Preámbulo.

Crisis del software.

Desde que las computadoras son utilizadas comercialmente en la década de los cincuenta, estas han ido mejorando sustancialmente año con año.

Por otra parte el software no ha llevado el mismo nivel de crecimiento y la demanda de software se ha incrementado y por lo tanto su costo no ha disminuido sino aumentado con el paso del tiempo.

Según [Pressman] las "fabricas del software" envejecieron:

- Desarrollos en sistemas de información hechos hace más de 25 años y que han sido alteradas durante su uso ya no las entiende nadie o si no han sido modificadas dificilmente alguien tratará de hacerlo porque no se tiene documentación y/o no se conoce el lenguaje de programación.
- Aplicaciones avanzadas de ingeniería que fueron construidos sin técnicas de desarrollo de software.
- Sistemas empotrados atrasados que no se actualizan debido a diversos factores.

Esta situación está cambiando un poco de manera obligada debido al problema del año 2000. Las empresas se ven en la necesidad de actualizar el software o morir. Sin embargo la mayoría de las empresas únicamente se encuentran "parchando" los sistemas con la justificación de que no hay dinero o tiempo para remplazar los programas.

En México debido a la nueva crisis (crisis recursivas) el gasto (público y privado) se redujo, aunque los cambios sobre el y2k deben seguir, por lo tanto los recortes son en compra de equipo y software nuevo. El gasto público para el 99 se redujo en 25 % (Diario Reforma).

Por otra parte muchas empresas se han cansado de los problemas que presenta el software y han empezado a disminuir los departamentos de informática y a contratar gente externa para que se encargue de algunas tareas. (Outsourcing).

Sin embargo, el software apenas ha logrado unas mejoras, pero el resultado general sigue resumiéndose en:

- La calendarización y la estimación de costos se realizan de manera inadecuada. (Casos famosos: Windows 95, compra de Ashto Tate (dbase) por Borland (Inprise).
- Problemas de calidad del software.
- Las expectativas del cliente (entiéndase usuario en todos sus niveles) no quedan satisfechas.

Estas situaciones son en gran medida las que han provocado lo que se conoce como "la crisis del software" y que tiene que ver con:

Problemas de productividad. No se construye software a la velocidad requerida.

Atención de la demanda. Hacia 1970 se estimaba que la demanda de software aumentaría entre el 11.5 y 17% anual, no pudiendo ser atendida y arrojando un déficit anual del 6 al 9.5% en E.U. (en realidad entre 1975 y 1985 crecio entre 21 y 23% anual)

Efectividad del esfuerzo. En la década de los setenta, el Departamento de Defensa de los Estados Unidos:

El mayor usuario de computadoras en el mundo informo de sus pagos en proyectos de cómputo.

Gasto el 48% de los recursos destinados a la contratación de proyectos de software que nunca recibió.

El 27% por sistemas que le fueron entregados pero que nunca utilizó.

El 21% por productos que le fueron entregados con errores importantes, y que tuvo que abandonar o reelaborar (14%) o modificar (7%).

Parálisis debido al mantenimiento. Yourdon entre otros ha considerado que en promedio las empresas gastan el 79% su presupuesto para desarrollo en el mantenimiento del software existente.

Para ser más precisos con el mantenimiento de software lo podemos dividir en :

Mantenimiento: cuando se corrigen los errores

Evolución: cuando se responde a cambios en los requerimientos.

Preservación, cuando se mantiene en operación un software viejo y decadente.

Problemas de costos. El costo del software se ha incrementado al contrario del precio del hardware.

En 1955, el costo del software era menor al 10% del costo del hardware

En 1975 la diferencia del costo era tres veces mayor que el hardware.

Para 1985 la diferencia se incremento 9 veces.

"Si la indsutria automovilística hubiera hecho lo que la industria de las computadoras en los últimos treinta años, un Rolls-Royce costaría dos dólares y rendiría un millón de millas." *Computerworld.*

Problemas de confiabilidad. El software es uno de los productos construidos por el hombre más susceptibles a fallas.

Si la industria del software fuera como la industria automotriz, la mayor parte de las compañías de software estarían hundidas en demandas.

En junio de 1962, el Mariner I se salió de curso y tuvo que ser destruido: el problema - que costo 18.5 millones de dólares-, se debió a un error en uno de los programas que guiaban la nave.

El 15 de enero de 1990, el sistema de larga distancia de la AT&T sufrió una falla catastrófica que dejó paralizada la mayor parte de la red telefónica nacional de Estados Unidos durante nueve horas. El problema fue en el software de enrutamiento.

El diseño deficiente de la interfaz con el usuario fue el factor principal de la identificación incorrecta de una imagen de radar, que resultó en el abatimiento del vuelo iraní y la muerte de sus 290 pasajeros.

Una causa frecuente de los problemas son una serie de mitos que rodean al software. Estos mitos se pueden clasificar en tres tipos: administrativos, del cliente y del programador. Algunos de ellos:

Mito administrativo: "Mi gente cuenta con las herramientas más avanzadas en desarrollo de software y las computadoras más actuales".
Realidad: Se necesita mucho más que las computadoras más avanzadas. Existen herramientas de ingeniería de software (CASE) que son muy efectivas, pero que con frecuencia no se aprovechan.

Mito administrativo: "Si me atraso en mis objetivos, podré contratar más programadores para cumplirlos".
Realidad: El software no es un proceso mecánico como la manufactura; el incremento de personal hace más tardado el proyecto pues se requiere de tiempo para capacitar a los nuevos integrantes.

Mito del cliente: "Es suficiente expresar la idea general del sistema, después se darán los detalles".
Realidad: Es un error despreciar los parámetros, características y requisitos del sistema en el proceso de diseño, pues a la larga es mucho más costoso. Es aconsejable establecer mecanismos de comunicación eficientes entre el cliente y el programador.

Mito del cliente: "Los requerimientos cambian constantemente, pero éstos se pueden integrar fácilmente ya que el software es flexible".
Realidad: Es cierto que los requerimientos cambian y que el software es flexible; sin embargo, los cambios son costosos, sobre todo en etapas finales del proyecto. Por lo tanto, se deben tomar en cuenta todos los detalles posibles, así como proveer al sistema con la estructura necesaria para soportar los cambios que puedan suscitar.

Mito del programador: "Una vez que se escriba el código y que el programa funcione, se cumplió con el trabajo"

Realidad: Alguien dijo: “cuanto más rápido te pongas a codificar, más tiempo te tardarás en acabar el sistema”. Las estadísticas nos dicen que del 50 al 70% del esfuerzo se emplea después de que se entregó el sistema.

Mito del programador: “Hasta que no corra el programa, no podré conocer su calidad”.
Realidad: La calidad del software empieza desde el diseño. Si es consistente indica que se comenzó con el pie derecho y esto no es latente hasta que el software se encuentre en forma ejecutable.

Mito del programador: “Lo que puede distribuirse de un sistema exitoso es el programa ejecutable”.
Realidad: El programa ejecutable sólo es parte del paquete que también incluye manuales de usuario, referencias y guías para el proceso de mantenimiento.

Complejidad del software.

"La complejidad del software es una propiedad inherente, no accidental". Fred Brooks.

Esta complejidad se deriva de cuatro elementos:

1. La complejidad del dominio del problema.
2. La dificultad de administrar el proceso de desarrollo de software.
3. La naturaleza abstracta del software.
4. El comportamiento discreto del software.

La complejidad del dominio del problema.

Los problemas que se tratan de resolver con el software, a menudo involucran elementos de complejidad insalvable, en los cuales encontramos requerimientos que compiten entre si. (aún contradictorios). *Ejemplo: considerese un sistema complejo -de control de un avión- y hay que añadirle los requerimientos no funcionales como rendimiento, costo, etc.*

La comunicación entre usuarios y desarrolladores:

El usuario encuentra difícil expresar sus ideas.

Los usuarios solo tienen ideas poco claras de lo que esperan de un sistema de software.

Tanto los usuarios como los desarrolladores, tienen poca experiencia en el dominio del otro.

Ambos tienen diferentes perspectivas acerca de la naturaleza del problema, y por lo tanto diferentes formas de resolverlo.

Renuencia de los usuarios al cambio.

Los sistemas de software están siempre dentro de otros sistemas , de los cuales heredan su complejidad y a cuyos cambios debe adaptarse.

Dificultad en la administración del proceso.

Los proyectos de software presentan dificultades administrativas que no se dan en los demás proyectos de ingeniería.

El desarrollo de software es además, un desafío intelectual, y en esa medida, es difícil estimar su costo y realizarlo trabajando en equipo.

Naturaleza abstracta.

El software es una representación abstracta de un proceso.

Los desarrolladores tienden a construir casi todos los bloques del sistema por su propia cuenta.

El desarrollo de software sigue siendo una labor intensiva de trabajo.

Ejemplo: construcción de un un dispositivo mecanico, no reinventamos las tuercas.

No todas las empresas valoran el software como un activo de la misma.

Comportamiento Discreto.

La mayor parte de los sistemas de ingeniería pueden ser modelados por medio de funciones continuas. Los sistemas de software tienen un comportamiento discreto.

Cuando hablamos de una función continua hablamos de sistemas que no esconden ninguna sorpresa. Pequeños cambios en las entradas, siempre causaran pequeños cambios en las salidas.

Dentro de una aplicación, existen cientos o aún miles de variables, y más de un hilo de control. El conjunto de variables y sus valores constituyen el estado del sistema.

Los sistemas discretos tienen un número finito de posibles estados. Pero en una gran aplicación, se sucede una combinación combinatoria, que hace que este número sea muy grande.

Cada evento externo al sistema de software, tiene la capacidad de colocar al sistema en un nuevo estado.

El límite de Miller.

La capacidad de los sistemas de software frecuentemente exceden la capacidad del intelecto humano.

El psicólogo George Miller dice que el ser humano solamente puede manejar, procesar o mantener la pista de aproximadamente siete objetos, entidades o conceptos a la vez.

En problemas con múltiples elementos, arriba de entre 7 y 9 elementos los errores en los procesos crecen desorbitadamente.

¿Cómo se enfrenta la complejidad?

¿Si los sistemas de software son complejos, y nuestra capacidad de hacer frente a cuestiones complejas es limitada, como hemos podido construir software?

Por medio de la **descomposición**.

La técnica para enfrentar la complejidad es conocida desde los tiempos antiguos: *divide y vencerás.*

Cuando se diseña un sistema de software, es esencial descomponerlo en pequeñas partes.

De esta manera satisfacemos la capacidad del canal de conocimiento humano.

Descomposición algorítmica

Descomposición orientada a objetos.

Descomposición algorítmica.

La descomposición algorítmica se aplica para descomponer un gran problema en pequeños problemas.

La unidad fundamental de este tipo de descomposición es el subprograma.

El programa resultante toma la forma de un árbol, en el que cada subprograma realiza su trabajo, llamando ocasionalmente a otro programa.

Este tipo de descomposición, surgió en los años 60's y 70's. Tiene una fuerte influencia en lenguajes como FORTRAN y COBOL.

También se le conoce con el nombre de diseño estructurado top-down.

Descomposición orientada a objetos.

El mundo es visto como un conjunto de entidades autónomas, que al colaborar muestran cierta conducta.

Los algoritmos no existen de manera independiente, estos están asociados a los objetos.

Cada objeto exhibe un comportamiento propio bien definido, y además modela alguna entidad del mundo real.

Comparación entre descomposición algorítmica y orientada a objetos.

La forma algorítmica muestra el orden de los eventos.

La forma orientada a objetos enfatiza las entidades que causan una acción.

La forma orientada a objetos surge después de la algorítmica.

Lo más recomendable es aplicar una descomposición orientada a objetos, la cual arrojara una estructura, para continuar con una descomposición algorítmica.

Conceptos básicos de objetos.

La programación tradicional separa los datos de las funciones, mientras que la programación orientada a objetos define un conjunto de objetos donde se combina de forma modular los datos con las funciones.

Aspectos principales:

1. Objetos.

   El objeto es la entidad básica del modelo orientado a objetos.

   El objeto integra una estructura de datos (atributos) y un comportamiento (operaciones).

   Se distinguen entre sí por medio de su propia identidad, aunque internamente los valores de sus atributos sean iguales.

2. Clasificación.

   Las clases describen posibles objetos, con una estructura y comportamiento común.

   Los objetos que contienen los mismos atributos y operaciones pertenecen a la misma clase.

   La estructura de clases integra las operaciones con los atributos a los cuales se aplican.

3. Instanciación.

   El proceso de crear objetos que pertenecen a una clase se denomina instanciación. (El objeto es la instancia de una clase).

   Pueden ser instanciados un número indefinido de objetos de cierta clase.

4. Generalización.

   En una jerarquía de clases, se comparten atributos y operaciones entre clases basados en la generalización de clases.

   La jerarquía de generalización se construye mediante la herencia.

   Las clases más generales se conocen como superclases. (clase padre)

   Las clases más especializadas se conocen como subclases (clases hijas).

   La herencia puede ser simple o múltiple.

5. Abstracción.

   La abstracción se concentra en lo primordial de una entidad y no en sus propiedades secundarias.

   Además en lo que el objeto hace y no en cómo lo hace.

   Se da énfasis a cuales son los objetos y no cómo son usados. *Logrando el desarrollo de sistemas más estables.*

6. Encapsulación.

   Encapsulación o encapsulamiento es la separación de las propiedades externas de un objeto de los detalles de implementación internos del objeto.

   Al separar la interfaz del objeto de su implementación, se limita la complejidad al mostrarse sólo la información relevante.

   Disminuye el impacto a cambios en la implementación, ya que los cambios a las propiedades internas del objeto no afectan su interacción externa.

   Se protege al objeto contra posibles errores, y se permite hacer extensiones futuras en su implementación.

   Se reduce el esfuerzo en migrar el sistema a diferentes plataformas.

7. Modularidad.

   El encapsulamiento de los objetos trae como consecuencia una gran modularidad.
   Cada módulo se concentra en una sola clase de objetos.
   Los módulos tienden a ser pequeños y concisos.
   La modularidad facilita encontrar y corregir problemas.
   La complejidad del sistema se reduce facilitando su mantenimiento.

8. Extensibilidad.

   La extensibilidad permite hacer cambios en el sistema sin afectar lo que ya existe.
   Nuevas clases pueden ser definidas sin tener que cambiar la interfaz del resto del sistema.
   La definición de los objetos existentes puede ser extendida sin necesidad de cambios más allá del propio objeto.

9. Polimorfismo.

   El polimorfismo es la característica de definir las mismas operaciones con diferente comportamiento en diferentes clases.
   Se permite llamar una operación sin preocuparse de cuál implementación es requerida en que clase, siendo responsabilidad de la jerarquía de clases y no del programador.

10. Reusabilidad de código.

    La orientación a objetos apoya el reuso de código en el sistema.
    Los componentes orientados a objetos se pueden utilizar para estructurar librerías resuables.
    El reuso reduce el tamaño del sistema durante la creación y ejecución.
    Al corresponder varios objetos a una misma clase, se guardan los atributos y operaciones una sola vez por clase, y no por cad objeto.
    La herencia es uno de los factores más importantes contribuyendo al incremento en el reuso de código dentro de un proyecto.
    Nuevas subclases de clases previamente definidas se pueden crear en el reuso de una clase, pudiéndose crear nuevas operaciones, o modificar las ya existentes.

Comparación con el análisis y diseño estructurado.

- El análisis y diseño estructurado se concentra en especificar y descomponer la funcionalidad del sistema total. *Los cambios a los requisitos cambiaran totalmente la estructura del sistema.*
- Ambas metodologías cuentan con modelos similares: objetos, dinámico y funcional.
- OMT está dominado por el modelo de objetos, en el modelo estructurado domina el funcional y el de objetos es el menos importante.
- El modelo estructurado se organizan alrededor de procedimientos. Los modelos OO se organizan sobre el concepto de objetos.
- Comúnmente los requisitos cambian en un sistema y ocasionan cambios en la funcionalidad más que cambios en los objetos. *El modelo estructurado puede ser útil en sistemas en que las funciones son más importantes y complejas que los datos.*
- En el modelo estructurado la descomposición de un proceso en subprocesos es bastante arbitraria. *En el modelo de objetos diversos desarrolladores tienden a descubrir objetos similares, incrementando la reusabilidad entre proyectos.*
- El enfoque orientado a objetos integra mejor las bases de datos con el código de programación.

Introducción.

En la década de los 80's la programación orientada a objetos comenzó a entrar a las empresas, pero es a hasta finales de la década de los 80 cuando surgen infinidad de metodologías de análisis y diseño orientado a objetos. Es decir, el análisis que existía anteriormente - análisis estructurado- no se adecuaba del todo a la POO.

Aparecieron algunas propuestas apoyadas por su correspondiente bibliografía. Citaremos las principales:

- Sally Shlaer y Steve Mello. Publican en 1989 y 1991 sobre análisis y diseño. Su enfoque se conoce como Diseño Recursivo.
- Peter Coad y Ed **Yourdon** en 1991 escribieron con un enfoque hacia los métodos ligeros orientados a prototipos de Coad.
- La comunidad de Smalltalk de Portland, Oregon, aportó el Diseño Guiado por la Responsabilidad (Responsibility-Driven Design) (1990) y las tarjetas de clase responsabilidad (Class-Responsibility-Collaboration) (CRC) (1989).
- Grady **Booch** trabajando para Rational Software, desarrollando sistemas en Ada. 1994 y 1995.
- Jim **Rumbaugh** trabajando en laboratorios de investigación de General Electric. Publica un libro de la Técnica de Modelado de Objetos (Object Modeling Technique) (OMT). 1991 y 1996.
- Ivar **Jacobson** trabajando para Ericcson en conmutadores telefónicos, introdujo el concepto de Casos de Uso (use cases). 1994 y 1995.

El caso es que existían múltiples metodologías, cada una con su grupo de

seguidores. Muchas se parecen entre sí, con distintas notaciones o términos distintos.

Algo de historia más reciente:

En la OOPSLA '94 se conoce que Rumbaugh deja a General Electric para trabajar junto con Booch en Rational Software, con la intención de unificar sus métodos.

RATIONAL SOFTWARE

En el mismo evento en 1995, se anuncia que Rational Software había comprado Objectory y que Jacobson se uniría a su equipo de trabajo.

En 1996 Grady Booch, Jim Rumbaugh e Ivar Jacobson ("los tres amigos") proponen su método al que le llaman *Unified Modeling Language* (**UML,** Lenguaje Unificado de Modelado).

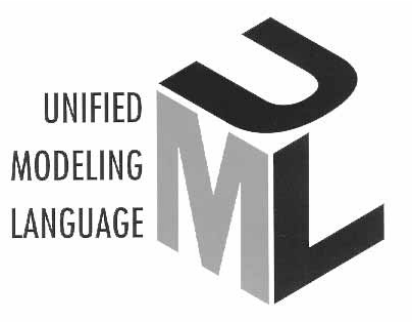
Grady Booch

Jim Rumbaugh

Ivar Jacobson.

En conclusión, **UML** unifica esencialmente los métodos de Booch, Rumbaugh (OMT) y Jacobson.

En la actualidad, UML se encuentra aceptado como un estándar por el ***OMG*** *(Object Management Group* o Grupo de administración de objetos).



UML es un lenguaje de modelado y no un método. Un método consistiría en un lenguaje y en un proceso para modelar.

El **lenguaje de modelado** es la notación que usan los métodos para expresar los diseños. El **proceso** es la sugerencia sobre los pasos que debemos seguir para lograr el análisis y diseño.

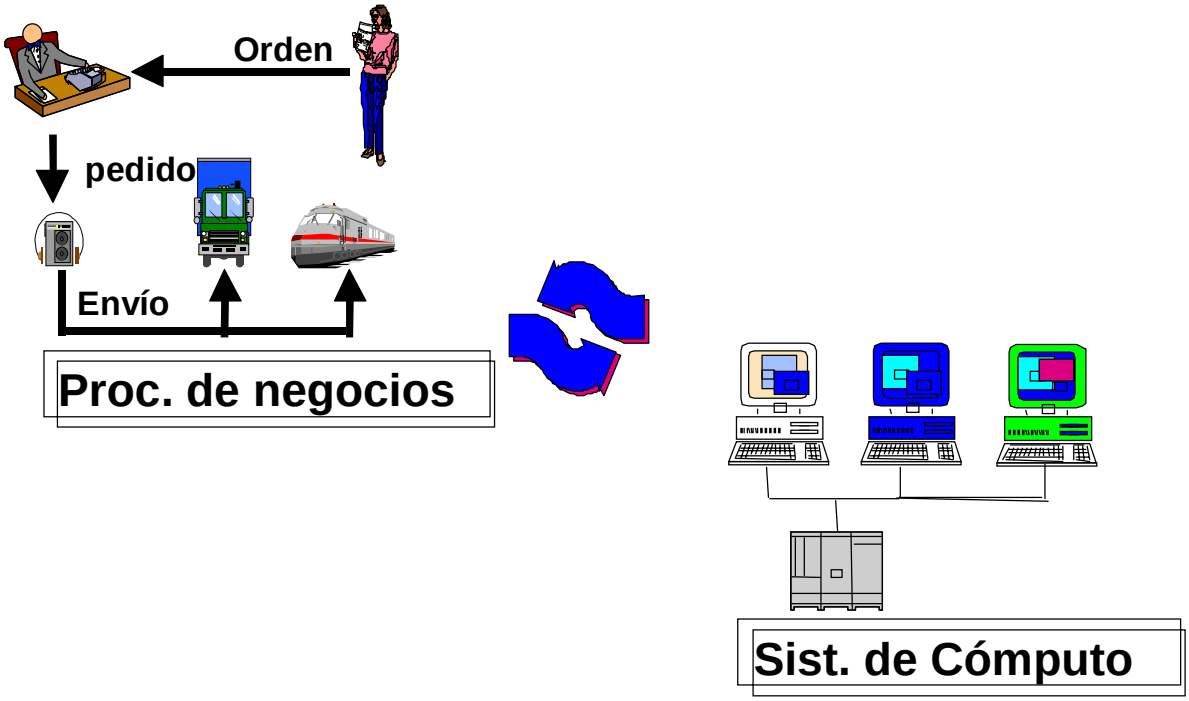
Actualmente Booch, Jacobson y Rumbaugh trabajan en la creación de un proceso unificado llamado anteriormente **Objectory**. Ahora se conoce como R*ational Unified Process* (Proceso Unificado Racional).

Modelado Visual.

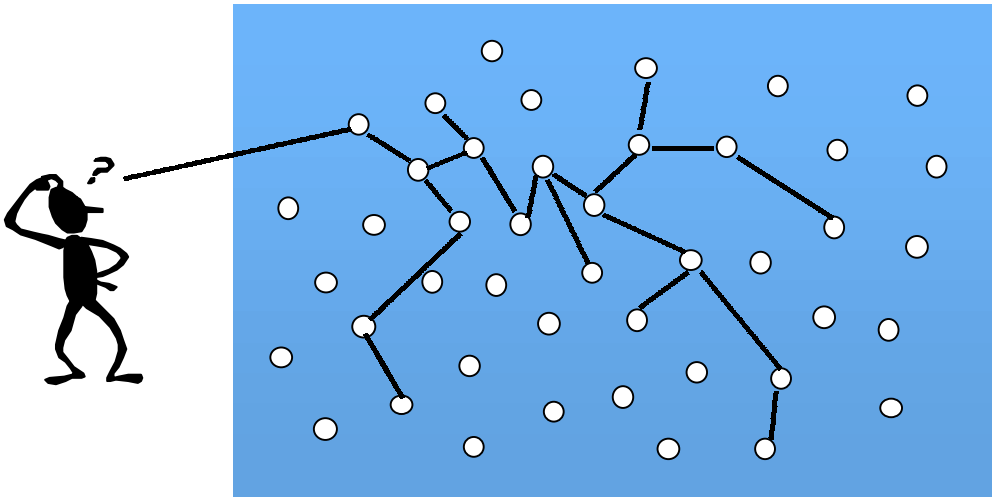
El Modelado Visual es el modelado de una aplicación usando notaciones gráficas.

"El modelado captura las partes esenciales de un sistema".

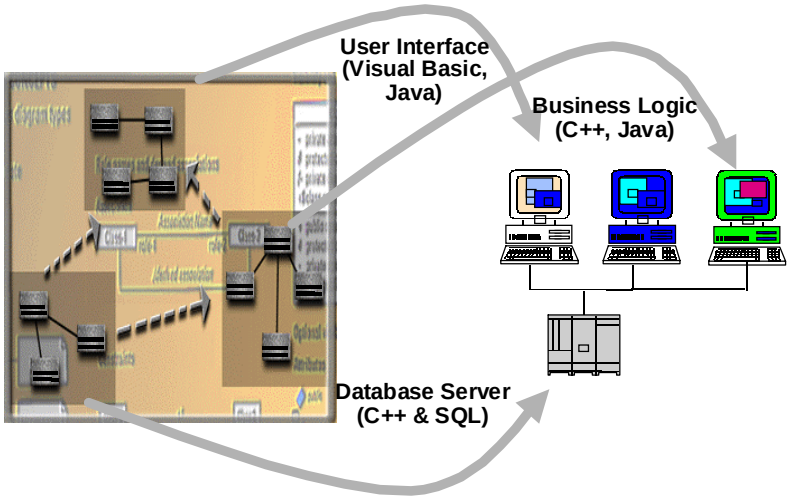
Rumbaugh



El Modelado Visual captura los Procesos de Negocios.



- Se utiliza para capturar los procesos de negocios desde la perspectiva del usuario.
- El Modelado Visual se utiliza para analizar y diseñar una aplicación, distinguiendo entre los dominios del negocio y los dominios de la computadora.
- Ayuda a reducir la complejidad. *Recordar el límite de Miller*.
- El Modelado Visual se realiza de manera independiente al lenguaje de implementación.



- Promueve el reuso de componentes.

UML.

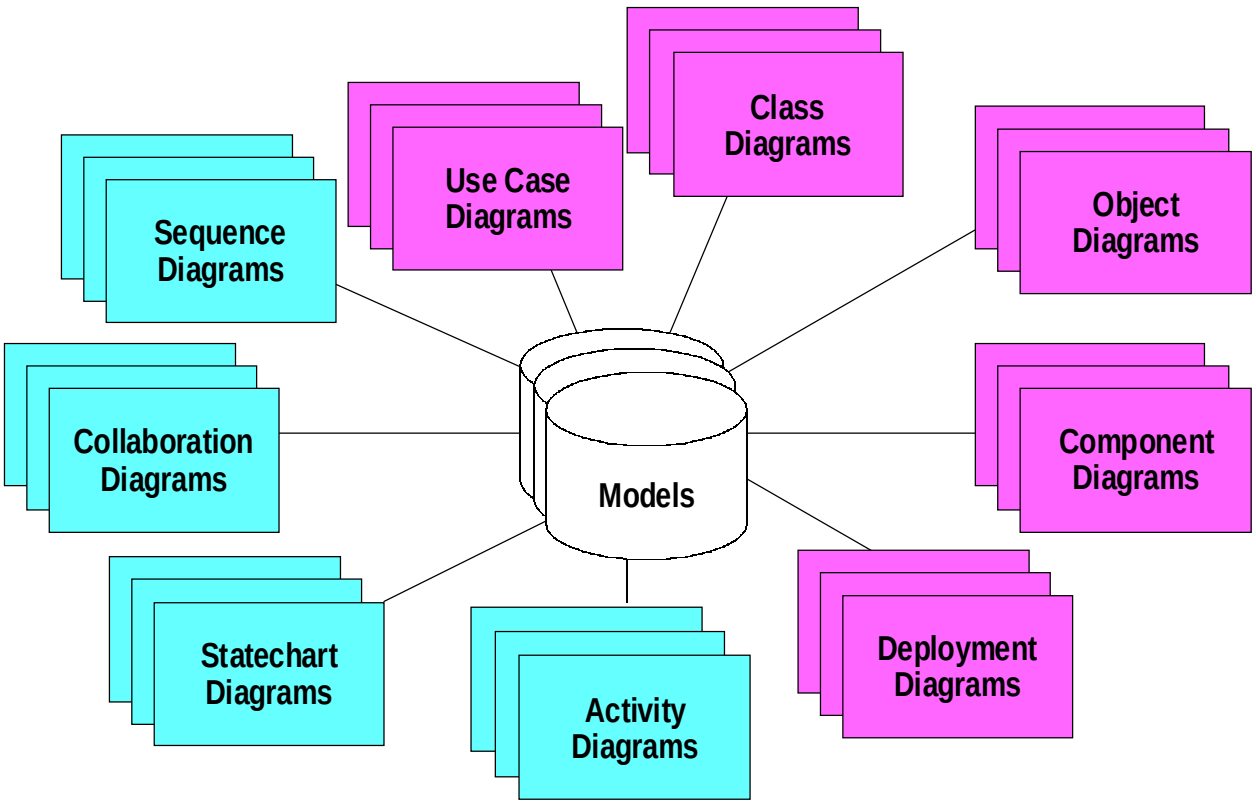
Pregunta obligada: ¿qué es UML?

Siglas de *Unified Modeling Language*[1], resulta de la unificación de los principales métodos de análisis y diseño orientado a objetos.

Es un **lenguaje de modelado** y no un método. Un lenguaje de modelado define la notación que es utilizada por los métodos para representar los diseños.

El método o proceso sería los pasos a seguir para llevar a cabo el análisis y diseño.

UML debe en parte su gran aceptación a que no incluye un proceso como parte de su propuesta.



[1] Lenguaje Unificado de Modelado

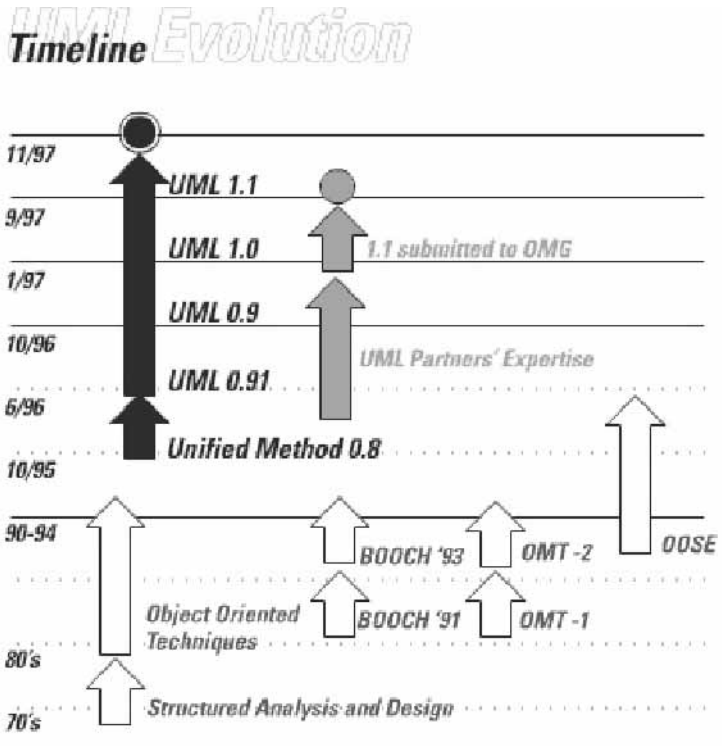
"Un modelo es una descripción completa de un sistema desde una perspectiva particular".

Rational

UML combina lo mejor de:

- Conceptos de Modelado de Datos (Diagramas Entidad Relación).
- Modelado de Negocios.
- Modelado de Objetos.
- Modelado de Componentes.

UML es un estándar de **OMG** (*Object Management Group*) a partir de noviembre de 1997, para la visualización, especificación, construcción y documentación de sistemas de software.



Puede ser usado con cualquier proceso, a lo largo del desarrollo del ciclo de vida, y de manera independiente de la tecnología de implementación.

UML puede ser usado para:

- Desplegar los **límites de un sistema** sus principales funciones mediante casos de uso y actores.
- Representar la **estructura estática** de un sistema usando diagramas de clases.
- Modelar los **límites de un objeto** con diagramas de estados.
- Mostrar la **arquitectura de la implementación** física con diagramas de componentes y de emplazamiento o despliegue.

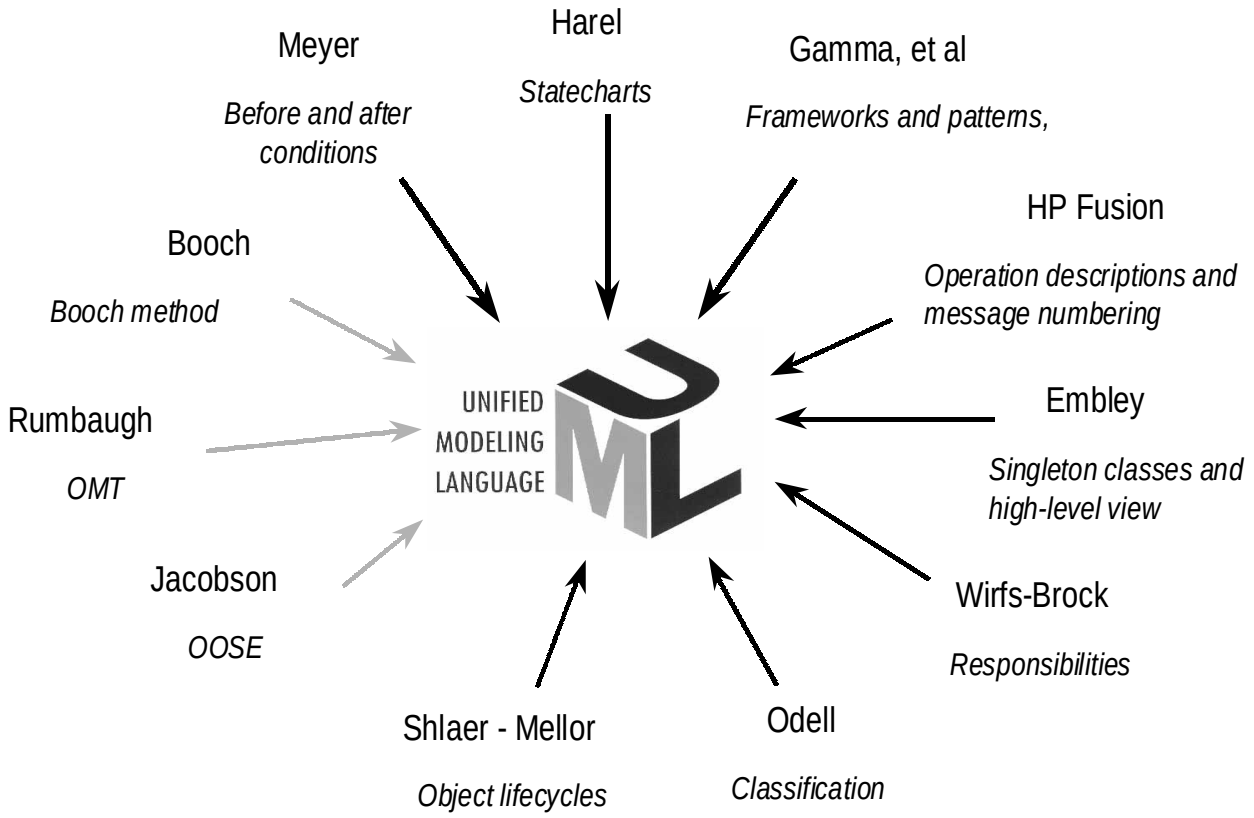
UML conviene por:

- Ser un **estándar abierto**.
- Soporta la **totalidad** del ciclo de vida de desarrollo del software.
- Soporta **diversas** áreas de aplicación.
- Esta basado en la **experiencia** y **necesidades** de la comunidad de usuarios.
- Es soportado actualmente por diversas herramientas.

Asociados de UML

- Rational Software Corporation
- Hewlett-Packard
- I-Logix
- IBM
- ICON Computing
- Intellicorp
- MCI Systemhouse
- Microsoft
- ObjecTime
- Oracle
- Platinum Technology
- Taskon
- Texas Instruments/Sterling Software
- Unisys

Contribuciones.



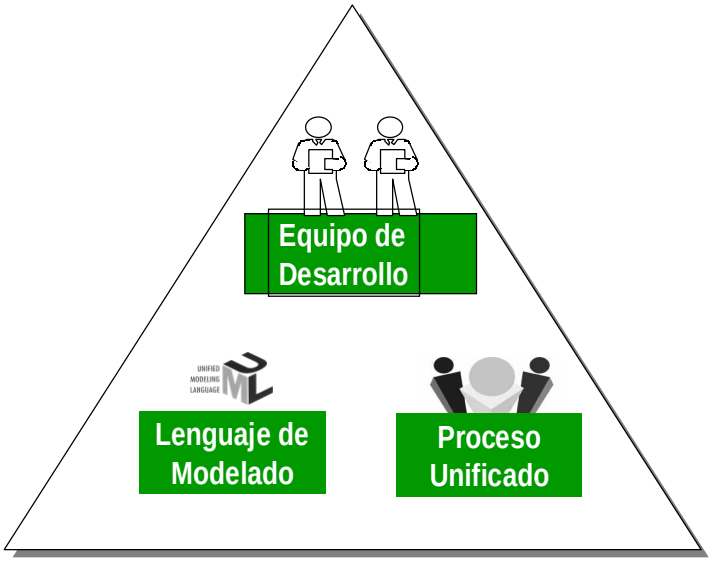
Proceso Unificado Racional.

Como es de suponer, es necesario un proceso que describa la serie de pasos a seguir para llegar al resultado final.

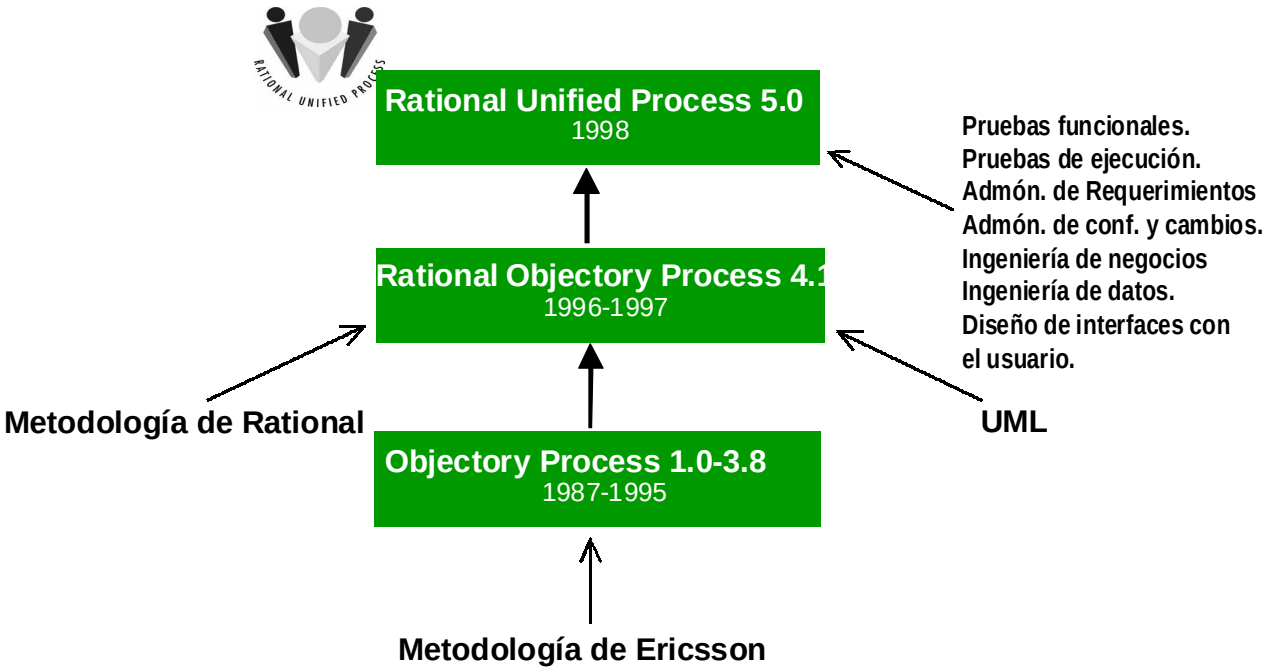
> Un **proceso** define **Quien** está haciendo **Que**, **Cuando** lo hace, y **Como** hacerle para alcanzar un objetivo.

Aunque el curso no abarcará formalmente el Proceso Unificado Racional (*Rational Unified Process*), se presentarán algunas ideas básicas.

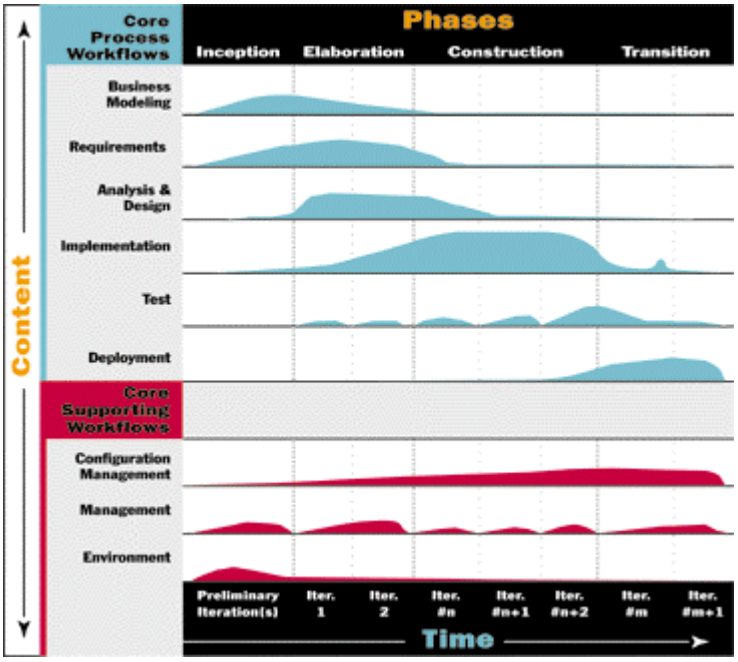
En principio, sabemos que el lenguaje de modelado visual no es suficiente para realizar el modelo de un sistema. Es necesario:



Evolución del Proceso Unificado.



Fases del ciclo de vida.



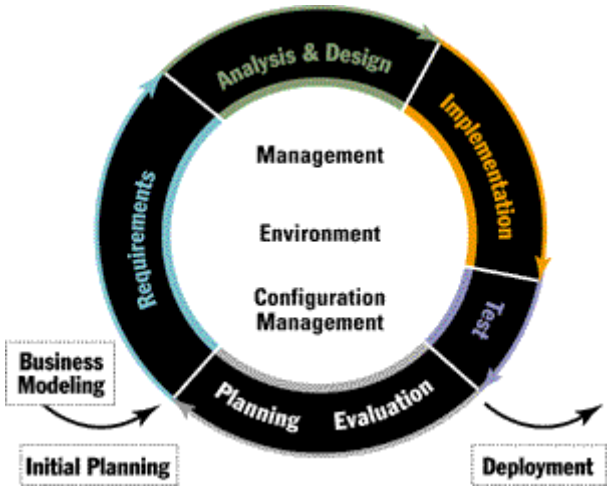
El RUP organiza a los proyectos en términos de flujos de trabajo y fases, las cuales consisten de una o más iteraciones. En cada iteración, el énfasis en cada flujo de trabajo variará a lo largo del ciclo de vida.

Inicio (*inception)*. Define el alcance del proyecto y los procesos del desarrollo de negocios.

Elaboración. Planeación del proyecto, especificación de características y la arquitectura base.

Construcción. Construcción del proyecto.

Transición. Transición del producto a los usuarios.



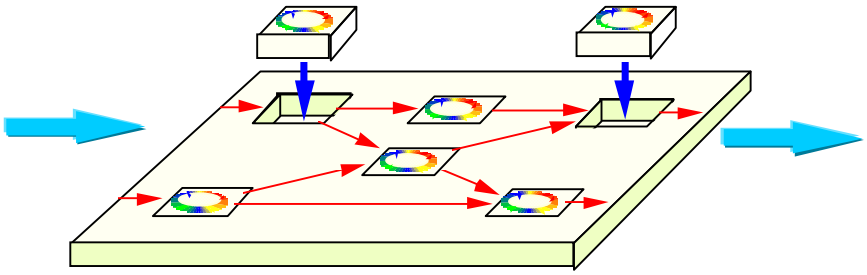
Cada iteración del ciclo del proyecto iniciará con un plan, el cual será llevado a cabo y concluido con una evaluación para ver que tanto se han cumplido los objetivos.

Contemplar a cada iteración como un miniproyecto. Se hace el análisis, diseño, codificación, pruebas y evaluación de cada iteración.

El propósito de este tipo de proceso es reducir el riesgo.

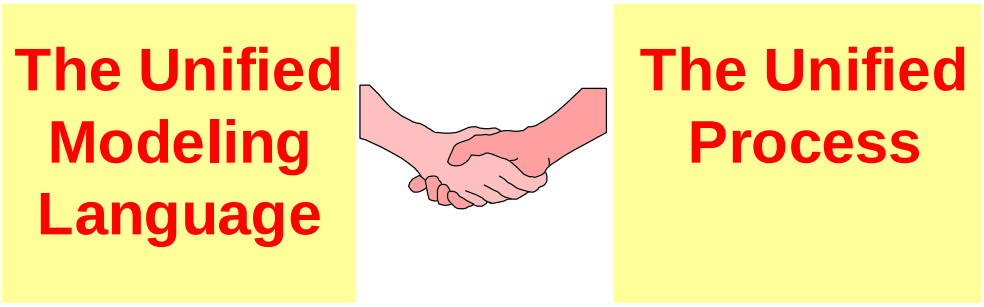
En conclusión el Proceso Unificado:

- Es un esqueleto del proceso a desarrollar.



- Iterativo e incremental.
- Maneja Casos de Uso.
- Es diseñado para ser flexible y extendible:
  - Permite una variedad de estrategias de ciclos de vida.
  - Elegir que "artefactos" producir.
  - Define actividades y trabajadores.
- **No** es un Proceso Universal.

Dos partes de un Conjunto Unificado.



Análisis de Requerimientos.

Un proyecto difícilmente puede ser exitoso sin una especificación **correcta** y **exhaustiva** de los requerimientos.

Los **requerimientos** son una descripción de las necesidades o deseos de un producto.

Existen diversos métodos para la obtención de los requerimientos. Se mencionan algunos puntos que ayudan a definirlos[2]:

- **Panorama general.**

  Una descripción textual del objetivo principal del sistema de software.

- **Clientes**.

  Definición del cliente o clientes del sistema.

- **Metas**.

  Especificar los principales fines del sistema de software.

- **Funciones del sistema.**

  Son las operaciones que tendrá que realizar el sistema. El objetivo es irlas identificando de manera general. Es común que resulten oraciones del tipo:

  El sistema deberá hacer X.

  Donde X es la descripción de la función.

Una vez identificada una función, es útil clasificarla para poder establecer prioridades.

Una clasificación sería[2]:

Categoría	Significado
Evidente	Debe realizarse, y el usuario debería saber que se ha realizado
Oculta	Debe realizarse, aunque no es visible para los usuarios. Las funciones ocultas a menudo se omiten durante el proceso de obtención de los requerimientos.
Superflua	Opcionales; su inclusión no repercute significativamente en el costo ni en otras funciones.

Formato de ejemplo:

Referencia	Función	Categoría
R1.1	*Descripción textual*	Evidente

- **Atributos del sistema.**

  Son cualidades no funcionales que a menudo se confunden con las funciones.

  Se refiere a las características del sistema. Ejemplos:

  Facilidad de uso. Tolerancia a fallas. Tiempo de respuesta.

  Interfaz. Costos. Plataformas.

  Se catalogan como obligatorias u opcionales.

Pueden ser para todo el sistema o definirse para cada función. Si se definen para todo el sistema conviene especificar con cuales funciones se relacionan.

Formato de ejemplo, añadiendo los atributos a las especificaciones de las funciones:

Ref.	**Función**	**Categoría**	**Atributo**	**Detalles y restricciones**	**Categoría**
R1.1	*Descripción textual*	Evidente	*Descripción*	Especificaciones del atributo	Obligatoria u opcional

Es conveniente añadir únicamente aquellos que se relacionen directamente con las funciones. Pueden quedar atributos relacionados con todo el sistema en una tabla separada.

Casos de Uso.

Los casos de uso se utilizan para **mejorar** la comprensión de los requerimientos.

> *"Un caso de uso es un documento narrativo que describe la secuencia de eventos de un actor que utiliza un sistema para completar un proceso."*
>
> ***Jacobson****[1]*

> *" Un caso de uso es una secuencia de acciones que dan un resultado observable para un actor particular"*
>
> ***Jacobson****[10]*

Los casos de uso son historias o casos de utilización de un sistema; no son exactamente los requerimientos ni las especificaciones funcionales.

- Capturan la funcionalidad del sistema desde la perspectiva del usuario.
- Muestran una interacción típica entre un usuario y un sistema de cómputo.
- Especifican el contexto de un sistema.
- Capturan los requerimientos de un sistema.
- Validan la arquitectura del sistema.
- Manejan el desarrollo y genera casos de prueba.
- Desarrollados por analistas y expertos del dominio del problema

Un caso de uso es una descripción de un proceso de principio a fin relativamente amplia, descripción que suele abarcar muchos pasos o transacciones; normalmente no es

un paso ni una actividad individual del proceso.

Ejemplos de procesos:

- Retirar efectivo de un cajero automático.
- Ordenar un producto.
- Registrar los cursos que se imparten en la escuela.
- Verificar la ortografía de un documento.
- Realizar una llamada telefónica.

Metas e interacciones.

Es importante distinguir entre metas de usuario e interacciones del sistema:

Interacciones con el sistema. Implican casos de uso que reflejan cosas que el usuario hace con el sistema, en lugar de lo que el usuario trata de conseguir. Ejemplos: "define estilo", "cambia estilo", "mueve un estilo de documento a otro".

Metas del usuario. Se refiera a las intenciones del usuario y no a como lo realizaría. Ejemplo: "garantizar el formateo consistente de un documento", "hacer que el formato de un documento sea igual que el otro."

Si se empieza por crear los casos de uso de **interacciones con el sistema**, se pasarán por alto otras maneras creativas de cumplir con mayor eficacia los objetivos del usuario.

Es recomendable primero ocuparse de las metas del usuario, tratando de encontrar casos de uso que cumplan con las mismas.

Estructura de los casos de uso.

Aunque forman parte de UML no se propone un formato rígido, para que se adapte a las necesidades del equipo de desarrollo.

Es común que se hable de dos niveles de casos de uso:

- Casos de uso de alto nivel.
- Casos de uso expandidos.

Casos de uso de alto nivel.

Describe un proceso de manera breve, formado por unos cuantos enunciados.

Usados durante el examen inicial de los requerimientos, a fin de entender rápidamente el grado de complejidad y de funcionalidad del sistema.

Formato de ejemplo:

Caso de uso: Nombre del caso de uso

Actores: Lista de actores, indicando quien inicia el caso de uso.

Tipo: Tipo de caso de uso.

Descripción: Breve descripción textual.

Ejemplo:

Se presenta el caso de uso de comprar artículos en una tienda con terminales de punto de venta.

Caso de uso: Comprar Productos.

Actores: Cliente, Cajero

Tipo: Primario.

Descripción: Un Cliente llega a la caja registradora con los artículos que comprará. El Cajero registra los artículos y cobra el importe. Al terminar la operación, el Cliente se marcha con los productos.

Casos de uso expandidos.

Describe el proceso más a fondo que el de alto nivel.

La diferencia básica es que consta de una sección que describe el **curso normal de los eventos**, paso por paso.

Conviene usarlo para describir los casos más importantes y de mayor influencia.

Formato de ejemplo:

Caso de uso: Nombre del caso de uso

Actores: Lista de actores, indicando quien inicia el caso de uso.

Propósito: Intención del caso de uso.

Resumen: Repetición del caso de uso de alto nivel o alguna síntesis similar.

Tipo: Tipo de caso de uso.

Referencias cruzadas: Casos de uso relacionados y funciones del sistema también relacionadas.

Curso normal de los eventos.*

Acción del actor	Respuesta del sistema
Acciones numeradas de los actores.	Descripciones numeradas de las respuestas del sistema.

*Es la parte principal del formato expandido; describe los detalles de la conversión interactiva entre los actores y el sistema. Explica la secuencia más común de los eventos: la historia normal de las actividades y la terminación exitosa.

Cursos alternos.

- Alternativas que pueden ocurrir, indicando el número de acción. Descripción de excepciones.

Describe importantes opciones o excepciones que pueden presentarse en relación al curso normal. Si son complejas pueden expandirse y convertirse en casos de uso.

Ejemplo:

Caso de uso simplificado únicamente para pagos en efectivo.

Caso de uso: Comprar productos en efectivo.

Actores: Cliente, Cajero.

Propósito: Capturar una venta y su pago en efectivo.

Resumen: Un Cliente llega a la caja registradora con los artículos que comprará. El Cajero registra los artículos y recibe un pago en efectivo. Al terminar la operación, el Cliente se marcha con los productos.

Tipo: Primario y esencial.

Referencias cruzadas: Funciones r1.1, r1.2, r2.1,...

Curso normal de los eventos.

Acción del actor	Respuesta del sistema
1. Este caso de uso comienza cuando un Cliente llega a una caja de TPDV (Terminal Punto de Venta) con productos que desea comprar.	
2. El cajero registra el identificador de cada producto. Si hay varios productos de una misma categoría, el Cajero también puede introducir la cantidad	3. Determina el precio del producto e incorpora a la transacción actual la información correspondiente. Se presenta la descripción y el precio del producto actual.
4. Al terminar de introducir el producto, el Cajero indica al TPDV que se concluyó la	5. Calcula y presenta el total de la venta.

captura del producto.

6. El Cajero le indica el total al Cliente.

7. El Cliente efectúa un pago en efectivo posiblemente mayor que el total de la venta.

8. El Cajero registra la cantidad de efectivo recibida.

9. Muestra al Cliente la diferencia. Genera un recibo.

10. El Cajero deposita el efectivo recibido y extrae el cambio del pago.

11. Registra la venta concluida.

El Cajero da al Cliente el cambio y el recibo impreso.

12. El Cliente se marcha con los artículos comprados.

Cursos alternos.

- Línea 2: introducción de identificador inválido. Indicar error.
- Línea 7: el cliente no tenía suficiente dinero. Cancelar la transacción de venta.

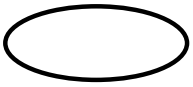
Diagramas de casos de uso.

Jacobson diseño un diagrama para la representación gráfica de los casos de uso, el cual es parte de UML.

Un diagrama de casos de uso explica gráficamente un conjunto de casos de uso de un sistema, los actores y la relación entre estos y los casos de uso.

Tiene dos componentes esenciales: actores y casos de uso.

Notación para casos de uso:



Caso de Uso

Notación para el actor:[2]

Actor

Un actor es una entidad externa que participa un papel con respecto al sistema. Por lo regular lo estimula con eventos de entrada o recibe algo de él.

Los actores llevan a cabo casos de uso. Un mismo actor puede realizar muchos casos de uso, y un caso de uso puede tener a varios actores.

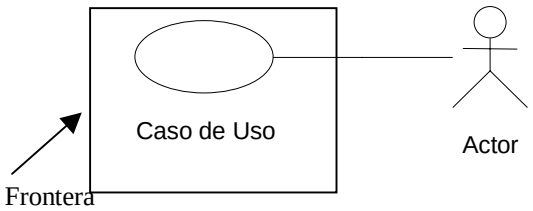
[2] Este es el icono estándar de UML, sin embargo a veces se utiliza un icono de una computadora para diferenciar al actor que es un sistema de cómputo y no una persona.

Los actores pueden ser cualquier tipo de sistema:

- Papeles que desempeñan las personas.
- Sistema de cómputo.
- Aparatos eléctricos o mecánicos.

Los actores sirven para **identificar** los casos de uso que realiza cada uno de ellos. Sin embargo, los actores sólo son un modo de llegar a ellos.

Diagrama de ejemplo:



Uses y *extends*.

Aparte de los vínculos entre los actores y los casos de uso, UML define dos tipos de vínculos:

Uses (usa)

Las relaciones *uses* ocurren cuando se tiene una porción de comportamiento que es similar en más de un caso de uso y no se quiere copiar la descripción de tal conducta.

Extends (extiende)

Las relaciones *extends* se ocupan cuando se tiene un caso de uso que es similar a otro, pero que hace un poco más.

¿Cuál es en realidad la diferencia entre *uses* y *extends*? Después de todo, ambos implican la factorización de comportamientos comunes de varios casos de uso.

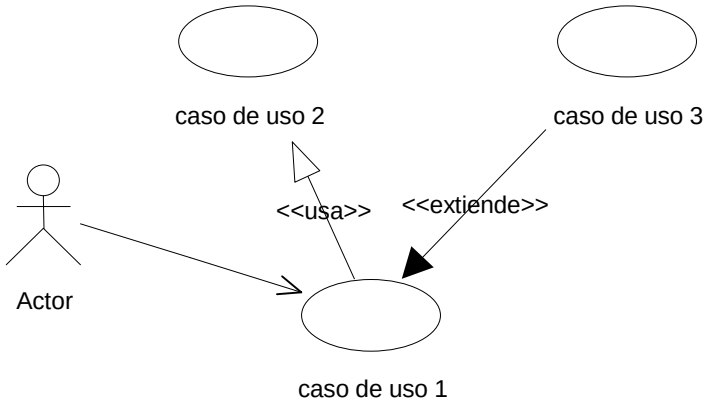
La intención es la que cambia:

En el caso de *extends*, los actores tienen que ver con los casos de uso que se están extendiendo. Un actor entonces se encargará tanto del caso de uso base como de todas las extensiones.

Para *uses* en cambio, es frecuente que no haya un actor asociado con el caso de uso común. Y si lo llega a tener, no se considera que esté llevando a cabo los demás casos de uso.

En conclusión se puede decir que:

- Es factible utilizar *extends* cuando se describa una **variación** de una conducta normal.
- Se ocupa *uses* para **repetir** cuando se trate de uno o varios casos de uso y desee evitar repeticiones.



Tipos de casos de uso.

La buena clasificación de los casos de uso ayuda a determinar prioridades en el desarrollo del software. La clasificación no forma parte del estándar de UML:

- **Primarios**. Representan los procesos comunes más importantes.
- **Secundarios**. Representan procesos menores o raros.
- **Opcionales**. Representan procesos que pueden no abordarse.

Casos de uso esenciales.

Los casos de uso esenciales son casos expandidos que se expresan en una forma teórica que contiene poca tecnología y pocos detalles de implementación; las decisiones de diseño se posponen y se abstraen de la realidad, especialmente las que tienen que ver con la interfaz del usuario.

Los casos de alto nivel siempre son de carácter esencial, debido a su brevedad y abstracción.

La característica fundamental es que permiten captar la esencia del proceso y sus motivos fundamentales, sin verse abrumado con detalles de diseño.

Casos de uso reales.

Describen especialmente el proceso a partir de su diseño concreto actual, sujeto a las tecnologías específicas de entrada y salida.

Los casos de uso reales se crean - en teoría - durante la fase de diseño en un ciclo de desarrollo.

Ejemplos de casos de uso esenciales y reales.

Esencial

Acción del actor	Respuesta del sistema
1. El Cajero registra el identificador en cada producto.	2. Determina el precio del producto y agrega la información sobre él a la actual transacción de venta.
Si hay más de un producto igual, el Cajero puede introducir de igual manera la cantidad.	Aparecen la descripción y el precio del producto actual.
...	...

Real

Acción del actor	Respuesta del sistema
1. en cada producto, el Cajero teclea el Código Universal de Productos (CUP) en el campo de entrada de la Ventana1.	2. Muestra el precio del producto y agrega la información sobre él a la actual transacción de venta.
Después oprime el botón "Introducir producto" con el ratón u oprimiendo la tecla <enter>	La descripción y el precio del producto actual se muestran en el cuadro de texto 2 de la ventana 1.

... ...

Proceso para la etapa de casos de uso.

1. Identificar actores y casos de uso.
2. Escribir los casos de uso en el formato de alto nivel. Otorgarles un tipo: primarios secundarios u opcionales.
3. Dibujar el diagrama de casos de uso.
4. Escribir los casos de uso más importantes, en el formato esencial expandido.
5. Los casos de uso reales se deben por lo general posponer para otra iteración o etapa del diseño, pues implica decisiones de diseño e implementación. Elabore casos de uso reales en etapas tempranas del ciclo de vida si existe alguna de las siguientes dos razones principales:
   - Las descripciones concretas facilitan notablemente la comprensión.
   - Los Clientes exigen especificar sus procesos en esta forma.

Conclusiones.

Son una herramienta esencial para la captura de requerimientos, la planificación, o el control de proyectos iterativos.

La mayoría de los casos de uso se generarán durante la fase inicial del proyecto, pero se descubrirán otros a medida que se avance.

El número de casos de uso puede variar de un sistema a otro y de acuerdo al nivel de granularidad de los diseñadores.

Es posible encontrar el término ***script*** como sinónimo al de casos de uso, por ejemplo el autor Ian Graham.

A veces se usa el término **escenario** relacionado con el de casos de uso, inclusive como sinónimo. En el contexto del UML, la palabra escenario se refiere a una sola ruta a través de un caso de uso.

Finalmente, es importante recalcar que existe más de una manera de llevar a cabo un caso de uso. En términos del UML se dice que un caso de uso puede tener muchas **realizaciones**.

Ejemplo de un diagrama de casos de uso

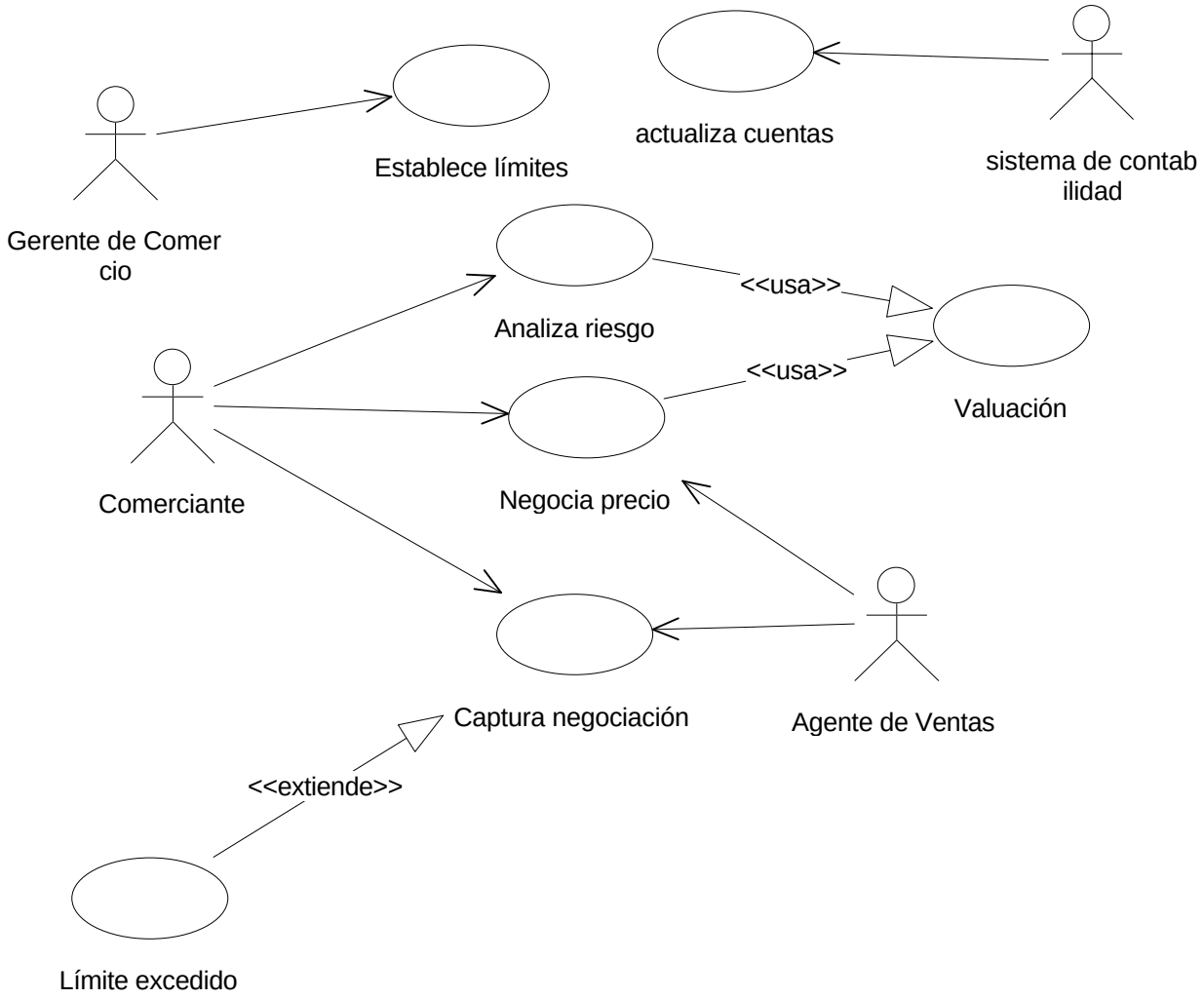


Diagrama de clases.

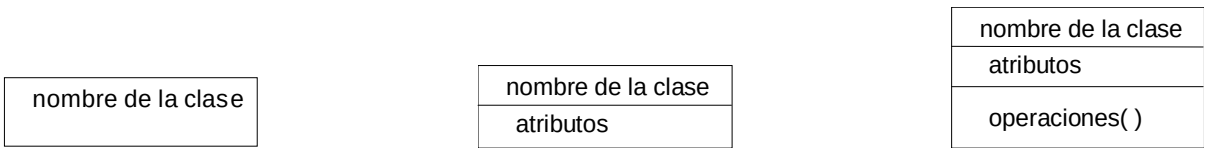
El diagrama de clases existe en diversas metodologías orientadas a objetos, este es muy útil para representar la estructura de un sistema en diversos niveles de comprensión.

El diagrama de clase describe los tipos de objetos que hay en el sistema y las diversas clases de relaciones estáticas que existen entre ellos.

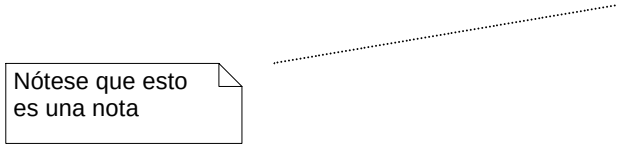
Los diagramas de clase muestran:

- Asociación.
- Generalización.
- Atributos.
- Operaciones.
- Agregación.

Notación de UML para representar una clase.



Notación de UML para representar una nota.



La siguiente tabla maneja algunos conceptos básicos de objetos y sus diferentes términos para otros autores o metodologías:

UML	Clase	Asociación	Generalización	Agregación
Rumbaugh (OMT)	Clase	Asociación	Generalización	Agregación
Booch	Clase	Usa	Hereda	Contiene
Coad	Clase y objeto	Conexión de instancias	Especialización - generalización	Parte-todo
Jacobson (Objectory)	Objeto	Asociación por reconocimiento	Hereda	Consiste en
Shlaer / Mellor	Objeto	Relación	Subtipo	n/a

Sin embargo, antes de seguir a describiendo los componentes de un diagrama de clases, vamos a mencionar las diferencias que puede tener un diagrama de acuerdo a la perspectiva hacia la cual se enfoque.

Se habla de tres perspectivas o modelos de acuerdo a Steve Cook y John Daniels:

- **Conceptual**. Bajo este modelo se dibuja un diagrama que represente los conceptos del domino que se está estudiando.

  Los modelos conceptuales se deben dibujar sin importar el software con que se implementarán, por lo que se considera independiente del lenguaje.

- **Especificación**. En esta perspectiva lo que se observa son las interfaces del software, no su implementación. Recordar que una interfaz puede tener muchas implementaciones distintas (ambiente de implementación, características de

desempéño, etc...)

- **Implementación**. En este modelo se tienen clases que exponen por completo la implementación.

La comprensión de la perspectiva; aunque no forma parte de UML, es muy importante para la creación y entendimiento del diagrama.

La mayor parte de los usuarios de los métodos orientado a objetos adopta un modelo de implementación; sin embargo, los otros modelos son regularmente más útiles.

Modelo conceptual.

En la etapa inicial se recomienda desarrollar el modelo conceptual, siendo este el artefacto más importante a crear durante el análisis orientado a objetos. Identificar objetos o conceptos constituye la esencia del análisis orientado a objetos.

Hay que recordar que los casos de uso son muy importantes como parte del análisis de requerimientos, pero no están realmente orientados a objetos.

Un modelo conceptual debe representar cosas del mundo real, no componentes de software.

Por lo tanto, una perspectiva o modelo conceptual es una representación de conceptos en un dominio del problema.

El modelo conceptual puede mostrarnos:

- Conceptos.
- Asociaciones entre conceptos.
- Atributos de conceptos.

Nota: Para el modelo conceptual, es común utilizar el término de concepto o clase como sinónimo.

Los siguientes elementos no son adecuados en un modelo conceptual:

- Componentes de software, como una ventana o una base de datos, salvo que el dominio a modelar se refiera a conceptos de software.
- Las responsabilidades o métodos.

Proceso de construcción del modelo conceptual.

1. Identifique las clases usando la lista de categorías de conceptos y la identificación de frases nominales relacionadas con los requerimientos. Aproveche los casos de uso.
2. Dibuje el diagrama inicial.
3. Agregue las asociaciones necesarias.
4. Agregue los atributos que se necesite para cumplir con las necesidades de información.

Identificación inicial de clases.

La descomposición en el análisis orientado a objetos se lleva a cabo mediante una división por clases o conceptos, en lugar de por funciones.

Por lo tanto, la tarea primordial de la fase de análisis consiste en identificar los conceptos - o clases - en el dominio del problema y documentar los resultados en un modelo conceptual.

Existen dos estrategias principales para la obtención de clases:

- Identificación de clases a partir de una lista de categorías de conceptos.
- Obtención de clases a partir de la identificación de frases nominales.

Lista de categorías de conceptos.

Una lista de categorías de conceptos contiene muchas categorías comunes que se deberían tener en cuenta. La lista de categorías que se presenta a continuación, contiene además algunos ejemplos de clases para los dominios de una tienda y de un sistema de reservaciones de líneas aéreas.

Categoría del concepto	Ejemplos
Objetos físicos o tangibles	Terminal de punto de venta Avión
Especificaciones, diseño o descripciones de cosas	Especificación de producto Descripción de vuelo
Lugares	Tienda Aeropuerto
Transacciones	Venta, Pago

	Reservación
Línea o renglón de elemento de transacciones	Ventas línea de producto
Papel de las personas	Cajero Piloto
Contenedores de otras cosas	Tienda, Carro de compras Avión
Cosas dentro de un contenedor	Producto Pasajero
Otros sistemas de cómputo o electromecánicos externos al sistema	Sistema de autorización de tarjeta de crédito Control de tráfico aéreo
Conceptos de nombres abstractos.	Acrofobia
Organizaciones	Departamento de ventas Objeto línea aérea
Eventos	Venta, Robo, Junta Vuelo, Accidente, Aterrizaje
Procesos	Venta de un producto Reservación de asiento
Reglas y políticas	Política de reembolso Política de cancelaciones
Catálogos	Catálogo de producto Catálogo de partes
Registro de finanzas, de trabajo, de contratos de asuntos legales	Recibo, Mayor, Contrato de empleo Bitácora de mantenimiento.
Instrumentos y servicios financieros	Línea de crédito Existencia
Manuales, libros	Manual de personal Manual de reparaciones.

Frases nominales.

Esta técnica consiste en identificar las frases nominales[3] en las descripciones textuales del dominio de un problema y considerarlas clases o atributos idóneos.

Los casos de uso expandidos son utilizados como una fuente para identificar a las clases.

Ejemplo:

Actores	**Respuesta del sistema**
1. Esta caso de uso comienza cuando un **Cliente** llega a una **caja de terminal de punto de venta** con **productos** que desea comprar. 2. El **Cajero** registra **el código universal de productos** en cada **producto**. Si hay más de un **producto** el **Cajero** puede introducir también la **cantidad**.	3. Determina el **precio del producto** y a la **transacción de ventas** le agrega la información sobre el producto. Se muestran la **descripción** y el **precio** del **producto** actual.

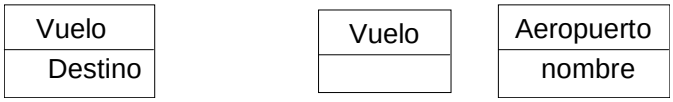
Esta técnica tiene la desventaja de depender del lenguaje utilizado, pues podemos tener diversas frases nominales para un mismo concepto o atributo.

No todas las frases nominales se consideran conceptos o clases, pueden ser atributos de clase.

Sin embargo, el error más frecuente cuando se crea el modelo conceptual es el de representar una clase como un atributo.

Si en el mundo real no consideramos algún concepto como número o texto, probablemente sea un concepto y no un atributo.

Ejemplo: Para un modelo de reservaciones de boletos en líneas aéreas.



Selección de clases.

Todas las clases deben tener sentido en el área de la aplicación, la relevancia al problema debe ser el único criterio para la selección.

- Se deben escoger con cuidado los nombre para las clases, para que estos no sean ambiguos.
- Se deben eliminar las clases **redundantes**, si las dos expresan la misma información. La clase más descriptiva debe de conservarse.
- Se deben eliminar las clases **irrelevantes**, que tienen poco o nada que ver con el problema. Esto requiere juicio porque en un contexto una clase puede ser importante mientras que en otro contexto podría no serlo.
- Se deben eliminar las clases que debieran ser **atributos** más que clases, cuando los nombres corresponden a propiedades (como un número o texto), más que a entidades independientes.

[3] Una frase nominal es una o más palabras que tienen un sustantivo o pronombre, o que funcionan como tal.

Si existe duda de eliminar o no una clase, es preferible conservarla; ya se eliminará en etapas posteriores si es innecesaria.

En este punto se puede construir el diagrama de clases básico, conteniendo únicamente los nombres de las clases.

Conceptos de UML.

Clase. Es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, métodos, relaciones y semántica. Es independiente del molo (conceptual, de especificación o de implementación).

Clase de implementación. Es una clase implementada de software, inclusive en un lenguaje en particular.

Operación. Es un servicio que puede solicitarse a un objeto para que realice un comportamiento.

Método. Es la implementación de una operación que especifica el algoritmo o procedimiento.

Tipo. La definición de tipo en UML se parece al de clase, ya que describe un conjunto de objetos parecidos con atributos y operaciones, pero no puede incluir métodos. Un tipo es una especificación de una entidad de software y no una implementación. Por lo tanto, un tipo en UML es independiente del lenguaje.

Asociaciones.

Una asociación representa relaciones entre instancias de clases, estas indican alguna conexión significativa entre las instancias.

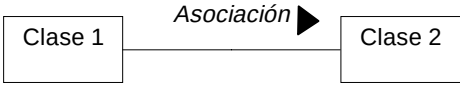
Desde la perspectiva conceptual, las asociaciones representan relaciones conceptuales entre clases.

La asociación se considera bidireccional, lo que quiere decir que es posible una conexión lógica entre los objetos de una clase y los de la clase asociada.

Bajo este modelo se considera abstracto y no es una afirmación sobre las conexiones entre las entidades del software.

Notación para una asociación en UML

Una asociación en UML se representa como una línea entre clases con un nombre de la asociación de manera opcional.



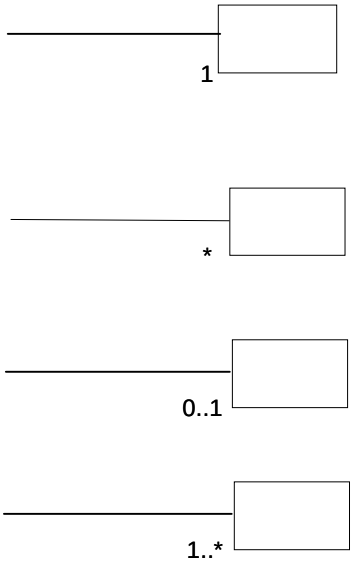
La flecha en el nombre es opcional e indica la dirección en que debe leerse el nombre de la asociación. En ausencia, por convención la asociación se lee de izquierda a derecha o de arriba hacia abajo.

Multiplicidad.

Cada asociación tiene dos papeles y se encuentran en cada extremo de la asociación; donde cada papel es una dirección en la asociación, y cada uno de ellos tiene una multiplicidad, la cual es una indicación de la cantidad de objetos que participarán en la relación dada.

En general, la multiplicidad indica los límites inferior y superior de los objetos participantes.

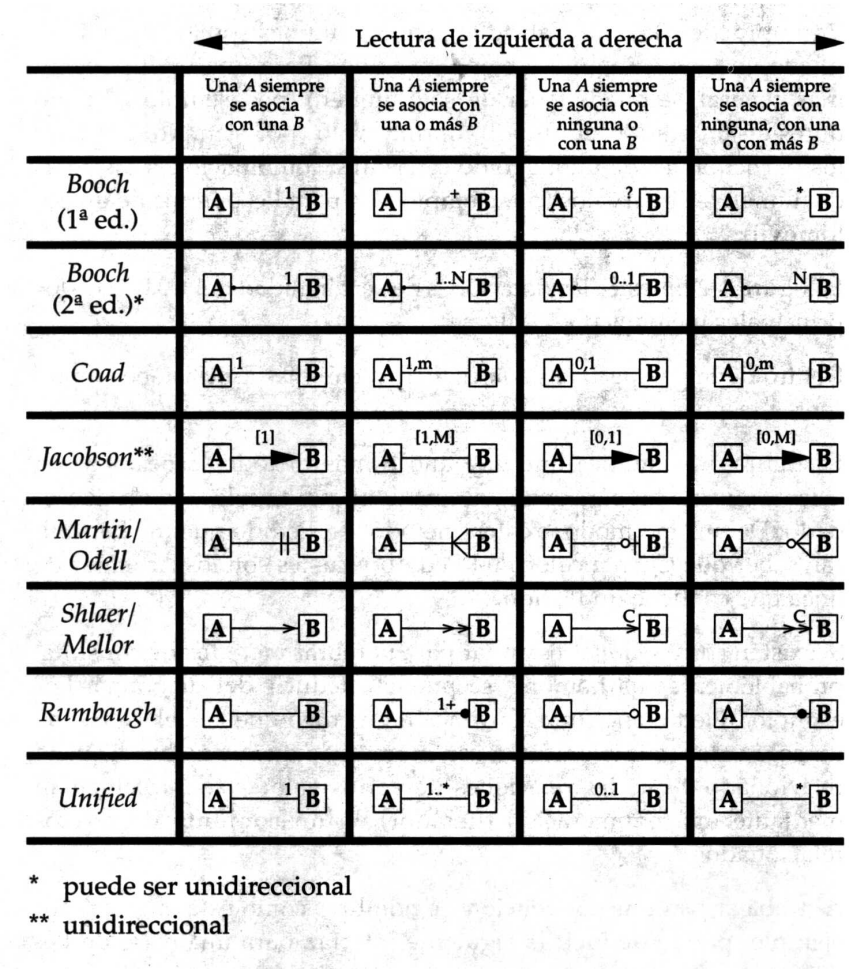
Notación de multiplicidad en UML:



En UML, la dirección de un papel se indica mediante una flecha en la asociación que representa la **navegabilidad** de la misma. Lo que en un nivel de modelado de implementación implicaría el manejo de un apuntador hacia la otra clase asociada.

El término papel también es conocido como rol en algunas metodologías como OMT.

Diferentes notaciones de multiplicidad o cardinalidad:



Identificación de asociaciones.

La identificación de las asociaciones puede hacerse también con una lista de asociaciones comunes. Aunque, en términos generales podríamos decir que siempre conviene tener en cuenta el siguiente tipo de asociaciones:

- A es una parte física o lógica de B.
- A está física o lógicamente contenido en B.
- A está registrado en B.

La lista completa se presenta a continuación:

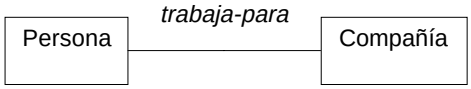
Categoría	Ejemplos
A es una parte física de B	Caja - TPDV Ala - Avión
A es una parte lógica de B	VentasLíneadeProducto - Venta TramodeVuelo - RutadeVuelo
A está físicamente contenido en B	TPDV - Tienda, Producto - Estante Pasajero - Avión
A está contenido lógicamente en B	DescripcióndeProducto - Catálogo Vuelo - ProgramadeVuelo
A es una descripción de B	DescripcióndeProducto - Producto DescripcióndeVuelo - Vuelo
A es un elemento en una transacción o reporte B	VentasLineadeProducto - Producto TrabajodeMantenimiento - Mantenimiento
A se conoce / introduce / registra / presenta / captura en B	Venta - TPDV Reservación - ListadePasajeros
A es miembro de B	Cajero - Tienda Piloto - Avión
A es una subunidad organizacional de B	Departamento - Tienda Mantenimiento - LíneaAérea
A usa o dirige a B	Cajero - TPDV Piloto - Avión
A se comunica con B	Cliente - Cajero AgentedeReservaciones - Pasajero
A se relaciona con una transacción B	Pago - Venta Pasajero - Boleto
A es una transacción relacionada con otra	Pago- Venta

transacción B	Reservación - Cancelación

A está contiguo a B	TPDV - TPDV Ciudad - Ciudad
A es propiedad de B	TPDV - Tienda Avión - LíneaAérea

Algunos aspectos importantes para la identificación de asociaciones:

- Concentrarse en las asociaciones en que el conocimiento de la relación ha de preservarse durante algún tiempo.
- No incluir las asociaciones redundantes ni las derivables.
- En el modelo conceptual una asociación **no** es una proposición sobre los flujos de datos, ni conexiones de objetos en software; es una proposición de que una relación es significativa en un sentido puramente analítico: en el mundo real.
- El nombre de la asociación regularmente es un verbo que relaciona a las clases, generando una secuencia legible y significativa dentro del contexto del modelo.
- El valor de la multiplicidad depende del contexto. Por ejemplo, una relación:



¿Cuál será la multiplicidad de cada papel, de la asociación?

El modelo puede ser para una o varias instancias de Compañía, para el departamento de impuestos le interesan **muchas**, mientras que a un sindicato probablemente le interese solo **una**.

- Un buen modelo ocupa un punto intermedio de un modelo basado en la necesidad mínima de conocimiento y otro que contenga todas las relaciones posibles.

Atributos.

Un atributo es un valor lógico de un dato en un objeto. Este puede ser visto desde diferentes perspectivas. Pensemos en una clase Cliente, con un atributo nombre.

- Desde una perspectiva conceptual el atributo nombre indica que los Clientes tienen nombres.
- Desde un modelo de especificación, este atributo indica que un objeto Cliente puede decir su nombre y tiene algún modo de establecer su nombre.
- En el modelo de implementación se dice que Cliente tiene un miembro o variable de instancia para su nombre.

Desde el modelo conceptual, se deben identificar los atributos de las clases que se necesitan para satisfacer los requerimientos de información de los casos de uso en cuestión. Aquellos en que los requerimientos indican o conllevan la necesidad de recordar información.

Notación de los atributos en UML

Dependiendo del detalle del diagrama y del modelo, la notación de un atributo puede mostrar:

- Nombre.
- El tipo.
- Valor predeterminado o por omisión.
- Visibilidad.

Sintaxis:

Visibilidad nombre : tipo = valor por omisión.

Identificación de los atributos

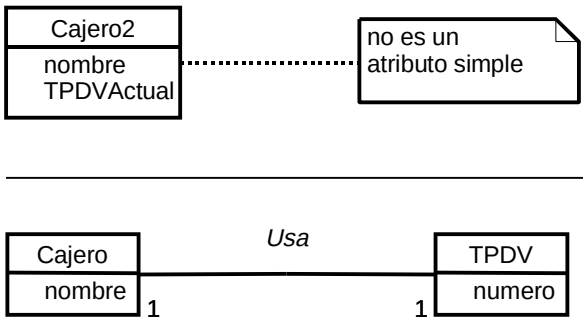
Cuando se reconocieron las clases para el modelo conceptual, aparecieron los primeros atributos, algunos de los cuales deben de ser agregados a una de las clases definidas.

Los tipos más simples de atributos son los que suelen considerarse, también llamados tipos primitivos de datos. Generalmente, un atributo no debería ser un concepto complejo del dominio del problema. Esto es válido sobre todo para el modelo conceptual.

Atributos simples comunes:

- Número.
- Cadena o texto.
- Fecha.
- Hora.

En el siguiente ejemplo vemos un atributo que debería de ser clase, para el dominio del sistema de ventas:



Según Rumbaugh, los atributos deberían de ser valores puros de datos (tipos de datos), en los cuales la identidad única no es significativa para el dominio del problema.

A partir de esto podemos suponer que no es significativo generalmente:

- Distinguir entre instancias aisladas de un número. Ejemplo: 123.
- Distinguir entre instancias aisladas de Números Telefónicos iguales.
- Distinguir entre instancias aisladas de Dirección que contengan la misma dirección.

Por el contrario sería necesario distinguir entre dos instancias aisladas de Persona, aunque el nombre fuera el mismo, pues cada objeto puede representar a diferentes personas que tengan el mismo nombre. Es decir; nos interesaría identificar a cada persona de manera independiente.

Recordar que si no se está seguro conviene definirlo como una clase y no como un atributo.

Atributos compuestos.

Una vez que se identificaron los atributos simples es posible identificar algunos atributos compuestos. Para estos si se pueden mostrar el tipo en el diagrama del modelo conceptual, dado que no se trata de un dato simple.

Represente como un tipo compuesto lo que inicialmente puede considerarse un tipo primitivo de datos:

- Si se compone de secciones independiente. Ejemplo: nombre de persona.
- Se asocian generalmente operaciones de análisis o validación. Ejemplo: número de seguro social, de tarjeta de crédito.
- Contiene otros atributos. Ejemplo: Un precio promocional, que puede contener fecha de inicio y de terminación.
- Si se trata de una cantidad o unidad de un número. Ejemplo: importe de pago.

Este último punto se refiere que - algunas veces - una cantidad puede ser representada como una clase aparte si esta requiere una mayor flexibilidad o robustez. Es similar al segundo punto en cuanto a que puede tener operaciones relacionadas.

Suponga en el sistema de ventas que este deba adaptarse a varias monedas al destinarse el sistema a varios países.

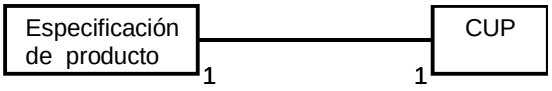
Además, en el diagrama de clases es factible mostrar un atributo compuesto dentro de la clase o como una clase independiente en el diagrama del modelo conceptual.

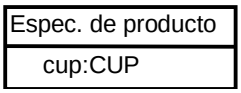
- Se trataba de un valor del cual no se requiere tener la identidad de cada instancia, por lo que podría mostrarse en la sección de atributos de la clase.

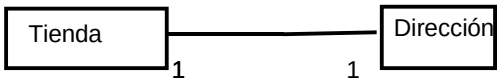
- Por ser un atributo con sus propias características, es factible mostrarlo como una clase independiente.

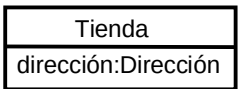
En realidad, no hay una solución ideal, dependerá de lo que queramos destacar en el diagrama y de la importancia del concepto para el dominio del problema. Recordar que un modelo conceptual es una herramienta de comunicación; las decisiones sobre lo que debería mostrarse han de tomarse teniendo eso en cuenta.

Ejemplo con el sistema de ventas para el código universal de productos y dirección:

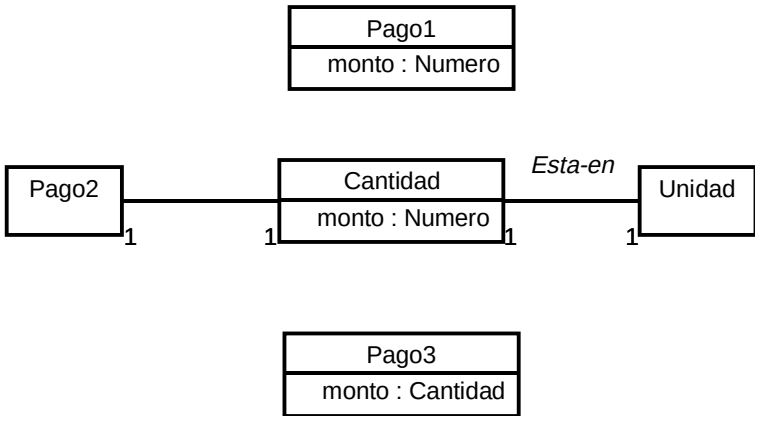








Ejemplo para el sistema de ventas en el importe de pago:



Generalidades

Los atributos no deberían servir para relacionar conceptos en el modelo conceptual. Es un error frecuente agregar un tipo de *atributo de llave foránea*, pues suele hacerse con los diseños de bases de datos relacionales.

No todos los atributos son evidentes en esta etapa, pero podemos apoyarnos identificando los datos involucrados en las salidas del sistema.

Podemos tener atributos derivados. Un atributo derivado es aquel que puede ser deducido de otra información, pero que se considera una característica más que una operación de la clase. En UML se representa por medio del símbolo "/", como prefijo del nombre del atributo.

Diccionario de clases.

Una parte fundamental de la documentación, es registrar la información que se pretende manejar en el sistema. En un análisis estructurado esto se conoce como diccionario de datos.

El diccionario de clases puede tener la información presentada de diversas maneras. Presentamos a continuación el formato común:

Descripción de clases:

Nombre de la clase	**Descripción**

Sin embargo, este formato es conveniente sustituirlo por documentación del tipo de tarjetas CRC.

Las tarjetas de Clase-Responsabilidad-Colaboración representan las clases en tarjetas de 4 x 6 pulgadas, describiendo las responsabilidades de cada clase.[4]

Nombre de la clase	
Responsabilidad	*Colaboración*

La idea de las tarjetas CRC es captar el propósito de la clase en unas cuantas

[4] Ver artículo de Cuningham y Beck

frases. Con cada responsabilidad se indica cuáles son las otras clases con las que tiene que trabajar para cumplirla.

Es importante pensar en las responsabilidades, ya que evita pensar en las clases como simples depositarias de datos y ayuda a centrarse en el comportamiento de alto nivel de cada clase.

No se deben generar largas listas de responsabilidades, de ser así tal vez se estén describiendo responsabilidades de bajo nivel o necesite dividirse la clase.

Por otro lado, es importante documentar los atributos de cada clase indicando cual es la intención del mismo.

Nombre del atributo	**Descripción**

Existen otros datos que son importantes para el diccionario de clases, pero de momento no se tienen; como la visibilidad de los atributos, su tipo, una lista de las operaciones de cada clase, etc.

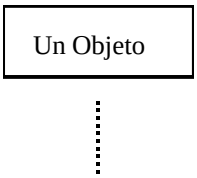
Diagramas de Secuencia.

Para analizar el comportamiento de las clases de un sistema, UML proporciona lo que se conoce como **diagramas de interacción**, que describen la manera en que **colaboran** los objetos.

Existen dos tipos de diagramas de interacción:

- Diagrama de secuencia.
- Diagrama de colaboración.

En esta etapa vamos a tomar en cuenta el diagrama de secuencia, el cual se construye para cada caso de uso.



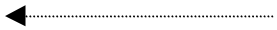
Principales componentes del diagrama de secuencia:

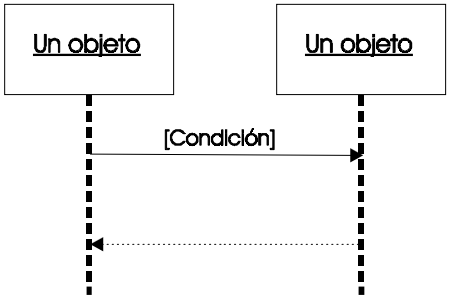
- **Objeto** de una clase, que se representa en un rectángulo y con su texto subrayado.
- **Línea de vida**. A la línea vertical se le llama línea de vida del objeto, y representa la vida del objeto durante la interacción. [Jacobson].
- Una **flecha** entre las líneas de vida de dos objetos representa un **mensaje** que un objeto envía a otro. ⟶

  Cada mensaje es etiquetado por lo menos con el nombre del mensaje, pero pueden incluirse argumentos e información de control.

Existe también la **autodelegación**, que es cuando un objeto se envía un mensaje a sí mismo.

- **Condición. I**ndica que un mensaje se envía sólo si la condición es verdadera. Sintaxis: [condición]
- **Marcador de iteración**. Muestra que un mensaje se envía muchas veces a varios objetos receptores. Sintaxis: *[condición]
- **Regreso**. Indica el regreso de un mensaje, no un nuevo mensaje. Sin embargo, hay que tener cuidado con ellos porque pueden saturar el diagrama, siendo que muchos regresos son obvios. Se representan con una flecha de línea punteada.





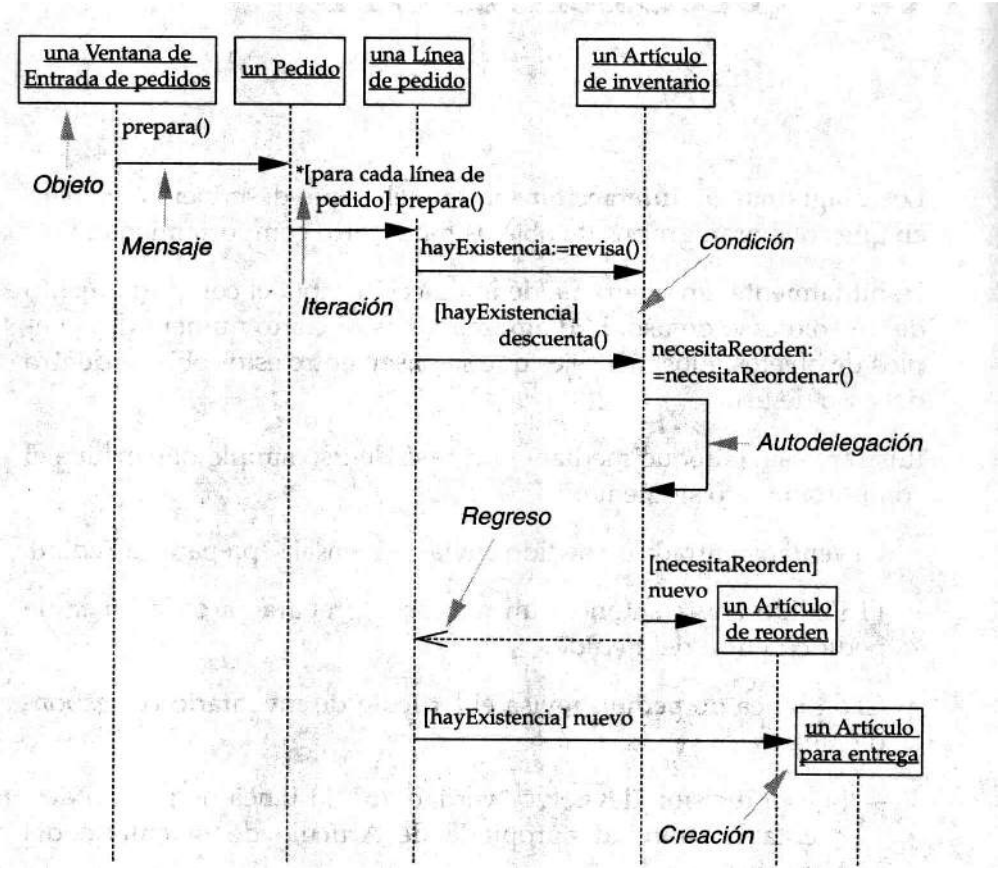
Ejemplo:

Supóngase un caso de uso de pedido con el siguiente comportamiento:

- *La ventana Entrada de pedido envía un mensaje "prepara" a Pedido.*
- *El pedido envía entonces un mensaje "prepara" a cada Línea de pedido dentro del Pedido.*
- *Cada Línea de pedido revisa el Artículo de inventario correspondiente.*
    - *Si esta revisión devuelve "verdadero", la Línea de pedido descuenta la*

cantidad apropiada de Artículo de inventario del almacén.

- *En caso contrario, quiere decir que la cantidad del Artículo de inventario ha caído más abajo del nivel de reorden y entonces dicho Artículo de inventario solicita una nueva entrega.*



Procesos concurrentes.

Los diagramas de secuencia son también útiles para mostrar los procesos concurrentes.

Para esto es necesario añadir al diagrama las **activaciones**, que indican explícitamente cuando un método está activo. Se puede usar en cualquier momento para

todos los diagramas, aunque hay quienes las usan únicamente para procesos concurrentes.

Entonces una activación muestra que un objeto esta efectuando una operación o se encuentra esperando la devolución de una subrutina.

Una **media flecha** indica un mensaje **asíncrono**. Un mensaje asíncrono no bloquea al invocador, por lo cual puede continuar con su proceso.

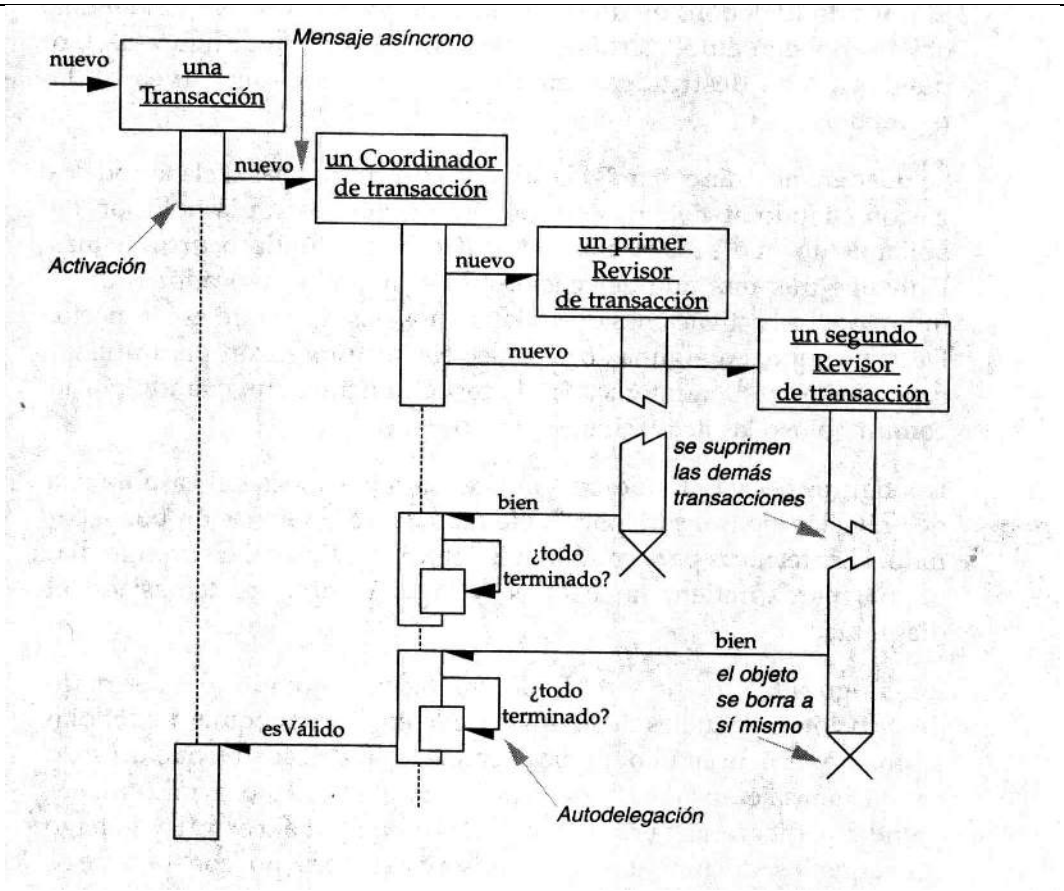
Un mensaje asíncrono puede:

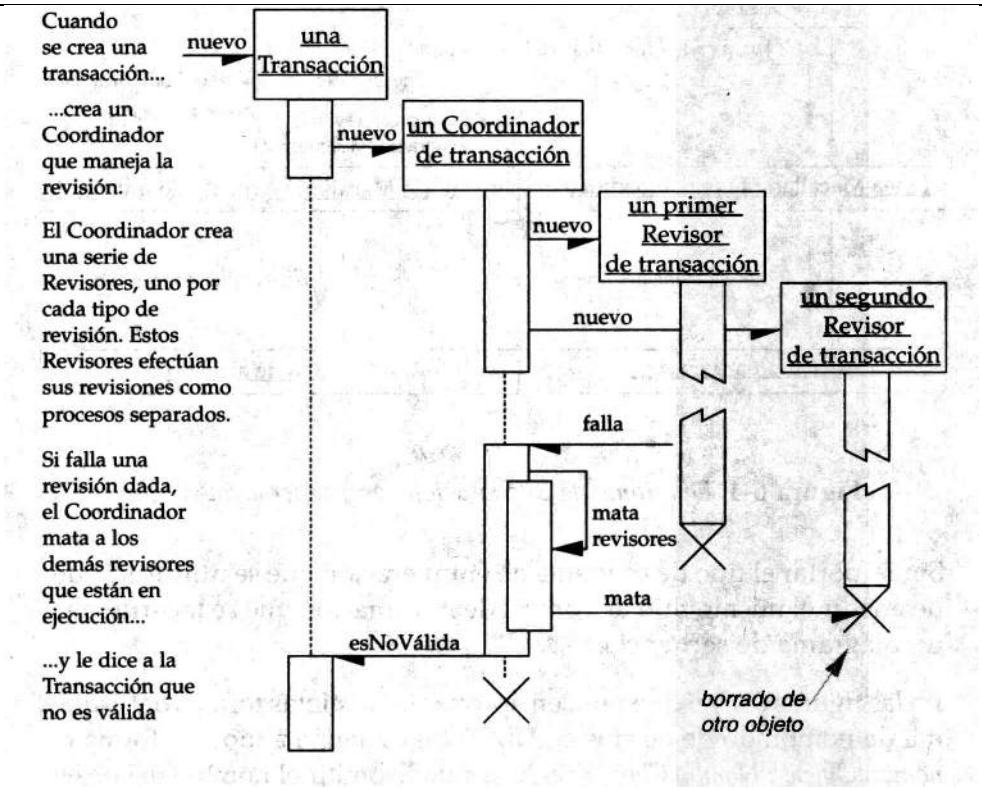
- Crear un nuevo proceso.
- Crear un nuevo objeto.
- Comunicarse con un proceso que ya está operando.

Un objeto puede ser **eliminado** por otro objeto o puede autodestruirse; esto se muestra en el diagrama con una X.

También se pueden agregar descripciones textuales que expliquen lo que sucede en el diagrama, sobre todo para diagramas finales que formen parte de la documentación final.

A continuación se muestran dos diagramas que muestran la notación mencionada, cada uno mostrando una situación diferente para un caso de uso de transacciones bancarias.


nuevo
una Transacción
Mensaje asíncrono
nuevo
un Coordinador de transacción
Activación
nuevo
un primer Revisor de transacción
nuevo
un segundo Revisor de transacción
se suprimen las demás transacciones
bien
¿todo terminado?
bien
el objeto se borra a sí mismo
¿todo terminado?
esVálido
Autodelegación


Cuando se crea una transacción...
...crea un Coordinador que maneja la revisión.
El Coordinador crea una serie de Revisores, uno por cada tipo de revisión. Estos Revisores efectúan sus revisiones como procesos separados.
Si falla una revisión dada, el Coordinador mata a los demás revisores que están en ejecución...
...y le dice a la Transacción que no es válida
nuevo
una Transacción
nuevo
un Coordinador de transacción
nuevo
un primer Revisor de transacción
nuevo
un segundo Revisor de transacción
falla
mata revisores
mata
esNoVálida
borrado de otro objeto

Contratos.

Antes de pasar al diseño de la aplicación del software, es necesario definir el comportamiento que tiene el sistema a través del comportamiento de los objetos. El **comportamiento** es una descripción de lo que hace, sin explicar como lo hace.

Los **contratos** son documentos que describen el comportamiento de las operaciones del sistema. Está basado en la técnica de diseño por contratos de Bertrand Meyer, siendo esta una característica del lenguaje Eiffel.

Un **contrato** es un documento que describe lo que una operación se propone lograr. Suele redactarse en un estilo declarativo, enfatizando **lo que** sucederá y no **cómo** se conseguirá.

Formato general de un contrato.

Nombre:	Nombre de la operación y parámetros.
Clase:	Clase a la que pertenecería la operación.
Responsabilidades:	Descripción informal de las responsabilidades que debe cumplir la operación.
Notas:	Notas de diseño, algoritmos e información afín.
Excepciones:	Reacción ante situaciones especiales.
Precondiciones:	Suposiciones acerca del estado del sistema antes de ejecutar la operación.
Poscondiciones:	El estado del sistema después de la operación.

A continuación ahondaremos un poco más en las principales partes del contrato:

Precondiciones.

Es un enunciado de cómo esperamos encontrar el mundo antes de ejecutar una operación.

La precondición hace explícito que quien invoca es el responsable de la comprobación. Sin este enunciado explícito, podemos tener muy poca comprobación (ambas partes suponen que la otra es responsable) o tenerla en exceso.

Poscondiciones.

Después de las responsabilidades, la parte más importante del contrato son las poscondiciones.

Una poscondición es un enunciado sobre cómo debería verse el mundo después de la ejecución de una operación.

Las poscondiciones se expresan dentro del contexto del modelo conceptual, ya que parte de las instancias y asociaciones posibles a partir del modelo. Sin embargo, es posible que se vea la necesidad de registrar en el modelo nuevas clases, atributos o asociaciones.

Deben describir el estado del mundo, no las acciones a realizar. Es conveniente expresarlas en tiempo pasado.

Proceso para el desarrollo de contratos.

- Identifique las operaciones a partir de los diagramas de secuencia.
- Elabore un contrato para cada operación del sistema.
- Una vez anotado el nombre y parámetros de la operación, continué con la sección de responsabilidades de la operación.
- Complete la sección de poscondiciones, dejando para el final las precondiciones.

Generalmente las poscondiciones implican:

- Creación y eliminación de instancias.
- Modificación de atributos.
- Asociaciones formadas y canceladas.

- Agregue las notas, si considera necesario explicar detalles de diseño.
- Por último, especifique las excepciones de la operación. Recuerde que una excepción ocurre cuando se invoca una operación estando satisfechas sus precondiciones y, sin embargo, no puede regresar con sus poscondiciónes satisfechas.

Ejemplos de contratos.

Ejemplo 1.

Nombre:	introducirProducto (cup:número, cantidad: entero).
Clase:	TPDV.
Responsabilidades:	Registrar la venta de un producto y agregarlo a la venta. Desplegar la descripción del producto y su precio.
Notas:	Optimizar el acceso a la base de datos.
Excepciones:	Si el CUP no es válido, indique que se cometió un error.
Precondiciones:	Se conoce el CUP.
Poscondiciones:	Si se trató de una nueva venta, una **instancia** Venta fue **creada**. Si se trató de una nueva venta, la nueva instancia de Venta fue **asociada** a la TPDV. Se creó una instancia de VentasLineadeProducto. Se asoció VentasLineadeProducto a la Venta. Se estableció VentasLineadeProducto.cantidad con el valor de

	cantidad. La instancia VentasLineadeProducto fue asociada a una EspecificacióndeProducto, basado en la correspondencia del código universal de producto.

Ejemplo 2.

Nombre:	terminarVenta().
Clase:	TPDV.
Responsabilidades:	Registrar que es el final de la captura de los productos de la venta y desplegar el total de la venta.
Notas:	
Excepciones:	Si no está realizándose une venta, indicar que se cometió un error.
Precondiciones:	Se realizó la venta de al menos un producto.
Poscondiciones:	Estableció al atributo Venta.estaTerminada en verdadero.

Conclusiones.

El diseño por contrato es una técnica valiosa que ayuda a la construcción de interfaces claras. Forma parte directa del lenguaje Eiffel porque proporciona el manejo de afirmaciones, el cual es un concepto que no manejamos en este curso, ya que la mayor intención es apoyar a la comprensión del problema antes de entrar al diseño del sistema.

Diseño

La línea divisoria entre el análisis y el diseño es muy delgada, y varia de acuerdo a

la metodología o el autor, aunque la mayoría concuerda en que pretende un mayor acercamiento a la implementación; es decir, a la construcción de la solución propuesta por el análisis.

Casos de uso reales.

Un caso de uso real describe el diseño concreto del caso de uso a partir de una tecnología particular de entrada y salida, así como de su implementación global.

Es común para un caso de uso en donde interviene la interfaz con el usuario, incluir diagramas de las ventanas y una explicación de la interacción con la misma.

La alternativa es construir *storyboards* o secuencias de pantalla de la interfaz relacionada a los casos de uso.

Ejemplo:

Caso de uso: Comprar productos en efectivo.

Actores: Cliente, Cajero.

Propósito: Capturar una venta y su pago en efectivo.

Resumen: Un Cliente llega a la caja registradora con los artículos que comprará. El Cajero registra los artículos y recibe un pago en efectivo. Al terminar la operación, el Cliente se marcha con los productos.

Tipo: Primario y **real**.

Referencias cruzadas: Funciones r1.1, r1.2, r2.1,...

Ventana 1:

Curso normal de los eventos.

Acción del actor	Respuesta del sistema
1. Este caso de uso comienza cuando un Cliente llega a una caja de TPDV (Terminal Punto de Venta) con productos que desea comprar.	
2. Con cada producto, el Cajero teclea el CUP en (A) de la ventana 1. Si hay más de un producto, es opcional capturar la cantidad en (E). Se oprime (H) después de capturar cada producto.	3. Agrega la información sobre el producto a la actual transacción de ventas. La descripción y el precio del producto actual se muestran en (B) y en (F) de la ventana 1.
4. Al terminar de capturar los productos, el Cajero oprime el botón (I) para indicarle al TPDV que terminó de capturar los productos.	5. Calcula y presenta en (C) el total de la venta.
6. El Cajero le indica el total al Cliente.	

Diagramas de colaboración.

La segunda forma del diagrama de interacción es el diagrama de colaboración.

En un diagrama de colaboración, los objetos se muestran como iconos. Las flechas indican; como en los diagramas de secuencia, los mensajes enviados dentro de un caso de uso determinado, salvo que los mensajes se numeran.

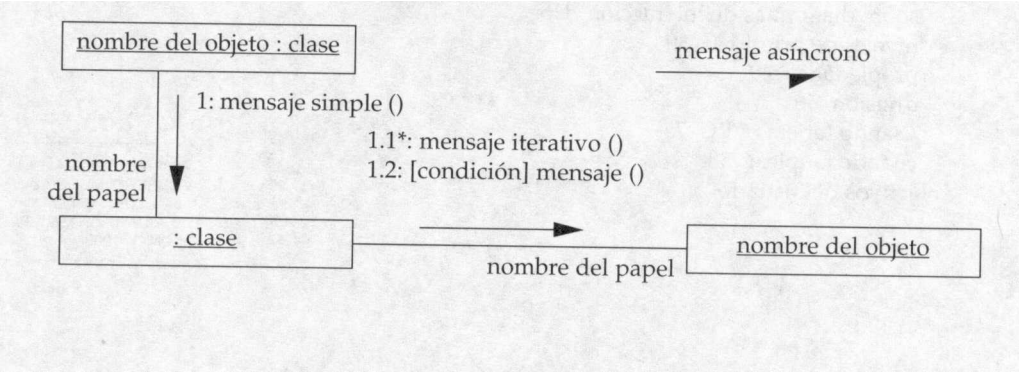
Los diagramas de colaboración describen las interacciones entre los objetos en un formato de grafo o red.

Aunque este diagrama no permite ver de forma tan clara la secuencia como las líneas verticales, la disposición de los objetos permite apreciar otros aspectos de la vinculación de los objetos, inclusive para determinar la agrupación de los mismos.

Constituye una de las herramientas más importantes que se generan en el análisis y diseño orientado a objetos.

El tiempo y el esfuerzo dedicados a su preparación deberían absorber un porcentaje considerable de la actividad total destinada al proyecto.

La sintaxis de UML es la siguiente:



Obsérvese que para la representación del objetos se puede poner *nombreObjeto : NombreClase*, donde se puede omitir el nombre del objeto o de la clase. En caso de que

se omita el nombre de la clase se deben de conservar los dos puntos.

El vínculo es una trayectoria de conexión entre dos instancias, e indica alguna forma de navegación y visibilidad que es posible entre las instancias (*instancia de una asociación*).

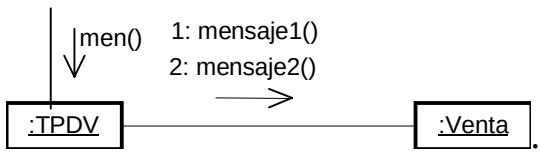
Algunos aspectos importantes:

- Los mensajes entre objetos pueden representarse por medio de una flecha con un nombre y situada sobre una línea del vínculo. Puede haber un número indefinido de mensajes.
- Es posible que un objeto se mande un mensaje a si mismo (**autodelegación**).
- Los parámetros de un mensaje pueden anotarse dentro de los paréntesis después del nombre del mensaje.
- Puede incluirse un valor de regreso anteponiéndole al mensaje un nombre de variable y un operador de asignación ":=" ó ":".
- La sintaxis completa para los mensajes:

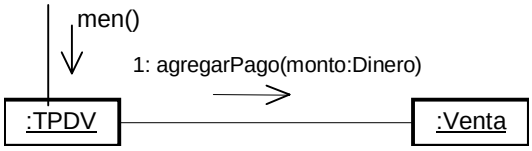
  Retorno := mensaje (parámetro : tipoParámetro, ...) : tipoRetorno

- Bajo este diagrama se numeran los mensajes utilizando un esquema decimal, de manera que sea evidente que operación llama a cual.

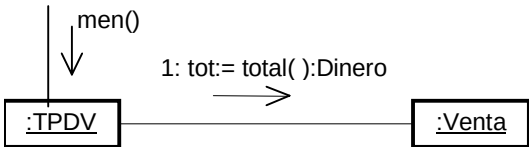
Ejemplo: paso de mensajes.



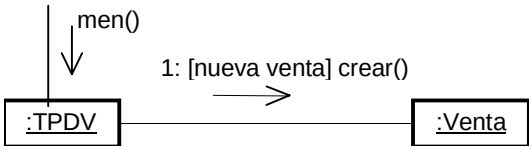
Ejemplo: paso de parámetros



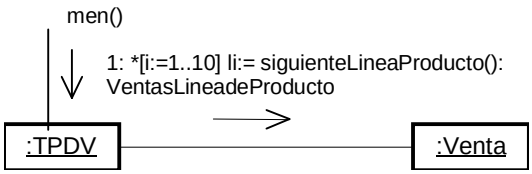
Ejemplo: valor de regreso.



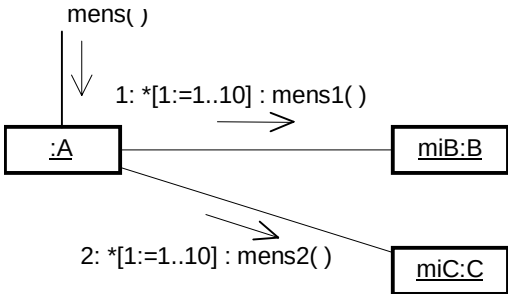
Ejemplo: condición



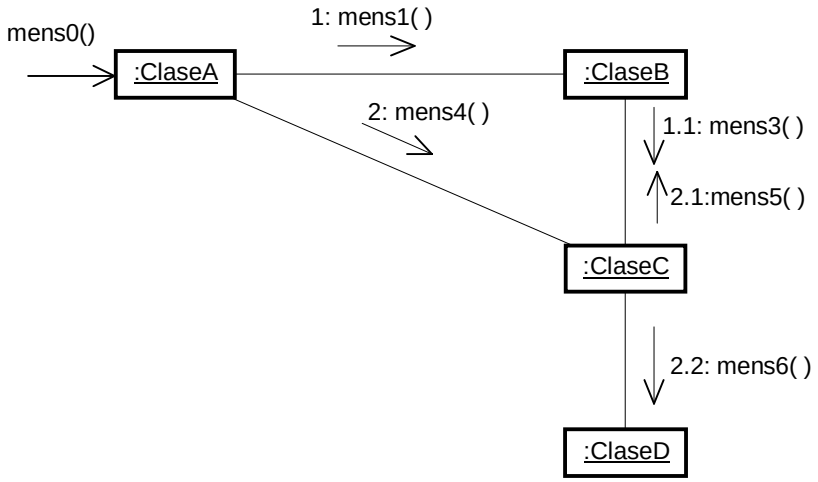
Ejemplo: iteración



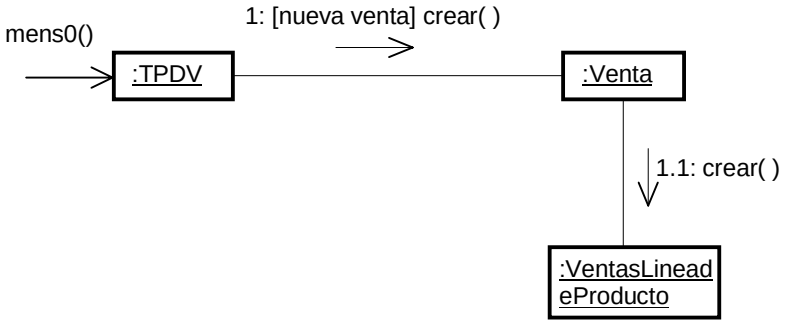
Ejemplo: iteración con mensajes a múltiples objetos de distintas clases.



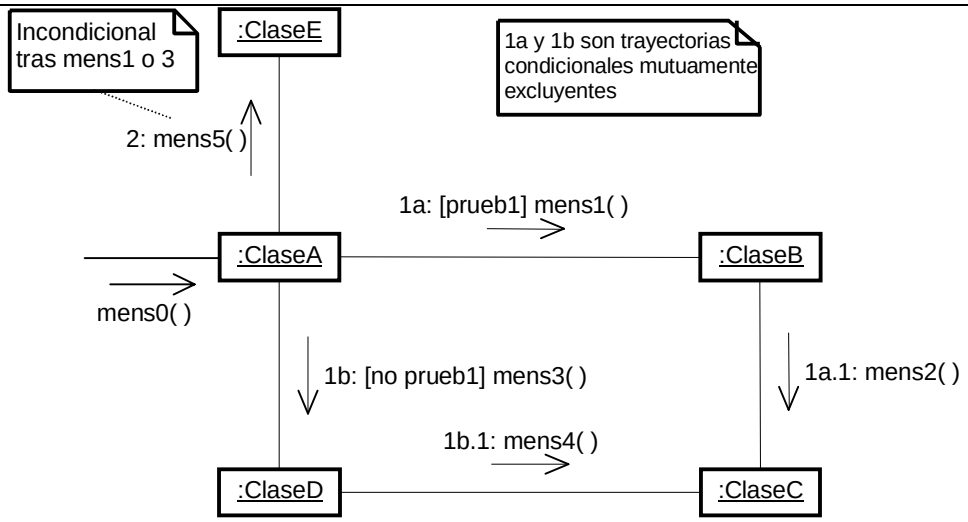
Ejemplo: numeración de secuencias.



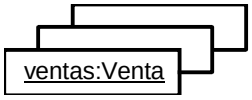
Ejemplo con mensaje condicional



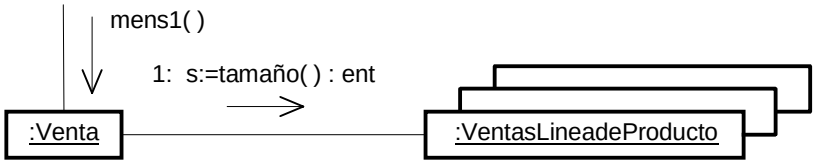
Ejemplo con trayectorias condicionales mutuamente excluyentes:



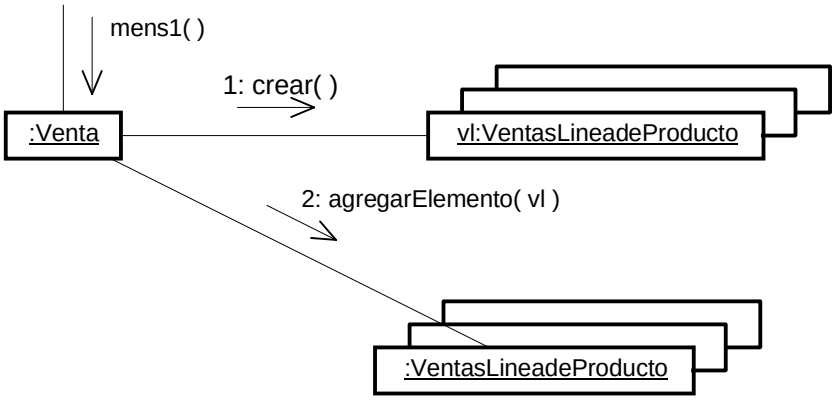
Representación de multiobjetos o conjuntos de instancias:



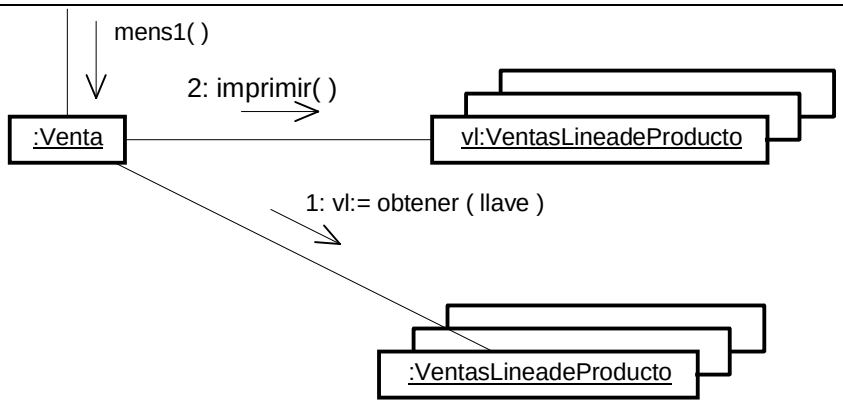
Ejemplo 1:



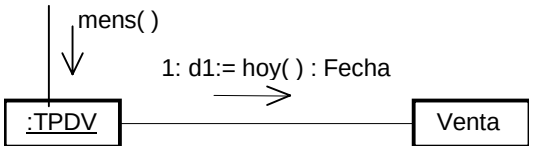
Ejemplo 2:



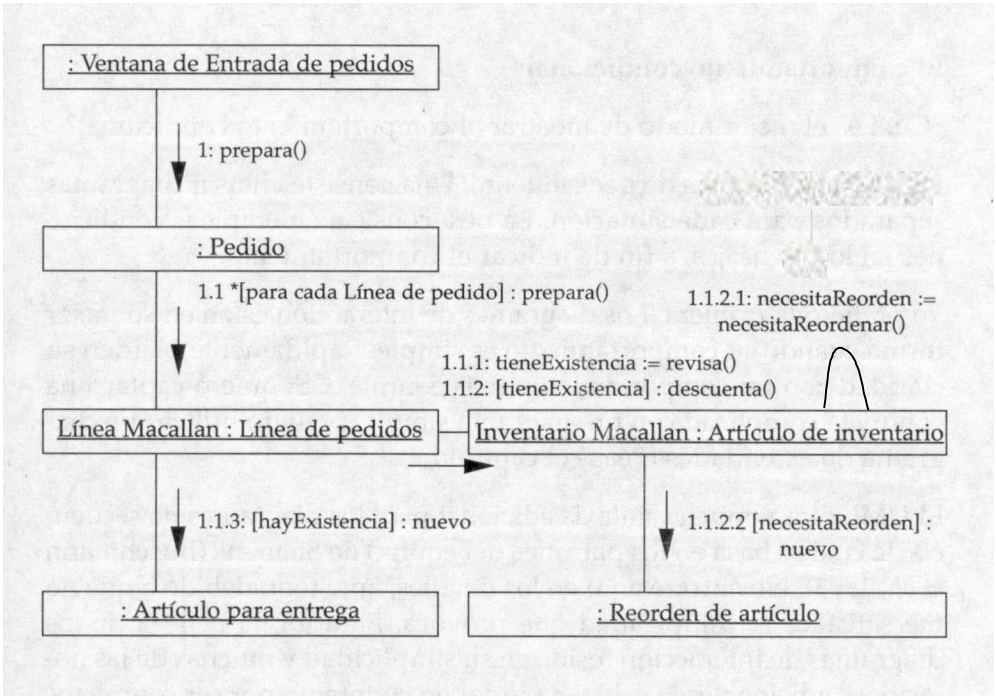
Ejemplo 3:



Ejemplo: mensaje enviado a una clase.



Un ejemplo más completo:



Elaboración de diagramas de colaboración:

El apoyo para crear los diagramas de colaboración es la documentación previa:

- Los **casos de uso** indican los eventos del sistema que se muestran explícitamente en los **diagramas de la secuencia** el mismo.
- La mejor conjetura inicial sobre el efecto producido por los eventos del sistema se describe en los contratos de sus operaciones.
- Los eventos del sistema -comúnmente de interfaz con el usuario u otro medio- representan iniciadores de los diagramas de interacción que describen visualmente cómo los objetos interactúan para realizar las tareas en cuestión.

Para cada evento externo al proceso que se está modelando se debe de construir un diagrama de colaboración cuyo mensaje inicial sea el de sus eventos.

La base son los diagramas de secuencia, pero los diagramas de colaboración en esta etapa tienen información añadida.

Se deben conjuntar las diferentes opciones de los cursos normales y alternos de los casos de uso.

Apoyarse en los contratos de operaciones para crear diagramas con una mayor riqueza. Revisar las poscondiciones de los contratos. Ya que los diagramas de colaboración deben cumplir con estas.

Los casos de uso reales dan una idea de la interacción externa.

Visibilidad.

Es la capacidad de un objeto para ver otro o hacer referencia a él.

Los diagramas de colaboración describen gráficamente los mensajes entre objetos. **Para que un objeto emisor envíe un mensaje a un objeto receptor, el emisor tiene que ser visible a este.**

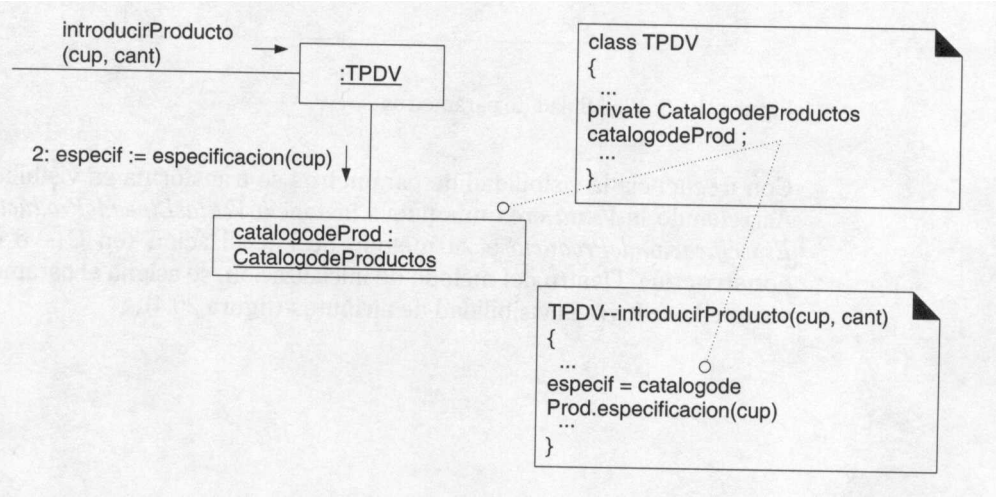
La visibilidad se refiere al alcance o ámbito de los objetos. Existen cuatro formas comunes de visibilidad:

1. Visibilidad de atributos.
2. Visibilidad de parámetros.
3. Visibilidad declarada localmente.
4. Visibilidad global.

Visibilidad de atributos.

Existe visibilidad de atributos de A a B cuando B es un atributo de A.

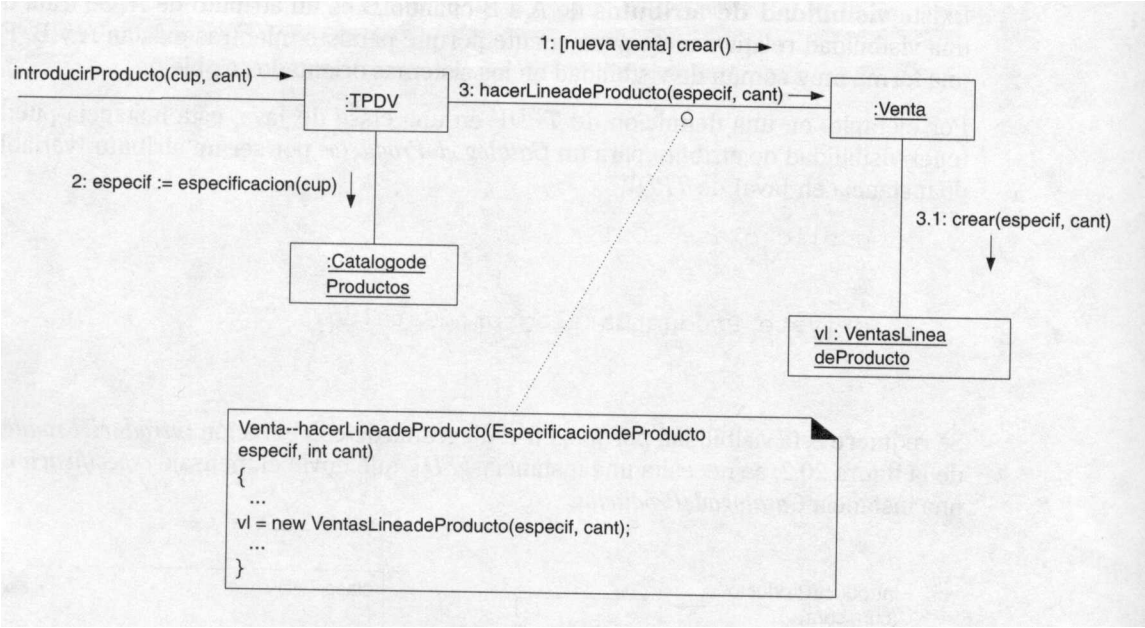
Se trata de una visibilidad relativamente permanente porque persiste mientras existan A y B. Es la forma más común de visibilidad.



Visibilidad de parámetros.

Existe visibilidad de parámetros de A a B cuando B se transmite como un parámetro a un método de A.

Es una visibilidad relativamente temporal porque persiste sólo dentro del ámbito del método. Es la segunda forma más común de visibilidad.

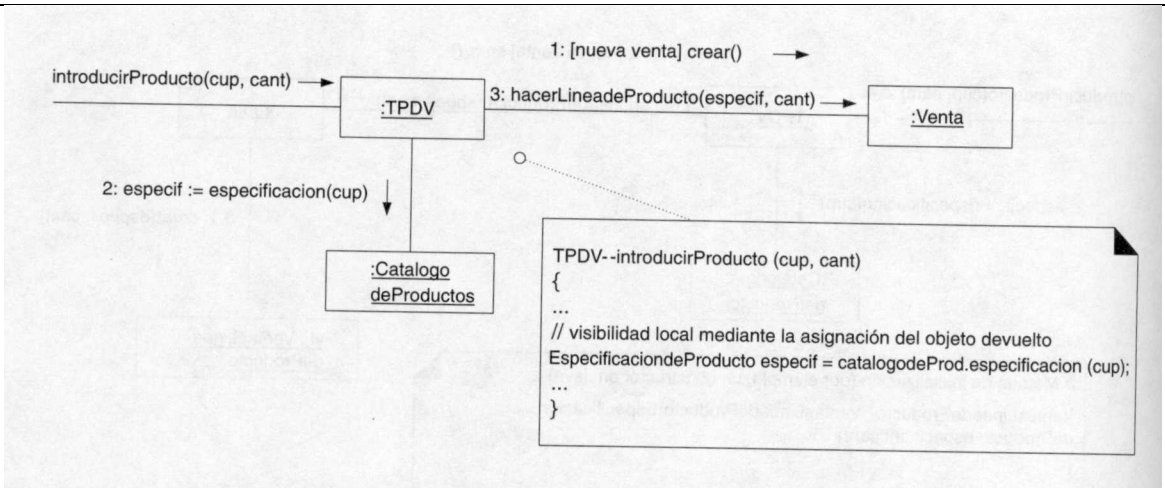


Visibilidad declarada localmente.

Existe visibilidad declarada localmente de A a B cuando se declara que B es un objeto local dentro de un método de A.

Visibilidad relativamente temporal porque persiste sólo dentro del ámbito del método. Puede ser de dos formas:

- Al crear una nueva instancia local y asignarla a una variable local.
- Al asignar a una variable local el objeto devuelto proveniente de la llamada a un método.



Visibilidad global.

Existe visibilidad global de a a B cuando B es global para A.

Visibilidad relativamente permanente, porque persiste mientras existan A y B. Es el tipo de visibilidad menos frecuente - y menos conveniente - en los sistemas orientados a objetos.

Notación opcional en UML para indicar la visibilidad.

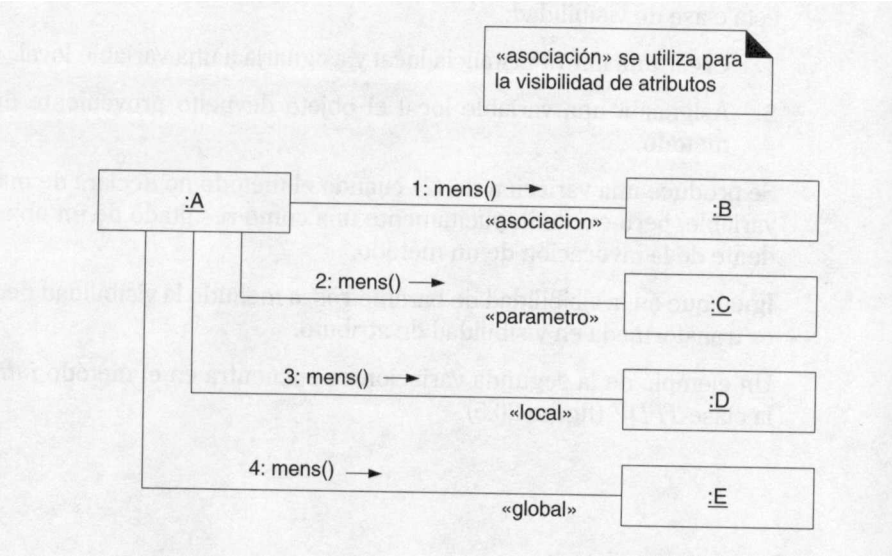


Diagrama de clases orientado al diseño.

El diagrama de clases en la etapa de diseño toma la perspectiva de especificación, donde se incluye las clases de software que participan en la solución y se complementan con detalles de diseño.

Para crear el diagrama de clases de diseño es necesario contar antes con los diagramas de interacción y por supuesto con el modelo conceptual.

En la práctica, el diagrama de clases de diseño suele prepararse a la par de los diagramas de colaboración.

La intención del diagrama es describir gráficamente las especificaciones de las clases de software y de las interfaces - como en Java - en una aplicación.

Normalmente contiene la siguiente información:

- Clases, asociaciones y atributos.
- Interfaces.
- Métodos.
- Información sobre los tipos de los atributos.
- Navegabilidad.
- Dependencias.

Recordar que un modelo conceptual o diagrama de clase bajo el modelo conceptual contiene definiciones de conceptos del mundo real, mientras que el diagrama de clases de diseño - o del modelo de especificación - contiene definiciones de las entidades del software.

Proceso para elaborar un diagrama de clases orientado al diseño.

1. **Identificar las clases que participan en la solución del software. Apoyarse en los diagramas de interacción.**

   Comúnmente basta con examinar los diagramas de interacción y listar las clases que aparecen. Se parte del hecho que los diagramas de interacción muestran en forma exhaustiva el comportamiento de los objetos del sistema.

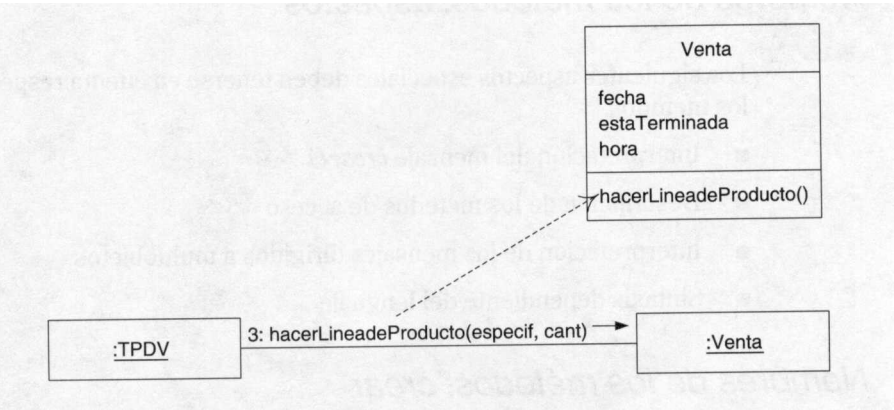
   Es posible que no aparezcan algunas clases del modelo conceptual. Es posible que en posteriores ciclos del desarrollo sea necesario incorporarlos.

2. **Dibujar el diagrama de clases inicial.**

3. **Copiar los atributos provenientes de los conceptos asociados del modelo conceptual.**

4. **Agregar los nombres de los métodos analizando los diagramas de interacción.**

   Para identificar los métodos de las clases se analizan los diagramas de colaboración.



   En términos generales, el conjunto de los mensajes enviados a la clase X a través de los diagramas de colaboración indica la mayoría de los métodos que ha de definir la

clase X.

Interpretación del mensaje *crear()*. Es la forma independiente del lenguaje UML con que se indica instanciación o inicialización. Al traducir a un lenguaje de programación hay que expresarlo en términos de sus expresiones de instanciación.

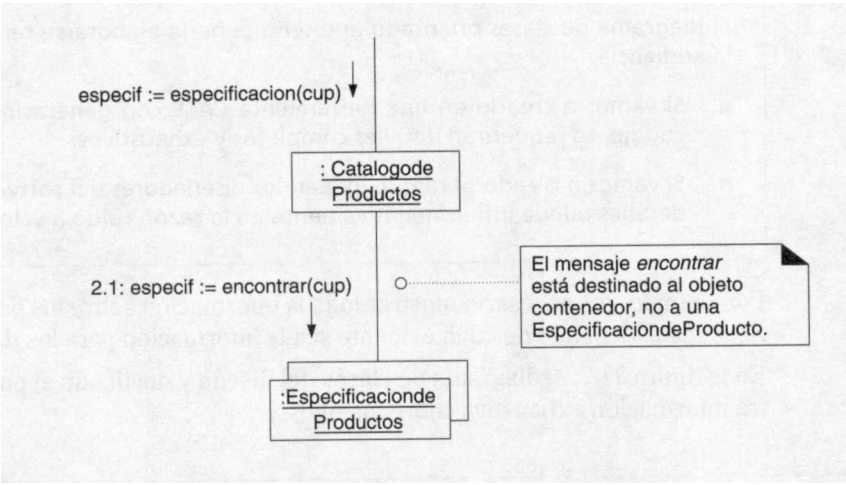
En el diagrama de clase se acostumbra omitir los métodos relacionados con la creación y los constructores procedentes del diagrama de clases.

Métodos de acceso. Los métodos de acceso son aquellos que recuperan o establecen el valor de los atributos (comúnmente llamados métodos *get* y *set*).

Normalmente estos métodos no se incluyen en el diagrama de clases porque saturan el diagrama sin aportar nada a él: para N atributos hay 2N métodos sin el menor interés.

Multiobjetos. Un mensaje a un multiobjeto se interpreta como si estuviera destinado al objeto contenedor / colección.

Por ejemplo, un método *encontrar*, que busque a un objeto dentro de una colección forma parte de la clase colector y no de la del objeto contenido. Por lo tanto sería incorrecto agregar al método encontrar a la clase contenida en el colector.



Sintaxis independiente del lenguaje. Se recomienda usar el formato básico de UML, aun cuando el lenguaje de implementación que se planea utilizar aplique otra sintaxis.

5. **Incorporar la información sobre los tipos a los atributos y a los métodos.**

   Es opcional mostrar el tipo de los atributos, los parámetros del método y los valores de retorno del método. Tomar en consideración dos aspectos:

- Si el diagrama se va a crear en una herramienta CASE con generación automática de código, se requerirán detalles completos y exhaustivos.
- Si se va a crear para que lo lean los diseñadores del software, el exceso de detalles puede influir negativamente.

6. **Agregar las asociaciones necesarias para dar soporte a la visibilidad requerida de los atributos.**

   Las asociaciones se escogen con un riguroso criterio que esté orientado al software, donde el punto sea establecer las asociaciones que se requieren para satisfacer con los diagramas de interacción la visibilidad y las necesidades constantes de memoria.

7. **Agregar las flechas de navegabilidad a las asociaciones para indicar la dirección de la visibilidad de los atributos.**

   Se da el nombre de papel al fin de una asociación. En UML se puede añadir al papel una flecha de navegabilidad. La navegabilidad es una propiedad del papel e indica la posibilidad de navegar unidireccionalmente en una asociación, desde los objetos fuente hasta la clase destino.

   La navegabilidad significa visibilidad; generalmente visibilidad de atributos. Suele traducirse como si la clase fuente tuviera un atributo que se refiere a una instancia de la clase destino.

   La visibilidad y las asociaciones requeridas entre clases se indican con los

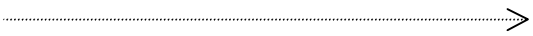
diagramas de interacción.

Existen tres situaciones principales que muestran la necesidad de definir una asociación con una flecha de navegabilidad de A a B.

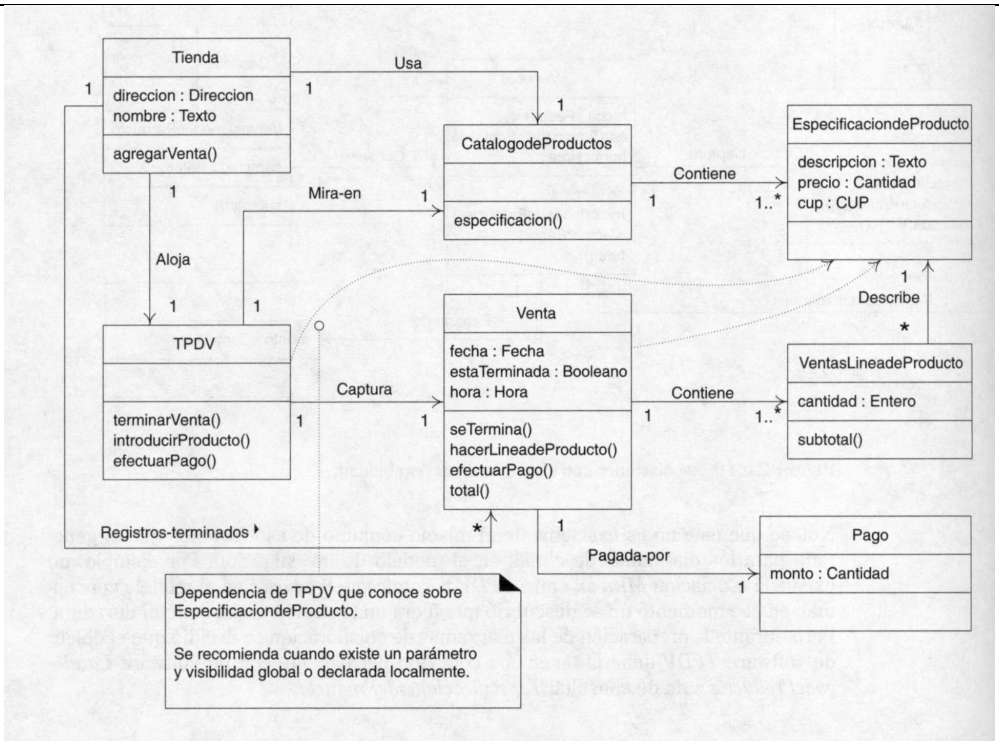
- A envía un mensaje a B.
- A crea una instancia B.
- A necesita mantener una conexión con B.

8. Agregar las líneas de las relaciones de dependencia para indicar la visibilidad no relacionada con los atributos.

UML incluye una *relación general de dependencia*, la cual indica que un elemento conoce la existencia de otro. No está limitado al diagrama de clases y se denota con una línea punteada y una flecha.

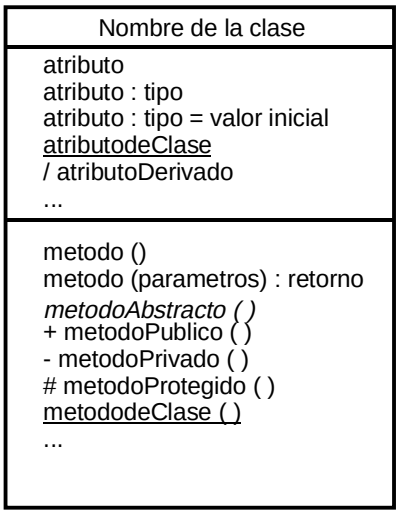


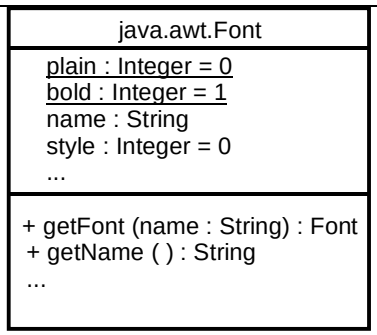
En el diagrama de clases, la relación de dependencia sirve para describir la visibilidad entre atributos que no se relaciona con ellos: la visibilidad de parámetros, local o declarada localmente.



Notación en UML para miembros de clase.

Ya se han mencionado algunas características de la notación de los miembros de una clase, especificando la notación para tributos de clase. Ahora se verán algunas notaciones adicionales para miembros de clase.




java.awt.Font
plain : Integer = 0
bold : Integer = 1
name : String
style : Integer = 0
...
+ getFont (name : String) : Font
+ getName ( ) : String
...

Codificación.

Terminada cada iteración de análisis y diseño se debe pasa a la etapa de codificación. A continuación se presenta el orden sugerido:

1. Creación de las definiciones de clase a partir de los diagramas de clase del modelo de especificación.

   - Añadir los atributos de referencia. Un atributo de referencia es aque que remite a un objeto complejo y no a un tipo de dato simple. Están indicados por las asociaciones y la navegabilidad.

     Si se usan un nombre de papel en la asociación, puede usarse para nombrar al atributo de referencia si este se encuentra del lado de la clase que debe ser visible para la clase origen.

2. Especificación de los métodos a partir de los diagramas de colaboración y los contratos de operaciones.

3. Actualización de las definiciones de las clases. A partir de la codificación se pueden realizar algunas modificaciones por diversas cuestiones, como agregar un método auxiliar. Es necesario actualizar con la nueva información el diagrama de clases del diseño y la documentación complementaria.

4. Implementación de las clases de la menos a la más acoplada.

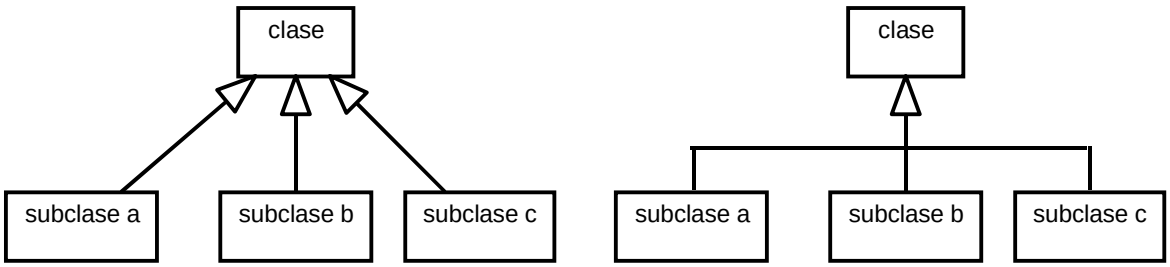
Refinamiento del modelo conceptual.

Generalización.

Es la actividad de identificar los aspectos comunes de los conceptos y en definir las relaciones entre la superclase y la subclase. Es una forma de efectuar clasificaciones taxonómicas entre los conceptos que después se clasifican en jerarquías de clases.

Notación de UML

La relación de generalización entre los elementos se indica con una punta de flecha grande y hueca que señala el elemento más general partiendo del más especializado.



Conceptos e identificación de superclases y subclases.

La definición de una superclase es más general o amplia que la de una subclase.

Las subclases y las superclases están relacionadas por su pertenencia a un conjunto. Todos los miembros de un conjunto de subclase pertenecen al respectivo conjunto de superclase.

Regla del 100%

Cuando se crea una jerarquía de clases, se hacen afirmaciones de las superclases que se aplican también a las subclases.

El 100% de la definición de la superclase debería aplicarse también a la subclase. Ésta ha de conformarse con la superclase al 100% de sus atributos y asociaciones.

Regla es-un.

Una subclase debería ser un miembro del conjunto de la superclase.

Todos los miembros de un conjunto de subclases han de pertenecer a su conjunto de superclases.

La *subclase* **es un** tipo de *superclase*

Una subclase potencial deberá asumir de conformidad:

- La regla del 100%. *Conformidad con la definición.*
- La regla es-un. *Conformidad con la pertenencia a un conjunto.*

Generar subclases a partir de una clase.

Se crea una subclase de una superclases cuando:

1. La subclase tiene otros atributos más de interés.
2. La subclase tiene otras asociaciones más de interés.
3. Se opera sobre el concepto de la subclase, se maneja, se reaccione ante ella o se manipula de modo diferente a como se haría con la superclase u otras subclases.
4. El concepto de subclase representa una cosa animada que se comporta de manera distinta a la superclase o a otra subclase en aspectos que resultan relevantes.

Generar una superclase.

Se crea una superclase en una relación de generalización con subclases cuando:

- Las subclases potenciales representan variaciones de un concepto semejante.
- Las subclases se conforman a las reglas del *100%* y de *es-un*.
- Todas las subclases con un mismo atributo pueden factorizarse y expresarse en la superclase.
- Todas las subclases tienen la misma asociación que puede factorizarse y relacionarse con la superclase.

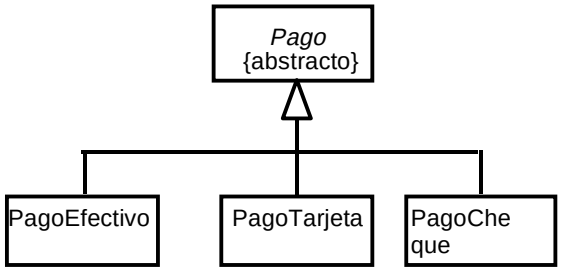
Clases abstractas.

Las clases abstractas se tienen que identificar desde el modelo conceptual pues limitan la posibilidad de crear objetos directamente de esas clases.

Se le da el nombre de clase abstracta a una clase C, si todos sus miembros han de ser también miembros de una subclase de C.

Notación de UML.

Al igual que en un método abstracto, una clase abstracta se representa con su nombre en cursivas. Puede añadirse o sustituirse una restricción del tipo: **{abstracto}**

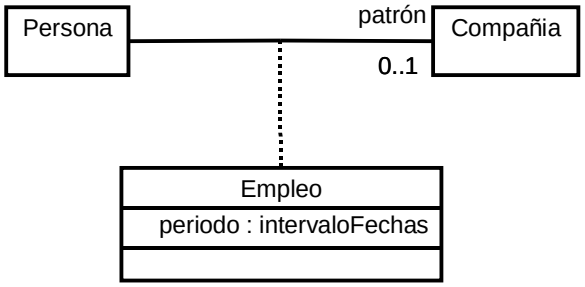


Clase de asociación.

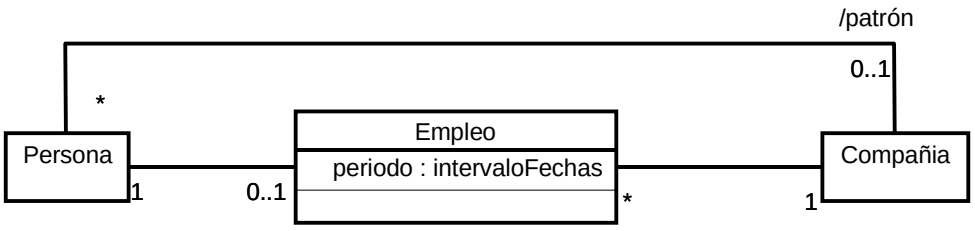
Las clases de asociación permiten añadir atributos, operaciones y otras características a las asociaciones.

Suponga que una Persona puede trabajar solo para una Compañía, y se necesita almacenar la información sobre el periodo de tiempo que trabaja cada empleado para cada Compañía.

Es necesario por lo tanto un atributo intervaloFechas que no es de la clase Persona y menos aún de Compañía. La solución sería añadirlo como atributo de la asociación.



Otra representación sería *promover* la clase de asociación a una clase completa:



La clase de asociación en UML determina que sólo puede haber una instancia de la clase de asociación entre dos objetos cualquiera participantes. Si se quieren más instancias sería necesario convertir la clase de asociación en una clase completa.

La clase de asociación es también utilizada al tener una asociación de muchos a muchos.

Relaciones de agregación y composición.

Se dice que la agregación es la relación de componente y es común dar ejemplos donde una entidad física está formada por componentes: un carro tiene como componentes el motor, las ruedas, etc.

Ahora bien, ¿cuál es la diferencia entre **agregación** y **asociación**?

Curiosamente ni los expertos se han puesto de acuerdo en una definición aceptada por todos de la diferencia entre agregación y asociación. Por ejemplo:

- Peter Coad ejemplificó la agregación como la relación entre una organización y sus empleados.
- Jim Rumbaugh afirmó que una compañía no es la agregación de sus empleados.

UML además de la agregación simple maneja otra variedad conocida como composición. Tanto la agregación como la composición se consideran una propiedad de un papel de asociación.

Agregación.

También conocida como agregación compartida significa que la multiplicidad en el extremo del todo o compuesto puede ser más de una. Es decir, la parte puede estar en muchas instancias compuestas.

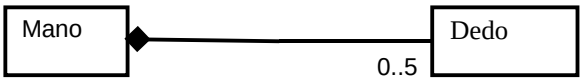
Se representa con un diamante o rombo en blanco.

Composición.

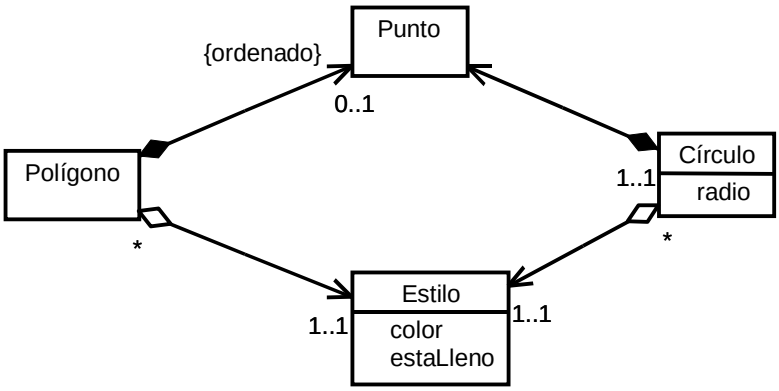
En la composición, el objeto componente puede pertenecer a un todo o compuesto único, y se espera que las partes vivan y mueran con el todo. Nadie más puede contener al componente.

Cualquier borrado del todo se extiende en cascada a sus partes. No olvidar sin embargo que el borrado en cascada está implícito en cualquier papel con multiplicidad de 1..1.

Se representa en UML con un diamante sombreado en el extremo del compuesto.



Ejemplo:



Identificación de la agregación.

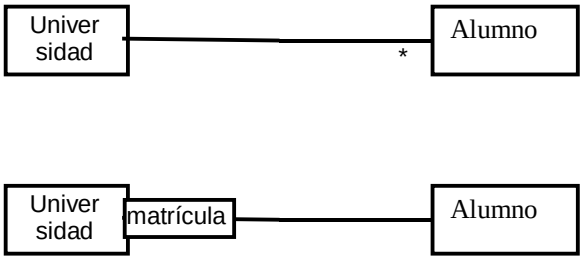
- Si la duración de la parte es dependiente de la que tiene el compuesto.
- Si representa un ensamble físico o lógico de parte-todo.
- Si algunas propiedades del compuesto se difunden hacia las partes, por ejemplo la ubicación.
- Si las operaciones aplicadas al compuesto se propagan hacia las partes, como el movimiento, grabación, destrucción.

Asociaciones calificadas.

Una asociación calificada relaciona clases de objetos con un calificador, el cual es un atributo que reduce la multiplicidad efectiva de una asociación.

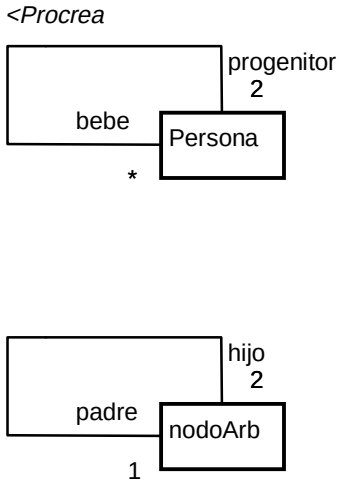
- El calificativo reduce la multiplicidad del lado opuesto de la asociación.
- Se comporta en términos de implementación como un índice de la asociación.

La notación es una pequeña caja en la asociación del lado opuesto de la multiplicidad que se esta reduciendo.



Asociaciones reflexivas.

Una clase puede estar relacionada consigo misma, relacionando distintos objetos de una misma clase.



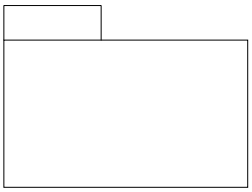
Diagramas de paquetes.

En muchas ocasiones es necesario fragmentar un sistema grande en sistemas más pequeños. En principio la programación orientada a objetos ofrece un gran cambio con respecto a la descomposición funcional. A pesar de esto, a veces tenemos en un sistema demasiadas clases y seria conveniente separarlas.

UML apoya el concepto de agrupar las clases en unidades de nivel más alto, este agrupamiento se conoce como **paquete**.

En general el concepto de paquete puede ser utilizado para agrupar cualquier elemento del modelo, pero lo común es utilizarlo para las clases. Un diagrama de paquetes serviría entonces para mostrar los paquetes de clases y las dependencias entre ellos.

Un paquete se representan en UML de la siguiente forma:



Por el otro lado están las dependencias. Se dice que existe una dependencia entre dos elementos si los cambios a la definición de un elemento pueden causar cambios al otro. En términos de clases, la dependencia puede darse por varias razones:

- Una clase envía un mensaje a otra.
- Una clase tiene a otra como parte de sus datos.
- Una clase menciona a otra como parámetro de una operación.

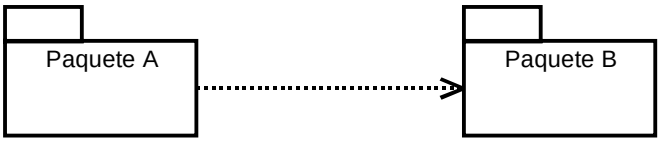
En general la dependencia esta relacionada con la visibilidad de una clase sobre otra.

En teoría, sólo los cambios a la interfaz de la clase deberían afectar a otra clase, ya que los mensajes pueden dejar de ser válidos. El diseño de nuestras clases debería reducir o minimizar las dependencias, reduciendo los efectos del cambio a la interfaz y siendo por lo tanto más fácil realizar cambios a un sistema.

En base a las dependencias de las clases se puede inferir la dependencia entre paquetes de clases:

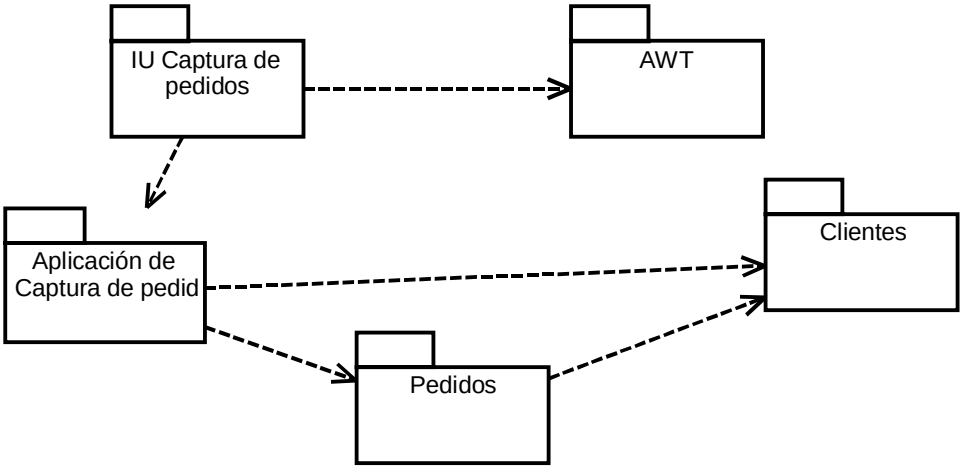
Existe una dependencia entre dos paquetes si existe algún tipo de dependencia entre dos clases cualquiera en los paquetes.

La dependencia de un paquete a otro se muestra de la siguiente forma:

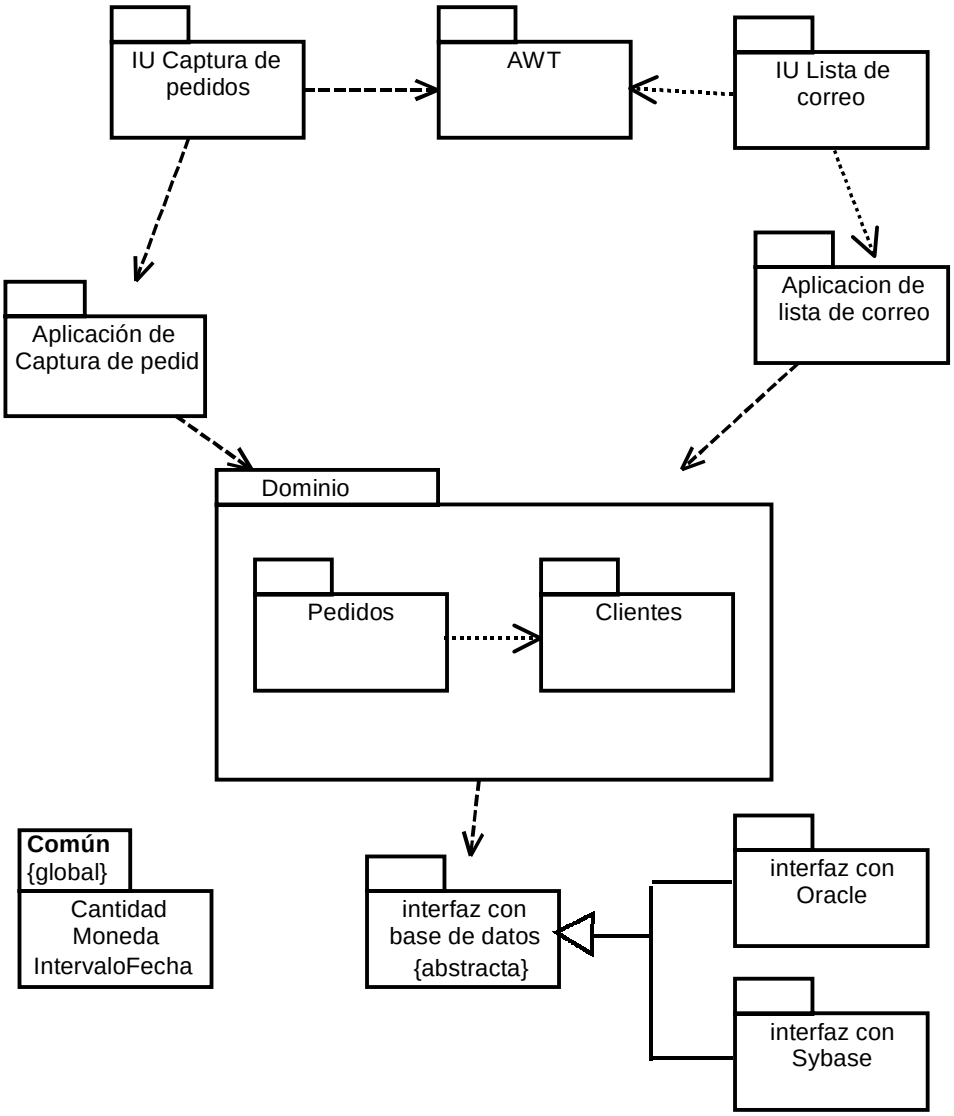


La dependencia no se considera transitiva. De modo que si el paquete B tuviera una dependencia hacia un paquete C; el paquete a no tendría necesariamente una dependencia hacia el paquete C.

Lo que si nos dice la dependencia es que si un paquete A tiene una dependencia hacia un paquete B el paquete A puede ver todas las clases públicas del paquete y sus métodos públicos. (además de los atributos públicos si los hubiera)



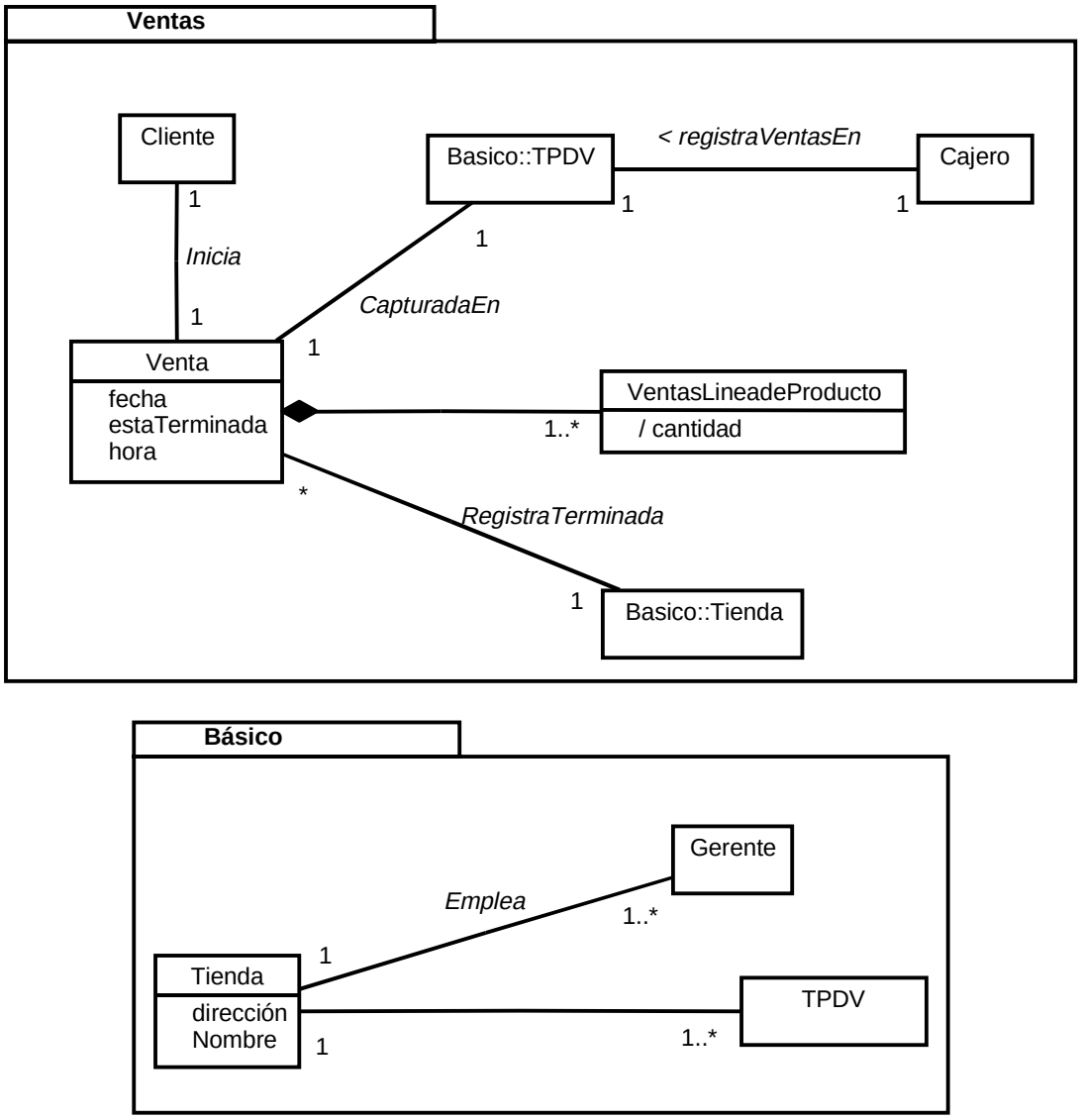
El diagrama anterior muestra los paquetes y su dependencia. Además, puede mostrarse en un diagrama de paquetes de otro nivel el contenido de un paquete, que puede ser una lista de clases, otro diagrama de paquetes o un diagrama de clases.



En este diagrama podemos apreciar:

- Un paquete llamado Común marcado como {global}, indicando que todos los paquetes del sistema tienen una dependencia hacia el.
- Uso de generalización, indicando que el paquete especializado debe atenerse a la interfaz del paquete general. El paquete genérico sólo define la interfaz (es abstracto)

- Uso de paquetes anidados -paquete Dominio- y listado de clases -paquete Común- .



La división implica tratar de mantener las dependencias al mínimo y evitar en lo posible los ciclos de la estructura de dependencias.

Una forma de eliminar los ciclos es utilizar la generalización de paquetes. Esta implica una dependencia de la subclase a la superclase.

Los paquetes son una herramienta vital para los proyectos grandes. Proporcionan mayor legibilidad a diagramas de clases que se han vuelto demasiado grandes y ayudan a

realizar pruebas a nivel de paquete.

Diagramas de Estado.

Los diagramas de estado ayudan a determinar el comportamiento de los sistemas, ya que muestran gráficamente los eventos y estados de los objetos.

Existen varios usos para los diagramas de estado

- Diagramas de estados de casos de uso
- Diagramas de estado del sistema
- Diagramas de estado de los objetos.

Hay autores que apoyan cada uno de los tipos de diagramas de estado. Para fines de nuestro curso y por considerarlos de mayor utilidad se trabajara únicamente sobre diagramas de estado de los objetos.[5]

Un diagrama de estados de un objeto describe todos los estados posibles en los que puede estar un objeto determinado y la manera en que cambia de un estado a otro a partir de los eventos que recibe. Este diagrama por lo tanto representa el comportamiento del objeto durante todo su ciclo de vida.

Elementos del diagrama de estados.

Estado. Es la condición de un objeto en un momento determinado. Está definido por los valores de los atributos del objeto. El estado especifica la respuesta de un objeto a los posibles eventos de entrada.

Ejemplo: El estado de un teléfono mientras no recibe llamada puede ser *ocioso.*

- La respuesta de un objeto puede incluir un cambio de estado.
- El estado corresponde al intervalo entre dos eventos recibidos por un objeto.

- Los estados ocupan tiempo.

Evento. Es un acontecimiento importante o digno de señalar.

Ejemplo: Levantar el auricular de un teléfono.

Ejemplo: El avión despega.

Ejemplo: El disco duro falló.

- Un evento es un estímulo de un objeto a otro.
- Puede ser una señal de que un acontecimiento ha ocurrido.
- Un evento separa dos estados.
- La respuesta a un evento depende del estado del objeto que lo recibe.
- Los eventos representan puntos instantáneos en el tiempo.
- Los eventos se derivan de las operaciones definidas en los diagramas de secuencia.

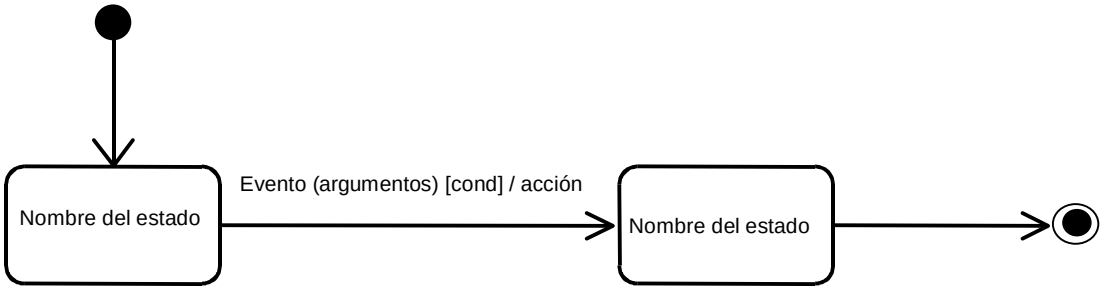
Transición. Es una relación entre dos estados. La transición nos dice que cuando ocurre un evento, el objeto pasa del estado actual aun nuevo estado.

Ejemplo: El evento *levantar el auricular* puede pasar al teléfono de un estado *ocioso* a un estado *activo*.

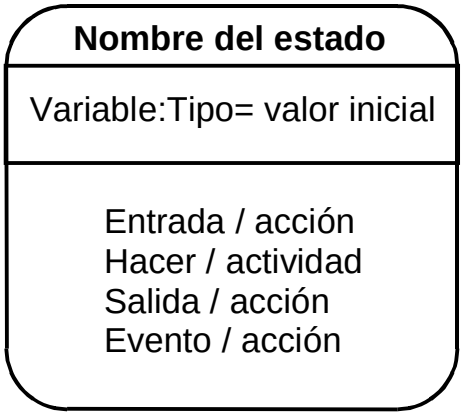
- Una transición se dispara cuando un evento ocurre.

Notación en UML

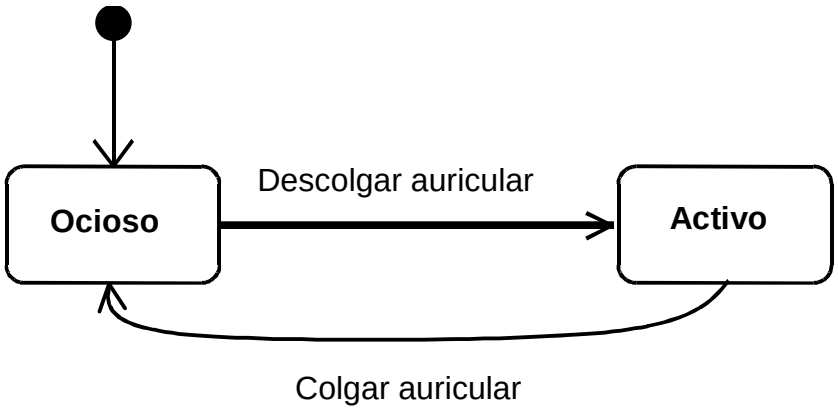
[5] En [1] se dice que en la mayor parte de las técnicas OO los diagramas de estados se dibujan para una sola clase.



Representación completa de un estado:



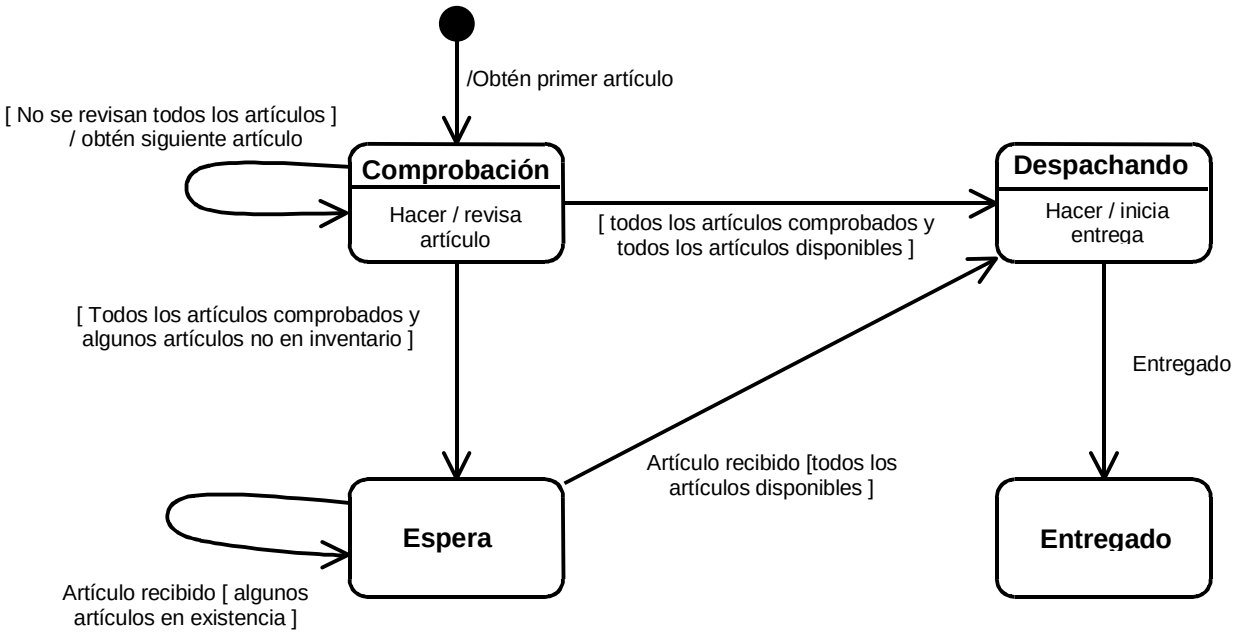
Ejemplo: diagrama de estados para un teléfono:



Como en la mayoría de los diagramas de UML, el nivel de detalle puede variar de acuerdo a la información que se quiera representar.

La sintaxis completa de una etiqueta de transición tiene tres partes opcionales:

- **Evento**.
- **Condición**. También conocido como guardia, es una condición lógica que de acuerdo a su valor de verdad o falsedad determina si se lleva a cabo la transición.
- **Acción**. Se asocian con un evento y se consideran como procesos que suceden con rapidez y no son interrumpibles.



Elementos de un estado:

Evento de **entrada**. Se ejecuta siempre que se entre al estado a través de una transición.

Evento de **salida**. Se ejecuta siempre que se sale del estado por medio de una transición.

Actividad. Es el proceso que debe realizar el estado. Una actividad puede ser interrumpida por un evento que cambia al objeto de un estado a otro.

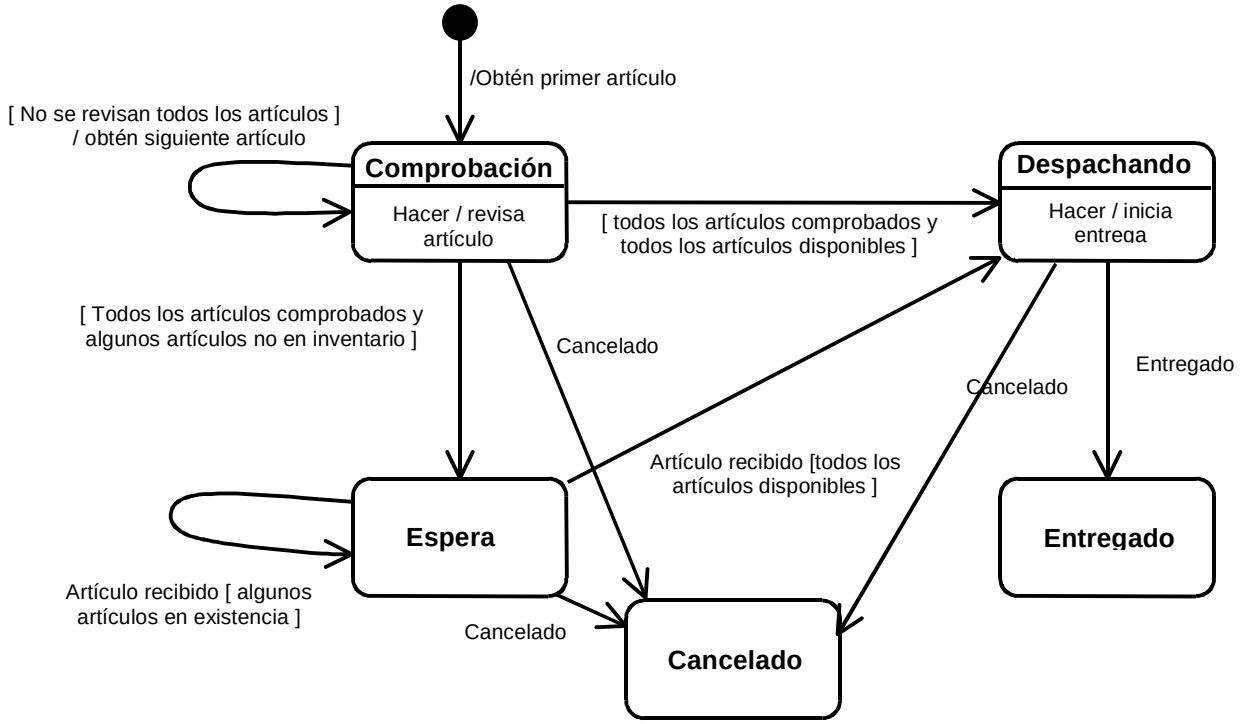
Autotransición. Una transición que vuelve al mismo estado. ¿Qué sucede si ese estado tiene eventos de entrada y/o salida?

Superestado.

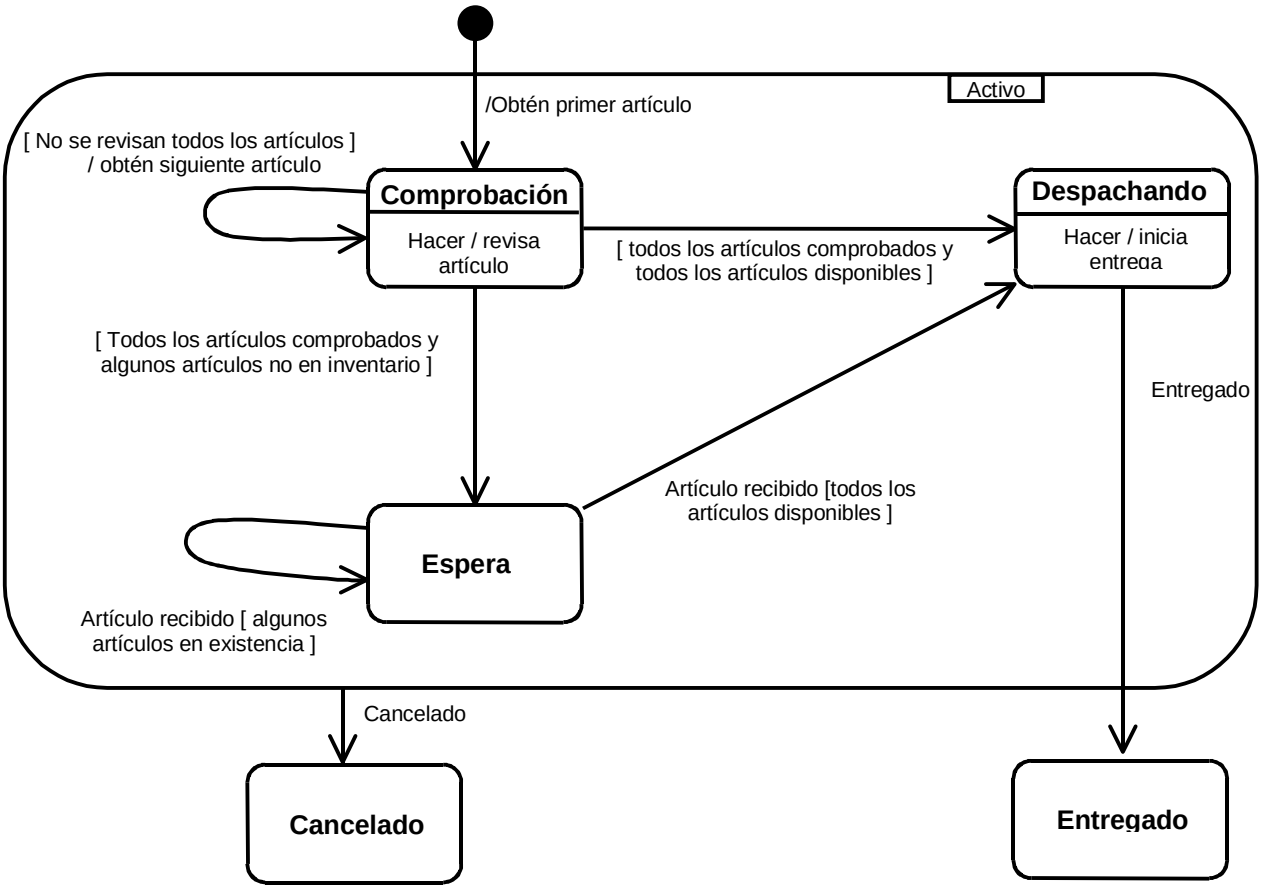
Es posible crear un superestado que contenga a un conjunto de subestados. Los subestados heredan todas las transiciones sobre el superestado.

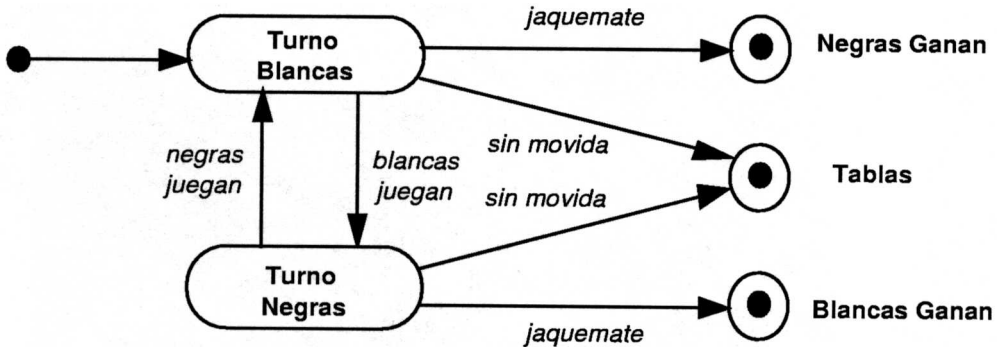
Un superestado nos ayuda a reducir la saturación de transiciones entre estados.

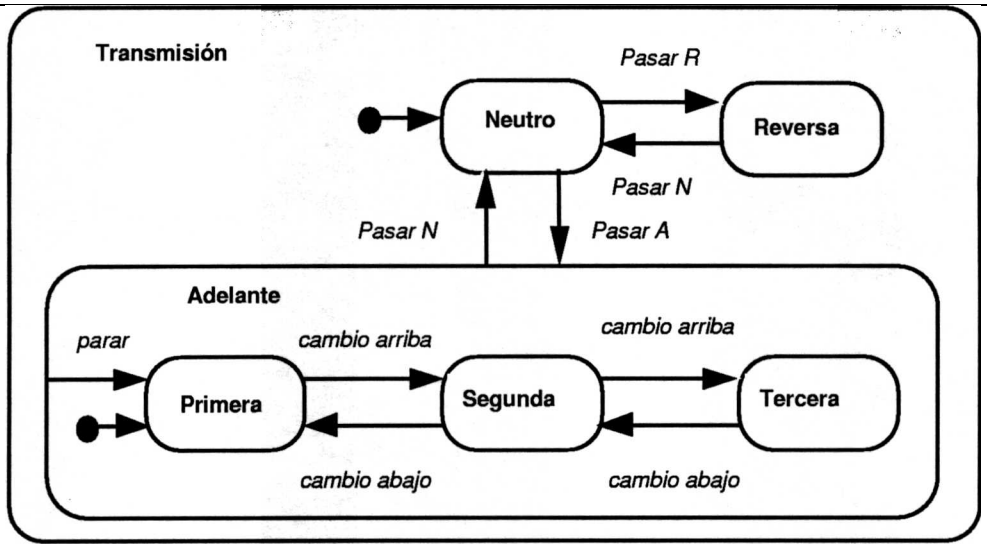
Considere el ejemplo anterior en el que fuera necesario cancelar el pedido desde cualquier punto antes de que sea entregado.



Representado con un superestado:


Activo
/Obtén primer artículo
[No se revisan todos los artículos]
/ obtén siguiente artículo
Comprobación
Hacer / revisa artículo
[todos los artículos comprobados y todos los artículos disponibles]
Despachando
Hacer / inicia entrega
[Todos los artículos comprobados y algunos artículos no en inventario]
Entregado
Artículo recibido [todos los artículos disponibles]
Espera
Artículo recibido [algunos artículos en existencia]
Cancelado
Cancelado
Entregado

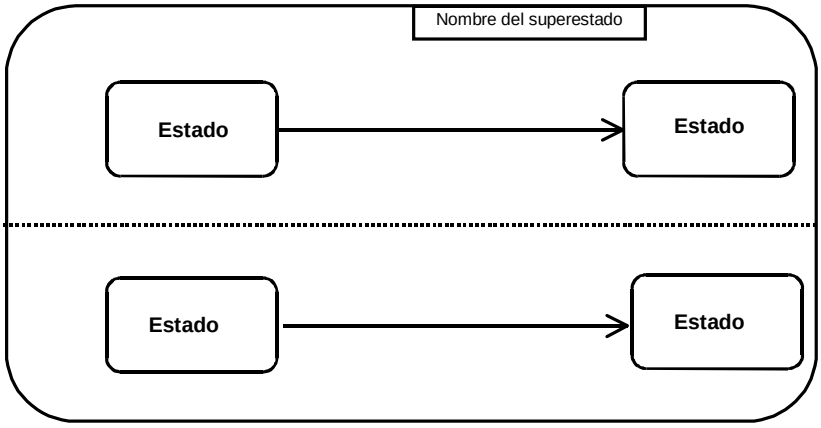

Turno Blancas
jaquemate
Negras Ganan
negras juegan
blancas juegan
sin movida
sin movida
Tablas
Turno Negras
jaquemate
Blancas Ganan


Transmisión
Pasar R
Neutro
Reversa
Pasar N
Pasar N
Pasar A
Adelante
parar
cambio arriba
cambio arriba
Primera
Segunda
Tercera
cambio abajo
cambio abajo

Diagramas de estados concurrentes.

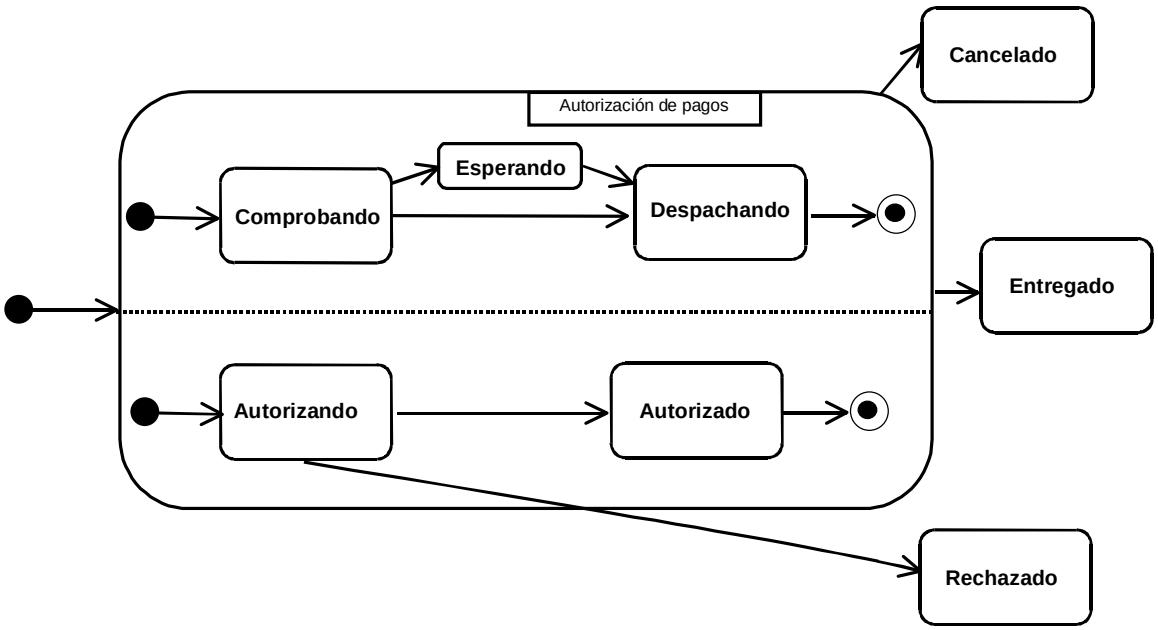
Es posible que sea necesario modelar diagramas de estados concurrentes. Estos son útiles cuando un objeto dado tiene conjuntos de comportamientos independientes.

Sin embargo, no se deben permitir demasiados comportamientos concurrentes para un objeto. Si este es el caso se deberá considerar dividir el objeto en varios.



Las secciones concurrentes de un diagrama de estados son lugares en los que en cualquier punto, el objeto está en estados diferentes, uno por cada diagrama.

Ejemplo:

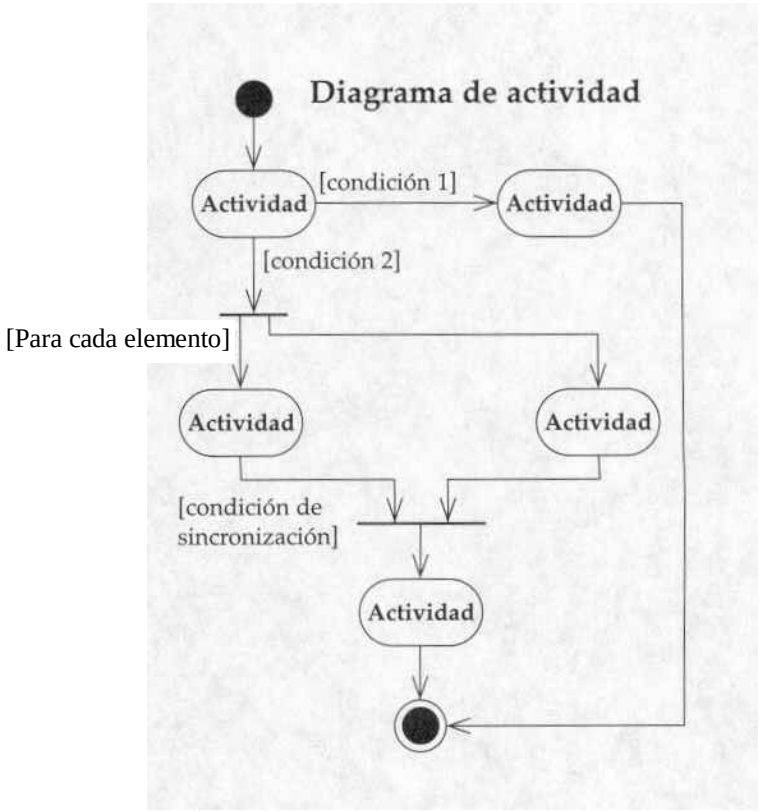


Diagramas de actividades.

Los diagramas de actividades combinan ideas de diversas técnicas no proporcionadas originalmente por los tres amigos: diagramas de eventos de Jim Odell, técnicas de modelado de estados y redes de Petri.

Los diagramas de actividades son ocupados esencialmente para descripciones de comportamiento de los sistemas que contienen procesos potencialmente paralelos.

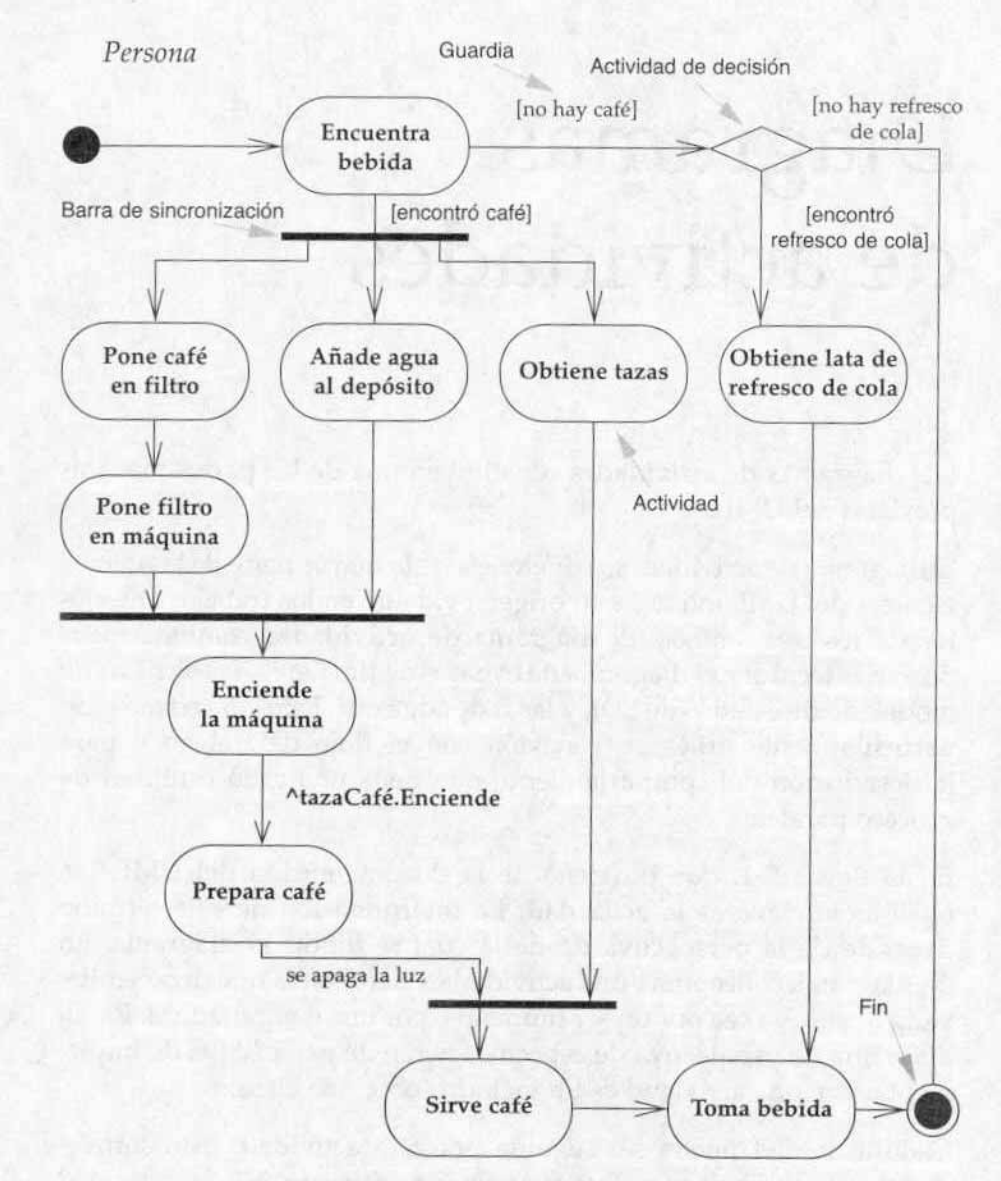
La representación en UML es de la siguiente forma:



En este diagrama se reconocen las actividades. Una actividad es desde el modelo conceptual una tarea que debe ser llevada a cabo por una persona o por una computadora. Bajo el modelo de especificación o implementación, una actividad es un método de una clase.

Un diagrama de actividades tiene similitudes con los diagramas de flujo. Sin embargo, estos últimos se limitan a procesos secuenciales. Los diagramas de actividades pueden manejar procesos paralelos, determinando el orden en que se harán las actividades.

El siguiente ejemplo proporcionado por UML nos muestra un diagrama de actividades para una persona que quiere tomar café o refresco en su defecto.



Es evidente que estos diagramas se prestan para modelar programas concurrentes,

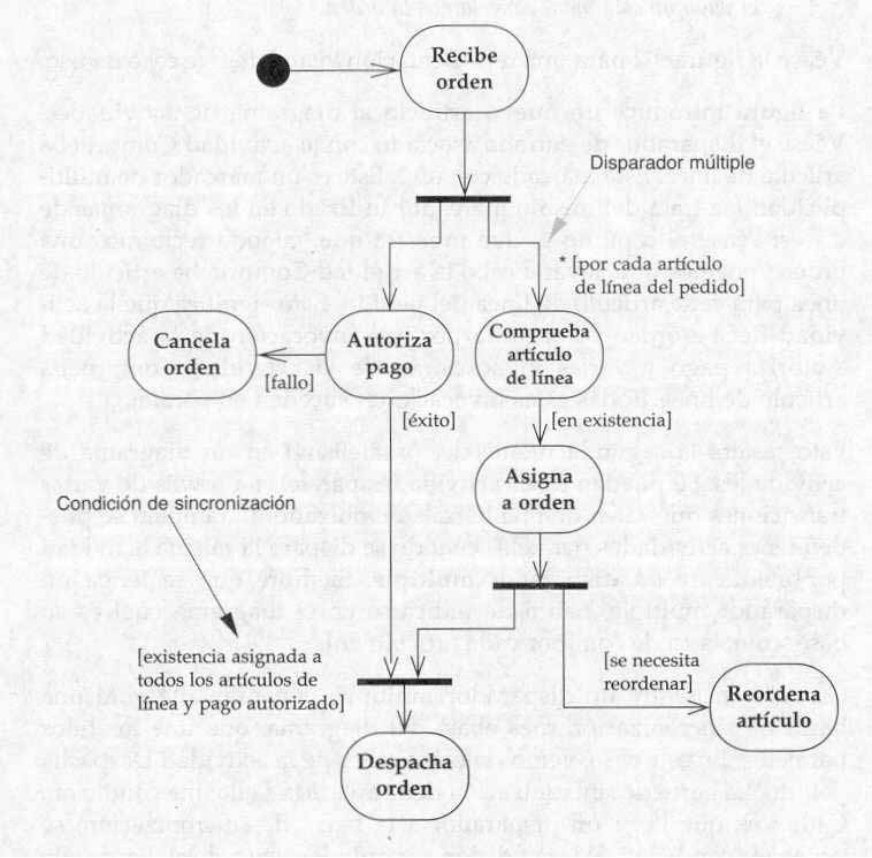
ya que pueden plantear gráficamente cuáles son los hilos y cuándo necesitan sincronizarse.

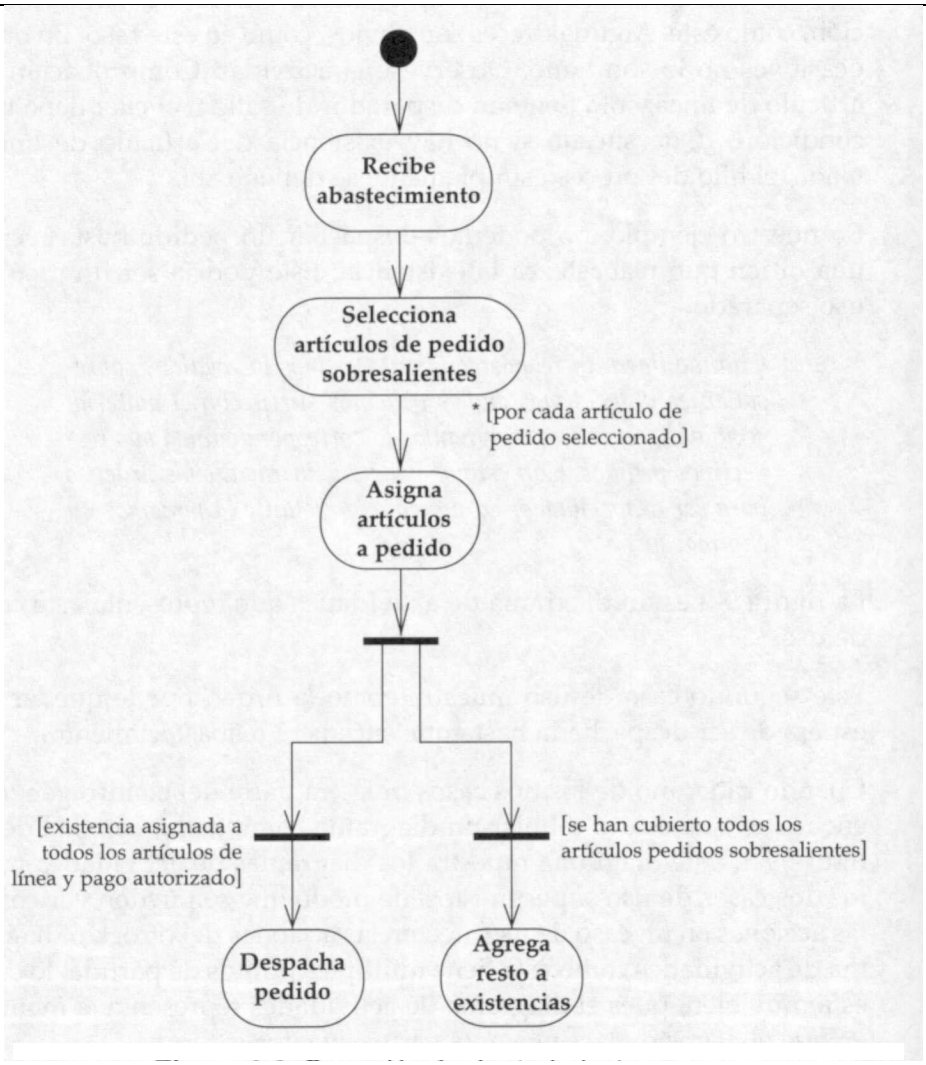
En los procesos concurrentes es necesario especificar las sincronizaciones; pare lograr esto, el diagrama de actividades proporciona la barra de sincronización. Una barra de sincronización indica que el *disparo* de salida ocurre sólo cuando se han llevado a cabo todos los *disparos* de entrada.

La barra de sincronización puede ir opcionalmente etiquetada con una condición. Si la barra no tiene una condición significa que tiene una condición predeterminada: que todos los disparadores de entrada ya han ocurrido antes de emitir un disparador de salida.

En el diagrama anterior también se puede apreciar la decisión compuesta. Esta nos permite describir decisiones anidadas.

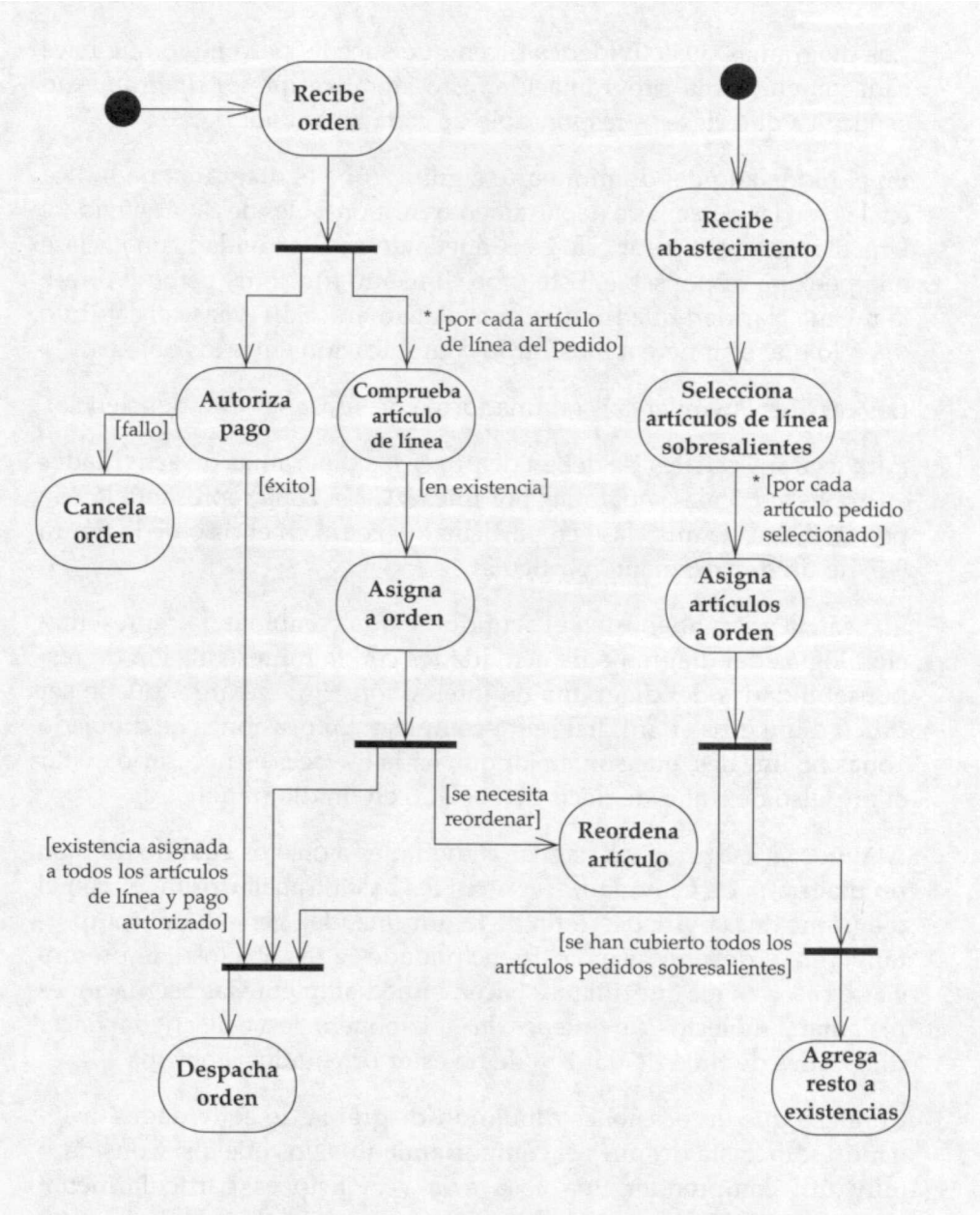
Los diagramas de actividades normalmente se usan para describir métodos complejos o casos de uso potencialmente concurrentes.


Recibe orden
Disparador múltiple
* [por cada artículo de línea del pedido]
Cancela orden
Autoriza pago
Comprueba artículo de línea
[fallo]
[éxito]
[en existencia]
Asigna a orden
Condición de sincronización
[existencia asignada a todos los artículos de línea y pago autorizado]
[se necesita reordenar]
Reordena artículo
Despacha orden



El diagrama anterior de reabastecimiento de pedidos puede ser combinado con el diagrama de emisión de pedidos. Este nuevo diagrama representaría las múltiples actividades de los procesos de negocio dependiendo de varios eventos externos.

Un diagrama de actividades combinado nos ofrece una panorámica más amplia al representar varios casos de uso a la vez.

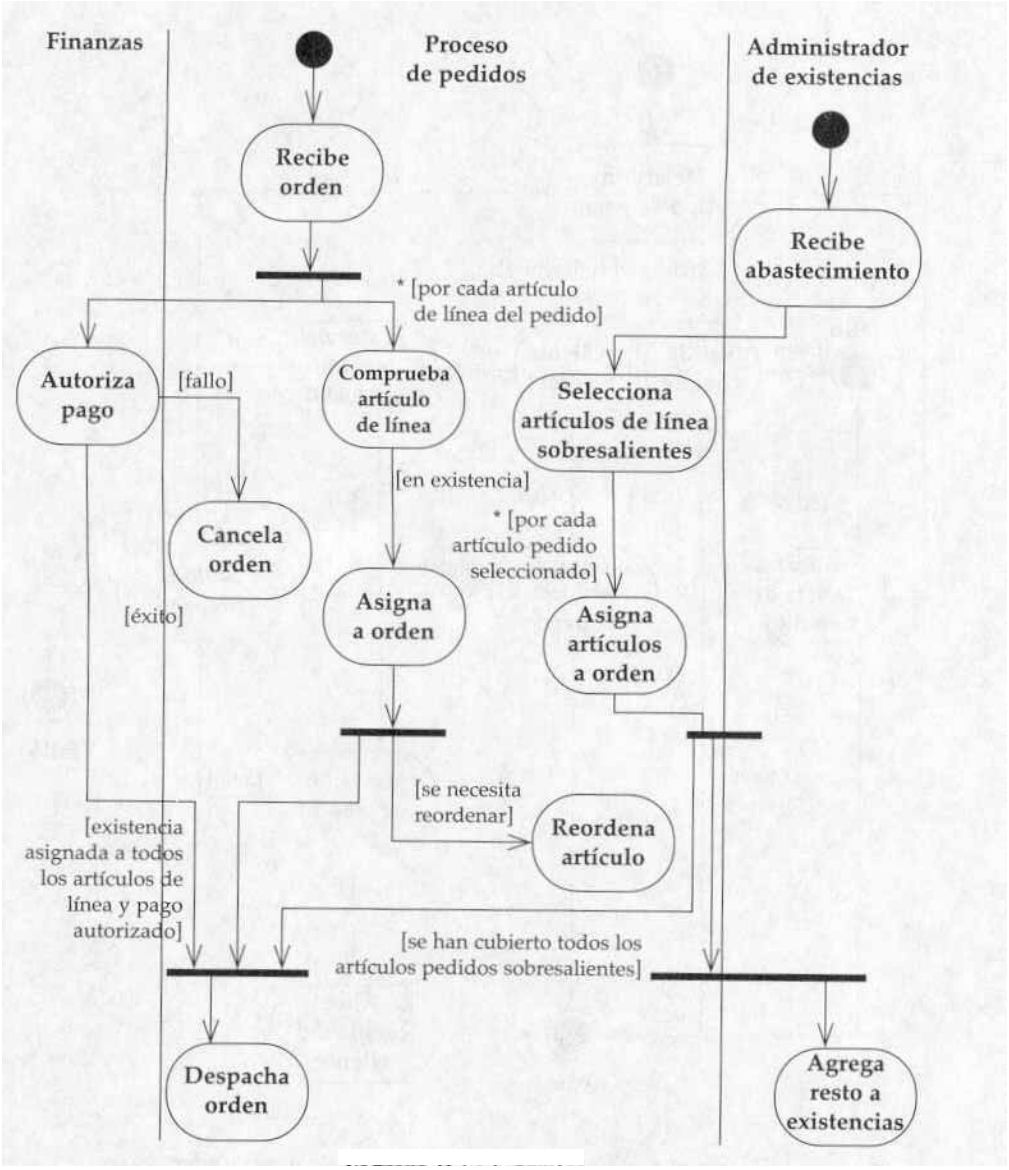


Carriles.

Los diagramas de actividades indican *que sucede*, pero no *quien lo hace*. Es decir, no se indica qué clase o concepto es responsable de cada actividad.

Los carriles son una forma de disminuir esta deficiencia de los diagramas de actividades.

La idea de los carriles es acomodar los diagramas de actividades en zonas verticales separadas por líneas punteadas. Cada zona representa la responsabilidad de una clase en particular.



El objetivo de los carriles es combinar la representación lógica de los diagramas de actividades con la representación de las responsabilidades de los diagrama de interacción.

Conclusiones.

- Los diagramas de actividad manejan y promueven el comportamiento paralelo. Y son de gran ayuda para el desarrollo de aplicaciones multihilos.
- Ayudan a analizar los casos de uso.
- De igual manera ayudan a la comprensión del flujo de trabajo en varios casos de uso.
- Fallan al no dejar muy claros los vínculos entre las actividades y los objetos.
- No están orientados a objetos.

Fin de la segunda iteración del análisis.

Actualizar toda la documentación: diagramas, tarjetas CRC, descripción de atributos, contratos de operaciones etc.

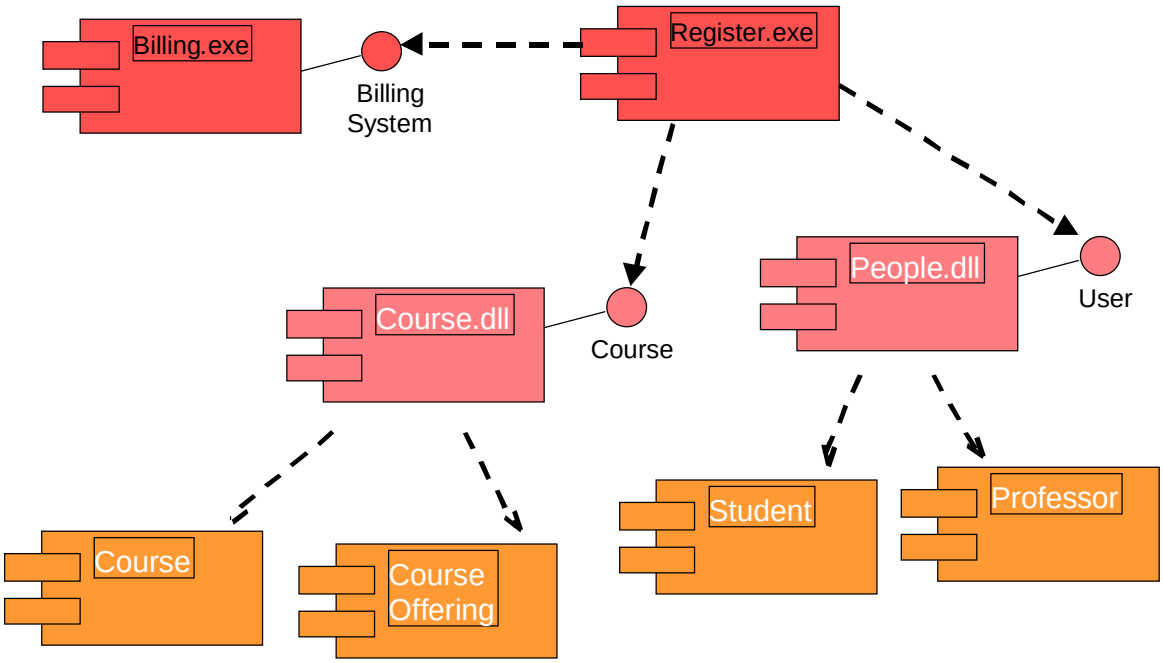
Inicio de segunda iteración del diseño.

Actualizar casos de uso reales y diagramas de colaboración y el diagrama de clases del diseño con las nuevas clases, generalización, agregación, etc.

Diagramas de componentes y de emplazamiento.

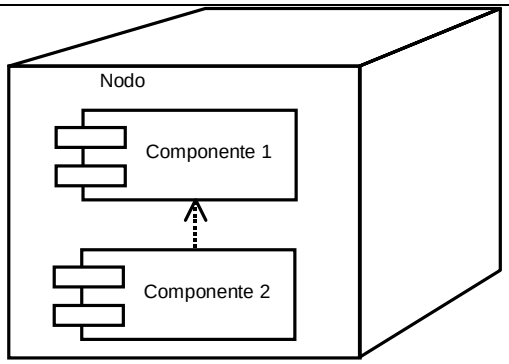
El diagrama de componentes representa las esturctura física de la implementación. Forma parte de la arquitectura de la especificación, y ayuda a proponer:

- La organización del código fuente,
- La construcción de versiones ejecutables finales.
- La especificación de la base de datos.



Por otro lado, un diagrama de emplazamiento trata de capturar la topología de un sistema de hardware. Trata de ubicar la distribución de los componentes y ayuda a identificar los cuellos de botella.

De este modo, la combinación de los diagramas de componentes y de emplazamiento muestra las relaciones físicas entre los componentes de software y de hardware del sistema.



Cada nodo en un diagrama de emplazamiento representa algún tipo de unidad de cómputo.

Las dependencias entre los componentes son las mismas que las dependencias de paquetes.

Los componentes en un diagrama representan módulos físicos de código. Por lo regular un componente coincide con un paquete, por lo que, el diagrama de emplazamiento muestra dónde se ejecuta cada paquete del sistema.

Un componente puede tener más de una interfaz, y habrá que especificar que componente se comunica con que interfaz.

Este tipo de diagramas no es muy utilizado de manera formal, pero conforme crezca la necesidad de modelar sistemas distribuidos podríamos ver un mayor número de diagramas de emplazamiento y la notación ira evolucionando conforme la importancia de estos diagramas vaya creciendo.