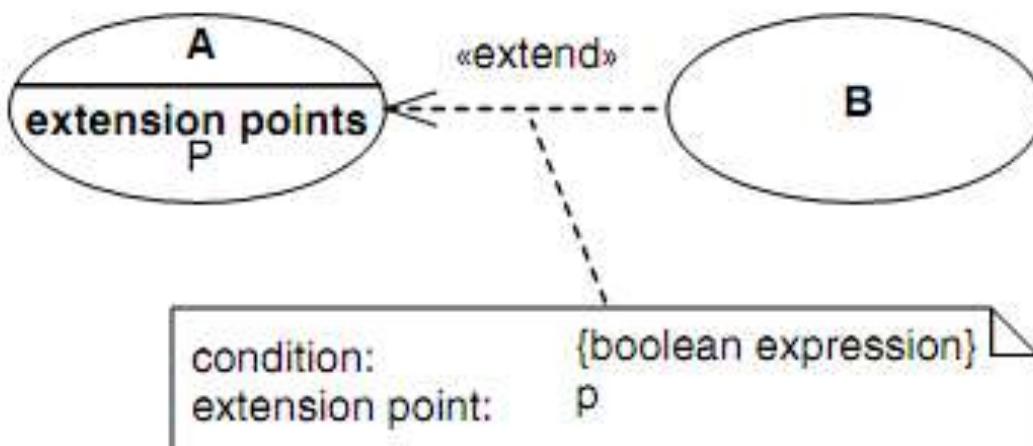
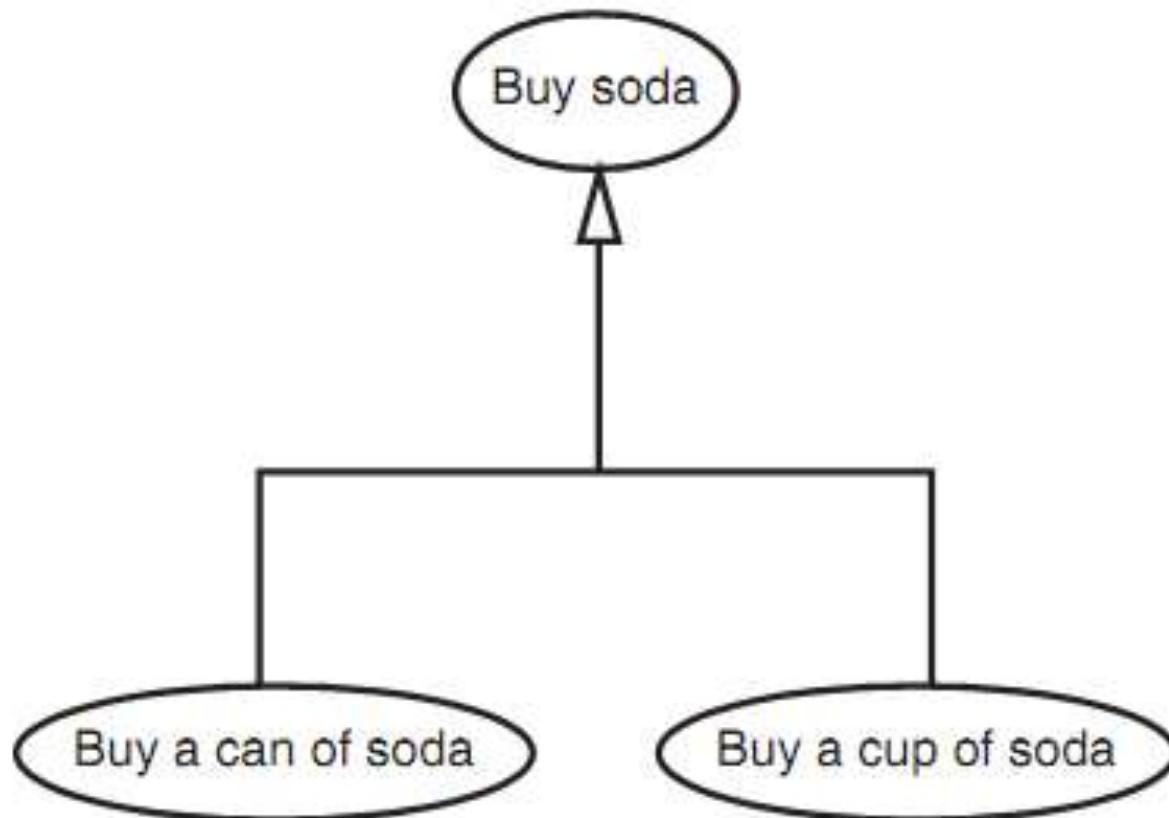


FIGURE 2.124 Notation for the extend relationship.

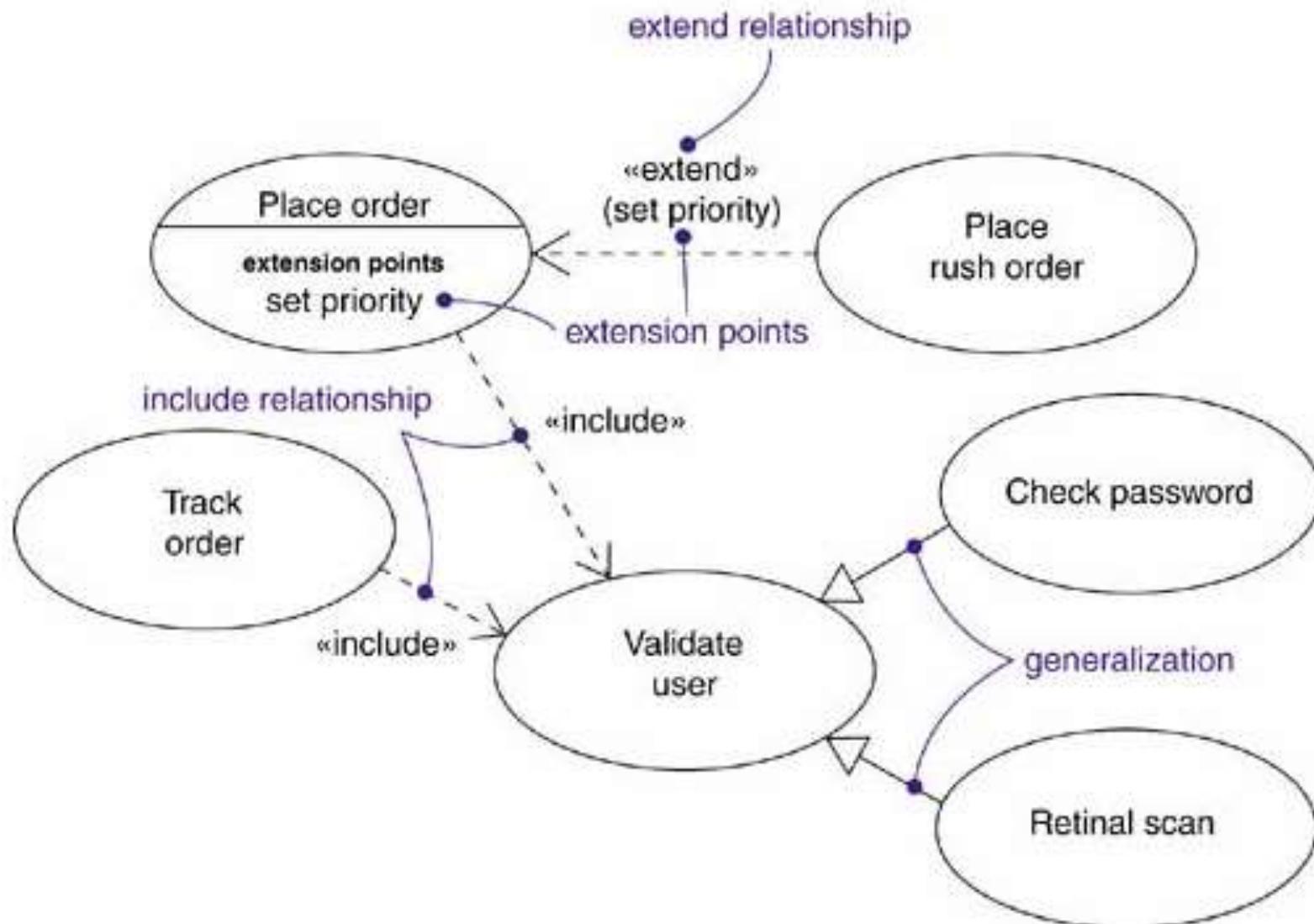
Relación de Generalización

- Un caso de uso también se puede especializar en uno o más casos de uso hijos (generalización de casos de uso)
- Cualquier caso de uso hijo se puede utilizar en una situación en la cual se espera al caso de uso padre

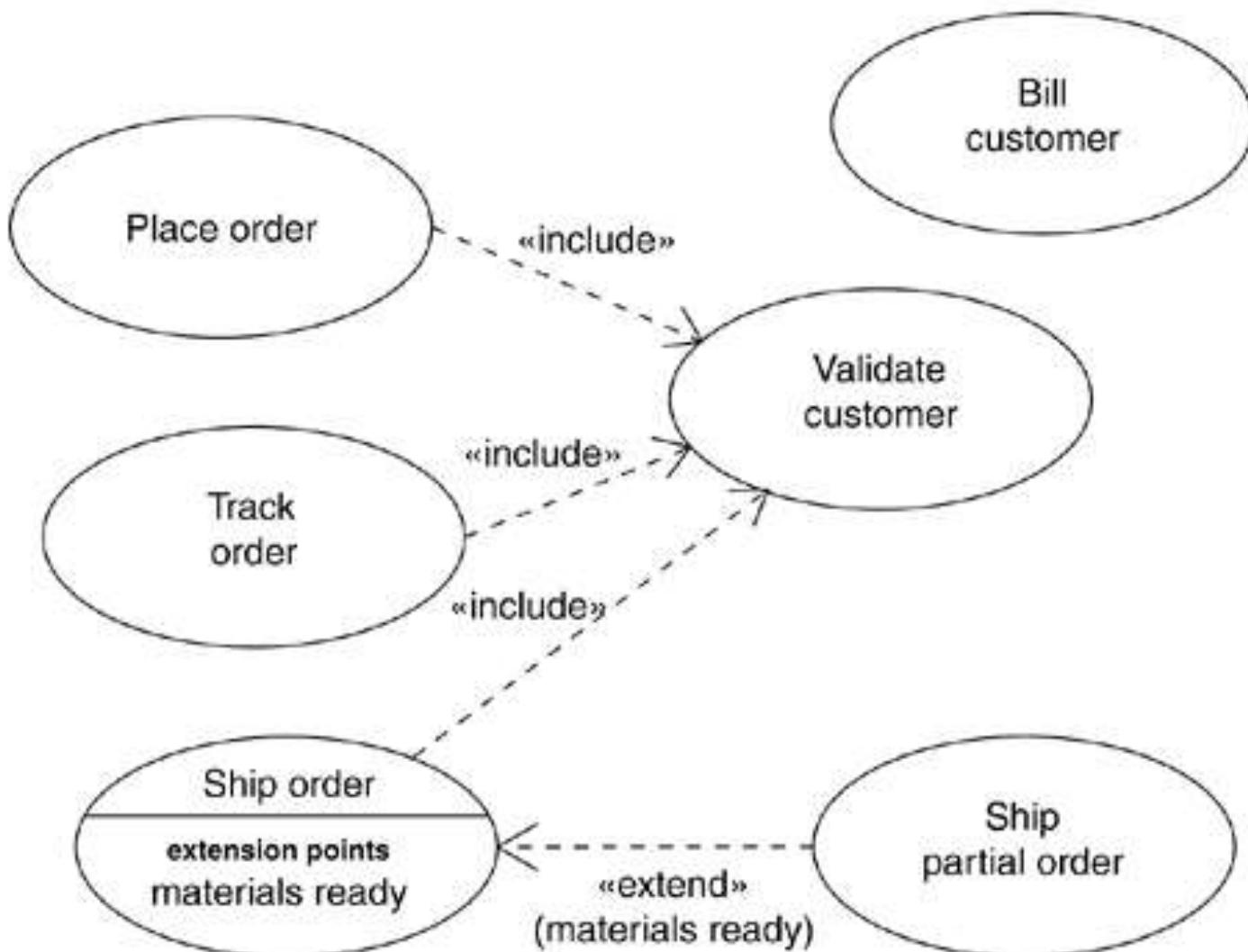
Relación de Generalización



Inclusión, extensión y generalización



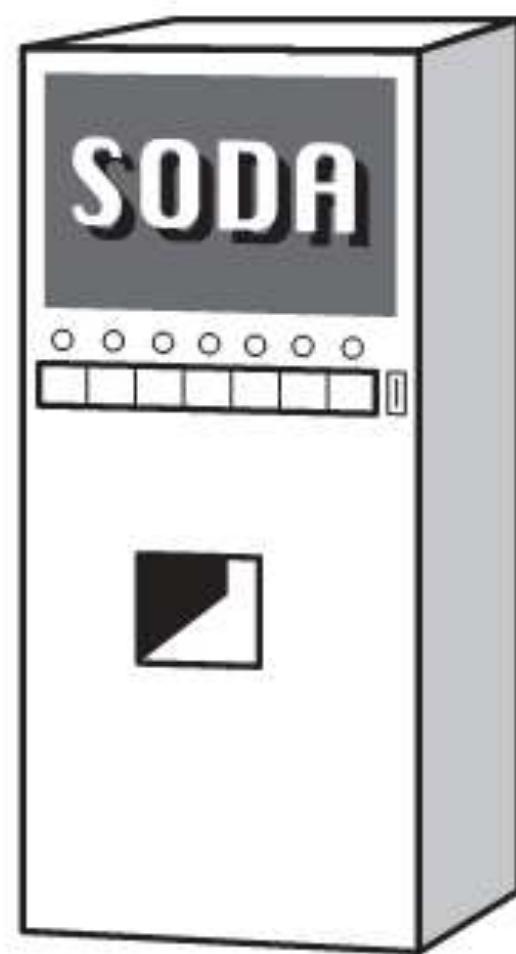
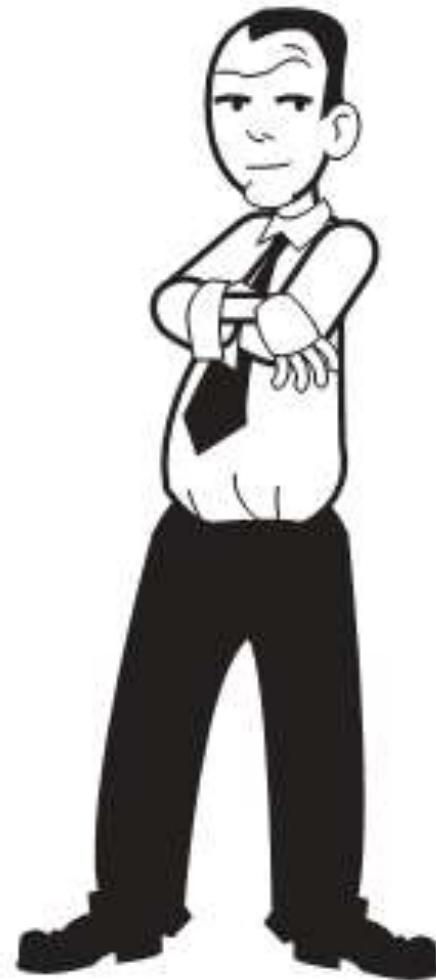
Inclusión, extensión y generalización



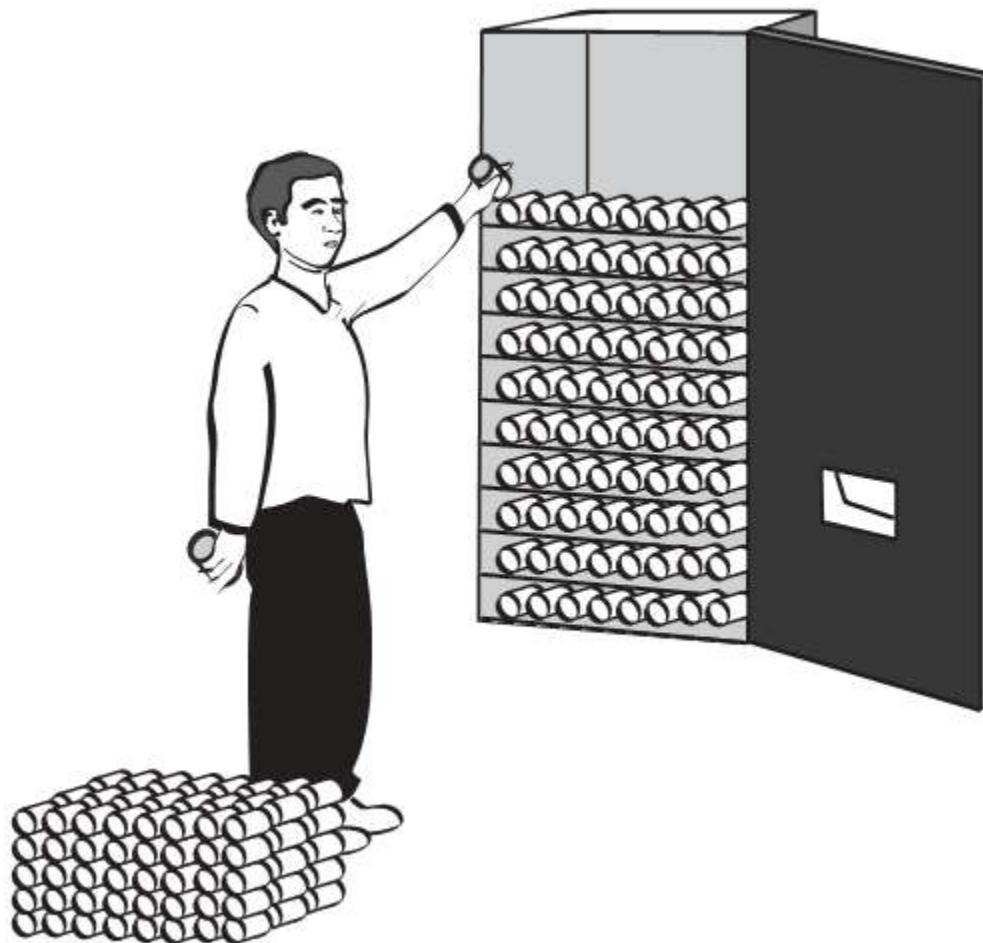
PRÁCTICA DIRIGIDA

“LA MÁQUINA DE GASEOSAS”

Comprar Gaseosa



Reaprovisionar gaseosas



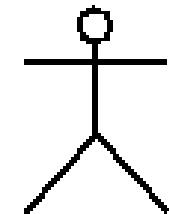
Recoger dinero



Elementos del Modelo

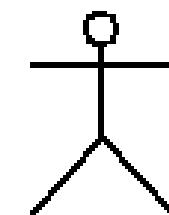
- Actores
 - Cliente
 - Reponedor
 - Recaudador
- Casos de Uso
 - Comprar Gaseosa
 - Reaprovisionar Gaseosas
 - Recoger dinero

Maquina de Gaseosas



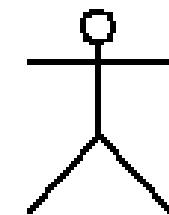
Cliente

Comprar gaseosa



Reponedor

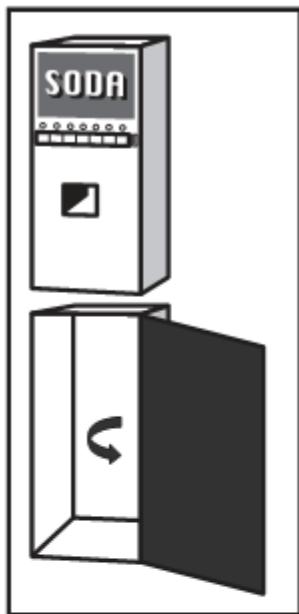
Reponer gaseosas



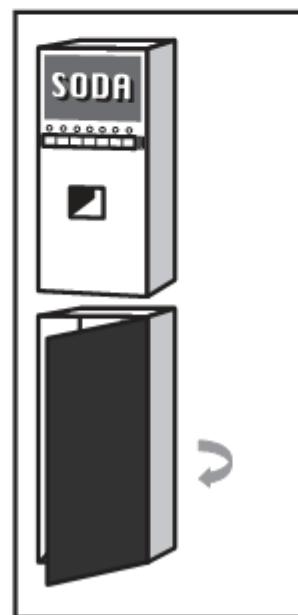
Recaudador

Recoger dinero

Inclusión



Expose the inside



Unexpose the inside

Maquina de Gaseosas

