

ANÁLISIS Y DISEÑO ORIENTADO A OBJETOS CON UML

Auditorio y Objetivos

■ Dirigido a

- Personas con la necesidad de aprender las características y métodos de la tecnología de objetos, principalmente aquellas que desarrollan sistemas complejos.

■ Objetivos

- Al finalizar el curso, los participantes podrán:
 - Explicar el proceso de desarrollo iterativo e incremental
 - Definir los requerimientos de un sistema desde el punto de vista del usuario
 - Crear un modelo orientado a objetos del comportamiento y de los aspectos estructurales de los requerimientos de un sistema
 - Crear una arquitectura lógica de un sistema
 - Diseñar un sistema aplicando los conceptos de abstracción, encapsulamiento, herencia, polimorfismo y patrones

Contenido

- Introducción
 - Antecedentes del análisis y diseño orientado a objetos y UML
- Desarrollo Iterativo e Incremental
 - Ciclo de vida del desarrollo de sistemas por medio de una aproximación iterativa e incremental
- Comportamiento del Sistema
 - Análisis de requerimientos a través de Casos de Uso (Use case)
 - Desarrollo de escenarios
- Objetos y Clases
 - Definición de objetos, clases, estereotipos y paquetes

Contenido

- Interacción de Objetos
 - Representación gráfica del escenario
- Definición de Clases
 - La aplicación del análisis de Casos de Uso para definir clases en el sistema
 - Definición de paquetes
 - Creación de diagramas de clase
- Relaciones
 - Definición de relaciones necesarias para la interacción de objetos
- Operaciones y Atributos
 - Definición de la estructura y comportamiento de una clase

Contenido

- Herencia
 - Aplicación de los principios de generalización y especialización para definir relaciones de superclase/subclase
- Comportamiento de Objetos
 - Desarrollo de diagramas de transición de estado para mostrar gráficamente el comportamiento de un objeto
- Homogeneización
 - Mezclar clases descubiertas en diferentes Casos de Uso
- Arquitectura
 - Discusión de la Arquitectura "4+1" Vistas

Contenido

- Mecanismos Clave
 - Discusión de estrategias de mecanismos clave
 - Designación de clases
 - Diseño de la interfaz de usuario
 - Incorporación de patrones
- Diseño de relaciones
 - Soporte C++ para relaciones
- Diseño de Atributos y Operaciones
 - Soporte C++ para atributos y operaciones
- Diseño para Herencia
 - Soporte C++ para herencia

Contenido

- Resumen
 - Resumen del curso de análisis y diseño
- Lectura recomendadas
 - Lista de libros
- Planteamiento del Problema de Nómina
 - Requerimientos para un sistema de nómina
- Solución del Problema Nómina
 - Elaboración del análisis y diseño para el problema de nómina

Introducción



Objetivos: Introducción

- Usted será capaz de:
 - Explicar la crisis del software
 - Discutir el poder de la tecnología de objetos
 - Discutir dónde puede emplearse la tecnología OO
 - Definir análisis y diseño
 - Explicar el origen del UML

La Crisis del Software

- El Departamento de Vehículos a Motor de California invirtió más de \$43 mdd en un sistema que fusionaba los Sistemas de Licencias a Conductores del estado y el Registro de Vehículos
 - El sistema fue abandonado sin haberlo usado
- American Airlines realizó sin éxito un esfuerzo de \$165 mdd para ligar su software de reservación de vuelos con los sistemas de reservación de Marriott, Hilton y Budget
- El sistema de control de equipaje del Aeropuerto de Denver costó millones de dólares, sobretodo debido al retraso en la abertura del aeropuerto

Fallas de software reportadas por W. Wayt Gibbs en el numero de Septiembre de 1994 de la revista *Scientific American*

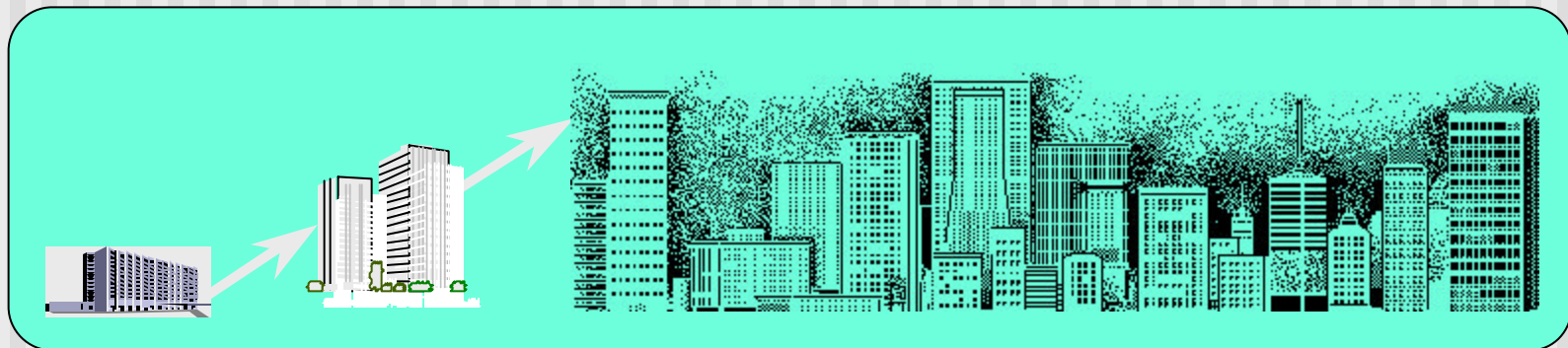
Algunos ejemplos extremos, PERO hay varios desastres similares en escala menor

La Crisis del Software (cont.)

- En Marzo de 1989, Arthur Andersen reportó
 - Más de \$300 mil mdd por año invertidos en actividades de software comercial en los Estados Unidos
 - Sólo el 8% del software entregado brinda resultados y funciona
- ¿Cuáles son las razones de la crisis del software?
 - Constantes cambios en los requerimientos
 - Fallas en el manejo de riesgos
 - Complejidad del software

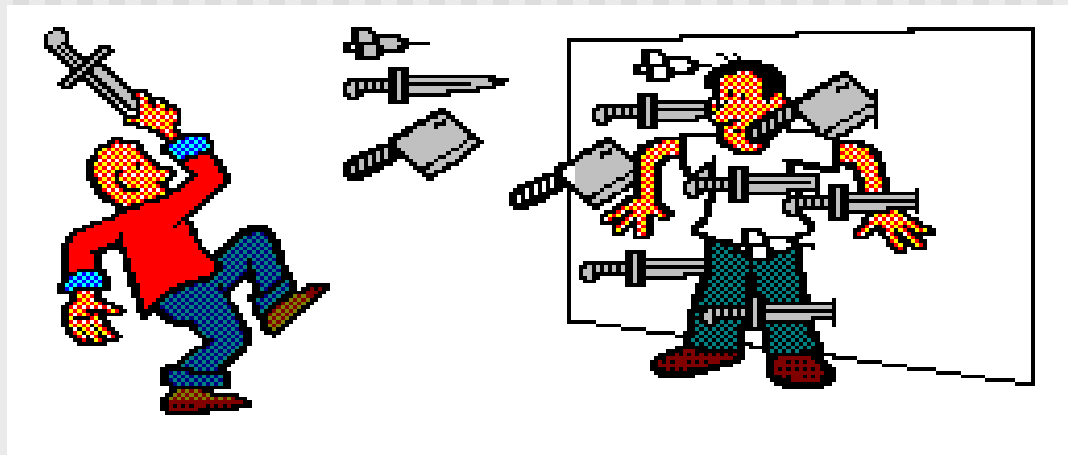
Cambios en los requerimientos

- Los requerimientos del negocio se definen en ciclos de desarrollo más pequeños
 - Los usuarios esperan más en términos de flexibilidad
- Los requerimientos iniciales generalmente están pobremente definidos



Fallas en el Manejo de Riesgos

- El ciclo de vida en cascada puede retrasar la identificación del problema
- No hay prueba de que el sistema funcionará, sino hasta el final del ciclo de vida
- El resultado, el máximo riesgo



Complejidad del Software

- Está creciendo la demanda de software de negocios
- Nadie entiende el sistema en su totalidad
- Deben mantenerse los sistemas anteriores, pero los desarrolladores de los mismos ya se han ido

Poder de la Tecnología de Objetos

- Un paradigma único
 - Los usuarios, analistas, diseñadores e programadores utilizan el mismo lenguaje
- Facilita la re-utilización de arquitectura y código
- Los modelos reflejan de manera más cercana al mundo real
 - Describe con mayor precisión los procesos y datos
 - Descomposición basada en partición natural
 - Más fácil de entender y mantener
- Estabilidad
 - Un cambio en los requerimientos no significa cambios masivos en el sistema en desarrollo

Ejemplo: Ventas

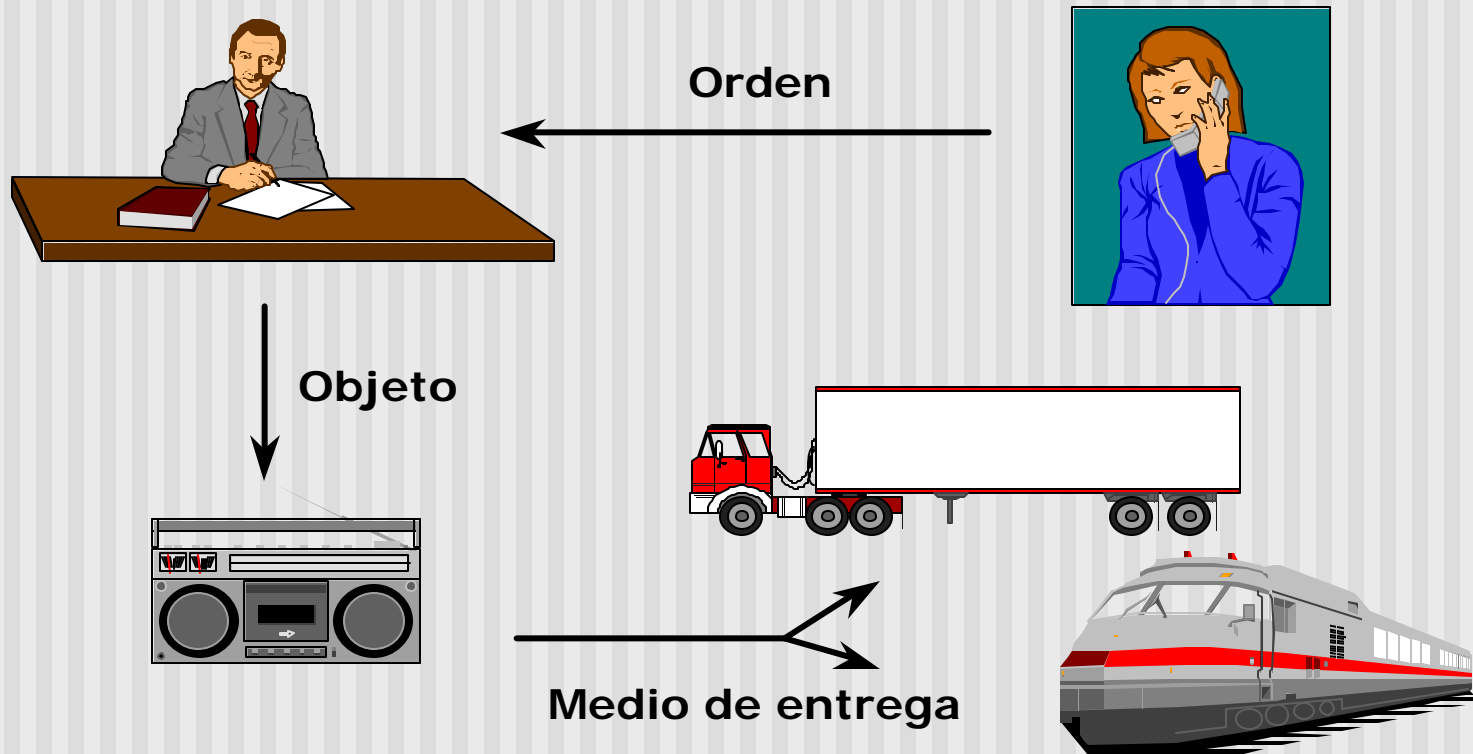
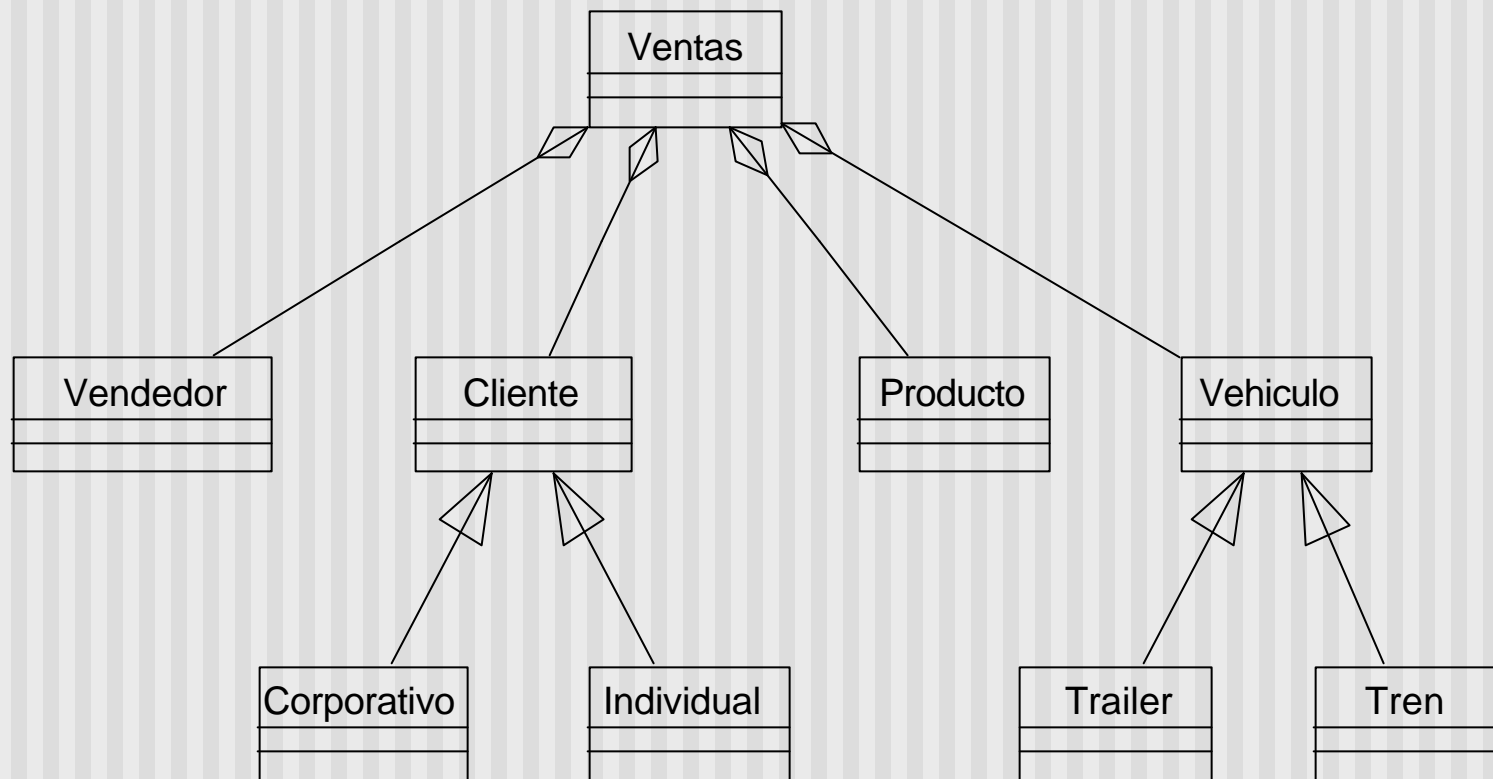
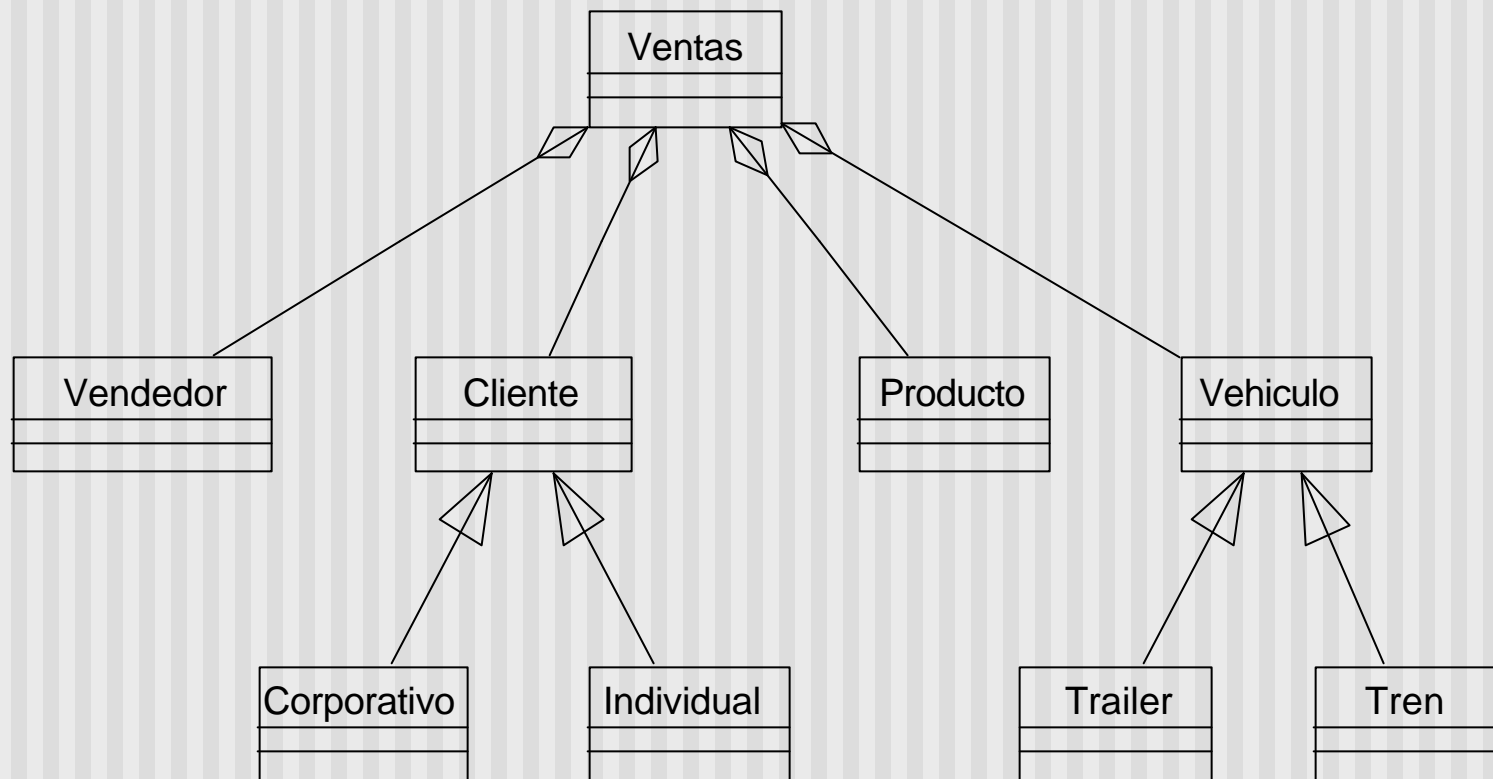


Diagrama de Clases de Ventas

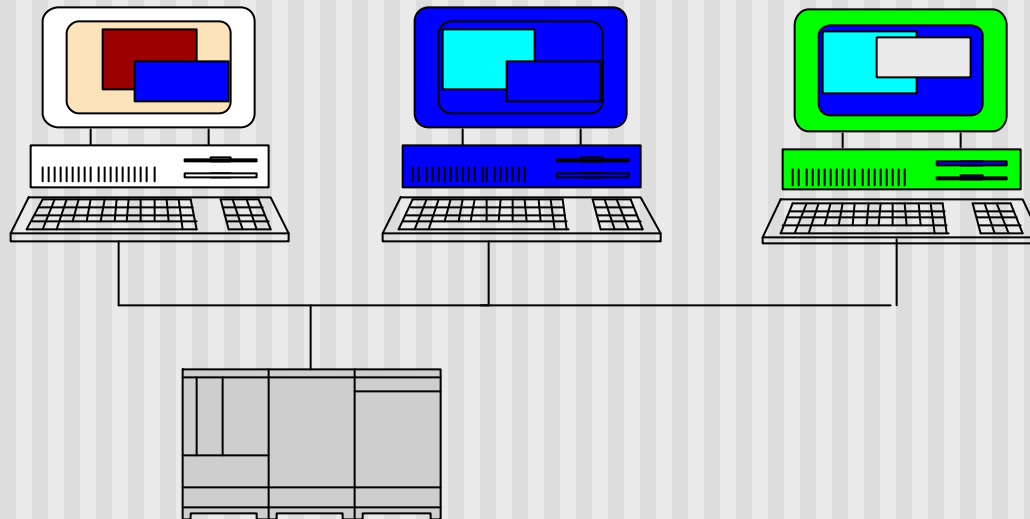


Efecto en el Cambio de Requerimientos



¿Dónde se está usando OO?

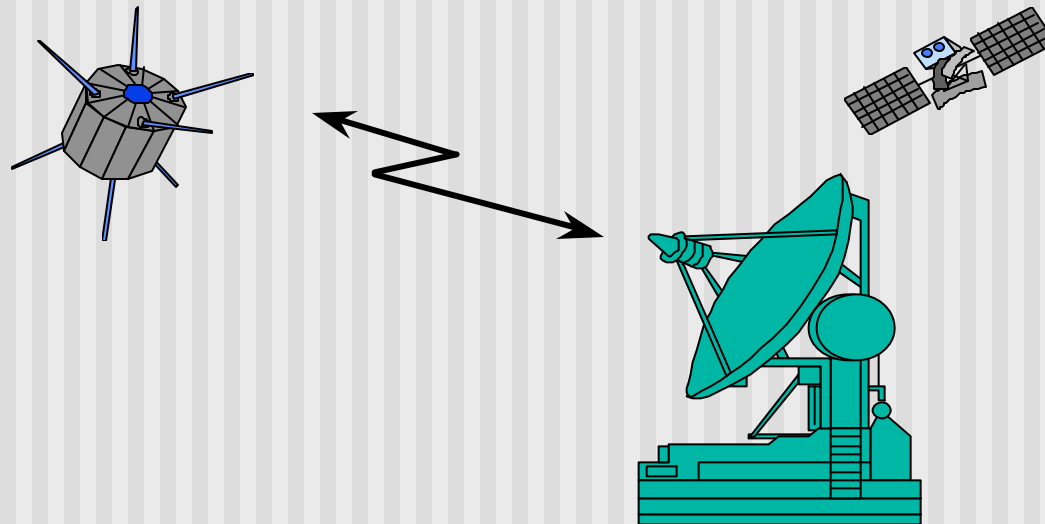
- Sistemas basados en GUI
 - La metodología OO facilita el diseño e implementación de sistemas con interfaz gráfica de usuario (GUI)



¿Dónde se está usando OO?

■ Sistemas Inmersos

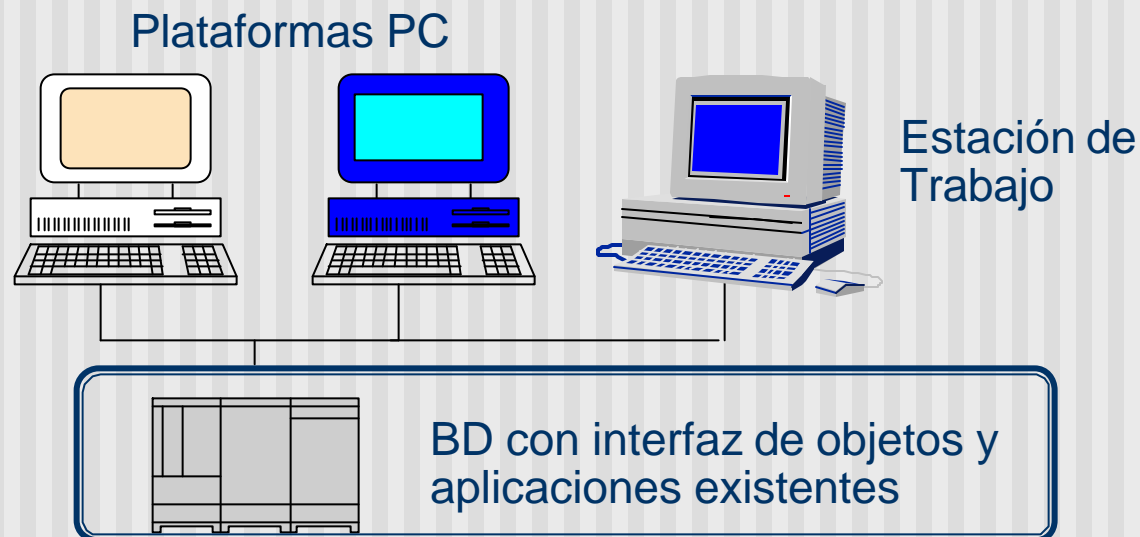
- Los métodos OO permiten desarrollar sistemas inmersos y de tiempo real con mayor calidad y flexibilidad



¿Dónde se está usando OO?

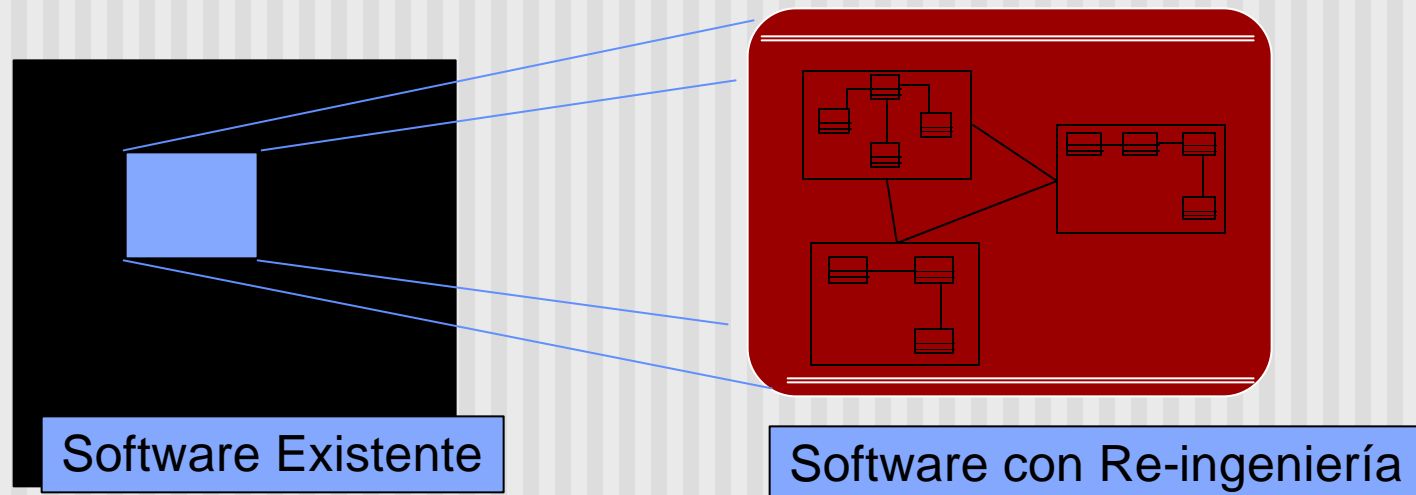
■ Cómputo Cliente/Servidor

- La metodología OO puede encapsular información de mainframe en objetos, permitiendo obtener aplicaciones pequeñas



¿Dónde se está usando OO?

- En la Re-ingeniería
 - Los métodos OO permiten hacer re-ingeniería a partes del sistema, protegiendo las inversiones hechas en aplicaciones de software existentes



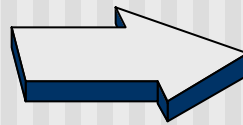
Análisis y Diseño Orientado a Objetos

OOA

*Desarrollo del modelo
de requerimientos*



Perspectiva del Usuario



OOD

*Agregar decisiones de
diseño y de detalle*

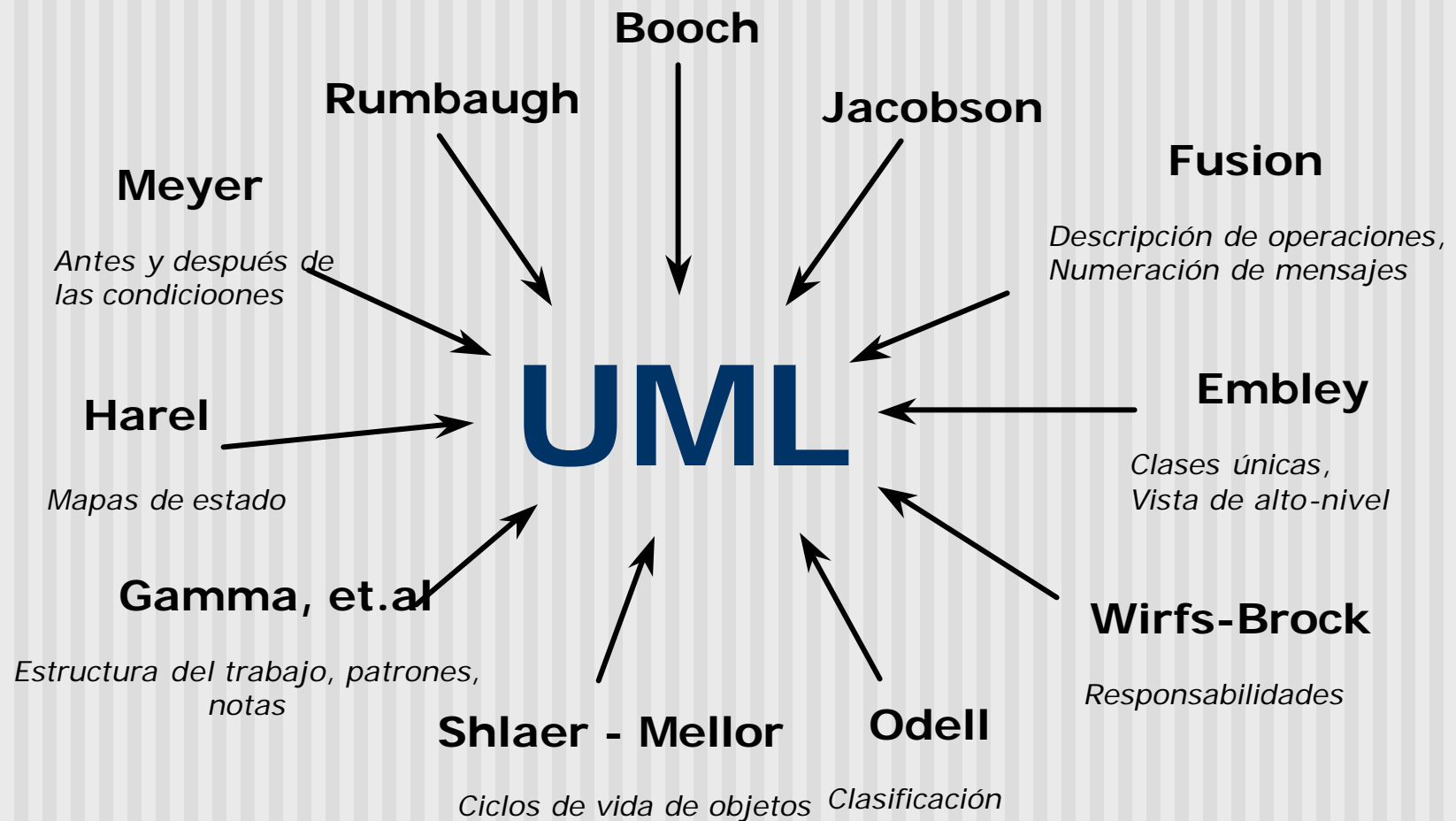


Perspectiva del Desarrollador

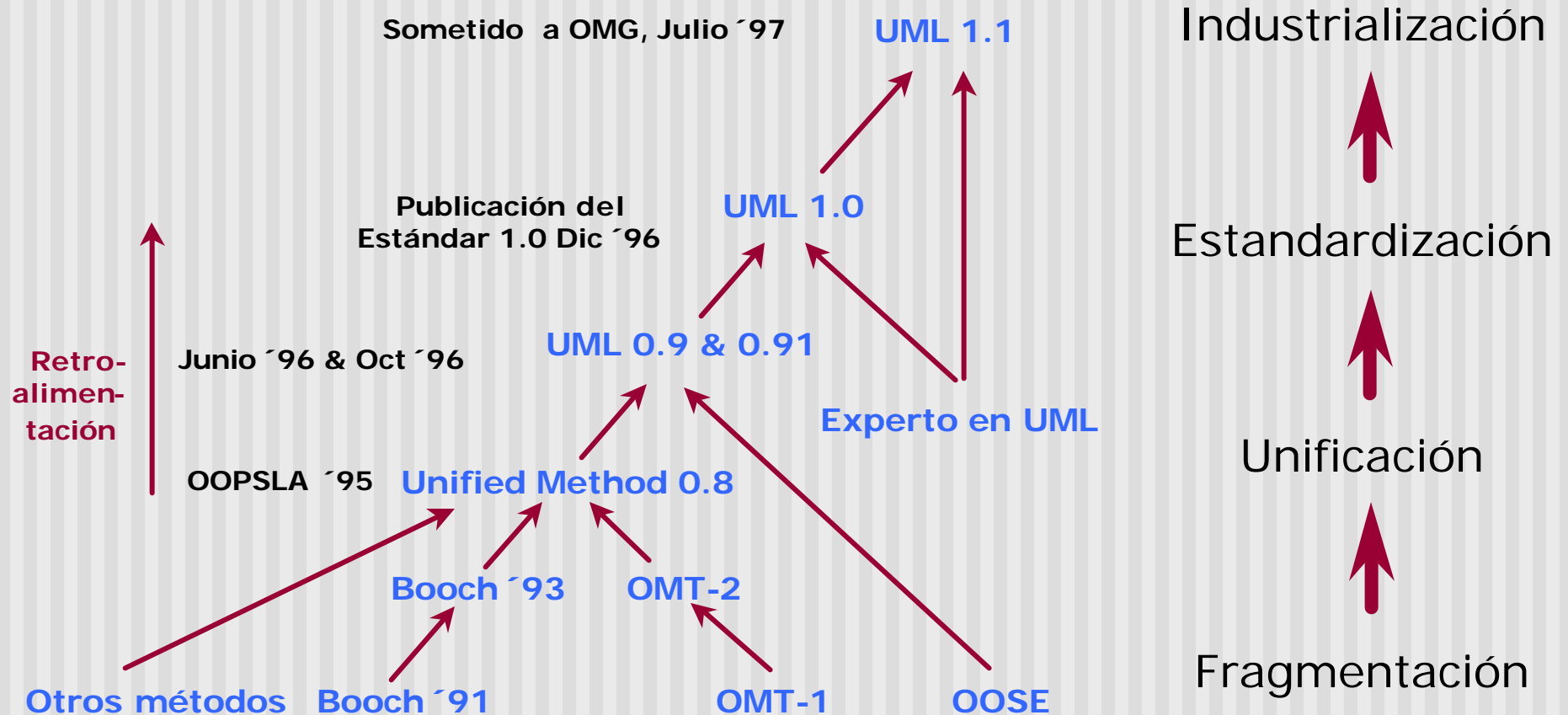
¿Qué es UML?

- El Lenguaje de Modelado Unificado (Unified Modeling Language, UML) es descrito en “The Unified Modeling Language for Object-Oriented Development” escrito por Grady Booch, Jim Rumbaugh, e Ivar Jacobson
 - Disponible en <http://www.rational.com>
- Basado en las experiencias personales de los autores
- Incorpora contribuciones de otras metodologías
- Sometido a aprobación a la OMG por Rational Software, Microsoft, Hewlett-Packard, Oracle, Texas Instruments, MCI Systemhouse y otros

Entradas al UML



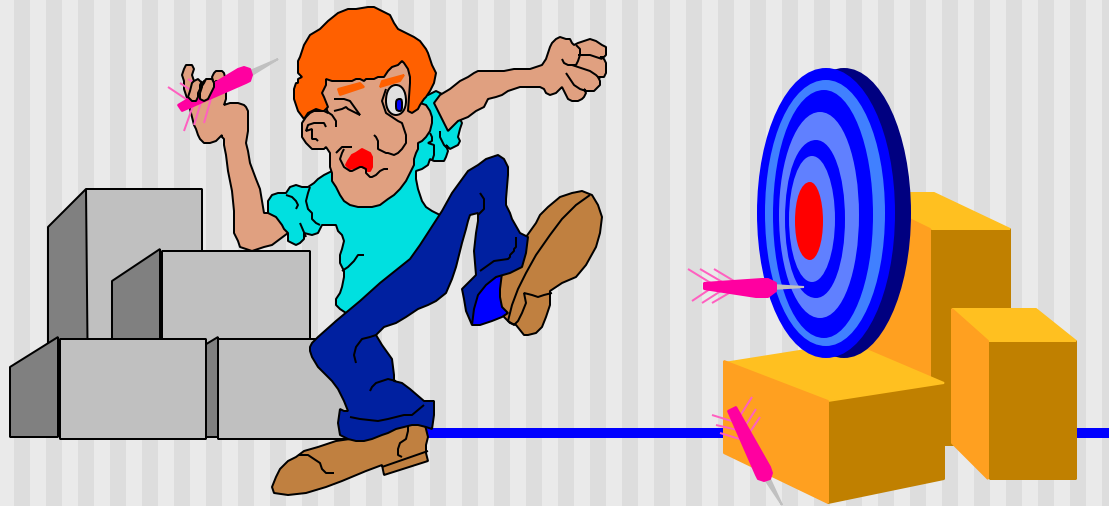
Evolución del UML



Beneficios de UML

- Ofrece un proceso de modelado sin fallas durante el análisis, para diseñar la implementación
- Define una notación expresiva y consistente
 - Facilita la comunicación con otros
 - Ayuda a señalar omisiones e inconsistencias
 - Soporta tanto análisis como diseño de pequeños y grandes sistemas

Desarrollo Iterativo e Incremental



Objetivos: Desarrollo Iterativo e Incremental

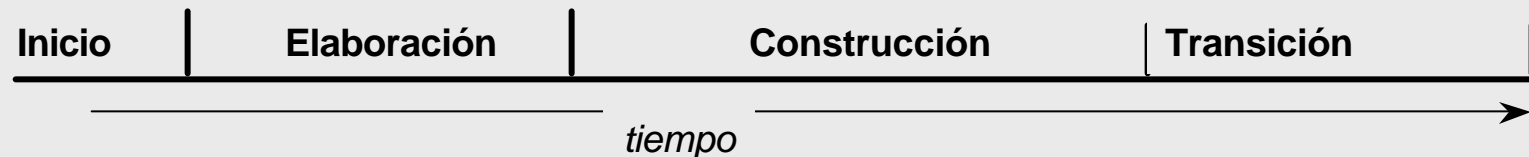
- Usted podrá:
 - Definir un proceso de desarrollo iterativo e incremental
 - Listar las fases, los productos y las actividades principales para cada fase de un proceso de desarrollo iterativo e incremental
 - Definir una iteración y listar sus actividades

¿Qué es Desarrollo Iterativo e Incremental?

- Desarrollo iterativo e incremental es el proceso de construir sistemas de software en pasos pequeños
- Beneficios
 - Reducción del riesgo basándose en la retroalimentación temprana
 - Mayor flexibilidad para acomodar requerimientos nuevos o cambios en los mismos
 - Incremento de la calidad del software

Ciclo de vida del Software

- El ciclo de vida del software se divide en una serie de ciclos de desarrollo, donde la salida de un ciclo de desarrollo es la generación de un producto de software
- Cada ciclo es una sucesión de fases
 - Inicio
 - Elaboración
 - Construcción
 - Transición



Fase de Inicio

- Propósito:
 - Establecer casos de uso de un sistema nuevo o para la actualización importante de un sistema existente
- Productos requeridos:
 - Requerimientos esenciales para el proyecto
 - Valoración del riesgo inicial
- Productos opcionales:
 - Un prototipo conceptual
 - Un modelo inicial del dominio (avance de un 10% - 20%)

Fase de Elaboración

- Propósito
 - Analizar el dominio del problema
 - Establecer una base arquitectónica sólida
 - Manejar los elementos de mayor riesgo del proyecto
 - Desarrollar un plan comprensivo que muestre como se completará el proyecto

Fase de Elaboración (cont.)

■ Productos

- Un modelo del comportamiento del sistema, que incluya el contexto del sistema, escenarios y un modelo del dominio (avance de un 80%)
- Una arquitectura de ejecutables
- Una visión del producto base de acuerdo al modelo de dominio
- Una valoración revisada del riesgo
- Un plan de desarrollo
- Criterios de evaluación
- Publicar descripciones
- Un manual de usuario preliminar (opcional)
- Estrategia de prueba
- Plan de pruebas

Fase de Construcción

- Propósito
 - Desarrollar un producto de software completo, de forma incremental, que esté en transición a la comunidad de usuarios
- Productos
 - Una serie de ejecutables liberados
 - Prototipos de comportamiento
 - Resultados que aseguren calidad
 - Documentación del sistema y del usuario
 - Plan de desarrollo
 - Criterio de evaluación para al menos la siguiente iteración

Fase de Transición

■ Propósito

- Hacer la transición del producto de software a la comunidad de usuario

■ Productos

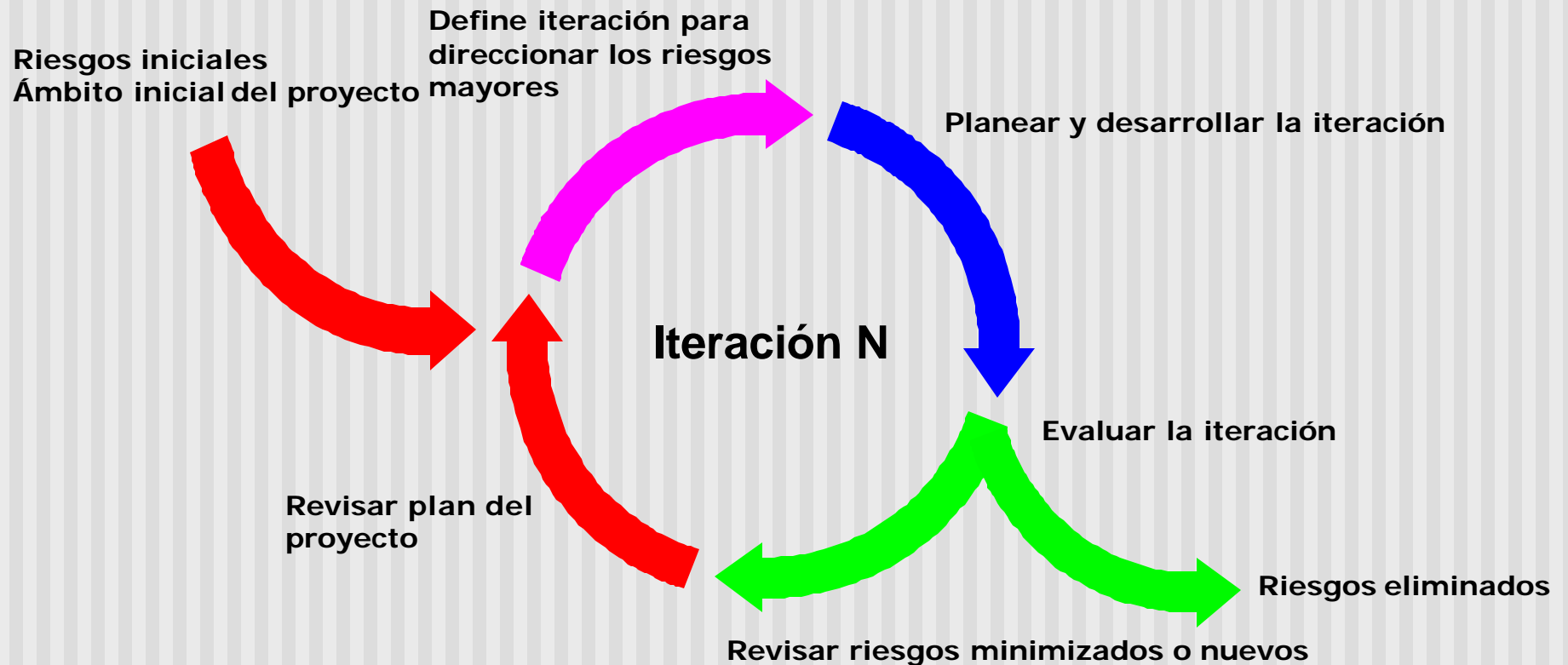
- Una serie de ejecutables liberados
- Resultados que aseguren calidad
- Documentación del sistema y del usuario actualizados
- Análisis “postmortem” del desempeño del proyecto

¿Qué es una Iteración?

- Una iteración es un loop o ciclo de desarrollo que desemboca en la liberación de un subconjunto del producto final
- Cada iteración pasa a través de todos los aspectos del desarrollo del software
 - Análisis de requerimientos
 - Diseño
 - Implementación
 - Pruebas
 - Documentación
- Cada liberación iterativa es una “pieza” totalmente documentada del sistema final

Iteración Preliminar	Iteración de Arquitectura	Iteración de Arquitectura	Iteración de Desarrollo	Iteración de Desarrollo	Iteración de Desarrollo	Iteración de Transición	Iteración de Transición
----------------------	---------------------------	---------------------------	-------------------------	-------------------------	-------------------------	-------------------------	-------------------------

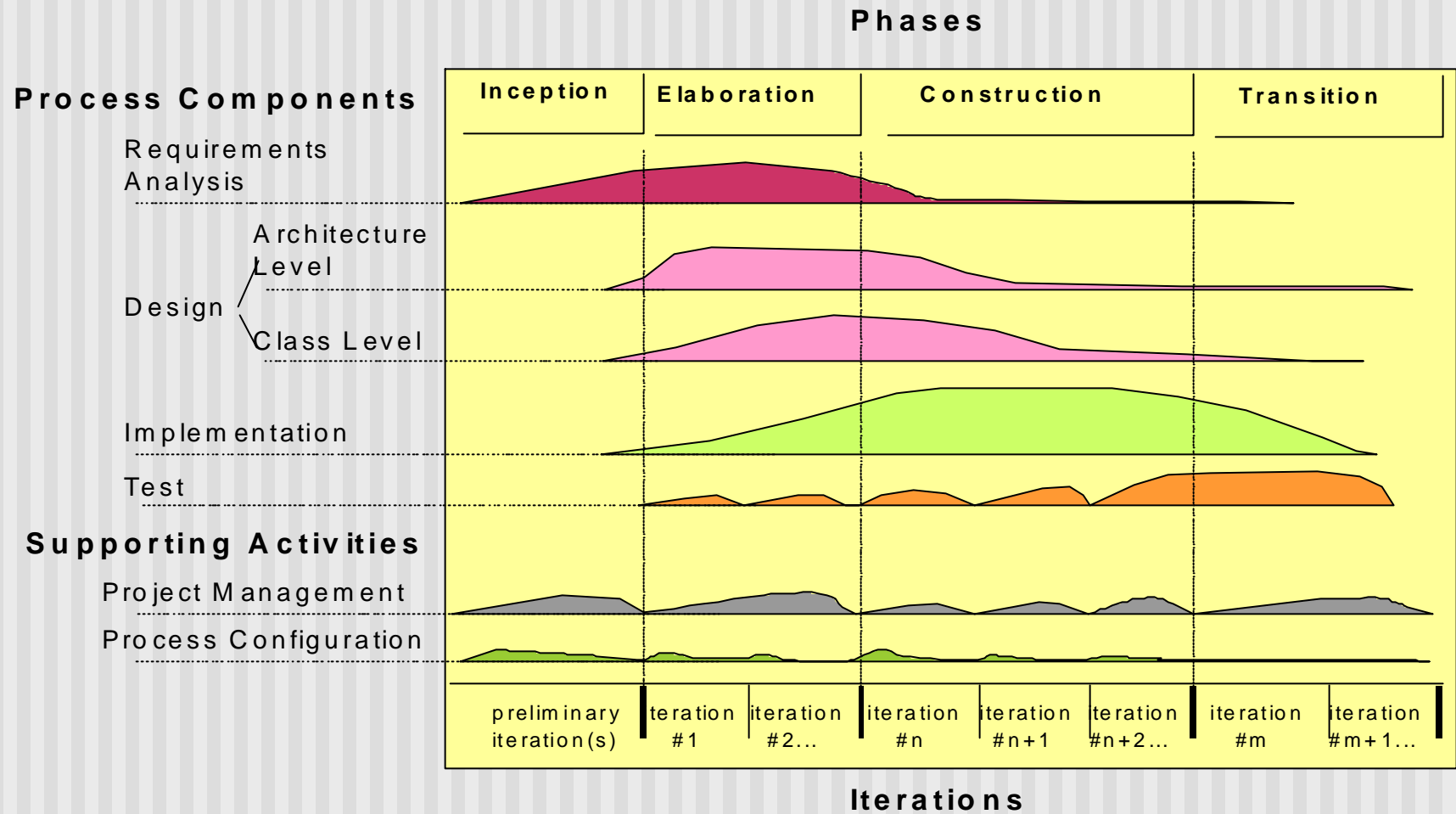
Reducción del Riesgo a través de Iteraciones



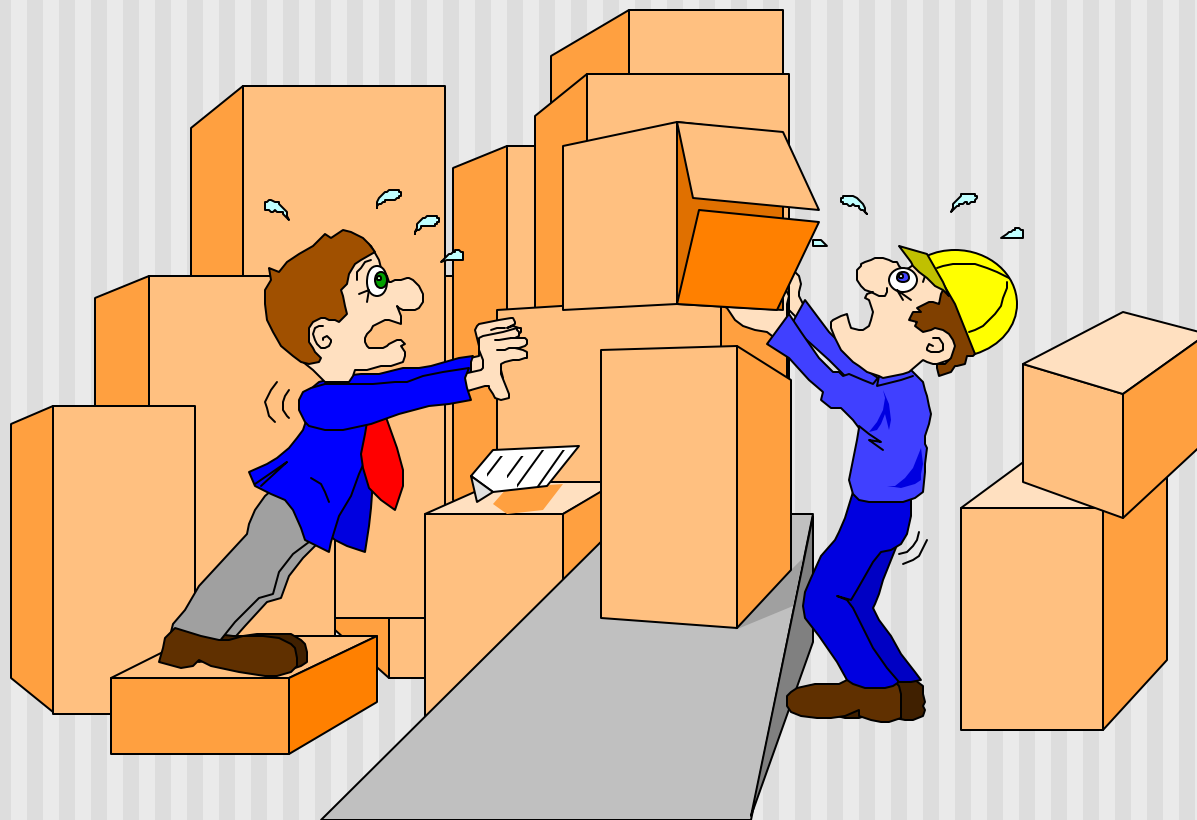
Planeación de Iteraciones

- Identificar y asignar prioridades a los riesgos del proyecto
- Seleccionar un pequeño número de escenarios que ejemplifiquen los riesgos de mayor prioridad
- Los escenarios seleccionados son utilizados por:
 - **Los desarrolladores, para identificar lo que se va a implementar en la iteración**
 - **Los evaluadores, para desarrollar planes y procedimientos de prueba para la iteración**
- Al final de la iteración
 - **Determinar los riesgos que han sido reducidos o eliminados**
 - **Determinar la posibilidad de nuevos riesgos descubiertos**
 - **Actualización del plan de iteraciones siguientes**

Reunión de todos los elementos



Comportamiento del Sistema



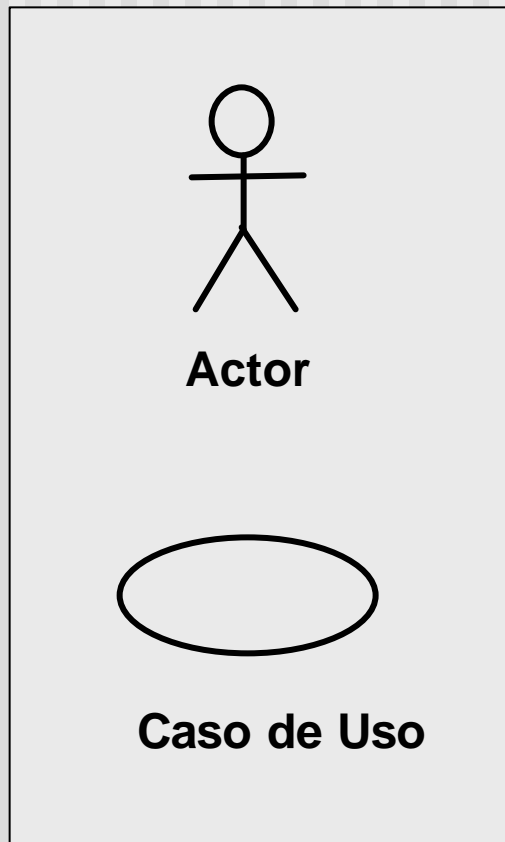
Objetivos: Comportamiento del Sistema

- Usted será capaz de:
 - Definir el comportamiento de un sistema
 - Definir los casos de uso y actores
 - Entender como documentar los casos de uso
 - Usar un diagrama de casos de uso para mostrar los actores, casos de uso y sus interacciones
 - Definir escenarios para los casos de uso

¿Qué es el Comportamiento del Sistema?

- El comportamiento del sistema es como este actúa y reacciona a su entorno
 - La actividad aparentemente visible y comprobable de un sistema
- El comportamiento del sistema se captura en casos de uso
 - Describen al sistema, su ambiente y las relaciones entre el sistema y su ambiente

Conceptos Importantes en el Modelado de Casos de Uso



- Un actor representa cualquier cosa que interactúa con el sistema
- Un caso de uso es una secuencia de acciones que un sistema desempeña y que produce un resultado observable por un actor

¿Qué es un Modelo de Casos de Uso?

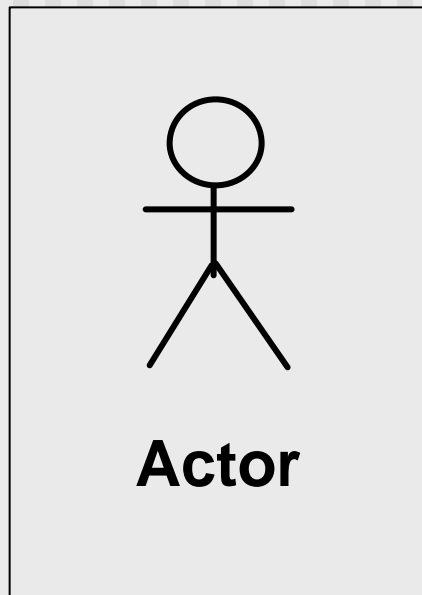
- Un modelo de casos de uso es una representación de las funciones intencionales del sistema (casos de uso) y sus alrededores (actores)
- El mismo modelo de casos de uso se emplea en el análisis de requerimientos, diseño y pruebas

El objetivo principal del modelo de casos de uso es comunicar la funcionalidad y el comportamiento del sistema hacia el cliente o usuario final

Beneficios de un Modelo de Casos de Uso

- El modelo de casos de uso
 - Se utiliza para comunicarse con los usuarios finales y expertos del dominio
 - Proporciona una etapa previa al desarrollo de sistemas
 - Asegura el entendimiento mutuo de los requerimientos
 - Se utiliza para identificar
 - ¿Quién interactuará con el sistema y qué debe hacer el sistema?
 - ¿Qué interfaz debe tener el sistema?
 - Se utiliza para verificar
 - Que se capturen todos los requerimientos
 - Que los desarrolladores hayan entendido los requerimientos

Actores



- Los actores no son parte del sistema, representan roles que un usuario del sistema puede ejecutar
- Un actor puede intercambiar información activamente con el sistema
- Un actor puede ser un recipiente pasivo de información
- Un actor puede representar a una persona, a una máquina o a otro sistema

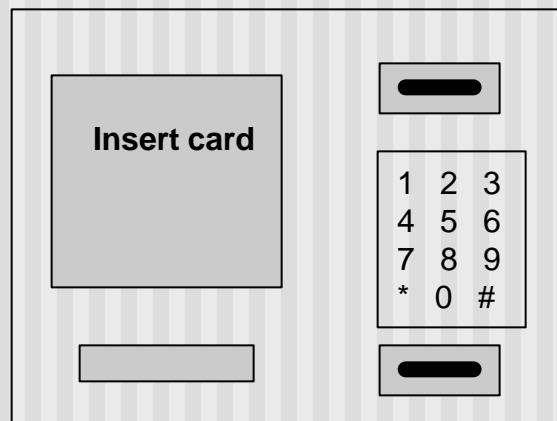
Identificación de Actores:

Preguntas Útiles

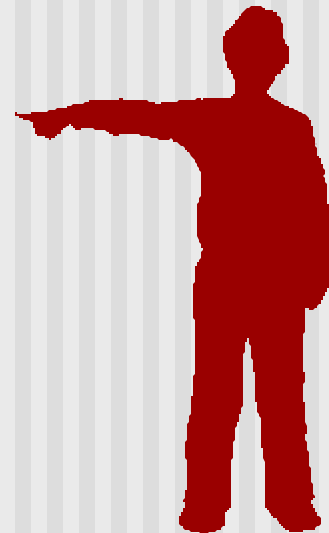
- ¿Quién está interesado en cierto requerimiento?
- ¿En qué parte de la organización se usará el sistema?
- ¿Quién proveerá al sistema con información, la usará y/o borrará?
- ¿Quién usará esta función?
- ¿Quién le dará soporte y mantenimiento al sistema?
- ¿El sistema usa una fuente externa?
- ¿Qué actores necesitan los casos de uso?
- ¿Puede un actor desempeñar roles diferentes?
- ¿Varios actores desempeñan el mismo rol?

Instancias de Actores

Ivan actúa
como un
actor



Tom actúa
como un
actor



Modelo de Casos de Uso

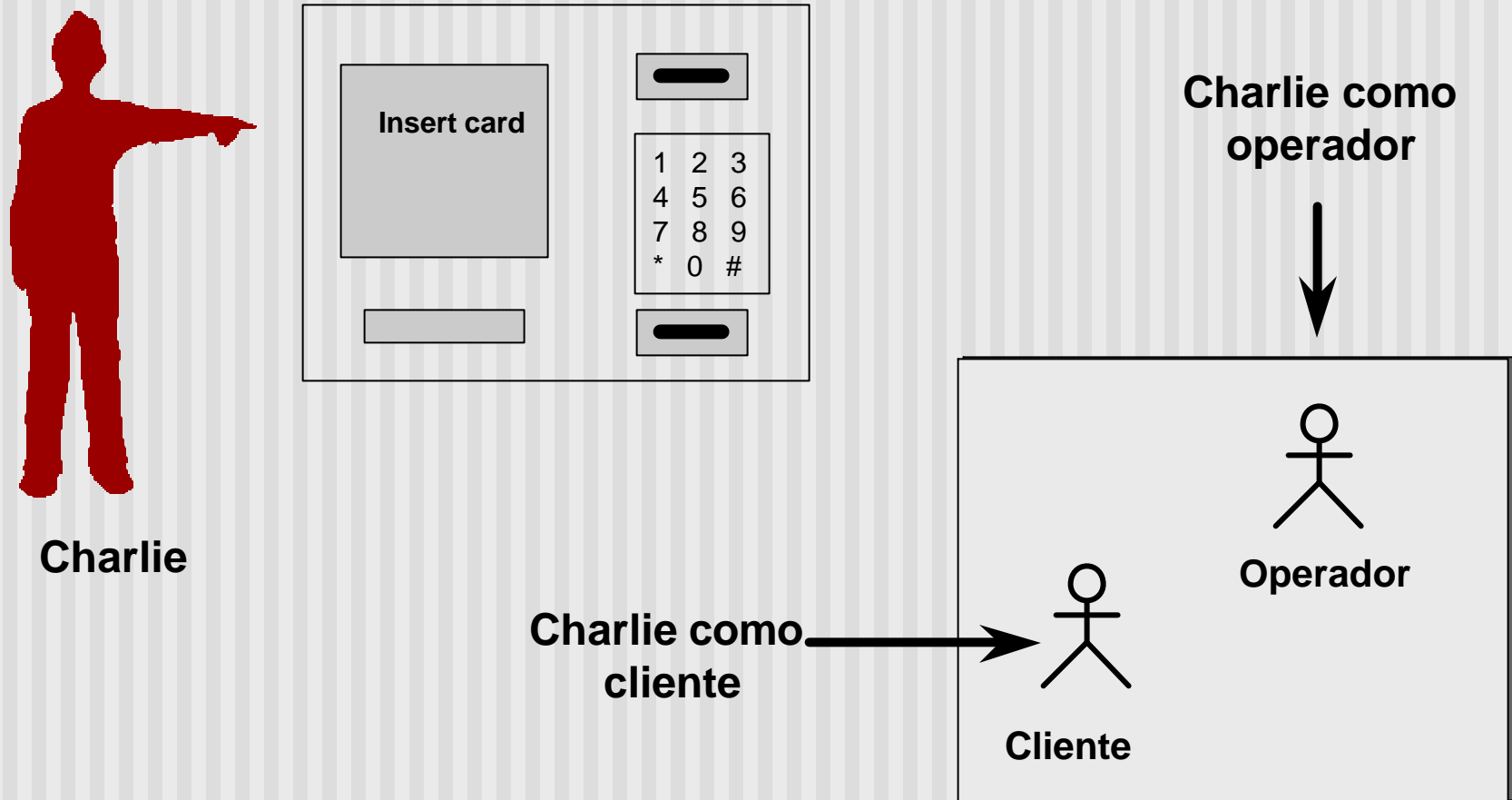


Actor

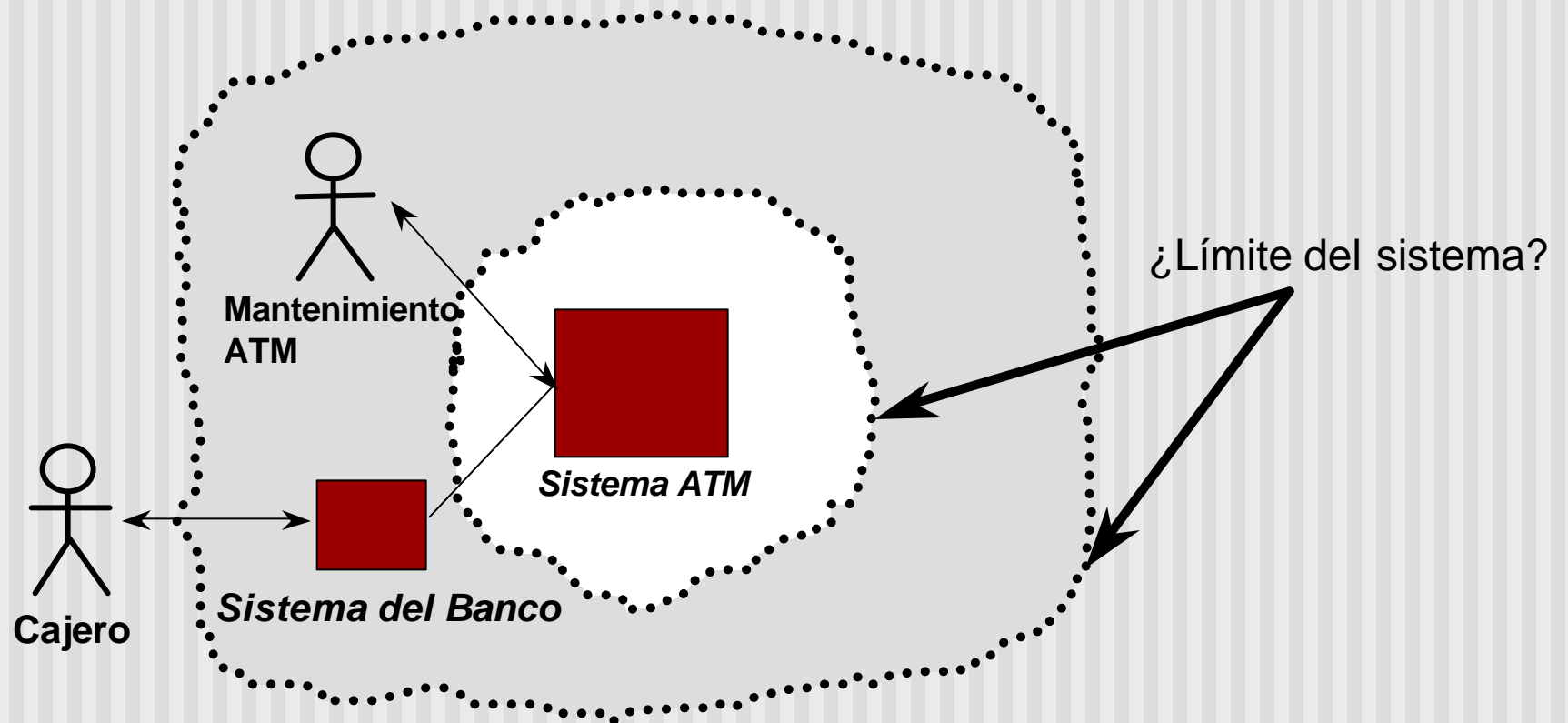


caso de uso

Un usuario puede actuar como varios actores



Límites de los actores y el sistema



Casos de Uso



- Un caso de uso modela un diálogo entre actores y el sistema
- Un actor inicia un caso de uso para invocar cierta funcionalidad del sistema
- Un caso de uso es un flujo de eventos completo y significativo
- El conjunto de todos los casos de uso, representa todas las formas posibles de uso del sistema

Identificación de Casos de Uso:

Preguntas Útiles

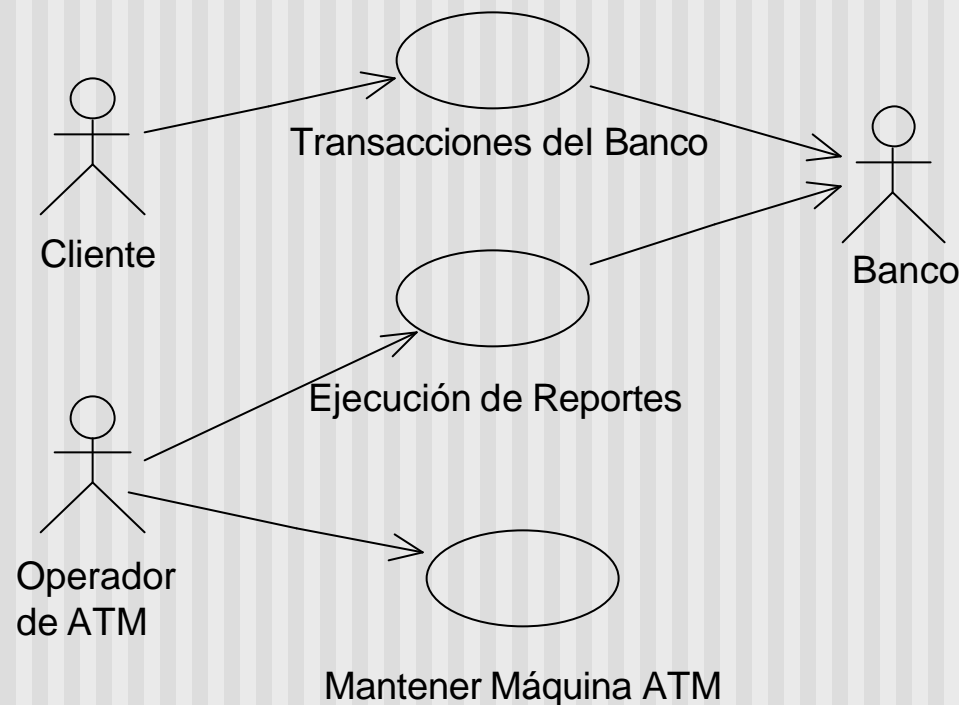
- ¿Cuáles son las tareas que realiza este actor?
- ¿El actor creará, almacenará, cambiará, borrará o leerá información en el sistema?
- ¿Qué caso de uso creará, almacenará, cambiará, borrará o leerá esta información?
- ¿Necesitará el actor informar al sistema sobre cambios externos repentinos?
- ¿Necesitará el actor recibir información en relación a ciertas ocurrencias en el sistema?
- ¿El sistema proporciona al negocio el comportamiento correcto?
- ¿Qué casos de uso van a darle soporte y mantenimiento al sistema?
- ¿Pueden todos los requerimientos funcionales ser ejecutados por los casos de uso?

Fuentes de Información para los Casos de Uso

- Declaración de especificaciones del sistema
- Definición del problema a resolver
- Literatura relevante al dominio
- Entrevistas con expertos del dominio
- Conocimiento personal del dominio o experiencia
- Sistemas Anteriores o Legados

Diagrama Casos de Uso

- Se dibuja un **diagrama de casos de uso** para ilustrar los casos de uso y los actores que interactúan enviándose estímulos el uno al otro

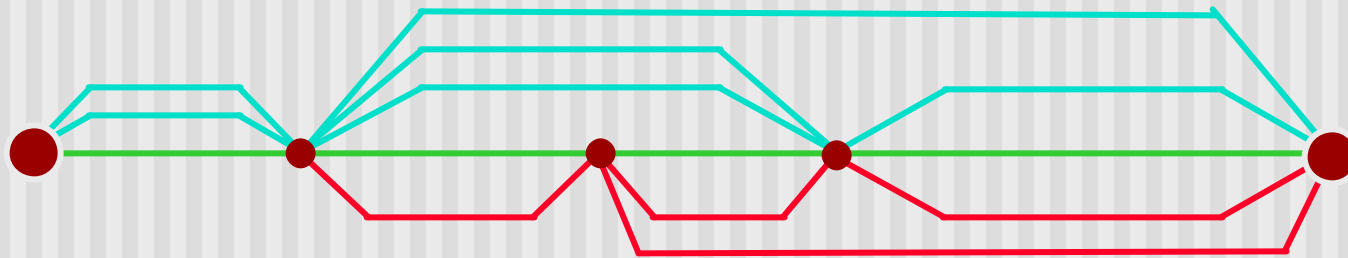


Documentación de un Caso de Uso

- Los casos de usos se documentan con:
 - Una breve descripción
 - **Se expone el propósito del caso de uso en unas cuantas líneas**
 - Flujo de eventos detallado
 - **Descripción del flujo primario y los flujos alternos de eventos que ocurren desde el inicio el casos de uso**
 - La documentación debe leerse como un diálogo entre el actor y el caso de uso
- Ambas partes de la documentación deben estar escritos en términos que el cliente entienda

Flujo de Eventos en un Caso de Uso

- Cada caso de uso
 - Tiene una secuencia de transacciones normal o básica
 - Debe tener varias secuencias alternativas de transacciones
 - Generalmente tiene secuencias de excepción a transacciones que manejan situaciones erróneas
 - También debe tener pre y post condiciones bien definidas



Flujo de Eventos en un Caso de Uso (cont.)

- Describe sólo los eventos que pertenecen al caso de uso, y no lo que ocurren en otros casos de uso
- Evitar el uso de terminología vaga como: “por ejemplo”, “etc.” e “información”
- El flujo de eventos deberá describir:
 - ¿Cómo y cuándo inicia y termina el caso de uso?
 - ¿Cuándo interactúa el caso de uso con los actores?
 - ¿Qué información se intercambia entre un actor y el caso de uso?
 - No describe los detalles de la interfaz de usuario
 - Describe el flujo básico de eventos
 - Cualquier flujo de eventos alternativo

¿Quién lee la documentación asociada a los Casos de Uso?



- **Cientes**: aprueban lo que el sistema debe hacer
- **Usuarios**: ganan entendimiento del sistema
- **Desarrolladores**: documento de comportamiento del sistema
- **Examinadores**: examinan el flujo de eventos
- **Analistas o Diseñadores**: proporciona las bases para el análisis y diseño
- **Evaluador**: se usa como base para la prueba de requerimientos
- **Líder de Proyecto**: proporciona elementos para la planeación de proyectos
- **Escritor Técnico**: base para la escritura de la guía de usuario

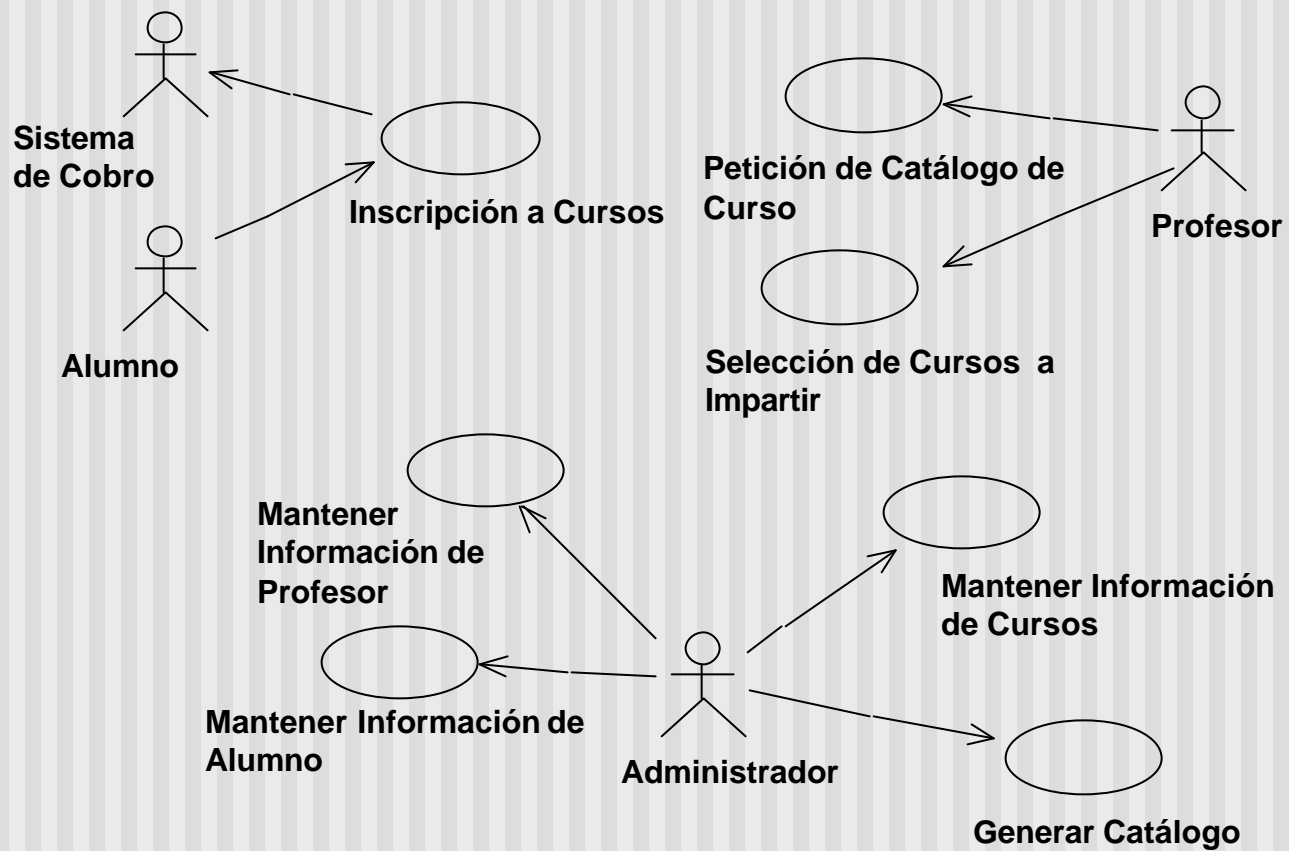
Ejemplo: Inscripción a Cursos

- Al inicio de cada semestre, los alumnos solicitan un catálogo que contiene la lista de los cursos que se impartirán en el semestre, en el cual se incluyen también datos relacionados como: profesor, departamento y pre-requisitos.
- El sistema nuevo deberá permitir que los alumnos seleccionen cuatro cursos para el semestre que inicia. Además, cada alumno indicará dos cursos alternativos en caso de que no pueda ser asignada la primera selección. Los nuevos cursos tendrán un máximo de diez alumnos y un mínimo de tres. Un curso con menos de tres alumnos será cancelado. Una vez que el proceso de inscripción se ha completado para un alumno, el sistema de registro envía la información al sistema de cobros, para que el alumno pueda pagar por el semestre.

Ejemplo: Inscripción a Cursos (cont.)

- Los profesores deben ser capaces de ingresar al sistema para indicar que cursos van a impartir. También podrán ver qué alumnos están inscritos en sus cursos.
- Para cada semestre, hay un periodo en el que los alumnos pueden cambiar su horario. Los alumnos deben ser capaces de ingresar al sistema durante este tiempo para agregar o cancelar cursos.

Diagrama de casos de uso



1. Breve Descripción: Caso de Uso **Inscripción a Cursos**

1.1 Breve Descripción

- Este caso de uso es iniciado por un alumno. Proporciona la capacidad para que un alumno cree, borre, modifique y/o revise un horario de curso para un semestre dado.

2. Flujo de Eventos: Casos de Uso

Inscripción a Cursos

2.1 Pre-condiciones

Ninguna

2.2 Flujo Principal

Este caso de uso inicia cuando un alumno introduce el número de id de alumno. El sistema verifica que el número de id de alumno sea válido (E-1) y permite que el alumno seleccione el semestre actual o uno futuro (E-2). El sistema permite que el alumno seleccione la actividad deseada:

Crear, Revisar, Modificar, Imprimir, Borrar o Salir.

Si la actividad seleccionada es:

A-1 Crear: Subflujo Crear un Horario Nuevo.

A-2 Revisar: Subflujo Revisar un Horario.

A-3 Modificar: Subflujo Modificar un Horario.

A-4 Imprimir: Subflujo Imprimir un Horario.

A-5 Borrar: Subflujo Borrar un Horario.

Salir: termina el caso de uso.

2. Flujo de Eventos: Casos de Uso Inscripción a Cursos (cont.)

2.3 Flujos Alternos

A-1 **Crear**: Subflujo Crear un Horario Nuevo:

El sistema despliega una pantalla de horario en blanco. Los alumnos introducen los 4 cursos primarios y 2 cursos alternativos (E-3). El alumno envía su requerimiento de cursos. Por cada curso primario seleccionado el sistema revisa que se satisfagan los pre-requisitos (E-4) y agrega al alumno al curso si éste está abierto (E-5). El sistema imprime el horario del alumno (E-6) y envía esta información al sistema de cobros para procesarla (E-7). El Caso de Uso inicia de nuevo.

A-2 **Revisar**: Subflujo Revisar un Horario:

El sistema recupera (E-8) y despliega la información para todos los cursos a los cuales el alumno se registro: nombre del curso, número del curso, número de lugares del curso, días de la semana, hora, lugar y número de horas necesarias. Cuando el alumno indica que el o ella ha terminado su revisión, el Caso de Uso inicia de nuevo.

2. Flujo de Eventos: Casos de Uso Inscripción a Cursos (cont.)

A-3 Modificar: Subflujo Modificar un Horario:

El sistema revisa que la fecha límite para los cambios no haya expirado (E-9). El sistema recupera (E-8) y despliega la siguiente información para todos los cursos a los cuales el alumno se inscribió: nombre del curso, numero del curso, numero de lugares del curso, días de la semana, hora, lugar y número de horas necesarias. El sistema permite que el alumno seleccione la actividad que deseada: **Borrar Curso**, **Agregar Curso** o **Salir**.

Si la actividad seleccionada es:

A-6 Borrar Curso: Subflujo Borrar un Curso.

A-7 Agregar Curso: Subflujo Agregar un Curso.

Salir, el sistema imprime el horario del alumno (E-6) y el Caso de Uso inicia de nuevo.

2. Flujo de Eventos: Casos de Uso **Inscripción a Cursos (cont.)**

A-4 Imprimir: Subflujo Imprimir un Horario:
El sistema imprime el horario del alumno (E-6). El Caso de Uso inicia de nuevo.

A-5 Borrar: Subflujo Borrar un Horario:
El sistema recupera (E-8) y despliega la información actual del horario. El sistema pide al usuario que confirme la eliminación del horario. Si se confirma, se borra el horario del sistema. Si la eliminación no se confirma, la operación se cancela y el Caso de Uso inicia de nuevo.

A-6 Borrar Curso: Subflujo Borrar un Curso:
El alumno introduce el número del curso para borrarlo. El sistema pide al usuario que confirme la eliminación del curso. Si se confirma, se borra el curso del horario del alumno. Si la eliminación no se confirma, la operación se cancela y el caso de uso inicia de nuevo.

2. Flujo de Eventos: Casos de Uso Inscripción a Cursos (cont.)

A-7 Agregar Curso: Subflujo Agregar un Curso:

El alumno introduce el número de curso para agregarlo. El sistema revisa que se satisfagan los pre-requisitos (E-4) y agrega el alumno al grupo si el curso esta abierto (E-5). El caso de uso inicia de nuevo.

Flujos de Excepción

E-1: Se introdujo un número id de alumno inválido. El usuario puede re-introducir el número id o finalizar el caso de uso.

E-2: Se introdujo un semestre inválido. El usuario puede re-introducir el semestre o finalizar el casos de uso.

E-3: El número de curso no es válido. El usuario puede re-introducir el número válido o finalizar el caso de uso.

E-4: Los pre-requisitos no fueron satisfechos por el usuario. Se le informa al alumno que un curso no puede ponerse en el horario y el motivo. De ser posible, se sustituye con un curso alternativo. El caso de uso continua.

2. Flujo de Eventos: Casos de Uso Inscripción a Cursos (cont.)

E-5: E curso seleccionado esta cerrado. De ser posible, se substituye con un curso alternativo. El caso de uso continua.

E-6: No es posible imprimir el horario. La información se guarda y se le informa al usuario que la petición de impresión del horario debe ser reintroducida. El caso de uso continua.

E-7: No es posible enviar la información al sistema de cobro. El sistema guarda toda la información de cobro y la re-envía después al sistema de cobros. El caso de uso continua.

E-8: El sistema no puede recuperar la información del horario. El caso de uso inicia nuevamente.

E-9: El sistema informa al usuario que no se puede modificar un horario. El caso de uso inicia nuevamente.

¿Qué son los escenarios?

- Un escenario es una instancia de un caso de uso
 - Es un flujo determinado de eventos en un caso de uso
- Cada caso de uso tiene múltiples escenarios
 - Escenarios primarios (happy path scenarios)
 - **Flujo normal: la forma en la que debe trabajar el sistema**
 - Escenarios secundarios
 - **Excepciones del escenario primarios**

Escenario para el Caso de Uso

Inscripción a Cursos

- John introduce el número id de alumno 369 52 3449 y el sistema lo valida. El sistema pregunta que a semestre desea inscribirse. John le indica al sistema el semestre actual y elige crear un horario nuevo.
- De una lista de cursos disponibles, John selecciona los siguientes 4 cursos primarios: English 101, Geology 110, World History 200, y College Algebra 110. Después selecciona 2 cursos alternos: Music Theory 110 y Introduction to Java Programming 180.
- El sistema determina que John tiene todos los pre-requisitos necesarios y lo agrega a las listas de los cursos correspondientes.
- El sistema indica que la actividad esta completa. El sistema imprime el horario del alumno y envía la información de cobro de cuatro cursos primarios al sistema de cobros para que sea procesado.

Escenarios Secundarios

- ¿Qué escenarios secundarios podrían considerarse para el caso de uso: "Inscripción a Cursos"?

Escenarios Secundarios (cont.)

- **Algunos escenarios secundarios a considerarse:**
 - El alumno no seleccionó los 4 cursos primarios
 - Algún curso primario no esta disponible
 - Los cursos primarios y secundarios no están disponibles
 - No se puede agregar el alumno a la lista de un curso
 - No se puede crear el horario del alumno

¿Cuántos escenarios se necesitan?

- Respuesta sencilla: tantos como se necesiten para entender el desarrollo del sistema
- Sugerencia:
 - Escenarios Primarios
 - Dedicar aproximadamente el 80% del tiempo a estos escenarios
 - Escenarios Secundarios
 - Elaborar algunos de los escenarios secundarios interesantes y de alto riesgo

Ejercicio: Comportamiento del Sistema

- Utilice el problema que proporciona el instructor
 - Dibujar un diagrama de casos de uso
 - Escribir una definición para cada actor
 - Para un caso de uso
 - Escribir una breve descripción (dos sentencias máximo)
 - Escribir el flujo de eventos
 - Listar algunos escenarios posibles

Objetos y Clases



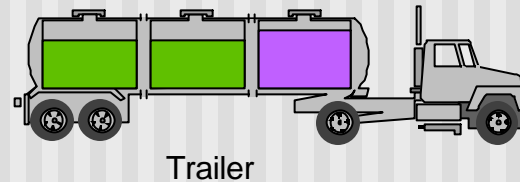
Objetivos: Objetos y Clases

- **Usted podrá:**
 - Definir y dar ejemplos de objetos
 - Definir y dar ejemplos de clases
 - Describir las relaciones entre clases y objetos
 - Definir estereotipos

¿Qué es un objeto?

- Informalmente, un objeto representa una entidad, ya sea física, conceptual o software

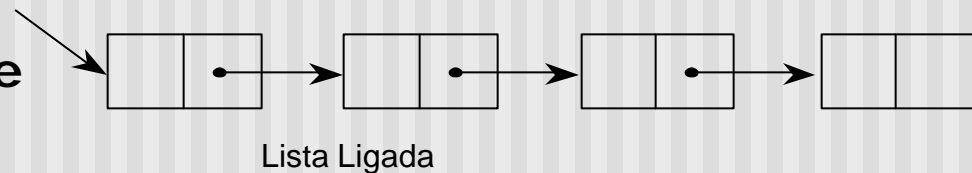
- Entidad física



- Entidad conceptual



- Entidad de software

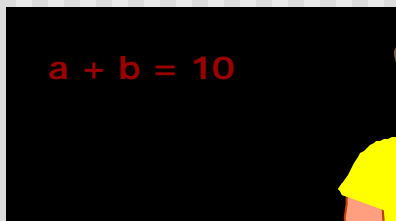


Una definición más formal

- Un objeto es un concepto, abstracción, o cosa con límites bien definidos y significado para una aplicación
- Un objeto es algo que tiene:
 - Estado
 - Comportamiento
 - Identidad

Un Objeto tiene Estado

- El estado de un objeto es una de las condiciones posibles en las que un objeto puede existir
- El estado de un objeto normalmente cambia con el paso del tiempo
- El estado de un objeto generalmente se implementa por una serie de propiedades (llamadas atributos), con los valores de las propiedades, más las relaciones que el objeto pueda tener con otros objetos



Profesora Clark

Nombre:	Joyce Clark
Empleado ID:	567138
Fecha de contrato:	March 21, 1987
Estado:	Tenured

Un Objeto tiene Comportamiento

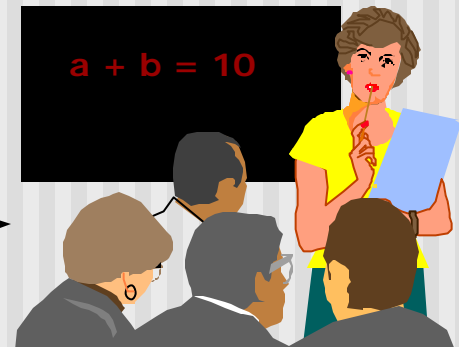
- El comportamiento determina como actúa y responde un objeto
- El comportamiento define como responde un objeto a peticiones de otros objetos
- El comportamiento visible de un objeto se modela por un conjunto de mensajes a los que puede responder (las operaciones que el objeto puede desempeñar)



Sistema de Inscripción

Asignar al Profesor Clark

(Regresa:confirmación)



Curso de Algebra 101

Un Objeto tiene Identidad

- Cada objeto tiene identidad única, aún si el estado es idéntico al de otro objeto



Profesor "J Clark"
imparte Algebra



Profesor "J Clark"
imparte Algebra



Profesor "J Clark"
imparte Algebra

¿Qué son las clases?

- Hay varios objetos identificados en cualquier dominio
- Una clase es una descripción de un grupo de objetos con propiedades comunes (atributos), comportamiento común (operaciones), relaciones comunes con otros objetos (asociaciones y agregaciones) y semánticas comunes.
 - Un objeto es una instancia de una clase
- Una clase es una abstracción en la que ella:
 - Enfatiza características relevantes
 - Suprime otras características
- La abstracción nos ayuda a manejar la complejidad

Ejemplo de Clase

Clase Curso

Estructura

Nombre
Ubicación
Días ofrecidos
Horas de créditos
Hora inicio
Hora término

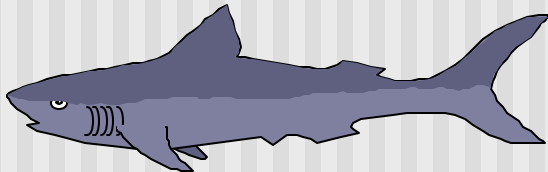
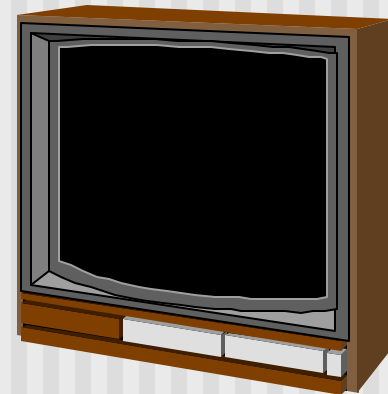
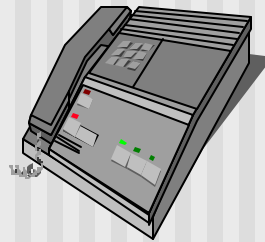
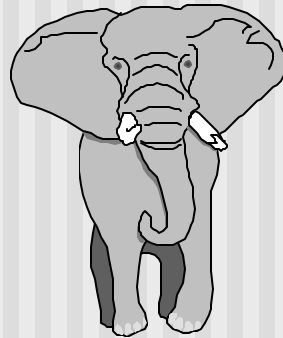


Comportamiento

Agregar un alumno
Borrar un alumno
Obtener catálogo de cursos
Determinar si está lleno

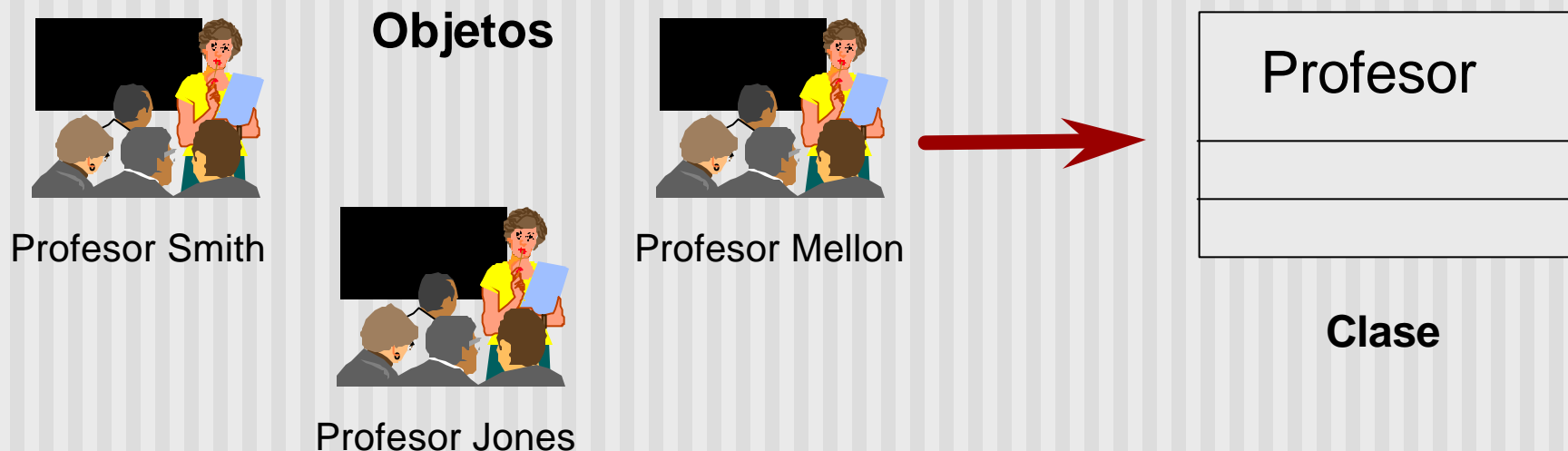
Clases de Objetos

- ¿Cuántas clases ve?



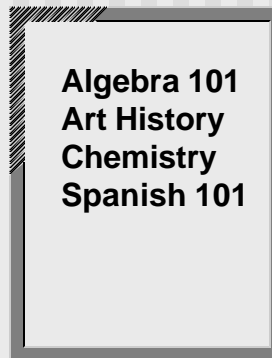
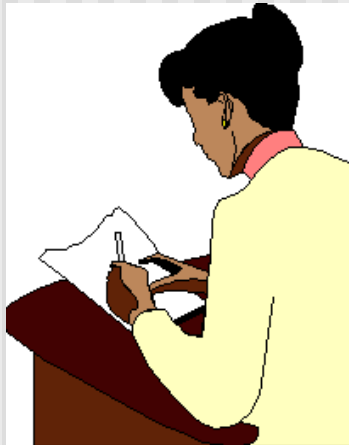
Relaciones entre Clases y Objetos

- Una clase es una definición abstracta de un objeto
 - Define la estructura y comportamiento de cada objeto en la clase
 - Sirve como una plantilla para crear objetos
- Los objetos pueden agruparse en clases



Guía para identificar Clases

- Una clase debe capturar una y solo una llave de abstracción
- Mala abstracción: la clase Alumno sabe la información del alumno y su horario para el semestre actual
- Buena abstracción: Separar las clases en una para alumno y otra para Horario del alumno



¿Cómo nombrar a una Clase?

- El nombre de una clase debe ser un nombre en singular que caracterice de la mejor forma a la abstracción
- La dificultad al nombrar a una clase puede indicar que una abstracción está pobremente definida
- Los nombres deben venir directamente del vocabulario del dominio

Guía de estilo para nombrar Clases

- Una guía de estilo debe dictar convenciones de nombres para clases
- Ejemplo de Guía de Estilo
 - Las clases se nombran usando sustantivos en singular
 - Los nombres de clases empiezan con una letra mayúscula
 - No se usan palabras subrayadas
 - **Los nombres compuestos de palabras múltiples se ponen juntas y la primera letra de cada palabra adicional se escribe en mayúscula**
- Ejemplo: Alumno, Profesor, SistemaCobro

Definición de Semántica de Clases

- Después de nombrar una clase, se debe hacer una breve y concisa descripción de la clase
 - Enfocarse en el propósito de la clase y no en la implementación
- El nombre de la clase y la descripción forman la base de un diccionario del modelo

Busque el “QUÉ” y no el “CÓMO”

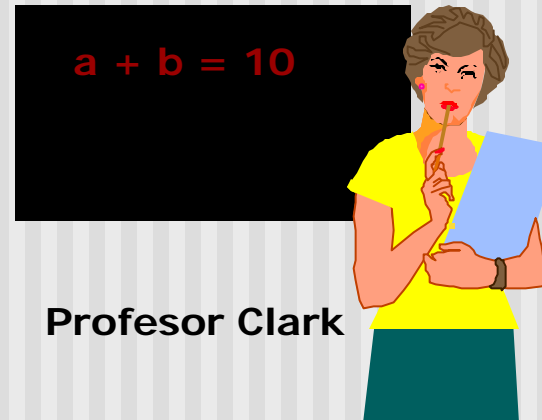
Ejemplo de Entradas al Diccionario del Modelo

- Nombre: StudentInformation
 - Definición: Información relacionada a una persona registrada para asistir a clases en la Universidad
- Nombre: Course
 - Definición de Trabajo: Una clase ofrecida por la Universidad

Al ir estudiando más el problema, se descubren clases y se mejoran las definiciones de las anteriores, agregándolas al diccionario del modelo

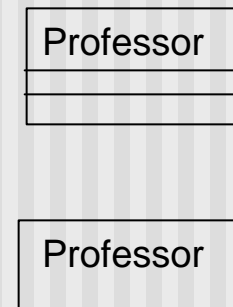
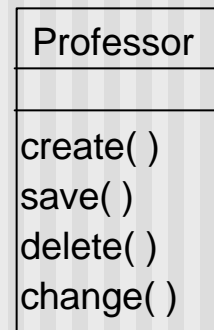
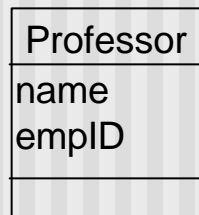
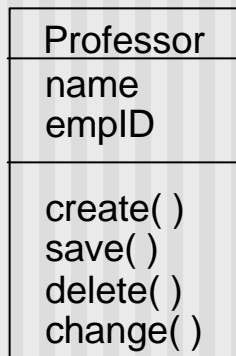
Representación de Clases

- Una clase se representa usando un rectángulo con tres divisiones



Divisiones de Clase

- Una clase comprende tres secciones
 - La primera sección contiene el nombre de la clase
 - La segunda sección muestra la estructura (atributos)
 - La tercera sección muestra el comportamiento (operaciones)
- La segunda y tercera sección pueden suprimirse si no es necesario que sean visibles en el diagrama

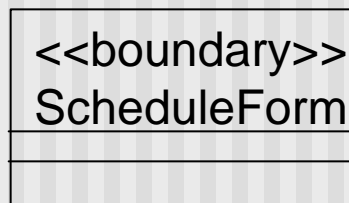


Estereotipos

- Un estereotipo es un nuevo tipo de elemento de modelado que extiende la semántica del metamodelo
 - **Deben estar basados en tipos o clases existentes en el metamodelo**
- Cada clase puede tener como máximo un estereotipo
- Estereotipos comunes
 - **Clase Boundary**
 - **Clase Entity**
 - **Clase Control**
 - **Clase Exception**
 - **Metaclass**
 - **Clase Utility**
- Los Estereotipos se muestran en la parte donde se escribe el nombre de la clase entre << >>

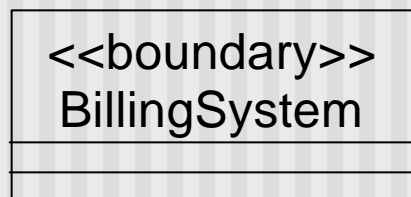
Clases Boundary

- Una clase boundary modela la comunicación entre los alrededores del sistema y sus funciones internas
- Clases boundary típicas
 - **Windows (interfaz de usuario)**
 - **Protocolo de Comunicación (interfaz del sistema)**
 - **Interfaz de impresora**
 - **Sensores**
- En el escenario de “Inscripción a Cursos”, se crea una pantalla de horario (ScheduleForm) para aceptar información del usuario



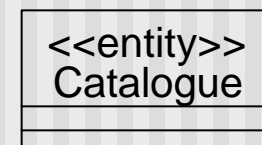
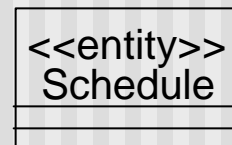
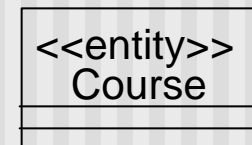
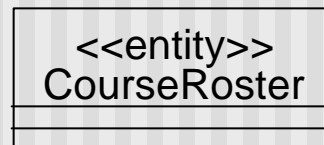
Interfaz con otros sistemas

- Una clase boundary se usa también para modelar una interfaz con otro sistema
- Las características importantes de este tipo de clases son:
 - **La información que va a pasarse al otro sistema**
 - **El protocolo de comunicación que se use para “hablar” con el otro sistema**
- En el escenario “Inscripción a Cursos”, la información debe enviarse al sistema de cobros (BillingSystem)
 - **Se crea una clase llamada BillingSystem para mantener la interfaz del sistema externo**



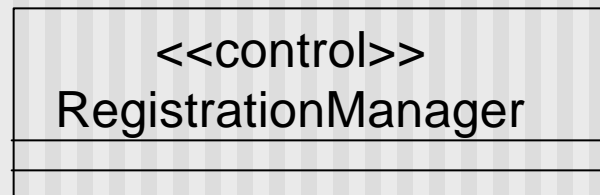
Clases Entity

- Una clase entity modela información y comportamiento asociado que es generalmente de larga vida (persistente)
 - Puede reflejar un fenómeno de la vida real
 - También puede necesitarse para las tareas internas del sistema
 - Los valores de sus atributos son proporcionados generalmente por un actor
 - Su comportamiento es independiente de los alrededores
- Clases entity en el caso de uso “Inscripción a Cursos”
 - **Course**
 - **Schedule**
 - **Catalogue**
 - **CourseRoster**



Clases Control

- Una clase control modela el comportamiento de control específico a uno o más casos de uso
- Una clase control
 - **Crea, inicia y borra objetos controlados**
 - **Controla la secuencia o coordinación de ejecución de objetos controlados**
 - **Controla elementos actuales para clases controladas**
 - **Es, la mayor parte de las veces, la implementación de un objeto intangible**
- En el escenario “Inscripción a Cursos”, la clase `RegistrationManager` controla el proceso de inscripción



Interacción de Objeto



Objetivos: Interacción de Objeto

- Usted podrá:
 - Usar diagramas de secuencia y colaboración para mostrar las interacciones de los objetos

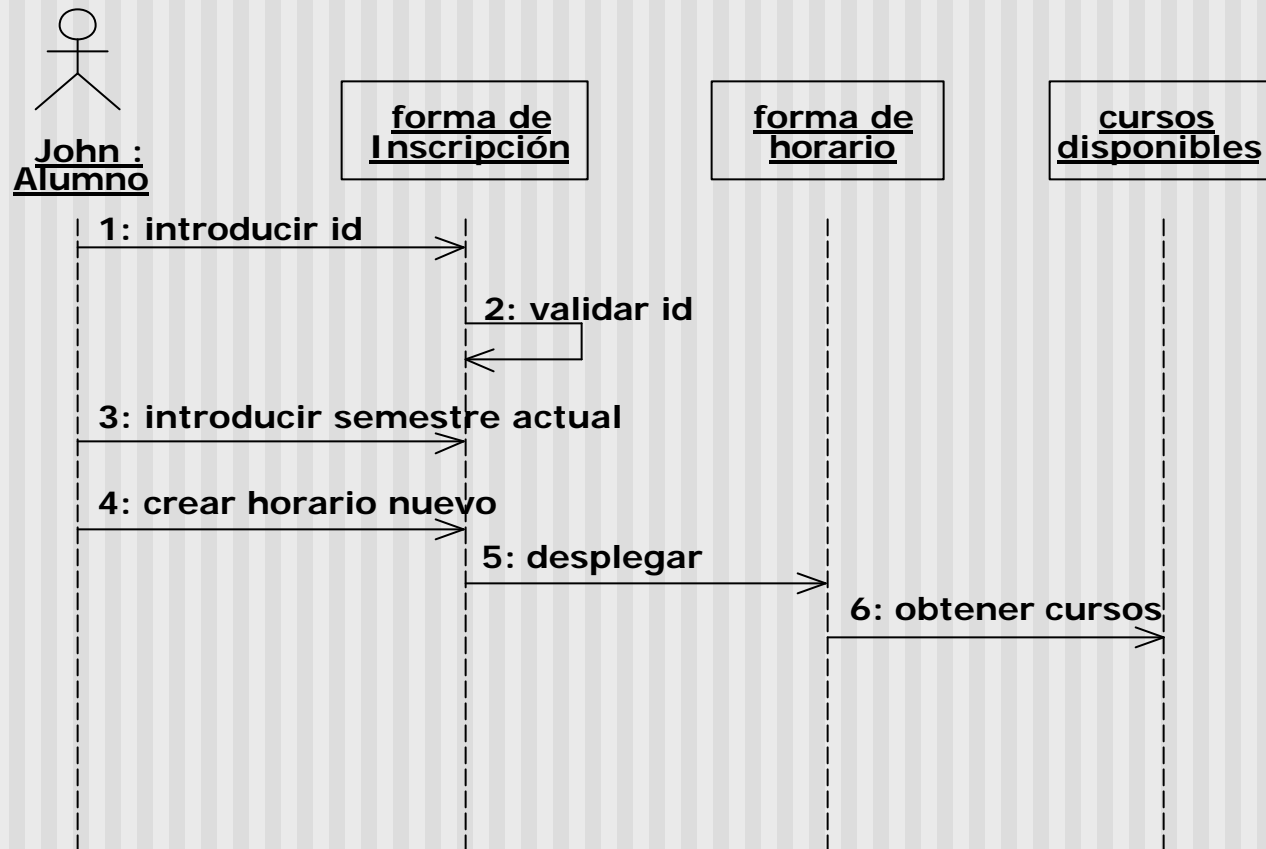
¿Qué es un Diagrama de Interacción?

- Un diagrama de interacción es una representación gráfica de las interacciones entre objetos
- Hay dos tipos de diagramas de interacción
 - Diagramas de Secuencias
 - Un diagrama de secuencias están ordenado de acuerdo al tiempo
 - Diagramas de Colaboración
 - Un diagrama de colaboración incluyen el flujo de datos
- Cada uno provee un punto de vista diferente de la misma interacción

¿Qué es un Diagrama de Secuencias?

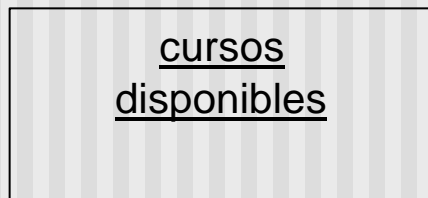
- Un diagrama de secuencia muestra interacciones de objetos ordenados en el tiempo
- El diagrama muestra
 - Los objetos que participan en la interacción
 - La secuencia de mensajes intercambiados
- Un diagrama de secuencias contiene:
 - Objetos con sus “líneas de vida”
 - Mensajes intercambiados entre objetos en orden secuencial
 - *Enfoque de control (Focus of Control) (opcional)*

Un Diagrama de Secuencias

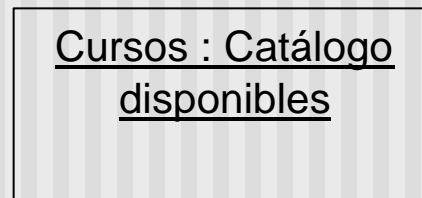


¿Cómo nombrar a los Objetos en un Diagrama de Secuencias?

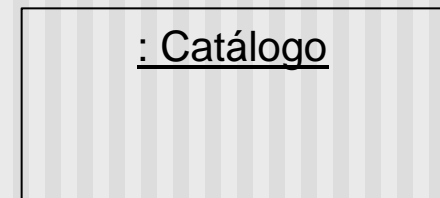
- Los objetos se dibujan como rectángulos con los nombres subrayados
- Las “líneas de vida” se representan con líneas de guiones descendentes



—
—
—
Objetos



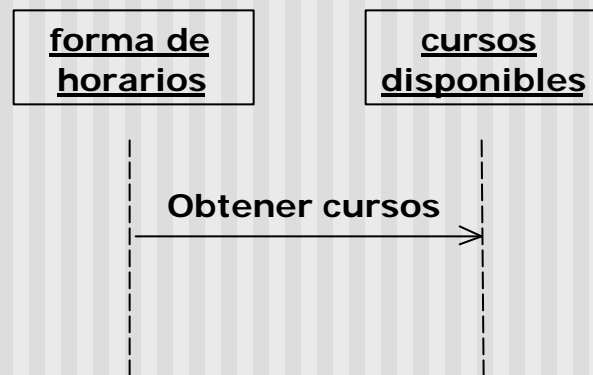
—
—
—
Objetos y Clases



—
—
—
Clases

Mostrar las interacciones entre objetos

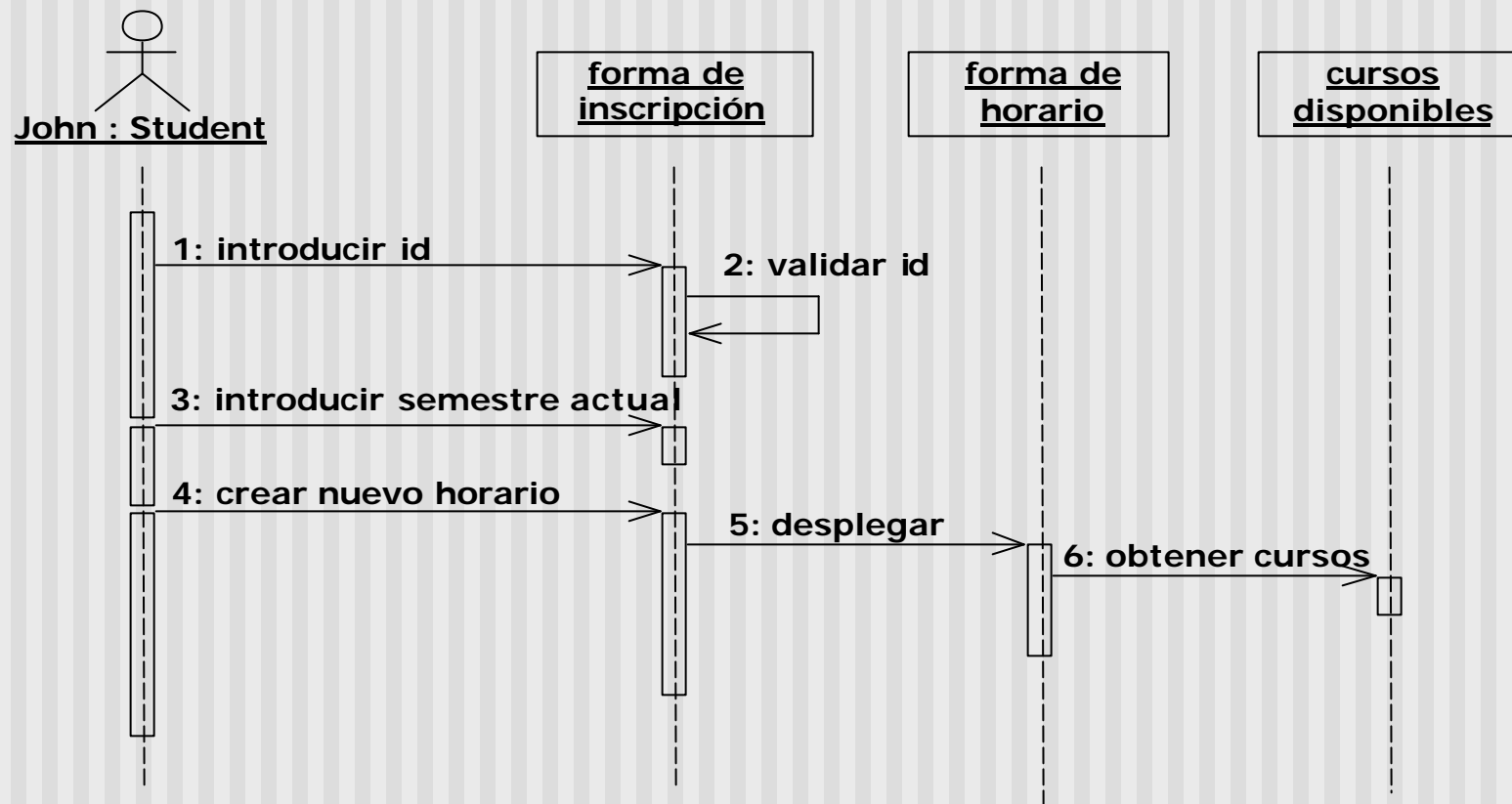
- La interacción de objetos se indica con flechas horizontales que se dirigen desde la línea vertical que representa al objeto cliente hasta la línea que representa al objeto proveedor
- Las flechas horizontales se etiquetan con un mensaje
- El orden en que ocurren los mensajes es indicado por la posición vertical, con el más cercano en la parte superior
- La numeración es opcional, ya que la orden se basa en la posición vertical



¿Qué es el Enfoque de Control?

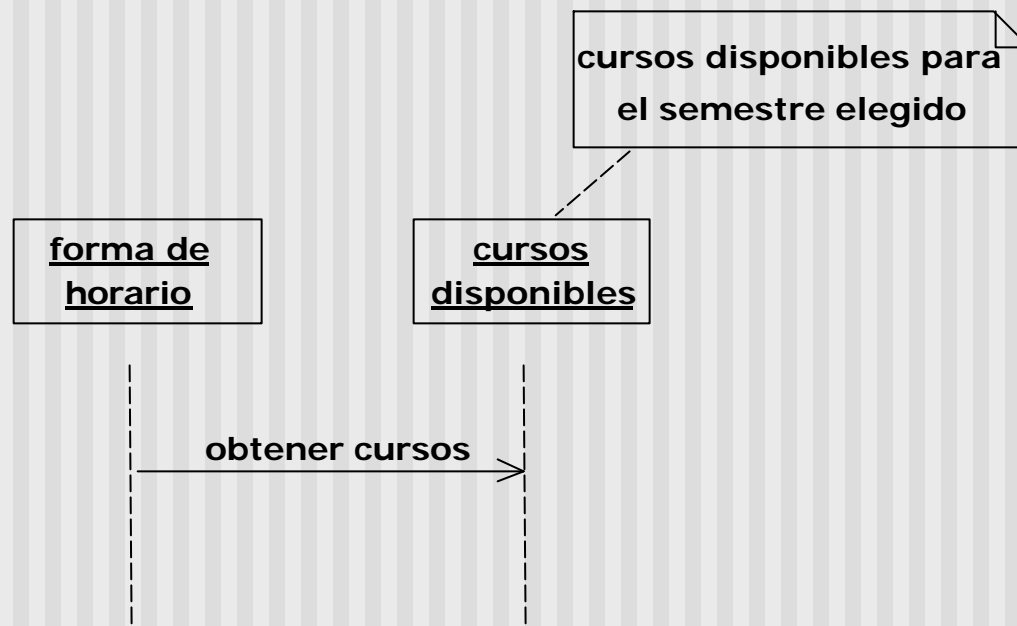
- El Enfoque de Control representa el tiempo relativo en el que el flujo de control se enfoca sobre un objeto
 - Representa el tiempo en que un objeto dirige sus mensajes
- El Enfoque de Control puede mostrarse en un diagrama de secuencia

Enfoque de Control



Notas

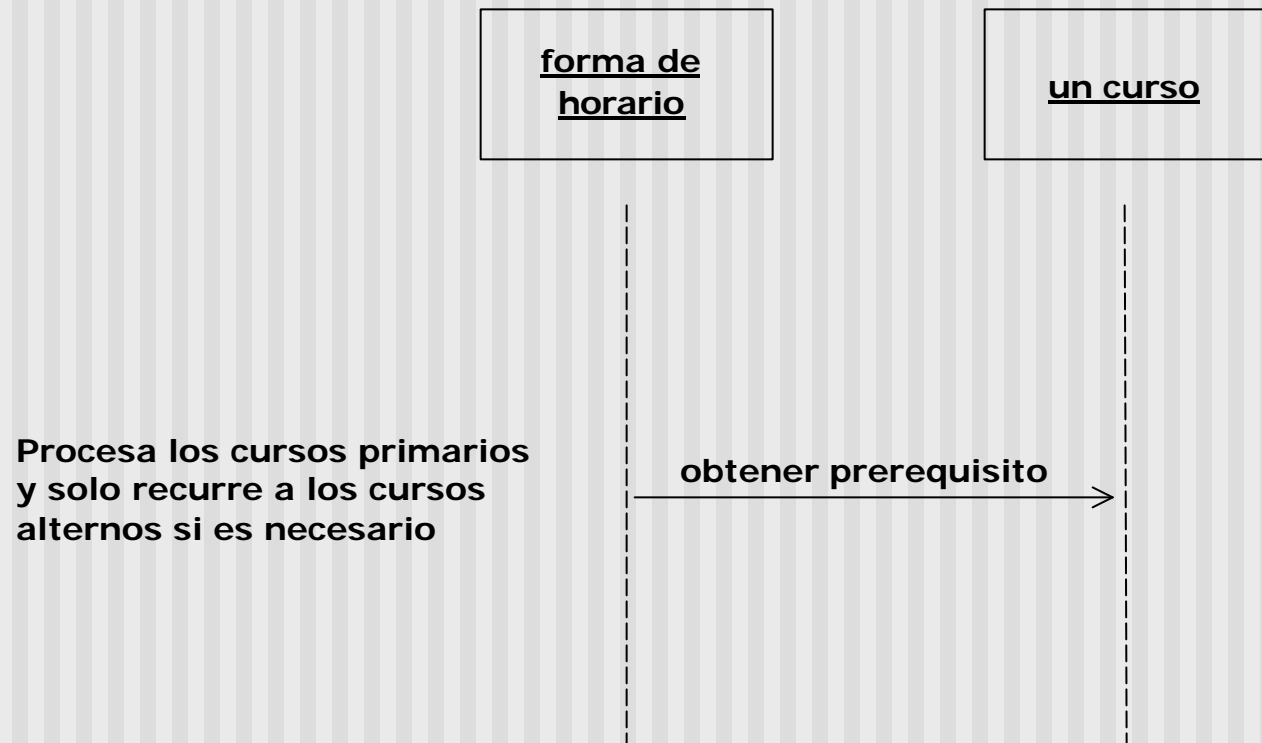
- Las notas pueden agregarse para agregar más información al diagrama



Scripts en Diagramas de Secuencias

- Para escenarios complejos, los diagramas de secuencias pueden ser mejorados mediante el uso de scripts
- Un script se escribe a la izquierda de un diagrama de secuencia con la secuencia de pasos alineados a las interacciones del objeto
- Los scripts se pueden escribir en lenguaje natural o en pseudo código

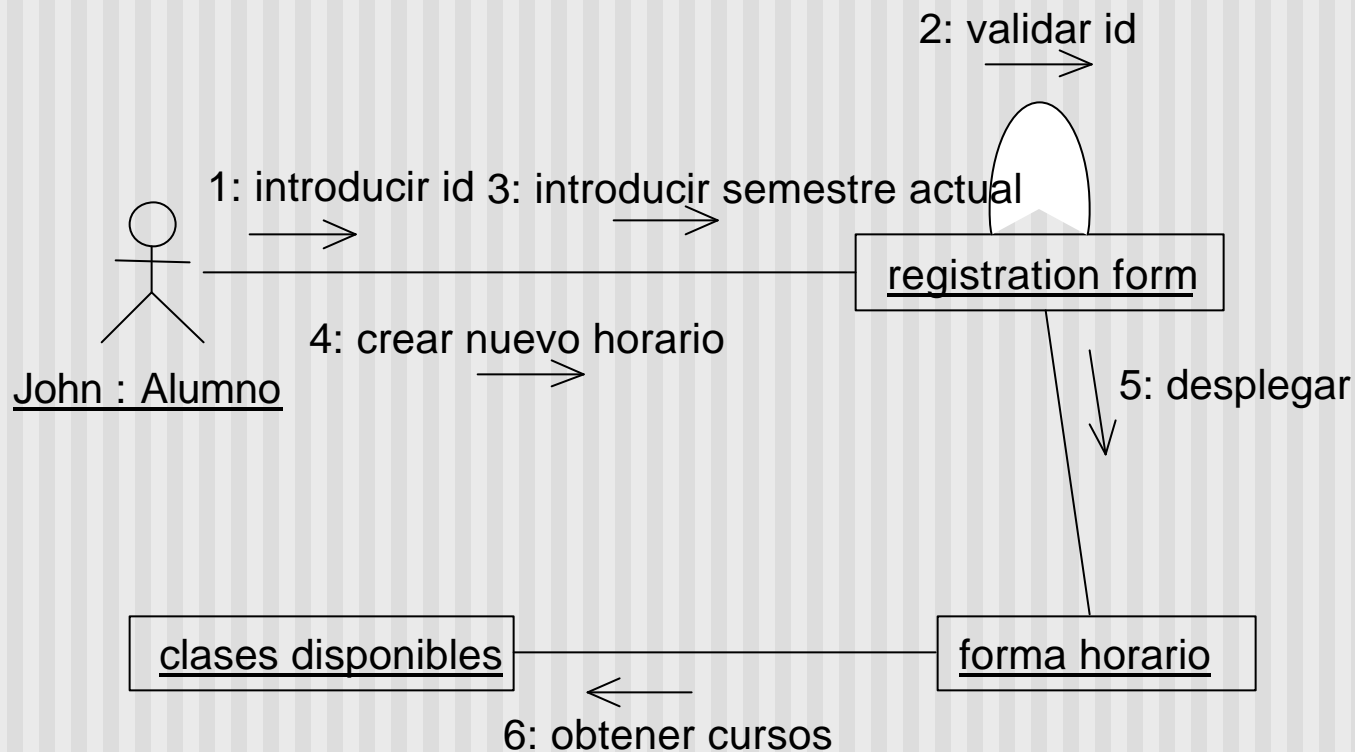
Ejemplo de Script



Diagramas de Colaboración

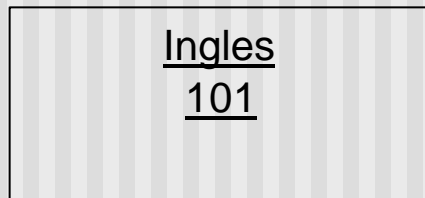
- Un diagrama de colaboración es una forma alternativa de representar el intercambio de mensajes de un conjunto de objetos
- El diagrama muestra las interacciones organizadas entorno a los objetos y a sus relaciones
- Un diagrama de colaboración contiene:
 - Objetos
 - Ligas entre objetos (relaciones)
 - Mensajes intercambiados entre objetos
 - Flujo de datos entre objetos, si hay alguno

Ejemplo de un Diagrama de Colaboración

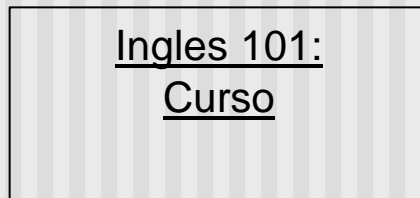


Representación de Objetos en un Diagrama de Colaboración

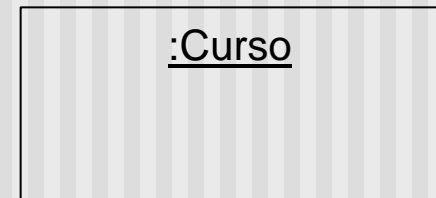
- Los objetos se dibujan como rectángulos con nombres subrayados



Objetos



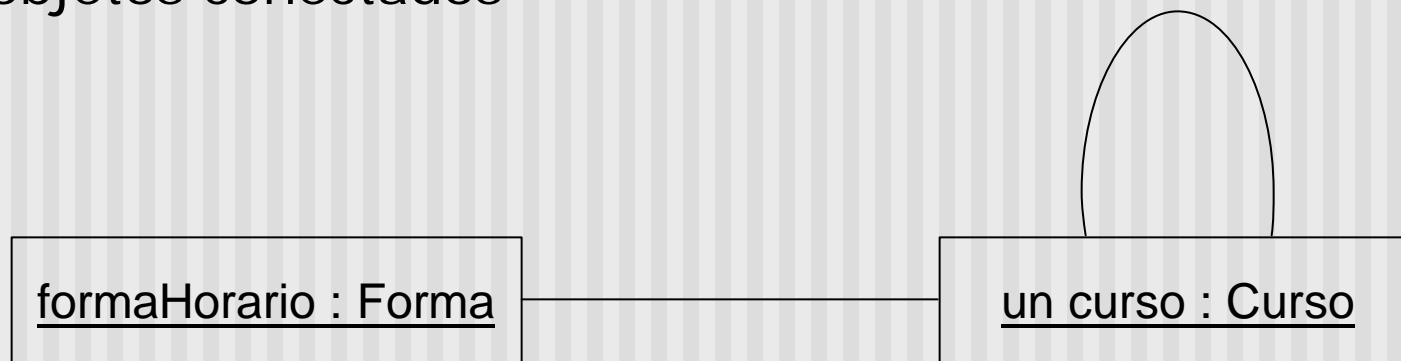
Objetos y Clases



Clases

Representación de Ligas en un Diagrama de Colaboración

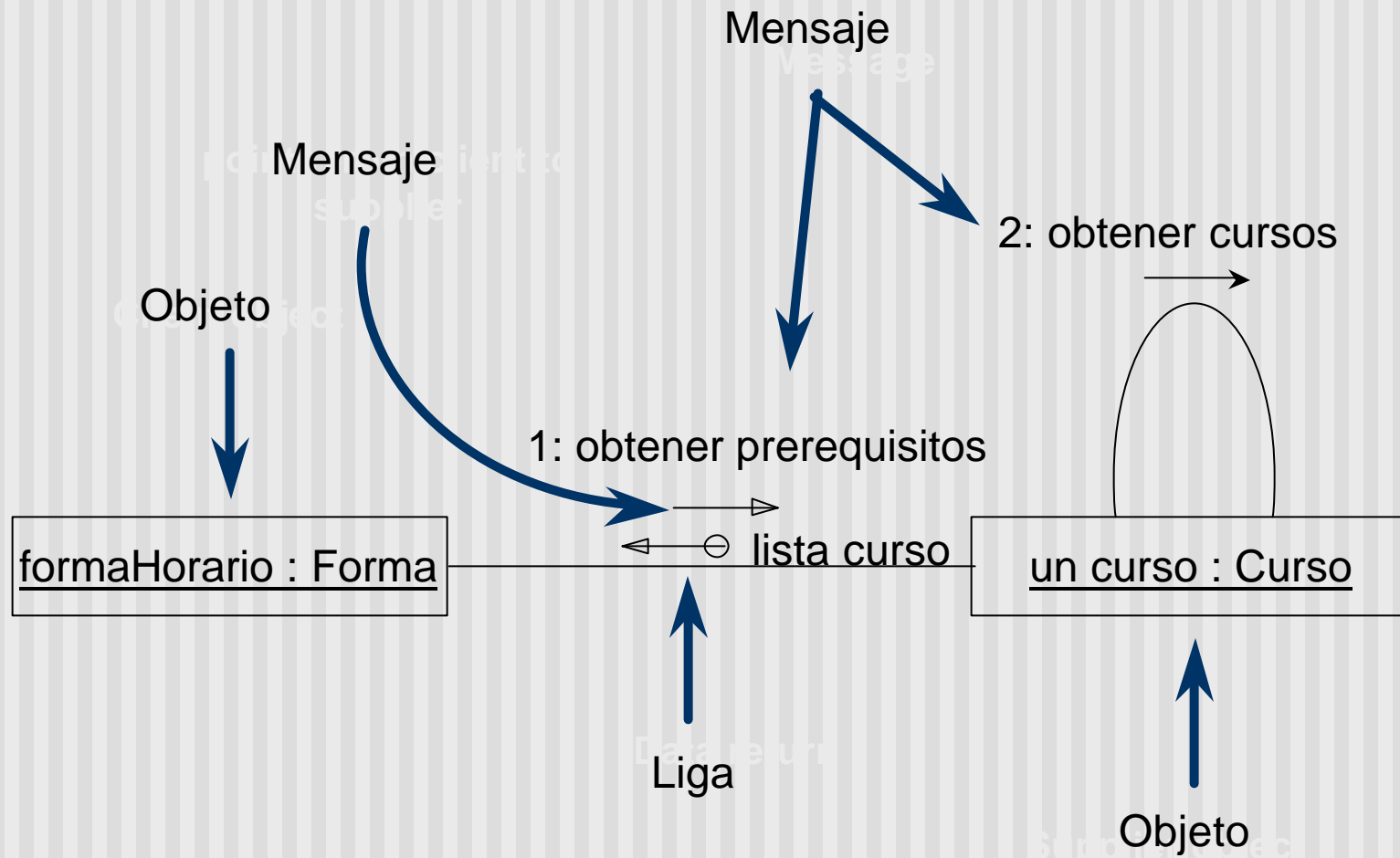
- Una liga de interacción en un diagrama de colaboración se representa como una línea que conecta iconos de objetos
- Una liga indica que hay una ruta de comunicación entre objetos conectados



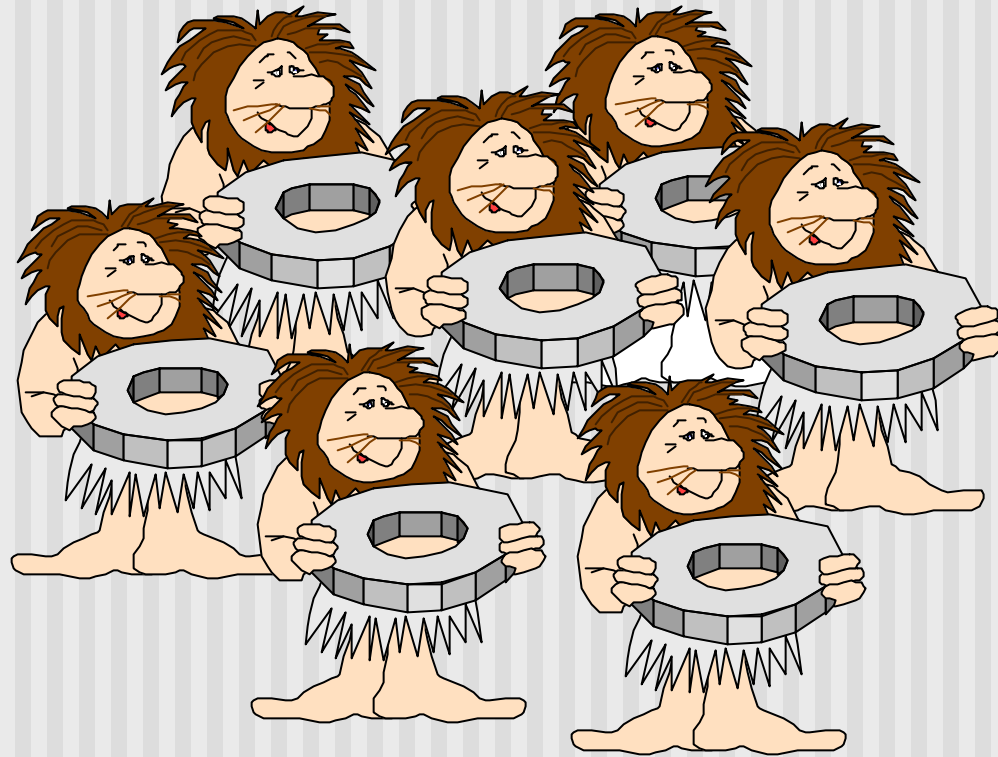
Anotaciones de Liga

- Una liga de interacción en un diagrama de colaboración se puede anotar con:
 - Una flecha apuntando del objeto cliente al objeto proveedor
 - El nombre del mensaje con una lista opcional de parámetros y/o valores de retorno
 - Un número opcional que muestre el orden relativo en el cual se envían los mensajes

Notación de Liga



Identificación de Clases



Objetivos: Identificación de Clases

- Usted podrá:
 - Discutir el análisis de casos de usos
 - Identificar objetos y clases llevando a cabo el análisis casos de usos
 - Usar tarjetas CRC para descubrir clases
 - Diagramar un escenario usando diagramas de interacción
 - Crear paquetes
 - Crear diagramas de clase iniciales

¿Qué es un Análisis Casos de Uso?

- El análisis casos de uso es el proceso de estudiar los casos de usos para descubrir objetos y clases para el desarrollo del sistema
 - Los escenarios se detallan y se representan gráficamente en diagramas de interacciones
 - **Se crean clases entity, boundary y control**
 - Las clases se agrupan en paquetes
 - Se crean diagramas de clases

Identificación de Objetos Entity

- Los objetos entity se identifican al examinar los sustantivos en los escenarios

- Los sustantivos encontrados pueden ser:
 - Objetos
 - Descripción del estado de un objeto
 - Entidades externas y/o Actores
 - Otros

Filtrado de Sustantivos

- Cuando se identifican sustantivos, debe estar consciente de que:
 - Varios términos se pueden referir al mismo objeto
 - Un término se puede referir a más de un objeto
 - El lenguaje natural es muy ambiguo
- Este acercamiento puede identificar muchos objetos sin importancia
 - La lista de sustantivos debe filtrarse

Observar los Sustantivos

- El siguiente expresión fue escrita para un sistema de bancario
 - “Los requerimientos legales se tomarán en cuenta”
- Si SOLO se consideraran los sustantivos, ¿qué pasaría?

Línea principal: Cada sustantivo debe considerarse en el contexto del dominio del problema -- no puede considerarse por sí mismo

Escenario: "Crear horario"

- John introduce el número id de alumno 369 52 3449 y el sistema valida el número. El sistema pregunta qué semestre. John indica el semestre actual y elige crear un horario.
- De una lista de cursos disponibles, John selecciona los cuatro cursos primarios English 101, Geology 110, World History 200, y College Algebra 110. Después selecciona los cursos alternos Music Theory 110 y Introduction to Java Programming 180.
- El sistema determina que John tiene todos los pre-requisitos necesarios al examinar el registro del alumno y lo agrega a la lista de los cursos.
- El sistema indica que la actividad se ha completado. El sistema imprime el horario del alumno y envía información de cobro para cuatro cursos al sistema de cobro para procesarlo.

Sustantivos del Escenario “Crear Horario”

- John
- Sistema
- Semestre
- Horario
- Geology 110
- College Algebra 110
- Cursos alternos
- Music Theory 110
- Prerequisitos necesarios
- Horario de estudiante
- Información de cobro
- Cuatro cursos
- Sistema de cobro
- Número de ID 369523449
- Número
- Semestre actual
- Lista de cursos disponibles
- Cursos primarios
- English 101
- Introduction to Java Programming 180
- World History 200
- Registro de estudiante
- Lista del curso
- Actividad

¿Qué sustantivos deben filtrarse?

Decisiones de Filtrado

- John -- filtrado (actor)
- Número de ID 369523449 -- filtrado (propiedad del alumno)
- Sistema -- filtrado (lo que se está construyendo)
- Número -- filtrado (lo mismo que el numero id del alumno)
- Semestre -- filtrado (estado -- cuando la selección aplica)
- Semestre actual -- filtrado (igual que el semestre)
- Horario -- candidato a objeto
- Lista de cursos disponibles -- candidato a objeto
- Cursos primarios -- filtrado (estado de un curso seleccionado)
- English 101 -- candidato a objeto
- Geology 110 -- candidato a objeto
- World History 200 -- candidato a objeto
- College Algebra 110 -- candidato a objeto

Decisiones de Filtrado

- Cursos alternos -- filtrado (estado de un curso seleccionado)
- Music Theory 110 -- candidato a objeto
- Introduction to Java Programming 180 -- candidato a objeto
- Prerequisitos necesarios -- filtrado (curso como otros cursos identificados)
- Registro de estudiante -- candidato a objeto
- Lista del curso -- candidato a objeto
- Actividad -- filtrado (Expresión en inglés)
- Horario de estudiante -- filtrado (igual que el nuevo horario)
- Información de cobro -- candidato a objeto
- Cuatro cursos -- filtrado (información necesitada por la información de cobro)
- Sistema de cobro -- filtrado (actor)

Candidatos a Objetos en el Escenario

- **Horario** -- lista de cursos por semestre para un alumno
- **Lista de cursos disponibles** -- lista de todos los cursos que se imparten en un semestre
- **English 101** -- una oferta para un semestre
- **Geology 110** -- una oferta para un semestre
- **World History 200** -- una oferta para un semestre
- **College Algebra 110** -- una oferta para un semestre
- **Music Theory 110** -- una oferta para un semestre
- **Introduction to Java Programming 180** -- una oferta para un semestre

Candidatos a Objetos en el Escenario

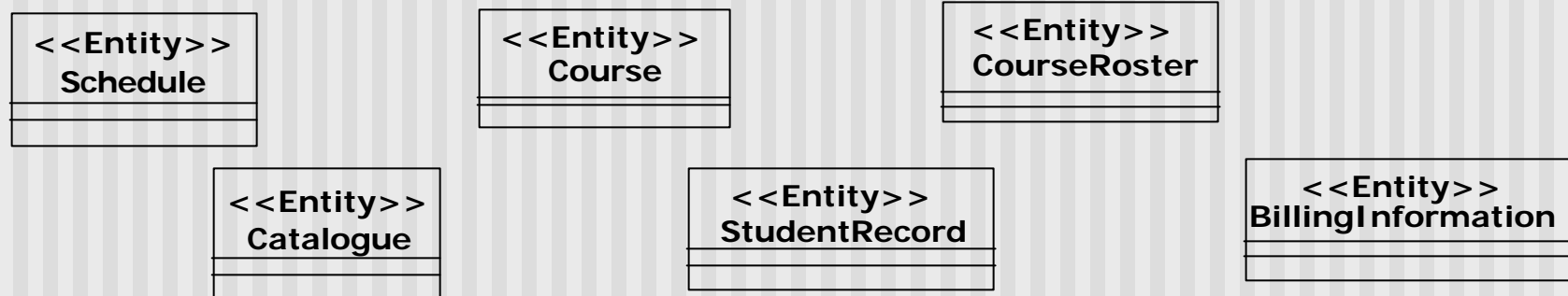
- **Registro de estudiante** -- una lista de cursos que el alumno tomo en semestres previos
- **Lista del curso** -- lista de alumnos para una oferta de curso específica
- **Información de cobro** -- información que necesita el actor sistema de cobro

Creación de Clases

- Los objetos entity encontrados se agrupan en clases
 - Basado en estructura y/o comportamiento similar
- Esto es sólo un intento inicial
 - Las clases pueden cambiar al examinar más escenarios

Clases Candidatas Entity -- Escenario "Crear Horario"

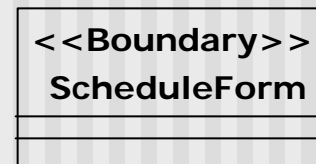
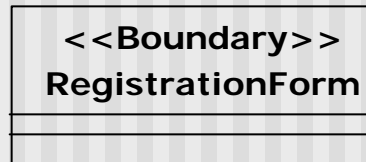
- Horario (**Schedule**) -- lista de cursos para un semestre para un alumno
- Catálogo (**Catalogue**) -- lista de todos los cursos que se imparten en un semestre
- Curso (**Course**) -- una oferta para un semestre
- RegistroEstudiante (**StudentRecord**) -- lista de cursos previamente tomados
- ListaCurso (**CourseRoster**) -- lista de alumnos para una oferta específica de curso
- InformacionCobro (**BillingInformation**) -- información necesitada por el actor sistema de cobro



Identificación de Clases Boundary

- Examinar cada par: actor/escenario y crear clases boundary obvias
 - Durante el diseño, la clase se refinará en base a los mecanismos de la interfaz de usuario elegida
- Ejemplo:
 - Al alumno se le presentan diferentes opciones en el caso de uso "Inscripción a Cursos"
 - Se crea una clase boundary llamada **RegistrationForm** para permitir que el alumno seleccione la opción deseada
 - El alumno debe introducir información del curso al sistema en el escenario "Crear Horario"
 - Se crea una clase boundary llamada **ScheduleForm** para mantener la información que el alumno introduce

Clases Candidatas Boundary -- Escenario "Crear Horario"



Prototipo de Ventana

- Los prototipos de ventanas pueden crearse para comunicar la apariencia y percepción de la clase boundary al usuario

The image shows a window prototype for a 'Course Registration' application. The window has a title bar with the text 'Course Registration' and standard window control buttons (minimize, maximize, close). Below the title bar is a menu bar with 'File' and 'Help' menus. The main content area contains the following elements:

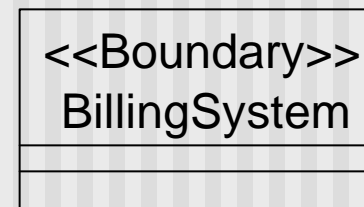
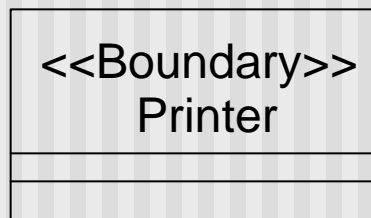
- A label 'Name' followed by a single-line text input field.
- A label 'ID Number' followed by a single-line text input field.
- A label 'Semester' followed by two radio buttons labeled 'Current' and 'Next'.
- A label 'Options' followed by a list box containing four items: 'Create schedule', 'Delete schedule', 'Modify schedule', and 'Review schedule'.
- At the bottom right, there are two buttons: 'Cancel' and 'OK'.

Identificación de Clases Boundary

- Las clases boundary también se crean por la comunicación de sistema-a-sistema
 - Puede ser otro sistema de software o una pieza de hardware (impresoras, alarmas, etc.)
- Las clases boundary se agregan para describir el protocolo de comunicación elegido

Clases Candidatas Boundary -- Escenario "Crear Horario"

- El horario del alumno se imprime en el escenario "Crear Horario"
 - Se crea una clase boundary Printer
- La información de cobro se envía al Sistema de Cobro en el escenario "Crear Horario"
 - Se crea una clase boundary BillingSystem



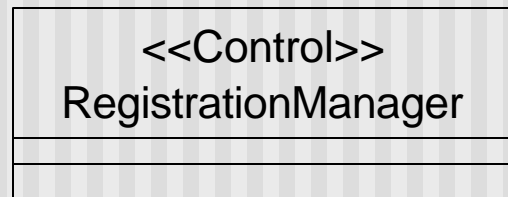
Identificación de Clase Control

- Las clases control contienen típicamente información secuencial
 - Precaución: las clases control NO deben desempeñar las responsabilidades que pertenecen típicamente a las clases entity y/o boundary
- En este nivel de análisis, una clase control se agrega típicamente para cada caso de uso
 - Responsable por el flujo de eventos en el caso de uso
- Este es sólo un breve inicio
 - Cuanto más casos de uso y escenarios se desarrollen, pueden eliminarse, dividirse o combinarse las clases de control

Reglas para el Caso de Uso "Inscripción a Cursos"

- Se agrega una clase control llamada **RegistrationManager**
 - Recibe información de la clase boundary **ScheduleForm**
 - Para cada curso seleccionado
 - Pide los **prerequisitos** del curso
 - Revisa para asegurarse de que todos los **prerequisitos** de un curso se tomaron al preguntar a **StudentRecord** si un **prerequisito** de curso se había completado
 - Sabe que hacer si no se tiene un **prerequisito**
 - Pregunta si el curso está abierto
 - Pide a **Course** que agregue al alumno (si el curso está abierto)
 - Sabe que hacer si no están disponibles 4 cursos
 - Crea los objetos **StudentSchedule** y **BillingInformation**
 - Pide al **BillingSystem** que envíe la **BillingInformation**

Clase Candidata Control -- Caso de Uso "Inscripción a Cursos"



Tarjetas Responsabilidad-Colaboración de Clases

- Las clases también pueden descubrirse usando tarjetas responsabilidad-colaboración de clases (Class-Responsibility-Collaboration Cards, CRC)
 - **Introducidas por Ward Cunningham y Kent Beck at OOPSLA en 1989**
- Una tarjeta CRC es una tarjeta de 3" x 5" que muestra
 - **El nombre y descripción de la clase**
 - **Las responsabilidades de la clase**
 - Conocimiento interno de la clase
 - Servicios proporcionados por la clase
 - **Los colaboradores para las responsabilidades**
 - Un colaborador es una clase cuyos servicios necesitan una responsabilidad

Tarjeta CRC para la Clase Curso

Nombre de Clase Curso	
Responsabilidades	Colaboraciones
Agregar un alumno	Alumno
Conocer los pre-requisitos	
Saber cuando se lleva a cabo	
Saber donde se lleva a cabo	

Servicio proporcionado ←

Conocimiento interno ←

Una sesión de tarjeta CRC

- Un grupo de personas ejecutan un escenario
- Se crea una tarjeta para cada objeto en el escenario
- Se le asigna un grupo de tarjetas a cada participante
 - La persona se convierte en la "clase"
- Los participantes actúan a los escenarios definidos
- Se anotan responsabilidades y colaboraciones en las tarjetas
- Se crean tarjetas para los objetos descubiertos

Beneficios de las Tarjetas CRC

- Al completar más y más escenarios, emergen los patrones de colaboración
- Las tarjetas pueden arreglarse físicamente para representar estas colaboraciones cerradas
- Esto puede ayudar a identificar jerarquías de generalización/especialización o jerarquías de agregación entre las clases
- Las tarjetas CRC son más efectivas para grupos que desconocen las técnicas OO, ya que ellos:
 - Evitan enfocarse a elementos OOP
 - Evitan generalización prematura
 - Fortalecen "object think"

¿Cómo lo estoy haciendo?

- Las cosas van bien si...
 - Todas las clases tienen nombre significativos, específicos del dominio
 - Cada clase tiene un pequeño grupo de colaboradores
 - No hay clases “indispensables” (una clase que colabora con todos necesita ser redefinida)
 - La información para cada clase se ajusta bien en una tarjeta de 3X5
 - Las clases pueden manejar un cambio en requerimientos
- Las cosas van mal si...
 - Varias clases no tienen responsabilidades
 - Una sola responsabilidad se le asigna a varias clases
 - Todas las clases colaboran con todas las clases

Diagramación de Escenarios

- Al descubrir objetos y clases, se documentan en diagramas de interacción
 - Estos diagramas puede ser, un diagrama de secuencias o un diagrama de colaboración
- Los diagramas de interacción contienen el flujo de eventos para un escenario dado
 - Los nombres de objetos son generales
 - e.g., un alumno en lugar de John
 - Los nombres de objetos pueden omitirse si no se necesitan para la comunicación
 - Se agregan notas y/o scripts de ser necesario

Diagrama de Secuencias para el Escenario "Crear Horario"

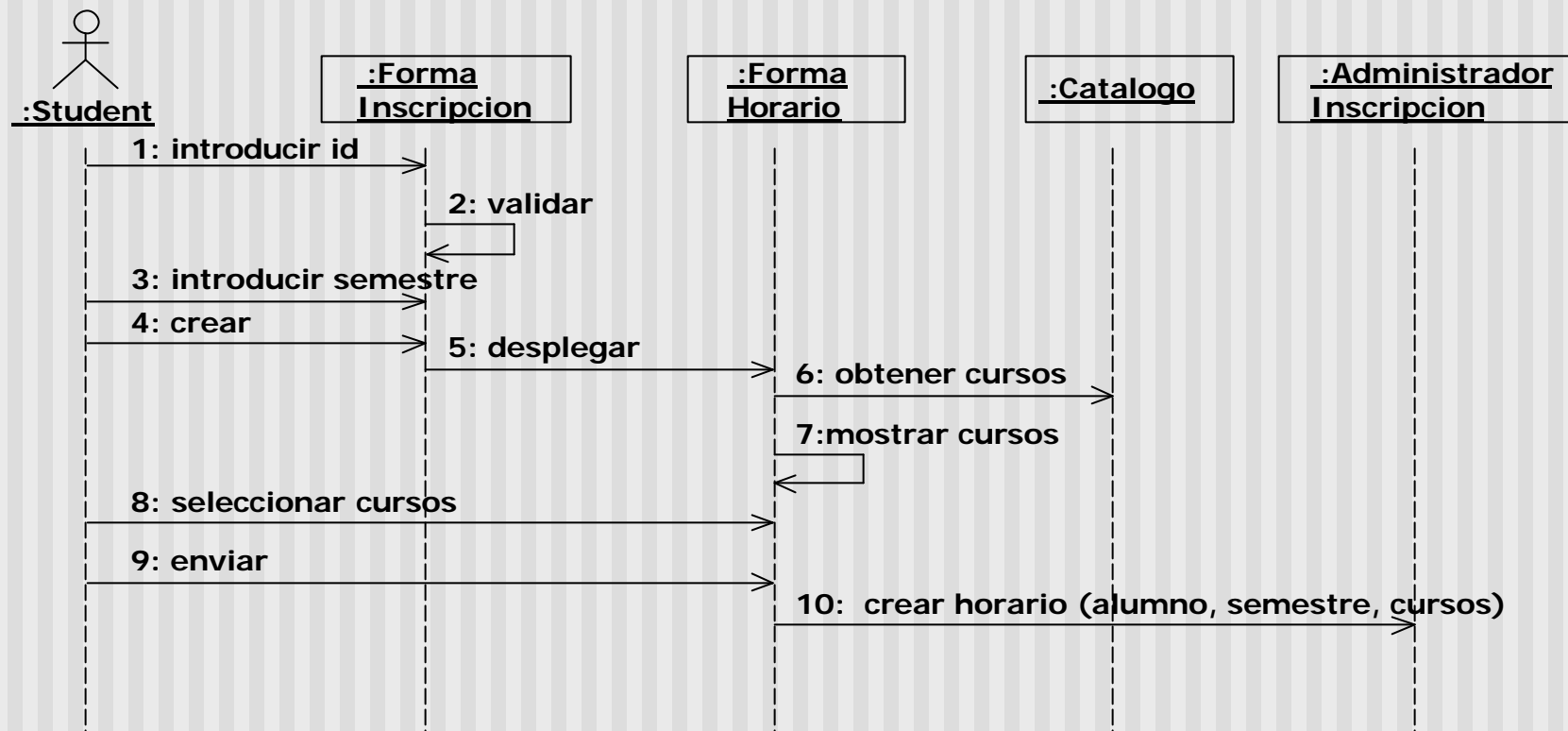


Diagrama de Secuencias para el Escenario "Crear Horario" (cont.)

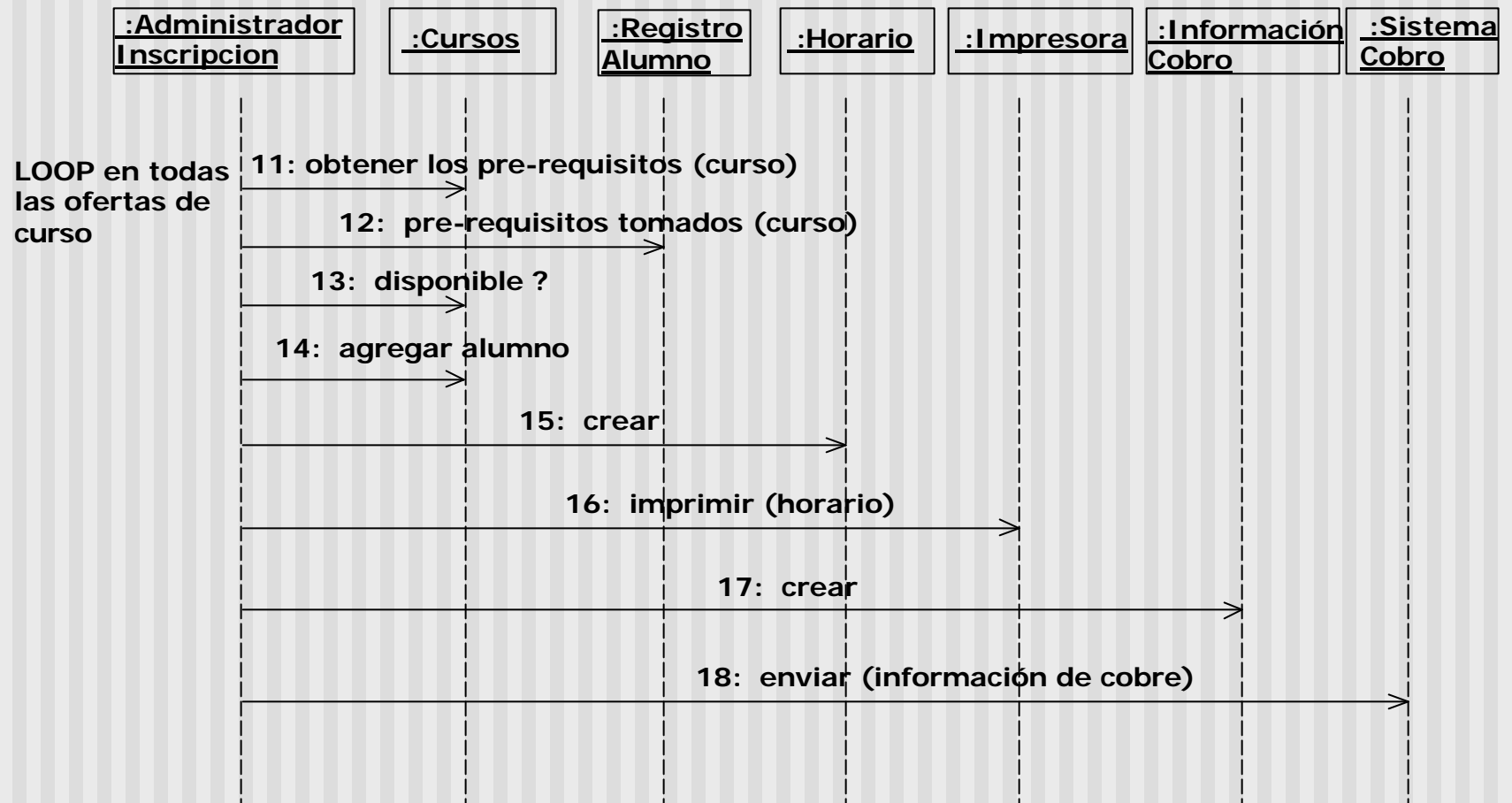
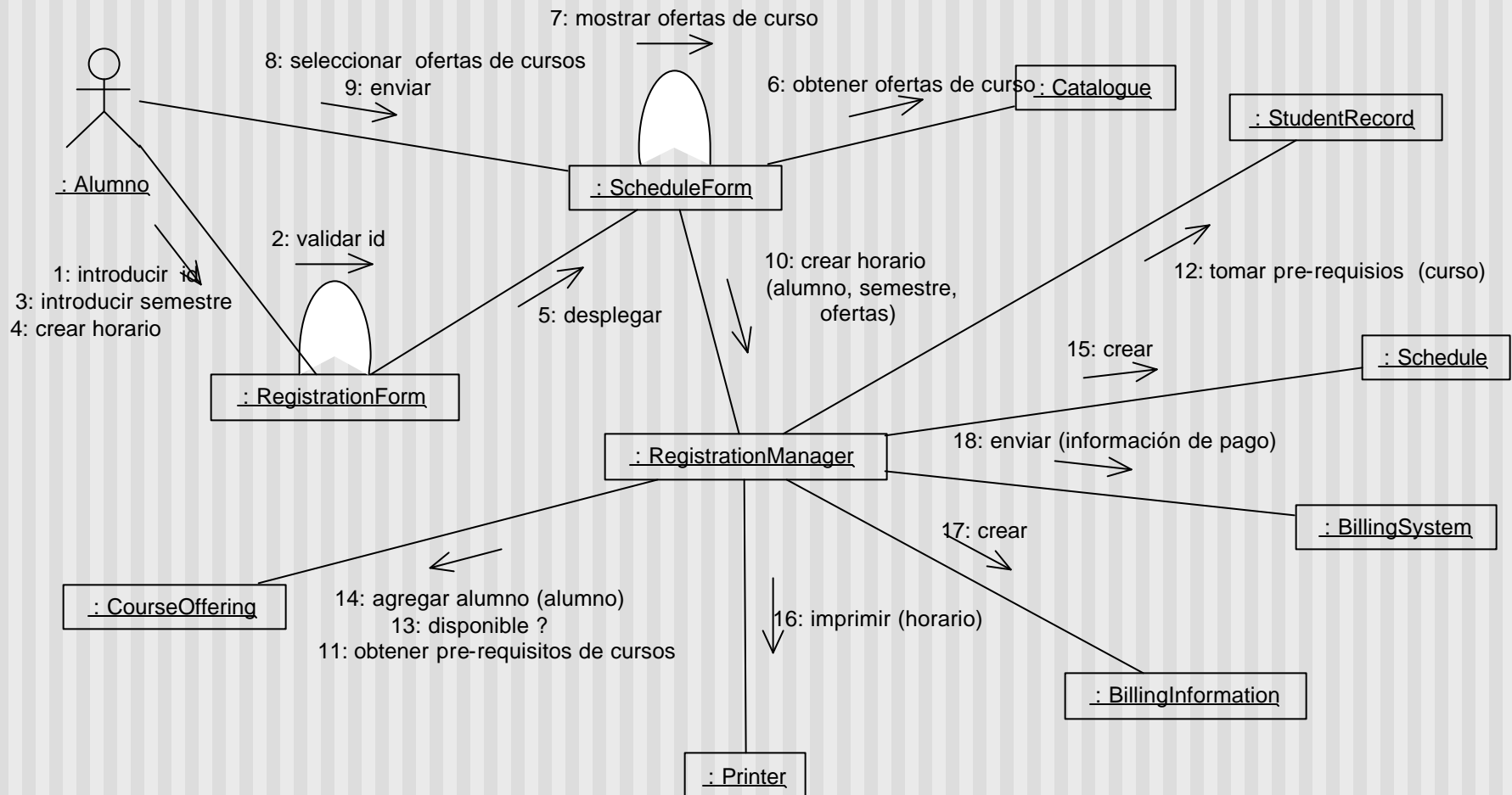
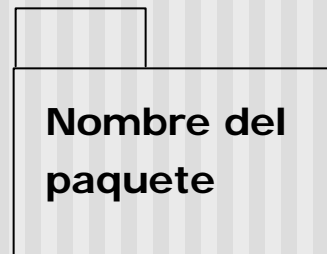


Diagrama de Colaboración para el Escenario "Crear Horario"



¿Qué es un Paquete?

- Un paquete es un mecanismo de propósito general para organizar elementos en grupos
- El número de clases crecen al analizar los casos de uso y los escenarios
 - Las clases pueden agruparse en paquetes
 - **Proporcionan la habilidad para organizar el modelo en desarrollo**
- Un paquete se representa como una carpeta etiquetada



Paquetes en el Sistema de Inscripción

- Las clases en el Sistema de Inscripción se pueden agrupar en tres paquetes:
 - **University Artifacts**, **Business Rules**, e **Interfaces**
- **UniversityArtifacts**
 - Schedule, Catalogue, CourseOffering, StudentRecord, CourseRoster, y Billing Information
- **BusinessRules**
 - RegistrationManager
- **Interfaces**
 - RegistrationForm, ScheduleForm, BillingSystem, and Printer

¿Qué es un Diagrama de Clases?

- La vista lógica se hace de varios paquetes y clases
- Un diagrama de clases es la vista lógica de algunos (o todos) los paquetes y clases
 - **Generalmente hay varios diagramas de clase**
 - El diagrama de clases principal es típicamente una vista lógica de los paquetes a alto nivel
- Cada paquete posee su propio diagrama de clases principal
- Los diagramas de clases adicionales se agregan como sea necesario
 - **Vista de las clases participando en un escenario**
 - **Vista de las clases “privadas” en el paquete**
 - **Vista de una clase, sus atributos y operaciones**
 - **Vista de una jerarquía de herencia**

Diagrama de Clases Principal para el Sistema de Inscripción

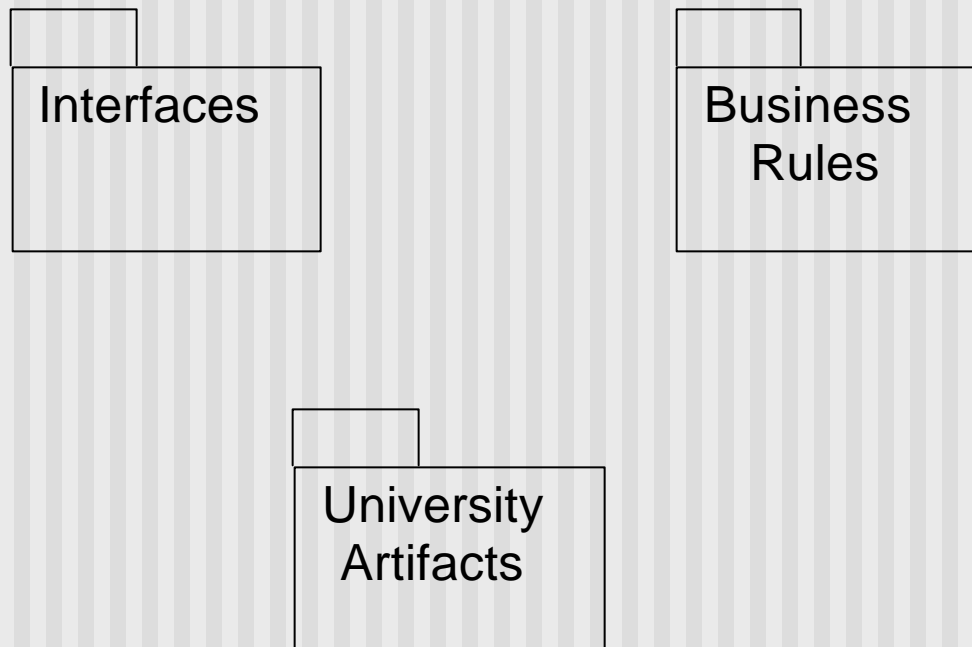


Diagrama de Clases Principal de University Artifacts

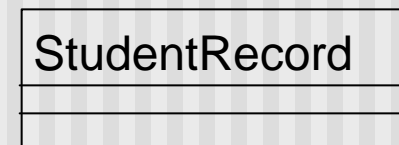
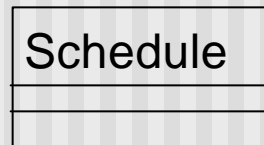
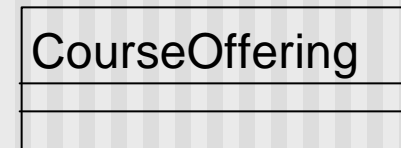
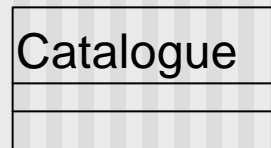
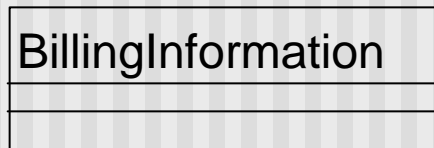


Diagrama de Clases Principal de Interfaces

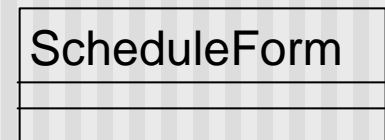
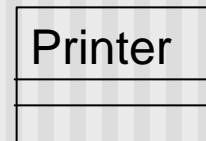
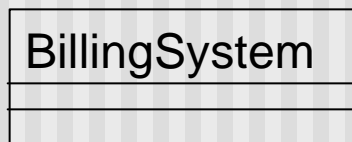
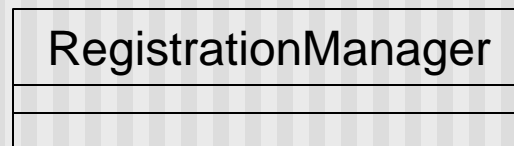


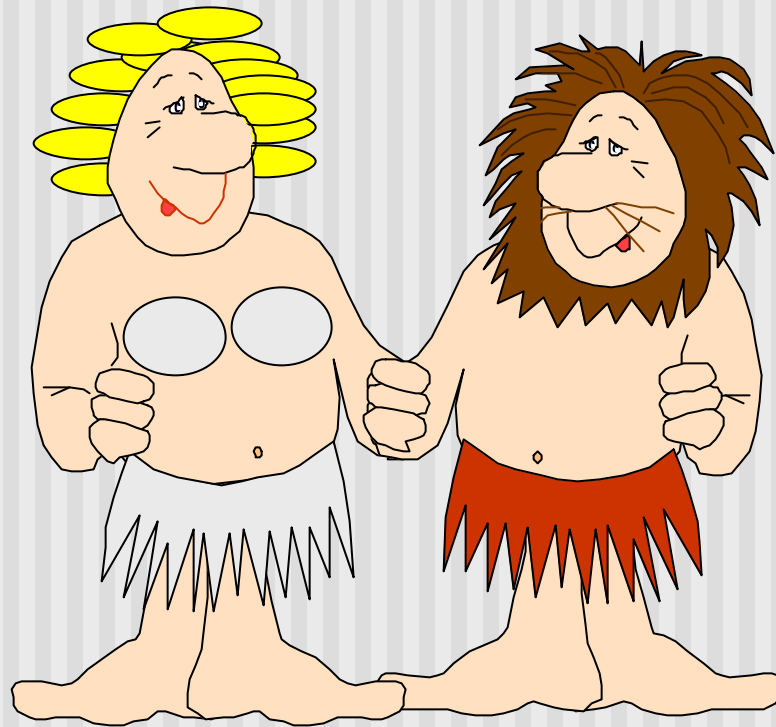
Diagrama de Clases Principal de Business Rules



Ejercicio: Identificación de Clases

- Tome un caso de uso desarrollado en la lección previa
 - Diagrame al menos un escenario en un diagrama de interacción
 - Cree clases entity, boundary y/o control que sean necesarias
 - Escriba una definición para cada clase
- Cree paquetes para el modelo
- Coloque las clases descubiertas en paquetes
- Cree diagramas de clase iniciales

Relaciones



Objetivos: Relaciones

■ Usted podrá:

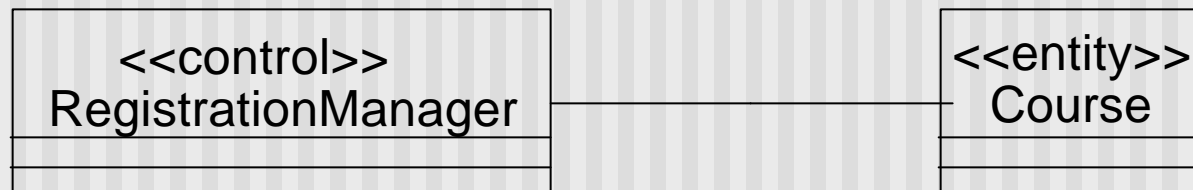
- Nombrar los dos importantes tipos de relaciones entre clases: asociación y agregación
- Definir asociación y representarla en diagramas de clases
- Usar nombres de asociación y nombres de rol para clarificar las asociaciones
- Definir y especificar la multiplicidad de una asociación
- Definir agregación y representarla en diagramas de clases
- Definir y representar una asociación reflexiva o agregada
- Usar clases de asociación
- Definir calificadores y representarlos en diagramas de clases
- Descubrir relaciones a partir de los diagramas de escenario

La Necesidad de Relaciones

- Todos los sistemas abarcan varias clases y objetos
- Los objetos contribuyen al comportamiento del sistema colaborando unos con otros
 - La colaboración se realiza a través de las relaciones
- Hay dos importantes tipos de relaciones durante el análisis
 - Asociación
 - Agregación

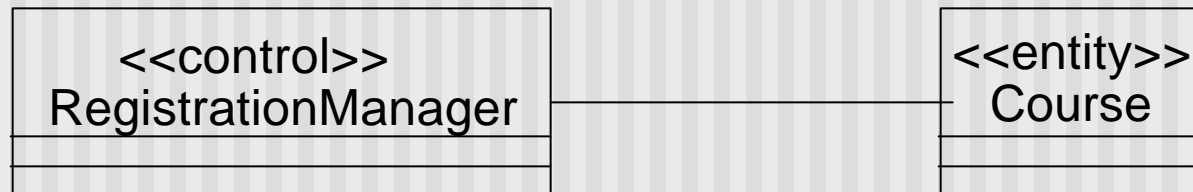
Asociaciones

- Una asociación es una conexión semántica bi-direccional entre clases
 - Esto implica que hay una liga entre objetos entre las clases asociadas
- Las asociaciones se representan en diagramas de clase por una línea que conecta las clases asociadas
- La información puede fluir en cualquier dirección o en ambas direcciones a través de la liga



Navegación

- Una asociación es una relación bi-direccional
 - Dada una instancia de RegistrationManager hay un objeto asociado Course
 - Dada una instancia de Course hay un objeto asociado RegistrationManager



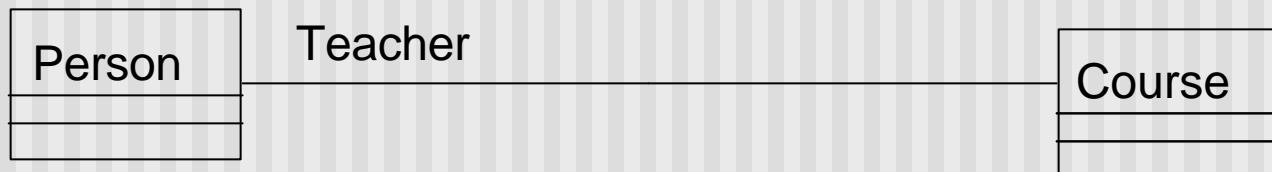
Nombrando Asociaciones

- Una asociación se debe nombrar para esclarecer su significado
- El nombre se representa con una etiqueta que se pone a lo largo de la línea de asociación, entre los iconos de clases
- Un nombre de asociación es generalmente un verbo o una frase con verbo



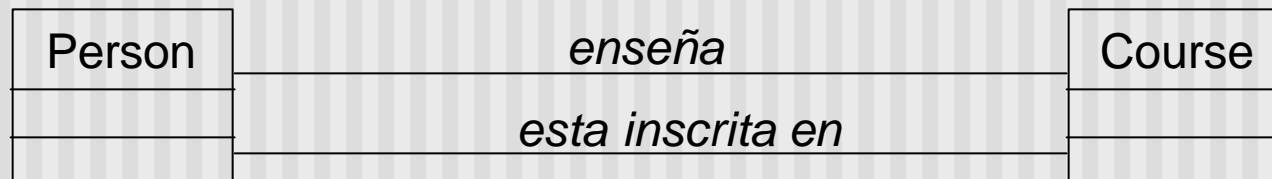
Definición de Roles

- Un rol denota el propósito o capacidad en la que una clase se asocia con otra
- Los nombres de roles son típicamente sustantivos o frases con sustantivo
- Un nombre de rol se pone a lo largo de la línea de asociación cerca de la clase que modifica
 - En uno o en ambos extremos de una asociación se pueden tener roles



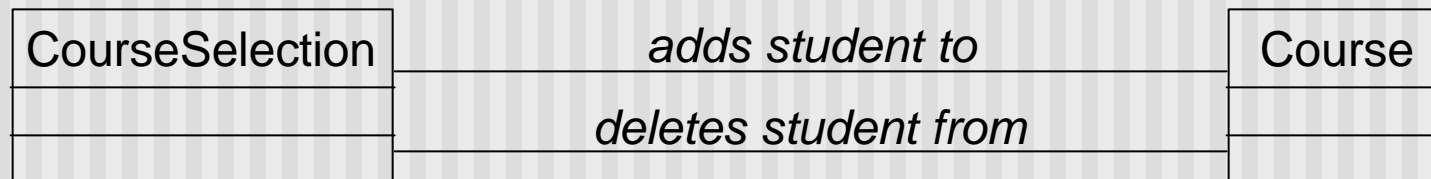
Asociaciones Múltiples

- Puede existir más de una asociación entre dos clases
- Si hay más de una asociación entre dos clases se les DEBE de nombrar



- Las asociaciones múltiples deben cuestionarse

Asociaciones Múltiples (cont.)



¿Modelo bueno o malo?

Multiplicidad para Asociaciones

- Multiplicidad es el número de instancias de una clase relacionada a UNA instancia de otra clase
- Para cada asociación, hay dos decisiones de multiplicidad que tomar: una por cada extremo de la asociación
- Por ejemplo, en la conexión entre Person jugando el rol maestro y Course
 - Para cada instancia de Person, varios (i.e., cero o más) Courses deben impartirse
 - Para cada instancia de Course, exactamente una instancia de Person es maestro

Indicadores de Multiplicidad

- Cada extremo de una asociación tiene indicadores de multiplicidad
 - Indica el numero de objetos que participan en la relación

Muchos

*

Exactamente uno

1

Cero o más

0..*

Uno o más

1..*

Cero o uno

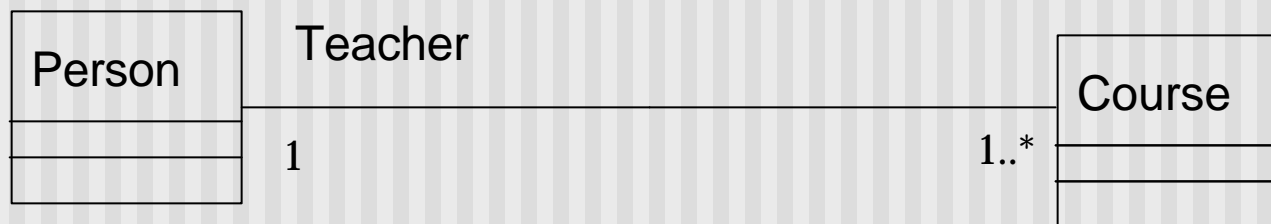
0..1

Rango específico

2..4

Ejemplo: Multiplicidad

- La multiplicidad expone varias hipótesis ocultas sobre el problema que se está modelando
 - ¿Puede estar un maestro en sabático?
 - ¿Puede tener un curso dos maestros?



¿Qué significa Multiplicidad?

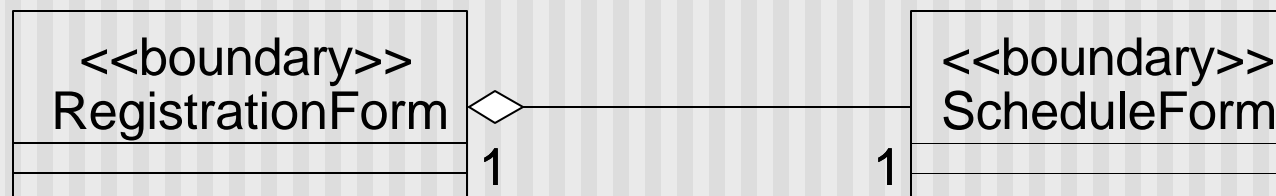
- La multiplicidad debe responder a dos preguntas
 - ¿La asociación es obligatoria u opcional?
 - ¿Cuál es el número mínimo y máximo de instancias que pueden ligarse a una instancia?



¿Qué le dice este diagrama?

Agregación

- La agregación es una forma especializada de asociación en la que un todo se relaciona con sus partes
 - La agregación es conocida como la relación “parte de” o “contiene a”
- Una agregación se representa como una asociación con un diamante en el extremo de la liga, del lado de la clase que denota el agregado (todo)
- La multiplicidad se representa de la misma manera que otras asociaciones

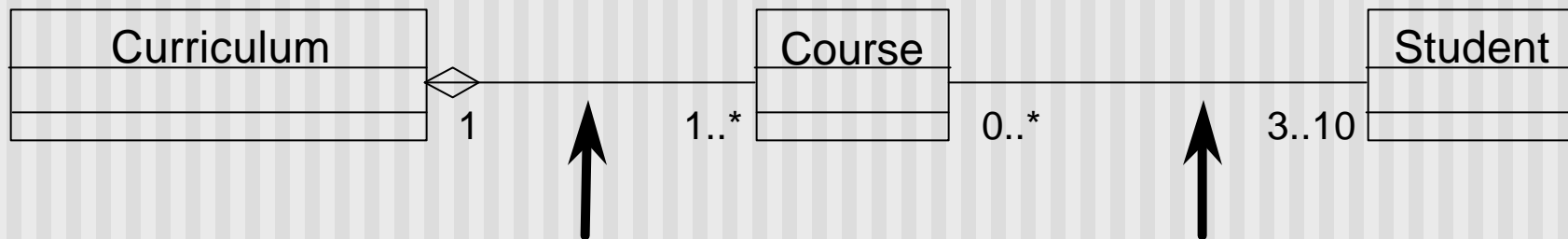


Pruebas de Agregación

- ¿Se usa la frase “parte de” para describir relaciones?
 - **Una Puerta es “parte de” un Carro**
- ¿Se aplican algunas operaciones en el todo y automáticamente a sus partes?
 - **Al mover el Carro, se mueve la Puerta**
- ¿Se propagan algunos valores de atributos del todo a todas o algunas de sus partes?
 - **El Carro es azul, la Puerta es azul**
- ¿Hay una asimetría intrínseca a la relación donde una clase se subordina a la otra?
 - **Una Puerta es parte de un Carro, un Carro NO es parte de una Puerta**

¿Asociación o Agregación?

- Si dos objetos están estrictamente limitados por una relación complementaria
 - La relación es un agregación
- Si dos objetos se consideran usualmente como independientes, y aún así están ligados
 - La relación es una asociación

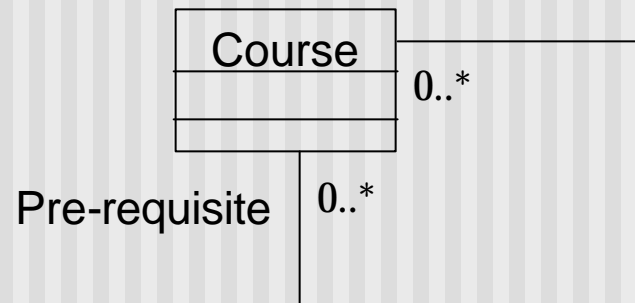


Curriculum y Course están muy ligados -- Curriculum está compuesto de 1 a muchos Courses

Objetos independientes

Asociaciones Reflexivas

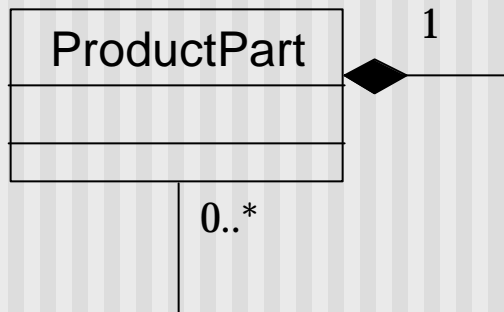
- En una asociación reflexiva, se relacionan los objetos de la misma clase
 - Indica que múltiples objetos en la misma clase colaboran juntas en otra forma



Un curso puede tener muchos pre-requisitos
Un curso puede ser pre-requisito de otros cursos

Agregaciones Reflexivas

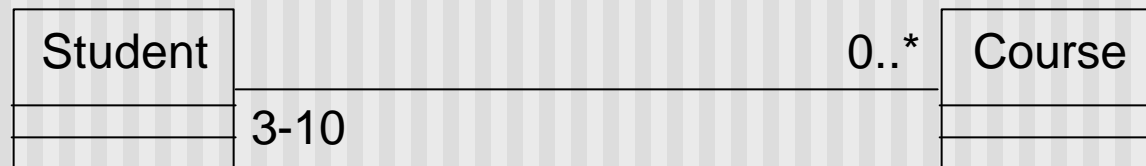
- Las agregaciones pueden también ser reflexivas
 - El tipo de problema clásico de productos y sus partes
- Esto indica una relación recursiva



Un objeto ProductPart está “compuesto de”
cero o más objetos ProductPart

Clase de Asociación

- Si quisiéramos rastrear los grados para todos los cursos que un alumno ha tomado, entonces...
- La relación entre Student y Course es una relación de muchos-a-muchos
- ¿Dónde ponemos el atributo de grado?

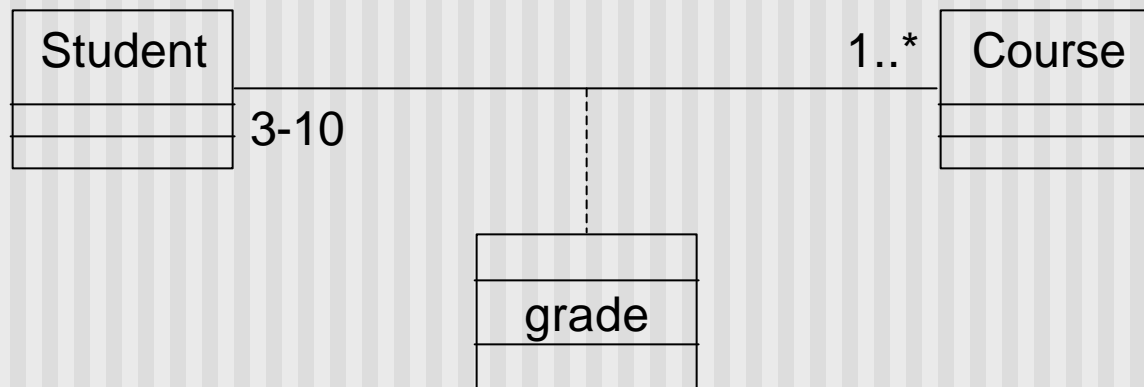


Clase de Asociación (cont.)

- El atributo grado no puede ponerse en la clase Course porque existen (potencialmente) varias ligas a objetos Student
- El atributo grado no puede ponerse en la clase Student porque existen (potencialmente) varias ligas a objetos Course
- Por lo tanto, el atributo en realidad pertenece a la liga Student-Course
- Una clase de asociación se usa para mantener dicha información

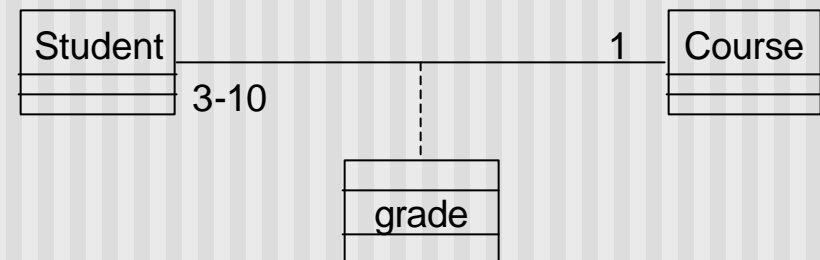
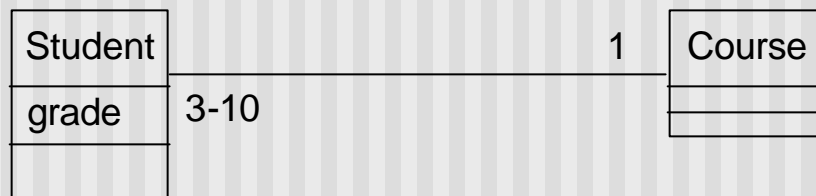
Dibujo de Clase de Asociación

- Se crea una clase de asociación usando el icono clase
- Se conecta el icono de la clase a la línea de asociación con una línea punteada
- La clase de asociación puede incluir múltiples propiedades de la asociación
- Sólo se permite una clase de asociación por asociación



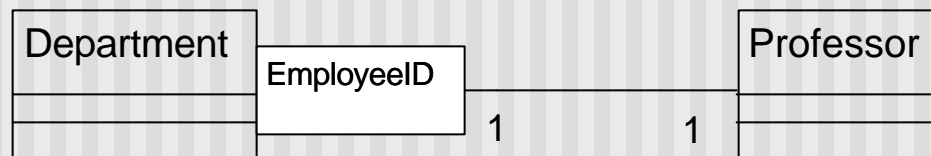
Clase de Asociación y Multiplicidad

- Las clases de asociación se emplean en asociaciones de muchos-a-muchos
- Si la multiplicidad en cualquiera de los extremos de una asociación es "a-una"
 - El atributo puede ponerse en la clase donde la relación es "a muchos", o
 - Puede aún usarse una clase asociación

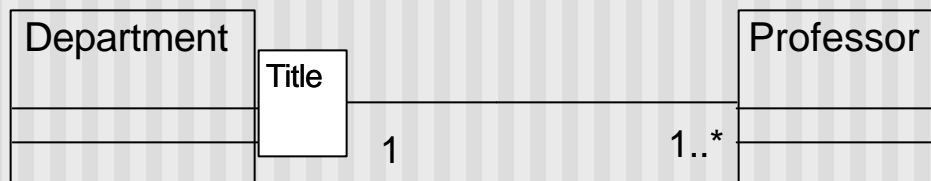


Calificadores

- Un calificador es un atributo o grupo de atributos cuyos valores dividen el conjunto de objetos relacionado a un objeto a través de una asociación



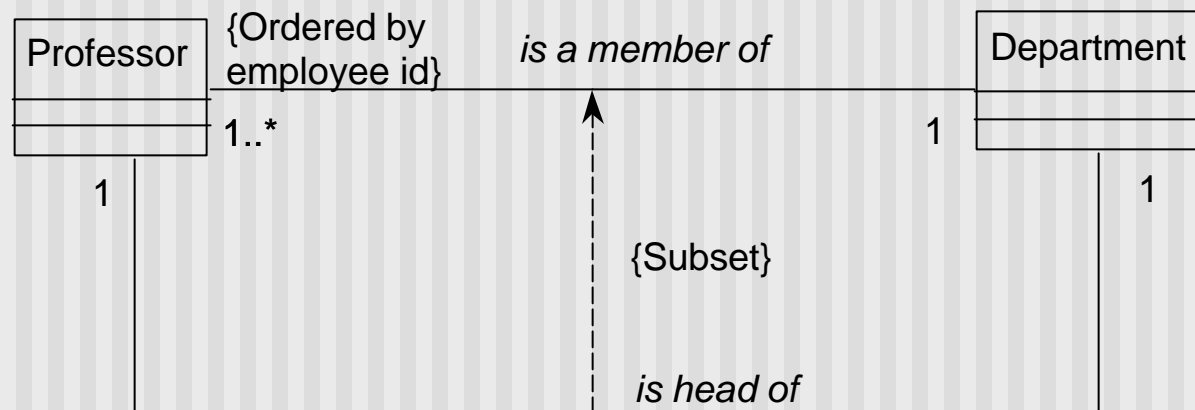
Dado un objeto Department y un valor para un Employee ID hay exactamente un objeto Professor



Dado un objeto Department y un valor para Title hay un conjunto de objetos Professor

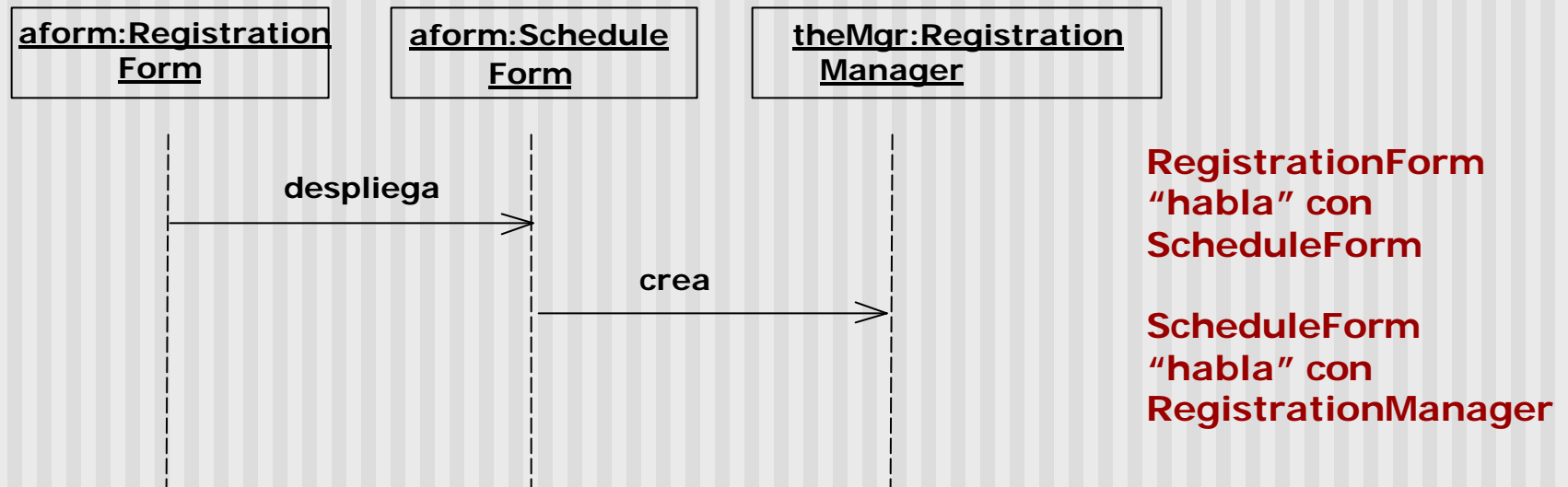
Restricciones

- Una restricción es la expresión de alguna condición que se debe preservar
 - Una restricción se muestra como una línea punteada



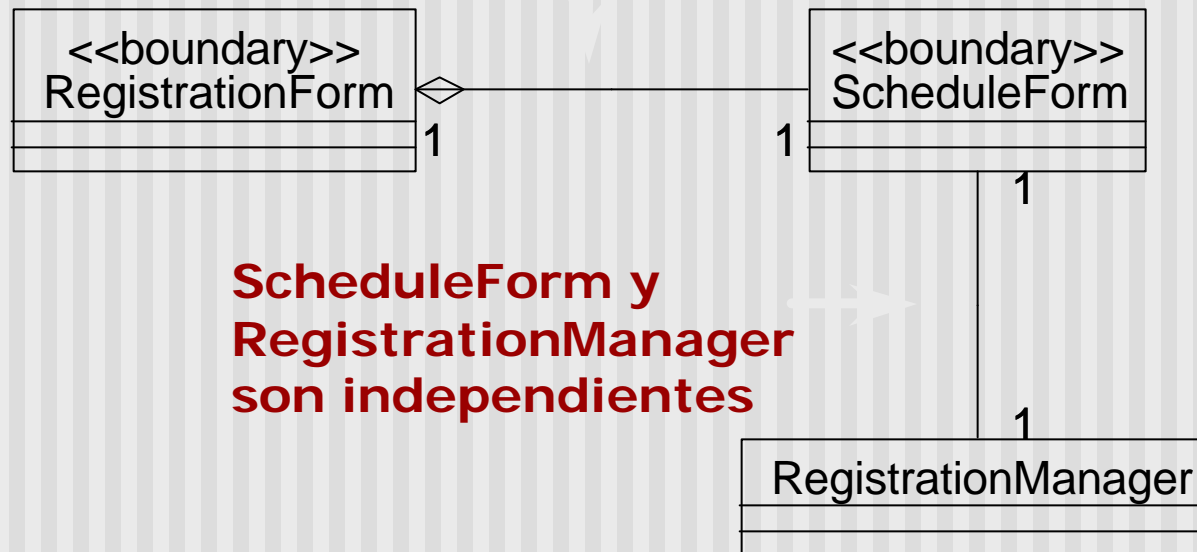
Identificación de Asociaciones y Agregaciones

- Deben examinarse los escenarios para determinar si una relación debe existir entre dos clases
 - Dos objetos pueden comunicarse solo si se “conocen” entre ellos
- Las asociaciones y/o agregaciones proporcionan una ruta de comunicación



¿Asociación o Agregación?

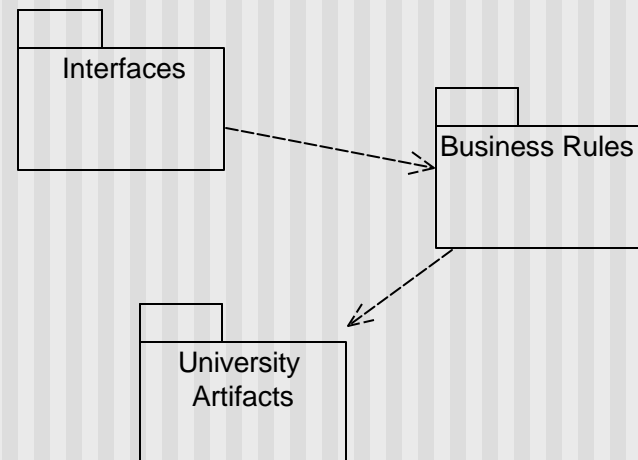
RegistrationForm y ScheduleForm están muy ligadas
-- una ScheduleForm es "parte de" la RegistrationForm



**ScheduleForm y
RegistrationManager
son independientes**

Relaciones de Paquetes

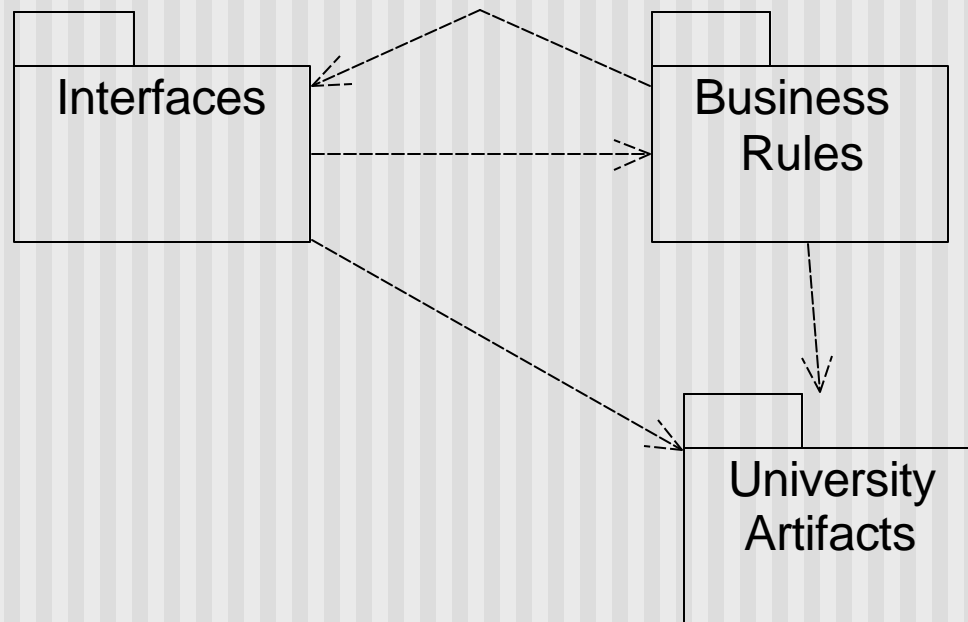
- Los paquetes se relacionan unos con otros a través de una relación de dependencia
- Si una clase en un paquete “habla” con una clase en otro paquete entonces se agrega una relación de dependencia al nivel del paquete
- Los diagramas de escenario y de clases se evalúan para determinar las relaciones entre paquete



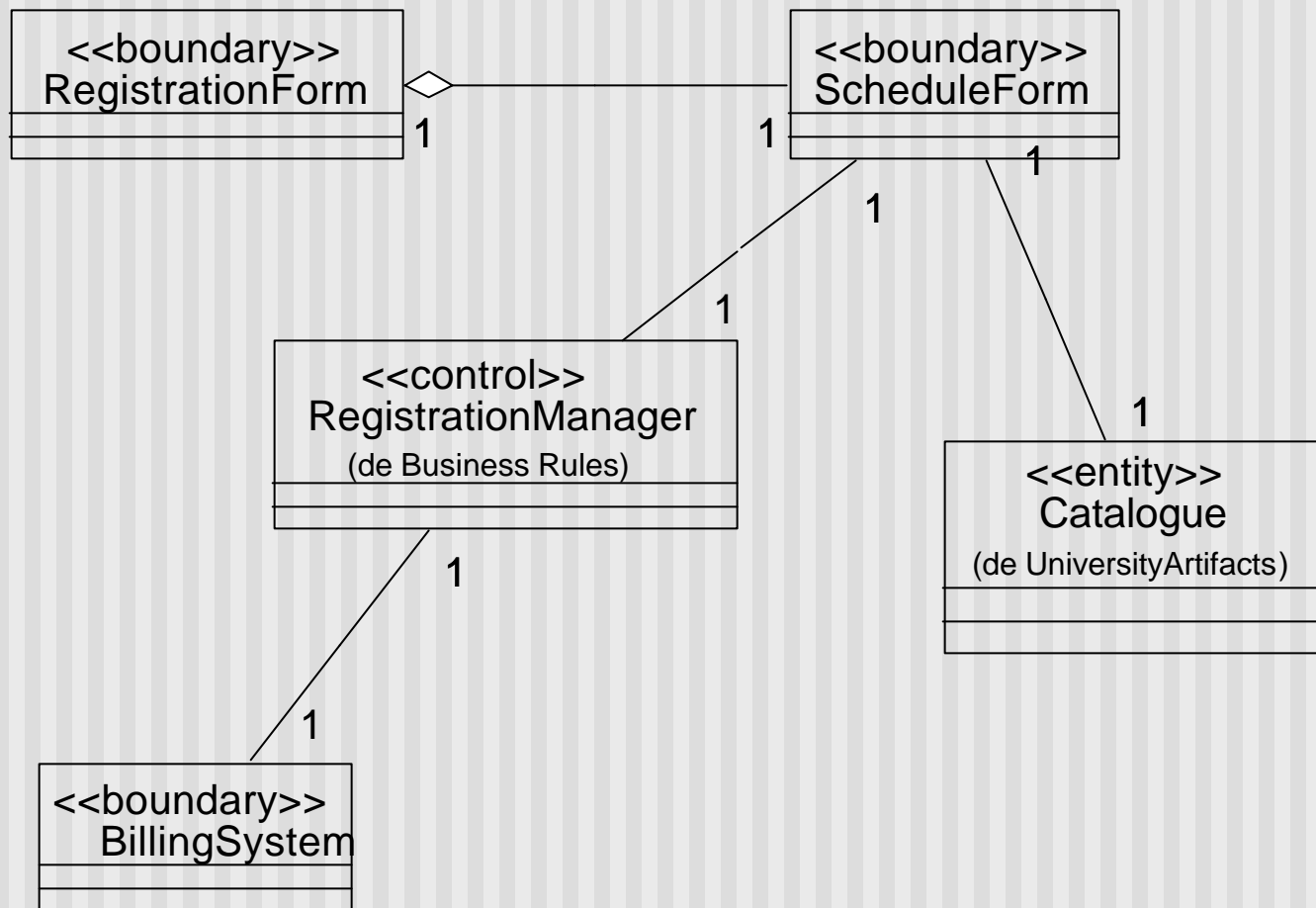
Relaciones en el Análisis y Diseño

- Durante el análisis, se establecen conexiones (asociaciones y agregaciones) entre clases
 - Estas conexiones existen debido a la naturaleza de las clases y no debido a una implementación específica
 - Hacer una estimación inicial de multiplicidad para exponer hipótesis ocultas
- Los diagramas de clases se actualizan para mostrar las relaciones agregadas
- Durante el diseño:
 - Se refinan y actualizan las estimaciones de multiplicidad
 - Se avalúan y refinan las asociaciones y agregaciones
 - Se evalúan y refinan las relaciones de paquetes
 - Se maduran los diagramas de clases

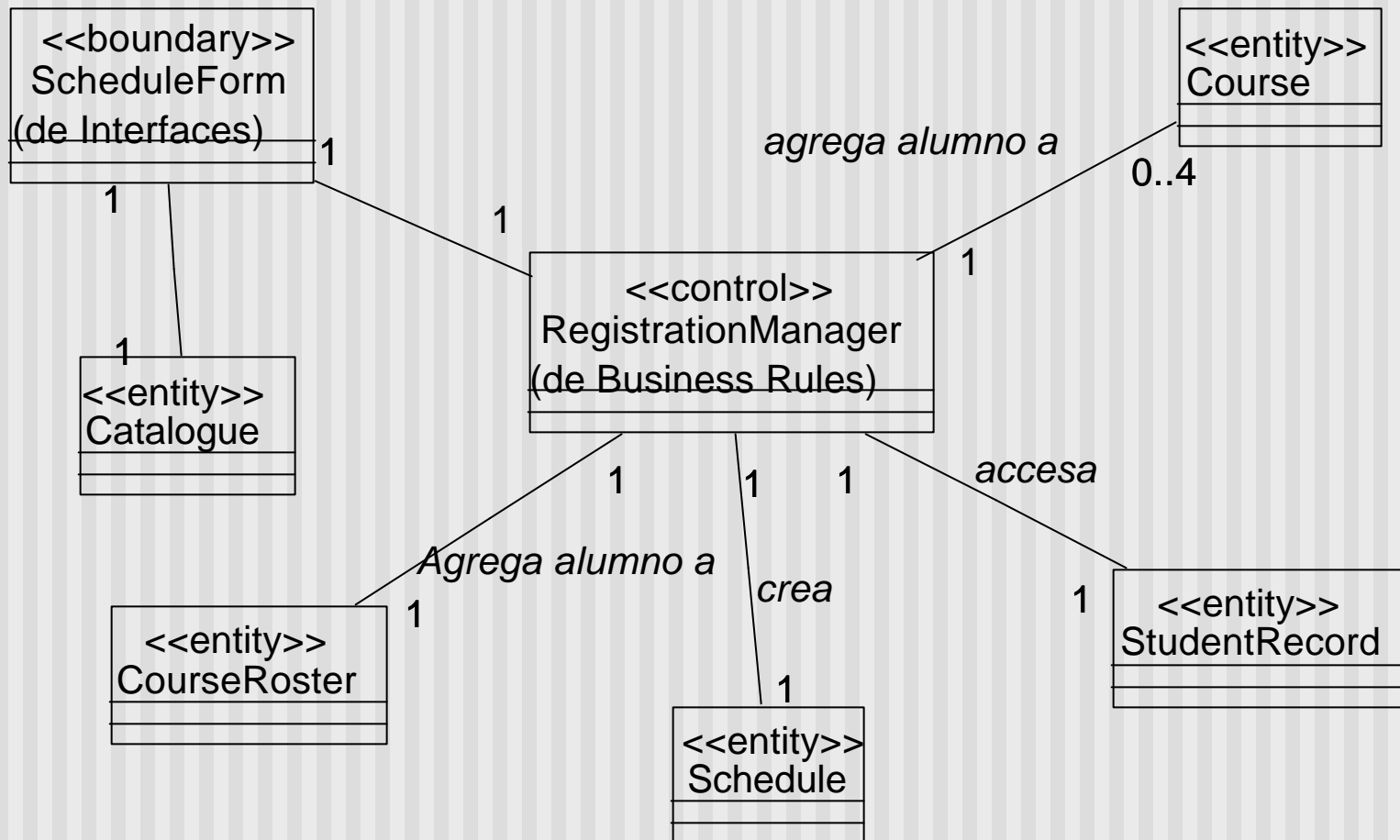
Actualización del Diagrama de Clases Principal para el Sistema de Inscripción



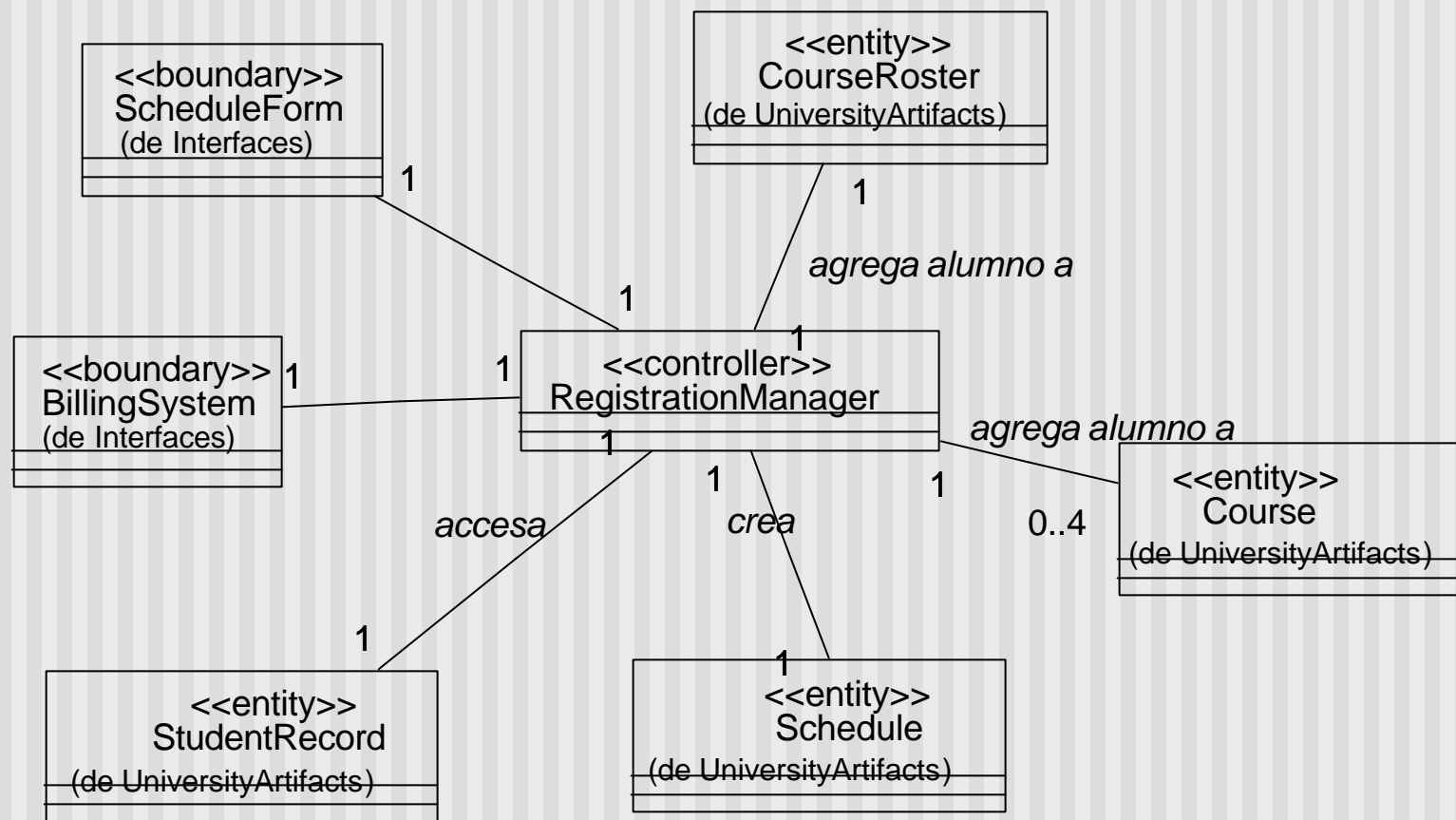
Actualización del Diagrama de Clases de Interfaces



Actualización del Diagrama de Clases de University Artifacts



Actualización del Diagrama de Clases de Business Rules



Ejercicio: Relaciones

- Usando los escenarios y diagramas de secuencias generados en lecciones anteriores
 - Actualizar los diagramas de clases mostrando relaciones entre clases
 - **Asegurarse de que se tomen las decisiones de multiplicidad inicial**
- Agregar relaciones a los paquetes para el sistema

Operaciones y Atributos



Objetivos: Operaciones y Atributos

- Usted podrá:
 - Definir operaciones para clases
 - Definir atributos para clases
 - Definir encapsulación y establecer sus beneficios
 - Representar atributos y operaciones en diagramas de clase

¿Qué es una operación?

- Una clase engloba un conjunto de responsabilidades que definen el comportamiento de los objetos de esa clase
- Las responsabilidades de una clase se llevan a cabo por sus operaciones
 - Esto no es necesariamente un mapeo de uno-a-uno
 - Responsabilidad de la clase Producto -- precio de venta
 - Operaciones para esta responsabilidad
 - **Buscar información de una base de datos**
 - **Calcular el precio**
- Una operación es un servicio que puede ser solicitado desde un objeto al comportamiento de efecto

Una operación debe desempeñar una función simple y cohesiva

Las operaciones dependen del dominio

- Listar las operaciones relevantes al dominio del problema
 - Las operaciones de la clase Persona serán diferentes dependiendo de “quién esté preguntando”

Perspectiva del Banquero

recibir renta
manejar cuenta
recibir líneaDeCrédito

Perspectiva del Doctor

examinar
tomarMedicina
irAlHospital
recibirFactura

Nombrando Operaciones

- Las operaciones deben nombrarse para indicar su resultado, no los pasos detrás de la operación.
- Ejemplos:
 - `calculateBalance()`
 - Pobrementemente nombrado
 - **Indica que se debe calcular el balance -- esta es una decisión de implementación/optimización**
 - `getBalance()`
 - Bien nombrado
 - **Indica solamente el resultado**

Nombrando Operaciones (cont.)

- Las operaciones deben nombrarse desde la perspectiva del proveedor no del cliente
- En una gasolinera, la gasolina se recibe de la bomba
 - La bomba tiene su responsabilidad a través de una operación -- ¿cómo se le debe llamar?
 - Nombres adecuados -- `dispense()`, `giveGas()`
 - Nombre malo -- `receiveGas()`
 - **La bomba da la gasolina -- no recibe la gasolina**

¿Qué es una operación primitiva?

- Una operación primitiva es una operación que no puede ser implementada usando solamente las operaciones internas de la clase
 - Todas las operaciones de una clase son típicamente primitivas
- Ejemplo:
 - Agregar un objeto a un conjunto -- operación primitiva
 - Agregar cuatro objetos a un conjunto -- no primitiva
 - Se puede implementar con llamadas múltiples a la operación de agregar un objeto a un conjunto

Firma de una Operación

- La firma de una operación consiste en:
 - Lista de argumentos opcional
 - Clases o valores de retorno
- Durante el análisis NO ES OBLIGATORIO llenar la firma de una operación
 - Esta información debe completarse en el diseño

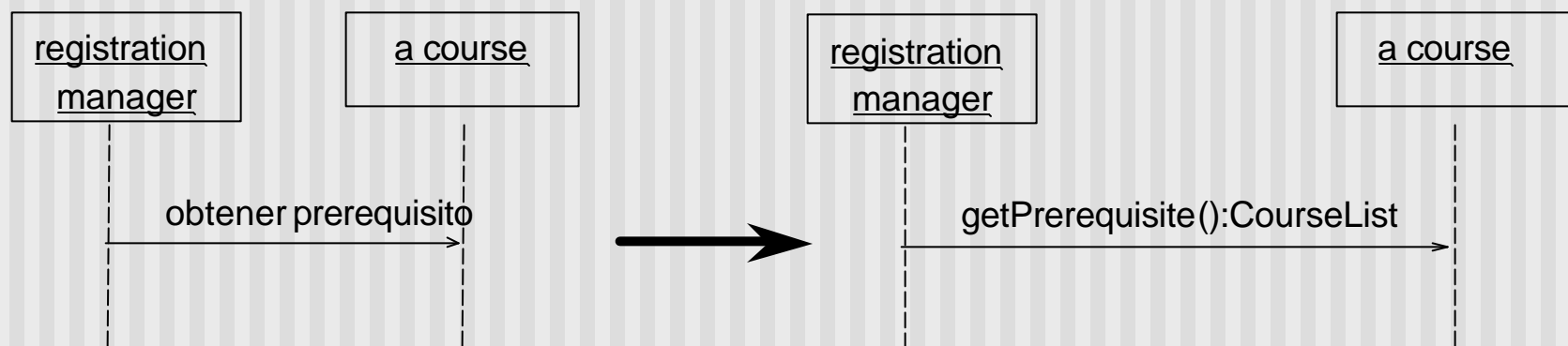
Despliegue de Operaciones

- Las operaciones se muestran en el tercer compartimiento de la clase

Course
getPrerequisite () : CourseList

Obteniendo Operaciones a partir de los Diagramas de Interacción

- Los mensajes desplegados en los diagramas de secuencias y/o colaboración son generalmente operaciones de la clase receptora
 - Los mensajes se traducen en operaciones y se agregan al diagrama de clase



Descubriendo Clases Adicionales

- Si se especifica una firma de operación, es posible descubrir clases adicionales
 - Argumento en la operación
 - Clase de retorno
- Ejemplo:
 - `getPrerequisite() : CourseList`
 - `addStudent(John : StudentInfo)`
- Las clases adicionales se agregan al modelo
 - Se despliegan en diagramas de clases cuando sea necesario

Descubriendo Relaciones Adicionales

- Los argumentos de una operación y/o la clase de retorno denotan una relación entre la clase que posee la operación y la clase del argumento y/o la clase de retorno
- Ejemplo:
 - La clase CourseRoster tiene una operación `addStudent(John: StudentInfo)`
 - Esto implica que hay una relación entre CourseRoster y StudentInfo
- Las relaciones adicionales se agregan al modelo
 - Se despliegan en diagramas de clases cuando sea necesario

¿Qué es un atributo?

- Un atributo es una definición de dato contenido en instancias de la clase
- Los atributos no tienen comportamiento -- no son objetos
- Los nombres de atributo son sustantivos simples
 - Los nombres deben ser únicos en la clase
- Cada atributo debe tener una definición clara y concisa
- Atributos buenos para la clase Alumno
 - Name -- nombre y apellido
 - Major -- campo superior de estudios
- Atributo malo para la clase Alumno -- selectedCourses
 - Esta es una relación no un atributo

Valores de Atributo

- El valor de atributo está dado por el estado de un objeto particular
- Cada objeto tiene un valor para cada atributo definido por su clase
- Por ejemplo, para un objeto de la clase Profesor:

Atributos
Nombre
Número de Identificación
Materia que imparte



Sue Smith
567892
Matemáticas



George Jones
578391
Biología

Los Atributos dependen del dominio

- Listar todos los atributos relevantes para el dominio del problema
 - Los atributos de una clase Persona serán diferentes dependiendo de “quién esté preguntando”

Perspectiva de Banquero

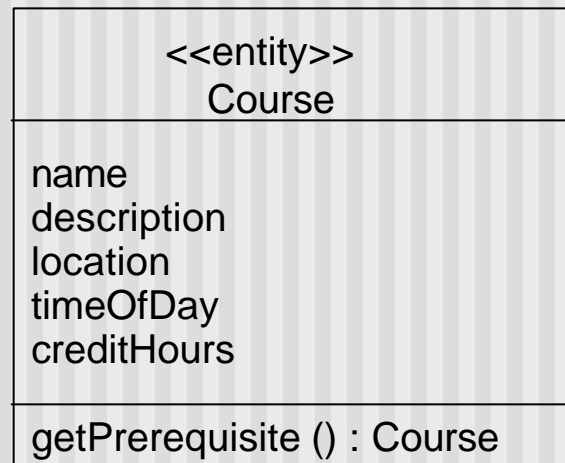
nombre
dirección
fechaDeNacimiento
NumeroCuenta

Perspectiva de Doctor

nombre
dirección
fechaDeNacimiento
altura
peso

Despliegue de Atributos

- Los atributos se muestran en el segundo compartimiento de la clase



Atributos Derivados

- Un atributo derivado es un atributo cuyo valor puede calcularse en base al valor de otro(s) atributo(s)
 - Se usa cuando no hay tiempo suficiente para re-calcular el valor cada vez que sea necesario
 - Tráfico del desempeño del tiempo de corrida vs. memoria requerida

Rectangle
length width /area

Tipo de Dato, Atributo y Valor Inicial

- Cada atributo tiene:
 - Tipo de Dato
 - Valor Inicial (Opcional)
- Durante el análisis NO ES OBLIGATORIO completar la definición del atributo
 - Esta información debe completarse en el diseño

¿Cómo se descubren los atributos?

- Se descubren atributos en el flujo de eventos de los casos de uso
 - Buscar sustantivos que no se consideraron buenos candidatos para clases
- Otros se descubren cuando la definición de la clase se crea
- Con la ayuda de expertos en el dominio, el cual nos puede proporcionar buenos atributos

Sólo modele los atributos que sean relevantes al dominio del problema

Ejemplo: Atributos en el Problema de Inscripción a Cursos

- “Cada curso tendrá una descripción”
 - Un atributo llamado descripción se agrega a la clase Curso

Course
description

Guía de Estilo para Nombrar Atributos y Operaciones

- Una guía de estilo debe dictar convenciones de nombres para atributos y operaciones
 - Proporciona consistencia a través del proyecto
 - Conduce a más modelos y código que se puede mantener
- Ejemplo
 - Los atributos y operaciones inician con una letra minúscula
 - No se usan subrayados
 - Los nombres compuestos de múltiples palabras se juntan y la primer letra de cada palabra adicional se escribe con mayúsculas

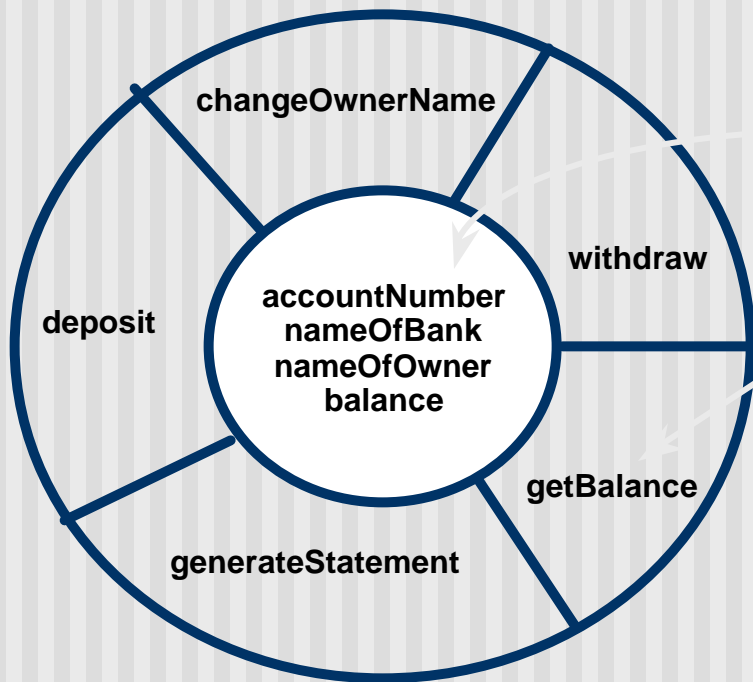
Despliegue de Atributos y Operaciones

- Los atributos y/u operaciones deben mostrarse en una clase
- Pueden crearse diagramas de clases adicionales para desplegar atributos y operaciones
 - Las relaciones típicamente no se despliegan en estos diagramas de clase

Encapsulado

- Un modo de ver una clase es la que consiste de dos partes: la interfaz y la implementación
 - La interfaz puede verse y usarse por otros objetos
 - La implementación es oculta para los clientes
- Ocultar detalles de implementación de un objeto se llama encapsulado u ocultamiento de información
- El encapsulado ofrece dos tipos de protección:
 - Protege el estado interno de un objeto al ser capturado por sus clientes
 - Protege el código del cliente de los cambios en la implementación del objeto

Ejemplo: Encapsulado



Sólo las operaciones proporcionadas por el objeto pueden cambiar los valores de un atributo

Las operaciones están provistas para desplegar valores de atributos que los clientes necesitan

Los clientes no pueden modificar el estado del objeto directamente

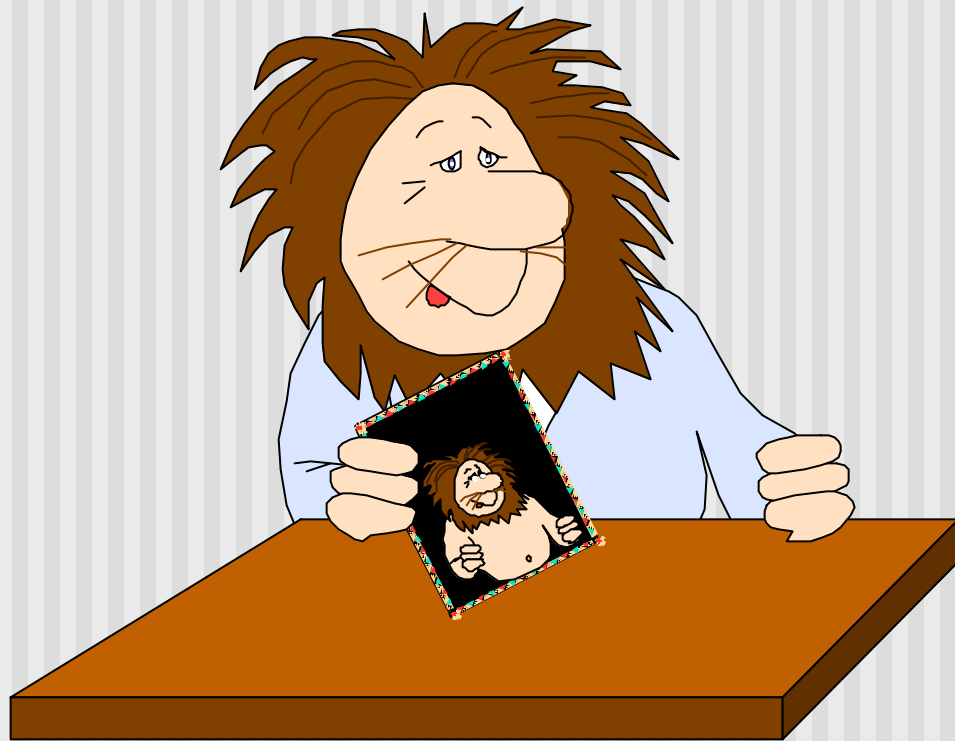
Beneficios del Encapsulado

- El código de la operación de un cliente puede utilizar la interfaz de otra clase
- El código del cliente no puede tomar ventaja de la implementación de una operación de otra clase
- La implementación puede cambiar por los siguientes motivos:
 - Corregir un defecto
 - Mejorar el desempeño
 - Reflejar un cambio de política
- El código del cliente no será afectado por los cambios en la implementación, de este modo se reduce el “efecto de rizo (ripple)” en el cual la corrección de una operación fuerza la corrección correspondiente en una operación del cliente
- El mantenimiento es más fácil y menos costoso

Ejercicio: Operaciones y Atributos

- Actualizar los diagramas de secuencias en lecciones previas y transformar los mensajes en nombres de operaciones concisas, tanto como sea necesario
- Crear diagramas de clases mostrando sólo atributos y operaciones
- Agregar relaciones adicionales basadas en argumentos de operaciones y/o clases de retorno

Herencia



Objetivos: Herencia

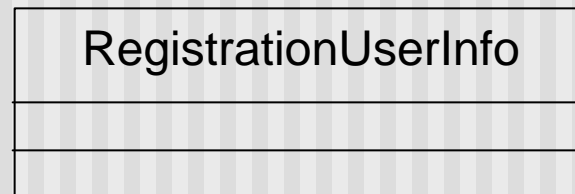
- Usted podrá:
 - Definir y discutir herencia, generalización y especialización
 - Representar jerarquías de herencia en diagramas de clases
 - Entender las técnicas para encontrar herencias
 - Definir herencia múltiple

Herencia

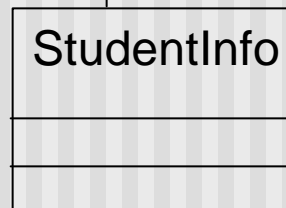
- La herencia define una relación entre clases donde una clase comparte la estructura y/o comportamiento de una o más clases
- La herencia define una jerarquía de abstracciones en las que una subclase hereda de una o más superclases
 - Con la herencia simple, la subclase hereda sólo de una superclase
 - Con la herencia múltiple, la subclase hereda de más de una superclase
- La herencia es una relación del tipo: “es una” o “tipo de”

Dibujo de una Jerarquía de Herencia

Superclase



Subclase



Relación de Herencia



Consideraciones de Herencia

- Debido a que una relación de herencia no relaciona objetos individuales
 - La relación no se nombra
 - La multiplicidad no tiene sentido
- Teóricamente, no hay límite en el número de niveles en una herencia
 - En la práctica, los niveles están limitados
 - Las jerarquías típicas de C++ son de 3 o 5 niveles
 - Las jerarquías de Smalltalk pueden ser un poco más profundas

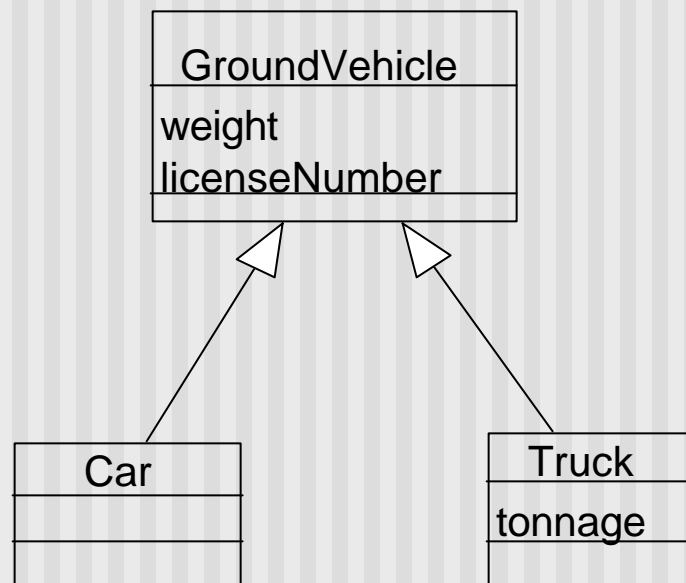
¿Qué es lo que tiene herencia?

- Una subclase hereda de sus padres:
 - Atributos
 - Operaciones
 - Relaciones
- Una subclase puede:
 - Agregar atributos, operaciones y relaciones
 - Redefine operaciones heredadas (¡sea cuidadoso!)

La herencia controla las similitudes entre clases

Atributos de Herencia

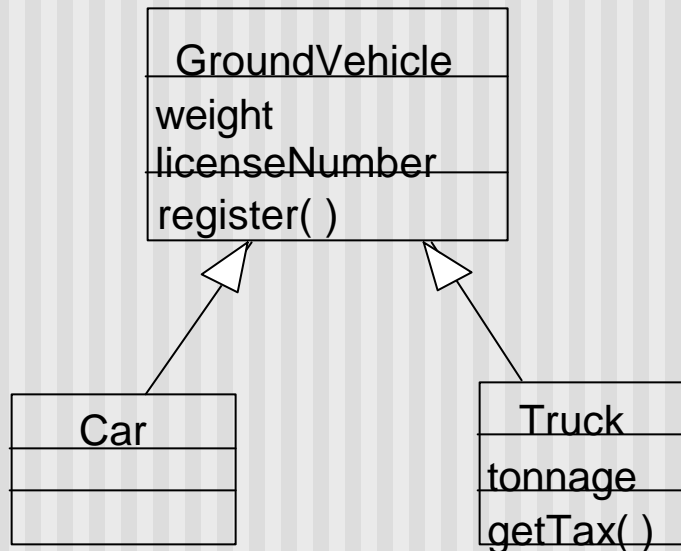
- Los atributos se definen en el más alto nivel de la jerarquía de herencia
- Las subclases de una superclase heredan todos sus atributos
- Cada subclase puede agregar atributos adicionales



Truck tiene tres atributos:
licenseNumber
weight
tonnage

Operaciones de Herencia

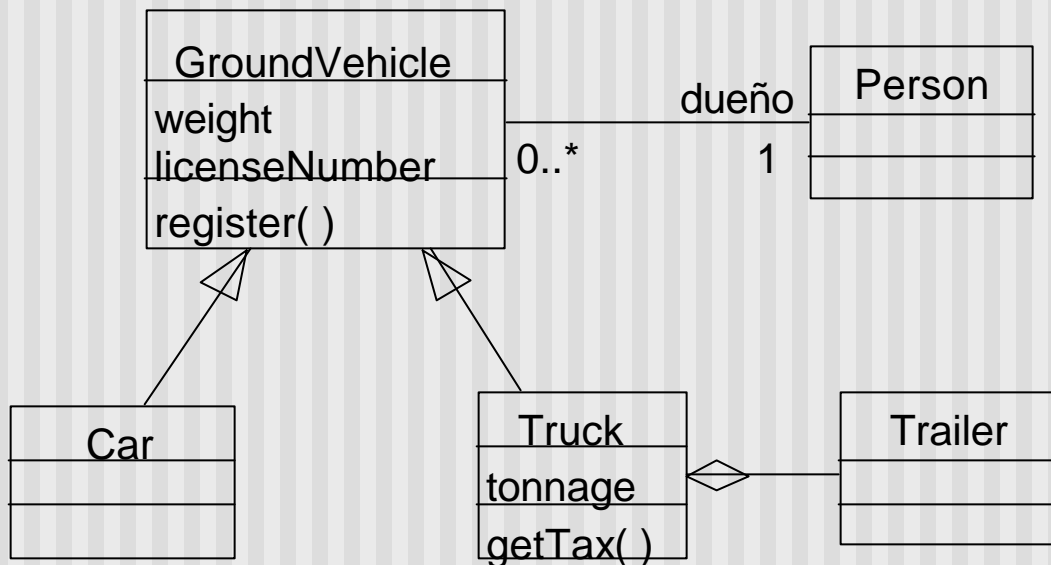
- Las operaciones se definen en el más alto nivel de la jerarquía de herencia
- Las subclases de una superclase heredan todas las operaciones
- Cada subclase puede aumentar o redefinir operaciones heredadas



Truck tiene tres atributos:
licenseNumber
weight
tonnage
y dos operaciones:
register()
getTax()

Relaciones de Herencia

- Las relaciones también se heredan y deben definirse en el más alto nivel en la jerarquía de herencia
- Las subclases de una superclase heredan todas sus relaciones
- Cada subclase puede participar en relaciones adicionales

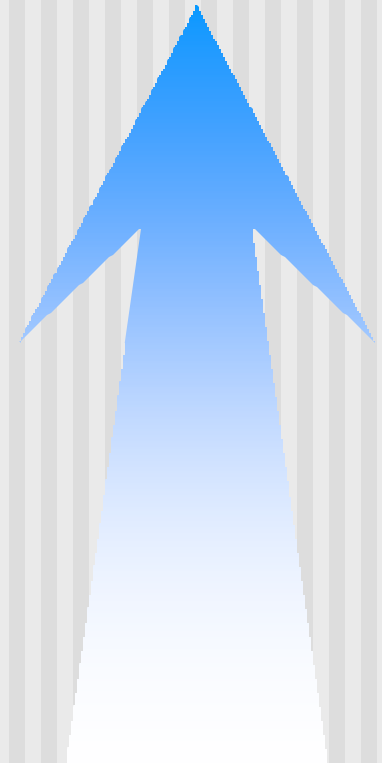
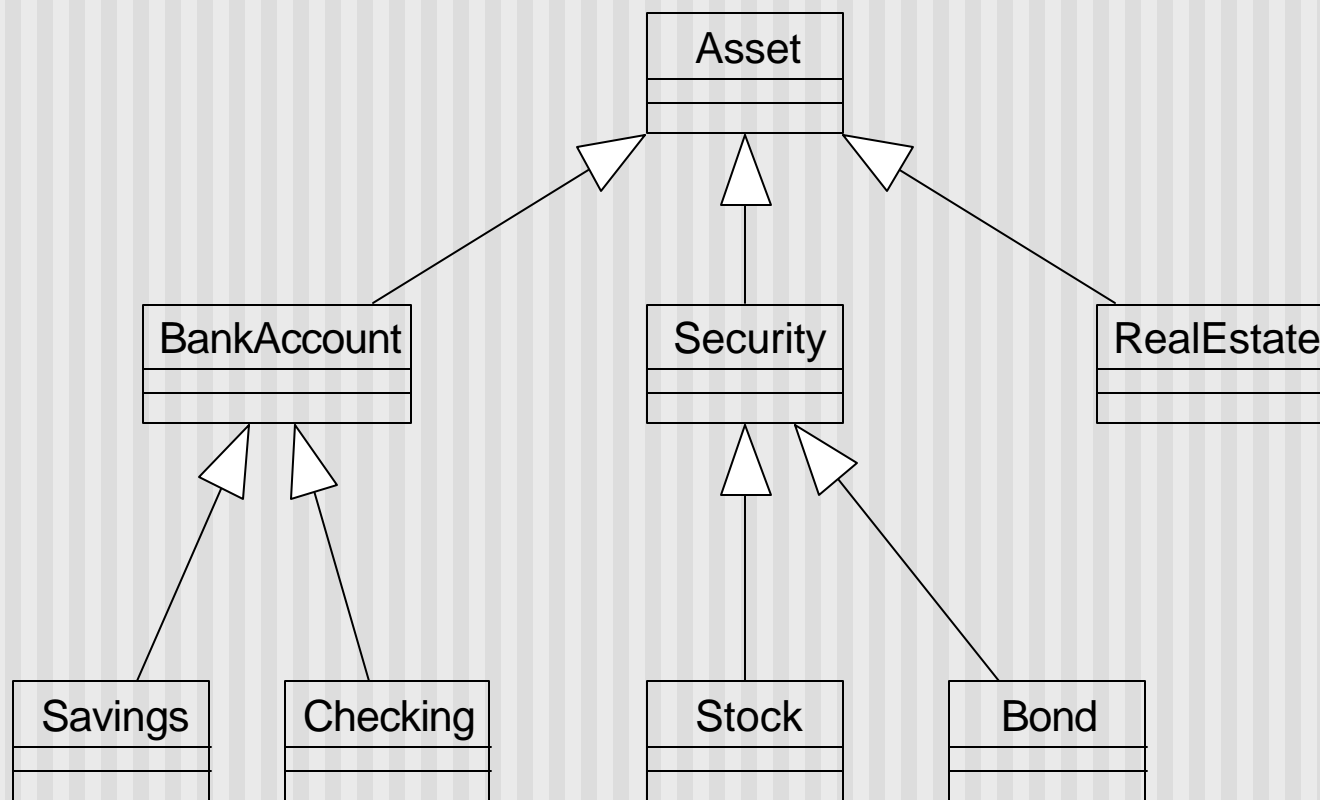


Car está relacionado con un dueño
Truck está relacionado con un dueño
Truck también tiene un Trailer

Generalización de Clases

- La generalización proporciona la capacidad de crear superclases que encapsulan la estructura y/o el comportamiento comunes a varias subclases
- Procedimiento de generalización
 - Identificar similitudes de estructura/comportamiento entre varias clases
 - Crear una superclase que encapsule el comportamiento/estructura comunes
 - Las clases originales se hacen subclases de la superclase nueva
- Las superclases son más abstractas que sus subclases

Ejemplo de Generalización

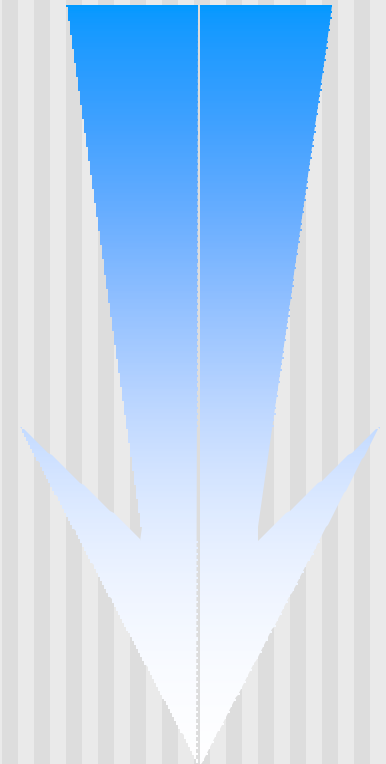
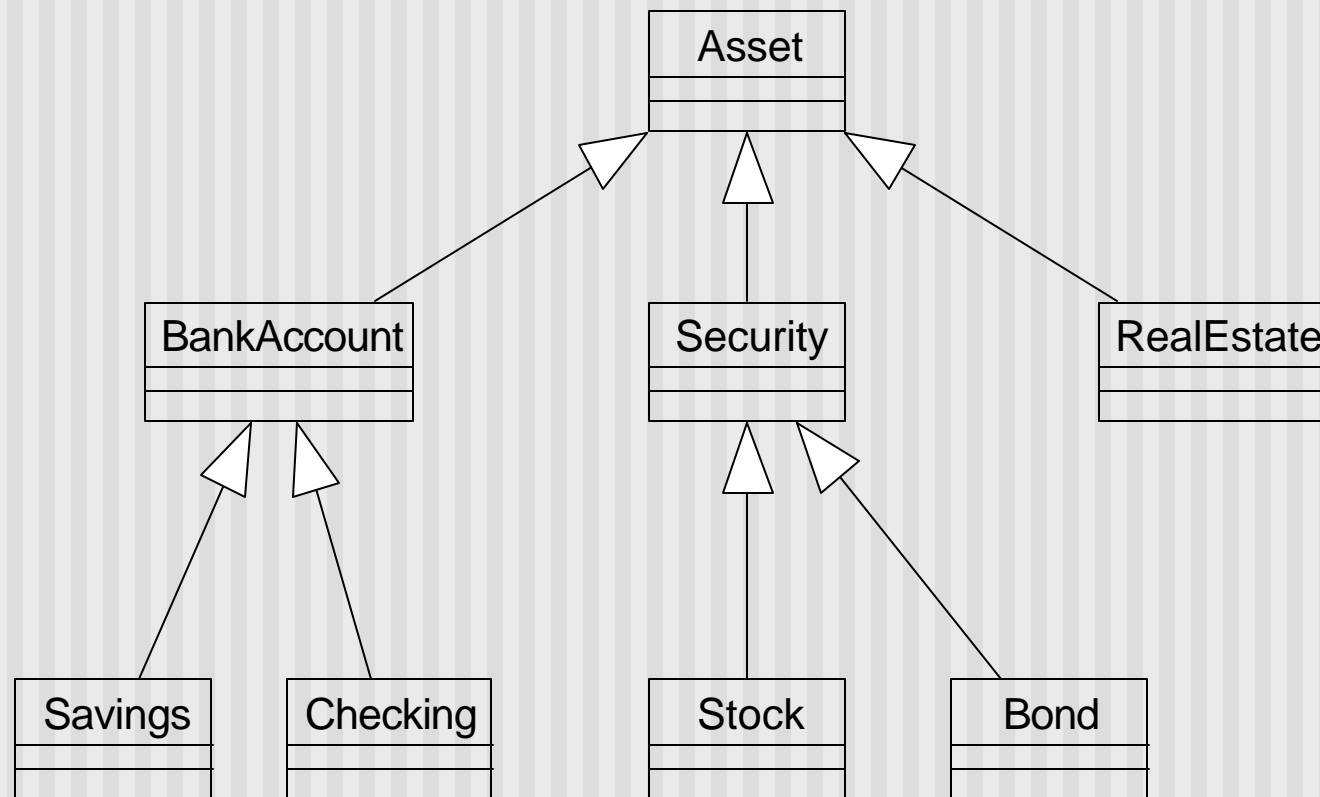


**Incremento de
abstracción**

Especialización de Clases

- La especialización proporciona la capacidad de crear subclases que representen refinamientos en los que la estructura y/o comportamiento de la superclase se agregan o modifican
- Procedimiento de Especialización
 - Advertir que algunas instancias exhiben estructura o comportamiento especializado
 - Creación de subclases para agrupar instancias de acuerdo a su especialización
- Las subclases son menos abstractas que sus superclases

Ejemplo de Especialización

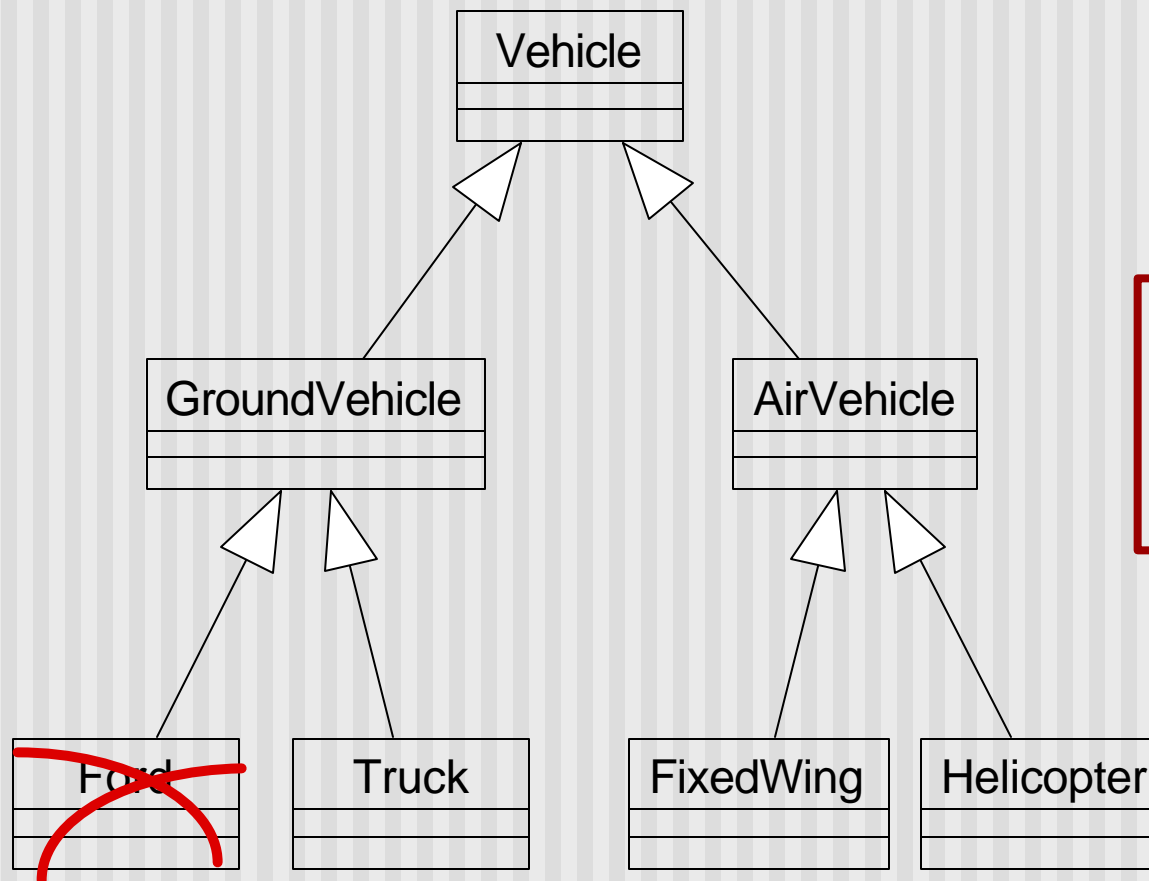


**Decremento
de
abstracción**

Jerarquías de Herencia

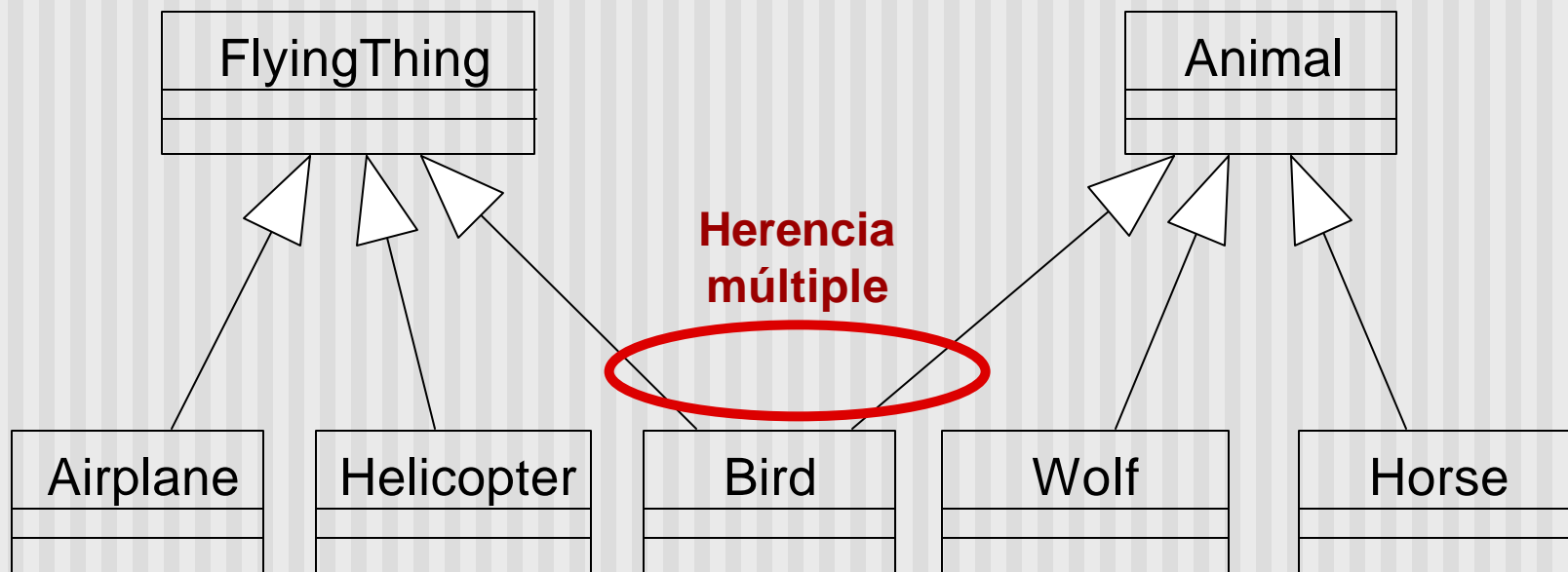
- Ambas técnicas, generalización y especialización, se usan en el desarrollo de jerarquías de herencia
- Durante el análisis, se establecen jerarquías de herencia entre abstracciones, de ser necesario
- Durante el diseño, las jerarquías de herencia se definen para:
 - Incrementar la reutilización
 - Incorporar clases de implementación
 - Incorporar librerías de clase disponibles

Niveles de Abstracción



Clases al mismo al mismo nivel de herencia deben estar al mismo nivel de abstracción

Herencia Múltiple



Bird hereda de ambos FlyingThing y Animal

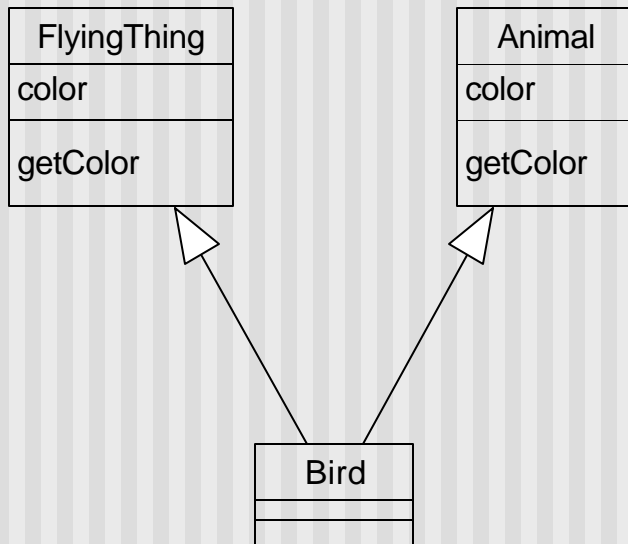
Conceptos de Herencia Múltiple

- Conceptualmente directo y necesario para modelar el mundo real correctamente
- En la práctica, puede conducir a dificultades en la implementación
 - No todos los lenguajes orientados a objetos soportan herencia múltiple directamente

**¡Use herencia múltiple sólo cuando sea necesario,
y
siempre con precaución !**

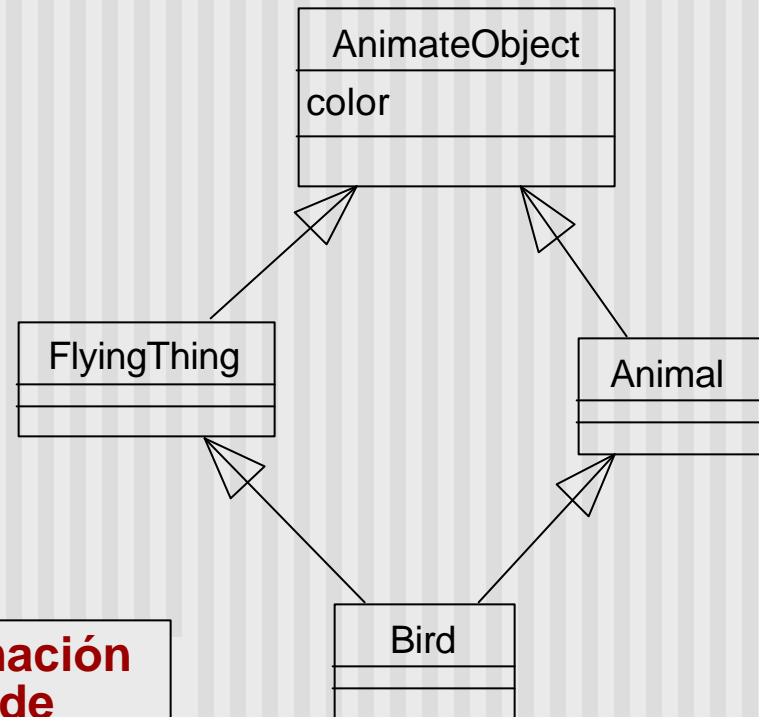
Problemas con Herencia Múltiple

Confusión con atributos y operaciones



Cada lenguaje/ambiente de programación elige modos para resolver este tipo de dificultades

Herencia repetida



Identificación de Herencia

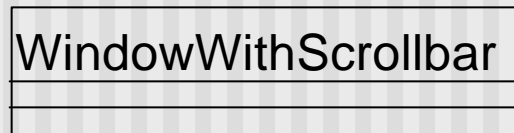
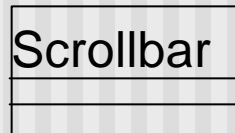
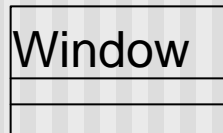
- Es importante evaluar todas las clases para encontrar posibles relaciones de herencia
 - Busque comportamiento común (operaciones) y estado (atributos) en las clases
- Técnica de Adición
 - Agregar operaciones/atributos nuevos a la(s) subclase(s)
- Técnica de Modificación
 - Redefinir operaciones
 - Debe tener precaución de no cambiar las semánticas

Herencia vs. Agregación

- Con frecuencia se confunde a la herencia y a la agregación
 - La herencia representa una relación “es un” o “tipo de”
 - La agregación representa una relación “tiene un”

Las palabras claves “es un” y “tiene un” ayudarán a determinar la relación correcta

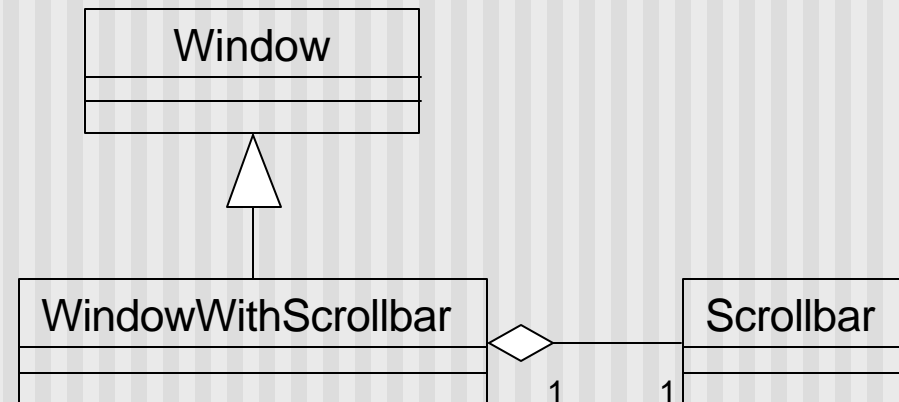
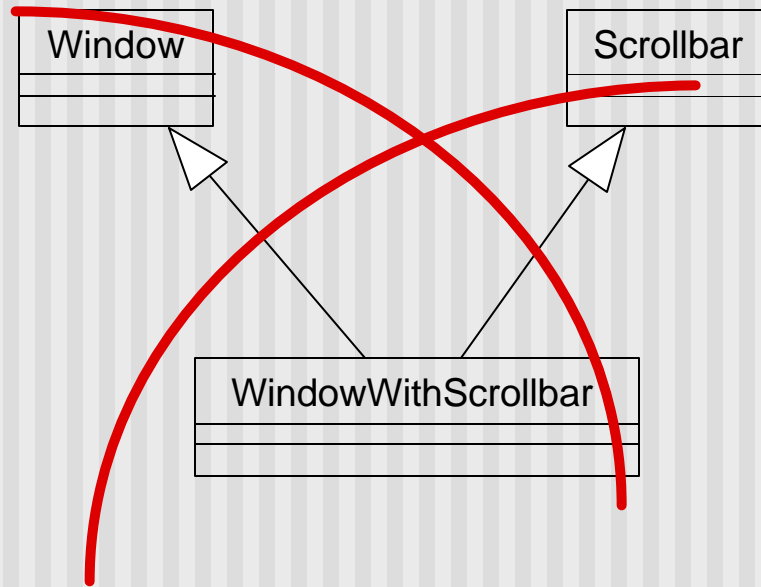
Window y Scrollbar



Un WindowWithScrollbar “es un” Window
Un WindowWithScrollbar “tiene un” Scrollbar

¿Qué relaciones deben usarse?

Window y Scrollbar (cont.)



Un WindowWithScrollbar “es un” Window
Un WindowWithScrollbar “tiene un” Scrollbar

Herencia vs. Agregación

Herencia	Agregación
Palabras clave “es un”	Palabras clave “tiene un”
Un objeto	Relaciona objetos en clases diferentes
Se representa con una flecha	Se representa con un diamante

¿Qué es Metamorfosis?

■ Metamorfosis

1. Un cambio en forma, estructura o función; especialmente el cambio físico que sufren algunos animales, como el renacuajo a rana
 2. Cualquier cambio notorio, como en el carácter, apariencia o condición
- Webster's New World Dictionary
 - Simon & Schuster, Inc., 1979

La Metamorfosis existe en el mundo
¿Cómo debe modelarse?

Ejemplo de Metamorfosis

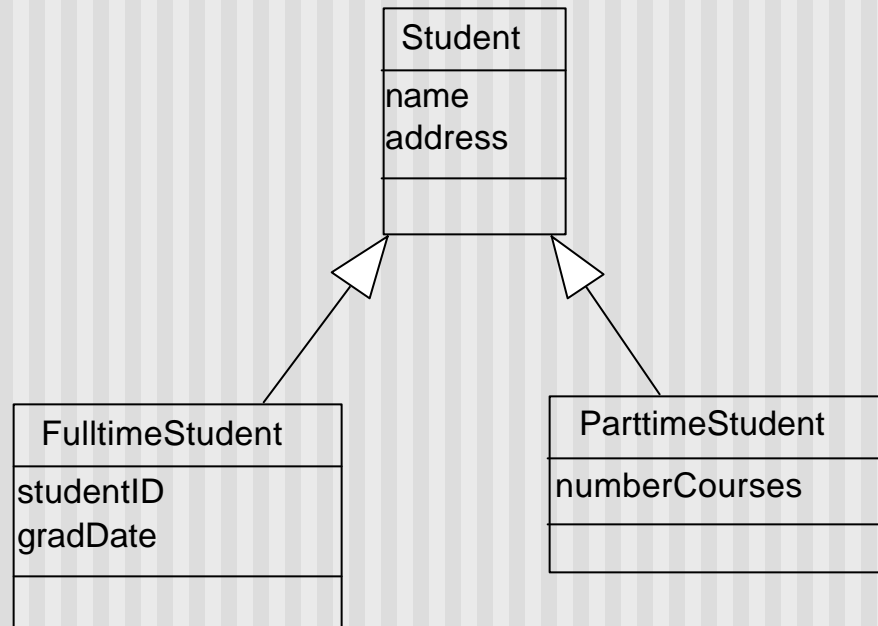
- En una universidad, hay alumnos de tiempo completo y de medio tiempo
 - Los alumnos de tiempo completo tienen un numero id y una fecha de graduación, pero los alumnos de medio tiempo no
 - Los alumnos de medio tiempo pueden tomar hasta de tres cursos. Los alumnos de tiempo completo no tienen ese límite

Part-timeStudent
name
address
numberCourses

Full-timeStudent
name
address
studentID
gradDate

Una Aproximación

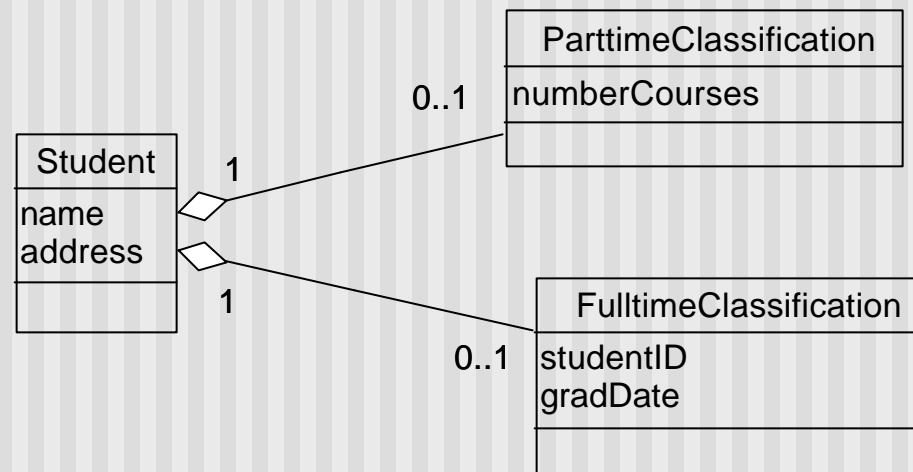
- Se puede crear una relación de herencia



¿Qué sucede si un alumno de medio tiempo desea convertirse en un alumno de tiempo completo?

Metamorfosis

- Es muy difícil cambiar la clase de un objeto
- Técnica de modelado mejorado
 - Poner la estructura y comportamiento que “cambia” en su propia clase



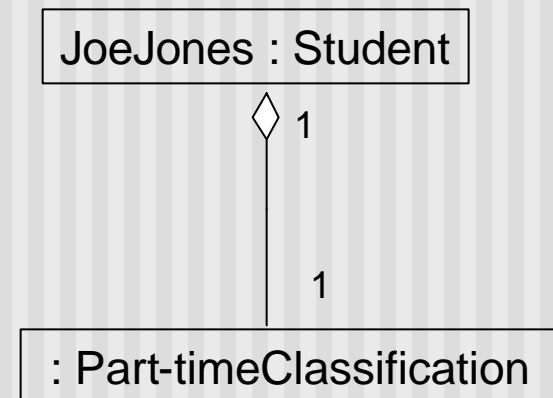
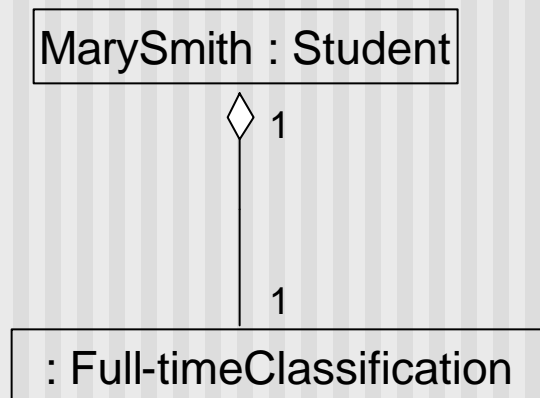
Metamorfosis (cont.)

Mary Smith

Estudiante de tiempo completo

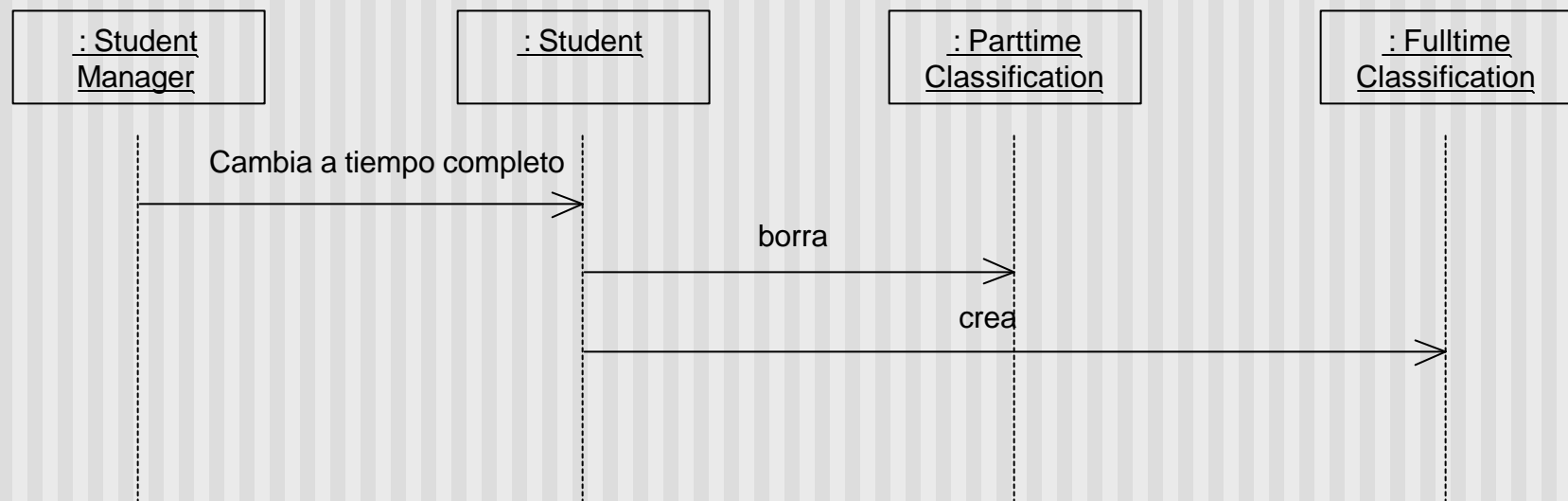
Joe Jones

Estudiante de medio tiempo



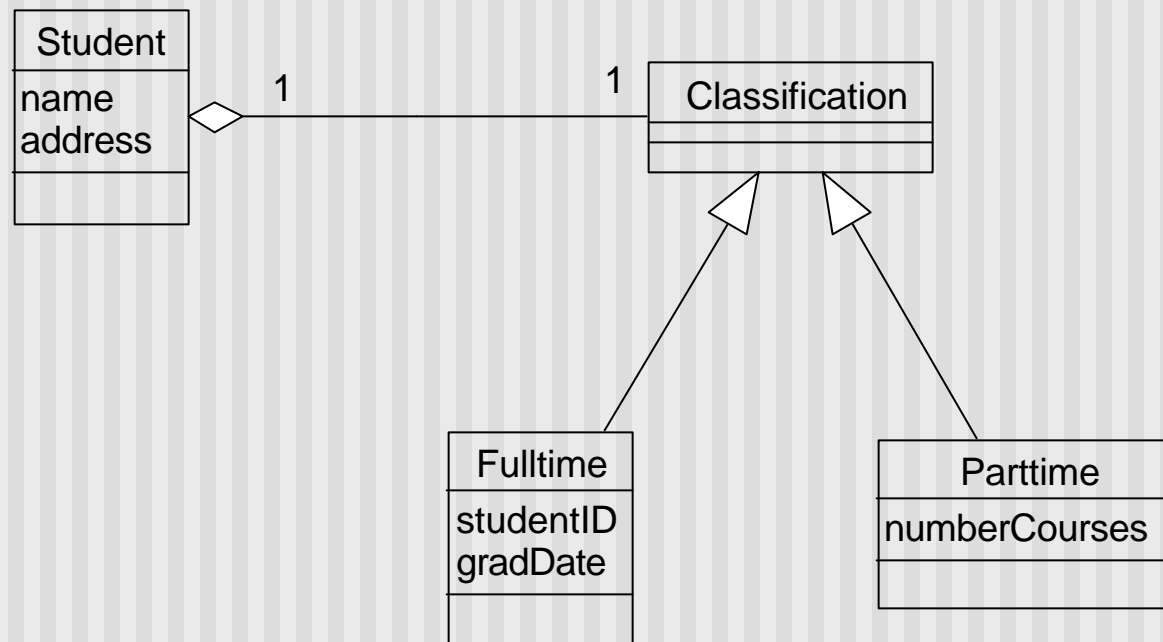
Metamorfosis (cont.)

- La metamorfosis se lleva a cabo por el objeto que “habla” con las partes cambiantes



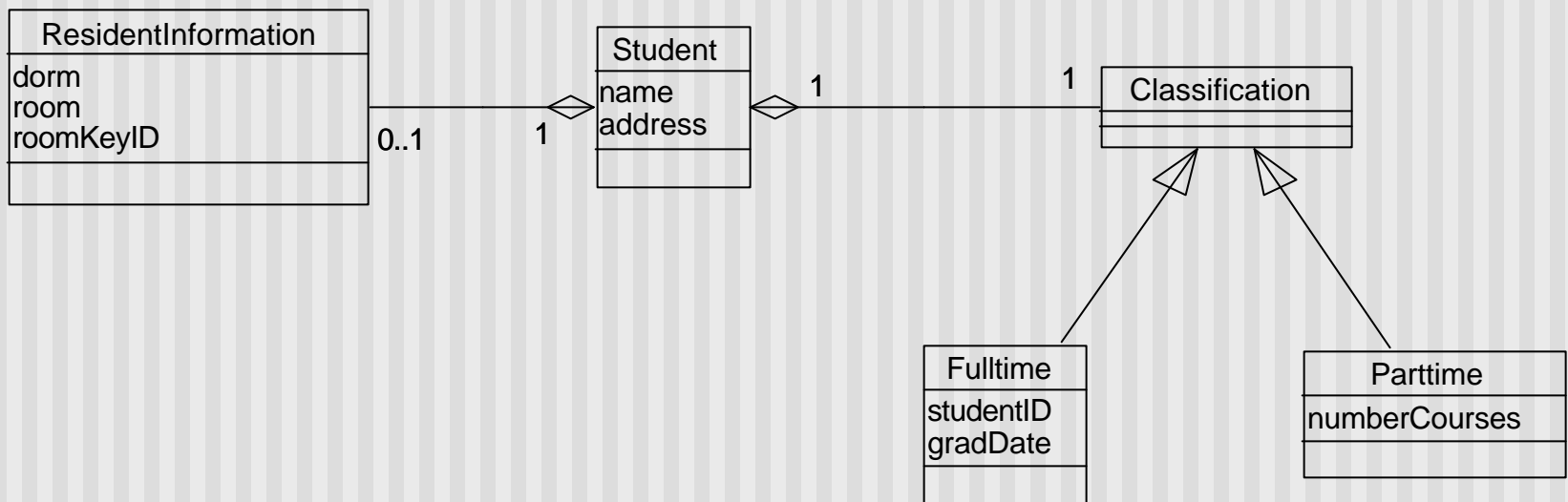
Metamorfosis y Herencia

- La herencia puede usarse para modelar estructura, comportamiento y/o relaciones comunes a las partes “cambiantes”



Metamorfosis y Flexibilidad

- Esta técnica también agrega flexibilidad al modelo
- Ejemplo: un alumno puede también vivir en el campus. En este caso, hay un identificador de dormitorio, un número de habitación y un número de llave de la habitación



Ejercicio: Herencia

- Examinar las clases definidas en el problema hasta el momento y agregar herencia donde sea necesario
 - Asegurarse de considerar cualquier metamorfosis
- Actualizar diagramas de clases como sea necesario

Comportamiento de Objetos



Objetivos: Comportamiento de Objetos

- Usted podrá:
 - Explicar la necesidad de los Diagramas de Transición de Estado
 - Entender cómo encontrar estados
 - Desarrollar un DTE simple que muestre
 - **Estados y Transiciones**
 - **Eventos**
 - **Condiciones de protección**
 - **Acciones y Actividades**
 - Entender el concepto de estados anidados
 - Explicar las relaciones entre diagramas de transición de estados, diagramas de objeto/interacción y diagramas de clase

¿Qué es un Diagrama de Transición de Estado?

- Un diagrama de transición de estado se usa para mostrar la historia de la vida de una clase dada, los eventos que causan una transición de un estado a otro, y las acciones que resultan de un cambio de estado
- El espacio de estados de una clase dada es la numeración de todos los estados posibles de un objeto
- El estado de un objeto es una de las condiciones posibles en las que puede existir un objeto
 - Contiene todas las propiedades del objeto
 - Generalmente estático
 - Más los valores actuales de cada una de estas propiedades
 - Generalmente dinámico

Dibujo de Estados

- Un estado se representa como un rectángulo con esquinas redondeadas en un diagrama de transición de estado



A diagram showing a single state represented by a rounded rectangle. The text "State Name" is centered inside the rectangle.

State Name

Estado y Atributos

- Los estados pueden distinguirse por los valores de ciertos atributos

Course
numStudents

<u>English101 : Course</u>
numStudents = 7

El número máximo de alumnos por curso es 10

numStudents < 10

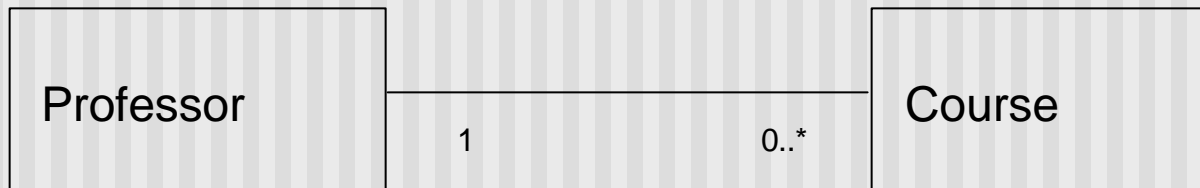
Open

numStudents >= 10

Closed

Estados y Ligas

- Los estados también pueden distinguirse por la existencia de ciertas ligas
- Las instancias de la clase Profesor puede tener dos estados:
 - Impartir cuando existe una liga a un curso
 - En sabático cuando no existe liga



Estados Especiales

- El estado inicial es el estado introducido cuando se crea un objeto
 - Un estado inicial es obligatorio
 - Sólo un estado inicial es permitido
 - El estado inicial se representa como un círculo sólido
- Un estado final indica el final de vida de un objeto
 - Un estado final es opcional
 - Puede existir más de un estado final
 - Un estado final se representa con un “ojo de buey”

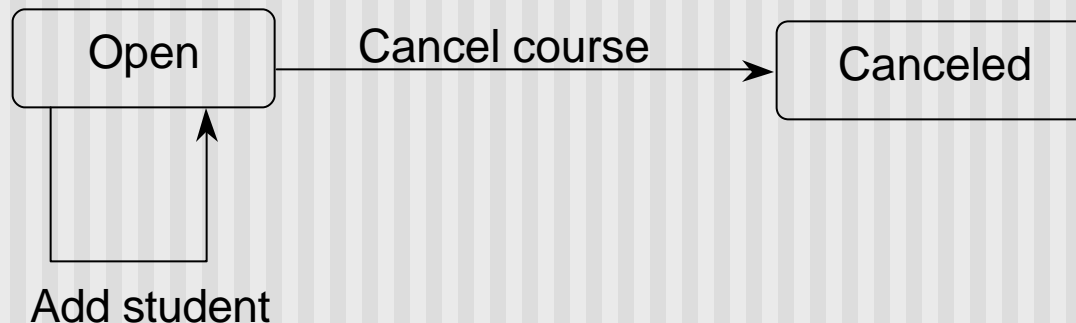


Eventos

- Un eventos es una ocurrencia que sucede en algún punto en el tiempo
 - El estado del objeto determina la respuesta a diferentes eventos
- Ejemplo:
 - Agregación un alumno a un curso
 - Creación de un curso nuevo

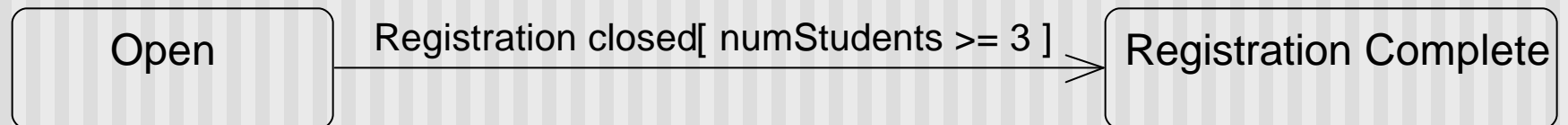
Transiciones

- Una transición es un cambio de un estado original a un estado sucesor como resultado de algunos estímulos
 - El estado sucesor también podría ser el estado original
- Una transición puede tomar lugar en respuesta a un evento
- Las transiciones pueden clasificarse con los eventos



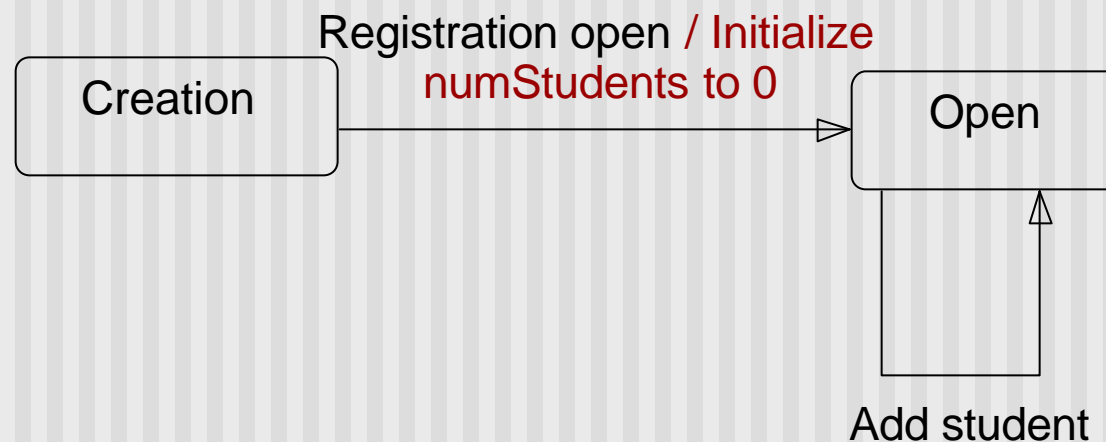
Condiciones de Seguridad

- Una condición de seguridad es una expresión booleana de valores de atributos que permiten una transición sólo si la condición es verdadera



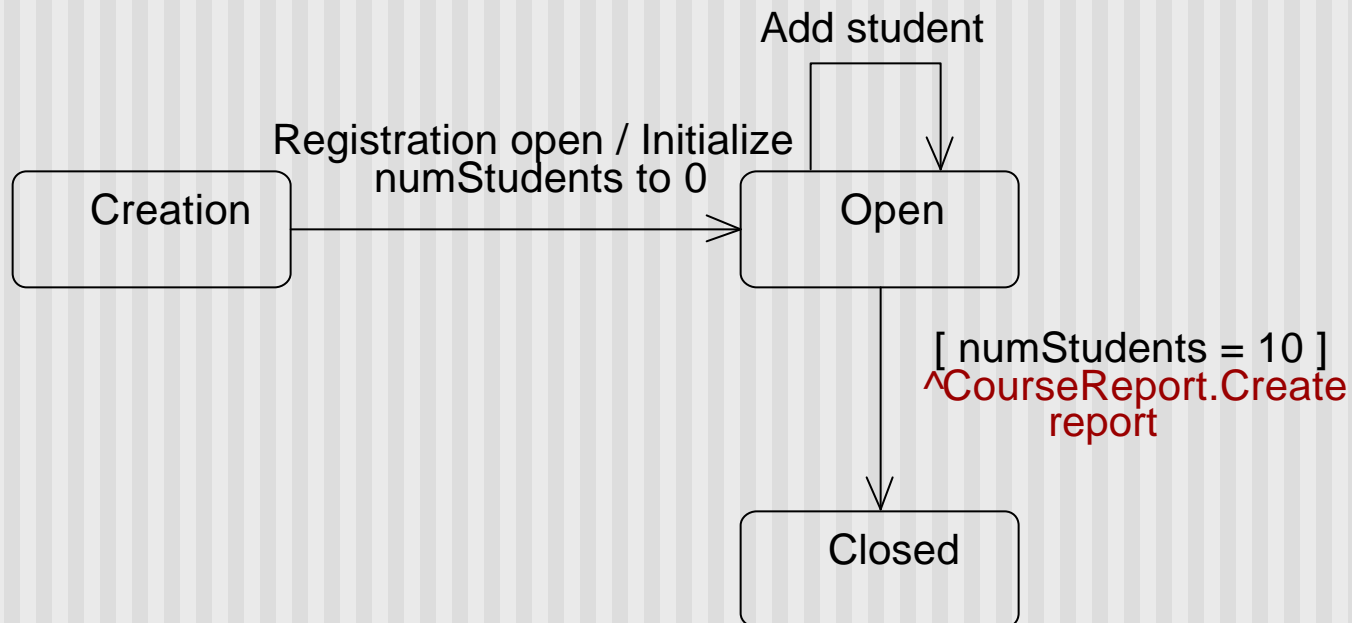
Acciones

- Una acción es una operación que se asocia a una transición
 - Toma una cantidad insignificante de tiempo para completarse
 - Se considera no-interrumpible
- Los nombres de acción se muestran en la flecha de transición precedida por una diagonal



Envío de Eventos

- Un evento puede disparar el envío a otro evento
 - Se muestra como: \wedge Class.event



Actividades

- Una actividad es una operación que toma tiempo para completarse
- Las actividades se asocian con un estado
- Una actividad
 - Inicia cuando se introduce el estado
 - Puede ejecutarse hasta el fin o puede ser interrumpida por una transición que sale



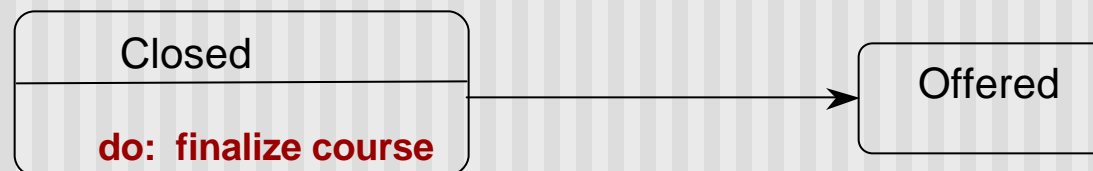
Envío de Eventos en un Estado

- Una actividad también puede enviar un evento a otro objeto

Dropping
do: ^CourseRoster.Drop student(Student)

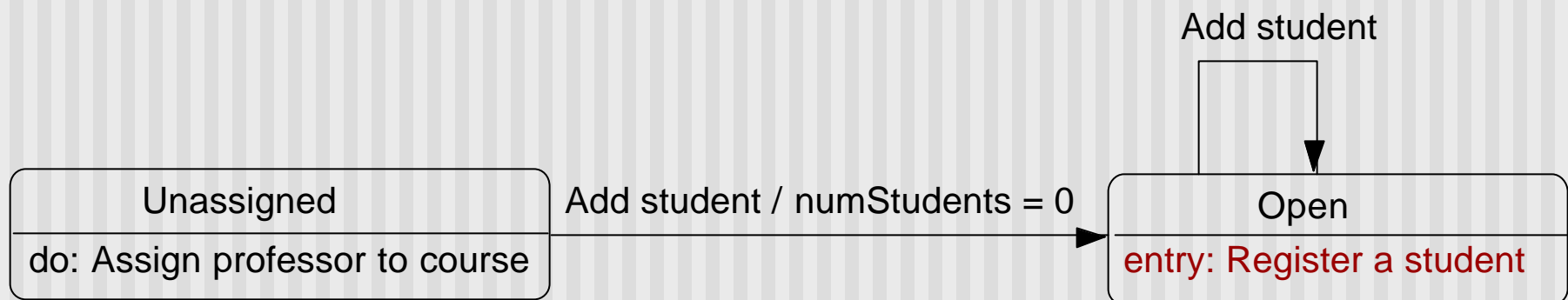
Transiciones Automáticas

- A veces, el único propósito de un estado es ejecutar una actividad
- Una transición automática ocurre cuando se completa la actividad
- Si hay múltiples transiciones automáticas
 - Se necesita una condición de seguridad en cada transición
 - Las condiciones deben ser mutuamente excluyentes



Acciones de Entrada y Salida

- Cuando una acción debe ocurrir sin importar como entra o sale del estado, se debe asociar la acción con el estado
 - En realidad, la acción se asocia con cada transición que entre o salga del estado
- La acción se muestra dentro del icono del estado precedida de la palabra **entry** o **exit**



Estado Anidado

- Los diagramas de transición de estado pueden volverse complejos e inmanejables
- Los estado anidados pueden usarse para simplificar diagramas complejos
- Un **superestado** es un estado que incluye estados anidados llamados **subestados**
- Las transiciones comunes de los subestados se representan en el nivel del superestado
- Es permitido cualquier número de niveles de anidación
- Los estados anidados pueden conducir a una reducción sustancial de complejidad gráfica, permitiéndonos modelar problemas más grandes y complejos

Diagrama de Transición de Estado para la Clase Curso sin Estados Anidados

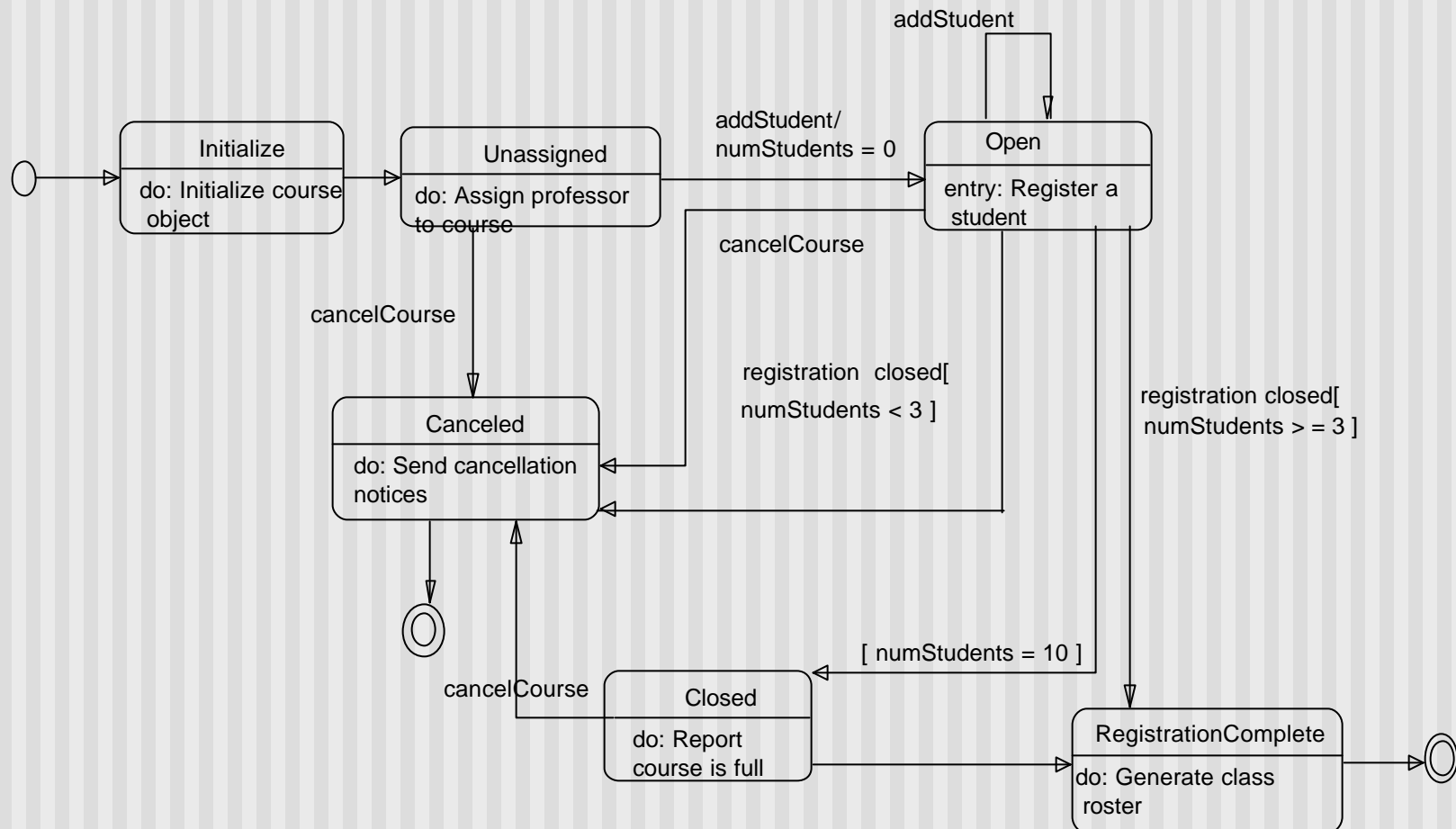
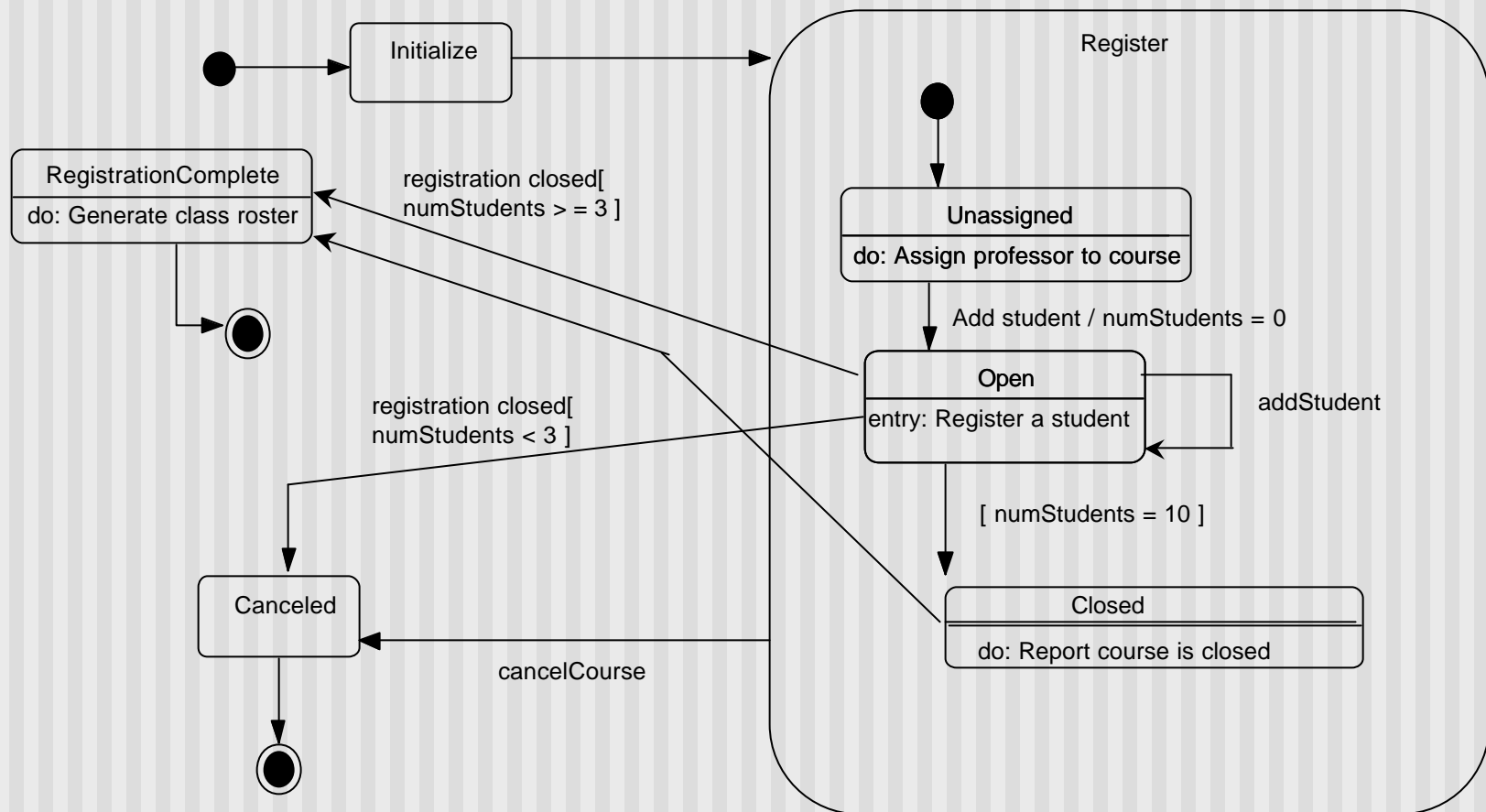


Diagrama de Transición de Estado para la Clase Curso con Estados Anidados



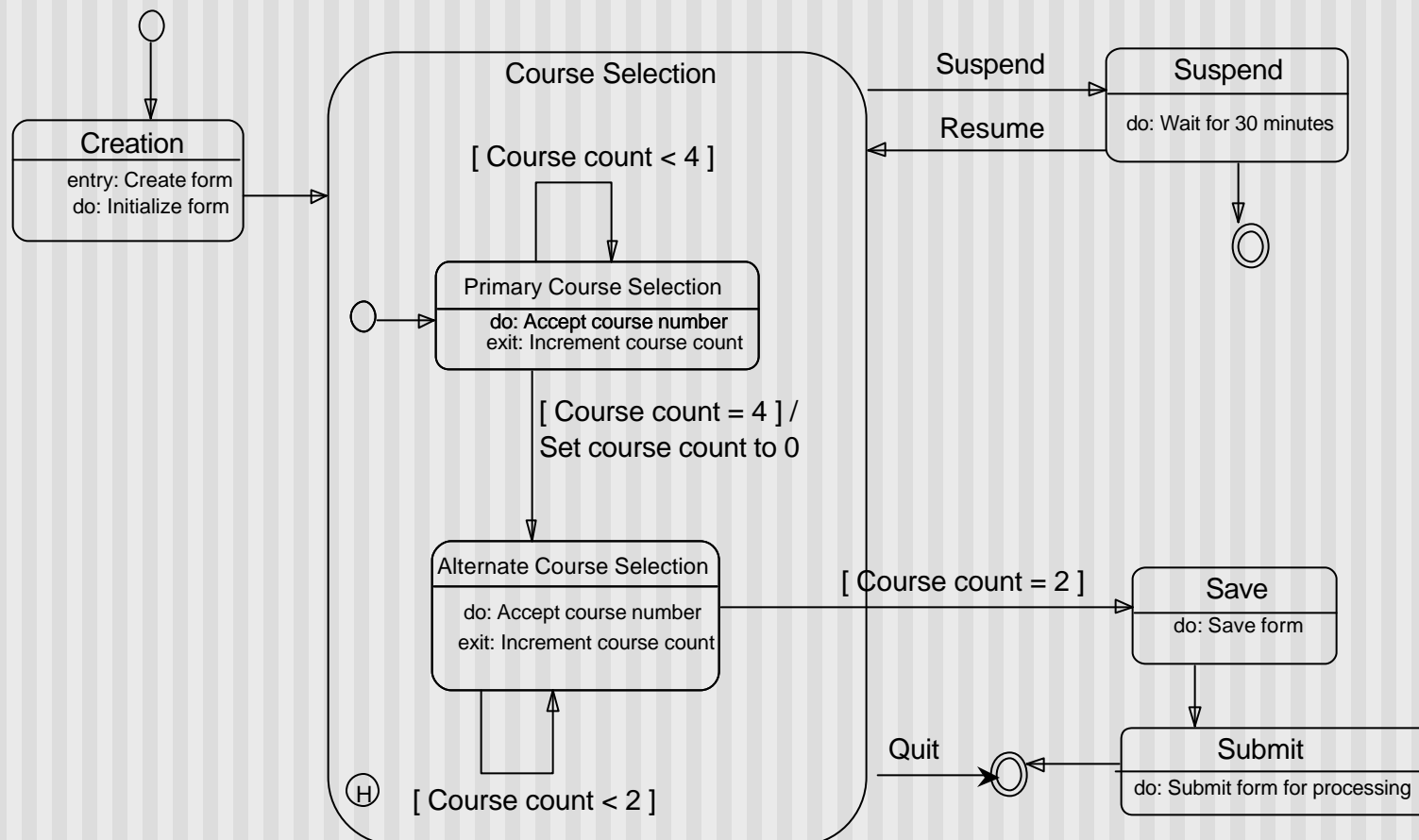
Estados Anidados con Memoria

- El uso de la característica de memoria indica que tras el regreso a un superestado, se introducirá el subestado más recientemente visitado
- Use la letra H en un círculo para denotar que la característica de memoria (history) se aplica al superestado
- Si no se usa la característica de memoria, siempre se tomará el subestado inicial cuando se introduzca el superestado

Ejemplo: Estado Anidados con Memoria

- En el sistema de Inscripción a Cursos, la clase CourseSelection hace lo siguiente
 - Acepta cursos primarios
 - Acepta cursos alternos
- El usuario puede salir en cualquier momento
- El usuario puede suspender una sesión por un máximo de 30 minutos mientras selecciona sus cursos
- La forma se guarda después de que se han seleccionado todos los cursos
- La forma se envía para procesarla después de que ha sido guardada

Estado Anidado con Memoria (Clase RegistrationForm)



Donde iniciar...

- Durante el análisis, primero hay que centrarse en las clases con comportamiento dinámico significativo
- Para una clase dada, busque estados posibles al:
 - Evaluar valores de atributos
 - Evaluar operaciones
 - Definir las reglas para cada estado, e
 - Identificar las transacciones validas entre estados
- Rastrear los mensajes en los diagramas de interacciones correspondientes a un objeto que tengan que ver con el modelado de la clase
 - El intervalo entre dos operaciones puede ser un estado

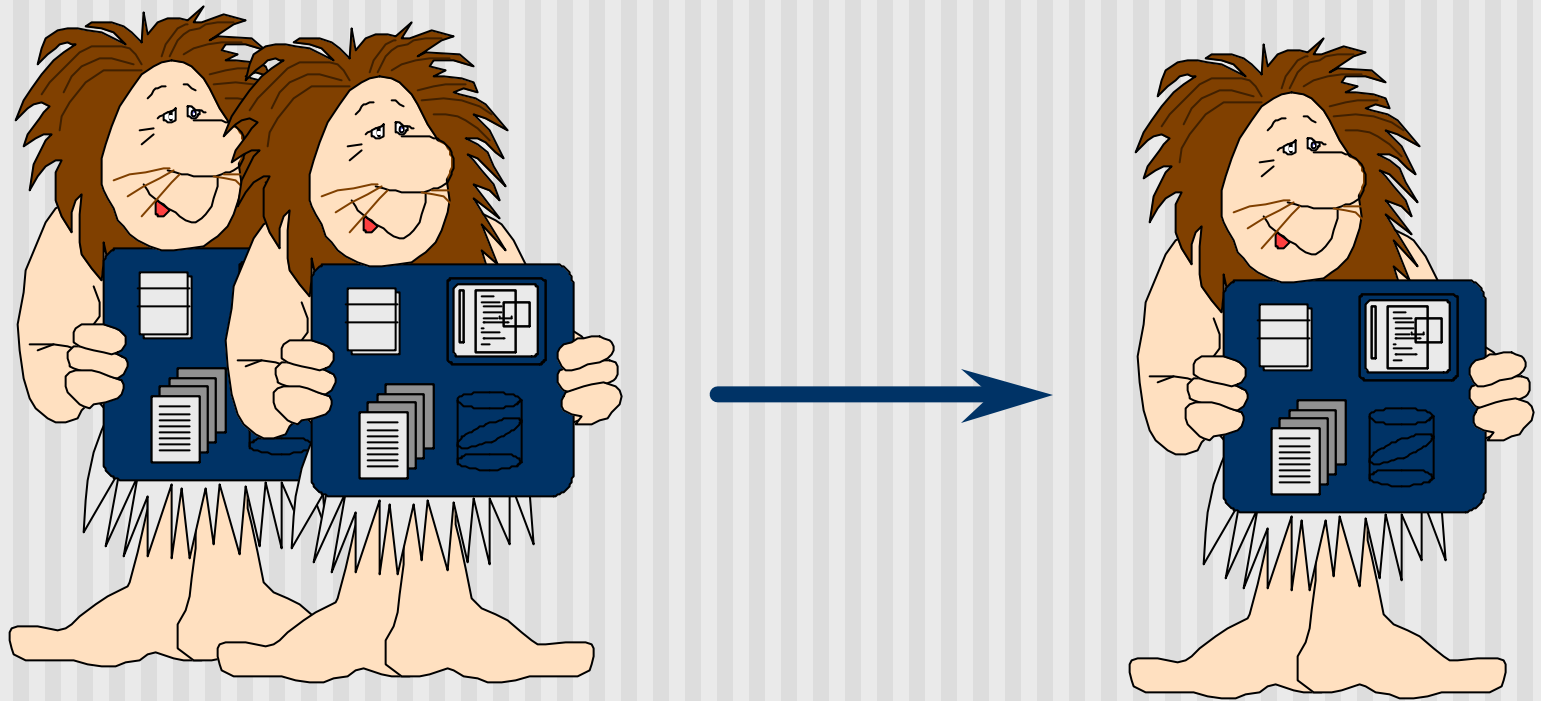
Ejercicio: Comportamiento de Objetos

- Crear un diagrama de transición de estados para modelar el comportamiento dinámico de la clase Empleado con los siguientes estados
 - Apply, Employed, Leave of Absence, Terminated, Retired
- Información del estado Apply
 - Se conduce una entrevista durante este estado
 - Se sale de este estado con el evento empleo
- Información del estado Employed
 - El estado Employed tiene tres subestados basados en su clasificación de pago
 - Hourly, Salaried, and Commissioned
 - La clasificación de pago puede cambiar en cualquier momento
 - Las clasificaciones de pago se crean/borran después de entrar/salir del subestado

Ejercicio: Comportamiento de Objetos (cont.)

- Información del estado Leave of Absence
 - Un empleado puede tomar un permiso de ausencia de hasta 1 año mientras esté en cualquier subestado de empleado
 - Al regresar de su permiso ingresa al mismo subestado de pago del estado Employed
- Información del estado Terminated
 - Un empleado puede ser despedido mientras este en cualquier subestado de empleado
 - Un empleado puede renunciar mientras este en cualquier subestado de empleado
 - El registro de empleado se marca como terminado en este estado
- Información del estado Retired
 - Un empleado se retira cuando tiene 65 años
 - La información de pensión se calcula en este estado

Homogeneización



Objetivos: Homogeneización

- Usted podrá:
 - Discutir el por qué de la homogeneización
 - Determinar cuando deben combinarse dos clases, cuando debe dividirse una clase, cuando debe eliminarse una clase
 - Evaluar diagramas de clases y diagramas de interacción para mantener la consistencia del modelo
 - Discutir las necesidades de reestructuración de paquete

¿Qué es Homogenización?

- Homogeneizar

“mezclar en un conjunto de elementos de manera uniforme, para homogeneizar”

Webster's New Collegiate Dictionary

- Entre más casos de uso y escenarios se desarrollen es necesario homogeneizar el modelo

- Esto es especialmente cierto si existen múltiples equipos que están trabajando en diferentes casos de uso

¿Qué buscar?

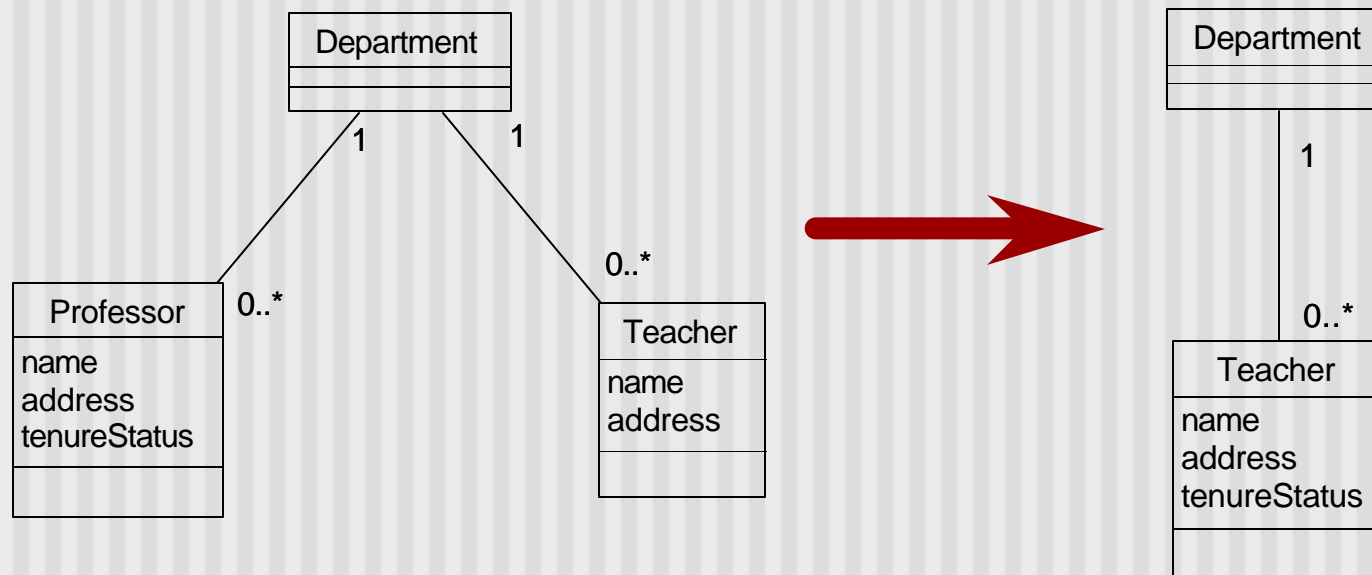
- Las clases se examinan para determinar si
 - Se pueden combinar dos o más clases
 - Se debe dividir una clase
 - Se debe eliminar una clase completamente
- Se reestructuran los paquetes para resolver problemas de
 - Acoplamiento
 - Reutilización
 - Visibilidad

Combinación de Clases

- Cuando se realiza el análisis de casos de uso con varios equipos, cada uno puede asignar un nombre diferente a una misma clase
 - Esto es especialmente cierto ya que el lenguaje natural se usa para el análisis de casos de uso
- Debe conducirse el desarrollo del modelo
 - Evaluar definiciones de clase
 - Evaluar la estructura y comportamiento de las clases
 - Buscar sinónimos
 - Tomar el nombre más cercano al dominio

Ejemplo: Combinación de Clases

- Se eligió a **Teacher**, ya que no todos los instructores de la Universidad han alcanzado el estatus de Professor

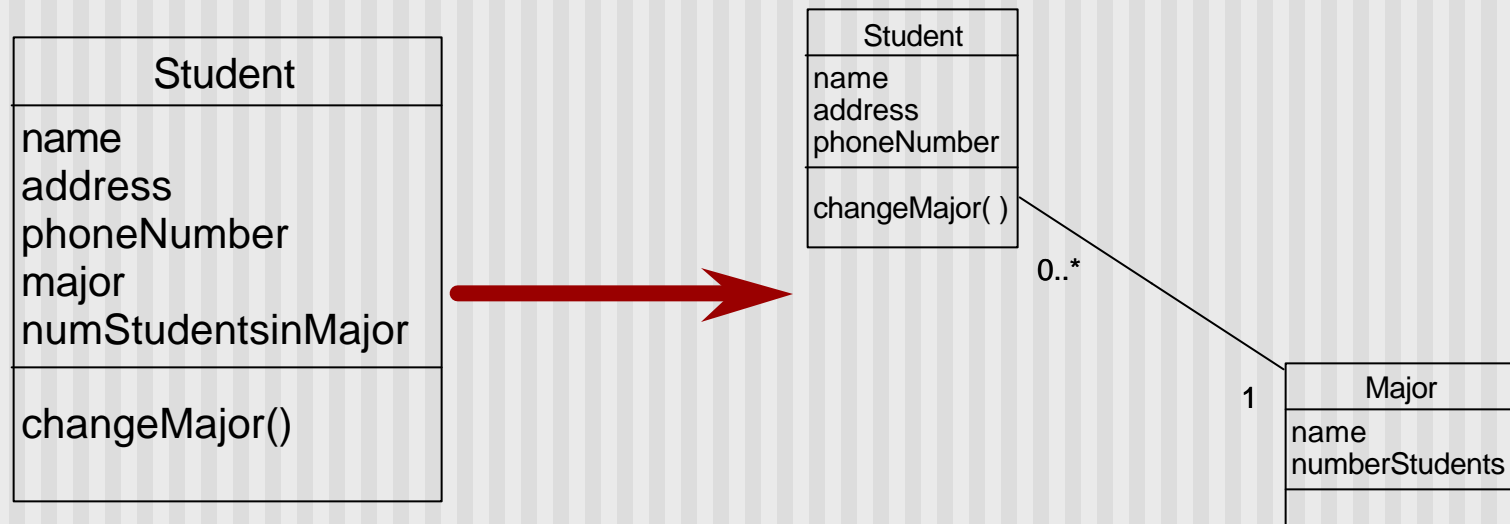


Ejemplo: Combinación de Clases

- Al colocar una clase de control por cada caso de uso, es necesario re-evaluar dichas clases
 - Las clases de control con comportamiento similar pueden combinarse
- Ejemplo: El Administator interactúa con los casos de uso **Maintain Student Information** y **Maintain Professor Information**
 - Se crearon dos clases de control
 - La secuencia de acciones en cada una de estas dos clases de control es muy similar (revisar, crear, borrar información del actor)
 - Las clases de control pueden combinarse en una clase de control que se llame `RegistrationUserManager`

División de Clases

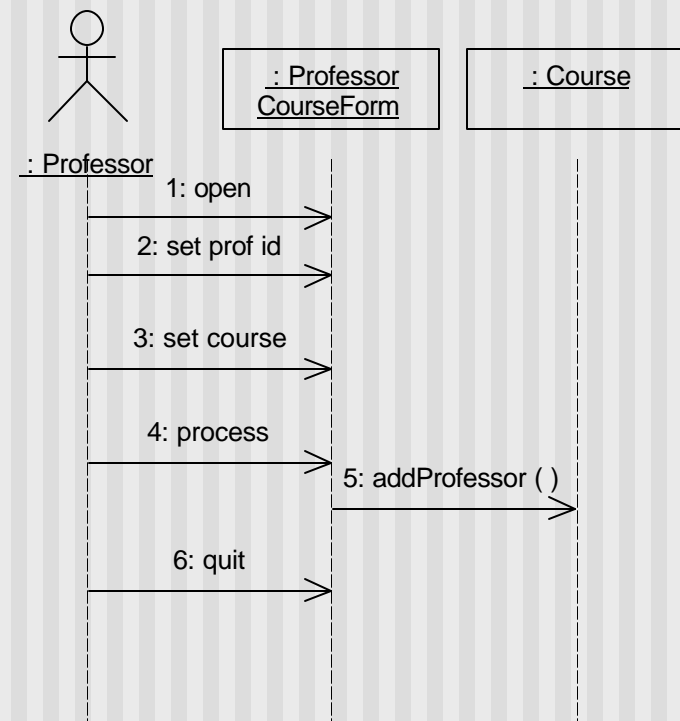
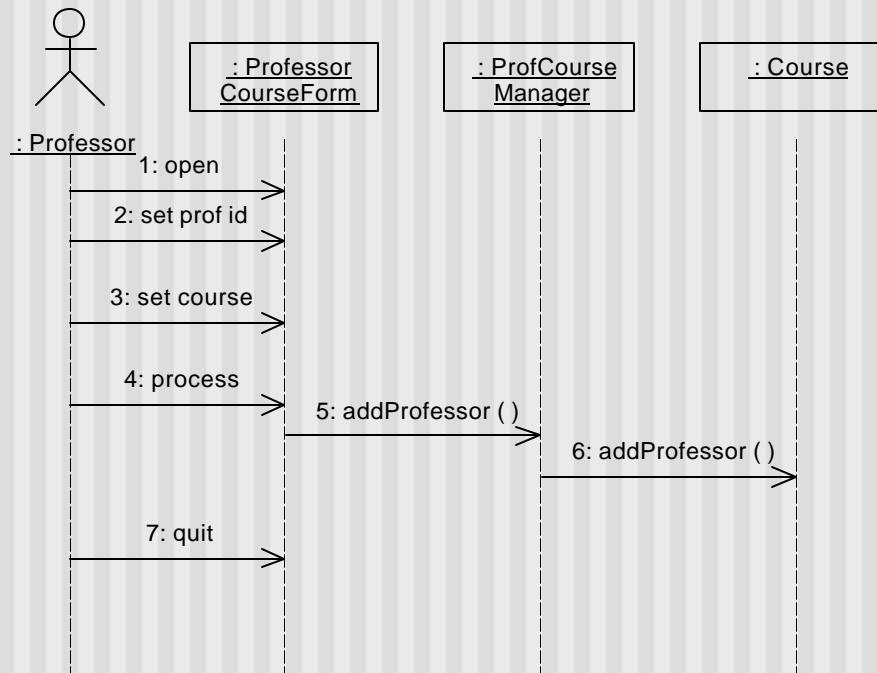
- Las clases con estructura y/o comportamiento que no sean cohesivos deben dividirse en clases diferentes
- Ejemplo:



Eliminación de Clases

- Debe eliminarse por completo una clase del modelo
 - Que no tenga ninguna estructura o comportamiento
 - Que no participe en ningún caso de uso
- En particular, examine las clases de control
 - La falta de responsabilidad en la secuencia puede conducir a la eliminación de la clase de control
- Ejemplo:
 - Un caso de uso para el actor Professor es "Seleccionar cursos para impartir"
 - Esto quiere decir que Professor introduce el nombre y número del curso y se crea una liga a la clase ProfessorInformation
 - Ya que no hay mucho comportamiento relacionado a la secuencia, se elimina esta clase de control

Diagrama de Secuencias Actualizado



Revisión de Consistencia

- Se debe revisar la consistencia de los Diagramas de Clases y los Diagramas de Interacción
 - ¿Se necesita al menos una operación para cada clase?
 - ¿Existe una clase para cada objeto en el Diagrama de interacción?
 - Si un diagrama de objeto o diagrama de interacción muestra un mensaje que va de un objeto de la clase A a un objeto de la clase B, verifique:
 - **Exista una operación en la clase B espera un evento y lo manipula**
 - **Exista una asociación correspondiente entre la clase A y la clase B en el diagrama de clases**
 - **El diagrama de transición de estado para la clase B (si se ha desarrollado uno) incluye ese evento**

Reestructuración de Paquetes

- Al desarrollar más casos de uso puede ser necesario reestructurar los paquetes del modelo
 - El fuerte acoplamiento entre paquetes puede significar que los paquetes deben combinarse
 - La dependencia reciproca entre paquetes puede significar que el paquete debe dividirse
 - Evalúe consideraciones de reuso
 - **Un paquete reusable debe tener algunas dependencias**
 - Evalúe las partes públicas y privadas de un paquete
 - **No todas las clases en un paquete deben ser parte de la interfaz pública del paquete**
 - **Agregar clases boundary si es necesario encapsular la interfaz del paquete**

Ejercicio: Homogeneización

- Discutir las decisiones de homogeneización que se necesitan en este punto del análisis

Diseño Arquitectónico



Objetivos: Diseño Arquitectónico

- Usted podrá:
 - Listar los atributos de una buena arquitectura
 - Explicar la arquitectura de "4 + 1" vistas
 - Explicar el propósito de los diagramas de componentes
 - Explicar el propósito de los diagramas de distribución

Visión Arquitectónica

- Dos cualidades comunes para virtualmente todos los proyectos OO son:
 - La existencia de una visión arquitectónica
 - La aplicación de un ciclo de vida incremental bien-manejado e interactivo
- La arquitectura debe ser simple
 - El logro del comportamiento común a través de abstracciones y mecanismos comunes

Una Definición de Arquitectura del Software

“La arquitectura del software tiene que ver con la organización de sistemas de software, la selección de sus componentes, las interacciones entre estos componentes, la composición de componentes interactuando en subsistemas más grandes progresivamente y el total de los patrones que guían estas composiciones. Tiene que ver no sólo con la estructura de sistemas, sino también con su funcionalidad, desempeño, diseño, selección entre alternativas y comprensión”

Mary Shaw

Atributos de las Arquitecturas Buenas

- Las buenas arquitecturas se construyen en capas de abstracción bien definidas:
 - Cada capa representa una abstracción coherente
 - Cada capa tiene una interfaz bien definida y controlada
 - Cada capa se construye sobre facilidades bien definidas y controladas en niveles de abstracción bajos
- Hay una clara separación entre la interfaz y la implementación de cada capa
 - Los cambios en la implementación de una capa no viola la hipótesis hecha por sus clientes

Desarrollo de la Arquitectura del Sistema

- El diseño de la arquitectura tiene que ver con el manejo de riesgo
- Las buenas arquitecturas se determinan mejor a través de desarrollo iterativo e incremental
- A un experimentado equipo pequeño, guiado por un Arquitecto en Jefe, se le debe asignar la responsabilidad para:
 - Definir y mantener la integridad arquitectónica del sistema
 - Aprobar todos los cambios a las interfaz de paquetes
 - Valorar riesgos del proyecto
 - Proponer el orden y contenido de cada iteración

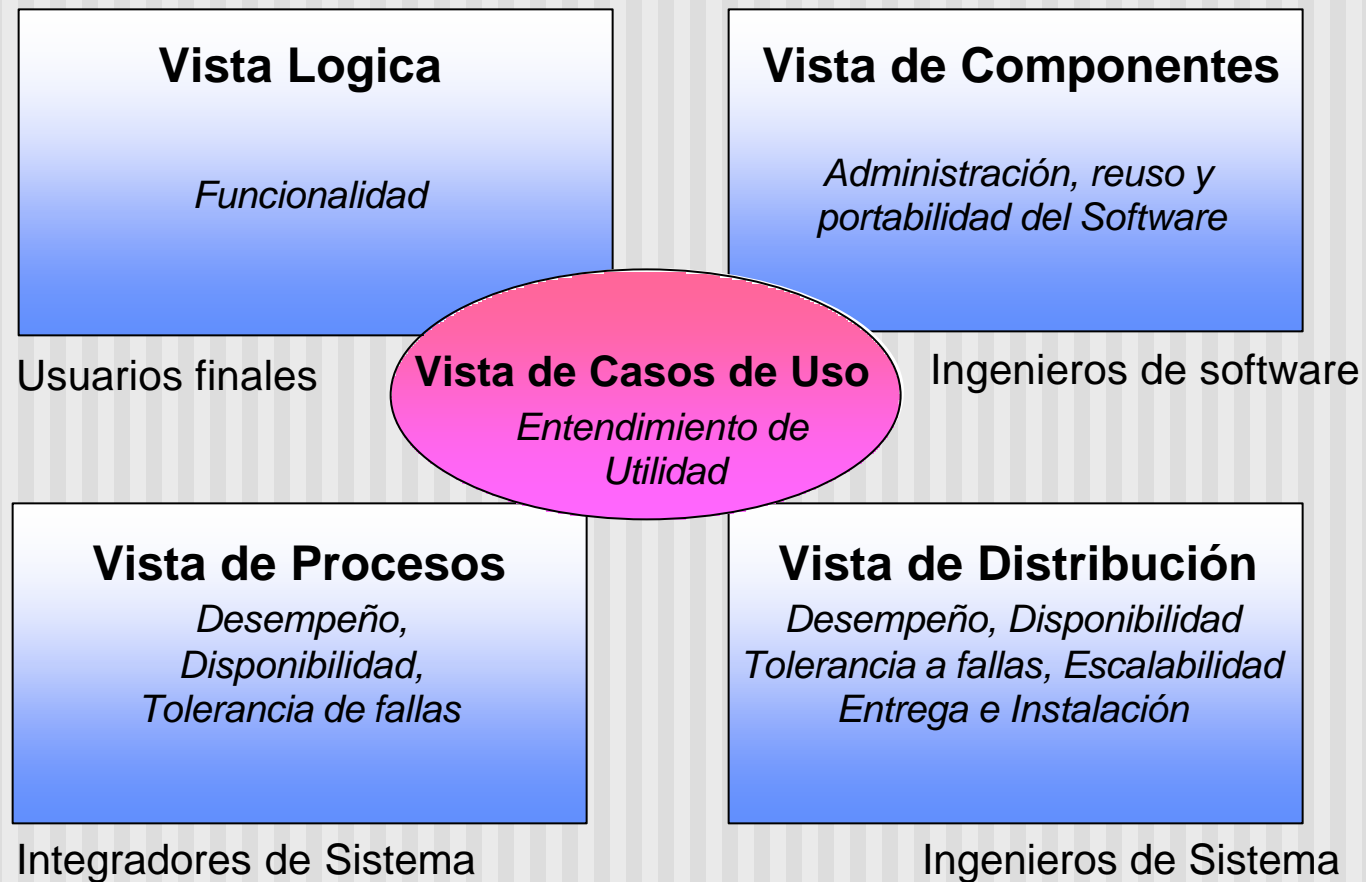
Dimensiones de la Arquitectura del Software

- Perspectivas diferentes para inversionistas diferentes
 - Usuario final, cliente, administrador de proyecto
 - Ingeniero de sistema, desarrollador, arquitecto, evaluador
- Las perspectivas múltiples requieren múltiples vistas
 - Los diagramas de clases no muestran el mapeo del sistema al hardware
 - Los diagramas de bloques de hardware no describen que partes del sistema son obtenidas de software comercial

Una Arquitectura requiere Múltiples Vistas

- Para describir completamente una arquitectura, se necesitan cuatro vistas:
 - Una Vista Lógica que proporciona una imagen estática de las principales clases y sus relaciones
 - Una Vista de Componentes que muestra como está el código organizado en paquetes y librerías, así como el software comercial (COTS, commercial off-the-shelf)
 - Una Vista de Procesos que muestra procesos y tareas
 - Una Vista de Distribución que muestra los procesadores, dispositivos y ligas en el ambiente operacional
- Finalmente, una Vista de Casos de Uso que explica como trabajan juntas las otras cuatro vistas

El Modelo "4+1 Vistas"

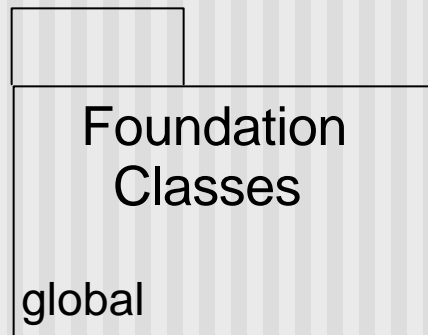


Vista Lógica

- La vista lógica de una arquitectura controla los requerimientos funcionales del sistema
 - Lo que el sistema proporciona en términos de servicios a sus usuarios
- Proporciona una imagen estática de las principales clases y sus relaciones
- La vista lógica se encuentra reflejada en diagramas de clases; paquetes que contienen clases y relaciones, lo cual representa la abstracción del sistema en desarrollo

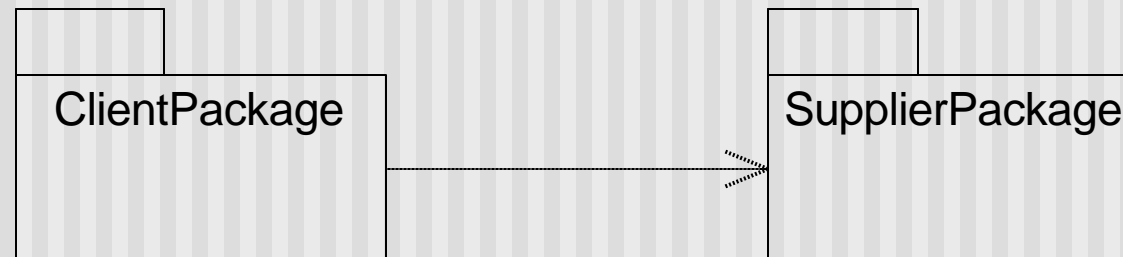
Paquetes Lógicos Globales

- Ciertos paquetes son usados por todos los demás paquetes
 - Foundation Classes
 - Conjuntos, listas, colas, etc.
 - Clases de manejo de errores
- Estos paquetes se marcan como globales



Implicaciones de Dependencia

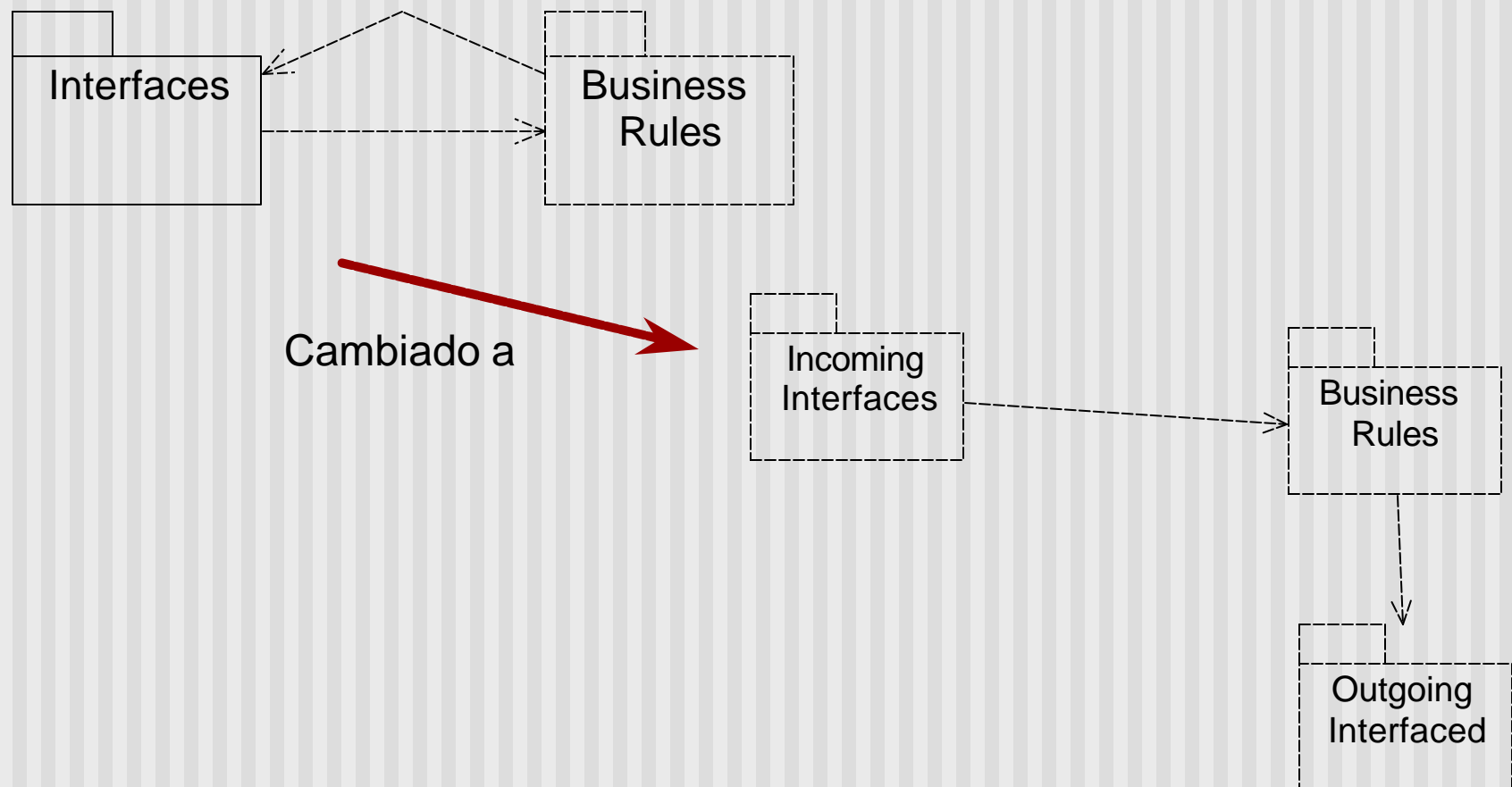
- Algunas implicaciones de la dependencia entre paquetes son:
 - Cada vez que se hace un cambio al paquete Proveedor, el paquete Cliente debe ser re-compilado y reevaluado
 - El paquete Cliente no puede ser reusado independientemente, ya que depende del paquete Proveedor



Evitar Dependencias Circulares

- Es deseable que la jerarquía del paquete sea a-cíclica
- Esto quiere decir que la situación siguiente debe evitarse (de ser posible)
 - El paquete A usa el paquete B el cual usa el paquete A
- Una dependencia circular como esta, significa que los paquetes A y B tendrán que ser tratados como un solo paquete
- También deben evitarse los círculos con más de dos paquetes
 - El paquete A usa el paquete B que usa al paquete C que usa al paquete A
- Las dependencias circulares pueden romperse, dividiendo uno de los paquetes en dos paquetes más pequeños

Dependencia Circular en el Sistema de Inscripción



Interfaz Pública de un Paquete

- Una interfaz de paquete debe encapsular su implementación detrás de una interfaz pública justo como lo hace una clase
- Cada clase en un paquete tiene un parámetro de control de exportación que puede establecerse a público o implementación (privado)
 - Solo las clases públicas pueden ser usadas por clases en otros paquetes
 - Las clases de implementación (privadas) sólo pueden ser usadas por su paquete

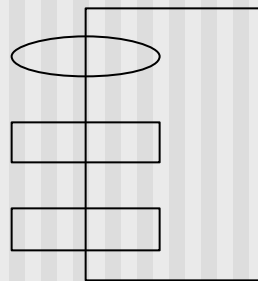
Vista de Componentes

- La vista de componentes tiene que ver con la organización del software en módulos para su desarrollo
- Los diagramas de componentes se crean para mostrar los paquetes y los componentes que conforman el sistema en desarrollo
 - Muestra la distribución de clases a componentes
 - Muestra la distribución de componentes en paquetes
- Los paquetes se organizan en capas, donde cada capa tiene una interfaz bien definida

¿Qué es un componente?

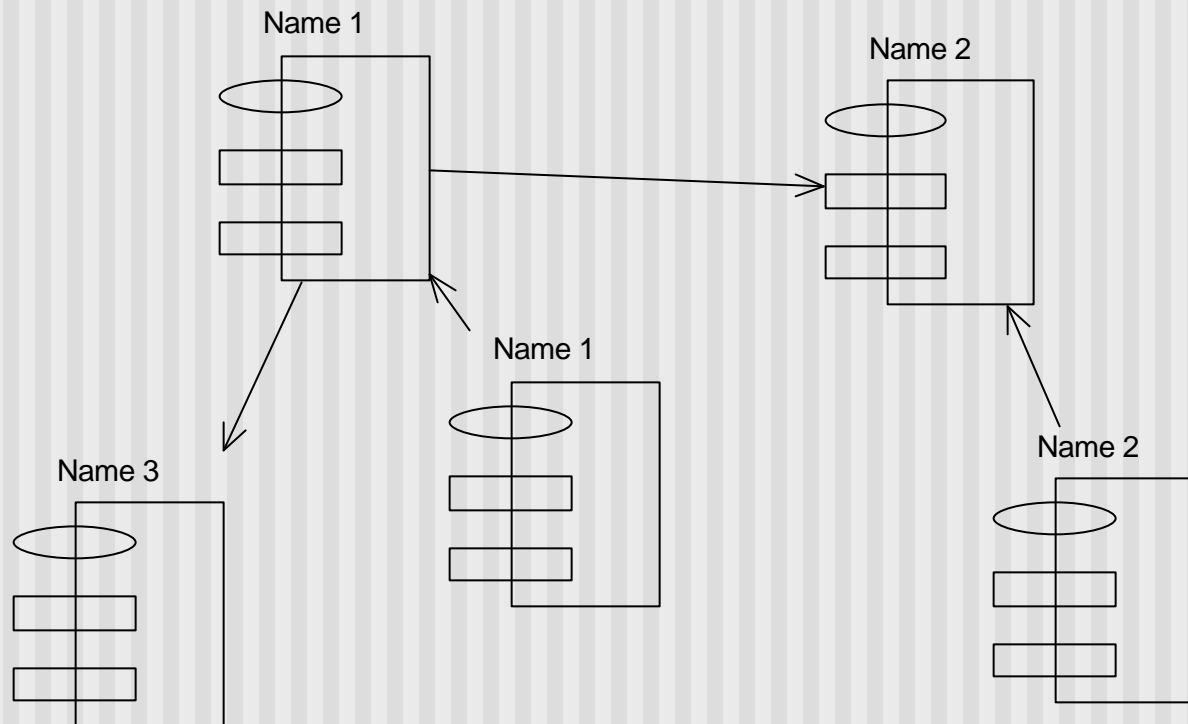
- Un componente es una unidad de código fuente que sirve como bloque constructor para la estructura física de un sistema
- Los estereotipos (con iconos alternos) pueden usarse para definir tipos de componentes específicos.
 - Ejemplos: exe, dll, main programs, headers, modules, forms
- Agrupar clases en un componente, ya sea que tenga una función cooperativa o que necesite estar en proximidad para la eficiencia en la implementación
- Notación de componente:

Component name



Diagramas de Componentes

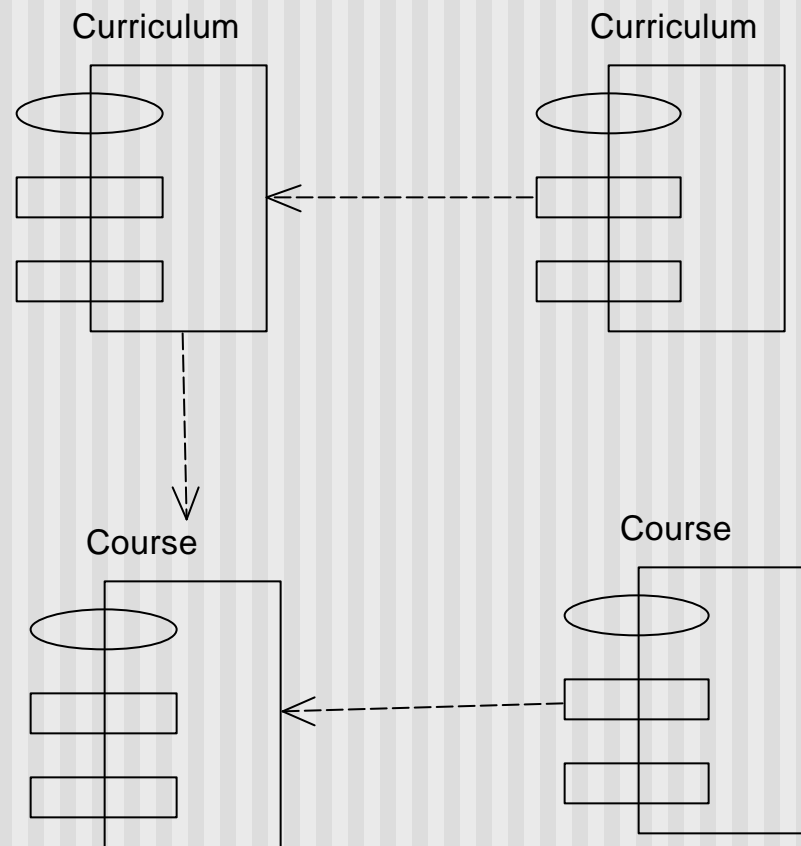
- Un diagrama de componentes muestra la distribución de clases y objetos en componentes durante la implementación, así como sus dependencias de compilación



Diagramas de Componentes (cont.)

- Se requiere un nombre para cada componente, este nombre denota típicamente el nombre simple del archivo físico correspondiente en el espacio de trabajo actual
- La única relación es la dependencia de compilación, representada por una línea punteada dirigida que apunta hacia donde existe la dependencia
- En C++, la dependencia de compilación se indica por las directivas `#include`
- No debe haber ciclos en un conjunto de dependencias de compilación

Ejemplo: Diagrama de Componentes



Paquetes en la Vista de Componentes

- Un paquete en la vista de componentes es una colección de componentes, algunos de los cuales son visibles a otros paquetes y otros están ocultos
- Un paquete agrupa componentes que están lógicamente relacionados
- Un paquete en la vista de componentes agrupa componentes de manera similar a la que un paquete en la vista lógica agrupa clases
- Cada componente en el sistema debe existir en un solo paquete en el nivel más alto del sistema
- Notación:

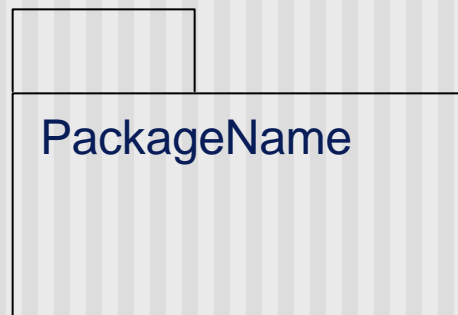
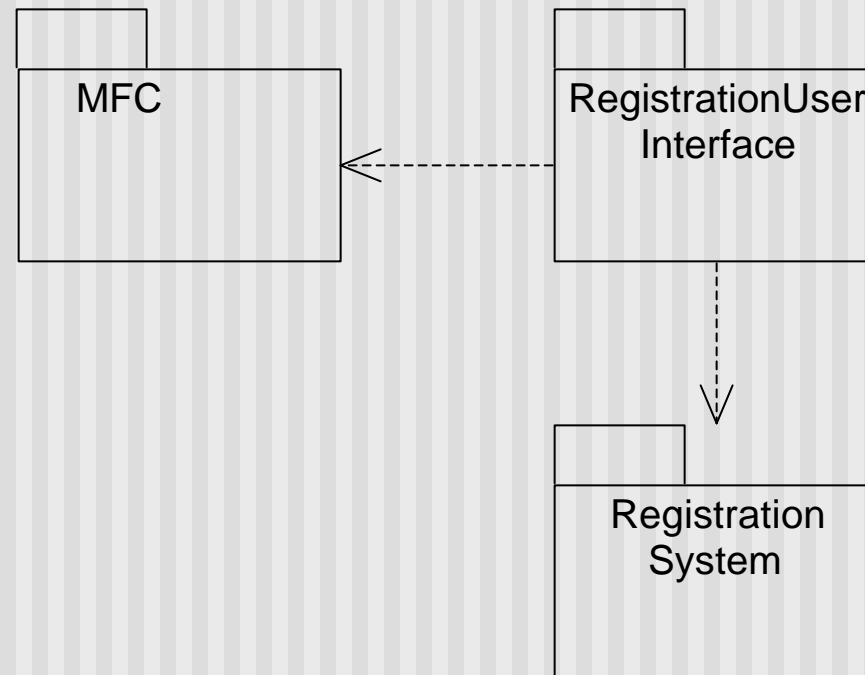


Diagrama de Componentes Principal

- Un diagrama de componentes principales es una familia de paquetes conectados por ligas dirigidas que representan dependencias

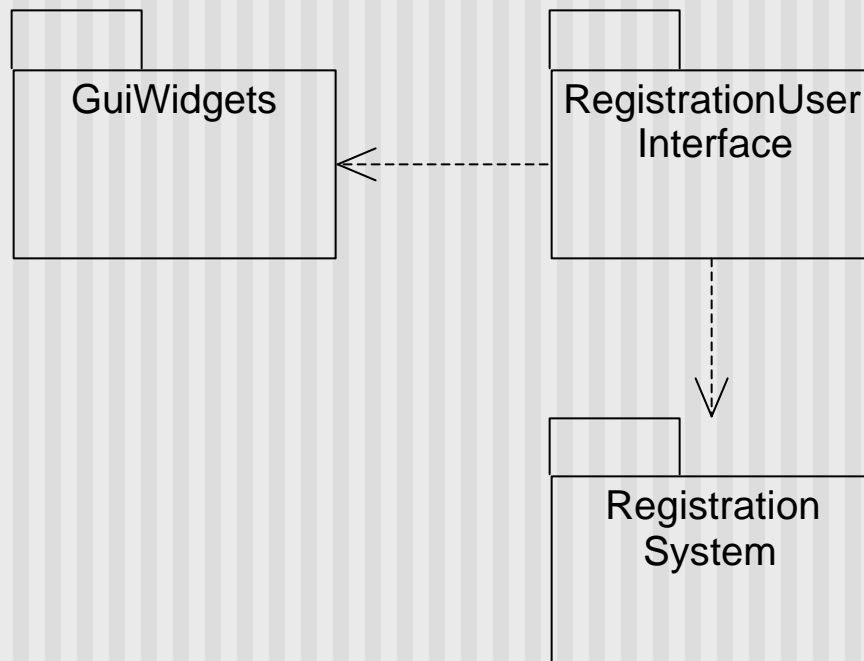
- Ejemplo.



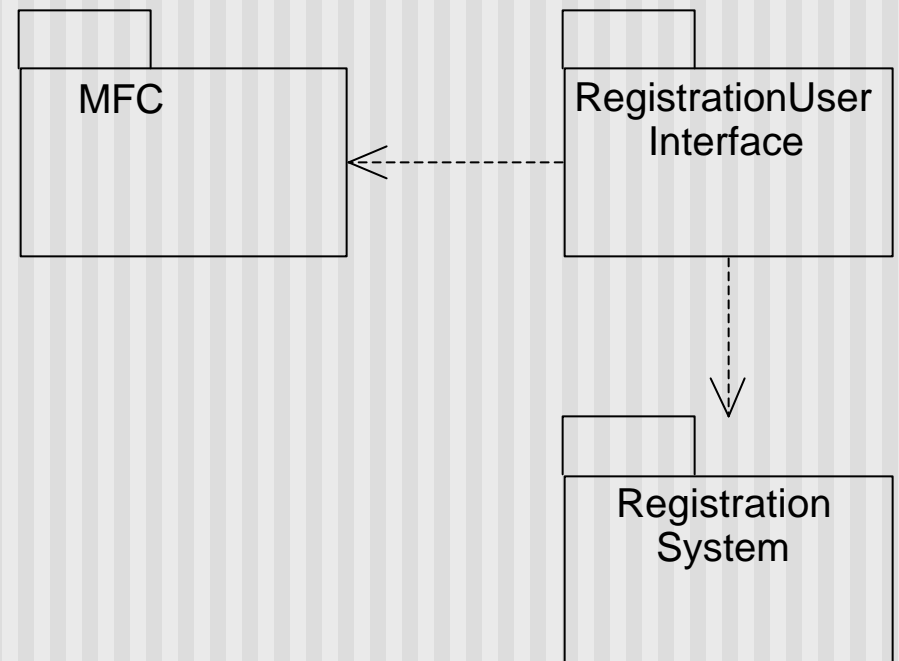
Relación entre Paquetes de la Vista Lógica y de Componentes

- En general, un paquete en la vista lógica puede corresponder directamente a un paquete de componentes en la vista de componentes
- La estructura lógica y física puede variar por las siguientes razones:
 - Se fusionan paquetes en la vista lógica, para mantener juntos a objetos que se comunican frecuentemente
 - Los paquetes en la vista de componentes se agregan para implementar funcionalidad de bajo nivel

Ejemplo: Correspondencia entre Paquetes en la Vista Lógica y de Componentes



Logical View Top-Level Diagram



Component View Top-Level Diagram

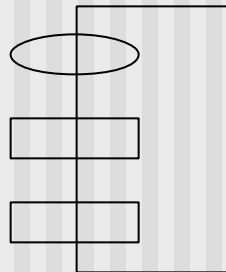
Vista de Procesos

- La vista de procesos de la arquitectura se enfoca en la descomposición de procesos
 - La vista muestra la distribución de componentes a procesos
- El diagrama de componentes se actualiza para mostrar la distribución de componentes a procesos
- La vista de procesos está dirigida a: la disponibilidad, la confiabilidad, el desempeño, la administración y la sincronización del sistema

Componentes de Proceso

- Las librerías ejecutables y ligadas se representan como componentes UML
 - Especificación de Paquete (DLL)
 - Especificación de Tarea (EXE)

Especificación del paquete (DLL)



Especificación de tarea (EXE)

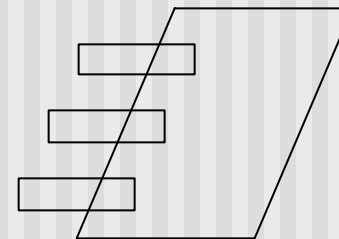
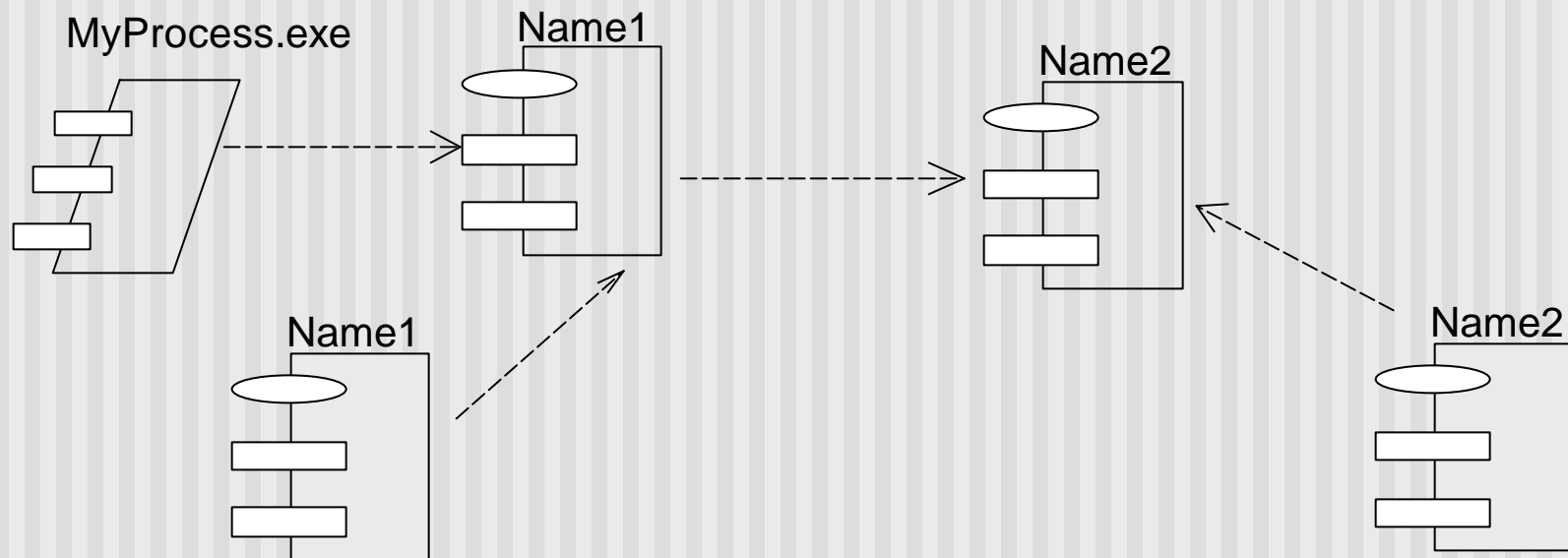


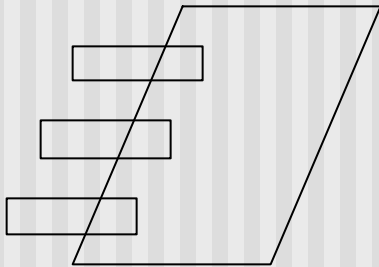
Diagrama de Componentes para un Proceso

- Cada componente puede depender de otros componentes



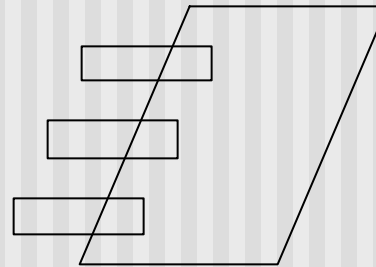
Procesos para el Sistema de Inscripción a Cursos

Curriculum.exe



**Proceso para la creación
y mantenimiento de
curriculum**

Registration.exe



**Proceso para que los alumnos
y profesores elijan curso**

Vista de Distribución

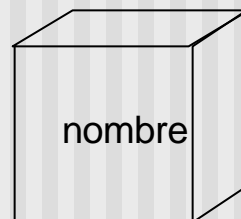
- La vista de distribución de la arquitectura mapea componentes a nodos de procesamiento
- Los requerimientos de desempeño y tolerancia a fallas se toman en cuenta
- Los diagramas de distribución se crean para mostrar los diferentes nodos (procesos y dispositivos) en el sistema

Diagrama de Distribución

- Un diagrama de distribución muestra la ubicación de los componentes en nodos, de tal forma que se obtenga una vista de distribución del sistema
 - Los procesadores y dispositivos son estereotipos comunes de Nodo.
- Los nodos se conectan en el diagrama a través de una línea, que refleje la ruta de comunicación entre ellos
- Los elementos esenciales de un diagrama de distribución son los nodos y las conexiones

Notación para Diagramas de Distribución

- Un nodo es un objeto físico en tiempo de ejecución que representa recursos computacionales
- Una conexión indica comunicación, generalmente a través de medios de acoplamiento directo al hardware



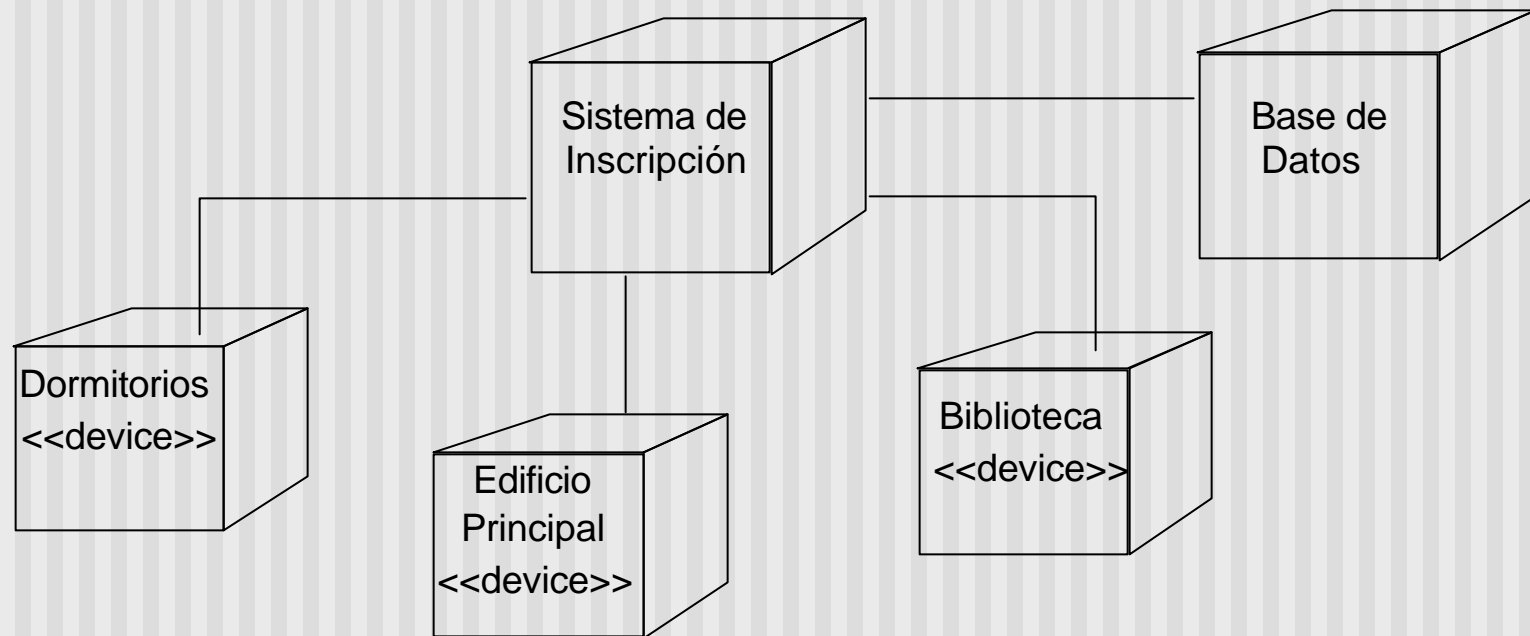
nodo

etiqueta

conexión

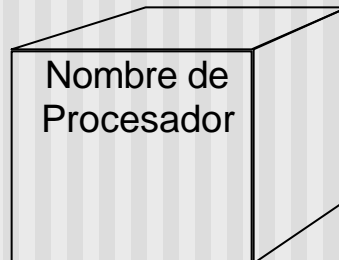
Ejemplo: Diagrama de Distribución para el Sistema de Inscripción

- Este diagrama muestra dos nodos y los dispositivos con los que se comunica el Sistema de Inscripción



Procesos

- Un proceso es un hilo de control de la ejecución de un programa o sistema (OO)
 - Un sistema grande puede dividirse en procesos múltiples o hilos de control
- Los objetos se asignan a procesos (sus asignaciones pueden ser dinámicas)
- Los procesos se asignan a procesadores (un conjunto de procesos puede ser dinámico)
- Notación:



proceso 1, proceso 2, ... proceso n

Mapeo de Paquetes de Desarrollo a Procesos Ejecutables

- El mapeo de paquetes de desarrollo a procesos ejecutables envuelve el entendimiento de la topología del sistema y las prioridades del sistema, que incluyen:
 - Arquitectura de procesador, velocidad y capacidad
 - Mantener las asociaciones de clases juntas para minimizar la comunicación de interprocesos (IPC interprocess communication)
 - Estrategia IPC -- cliente/proveedor u otro?
 - Técnicas de proceso distribuido
- Debe resolver elementos que envuelvan múltiples procesadores de hardware o sistemas distribuidos durante el diseño del sistema

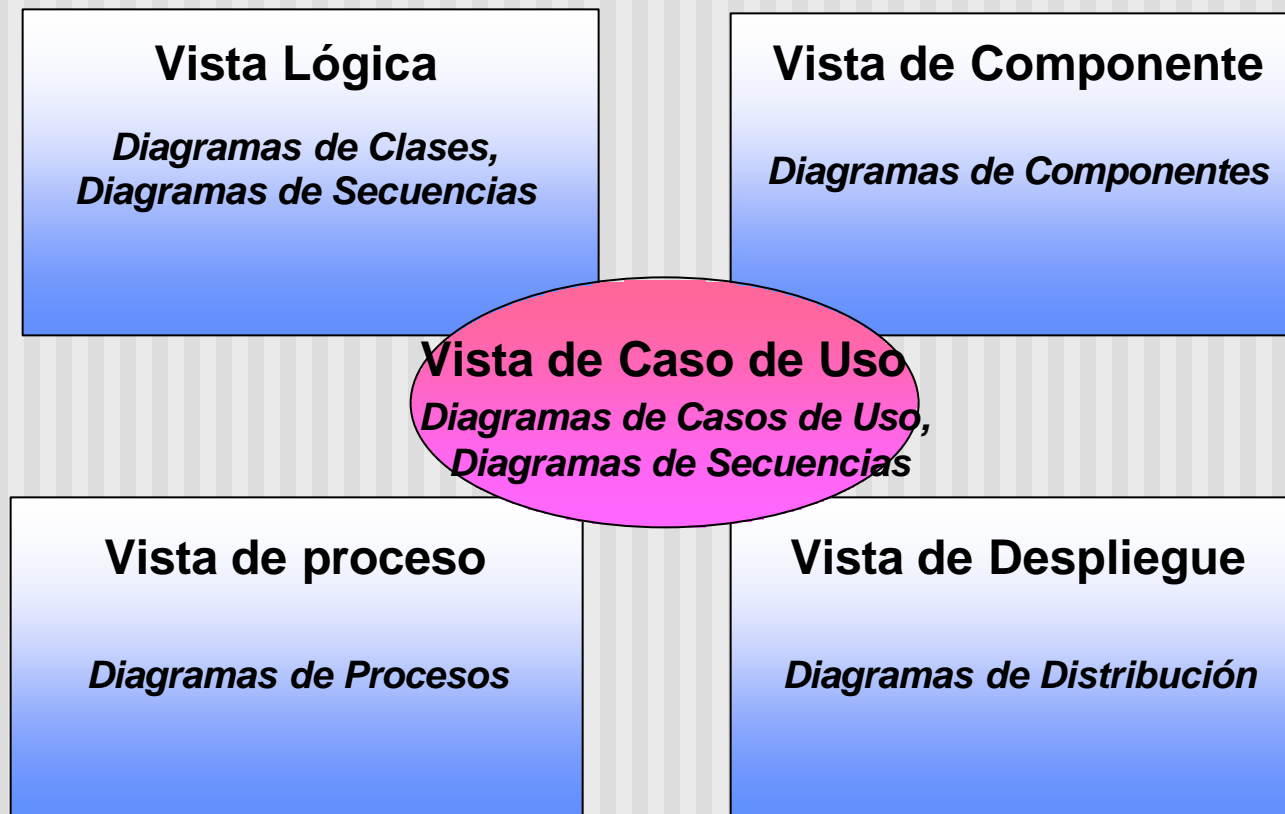
Mapeo de Procesos Ejecutables a Hardware

- Los procesos deben asignarse a un dispositivo de hardware para su ejecución
- Entre las consideraciones están:
 - Tiempo de respuesta y resultados del sistema
 - Comunicación: ancho de banda/capacidad
 - Localización física del hardware requerido
 - Necesidades de procesamiento distribuido
 - Balanceo de carga de procesos en sistemas distribuidos
 - Tolerancia de fallas
 -

Vista de Casos de Usos

- Los casos de usos son los conductores del diseño de una arquitectura
 - Abstracciones de requerimientos largos y complejos
 - Identificación de interfaz crítica
 - Forzar a los diseñadores a concretar elementos
- Demuestran y validan las vistas; lógica, de componentes, de procesos y de distribución de la arquitectura

Las "4 + 1 Vistas" del Modelo UML



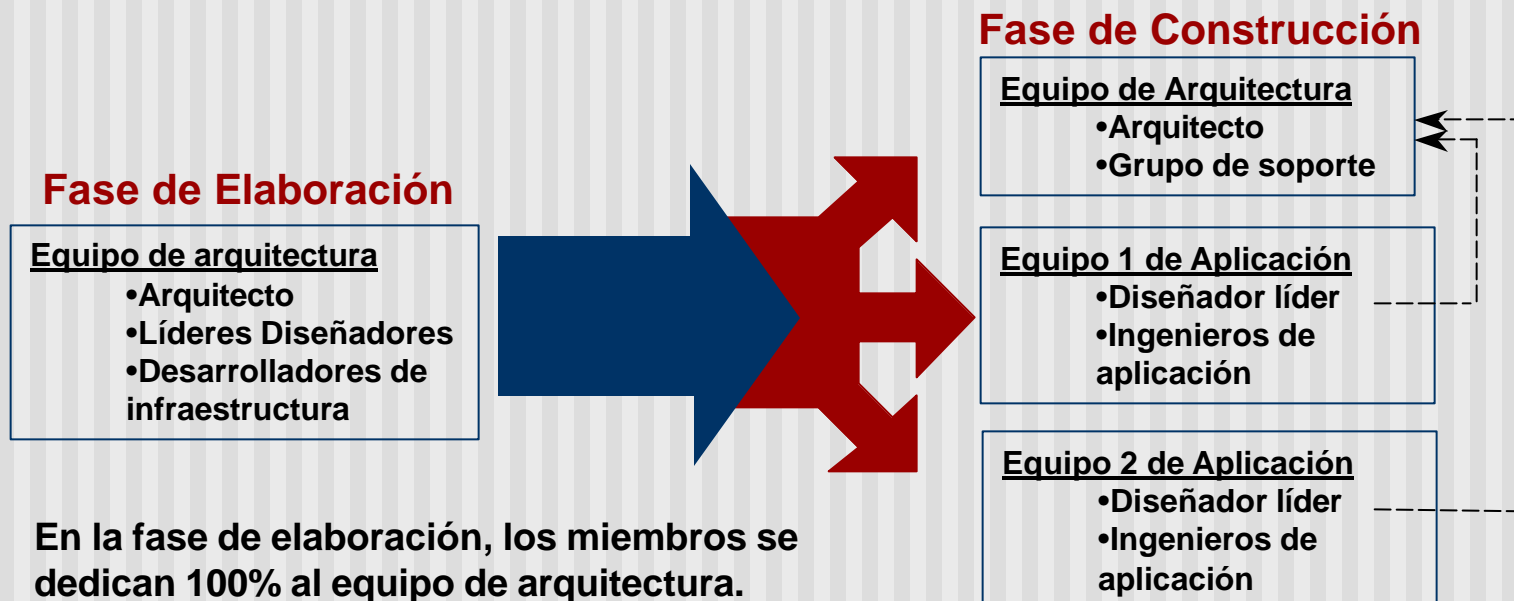
¿Cómo se documenta una Arquitectura?

- La arquitectura se documenta a través de un texto escrito
 - Aproximadamente 100 páginas para un sistema grande
- El documento incluye:
 - Una descripción de la filosofía de la arquitectura (las vistas) y la visión que dirige los requerimientos
 - Ver las ventajas y desventajas para considerar alternativas
 - Una vista de alto nivel de la vista lógica (paquetes y clases principales)
 - Escenarios específicos de la arquitectura
 - Vistas de alto nivel de las vistas de procesos y de distribución
 - Los mecanismos clave

¿Quién desarrolla la Arquitectura del Software?

- El equipo de arquitectura está compuesto por los mejores y más experimentados desarrolladores
- Establecido tempranamente en el proyecto (no después de la fase de elaboración)
- La mayoría de los proyectos de complejidad razonable requieren de un equipo de arquitectura, NO un sólo arquitecto
 - Encabezado por el arquitecto en jefe, quién dedica 100% de su tiempo
 - Incluye los líderes diseñadores para funciones más importantes o críticas del sistema

Evolución del Equipo de Arquitectura



En la fase de elaboración, los miembros se dedican 100% al equipo de arquitectura.

Durante la fase de construcción, los miembros se convierten en diseñadores líderes para equipos de aplicación y soporte de medio tiempo al equipo de arquitectura

•
•
•

Beneficios de un Equipo de Arquitectura

- Documentos a entregar
 - Documento de arquitectura
 - Partes del documento de diseño de bajo nivel
 - Guías de diseño y programación
 - Elementos de los planes de liberación
 - Auditorias de diseño al sistema a liberar
- La habilidad y efectividad del equipo de arquitectura es crítico para el éxito de un proyecto

Con una buena arquitectura, un equipo de desarrollo normal puede triunfar. Con una arquitectura débil, hasta los desarrolladores más expertos no tendrán éxito

Ejercicio: Diseño de Arquitectura

- Discutir consideraciones de arquitectura para el problema
- Agregar paquetes al modelo como sea necesario
 - Reubicar clases en diferentes paquetes como sea necesario

Mecanismos Clave



Objetivos: Mecanismos Clave

- Usted podrá:
 - Describir algunos mecanismos claves específicos a OO
 - Explicar los elementos asociados con la interfaz a bases de datos
 - Listar algunas consideraciones para evaluar sistemas de administración de bases de datos
 - Describir el manejo de excepciones y sus elementos asociados
 - Explicar los elementos asociados con comunicación inter-proceso

¿Qué son los Mecanismos Clave?

- Un mecanismo clave es una decisión estratégica de acuerdo a estándares, políticas y prácticas comunes, por ejemplo
 - Un acercamiento común a un manejo de error, o
 - Un modo común de comunicación entre procesos
- La mayoría del software tradicional se diseña con principios que aún se aplican en el diseño OO
 - Los problemas para resolver son similares, e.g., manejo de recursos, control de riesgos, etc.
- Algunas diferencias se deben a:
 - Soluciones estructuradas usando métodos OO
 - Los elementos de lenguajes de programación OO

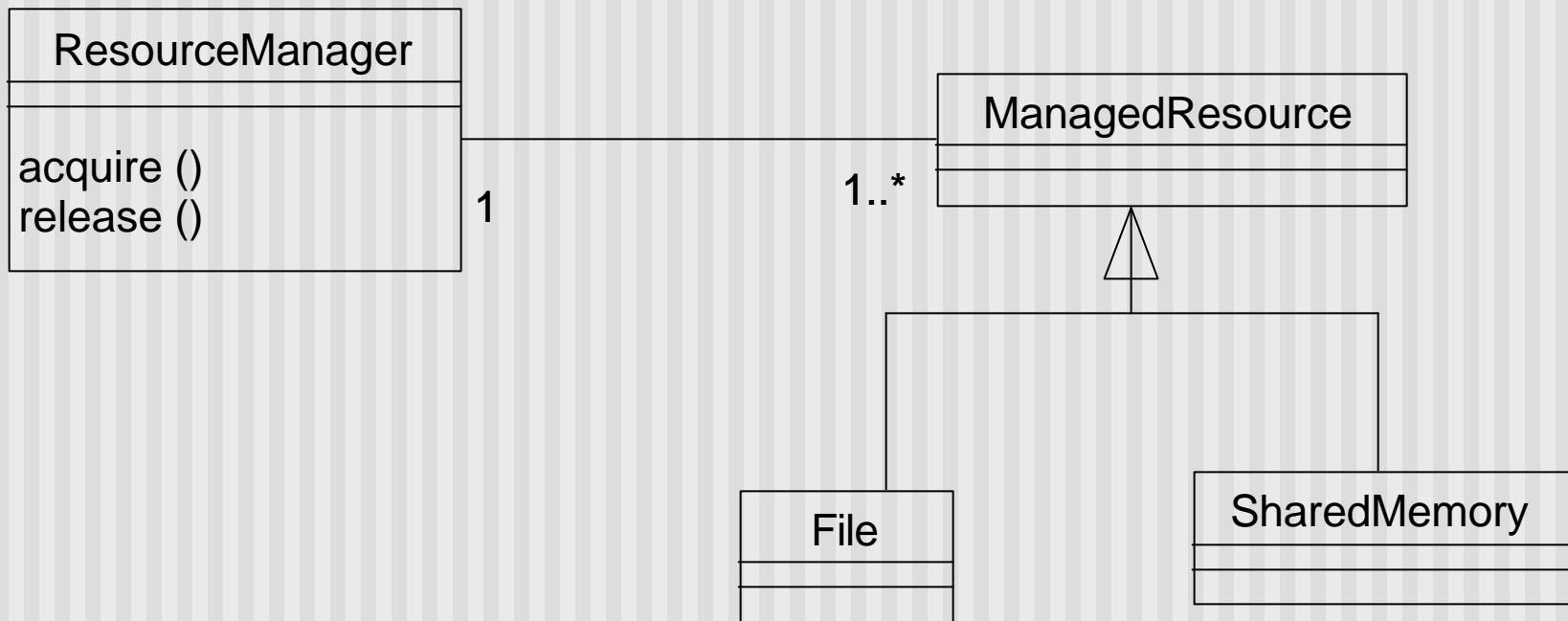
Mecanismos Claves Comunes

- Administración de recursos
- Manejo especial requerido para el inicio y salida del sistema
- Integración con sistemas de almacenamiento de datos persistentes
- Detección/manejo/reporte de errores
- Comunicación interproceso
- Envío de mensajes
- Apariencia y vista (look & feel) de la interfaz de usuario
- Reutilización de software

Administración de Recursos

- Una clase administradora de recursos puede emplearse para controlar el acceso a los recursos
- La clase administradora de recursos utiliza métodos de software tradicional, tales como semáforos para controlar el acceso a dichos recursos
- Esta clase proporciona operaciones que permiten a los clientes del recurso, obtenerlo, descargarlo, obtener su estado, etc.
- Una superclase que contenga la interfaz a los recursos administrados puede ser provista con los recursos del administrador para posibilitar su reuso

Administración de Recursos (cont.)



Inicio y Salida del Sistema

- Si aún no se ha cubierto durante el análisis, deben definirse los casos de uso para el inicio y salida del sistema
- Los escenarios deben desarrollarse para cada caso de uso -- tantos como sean necesarios para controlar a la mayoría de las situaciones normales y anormales
- Durante este proceso deben descubrirse nuevos estados y comportamientos para clases existentes y la necesidad debe surgir para las clases enteramente nuevas y así controlar el inicio y/o salida del sistema

Objetos Persistentes

- Un objeto persistente es aquel que lógicamente existe bajo el ámbito del programa que lo creó
- Los lenguajes de programación OO manejan objetos residentes en memoria, los cuales son esencialmente transitorios
- Un objeto persistente tiene la habilidad de guardar el valor de sus atributos en algún tipo de almacenamiento persistente
- Un objeto persistente puede también crearse en memoria e inicia con sus valores de atributo desde el almacenamiento persistente
- La estrategia total para proveer persistencia a los objetos en el sistema, es un mecanismo de control

Almacenamiento Persistente

- El almacenamiento persistente puede hacerse empleando un sistema de archivos simple o algún tipo de sistema de administración de base de datos
- Hay varios modelos para DBMSs:
 - Jerárquicos
 - Red
 - Relacional (RDBMS)
 - Orientado a Objetos (ODBMS)
- El Relacional y el ODBMS son los más comunes

Selección de una Aproximación a Persistencia

- La estrategia de diseño para retener objetos persistentes deberá considerar
 - Tiempos de acceso
 - Capacidad de almacenamiento
 - Confiabilidad del sistema de almacenamiento
 - Acceso a datos existentes
- El modelo elegido influirá al sistema y al diseño de objetos, de tal forma que los diseñadores deberán considerar:
 - Operaciones adicionales para almacenar y recuperar objetos persistentes
 - Adición de atributos para manejar detalles de almacenamiento del sistema
 - Clases adicionales para hacer interfaz con el DBMS

Criterios de Evaluación de un DBMS

- Se debe decidir el criterio de evaluación para elegir un DBMS
- Las siguientes laminas contienen cierto criterio encontrado en "Considerations For Evaluating Object Database Management Systems" escrito por Robert Gancarz y Grant Colley, Object Magazine, Marzo/Abril 1992
 - Este criterio también se aplica para elegir el sistema de administración de la base de datos

Criterios de Evaluación de un DBMS (cont.)

- Presencia del Vendedor
 - Mirar el poder financiero, estructura de la organización, procedimientos de soporte al cliente, soporte de entrenamiento y consultoría, alianzas con otras compañías
- Desempeño de la Base de Datos
 - Ninguna marca puede probar que un DBMS es más rápido para todas las aplicaciones
 - El desempeño depende de la aplicación
 - Los prototipos específicos de aplicación son muy importantes
- Capacidades de la Base de Datos
 - Se debe evaluar administración de transacciones, control de concurrencia, respaldo y recuperación, seguridad y soporte de un lenguaje de consulta (SQL)

Criterios de Evaluación de un DBMS (cont.)

- Arquitectura de la Base de Datos
 - Evaluar esquemas de control de concurrencia, mecanismos de seguridad y administradores de almacenamiento
- Herramientas de Desarrollo
 - Ver las herramientas para el diseño de la base de datos, modificación del esquema de las base de datos y navegación, así como también las herramientas de depuración y afinación de la base de datos
- Soporte al lenguaje de elección
 - Asegurarse de que hay soporte para el lenguaje de su elección para el desarrollo del sistema
- Facilidad de Migración
 - ¿Qué tan fácil/difícil es migrar al sistema de la base de datos?

Criterios de Evaluación de un DBMS (cont.)

- Integración con Sistemas Legados
 - ¿Qué tan fácil/difícil es integrarse al sistema de administración de base de datos existente?
- Soporte Multi-usuario
 - Evaluación del soporte para desarrollo multiusuario, administración de configuración, control de versiones y estrategias de seguridad

Nota: Invierta tiempo y esfuerzo para seleccionar el sistema de administración de base de datos apropiado para el proyecto

SIEMPRE es más caro corregir que hacerlo bien desde el principio!!!

Productos de la Base de Datos Relacional

- Hay dos factores principales que deben tomarse en cuenta cuando se diseña un sistema OO usando bases de datos relacionales
- Primero, existe una diferencia semántica natural entre el modelo basado en clases un diseño orientado a objetos y el modelo basado en tablas de una base de datos relacional
 - Se debe definir un mapeo o traducción entre los dos
- Segundo, se debe definir el comportamiento que hace interfaz con el RDBMS y que implementa esta traducción
 - ¿Debe insertarse este comportamiento en objetos persistentes o de algún modo debe mantenerse separado?

Mapeo a Bases de Datos Relacionales

- Típicamente, cada clase mapea a una tabla y cada instancia mapea a un renglón

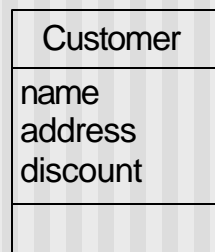
Customer
name
address
discount

Tabla Customer

customerID	name	address	discount
-------------------	-------------	----------------	-----------------

Mapeo a Bases de Datos Relacionales (cont.)

- Las relaciones de uno-a-muchos se implementan usando una llave foránea en la tabla que representa a la clase que tiene la multiplicidad mayor a uno en la relación



1

0..*

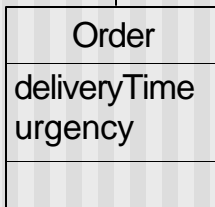


Tabla Customer

customerID	name	address	discount
------------	------	---------	----------

Tabla Order

orderID	delivery	urgency	customerID
---------	----------	---------	------------

Mapeo a Bases de Datos Relacionales (cont.)

- Se crean tablas para resolver las relaciones de muchos-a-muchos

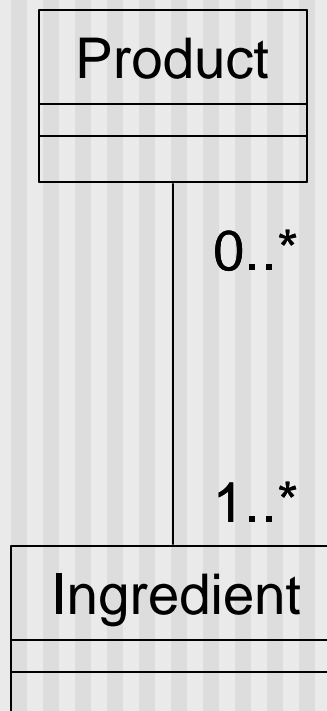


Tabla Product Ingredient

productID	ingredientID
------------------	---------------------

Mapeo a Bases de Datos Relacionales (cont.)

- Las superclases / subclases también pueden mapearse a tablas
 - Cada clase y subclase es una tabla
 - Se proporcionan vistas para la jerarquía

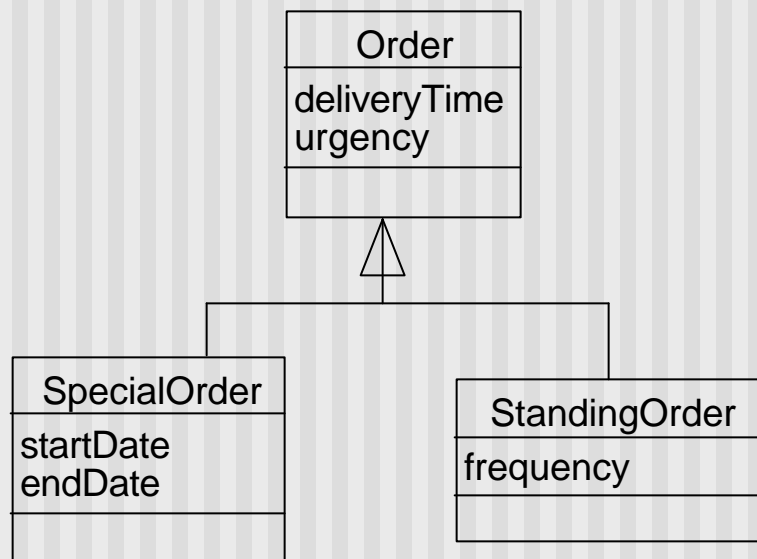


Tabla Order

orderID	deliveryTime	urgency
---------	--------------	---------

Tabla SpecialOrder

orderID	endDate	startDate
---------	---------	-----------

Mapeo a Bases de Datos Relacionales (cont.)

- Existen estrategias alternas para superclases y subclases
 - Repetir todos los atributos en la tabla de superclase
 - Problema: espacio desperdiciado
 - Repetir todos los atributos en la tabla de subclase
 - Problema: redundancia
- Se buscan la mejor relación con respecto al desempeño y espacio de almacenamiento para decidir que mapeo usar en cada situación de herencia

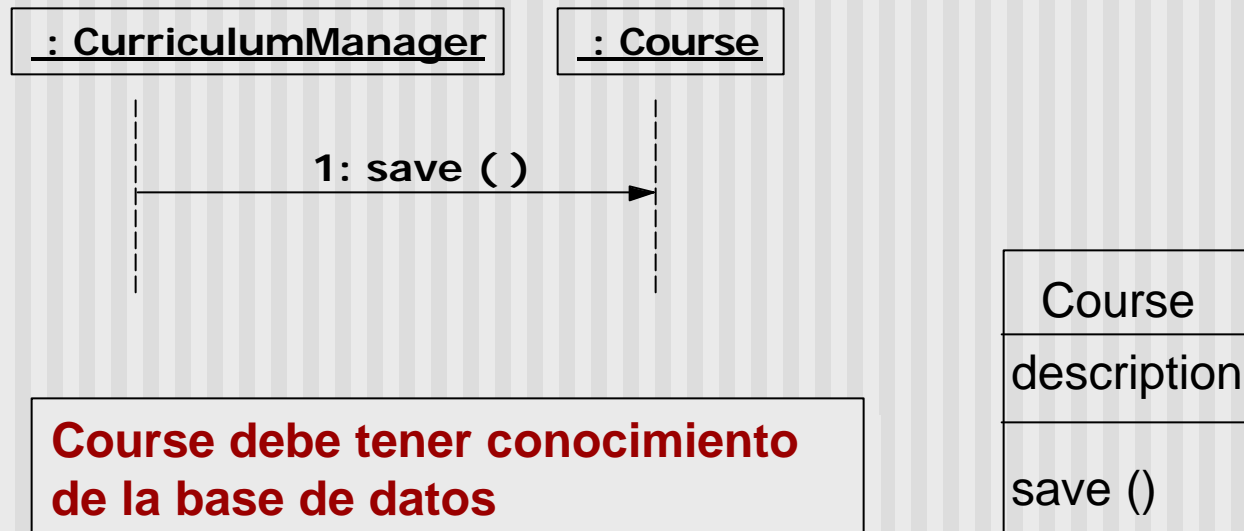
Interfaz con el RDBMS

- El elemento principal asociado con la creación de una interfaz con un RDBMS, es si se separa o no el comportamiento específico de la aplicación del comportamiento específico de la base de datos
- Suponga que nuestro sistema tiene una clase Cliente que se ha decidido que será persistente
 - ¿Deberá la clase Cliente contener los detalles del mapeo OO-a-RDBMS?
 - ¿Deberá la clase Cliente contener el comportamiento para hacer interfaz con el agente RDBMS (es decir, código que genera SQL para leer/escribir de/a la base de datos)?
 - ¿Deberá la clase Cliente incluso saber que es persistente?
- De cualquier forma se puede trabajar - cada uno tiene sus ventajas y desventajas

Interfaz con RDBMS (cont.)

- El comportamiento específico de la base de datos no está separado del comportamiento específico de la aplicación:
 - Cada clase persistente puede tener las funcionalidades de crear, leer, actualizar y borrar (CRUD) construidas en (operaciones que desempeñan mapeo OO-a-RDBMS y generan SQL para implementarlo)
 - Ventajas
 - No es técnicamente retador para implementar
 - Desventajas
 - Los modelos OO y RDBMS no son separables
 - La funcionalidad CRUD no siempre es limpiamente heredable

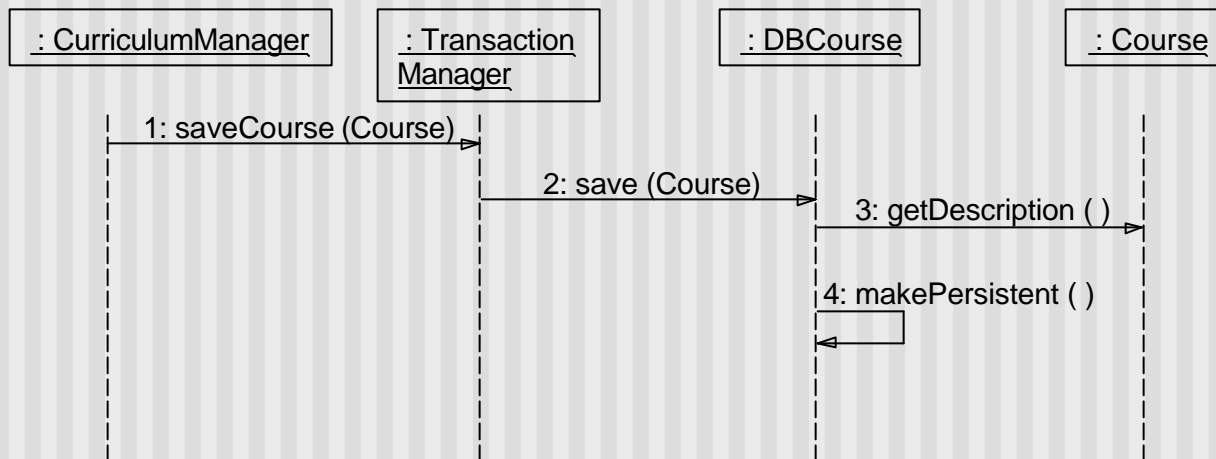
Comportamiento de la Base de Datos dentro de la Clase



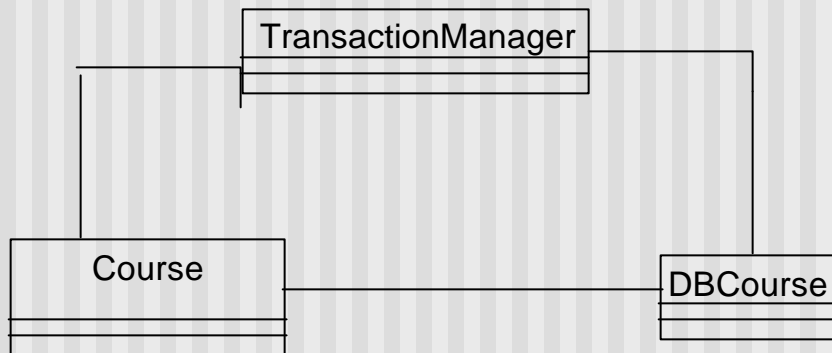
Interfaz con RDBMS

- El comportamiento específico de la base de datos se separa del comportamiento específico de la aplicación
 - Se puede modelar al sistema en dos particiones: Aplicación e Interfaz de la Base de Datos
 - Para cada clase persistente, una interfaz de base de datos asociada se define, la cual entiende el mapeo OO-a-RDBMS y tiene el comportamiento para hacer interfaz con el RDBMS
- Ventajas
 - El modelo OO es separable del modelo RDBMS
 - Existen herramientas disponibles para generar las clases de interfaz básicas a la base de datos
- Desventajas
 - Más técnicamente retador de implementar

Comportamiento de Base de Datos Separado



**TransactionManager
separa lo lógico (Course)
de lo físico (DBCourse)**



DBMS Orientado a Objetos

- Los ODBMS permiten el almacenamiento y recuperación de los objetos (con datos complejos encapsulados en cada objeto)
- Un ODBMS retiene típicamente
 - Objetos (valores de atributos)
 - Información de la clase sobre cada objeto
- No hay diferencia semántica entre el modelo OO y el modelo ODBMS - son idénticos
- No debe diseñarse comportamiento especial para hacer interfaz con el ODBMS

DBMS Orientado a Objetos (cont.)

■ Ventajas:

- Interfaz sin parches entre la aplicación y la base de datos
- Se necesita relativamente poco código para hacer a los objetos persistentes
- Muy efectivo con sistemas que deben atravesar estructuras de datos complicadas

■ Desventajas:

- Riesgo más alto de desarrollo ya que la tecnología y producto de ODBMS no son tan maduros como sus contrapartes de RDBMS
- El desempeño con estructuras simples de datos no proporcionan ventajas sobre RDBMS

- Se debe evaluar la inversión de tecnología relacional existente cuando se evalúe la tecnología de bases de datos OO

Detección de Errores

- Debe establecerse un mecanismo consistente para la detección de errores
- Los objetos deben detectar errores que podrían violar su integridad - esto incluye errores
 - Que surgen con la operación
 - Que resultan de parámetros recibidos de objetos clientes
 - Que resulten de valores de retorno proporcionados por objetos proveedores
- Se puede establecer un plan para monitorear la salud del sistema
 - Se define operación de prueba para cada clase que verifique la integridad o estructura interna, y
 - Monitoreo de objetos definidos para revisar periódicamente cada función de prueba de objeto

Manejo de Errores

- Aún en los sistemas OO, el manejo de errores debe diseñarse cuidadosamente -- en más del 30% del código final existen condiciones de manejo de error
- Los lenguajes OO (como 3.0 C++) proporcionan elementos de manejo de excepción para auxiliar en este diseño
- El manejo de excepciones permite que un error se maneje por un objeto distinto que el objeto que detectó el error
- Esto es con frecuencia apropiado, ya que el objeto detector no siempre conoce el impacto a un sistema por un error

Ejemplo de Manejo de Excepción

```
class String {
public:
    class RangeError { int badIndex; }; // error type
    char getChar(int index) const;
    // ...
};
char String::getChar(int index) const {
    if (index < 0 || index > lastValid)
        throw RangeError(index); // throw point
    return contents[index];
};
void foo() {

    try {
        String s = "hello";
        char c = s.getChar(0);
    }
    catch (const String::RangeError& why) { // catch point
        cout << "subscript out of range" << why.badIndex << endl;
    }
}
```


Throwing y Catching Excepciones

- Cuando hay un problema que no puede ser manejado en el contexto actual, se puede crear y “lanzar” una excepción
 - `throw Problem("things are bad");`
- ¿Qué hace `throw`?
 - Se crea y “regresa” al objeto excepción
- ¿A dónde va el objeto excepción?
 - Cuando se lanza una excepción se busca la llamada a la pila para el primer manejador (handler)
 - Habrá un manejador de excepción para cada tipo de excepción que pueda cargarse
 - `catch (Problem&) { // handle exceptions of type Problem }`

Elementos de Manejo de Excepción

- En el proceso de búsqueda de llamada a la pila, se llaman a los destructores de objetos locales
 - Variables automáticas
 - Parámetros de valor
 - Temporales
- No se llama a los destructores para
 - Variables dinámicas
 - Variables estáticas
- Nunca se ejecuta el código después del punto de lanzamiento
- Las excepciones se apoderan de la administración de recursos (ej. Cerrar archivos abiertos)

¿Deben Usarse Siempre las Excepciones?

- No deben usarse las excepciones en las siguientes situaciones:
 - Condiciones ordinarias de error
 - Si hay suficiente información disponible para manejar el error entonces NO hay una excepción
 - Para controlar el flujo del problema
 - Una excepción NO es un regreso alternativo

¿Cuándo Deben Usarse las Excepciones?

- Las Excepciones se lanzan como resultado de un error serio
 - No hay regreso al punto donde se lanzó la excepción
- Las Excepciones no deben usarse si el error puede manejarse (arreglarse) y el proceso continua
 - Puede llamarse una operación para “arreglar” el problema y el proceso puede continuar

Uso Típico de Excepciones

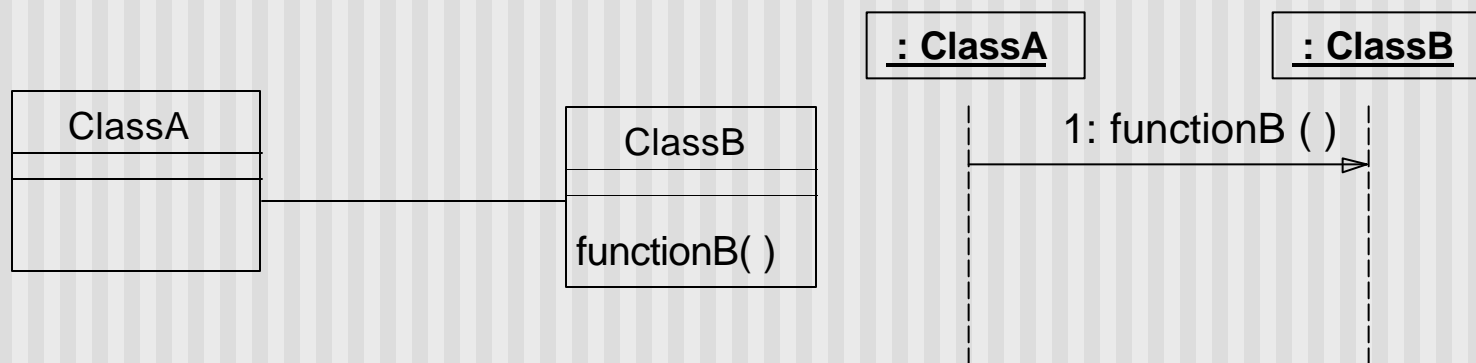
- Arreglar el problema y continuar el proceso sin procesar de nuevo la función que lanzó la excepción
- Calcular un resultado alternativo
- Lanzar el error a un contexto más alto
- Terminar el programa
- Ocultar funciones que usan esquemas de error ordinario
- Simplificar el código
- Hacer al código más fácil de mantener

Reporte de Error

- El almacenamiento del registro de errores de forma correcta y el reporte en línea son las claves de la mayoría de los sistemas
- El comportamiento del control de errores consistente puede implementarse en base a la clase Error usada en el manejo de excepción
- Este comportamiento puede incluir
 - Agregar el error a un registro de errores que abarque al sistema (system-wide error log)
 - Distribuir los errores del proceso los cuales facilitan el monitoreo en línea de errores
- Este tipo de acercamiento asegura la consistencia al separar la responsabilidad detallada de las clases de aplicación

Comunicación Interprocesos

- ClassA llama a la operación functionB() de la ClassB
- ¿Qué sucede cuando las clases ClassA y ClassB están en proceso diferentes?
- Esto se convierte en un resultado crítico en sistemas distribuidos
- Se necesita un mecanismo estándar para la comunicación interprocesos



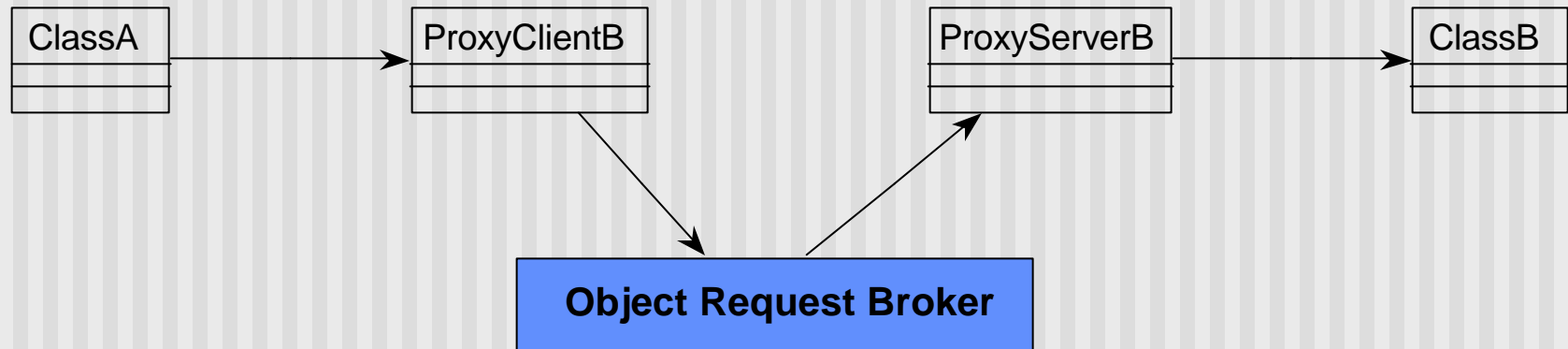
Comunicación Interprocesos (cont.)

- Una solución común que soporta el paradigma OO se ha desarrollado
 - Clases Proxy se insertan en cada proceso el cual proporciona la interfaz de las clases originales y encapsula la comunicación de nivel menor
- La distribución es transparente a clases de aplicación

Estándares Distribuidos de OO

- La elección de un estándar de distribución es un elemento de diseño (si su sistema usa objetos distribuidos)
- Hay dos estándares para la distribución OO
 - Common Object Request Broker Architecture (CORBA)
 - Component Object Model (COM+/DCOM/COM/OLE)
- Object Request Brokers (ORB) proporciona acceso transparente a objetos en un ambiente distribuido
 - ORB permite la conectividad de ubicación independiente cliente/servidor
 - Las decisiones de distribución pueden tomarse al tiempo de corrida

Classes Proxy



Planeación para Reuso

- Los componentes reusables deben considerarse previamente en el proceso de diseño para incorporarlo al sistema
- La evaluación de librerías de software comerciales e internas para aplicarlos al sistema por módulos
- Las librerías de clases son grupos de clases que colaboran para proporcionar algún servicio, interfaz o función
- Librerías de clases están comúnmente disponibles como:
 - Container objects
 - Interfaces to databases
 - User interface widgets

Actualización de Diagramas

- Los diagramas de clases se actualizan para mostrar los mecanismos claves elegidos
- Los diagramas de secuencias se actualizan para mostrar las interacciones entre clases descubiertas y clases que representan estrategias de mecanismos clave

Diagrama de Clases Actualizado

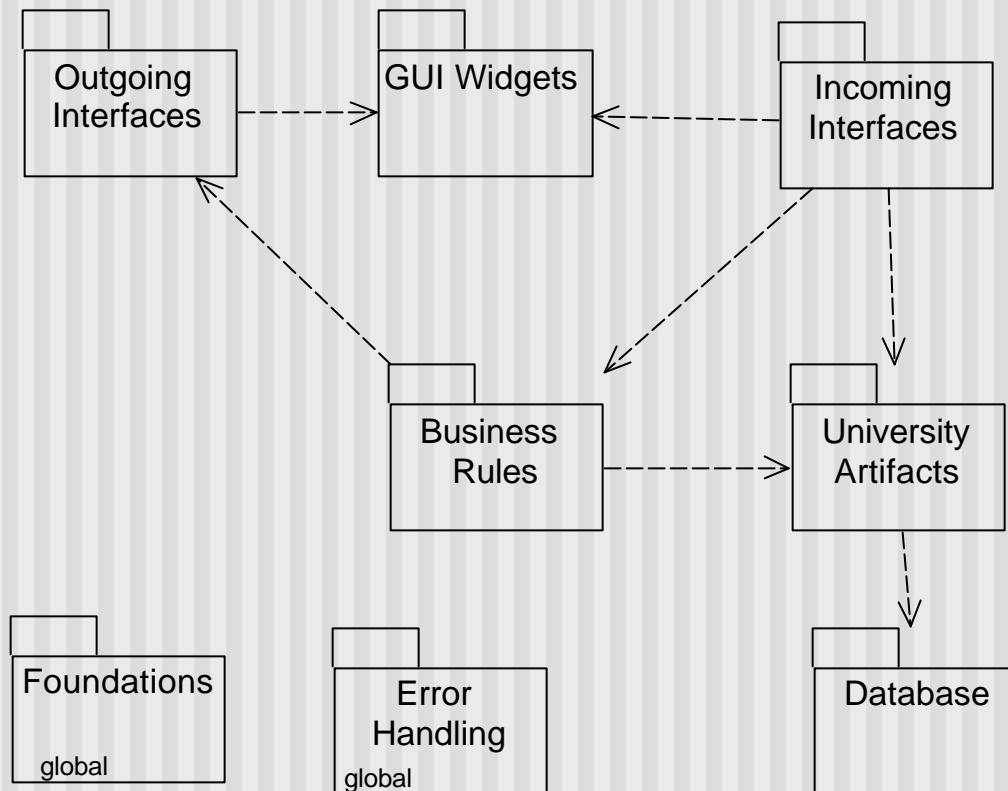
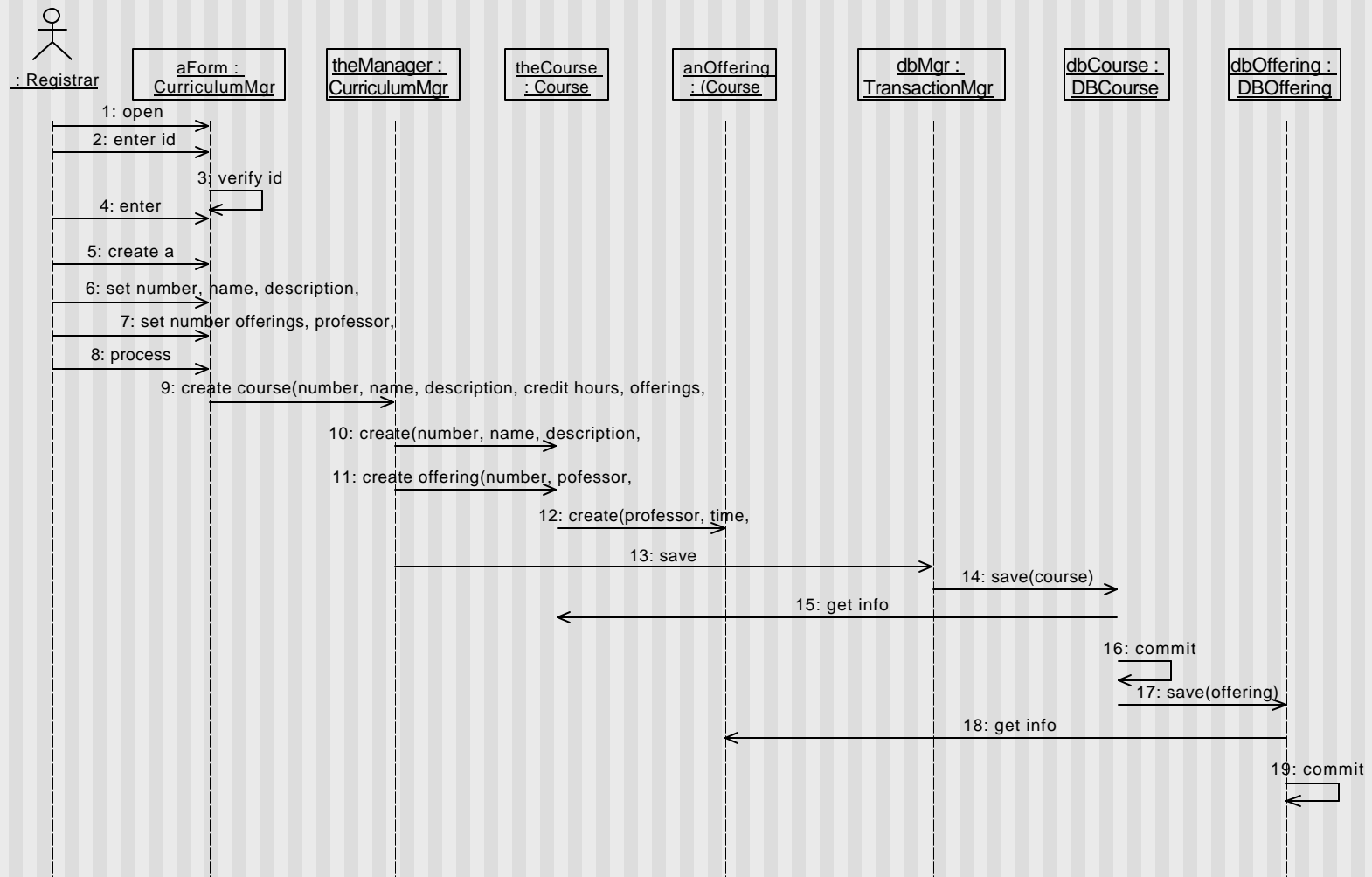


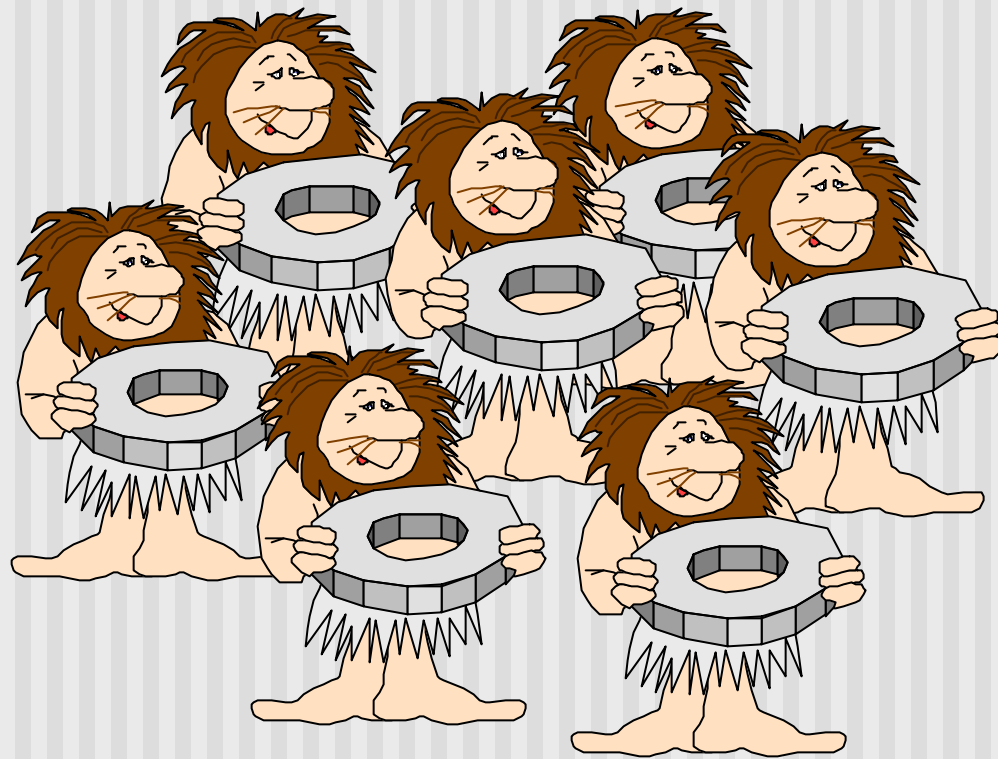
Diagrama de Secuencias Actualizado



Ejercicio: Mecanismos Clave

- Discuta las estrategias de mecanismos clave para el problema
- Actualice el diagrama de clases para mostrar la incorporación de mecanismos clave
- Actualice los diagramas que sean necesarios

Diseño de Clases



Objetivos: Diseñar Clases

- Usted podrá:
 - Discutir decisiones de diseño de interfaz de usuario
 - Agregar clases de nivel de diseño para solucionar problemas de diseño
 - Usar patrones de diseño para resolver problemas de diseño

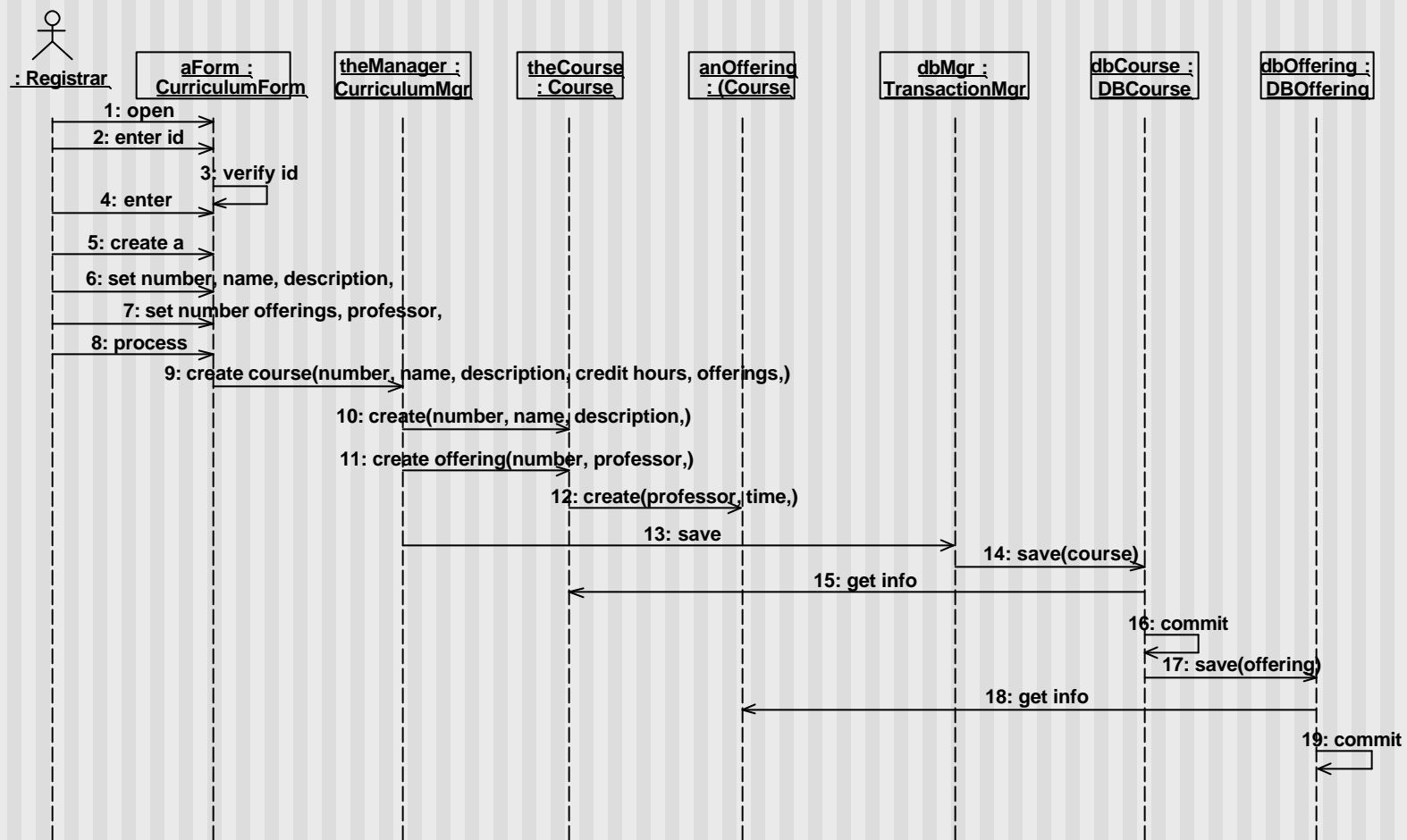
Diseño de Interfaz de Usuario

- Las clases boundary manejan las comunicaciones de la interfaz entre el usuario y el sistema
 - Proporcionan capacidad para enviar y recibir información del exterior del sistema
- Durante el análisis, se definen clases boundary de alto nivel
- Durante el diseño, se completa el diseño de la interfaz de usuario
 - Windows layouts
 - Número de ventanas
 - Manejo de eventos iniciados por los usuarios

Descubriendo de Requerimientos de Interfaz

- Cualquier prototipo de la interfaz de usuario hecho con anterioridad es un buen punto de partida para esta fase de diseño
- Los diagramas de secuencia y/o colaboración también proporcionan una buena fuente para los requerimientos de la interfaz
 - Algo en el sistema debe proporcionar la capacidad para recibir “mensajes” provenientes de un actor
 - Algo en el sistema debe proporcionar la capacidad de enviar todos los “mensajes” dirigidos a un actor

Diagrama de Secuencias para el Escenario de "Crear Cursos"

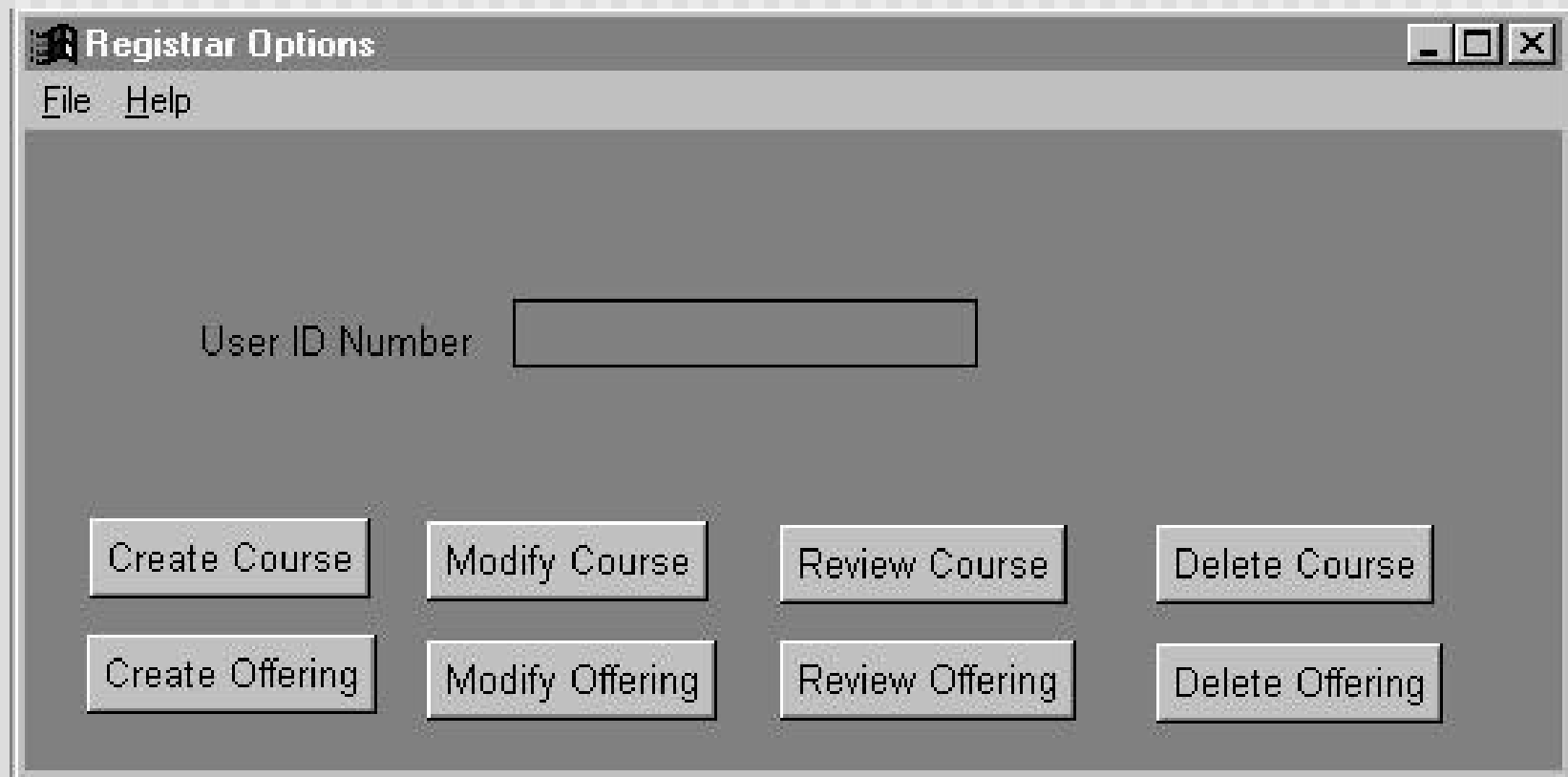


Requerimientos de la Interfaz de Usuario

- Actor "Registrar"
 - Introduce la información necesaria para crear un Course y sus ofertas
- Requerimientos de otros escenarios
 - El actor tiene la habilidad de crear, consultar, modificar y borrar Course, así como también ofertas de Course
- Decisiones de diseño
 - Una ventana que contenga todas las opciones disponibles para el actor "Registrar"
 - Una ventana que contenga la información de Course
 - Una ventana que contenga información de ofertas de Course
 - Botones disponibles en las ventanas de Course y de ofertas de Course que permitan guardar, cancelar o borrar la información

Ventana Principal del Actor

Registrar



The image shows a software window titled "Registrar Options". It has a menu bar with "File" and "Help". Below the menu bar is a large gray area. In the center of this area is the text "User ID Number" followed by a rectangular input field. At the bottom of the window, there are eight buttons arranged in two rows of four. The top row contains "Create Course", "Modify Course", "Review Course", and "Delete Course". The bottom row contains "Create Offering", "Modify Offering", "Review Offering", and "Delete Offering".

Registrar Options

File Help

User ID Number

Create Course Modify Course Review Course Delete Course

Create Offering Modify Offering Review Offering Delete Offering

Ventana de Course

The image shows a graphical user interface window titled "Course Information". The window has a standard Mac OS-style title bar with a menu icon, the title text, and minimize, maximize, and close buttons. Below the title bar is a menu bar with "File" and "Help" menus. The main content area contains four input fields: "Course Name" (a single-line text box), "Course Description" (a multi-line text area), "Course Number" (a single-line text box), and "Credit Hours" (a single-line text box). At the bottom of the window, there are four buttons: "Save", "Delete", "Course Offering", and "Cancel".

Course Name	<input type="text"/>		
Course Description	<input type="text"/>		
Course Number	<input type="text"/>	Credit Hours	<input type="text"/>
<input type="button" value="Save"/>	<input type="button" value="Delete"/>	<input type="button" value="Course Offering"/>	<input type="button" value="Cancel"/>

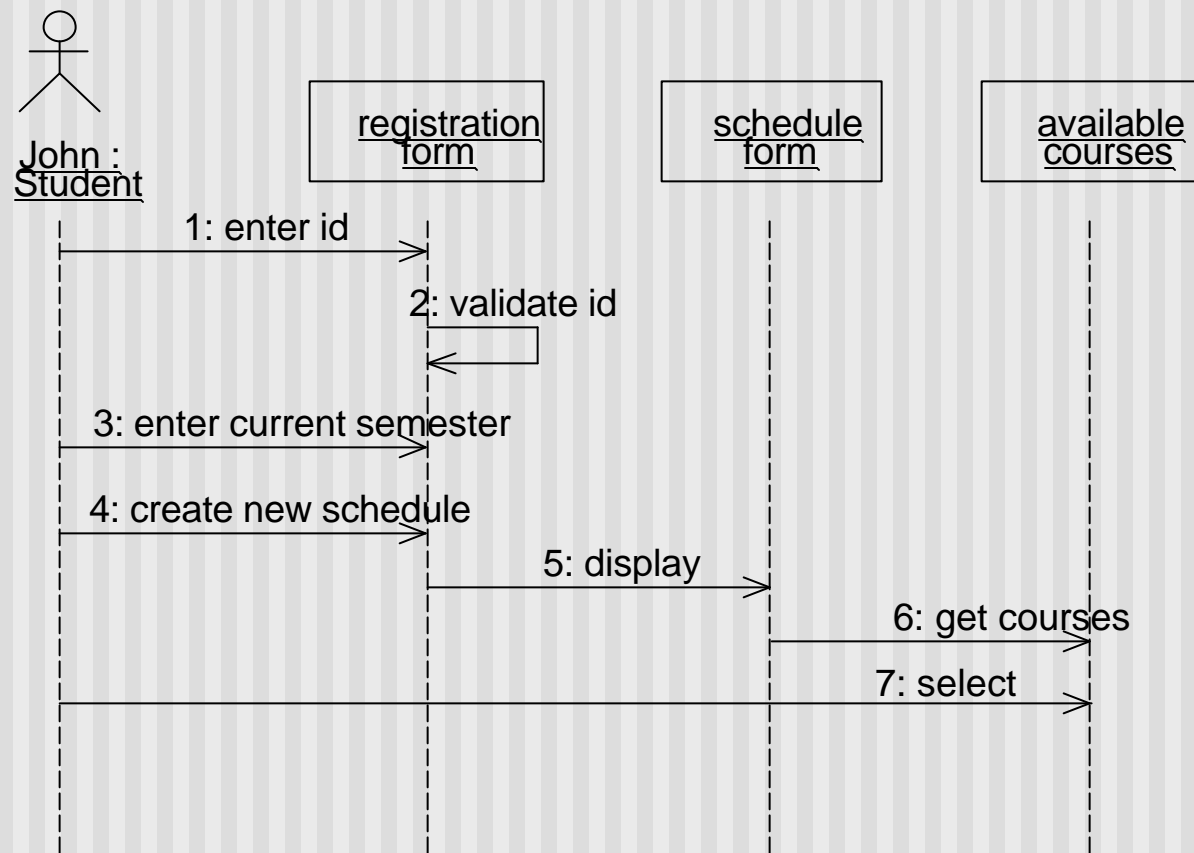
Ventana de Ofertas de Course

The image shows a graphical user interface window titled "Course Offering Information". The window has a standard menu bar with "File" and "Help" options. Below the menu bar, there are three input fields arranged vertically. The first field is labeled "Course Number" and is a small rectangular box. The second field is labeled "Professor" and is a longer rectangular box. The third field is labeled "Time and Location" and is the largest rectangular box. At the bottom of the window, there are four buttons: "Save", "Delete", "Next Offering", and "Cancel", each in its own rectangular button box.

Field Label	Input Field
Course Number	<input type="text"/>
Professor	<input type="text"/>
Time and Location	<input type="text"/>

Buttons: Save, Delete, Next Offering, Cancel

Inscripción a Course



Requerimientos para este Escenario

- Actor "Student"
 - Debe proporcionar cursos seleccionados para el semestre actual
 - La lista de cursos disponibles se despliega al actor
- Decisiones de diseño
 - Se crea una ventana que contenga listas de selección de cursos
 - Las listas contienen nombres de todos los cursos ofrecidos

Ventana Course Registration

Course Registration for Matt Jones

File Help

Primary Course Selections	Secondary Course Selections
English 101	Algebra
History 101	Art History
Spanish	
Geometry	

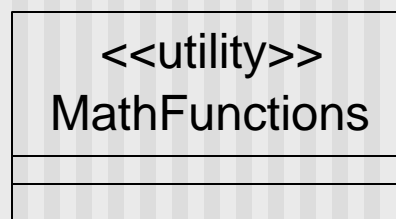
Cancel OK

¿Qué sucede cuándo se presiona el botón OK?

- Todos los constructores de la interfaz de usuario son diferentes
 - Algunos crean objetos que contienen información de la ventana
 - Otros crean estructuras de datos con información
- Algunas técnicas comunes
 - 1. Las clases Control reciben los datos de una ventana y procesan los datos
 - Los datos de la ventana pasan de la ventana a la clase Control,
 - 2. El botón sabe que hacer con los datos en la ventana

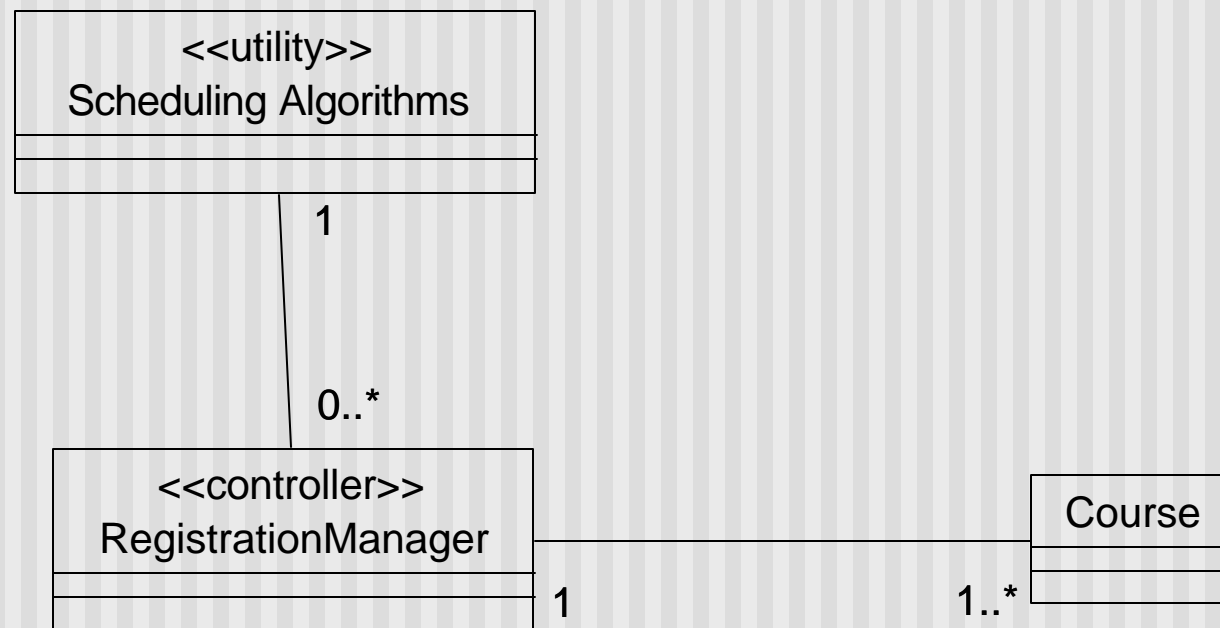
Clases **Utility**

- El estereotipo <<utility>> se usa para una clase que contiene una colección de subprogramas libres
 - Los subprogramas libres son funciones no-miembro, por ejemplo, funciones que no pertenecen a una clase en particular
- Las clases Utility se define generalmente;
 - Para proporcionar algunos servicios de algoritmos comunes, los servicios pueden reusarse en una variedad de contextos
 - Para librerías ocultas o aplicaciones no orientadas a objetos



Ejemplo: Clase Utility

- La clase utility **SchedulingAlgorithms** contiene funciones para resolver conflictos de horario



Ejemplo: Clase Utility (cont.)

- Para evitar utilerías múltiples (funciones libres de C++) que conviertan las unidades de distancia, se puede crear una clase utility para empaquetar todas las funciones bajo una interfaz

Usando Funciones Libres

```
extern float  
    inchToCentimeter(float inch);  
  
extern float  
    centimeterToInch  
        (float centimeter);
```

Usando Clases Utility de C++ (preferible)

```
#ifndef UNIT_UTILITIES  
#define UNIT_UTILITIES  
  
class/*_utility*/ Unit_Uilities {  
public:  
    static float inchToCentimeter(float inch);  
    static float centimeterToInch(float centimeter);  
};  
  
#endif // UNIT_UTILITIES
```


Clases Help

- Durante el diseño, una clase puede agregarse como “help”, ya que desempeña alguna funcionalidad necesaria
- Ejemplo:
 - El CurriculumForm debe verificar que el id introducido sea válido
 - Si la verificación se identifica con el formato del id, entonces la ventana puede desempeñar esta funcionalidad
 - Si la verificación se identifica con seguridad, entonces se necesita información adicional
 - **Lista de id válidos**
 - **Esta clase se agrega al sistema**

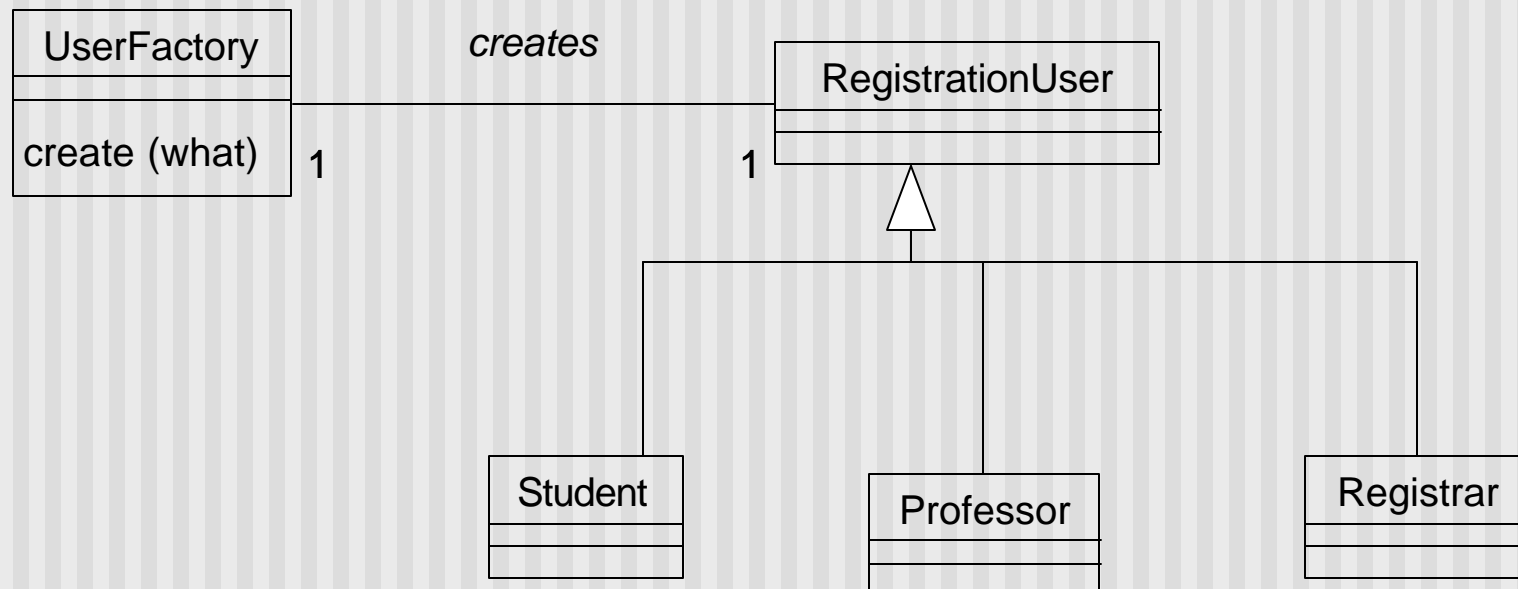
Aparición de Patrones

- Un patrón de diseño es una solución a un problema de diseño común
- Un Patrón
 - Describe un problema común de diseño
 - Describe la solución al problema
 - Discute los resultados y trade-offs de la aplicación del patrón
- Los patrones se están colectando, catalogando y usando para construir sistemas
 - Proporciona la capacidad de reusar diseño y arquitecturas exitosas
 - Da como resultado sistemas más fáciles de mantener
 - Incrementa productividad

Adaptación de Patrones

- Los sistemas de inscripción a curso tienen tres tipos de usuarios: alumnos, profesores y el administrador
- Se creó una jerarquía de RegistrationUser para los diferentes tipos de usuarios
- El tipo de usuario que se va a acrear depende de los datos introducidos
- Problema: quien crea el tipo específico de usuario
 - El patron Factory Method puede usarse para crear el tipo correcto de usuario al momento de ejecución
 - Se le da el dato al factory y se le pide que cree el tipo correcto de objeto

El Patrón Factory



La creación (what) de la operación UserFactory crea el tipo correcto de RegistrationUser

Otros Patrones

- Prototype: crea un objeto copiando un objeto prototipo
- Singleton: asegura que una clase tiene sólo una instancia y proporciona un punto global de acceso a la misma
- Adapter: convierte la interfaz de una clase a otra interfaz
- Iterator: proporciona una manera de acceder los elementos de un objeto agregación
- Memento: captura y externaliza el estado interno de un objeto de modo que el objeto pueda restablecer este estado después (esto se hace sin romper la encapsulación)

¿Cuántas Clases se necesitan?

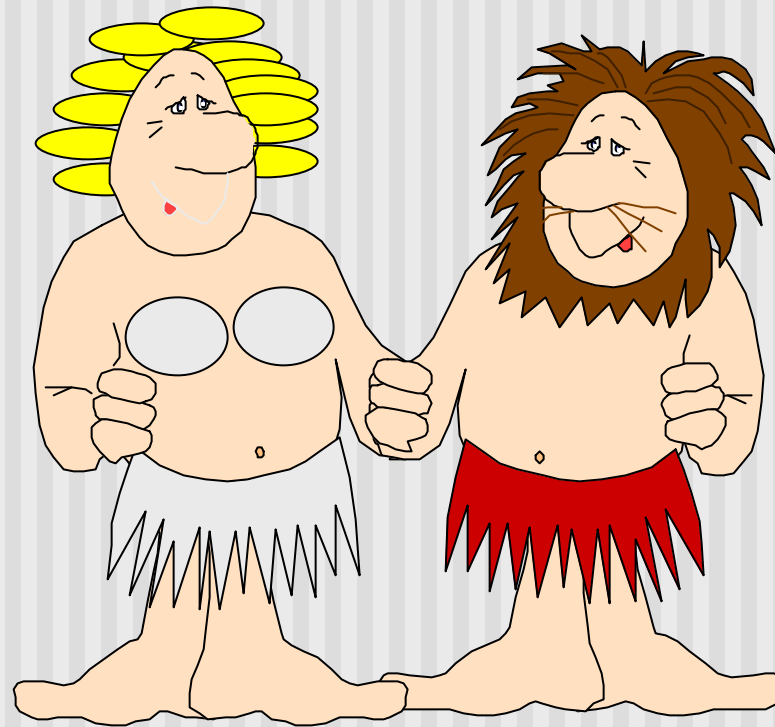
- Varias..., clases simples quiere decir que cada clase
 - Encapsula menos de la inteligencia de todo el sistema
 - Es más reusable
 - Es más fácil de implementar
- Pocas..., clases complejas quiere decir que cada clase
 - Encapsula una gran porción de la inteligencia de todo el sistema
 - Es menos propensa al reuso
 - Es más difícil de implementar

Regla: Una clase debe tener un único propósito y bien enfocado
Una clase debe hacer una cosa y hacerla bien

Ejercicio: Diseño de Clases

- Discuta que clases adicionales pueden agregarse al modelo para facilitar las decisiones de diseño
- Actualizar los diagramas, tanto como sea necesario

Diseño de Relaciones

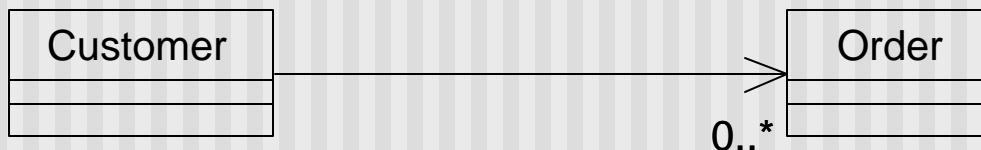


Objetivos: Diseño de Relaciones

- Usted podrá:
 - Determinar la navegación de las relaciones
 - Mejorar las relaciones de asociación y agregación
 - Discutir opciones de visibilidad de objetos
 - Discutir múltiples decisiones de diseño
 - Diseñar clases de asociación

Navegación

- En el análisis, las asociaciones son bi-direccionales
- En el diseño, una asociación puede ser uni-direccional
 - Una flecha se agrega a la asociación para mostrar que la navegación solo va en una dirección



Customer puede
“hablar” con Order

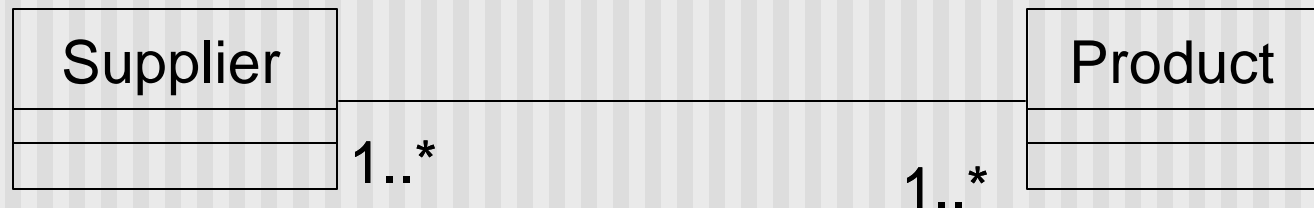
Order no puede
“hablar” con Customer

Decisiones de Navegación

- Durante el diseño nos fijamos si en realidad es necesario navegar en ambas direcciones
- La necesidad de navegación se revela en los casos de uso y en los escenarios
 - Dada una instancia de la clase A, ¿necesitamos encontrar todas las instancias asociadas de la clase B?
 - Dada una instancia de la clase B, ¿necesitamos encontrar todas las instancias asociadas de la clase A?

Preguntas de Navegación

- El sistema debe contestar preguntas como:
 - ¿De qué proveedor puedo comprar este producto? (producto - proveedor)
 - ¿Qué productos fueron ordenados para este proveedor en particular? (proveedor - producto)



La Navegación **Two-Way** vs. la Navegación **One-Way**

- Las asociaciones two-way son más difíciles y costosas de implementar que las asociaciones one-way
- Aunque se requiera la navegación two-way, una asociación one-way puede ser suficiente bajo ciertas circunstancias.
Por ejemplo:
 - La navegación en una de las direcciones es poco frecuente y no tiene exigencias estrictas de desempeño, o
 - El número de instancias de una de las clases es muy pequeño

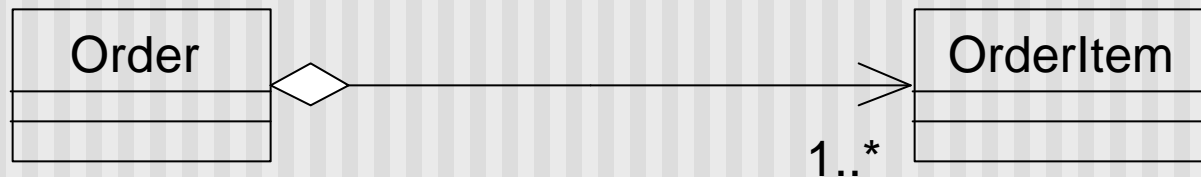
Ejemplo: Simplificación de una Asociación One-Way

- Situación 1: Frecuentemente necesito preguntar que proveedor(es) proporcionan cierto producto, pero sólo necesito pedir una lista de todos los productos proporcionados por un proveedor trimestralmente para el proceso de facturación
 - Implementar la dirección de Producto-a-Proveedor y buscar todas las instancias de Producto cuando se compile una lista de Producto para cada Proveedor
- Situación 2: Sólo existen dos proveedores
 - Implementar sólo la dirección Producto-a-Proveedor y buscar todos los registros de Producto cuando necesite recorrer la asociación en la dirección opuesta



Navegación para Agregaciones

- Una agregación también puede ser uni-direccional
- Se agrega una flecha a la línea de agregación



Refinando Agregaciones

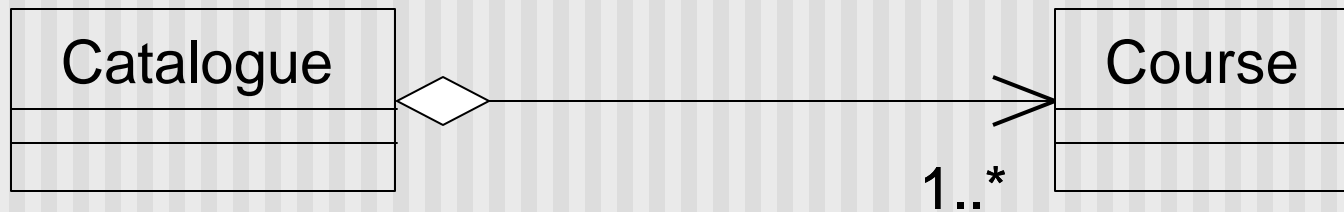
- Una relación de agregación quiere decir que el objeto origen debe contener conocimiento semántico del objeto destino
- Las relaciones pueden ser de dos tipos:
 - Por referencia
 - Por valor

Implicaciones Por-Valor y Por-Referencia

- Por valor implica que los objetos se crean y destruyen como consecuencia de la creación y destrucción de otro objeto
 - En otras palabras, los tiempos de vida de los objetos relacionados son iguales
- Por referencia implica que los tiempos de vida de los objetos relacionados pueden ser independientes
- Por lo tanto, la selección de por valor o por referencia determina como se diseña para que el lenguaje de programación elegido (como C++) cree y borre un objeto

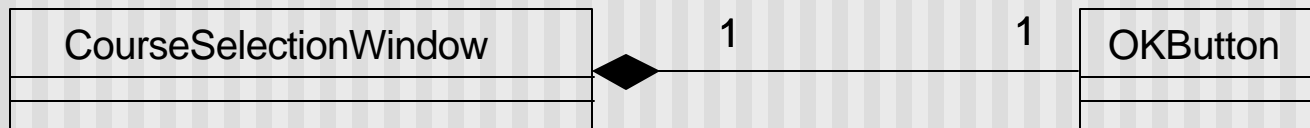
Relaciones Por-Referencia

- Las relaciones por-referencia denotan tiempos de vida independientes
 - Se muestra como un diamante vacío



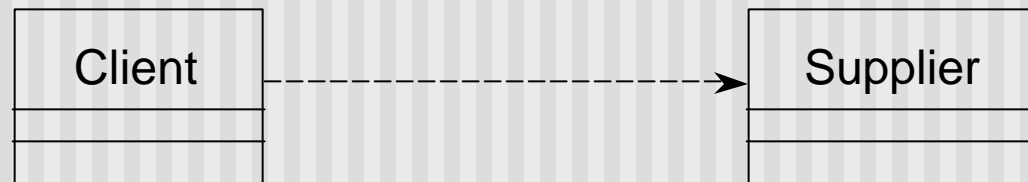
Relaciones Por-Valor

- Las relaciones por-valor indican tiempos de vida dependientes
 - Al crear el objeto se crea también el objeto relacionado
 - Al borrar el objeto se borra también el objeto relacionado
- Por ejemplo, cuando se crea CourseSelectionWindow también se crea OKButton



Relaciones de Dependencia

- Una relación de dependencia quiere decir que una clase depende de algunos servicios de otra clase
- Una relación de dependencia es como una flecha punteada

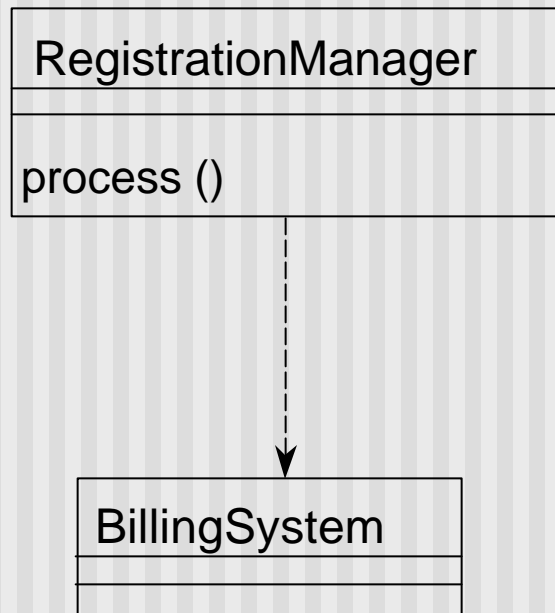


Un objeto Client depende de un objeto Supplier

Relaciones de Dependencia (cont.)

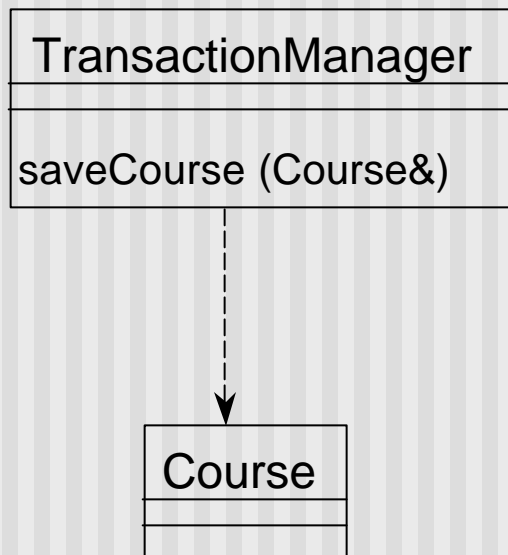
- Una relación de dependencia indica, ya sea que:
 - Las operaciones de la clase cliente crean objetos de la clase proveedor
 - Las operaciones de la clase cliente tienen firmas en donde aparecen clase de retorno o argumentos, por lo que son instancias o referencias a la clase proveedor

Las Operaciones crean Objetos de la Clase Proveedor



```
#include "BillingSystem.h"
void RegistrationManager::process(){
    BillingSystem theInterface;
    ...
}
```

Los Objetos Proveedor como Argumentos de una Operación



```
class Course;
class TransationManager {
    public:
    ...
    void saveCourse(Course&);
    private:
    ...
};
```


Visibilidad de Objeto

- Las relaciones proporcionan una ruta de comunicación entre objetos
- Para que los objetos “hablen” deben ser visibles
 - La visibilidad de objeto determina el diseño de una relación



Opciones de Visibilidad de Objetos

- Hay cuatro opciones de visibilidad
 - **Global**
 - El objeto proveedor es un objeto global
 - **Parámetro**
 - El objeto proveedor es un parámetro de una operación del objeto cliente
 - **Local**
 - El objeto proveedor es declarado localmente
 - **Campo**
 - El objeto proveedor es un campo en el objeto cliente

Modelo de Análisis

- La operación createCourse() de CurriculumController le pide al TransactionManager que guarde el nuevo objeto Course

Diagrama de Clases

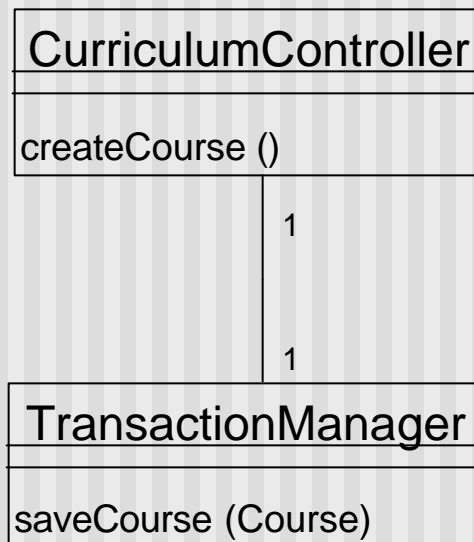
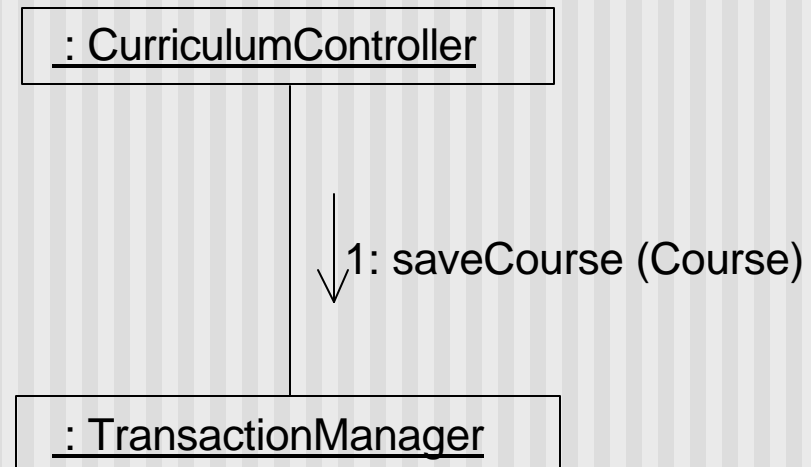
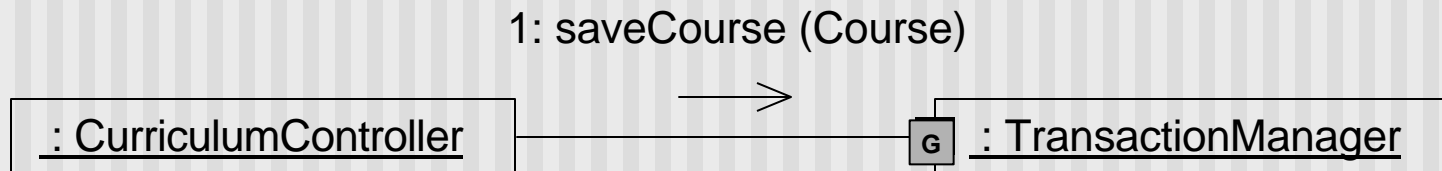


Diagrama de Colaboración

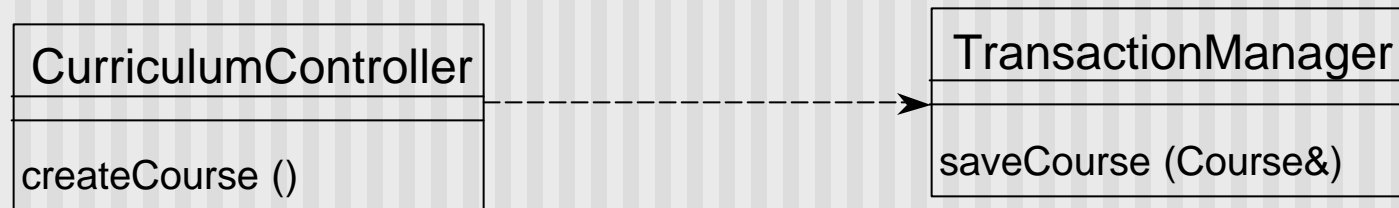


Modelo de Diseño -- Visibilidad Global

- El objeto TransactionManager se declara globalmente



- La visibilidad global se traduce en la relación de dependencia

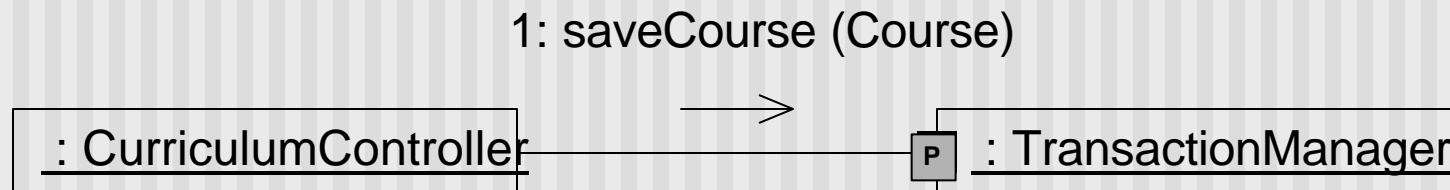


Visibilidad Global

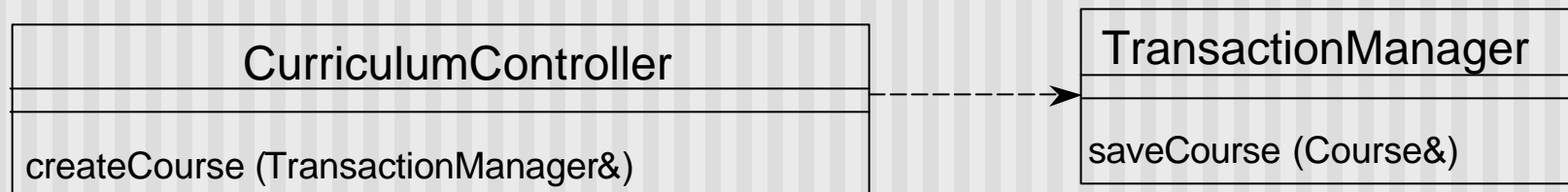
```
static TransactionManager theManager;  
class CurriculumController {  
    public:  
    void createCourse();  
    ...  
};  
class Course;  
void CurriculumController::createCourse() {  
    Course *aNewCourse;  
    ...  
    theManager.saveCourse(aNewCourse);  
    ...  
};
```

Modelo de Diseño -- Visibilidad de Parámetro

- El objeto TransactionManager es un parámetro de la operación createCourse() del CurriculumController
 - El objeto TransactionManager sólo es visible a la operación createCourse



- La visibilidad de parámetro se traduce en relación de dependencia



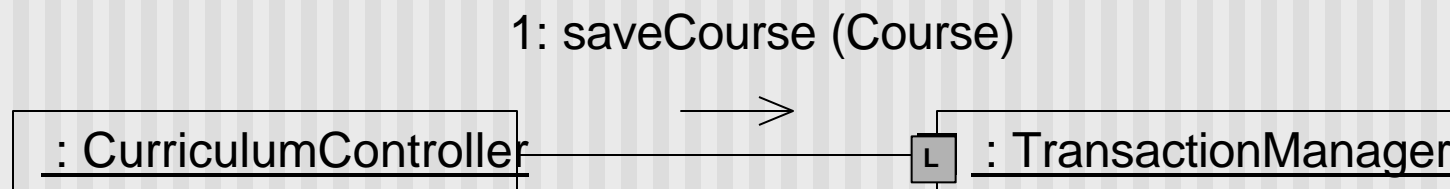
Visibilidad de Parámetro

```
class TransactionManager;
class Course;
class CurriculumController {
    public:
    void createCourse(TransactionManager&);
    ...
};
void CurriculumController::
    createCourse(TransactionManager& theManager) {

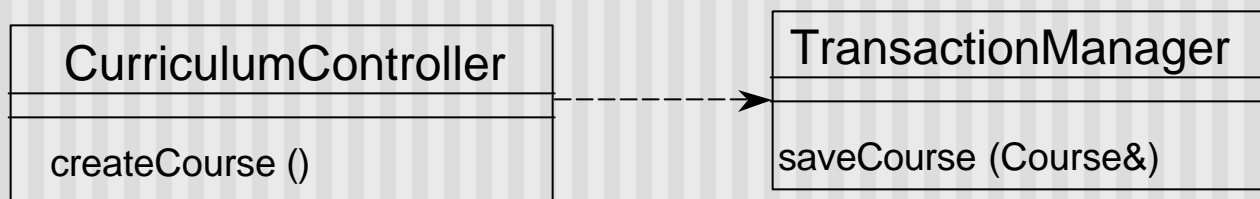
    Course *aNewCourse;
    ...
    theManager.saveCourse(aNewCourse);
    ...
}
```

Modelo de Diseño -- Visibilidad Local

- El objeto TransactionManager se declara dentro de la operación createCourse() del CurriculumController
 - El objeto TransactionManager sólo es visible a la operación createCourse()



- La visibilidad local se traduce a una relación de dependencia



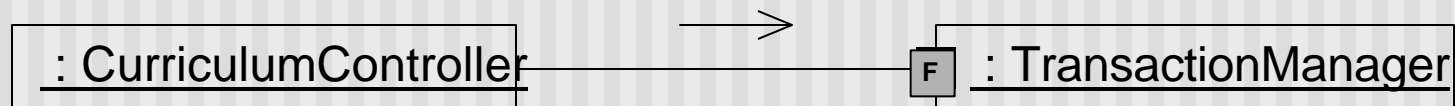
Visibilidad Local

```
#include "TransactionManager.h";
class Course;
class CurriculumController {
    public:
        void createCourse();
        ...
};
void CurriculumController::createCourse() {
    Course *aNewCourse;
    TransactionManager theManager;
    ...
    theManager.saveCourse(aNewCourse);
    ... }
```

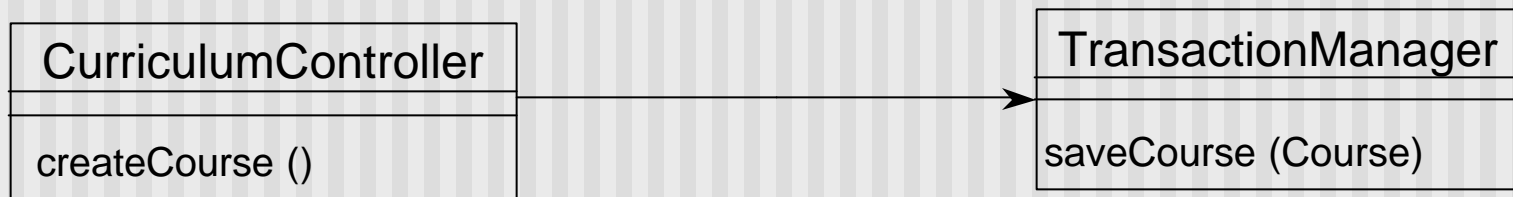
Modelo de Diseño -- Visibilidad de Campo

- El objeto TransactionManager es un dato miembro de la clase CurriculumController
 - El objeto TransactionManager es visible a todas las operaciones de la clase CurriculumController

1: saveCourse (Course)



- La visibilidad de campo se traduce a una relación de asociación (o agregación)



Visibilidad de Campo

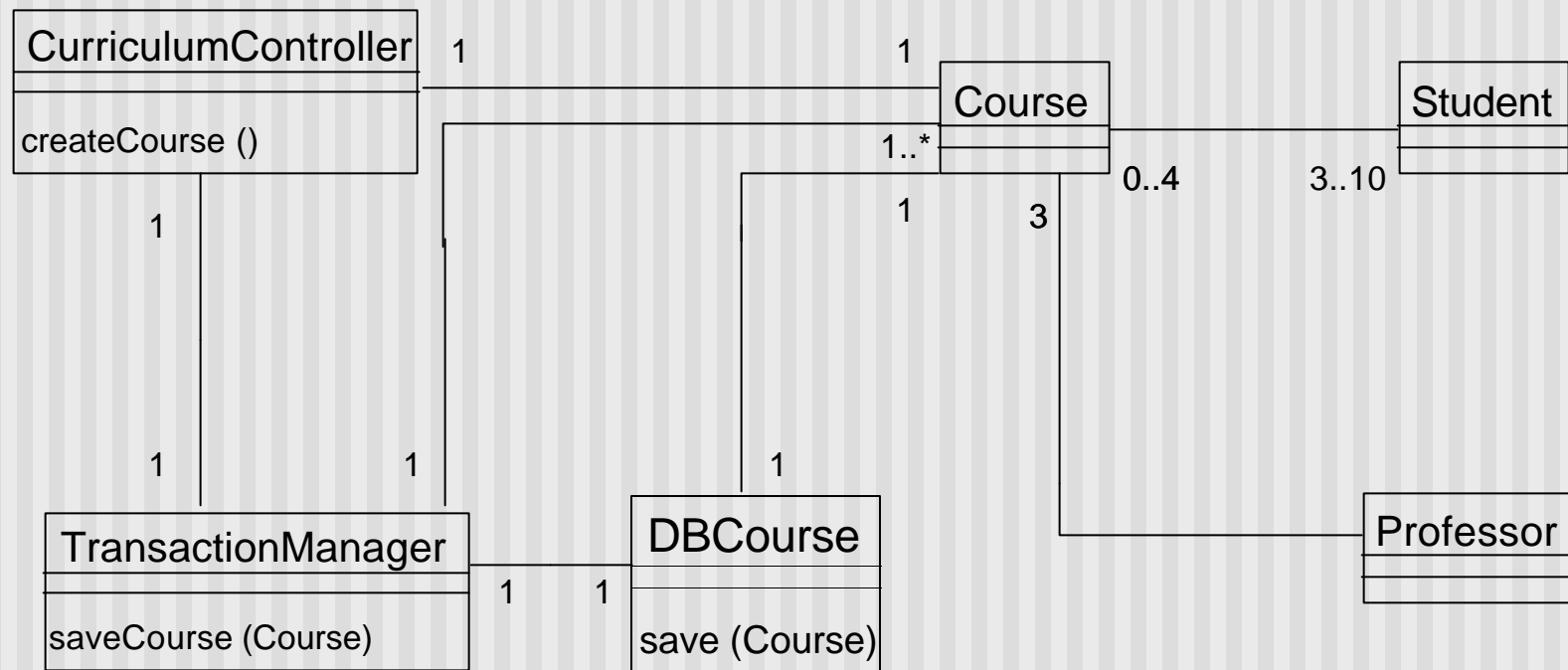
```
#include "TransactionManager.h";
class Course;
class CurriculumController {
    public:
        void createCourse();
        ...
    private:
        TransactionManager theManager;
};
void CurriculumController::createCourse() {

    Course *aNewCourse;
    ...
    theManager.saveCourse(aNewCourse);
    ... }
```

Ejemplo: Diseño de Relación

- El CurriculumController es responsable por la administración de toda la información de Course. Course se crea y se guarda en la base de datos Curriculum. Un TransactionManager es responsable por las interfaz a la base de datos. Hay una clase DBCourse que sabe como guardar la información de Course pertinente. Cada Course puede tener entre 3 y 10 alumnos inscritos y tiene sólo un Professor. Un alumno puede inscribirse a un máximo de 4 cursos. Cada profesor imparte tres cursos. Se ejecuta un reporte listando cursos, el profesor y los alumnos inscritos para las tres primeras semanas del semestre.

Modelo Antes del Diseño de la Relación



Decisiones de Diseño

- El CurriculumController usa al TransactionManager para cada Course que administra
 - La operación createCourse() no es la única operación que usa el TransactionManager
 - Se elige visibilidad de campo
 - El CurriculumController envía mensajes al TransactionManager, pero el TransactionManager no envía ningún mensaje al CurriculumController
 - La relación es uni-direccional (CurriculumController a TransactionManager)

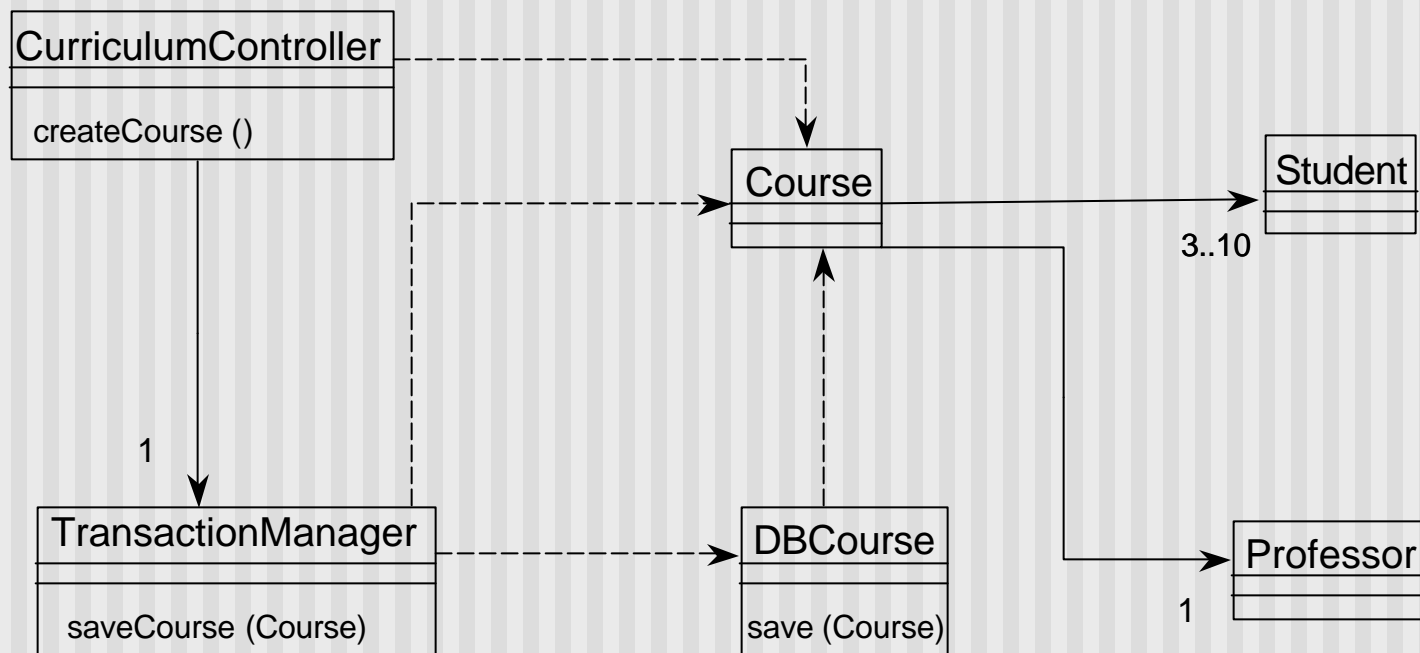
Decisiones de Diseño (cont.)

- El CurriculumController crea un nuevo curso dentro de la operación createCourse()
 - Se elige la visibilidad local
- La operación saveCourse() del TransactionManager se le pasa un objeto Course como parámetro
 - Se elige la visibilidad de parámetro
 - El TransactionManager usa al objeto DBCourse para guardar el objeto Course
 - Esta es la única operación que necesita al objeto Course
 - Se elige la visibilidad local

Decisiones de Diseño (cont.)

- Se le pasa un objeto Course a la operación salvar de DBCourse
 - Se elige visibilidad de parámetro
- Un Course debe saber de sus alumnos para generar el reporte
 - Estos requerimientos no establecen que un alumno debe saber de sus cursos
 - La relación se hace uni-direccional
- Un Course debe saber de su Professor para generar el reporte
 - Estos requerimientos no establecen que el Professor deba saber de sus cursos
 - La relación se hace uni-direccional

Modelo Después del Diseño de Relación

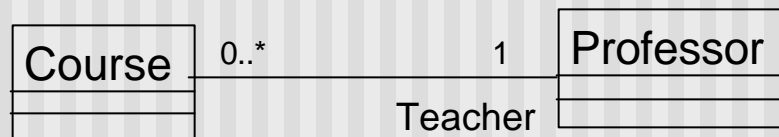


Multiplicidad para Relaciones

- La multiplicidad es el número de instancias que participan en una asociación
- Las estimaciones iniciales de multiplicidad hechas durante el análisis deben actualizarse y refinarse durante el diseño
- Todas las relaciones de asociación y agregación deben tener multiplicidad especificada

Implementación de Asociaciones con Multiplicidad de 1 o 0 a 1

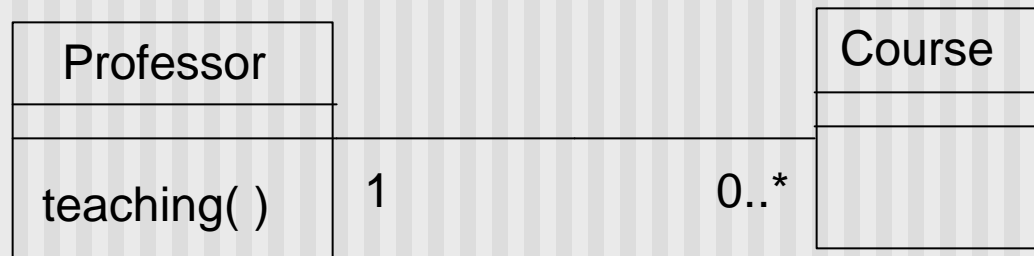
- Si cada curso tiene como máximo un Professor, entonces cada objeto Course puede incluir un apuntador simple al objeto Professor correspondiente



```
class Professor;
class Course {
    public:
    // public info
    private:
    Professor *teacher;
    ...
};
```

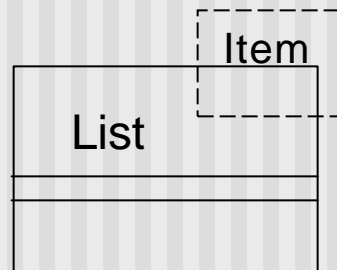
Opcionalidad

- Si una liga es opcional, se debe incluir una operación para probar la existencia de la liga
- Por ejemplo, si un Professor puede estar en sabático, debe incluirse una operación apropiada en la clase Professor para probar la liga

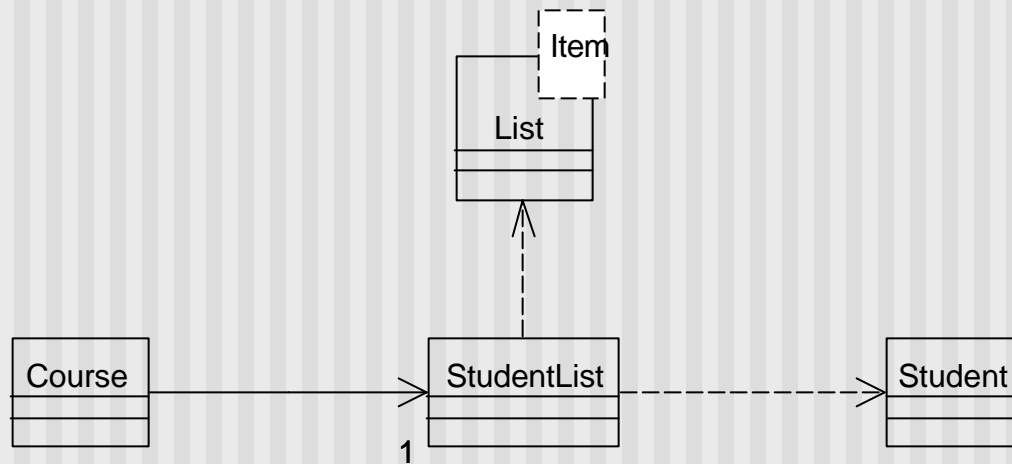


Diseño de Opciones para Multiplicidad de Más de Uno

- La multiplicidad de más de uno se diseña generalmente empleando clases “contenedoras”
 - Una clase contenedora es la clase cuyas instancias son colecciones de otros objetos
- Las clases contenedoras comunes incluyen:
 - Conjuntos, listas, diccionarios, pilas, colas, ...
 - Las clases contenedoras son con frecuencia clases con parámetros que se muestran como:



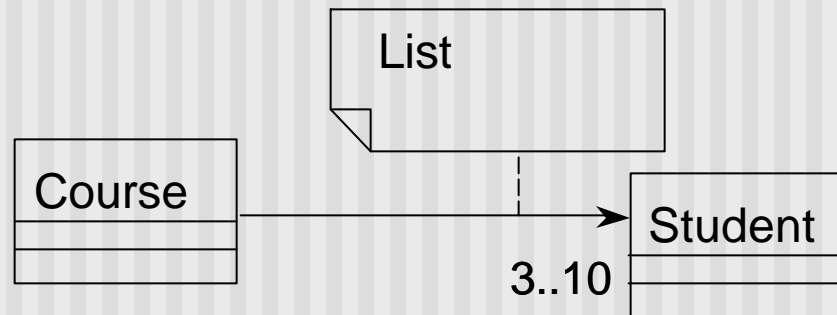
Ejemplo de Clase con Parámetro



```
#include "List.h"
class Course {
public:
    ...
private:
    List<Student>
        students;
    ...
}
```

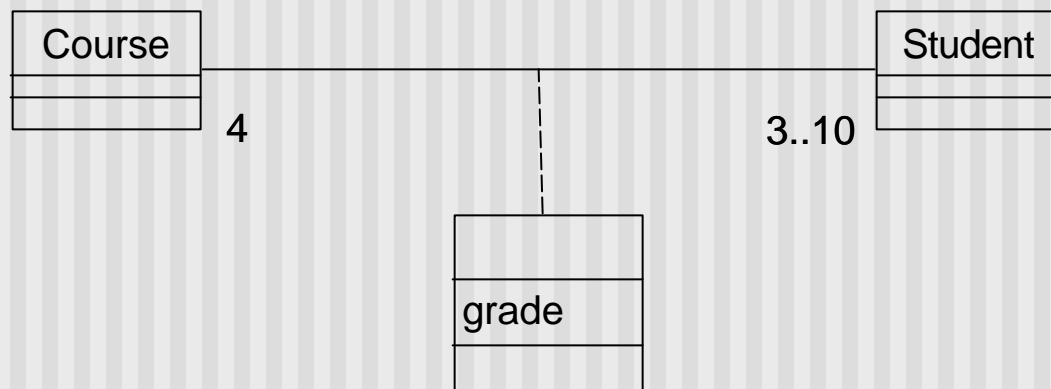
Notación de Clases Contenedoras

- Para reducir el desorden en los diagramas de clases, las clases contenedoras no se muestran típicamente en los diagramas de clases
- Si se necesita el tipo de contenedor para comunicación, se debe usar una nota



Clase Asociación

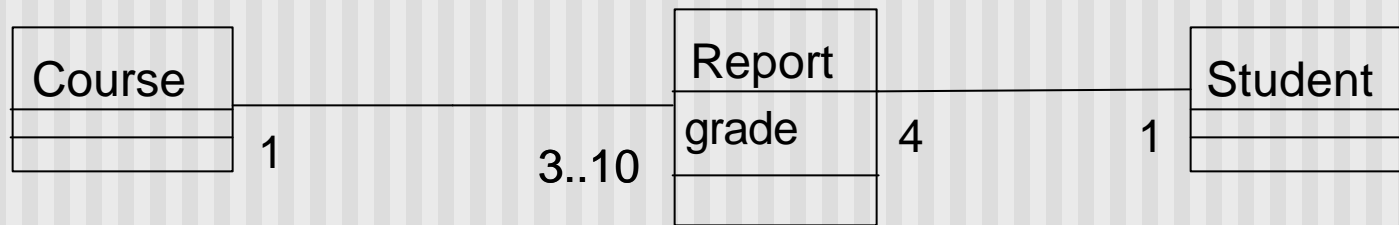
- Una clase de asociación contiene información que pertenece a una liga entre objetos no a cada objeto implicado en la relación



Diseño de Clases de Asociación

- Durante el diseño, las clases de asociación evolucionan:
 - Se transforma a la clase de asociación en una clase interpuesta entre las otras dos clases
 - Se establecen asociaciones con la multiplicidad apropiada entre la clase de asociación y las otras dos clases
 - La multiplicidad siempre es UNA de la clase de asociación a las clases ligadas
- Diseño de asociaciones nuevas
 - La navegación debe ser bi-direccional o uni-direccional

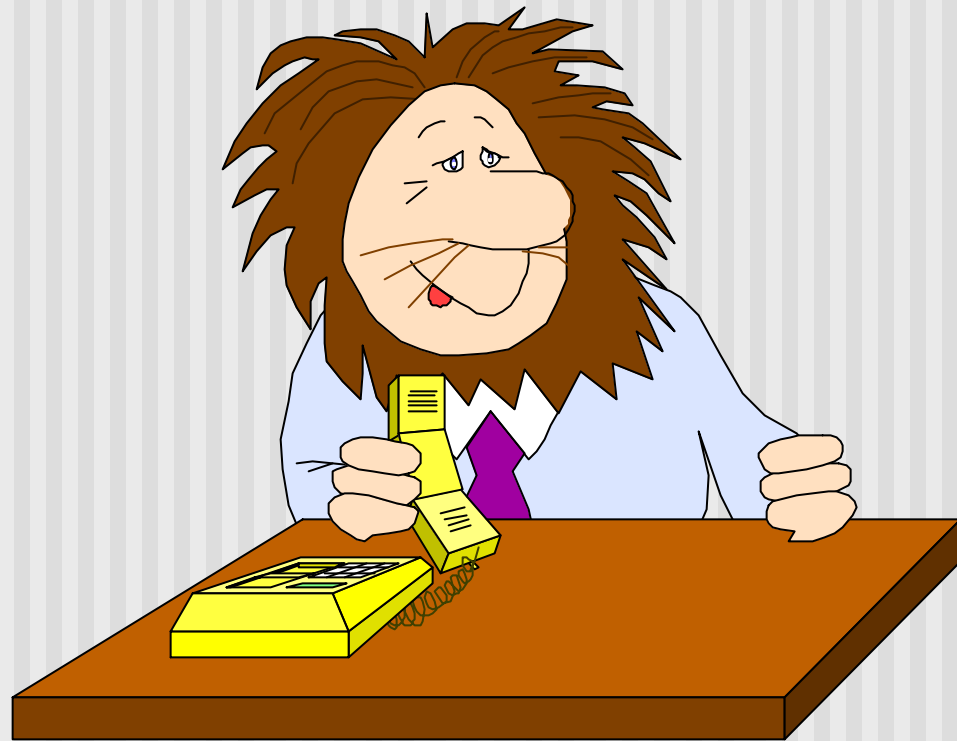
Diseño de Clases de Asociación (cont.)



Ejercicio

- Usando escenarios y diagramas de clases desarrollados
 - Discutir consideraciones del diseño de relación
- Actualice los diagramas de clases para mostrar las consideraciones del diseño de relación

Diseño de Atributos y Operaciones

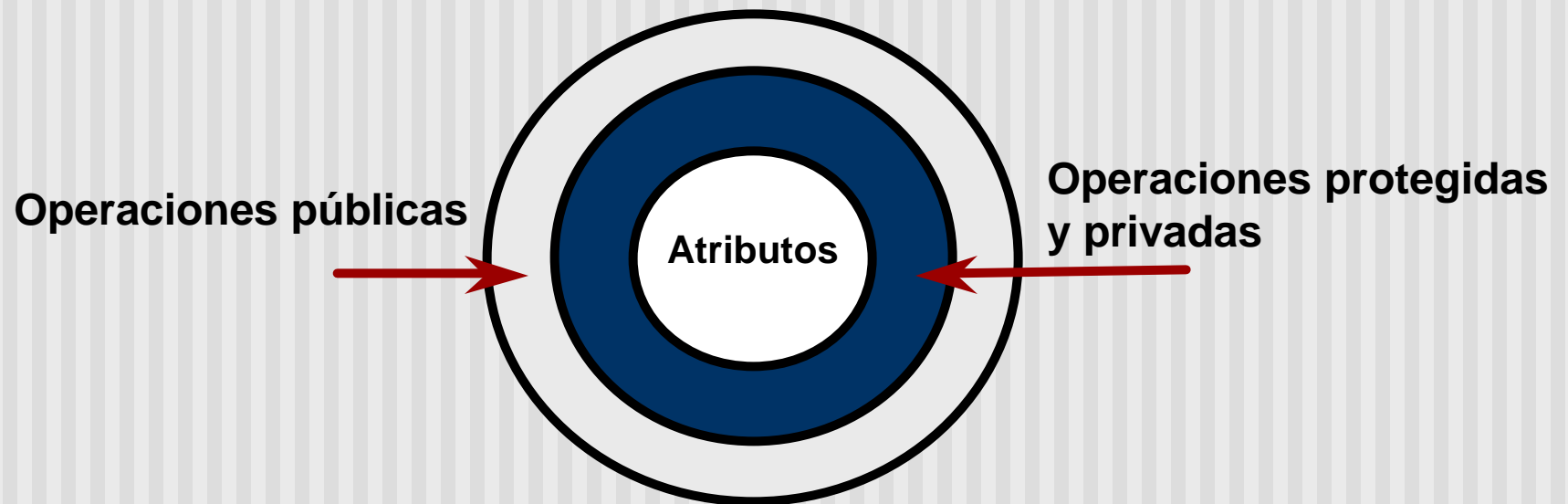


Objetivos: Diseño de Atributos y Operaciones

- Usted podrá:
 - Discutir el control de acceso y encapsulación
 - Usar la notación para control de acceso en diagramas de clases
 - Diseñar representaciones de atributos
 - Representación de tipos de atributos y valores por omisión
 - Listar los cuatro tipos de operaciones comúnmente proporcionadas por una clase
 - Representar las firmas de las operaciones
 - Describir los niveles de control de acceso que soporta C++
 - Discutir la relación de amistad y la encapsulación
 - Discutir las opciones de diseño disponibles en C++ para atributos
 - Discutir el soporte en C++ para operaciones

Control de Acceso y Encapsulación

- La notación UML tiene la habilidad de especificar el acceso de otros atributos y operaciones
 - El acceso debe ser público, protegido o privado
- El control de acceso se usa para reforzar la encapsulación



Control de Acceso Especificado en Diagramas de Clases

- Los símbolos de acceso pueden ser usados en un icono de clase para indicar la accesibilidad de atributos y operaciones
- El control de acceso se especifica para atributos y operaciones precediendo el nombre del miembro con los símbolos siguientes:
 - **+Acceso publico**
 - **#Acceso protegido**
 - **- Acceso privado**
- El acceso es otorgado explícitamente por la clase misma y el cliente no la toma por la fuerza

Ejemplo: Control de Acceso Especificado en Diagramas de Clases

Course
-maxStudents -name
+addStudent () +isFull () #determineCourseSize()

Representaciones de Atributo

- Durante el análisis, fue suficiente especificar sólo el nombre del atributo
 - A menos que la representación sea un requerimiento que restrinja al diseñador
- Las representaciones de atributos deben completarse en el diseño
- Se debe asignar un valor por omisión para cada atributo
 - `attributeName : Type = Default`
- Ejemplo: algunos atributos de la clase Document son:
 - `Title : String`
 - `Author : String`
 - `LastRevised : Date = Today`

Determinación de Tipos de Dato de Atributos

- El diseñador debe seleccionar una representación de dato apropiada para cada atributo de entre los siguientes:
 - Tipo de dato preconstruido (Built-in data type)
 - Tipo de dato definido por el usuario (User-defined data type)
 - Clases definidas por el usuario (User-defined class)
- Los detalles más finos del diseño de atributos dependen del lenguaje de implementación

Análisis

Course
- name - description - maxStudents

Diseño

Course
- name : String - description : DayType - maxStudents : short = 0

Diseño de Operaciones

- Durante el análisis
 - Las operaciones se crean para implementar el comportamiento expresado en los casos de uso
- Durante el diseño
 - Los detalles a nivel de implementación se agregan a las operaciones
 - Se agregan operaciones para completar la clase

Tipos de Operaciones

- Una clase completa debe tener operaciones categorizadas en cuatro tipos*:
 - Funciones de administración
 - Funciones de implementación
 - Funciones de acceso
 - Funciones de ayuda

*[Lippman, Stanley. C++ Primer,. Reading, Mass.: Addison-Wesley, 1991]

Operaciones de Administración

- Se debe proporcionar un subconjunto de operaciones de cualquier clase para proporcionar las funciones de administración de la clase
 - Inicialización, creación, destrucción, administración de memoria, etc.
 - Las operaciones de administración proporcionan estos servicios
- Las más comunes son los constructores y los destructores
 - Desempeñan la creación y destrucción de objetos de la clase
- Todas las clases necesitan proporcionar al menos estas operaciones de administración

Operaciones de Implementación

- El subconjunto de operaciones de una clase que proporcionan las capacidades básicas de la clase se llaman operaciones de implementación
- Estas operaciones son las que prevalecen desde el análisis
- Si se requiere o necesita cualquier rastreo, estas son las operaciones que se rastrean en el diseño

Operaciones de Acceso

- El diseño orientado a objetos encapsula la representación interna de la clase
 - Concepto de ocultamiento de información
- Las operaciones que soportan el acceso a los atributos se les hace referencia como operaciones de acceso
 - También conocidas como operaciones de obtener (get) y colocar (set)
- Cualquier atributo puede tener estas operaciones de acceso

Operaciones de Ayuda

- El ultimo conjunto de operaciones son aquellas llamadas operaciones de ayuda, que se encargan de las funciones auxiliares de la clase que generalmente no es para el usuario
 - No son parte de la interfaz pública
 - Operaciones privadas son usadas solo por miembros de la clase
 - Generalmente se invocan por otras operaciones de la clase

Firmas de las Operaciones

- Durante el diseño, se determina la firma de cada operación
 - Argumentos o parámetros de la operación
 - Tipo de datos de retorno de la operación

Análisis

Course
+addStudent (newStudent)

Diseño

Course
+addStudent (newStudent : Student*):void

Atributos y Operaciones para la Clase Course

Course
<ul style="list-style-type: none">-description : char*-name : char*-daysOffered : DayType-creditHours : short-location : UniversityPlace-maxStudents : short
<ul style="list-style-type: none">+isFull ():bool+addStudent (newStudent : Student*):void+getDescription ():const char*+save ():void+getName ():const char*+getDaysOffered ():dayType+getCreditHours ():short+getLocation ():const UniversityPlace+Course (name : char&,location : UniversityPlace&,desc : char&,days : DayType,hours : short,maxStudents : short)+~Course ()+Course (: const Course&)

Control de Acceso en C++

- El control de acceso de UML mapea directamente a C++. El control de acceso se indica por las siguientes palabras clave:
 - Public: Accesible a todos los clientes
 - Protected: Accesible solo a todas las subclases
 - Private: Accesible solo a la clase misma
- En C++, los atributos se llaman variables miembro mientras las operaciones se llaman funciones miembro.
- El acceso para todos los atributos son usualmente privados o protegidos
- El acceso para todas las operaciones que son parte de la interfaz externa es publica

Ejemplo: Control de Acceso para miembros de una Clase

```
class Course {
```

```
    public:
```

```
        void addStudent(Student*);
```

```
        bool isFull();
```

```
        ...
```



Operación pública

```
    protected:
```

```
        int determineCourseSize();
```



Operación protegida

```
    private:
```

```
        String name;
```

```
        short maxStudents;
```



Atributos privados

```
};
```

Friendship (Amistad)

- Un amigo es una clase u operación cuya implementación puede hacer referencia a partes privadas o protegidas de otra clase
- En C++ el mecanismo de amistad permite a una clase distinguir ciertas clases privilegiadas que tienen los derechos para ver las partes protegidas y privadas de la clase

Precaución: Friendships (Amistades) rompen la encapsulación de la clase, y por lo tanto debe escogerse cuidadosamente y usarse sólo cuando sea absolutamente necesario

Atributos de C++

- Los atributos se llaman variables miembros en C++
- C++ proporciona un rango de tipos de datos para cada variable miembro.
 - Built-in types incluyen:
 - Integers, including short, int, long
 - Floating point numbers, including float, double, long double
 - Character, including char
 - Derived types incluyen arrays, strings, structures, unions y pointers
 - User-defined types como enum y typedef
 - User-defined classes

Tipos Built-in y Derived

Course
-description : char*
-name : char*
-creditHours : short
-maxStudents : short

```
class Course {  
    public:  
    ...  
    private:  
        char *description;  
        char *name;  
        short creditHours;  
        short maxStudents;  
};
```


Tipos Definidos por el Usuario

Course
-description : char*
-name : char*
-creditHours : short
-maxStudents : short
-daysOffered : dayType

```
enum dayType { MW, MWF, TT };  
class Course {  
    public:  
    ...  
    private:  
        char *description;  
        char *name;  
        short creditHours;  
        short maxStudents;  
        dayType daysOffered;  
};
```

Clase Definida por el Usuario

```
class UniversityPlace {  
    public:  
    ...  
    private:  
    char *building;  
    short room;  
};
```

Course
-description : char*
-name : char*
-creditHours : short
-maxStudents : short
-daysOffered : DayType
-location : UniversityPlace

```
#include "UniversityPlace.h";  
enum dayType { MW, MWF, TT };  
class Course {  
    public:  
    ...  
    private:  
    char *description;  
    char *name;  
    short creditHours;  
    short maxStudents;  
    dayType daysOffered;  
    UniversityPlace location;  
};
```

Soporte para Operaciones en C++

- Las operaciones se llaman funciones miembro en C++
- C++ normalmente distingue las operaciones de administración de otras operaciones
- C++ soporta los conceptos de parámetros y valores de retorno en operaciones

Operaciones de Administración en C++

- Cada clase en C++ define operaciones de administración llamadas constructores y destructores
- Si los constructores y los destructores no están definidos, se proporcionan versiones por omisión
- Un constructor define como se construyen los objetos de una clase
- Un constructor de copia se usa para hacer una copia de un objeto existente
- Un destructor define lo que sucede cuando se destruye un objeto

Constructores

- Los constructores son funciones miembro que tienen el mismo nombre de la clase misma
- Los constructores se usan también para iniciar las variables miembro de una clase

```
class Course {  
    public:  
        Course();    // Constructor  
        ...  
    private:  
        short maxStudents;  
};  
  
Course::Course() : maxStudents(0) // Constructor  
{  
    // Constructor code  
};
```

Constructores de Copia

- Los constructores de copia se usan para iniciar un objeto usando los valores de otro objeto
- El compilador creará una copia por omisión si no se provee una
 - Todos los datos miembros en la copia se inician copiando el valor del objeto original
- Los constructores de copia son llamados cuando los objetos se pasan por valor

Constructor de Copia (Por Omisión) Correcto

```
class Circle
{
    public:
        Circle(float x, float y, float r):
            xPosition(x), yPosition(y), radius(r) {};
    private:
        float xPosition;
        float yPosition;
        float radius;
};
```

¿Qué sucede aquí?

```
class Employee {  
    public:  
        Employee(const char *n= "");  
        ~Employee () {delete []name;};  
    private:  
        char *name;  
};  
Employee::Employee(const char *n):  
    name(new char[strlen(n)+1]) {  
        strcpy(name,n);  
    }
```


Clase Employee Correcta

```
class Employee {
    public:
        Employee(const char *n= " ");
        Employee(const Employee &);
        ~Employee () {delete []name;};
    private:
        char *name;
};

Employee::Employee(const char *n):
    name(new char[strlen(n)+1]) {
        strcpy(name,n);
};

Employee::Employee(const Employee& emp):
    name(new char[strlen(emp.name)+1]){
        strcpy(name,emp.name);
}
```

Constructor de Copia

- El constructor de copia por omisión no puede ser usado si:
 - La clase contiene apuntadores o referencias a otros objetos
 - Se hace extra procesamiento cuando se crea y borra un objeto
 - Ejemplo: incrementar el número de objetos en el constructor y decrementar el número de objetos en el destructor

Inicialización de Miembros

- ¿Diseño bueno o malo?

```
class Employee {  
    public:  
        Employee(const String&);  
    private:  
        String name;  
};  
Employee::Employee(const String& n) {  
    name = n;  
}
```

Indicación: ¿Cómo se inicia name?

Inicialización de Miembros

- Se inicia el nombre de atributo usando el constructor string (cadena) por omisión
- El valor del nombre se cambia en el cuerpo del constructor

Inicialización de Miembros - Diseño

- El nombre del atributo inicia usando el constructor String(String &)

```
class Employee {  
    public:  
        Employee(const String&);  
    private:  
        String name;  
};  
Employee::Employee(const String& n):  
    name(n) {}
```

Inicialización de Miembros

- ¿Qué hay acerca de tipos built-in?
 - Los tipos Built-in no tienen constructores entonces no importa

Buen estándar

**Siempre usar inicio de miembro
Código más fácil de leer
Código más fácil de mantener**

Orden de Inicialización

- ¿Qué sucede aquí?

```
extern String LookupEmployee(long);  
class Employee {  
    public:  
        Employee(long);  
    private:  
        String name;  
        long id;  
};  
Employee::Employee(long i):  
    id(i), name(LookupEmployee(id)) {};
```

Orden de Inicialización

- La inicialización ocurre en el orden especificado en la declaración de la clase NO en la orden en el constructor
- En el ejemplo, LookupEmployee se llama usando una variable inicializada
- Arreglar
 - Cambiar el orden en la declaración de la clase

Buen estándar

La orden de declaración de la clase y el orden de inicialización deben ser las mismas

Operaciones de C++

- C++ soporta pasar parámetros a operaciones y regresar un solo elemento como tipo de regreso
- Por ejemplo: `bool addStudent (const Student & student);`
- C++ soporta diferentes modos de pasar parámetros
 - Pasar por valor
 - Pasar por referencia

Pasar por Valor

- Mecanismo por omisión en C++
- Se hace una copia del parámetro actual
- Cambia al parámetro formal, no cambia el parámetro actual

```
void Course::addStudent(Student aStudent){
    ...
    aStudent.setName("noName"); // name of aStudent set to "noName"
};
main() {
    Student theStudent;
    theStudent.setName("Terry");
    aCourse.addStudent(theStudent);
    // name is still set to Terry
    ... }
```

Pasar por Referencia

- Soportado por argumentos de referencia y apuntadores
- No se hace copiado de objetos
- Los cambios del parámetro formal cambian al parámetro actual

```
void Course::addStudent(Student& aStudent){
    ...
    aStudent->setName("noName"); // name of aStudent set to "noname"
};
main() {
    Course aCourse;
    Student *theStudent;
    theStudent = new Student;
    theStudent->setName("Terry");
    aCourse.addStudent(theStudent);
    // name is "noName"
    ... }
```

Apuntadores y Referencias

- Un apuntador no es lo mismo que una referencia
- Una apuntador
 - Es una dirección
 - Puede cambiar para apuntar a otro objeto
 - Puede no estar iniciado
 - Puede ser nulo
- Una referencia
 - Es un alias para un objeto
 - Debe estar iniciado
 - Una vez iniciado, la referencia no puede cambiar para referenciar a otro objeto

Apuntador y Referencia (cont.)

- Un apuntador debe usarse si
 - Hay una posibilidad de que no hay nada a que referenciar (nulo)
 - Hay necesidad de referenciar a cosas diferentes en tiempos diferentes
- Se debe usar una referencia si
 - Hay siempre un objeto que referenciar
 - No hay necesidad de cambiar la referencia

Modificador **Const**

- Un objeto **const** no puede ser modificado
- Al pasar una referencia a un objeto **const** se preserva lo que se pasó por medio de semánticas de valor sin tener que copiar

```
void foo(const T& object) // copy constructor
                           not called
{
    // cannot modify object
}
```

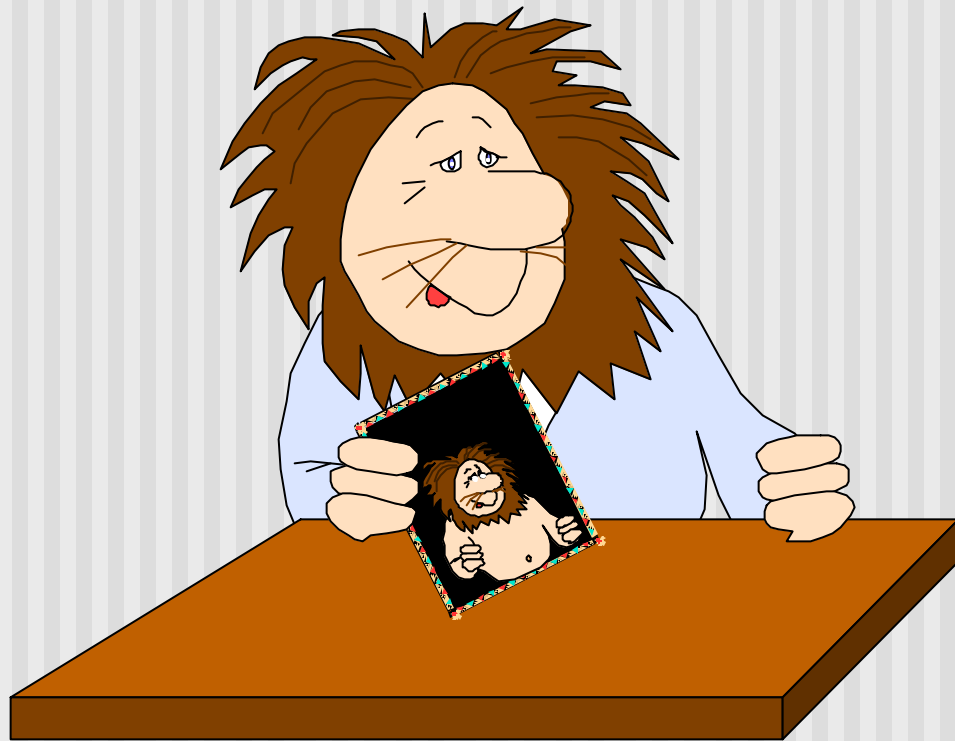
Tipos de Retorno

- C++ soporta regresar un objeto por valor o por referencia
- El regreso por valor resulta en la copia del objeto que regresa que se esta haciendo
 - Puede ser costoso
- El regreso por referencia no resulta en una copia
 - Requiere que el cliente asuma la carga de administración de memoria

Ejercicio

- Usando los escenarios desarrollados, discutir consideraciones de diseño de atributo y operación

Diseño de Herencia

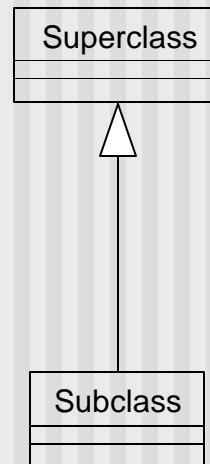


Objetivos: Diseño de Herencia

- Usted podrá:
 - Refinar la jerarquía de herencia del nivel de análisis para incrementar el reuso
 - Discutir como soporta C++ la herencia
 - Definir polimorfismo y describir como se soporta en C++
 - Diferenciar entre ligado estático y dinámico
 - Usar funciones virtuales para solicitar ligado dinámico
 - Determinar el nivel apropiado de acceso para datos y funciones miembros con herencia

Jerarquías de Herencia

- Durante el análisis, se establecen jerarquías de herencia
- Durante el diseño, las jerarquías de herencia se refinan a:
 - Incrementar reuso
 - Incorporar clases de implementación
 - Incorporar clases de librería disponibles



Refinar la Jerarquía de Herencia

- Se revisan los diagramas de análisis para identificar cosas comunes de
 - Atributos, y/o
 - Operaciones, y/o
 - Asociaciones
- Se definen superclases nuevas que contengan elementos comunes
- Esto reduce la cantidad de código que se va a escribir y refuerza la uniformidad, no se puede manejar un mismo elemento en dos clases diferentes si las dos clases lo heredan de una superclase común

Ejemplo: Refinar la Jerarquía

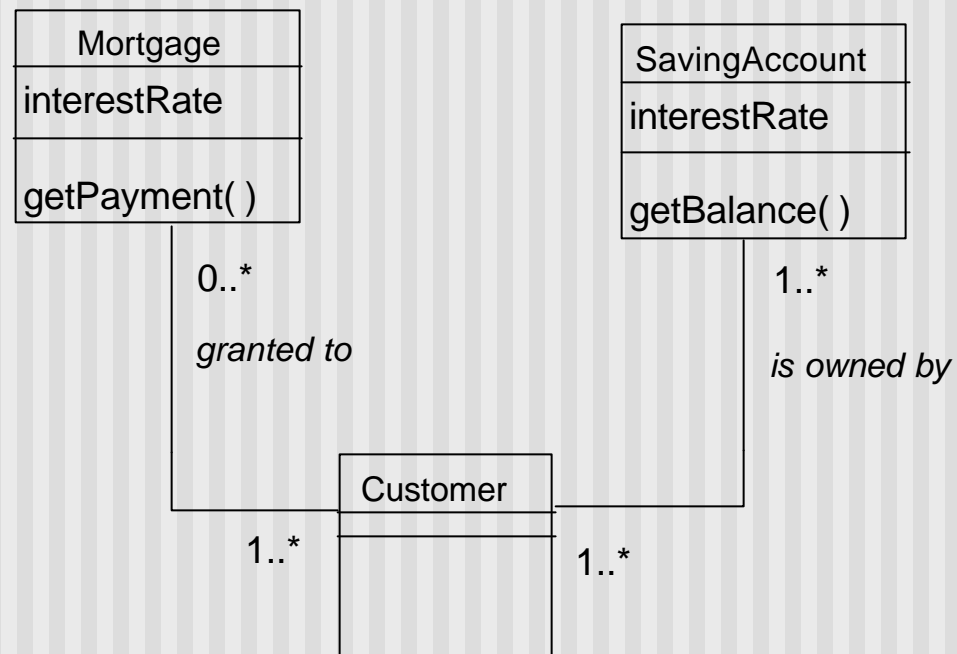


Diagrama de Clase a Nivel Análisis

Ejemplo: Refinar la Jeraquía (cont.)

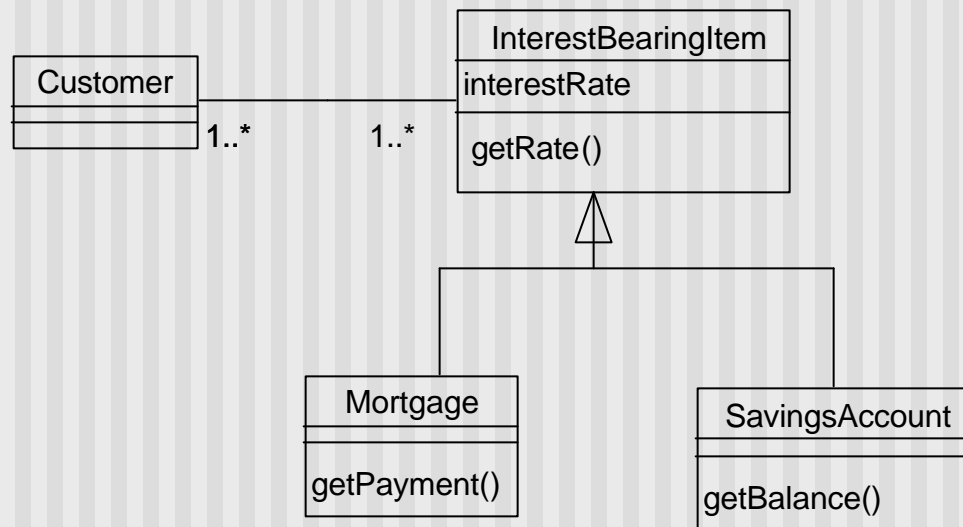


Diagrama de Clases a Nivel de Diseño

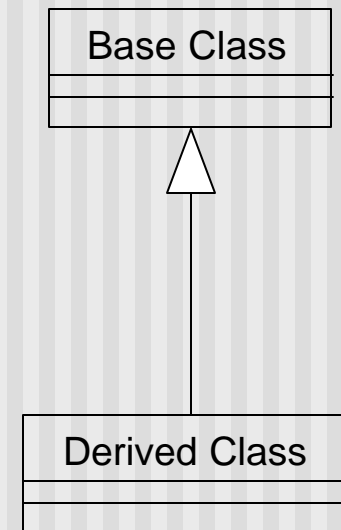
La asociación con Customer se cambió a superclase

interestRate se hereda a ambas subclases y debe manejarse de forma idéntica

Ambos `getPayment()` y `getBalance()` requieren cálculo del interés que es llevado a cabo por el método heredado `getRate()`

Soporte de Herencia en C++

- C++ proporciona soporte a nivel lenguaje, directo para herencia de atributos y operaciones
- La terminología de herencia es
 - Clase "Base" en lugar de superclase
 - Clase "Derivada" en lugar de subclase



Las Operaciones se heredan de Clases Base en C++

```
class BankAcct {  
    public:  
        void deposit ();  
};  
class SavingsAcct : public BankAcct {  
    public:  
        void getBalance ();  
};  
  
void client () {  
    SavingsAcct myAcct;  
    myAcct.getBalance ();    // derived  
    myAcct.deposit ();      // base  
};
```

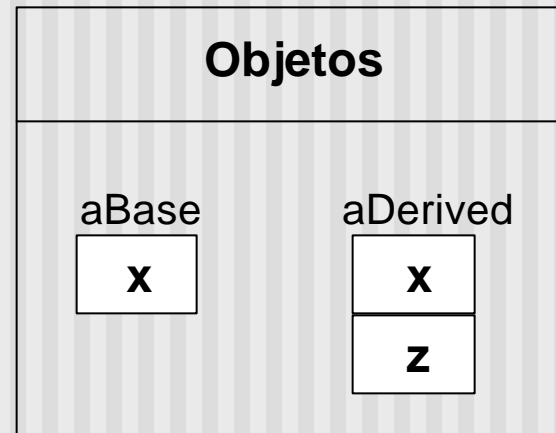
Las operaciones declaradas en una clase base no necesitan repetirse en la clase derivada

Un cliente de una clase derivada puede invocar miembros public de clases derivadas y base

Los Atributos se heredan de Clases Base en C++

```
class base {  
private:  
    int x;  
};  
class derived : public base {  
private:  
    int z;  
};
```

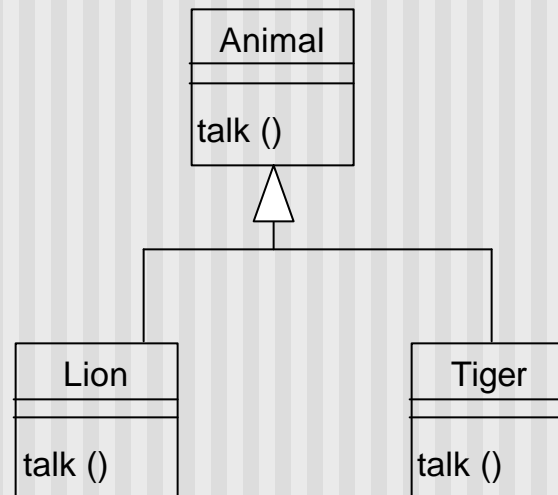
Los atributos declarados en un clase base no necesitan repetirse en la clase derivada



Polimorfismo

- El término griego **Polymorphos** quiere decir “tener muchas formas”
 - El polimorfismo es la habilidad de definir una sola interfaz con implementaciones múltiples
- Los clientes pueden invocar operaciones de objetos sin saber su tipo
 - Los clientes pueden implementarse “genéricamente” para invocar una operación de objeto sin saber el tipo de objeto
 - Si los objetos se agregan eso soporta la misma operación, el cliente no necesita ser modificado para manejar al objeto nuevo
- El polimorfismo permite que los clientes manipulen objetos en términos de su superclase común

Ejemplo de Polimorfismo



Sin Polimorfismo

```
if animal = "Lion" then
    do the Lion talk
else if animal = "Tiger" then
    do the Tiger talk
end
```

Con Polimorfismo

do the Animal talk

Polimorfismo y C++

- El polimorfismo es una ventaja de herencia realizada durante la implementación
- C++ proporciona soporte para el polimorfismo a través de
 - Ligado dinámico (o tardío)
 - Funciones virtuales
- El diseñador debe permitir explícitamente el polimorfismo a través del uso apropiado de funciones miembro virtuales de C++

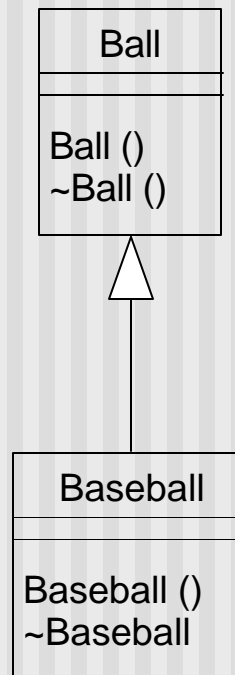
```
virtual void talk();
```

Ligado Dinámico vs. Estático

- Normalmente el método particular que se va a ejecutar como resultado de una llamada de función se conoce al momento de compilación. Esto se llama ligado estático (o temprano)
 - El compilador reemplaza la llamada de función con código diciendo el programa que direcciona para saltar y poder encontrar esa función
- Con el polimorfismo, el tipo particular de objeto para el que un método se va a invocar no se conoce hasta el tiempo de ejecución
 - El compilador no puede proporcionar la dirección al tiempo de compilación
 - El programa selecciona al método mientras esta corriendo
- Esto se conoce como ligado tardío o ligado dinámico

Herencia y destructores

- Si un destructor no es virtual entonces el borrado a través del apuntador de la clase base llamará al destructor incorrecto, si el objeto al que apunta es de la clase derivada



Herencia y destructores (cont.)

```
class Ball {  
    public:  
        Ball();  
        ~Ball();  
}
```

```
class Baseball : public Ball {  
    public:  
        Baseball();  
        ~Baseball();  
}
```

```
main()  
{  
    Ball *myBall;  
  
    myBall = new Baseball();  
  
    delete myBall;  
}
```

OK -- Baseball constructor called

not OK -- Ball destructor called

Virtual vs. Desempeño

- Los logros en el desempeño se realizan al no tener que ejecutar declaraciones **switch/case**
- Cada vez que se llama a una función virtual, la función correcta que se va a invocar se determina al examinar la tabla virtual
 - Se establecen algunas instrucciones por llamada
- Las funciones no pueden ponerse en línea (inline)
 - El compilador no sabe que poner en línea
- Mayor impacto en funciones pequeñas
 - El tiempo para la llamada de función es un porcentaje significativo del tiempo de ejecución de función

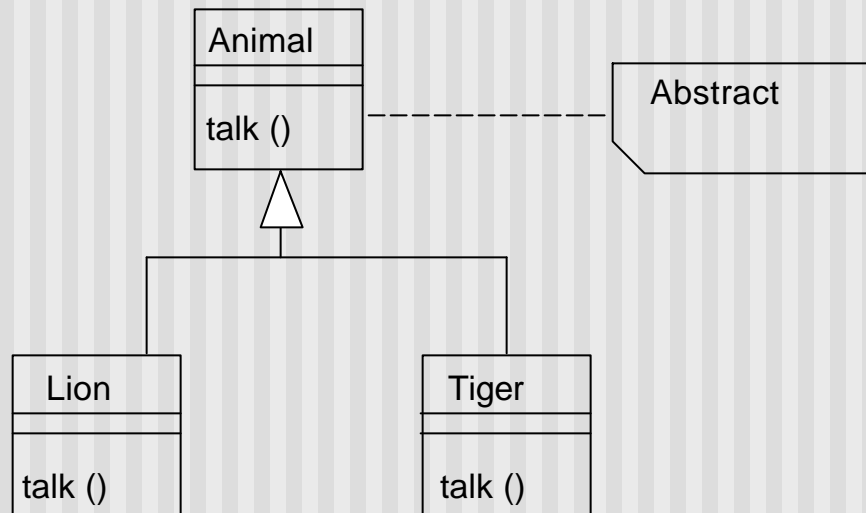
Sugerencia: Si el uso de funciones virtuales y clases virtuales hace más claro el diseño, y el código más fácil de entender entonces probablemente valga la pena

Diseño del Polimorfismo

- En el diseño, el desarrollador debe:
 - Examinar las jerarquías de herencia y determinar cuales operaciones deben ser polimorfas
 - Actualizar los diagramas de clases para indicar que cada clase y/o subclase deben proporcionar un método para una operación dada
 - Declarar todas las operaciones polimorfas que van a ser virtuales en la clase base que los define

Clases Abstractas

- Una clase abstracta es aquella a la que no se le pueden crear instancias
- Las clases abstractas deben tener al menos una subclase para ser útiles

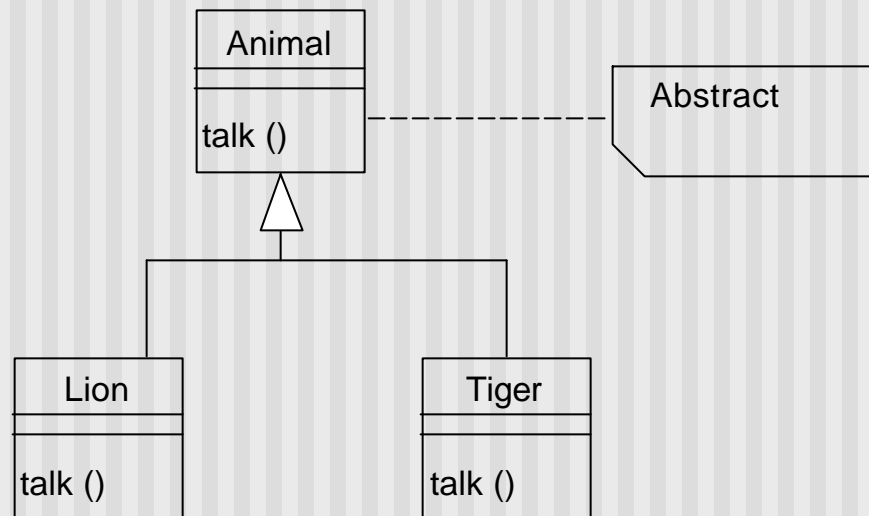


Todos los objetos son ya sea leones o tigres; no hay instancias directas de Animal

Diseño de Clases Abstractas

- Las clases abstractas se diseñan de forma diferente de las demás clases
- Con las clases abstractas el diseñador asume que las subclases agregaran su estructura y comportamiento
 - La clase abstracta no necesita proporcionar métodos para cada operación que define
 - La clase abstracta debe proporcionar el protocolo para todas las operaciones polimorfas

Ejemplo: Clases Abstractas y Protocolos



El Animal no necesita especificar el método `talk()`

Los métodos deben ser provistos por Lion y Tiger para `talk()` y estos métodos deben conformar al protocolo definido en Animal

C++ y Clases Abstractas

- C++ permite que un desarrollador afirme que un método de una clase abstracta no pueda ser invocado directamente, al iniciar su declaración a cero
- Tal método es llamado una **función virtual pura**
- C++ prohíbe la creación de instancias de clases que contengan funciones virtuales puras

```
class Animal {  
    ...  
    virtual void talk() = 0;  
};
```

← Asegura que no se pueden crear instancias de Animal

Clases Abstractas y Herencia

- Hay 3 consideraciones de diseño importantes de funciones involucradas aquí:
 - Proporcionar una interfaz de función solo a clases derivadas:
 - Usar una función virtual pura
 - Proporcionar una interfaz de función y comportamiento por omisión a clases derivadas:
 - Usar una función virtual (no pura) por omisión (con código que puede prevalecer)
 - Proporcionar una interfaz de función y comportamiento obligatorio a clases derivadas:
 - Usar una función no virtual (que NO está diseñada para permanecer en subclases)

Derivación Pública vs. Derivación Privada

- Las subclases se implementan en C++ usando derivación pública
- Con derivación pública, la interfaz pública de la clase base permanece pública en la clase derivada
 - Los objetos de la clase derivada pueden acceder todos los elementos públicos de la clase base
 - Las funciones miembro de la clase derivada tienen acceso a todos los atributos y operaciones públicas y con herencia protegida

Derivación Pública vs. Derivación Privada (cont.)

- C++ también permite derivación privada en la que la interfaz pública de la clase base se convierte en privada en la clase derivada
 - Los objetos de la clase derivada no pueden acceder elementos públicos de la clase base
 - Las funciones miembro de la clase derivada aún tiene acceso a todos los atributos y operaciones protegidos y públicos
- La derivación privada es de ayuda en implementaciones de reuso - no es herencia verdadera y no se usa para implementar subclases
 - Significa "se implementa en términos de"

Principio de Substitución de Liskov

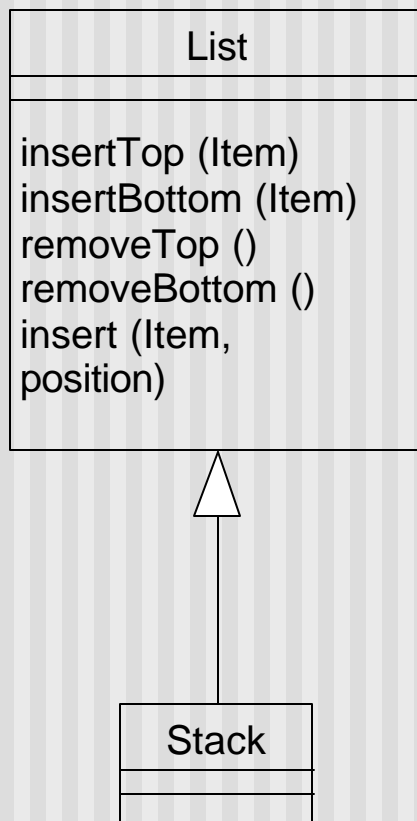
- Si para cada objeto O1 de tipo S hay un objeto O2 de tipo T tal que para todos los programas P definidos en términos de T, el comportamiento de P no se cambia cuando O1 se substituye por O2 entonces S es un subtipo de T

Menos formal:

- Puede siempre pasar un apuntador o referencia a una clase derivada a una función que espera un apuntador o referencia a una clase base

Estas reglas representan el estilo ISA de programación

Estilo de Programación ISA



¿Siguen estas clases el estilo de programación ISA?

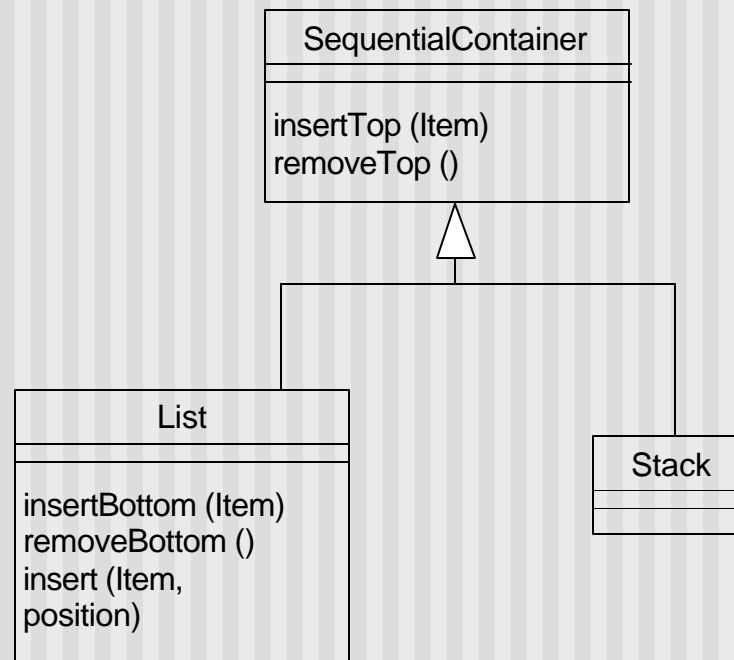
Estilo de Programación ISA (cont.)

- Las clases NO siguen el estilo ISA de programación
 - Un **Stack** necesita algo del comportamiento de una **List** pero no todo
- Si un método espera una **List**, entonces la operación **insert(position)** debe ser exitosa
 - Si a un método se le pasa un **Stack**, entonces el **insert(position)** fallará

Una subclase (Clase derivada) de estilo ISA NO debe tener más restricciones que su superclase

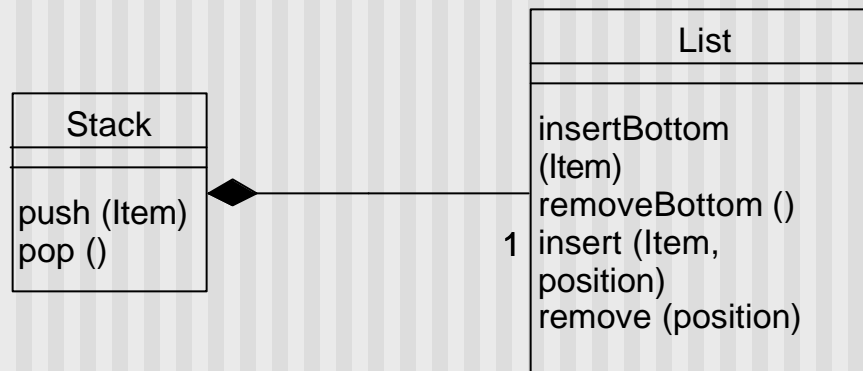
Factoring

- A veces puede usarse Factoring para arreglar el problema
 - No puede usarse si la clase List no puede cambiarse



Delegación

- También se puede usar delegación para arreglar el problema

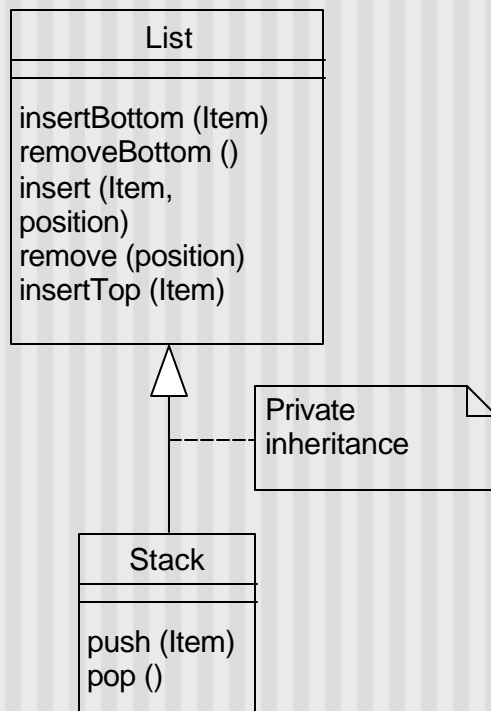


```
void Stack::push(Item I) {
    body.insertTop(I);
};

const Item Stack::pop() {
    return body.removeTop();
};
```

Herencia Privada

- La herencia privada se usa a veces para circumvent problemas de estilo ISA



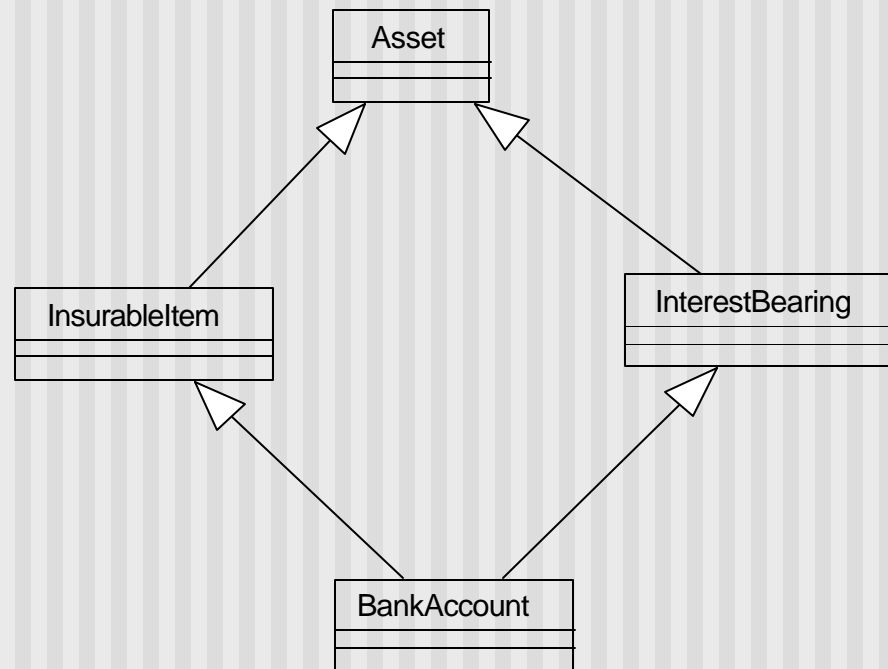
```
void Stack::push(Item I) {
    insertTop(I);
};

const Item Stack::pop() {
    return removeTop();
};
```

push() y pop() pueden acceder métodos de List pero las instancias de Stack no pueden

Herencia Múltiple

- Con herencia múltiple, una subclase hereda de más de una superclase

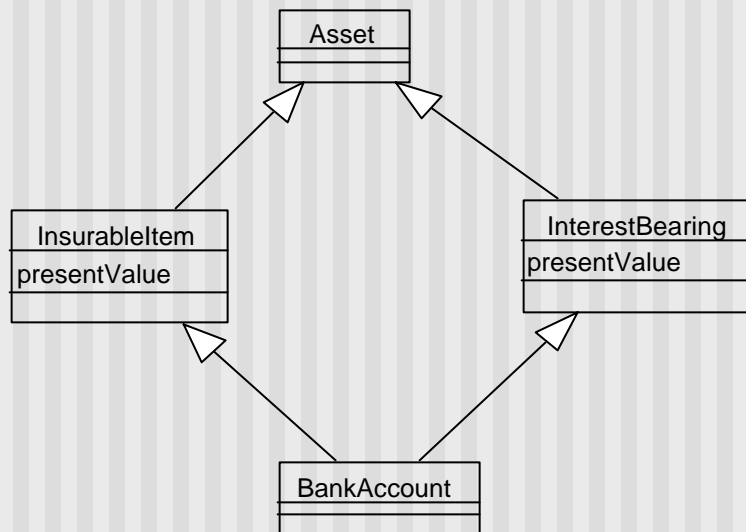


Soporte de C++ para Herencia Múltiple

- Mucha más complejidad asociada con el diseño de herencia múltiple
- Con frecuencia sobre-utilizado
- Deben resolverse dos problemas:
 - Colisión de nombres o
 - Herencia repetida

Colisiones de Nombres

- Las colisiones de nombres resultan cuando dos o mas superclases definen el mismo atributo u operación
- Ambos InsurableItem y InterestBearingItem tienen atributos que se llaman presentValue



**Un objeto BankAccount quiere imprimir el presentValue
¿Cuál se imprime?**

Resolución de Colisiones de Nombres

- Esta ambigüedad puede resolverse al calificar totalmente el nombre para indicar la fuente de la declaración

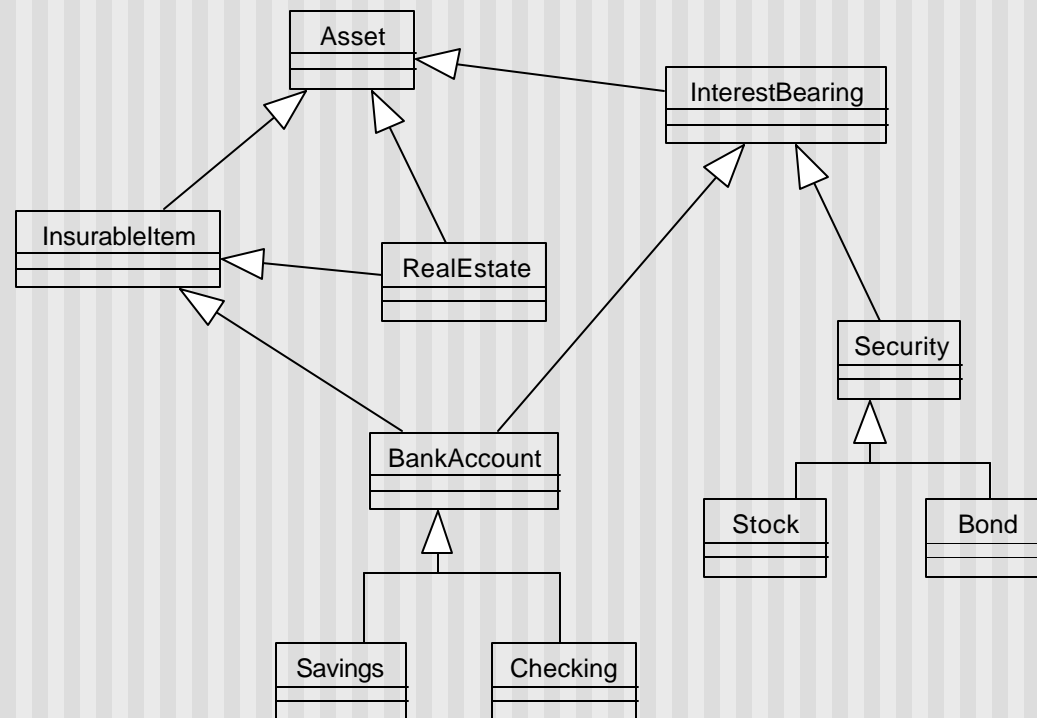
- `InsurableItem::presentValue`

O

- `InterestBearingItem::presentValue`

Ejemplo: Herencia Múltiple

- Entre más compleja sea la herencia, es más difícil detectar colisiones de nombres

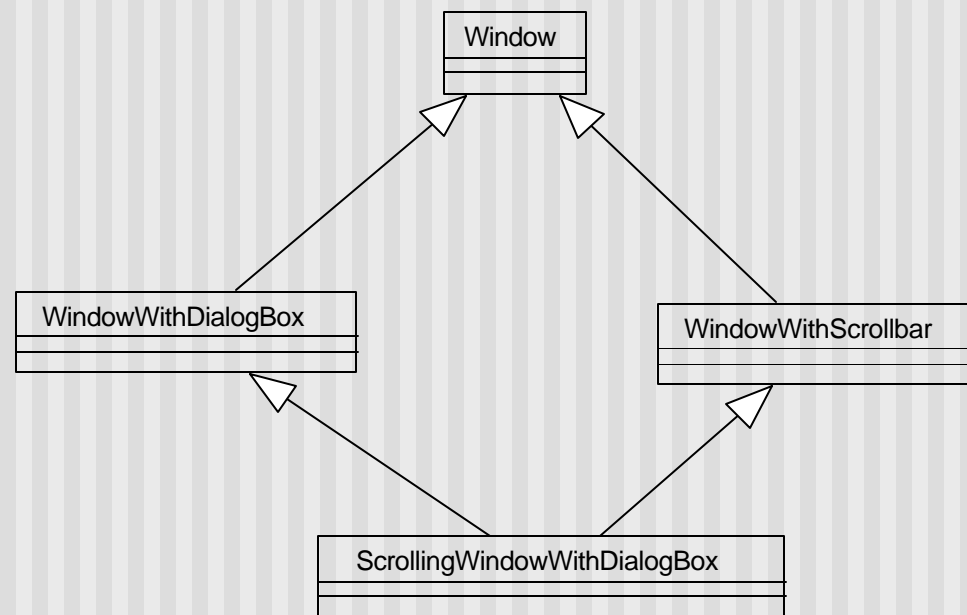


Ejemplo: Declaraciones en C++

```
class Asset . . .
class InsurableItem : public Asset
class InterestBearing : public Asset
class BankAccount : public InsurableItem,
                    public InterestBearing . . .
class RealEstate : public Asset,
                  public InsurableItem . . .
class Security : public InterestBearing . . .
class SavingsAccount : public BankAccount . . .
class CheckingAccount : public BankAccount . . .
class Stock : public Security . . .
class Bond : public Security . . .
```

Herencia Repetida

- Otro problema asociado con la herencia múltiple es la herencia repetida. Considere el siguiente ejemplo:



Herencia Repetida (cont.)

- Note la figura de diamante distintiva de la jerarquía de herencia
- Esto indica que la misma clase base esta siendo heredada por una clase derivada más de una vez. Por ejemplo, `ScrollingWindowWithDialogBox` esta heredando `Window` más de una vez
 - Con la herencia repetida, dos o más superclases comparten una superclase común
- ¿Cuántas copias de los atributos de Windows se incluyen en instancias de `ScrollingWindowWithDialogBox`?

Clases Base Virtuales

- En este caso, `ScrollingWindowWithDialogBox` solo necesita una copia de las variables de la instancia `Window`
- Para asegurar que una copia es heredada, se declara la clase base común como virtual cuando está siendo derivada a clases base intermedias
- Todas las clases intermedias derivan de la clase base común de forma virtual
- La única copia que es heredada se considera que se comparte por las múltiples rutas de derivación
- Uso del operador de resolución de ámbito para referir a elementos compartidos heredados de la clase base común no se requiere con la derivación virtual

Ejemplo: Clase Base Virtual

- Window es la clase base
- Cada ventana tiene una "x" y una "y" que indican el punto de origen
- La función miembro Window::paint pinta la ventana básica

```
class Window {  
public:  
    ...  
    virtual void paint( ) {  
        // paint window stuff only  
    }  
protected:  
    Point x, y; // origin  
    ...  
} ;
```

Ejemplo: Clases Derivadas Intermedias

- Las clases derivadas intermedias derivan de Windows de manera virtual
- Cada clase derivada hereda una "x" y una "y" de la clase base y la función miembro Window::paint

```
class WindowWithDialogBox :  
    public virtual Window {  
public:  
    ...  
    void dialogBoxPaint( );  
        // paint only dialog box  
    virtual void paint( );  
        // invoke Window::paint  
        // and  
        // paint dialog box  
};
```

```
class WindowWithScrollBar :  
    public virtual Window {  
public:  
    ...  
    void scrollBarPaint( );  
        // paint only scrollbar  
    virtual void paint( );  
        // invoke Window::paint and  
        // paint scrollbar  
    ...  
} ;
```

Ejemplo: Herencia Repetida

- La clase derivada `ScrollableWindowWithDialogBox` hereda una "x" y una "y", porque ambas clases padre fueron virtualmente derivadas de `Window`
- Note que esta clase no incluye la palabra reservada `virtual` de sus clases padres

```
class ScrollableWindowWithDialogBox :  
    public WindowWithDialogBox ,  
    public WindowWithScrollBar {  
public:  
    ...  
    virtual void paint( );  
    ...  
} ;
```

Herencia Repetida y Funciones Miembro

- Un error común al diseñar una clase derivada del más bajo-nivel es invocar la función común de la clase base más de una vez
- Por ejemplo, suponga que hace un código de la función **pintar** del más bajo-nivel como sigue:

```
virtual void ScrollableWindowWithDialogBox::paint() {  
    WindowWithDialogBox::paint( );  
    WindowWithScrollBar::paint( ); }  
}
```

- Entonces la función `Window::paint` será invocada dos veces, una de `WindowWithDialogBox::paint` y otra de `WindowWithScrollBar::paint`
- Esto podría ser solo una pérdida de tiempo o podría (en algunos sistemas) corromper la imagen en pantalla

Herencia Repetida y Funciones Miembro (cont.)

- En el código siguiente, se ha evitado el error

```
virtual void ScrollableWindowWithDialogBox::paint( )
{
    Window::paint( ) ;
        // invoke base class paint only once !

    WindowWithDialogBox::dialogBoxPaint( );
        // then paint the dialog box

    WindowWithScrollBar::scrollBarPaint( );
        // then paint the scrollbar
}
```

Herencia Múltiple

- La herencia múltiple es conceptualmente directa y se necesita para modelar exactamente varios problemas del mundo-real
- Los diseñadores novatos tienden al sobreuso de la herencia múltiple, por ejemplo, el usar herencia múltiple cuando se podría usar la agregación
- En la práctica, la herencia múltiple es un problema de diseño complejo y puede guiar a dificultades de implementación, incluyendo colisiones de nombres y herencia repetida

**Use herencia múltiple solo cuando sea necesario,
y
siempre con precaución!**

Ejercicio

- Discuta las decisiones de diseño de herencia para el problema que se esta desarrollando

Resumen General



Objetivos: Resumen

- Usted podrá:
 - Resumir el curso listando las actividades que ocurren durante cada fase del proceso de desarrollo

Resumen -- El Proceso de Desarrollo

- El proceso de desarrollo es una vista de alto nivel del proceso completo al desarrollar un producto de software
 - Comprende cuatro fases: inicio, elaboración, construcción y transición
- Las fases del ciclo de desarrollo se descomponen en una serie de iteraciones a través de las cuales el software evoluciona incrementalmente
 - Cada iteración se planea para controlar los elementos de mayor riesgo del sistema
- Las actividades del análisis y diseño ocurren durante las fases de inicio, elaboración y construcción

Resumen -- Fase Inicio

- El propósito de esta fase es establecer el caso de uso para un nuevo sistema o para la actualización de un sistema existente
- Las salidas incluyen un conjunto de requerimientos esenciales para el proyecto, una valoración inicial del riesgo y, opcionalmente un prototipo conceptual y un modelo inicial del dominio
- Un prototipo conceptual puede construirse para validar hipótesis y percepciones de riesgo (tales como funcionalidad, desempeño, tamaño, complejidad, base tecnológica)
 - Frecuentemente se intenta no tomar en cuenta al código

Resumen -- Fase de Elaboración

- Esta fase envuelve el análisis del dominio del problema y el establecimiento de una base arquitectónica para el sistema
- Las salidas de esta fase consisten en un modelo del comportamiento del sistema que incluye el modelo de casos de uso, escenarios, un modelo del dominio, una arquitectura de ejecutables y una estrategia de diseño que controle los mecanismos clave necesarios para el desarrollo del sistema
- El modelo de casos de uso contiene actores y casos de uso
 - Un actor es alguien o algo que intercambia información activamente con el sistema, proporciona entradas al sistema o recibe pasivamente información del sistema
 - Un caso de uso demuestra funcionalidad desempeñada por el sistema en respuesta a estímulos de un actor

Resumen -- Fase de Elaboración (cont.)

- Un escenario es una secuencia de declaraciones expresadas en lenguaje natural
 - Es una instancia de un caso de uso
- Los objetos se identifican al examinar los escenarios y los objetos relacionados se agrupan en clases
- El modelo de dominio inicial se actualiza para contener las clases nuevas
- Las arquitecturas buenas se construyen en capas bien definidas de abstracción donde cada capa
 - Representa una abstracción coherente
 - Tiene una interfaz bien definida y controlada
 - Se construye sobre facilidades bien definidas y controladas en niveles menores de abstracción

Resumen -- Fase de Elaboración (cont.)

- La arquitectura se valida al crear una versión ejecutable que:
 - Aplique alguno o todos los comportamientos de los escenarios claves
 - Su código de producción sea de calidad
 - Considere la mayoría, sino todas, las interfaces arquitectónicas claves
- Un mecanismo clave es una decisión basada en estándares, políticas y prácticas comunes de la organización
 - Ejemplo: administración de recursos, persistencia, manejo de errores, distribución de objetos y migración

Resumen -- Fase de Construcción

- En esta fase, las iteraciones envuelven el ciclo de vida del desarrollo de la aplicación
 - El desarrollo se realiza a través de una secuencia de iteraciones
- Cada iteración se compone de
 - Una valoración del análisis, diseño e implementación actual para identificar riesgos críticos sin resolver
 - Los escenarios ilustran los riesgos del modelo actual del análisis y diseño, por lo que se extiende a manejar estos riesgos
 - La implementación se extiende para retirar los riesgos identificados
 - Después de que se revisó y probó la implementación, se actualiza el modelo del análisis y diseño para reflejar los cambios hechos durante la implementación
- La siguiente iteración inicia con el modelo actualizado

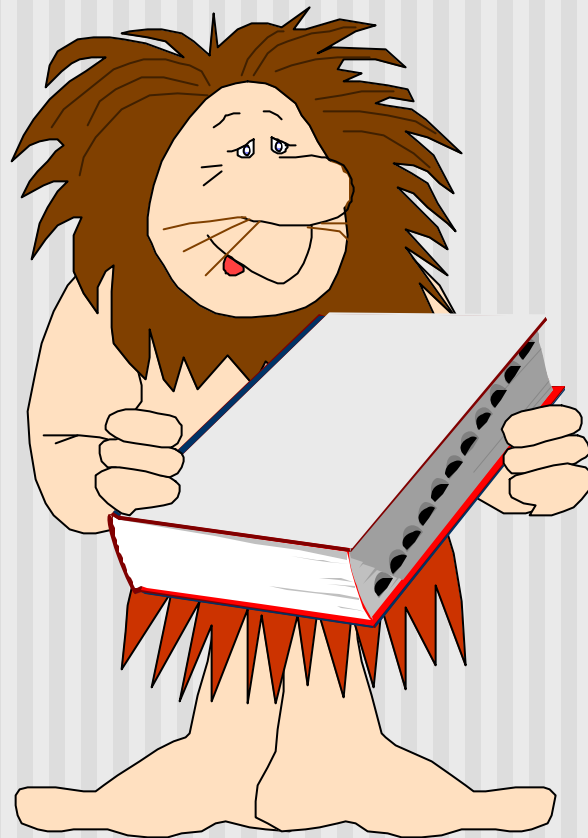
Resumen -- Fase de Construcción (cont.)

- Las actividades de esta fase incluyen:
 - La adición de clases al modelo reflejando el “CÓMO” debe desarrollarse el sistema
 - Clases de Interface
 - Clases de Controller
 - Clases de Helping
 - Refinamiento de la navegación de relaciones
 - Especificación del tipo de contenido de relación
 - Por referencia
 - Por valor
 - La madurez de algunas relaciones con respecto a su dependencia

Resumen -- Fase de Construcción (cont.)

- Actividades de diseño (continuación)
 - Especificación de tipos de datos de los atributos
 - Especificación de las firmas de las operaciones
 - Adición de operaciones de administración, acceso y ayuda
 - Refinamiento de jerarquías de herencia para incrementar la reutilización
 - Adición de funciones virtuales para soportar el polimorfismo
 - Resolución de problemas de herencia múltiple

Bibliografía



Análisis y Diseño

- G. Booch, **Object-Oriented Analysis and Design with Applications**, Benjamin/Cummings, Redwood City, CA, 1994
- J. Rumbaugh, et al., **Object-Oriented Modeling and Design**, Prentice-Hall, Englewood Cliffs NJ, 1991
- I. Jacobson, et al., **Object-Oriented Software Engineering**, Addison-Wesley, Reading, MA, 1992
- R. Wirfs-Brock et al., **Designing Object-Oriented Software**, Prentice-Hall, Englewood Cliffs NJ, 1990
- N. Wilkinson, **Using CRC Cards**, SIGS Books, New York, NY, 1995
- M. Lorenz, **Object-Oriented Software Development**, Prentice-Hall, Englewood Cliffs NJ, 1993

Análisis y Diseño

- G. Booch (edited by ed Eykholt), **Best of Booch: Designing Strategies**, SIGS Books & Multimedia, New York, NY, 1996
- J. Rumbaugh, OMT Insights: **Perspectives on Modeling from the Journal of Object-Oriented Programming**, SIGS Books & Multimedia, New York, NY, 1996

Diseño de Interfaz de Usuario

- D. Collins, **Designing Object-Oriented User Interfaces**, Benjamin/Cummings, Redwood City, CA, 1995
- G. Lee, **Object-Oriented GUI Application Development**, Prentice Hall, Englewood Cliffs, NJ, 1993

Arquitectura

- E. Rechtin, **Systems Architecting: Creating and Building Complex Systems**, Prentice-Hall, Englewood Cliffs NJ, 1991
- E. Gamma, et al., **Design Patterns: Elements of Reusable Object-Oriented Software**, Addison-Wesley, 1995
- D. E. Perry, and A. L. Wolf, **Foundations for the Study of Software Architecture**, ACM Soft. Eng. Notes, 17 (4), Oct. 1992, pp. 40-52
- P. Kruchten, **The 4+1 View Model of Architecture**, IEEE Software, November 1995

Software General

- S. McConnell, **Code Complete - A Practical Handbook of Software Construction**, Microsoft Press, Redmond, WA, 1993
- J. McCarthy, **Dynamics of Software Development**, Microsoft Press, Redmond, WA, 1995
- F. P. Brooks, Jr., **The Mythical Man-Month**, Addison-Wesley, Reading, MA, 1995
- D. Firesmith and E. Eykholt, **Dictionary of Object Technology**, SIGS Books, New York, NY, 1995

Metrics

- M. Lorenz and J. Kidd, **Object-Oriented Software Metrics**, Prentice Hall, Englewood Cliffs, NJ, 1994

Administración de Proyectos

- G. Booch, **Object Solutions**, Addison-Wesley, Reading, MA, 1996

Mecanismos Clave

- R. Orfali, D. Harkey, and J. Edwards, **The Essential Distributed Objects Survival Guide**, John Wiley & Sons, New York, NY, 1996
- D. Barry, **The Object Database Handbook**, John Wiley & Sons, New York, NY, 1996

Libros Introdutorios de C++

- S. Lippman, C++ Primer, Addison-Wesley, Reading, MA , 1991
- Stroustrup, C++ Programming Language, Addison-Wesley, Reading, MA, 1991
- P. Winston, On to C++, Addison-Wesley, Reading, MA, 1994
- S. Davis, C++ for Dummies, IDG Books, Foster City, CA, 1994
- B. Eckel, Thinking C++, Prentice-Hall, Englewood Cliffs, NJ, 1995

Libros Intermedios de C++

- S. Meyers, **Effective C++**, Addison-Wesley, Reading MA, 1992
- R. Murray, **C++ Strategy and Tactics**, Addison-Wesley, Reading MA, 1993
- T. Cargill, **C++ Programming Style**, Addison-Wesley, Reading MA, 1992
- R. Martin, **Designing Object-Oriented C++ Applications Using the Booch Method**, Prentice Hall, Englewood Cliffs, NJ, 1995
- J. Soukup, **Taming C++**, Addison-Wesley, Reading MA, 1994
- A. Riel, **Object-Oriented Design Heuristics**, Addison-Wesley, Reading MA, 1996
- S. Meyers, **More Effective C++**, Addison-Wesley, Reading MA, 1996

Libros Avanzados de C++

- Flaymig, Practical Data Structures in C++
- Vilot/Booch, C++ Programmer's Power Pack
- Stroustrup, The Nature and Design of C++, Addison-Wesley, Reading, MA, 1994
- J. Coplien, Advanced C++, Addison-Wesley, Reading, MA, 1992

Problema de Nómina



Este ejemplo esta basado en la aplicación de nómina encontrado en el libro “Designing Object Oriented C++ Applications Using the Booch Method,” Robert C Martin, Prentice-Hall, 1995.

Declaración del Problema de Nómina

- El sistema consiste de una base de datos de empleados de una compañía y sus datos asociados, así como tarjetas de chequeo. Todos los empleados se identifican por un número ID único de empleado. El sistema debe pagar a cada empleado la cantidad correcta, a tiempo, por el método que ellos especifican.
- Algunos empleados trabajan pagándoseles una cuota por hora. Entregan tarjetas de chequeo diariamente que registran la fecha y número de horas trabajadas. Si alguno trabaja más de 8 horas, se les paga el 50% adicional de su cuota correspondientes por cada hora extra. Los empleados por hora reciben su pago cada semana.
- Otros empleados reciben un salario y aún así entregan sus tarjetas de chequeo diariamente, ya que contienen las horas trabajadas. De esta manera, el sistema pueda guardar un registro de las horas trabajadas. Estos empleados reciben su pago el último día laborable del mes.

Declaración del Problema de Nómina

- Algunos de los empleados asalariados reciben también una comisión basada en sus ventas. De este modo, entregan sus ordenes de compra, en donde se reflejan la fecha y monto de la venta. La cuota de comisión se determina para cada empleado, siendo esta del 10%, 15%, 25% o 35%. Los vendedores reciben su pago cada quince días.
- Inicialmente, el cheque de pago del empleado debe recogerse con el Pagador. Los empleados pueden cambiar su método de pago. Pueden obtener sus cheques de pago por correo a la dirección postal que deseen o pueden solicitar depósito directo en una cuenta de banco de su elección.

Declaración del Problema de Nómina

- El Administrador de Nómina mantiene actualizada la información del empleado. El Administrador de Nómina es responsable de agregar, borrar y cambiar la información de los empleados; tal como su nombre, dirección, método de pago y clasificación de pago (por hora, asalariado, comisionado)
- La aplicación de nómina se ejecutará cada viernes y el último día laborable del mes. Le pagará a los empleados correspondientes en ese día. El sistema sabrá en que fecha se les debe pagar a los empleados, entonces generará registros de pagos de la última vez que el empleado recibió su pago en la fecha especificada.

Solución del Problema Nómina

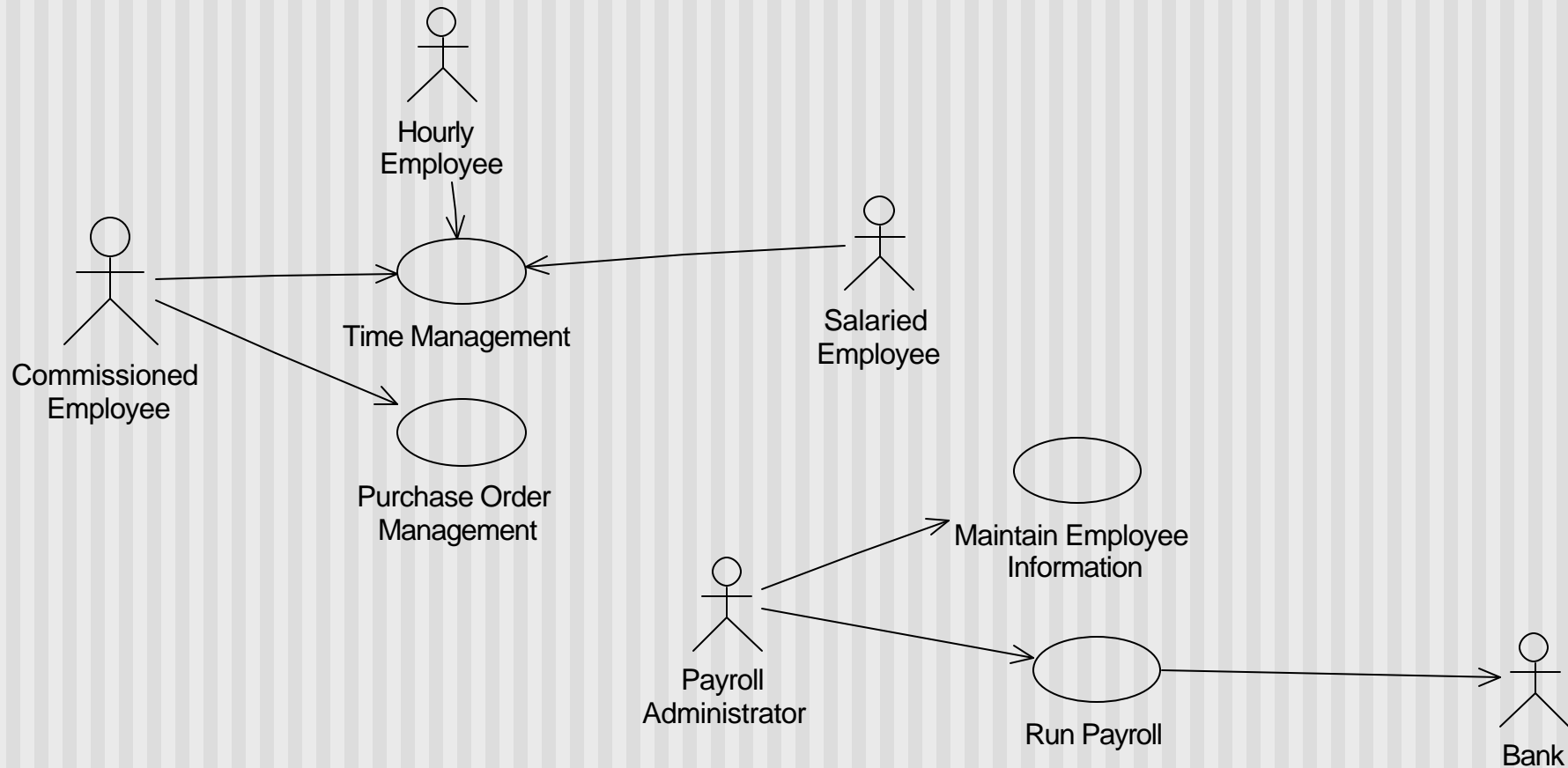


Este ejemplo esta basado en la aplicación de nómina encontrado en “Designing Object Oriented C++ Applications Using the Booch Method,” Robert C Martin, Prentice-Hall, 1995.

Ejercicio: Comportamiento del Sistema

- Usando el problema provisto por el instructor
 - Dibuje un diagrama de casos de uso
 - Escriba una definición para cada actor
 - Para un casos de uso
 - Escriba una breve descripción (dos sentencias máximo)
 - Escriba el flujo de eventos
 - Liste algunos escenarios posibles

Diagrama de Casos de Uso



Actores para el Sistema de Nómina

- Empleado por hora
 - Persona que recibe un pago por hora. Pago de tiempo extra (1 1/2 más de la cuota por hora) se recibe por todas las horas trabajadas en exceso de 40 horas por semana
- Empleado asalariado
 - Persona que se le paga un salario fijo
- Empleado comisionado
 - Empleado asalariado que también recibe una comisión por sus ventas
- Administrador de nómina
 - Persona responsable del mantenimiento de la información de los empleados. El administrador genera la nómina
- Banco
 - Entidad receptora de los pagos de nómina depositados en cuentas definidas por los empleados

Breves Descripciones

- Mantenimiento de Informacion de Empleado
 - Este caso de uso es iniciado por el Administrador de Nómina.
Proporciona la capacidad de agregar, modificar, borrar y/o revisar la informacion del empleado necesaria para el sistema de nomina
- Ejecucion de Nómina
 - Este caso de uso es iniciado por el Administardor de Nómina.
Proporciona un mecanismo para generar los pagos a los empleados

Breves Descripciones

■ Administración de Tiempo

- Este caso de uso es iniciado por cualquier tipo de Empleado.
Proporciona la capacidad de registrar, modificar, borrar y/o revisar horas trabajadas para una semana específica

■ Administración de Ordenes de Compra

- Este caso de uso es iniciado por un Empleado Comisionado.
Proporciona la capacidad de registrar, modificar, borrar y/o revisar ordenes de compra

Flujo de Eventos para el Caso de Uso de Mantenimiento de Información de Empleado

- 2.1 Pre-condiciones

Ninguna

- 2.2 Flujo Principal

Este caso de uso inicia cuando el Administrador de Nómina introduce su número id de empleado. El sistema verifica que el número id de empleado introducido sea válido. El sistema despliega las actividades disponibles y le pide al usuario que seleccione una: **ADD**, **DELETE**, **MODIFY**, **REVIEW**, o **QUIT**.

Si la actividad seleccionada es **ADD**, el A-1: Se desempeña un subflujo de New Employee.

Si la actividad seleccionada es **DELETE**, el A-2: Se desempeña un subflujo de Delete an Employee.

Si la actividad seleccionada es **MODIFY**, el A-3: Se desempeña un subflujo de Modify an Employee.

Flujo de Eventos para el Caso de Uso de Mantenimiento de Información de Empleado

Si la actividad seleccionada es **REVIEW**, el A-4: se desempeña un subflujo de Review an Employee.

Si la actividad seleccionada es **QUIT**, el caso de uso termina.

El usuario puede cancelar el caso de uso en cualquier punto en el flujo de eventos, en el que los cambios no toman lugar para el empleado que se está procesando y el caso de uso inicia de nuevo.

■ 2.3 Flujos alternos

A-1: Agregar un Empleado Nuevo

El sistema generará un número id de empleado y desplegará la pantalla del empleado. El sistema llenará el número id del empleado.

El usuario debe llenar la siguiente información: nombre y dirección del empleado. El usuario deberá escribir el tipo de empleado: por hora, asalariado o comisionado.

Flujo de Eventos para el Caso de Uso de Mantenimiento de Información de Empleado

Si el tipo de empleado es por hora, el usuario deberá proporcionar el rango de horas. Si el tipo de empleado es asalariado, el usuario deberá proporcionar el salario. Si el tipo de empleado es comisionado el usuario deberá proporcionar el salario y el porcentaje de comisión. El usuario deberá proporcionar el método de pago: recogerlo con el Pagador, depósito directo o correo. Si el método de pago es depósito directo, el usuario deberá proporcionar el número de cuenta del banco y su dirección. Si el método de pago es correo, el usuario deberá proporcionar la dirección postal. Toda la información es validada por el sistema (E-1) después de haberla introducido.

Cuando ya se introdujo toda la información, el usuario le pide al sistema que procese al empleado. El sistema guarda la información para su uso futuro (E-2) y el caso de uso inicia de nuevo.

Flujo de Eventos para el Caso de Uso de Mantenimiento de Información de Empleado

A-2: Borrar un empleado

El usuario introduce un número de id. El sistema lo valida (E-3) y despliega la información del empleado. El sistema pide al usuario que confirme el borrado. Si se confirma, el empleado es borrado del sistema. Si el borrado no es confirmado, la petición se cancela. El caso de uso inicia de nuevo.

A-3: Modificar un Empleado

El usuario introduce un número de id. El sistema lo valida (E-3) y despliega la información del empleado. El usuario actualiza los campos requeridos. El sistema verificará la información después de que se introdujo (E-1). Después de que se han hecho todos los cambios, el usuario le dice al sistema que procese al empleado. El sistema guarda la información para uso futuro (E-2) y el caso de uso inicia de nuevo.

Flujo de Eventos para el Caso de Uso de Mantenimiento de Información de Empleado

A-4: Revisar un Empleado

El usuario introduce un número de id. El sistema lo valida (E-2) y despliega la información del empleado. Cuando el usuario indica que ya terminó de revisar, el caso de uso inicia de nuevo.

■ 2.4 Flujos de excepción

E-1: Información de empleado invalida. El usuario sabe que se ha introducido información invalida. El usuario puede re-introducir la información para terminar el caso de uso.

E-2: El sistema no puede guardar la información del empleado. El usuario sabe porque no se puede guardar la información. El caso de uso inicia de nuevo.

E-3: La información del empleado no puede ser desplegada. El usuario sabe porque la información no puede ser desplegada. El caso de uso inicia de nuevo.

Flujo de eventos para el Caso de Uso de Ejecución de Nómina

■ 2.1 Pre-condiciones

Ninguna

■ 2.2 Flujo Principal

El caso de uso empieza cuando el Administrador de Nómina introduce su número id de empleado. El sistema verifica que el número id de empleado introducido sea valido y capaz de generar la nómina (E-1). El usuario introduce la fecha de nómina y le pide al sistema que genere la nómina. El sistema obtiene los datos de todos los empleados que deben recibir pago la fecha deseada (E-2). El sistema calcula el pago y las deducciones legales. Si el método de pago es recogerlo con el Pagador o Correo, el sistema imprime un cheque de pago. Si el método de pago es depósito bancario, el sistema creará una transacción bancaria. El caso de uso termina cuando todos los empleados que deban recibir pago en la fecha deseada hayan sido procesados.

Flujo de eventos para el Caso de Uso de Ejecución de Nómina

- 2.3 Flujos Alternos

Ninguno

- 2.4 Flujos de Excepción

E-1: Id invalido introducido. El usuario puede re-introducir un número id o terminar el caso de uso.

E-2: La informacion del empleado no puede ser obtenida. El usuario sabe porque no se puede obtener la información. El caso de uso inicia de nuevo desde el principio.

Flujo de Eventos para el Caso de Uso de Administración de Tiempo

■ 2.1 Pre-condiciones

Ninguna

■ 2.2 Flujo Principal

El caso de uso inicia cuando el Empleado introduce su número id de empleado. El sistema verifica que el número id de empleado introducido sea valido (E-1). El sistema despliega las actividades disponibles y le pide al usuario que seleccione una: **ADD**, **DELETE**, **MODIFY**, **REVIEW**, o **QUIT**.

Si la actividad seleccionada es ADD, el A-1: Se desempeña un subflujo de Add a New Timecard.

Si la actividad seleccionada es DELETE, el A-2: Se desempeña un subflujo Delete a Timecard.

Flujo de Eventos para el Caso de Uso de Administración de Tiempo

Si la actividad seleccionada es MODIFY, the A-3: Se desempeña un subflujo de Modify a Timecard.

Si la actividad seleccionada es REVIEW, el A-4: Se ejecuta un subflujo de Review a Timecard.

Si la actividad seleccionada es QUIT, el caso de uso termina.

El usuario puede cancelar el caso de uso en cualquier punto del flujo de eventos y el caso de uso inicia de nuevo.

Flujo de Eventos para el Caso de Uso de Administración de Tiempo

■ 2.3 Flujos alternos

A-1: Agregar una Tarjeta de tiempo nueva

El sistema desplegara la pantalla de la tarjeta de tiempo. El usuario debera llenar la siguiente informacion: numero id de empleado, fecha y horas trabajadas.

Cuando ya se introdujo toda la informacion, el usuario le pide al sistema que procese la tarjeta de tiempo. El sistema asocia la tarjeta de tiempo con el empleado correcto, guarda la informacion para uso futuro (E-2) y el casos de uso inicia de nuevo.

A-2: Borrar una Tarjeta de tiempo

El usuario introduce el numero id de empleado y la fecha. El sistema retrieves (E-3) y despliega la informacion de la tarjeta de tiempo. Se le pregunta al usuario si confirma el borrado. Si lo confirma, se borra la tarjeta de tiempo del sistema. Si no se confirma el borrado, se cancela la peticion. El casos de uso inicia de nuevo.

Flujo de Eventos para el Caso de Uso de Administración de Tiempo

A-3: Modificar Timecard

El usuario introduce el número de id del empleado y la fecha. El sistema obtiene (E-3) y despliega la información de timecard. El usuario actualiza el número de horas trabajadas y le pide al sistema que procese el timecard. El sistema guarda la información para uso futuro (E-2) y el caso de uso inicia de nuevo.

A-4: Revisar Timecard

El usuario introduce un número de id del empleado y una fecha. El sistema obtiene (E-3) y despliega la información de timecard. El usuario indica que ya finalizó la revisión y el caso de uso inicia de nuevo.

Flujo de Eventos para el Caso de Uso de Administración de Tiempo

■ 2.4 Flujos de Excepción

E-1: Id introducido no válido. El usuario puede re-introducir un número de id o finaliza el caso de uso.

E-2: El sistema es incapaz de guardar la información de timecard. El usuario sabe porque no se puede guardar la información. El caso de uso inicia de nuevo.

E-3: No se puede obtener la información de timecard. El usuario sabe porque no se pudo obtener la información. El caso de uso inicia de nuevo.

Flujo de Eventos para el Caso de Uso de Administración de la Orden de Compra

■ 2.1 Pre-condiciones

Ninguna

■ 2.2 Flujo Principal

Este caso de uso inicia cuando el empleado introduce su número de id. El sistema verifica que el número de id sea válido (E-1). El sistema despliega las actividades disponibles y pide que seleccione una: ADD, DELETE, MODIFY, REVIEW, o QUIT.

Si la actividad seleccionada es ADD, el A-1: Se desempeña un subflujo de agregacion de New Purchase Order.

Si la actividad seleccionada es DELETE, el A-2: Se desempeña un un subflujo de borrado de Delete a Purchase Order.

Flujo de Eventos para el Caso de Uso de Administración de la Orden de Compra

Si la actividad seleccionada es MODIFY, el A-3: Se desempeña un subflujo de modificación Modify a Purchase Order.

Si la actividad seleccionada es REVIEW, el A-4: Se desempeña un subflujo de revisión de Purchase Order.

Si la actividad seleccionada es QUIT, el caso de uso termina.

El usuario puede cancelar el caso de uso en cualquier punto del flujo de eventos, el caso de uso inicia de nuevo.

Flujo de Eventos para el Caso de Uso de Administración de la Orden de Compra

■ 2.3 Flujos alternos

A-1: Agregar New Purchase Order

El sistema despliega la pantalla de orden de compra. El usuario debe llenar la siguiente información: número de orden de compra, fecha, producto comprado, cantidad de la venta, nombre del cliente y dirección del cobro del cliente.

Cuando se introdujo toda la información, el usuario le pide al sistema que procese la orde. El sistema guarda la información para uso futuro (E-2) y el caso de uso inicia de nuevo.

Flujo de Eventos para el Caso de Uso de Administración de la Orden de Compra

A-2: Borrar una orden de compra

El usuario introduce un número de id de empleado y un número de orden de compra. El sistema obtiene (E-3) y despliega la información de la orden de compra. Se le pide al usuario que confirme el borrado. Si lo confirma, se borra la orden de compra del sistema. Si no se confirma, se cancela la petición. El caso de uso inicia de nuevo.

A-3: Modificar orden de compra

El usuario introduce un número de id de empleado y un número de orden de compra. El sistema obtiene (E-3) y despliega información de la orden de compra. El usuario actualiza los campos necesarios y le pide al sistema que procese la orden de compra. El sistema guarda la información para uso futuro (E-2) y el caso de uso inicia de nuevo.

Flujo de Eventos para el Caso de Uso de Administración de la Orden de Compra

A-4: Revisión de la orden de compra

El usuario introduce el número de id de empleado y el número de la orden de compra. El sistema obtiene (E-3) y despliega la información de la orden de compra. Cuando el usuario indica que ya termino de revisar, el caso de uso inicia de nuevo.

■ Flujos de Excepción

E-1: Id introducido no valido. El usuario puede re-introducir el número de id o finaliza el caso de uso.

E-2: El sistema es incapaz de guardar la información de la orden de compra. El usuario sabe porque no se pudo guardar la información. El caso de uso inicia de nuevo desde el principio.

E-3: La información de la orden de compra no puede ser obtenida. El usuario sabe porque no se pudo obtener la información. El caso de uso inicia desde el principio.

Algunos Escenarios para el Caso de Uso de Mantenimiento de la Información de Empleado

- Crear un empleado por hora
- Crear un empleado comisionado
- Crear un empleado asalariado
- Cambiar una categoria de pago de empleado
- Cambiar un método de entrega de pago de empleado
- Cambiar otra informacion de empleado
- Revisar informacion de empleado
- Borrar un empleado