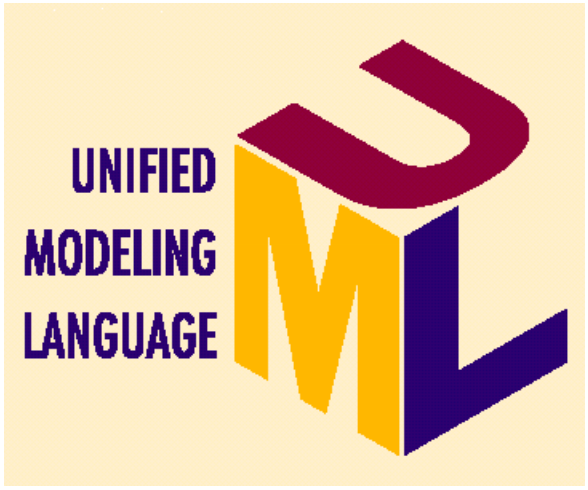
PROGRAMACIÓN ORIENTADA A OBJETOS II

Notas de clase.



Elaborado por:
Carlos Alberto Fernández y Fernández
Instituto de Electrónica y Computación
Universidad Tecnológica de la Mixteca

2001.

Contenido.

Preámbulo. 6
Crisis del software. 6
Complejidad del software. 13
La complejidad del dominio del problema. 13
Dificultad en la administración del proceso. 14
Naturaleza abstracta. 14
Comportamiento Discreto. 15
El límite de Miller. 15
¿Cómo se enfrenta la complejidad? 16
Descomposición algorítmica. 16
Descomposición orientada a objetos. 17
Comparación entre descomposición algorítmica y orientada a objetos. 17
Conceptos básicos de objetos. 18
Comparación con el análisis y diseño estructurado. 22
Introducción. 23
Modelado Visual. 27
UML. 29
UML combina lo mejor de: 30
UML puede ser usado para: 31
UML conviene por: 31
Asociados de UML 32
Contribuciones. 32
Proceso Unificado Rational. 33
Evolución del Proceso Unificado. 34
Fases del ciclo de vida. 34
En conclusión el Proceso Unificado: 36
Mejores prácticas en el desarrollo de Software. 37
Desarrollo Iterativo. 37
Beneficios del desarrollo iterativo. 38
Administración de Requerimientos. 39
Arquitectura basada en componentes. 40
Modelado Visual. 42
Control del calidad. 43
Control de cambios. 44
Elementos del proceso unificado. 45
Trabajadores. 45
Actividades 46
Artefactos. 48
Flujo: Administración del proyecto 49
Flujo: Modelado de Negocios. 50
Cuando es necesario el modelado de negocios? 51
Trabajadores. 51
Flujo: Requerimientos. 52
Trabajadores involucrados. 54
Modelo de Casos de Uso. 55
Estructura de los casos de uso. 56

Casos de uso de alto nivel. 57
Casos de uso expandidos. 58
Diagramas de casos de uso. 61
Relaciones entre Casos de uso. 62
Tipos de casos de uso. 64
Casos de uso esenciales. 64
Casos de uso reales. 65
Proceso para la etapa de casos de uso. 66
Conclusiones del modelo de casos de uso. 67
Ejemplo de un diagrama de casos de uso 68
Flujo: Análisis y Diseño. 70
Diferencia entre análisis y diseño. 70
Trabajadores y artefactos involucrados. 71
Definir una Arquitectura Candidata 72
Actividad: Análisis de la Arquitectura 73
Convenciones de Modelado. 73
Mecanismos de análisis 74
Identificación de abstracciones clave. 75
Arquitectura de capas inicial. 76
Artefacto: Diagramas de Paquetes, una introducción. 78
Actividad: Análisis de Casos de Uso. 80
Complementar la descripción del flujo de eventos de los casos de uso. 81
Artefacto: Diagrama de clases, una introducción. 82
Conceptos de UML. 83
Identificación inicial de clases. 84
Lista de categorías de conceptos. 85
Frases nominales. 86
Selección de clases. 88
Generalización. 89
Notación de UML 89
Conceptos e identificación de superclases y subclases. 89
Clases abstractas. 92
Buscar clases a partir de la descripción del comportamiento del caso de uso. 93
Distribuir el comportamiento. 97
Diagramas de interacción. 99
Diagramas de Secuencia. 99
Procesos concurrentes. 102
Diagramas de colaboración. 106
Elaboración de diagramas de colaboración: 113
Diagramas de Colaboración Vs Diagramas de Secuencia 114
Describir Responsabilidades. 115
Describir Atributos y Asociaciones. 117
Visibilidad. 124
Visibilidad de atributos. 124
Visibilidad de parámetros. 125
Visibilidad declarada localmente. 125
Visibilidad global. 126

Notación opcional en UML para indicar la visibilidad. 126
Asociación y Agregación..... 127
Asociación. 127
Multiplicidad. 128
Identificación de asociaciones. 129
Clase de asociación..... 132
Asociaciones calificadas. 133
Asociaciones reflexivas. 134
Relaciones de agregación y composición. 134
Agregación..... 135
Composición. 135
Identificación de la agregación. 136
Descripción de Mecanismos de Análisis. 137
Unificar Clases de Análisis 138
Evaluar Resultados 138
Diseño de la Arquitectura. 139
Mecanismos de Diseño e Implementación. 139
Más sobre Diagramas de Paquetes. 140
Diseño de Clases y Subsistemas..... 144
Identificando Clases de Diseño. 145
Identificando subsistemas..... 145
Interfaces. 147
Identificar Oportunidades de Reuso 148
Diseño de Casos de Uso 149
Describir interacciones entre objetos de diseño. 149
Describir comportamiento de persistencia relacionado..... 150
Refinar la descripción del flujo de eventos 151
Unificar clases, paquetes y subsistemas. 152
Diseño de los Subsistemas..... 153
Distribuir el comportamiento del subsistema a los elementos del subsistema. 153
Modelar los elementos del subsistema. 155
Describir las dependencias del subsistema. 155
Diagramas de Estado. 156
Elementos del diagrama de estados. 156
Superestado..... 160
Diagramas de estados concurrentes. 162
Clases orientadas al diseño. 164
Notación en UML para miembros de clase. 165
Diseño de Clases..... 166
Creación inicial de clases de diseño 166
Identificar clases persistentes. 166
Definir operaciones..... 167
Definir visibilidad de clases. 167
Definir métodos. 167
Definir estados. 168
Definir atributos..... 168
Definir dependencias. 168

Definir asociaciones. 169
Definir generalizaciones 169
Resolver colisiones de Casos de Uso 170
Manejo de requerimientos no funcionales en general. 170
Diagramas de actividades. 171
Carriles. 177
Diagramas de componentes y de despliegue. 179
Bibliografía. 181
Complementaria 181
Artículos 182

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 difícilmente 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 cambio un poco de manera obligada debido al problema del año 2000. Las empresas se vieron en la necesidad de actualizar el

software o morir. Sin embargo la mayoría de las empresas únicamente "parcharon" 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 debieron 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 creció 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"

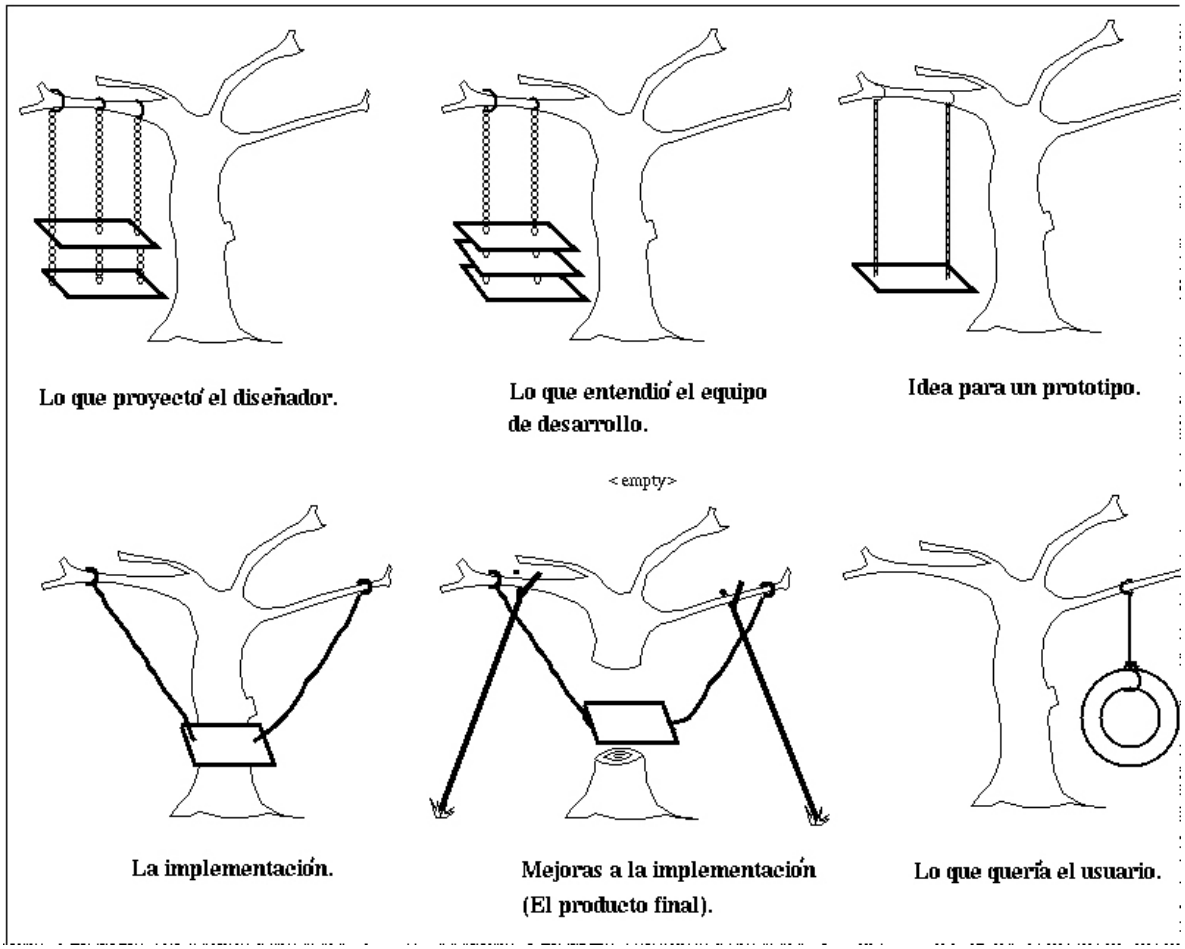
Realidad: Alguien dijo: "cuanto más rápido te pongas a codificar, más tiempo te tardarás en acabar el sistema". Las estadísticas nos dicen que del 50 al 70% del esfuerzo se emplea después de que se entregó el sistema.

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: considérese 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 dispositivo mecánico, 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.
¿Cómo 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 cada 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: estático, dinámico y funcional.
- Las metodologías de objetos están dominados por el modelo de estático, en el modelo estructurado domina el funcional y el estático 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*) en 1990 y las tarjetas de clase-responsabilidad (*Class-Responsibility-Collaboration*) (CRC) en 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*) conocida como OMT. 1991 y 1996.
- Ivar **Jacobson** trabajando para Ericcson en conmutadores telefónicos, introdujo el concepto de Casos de Uso (*use cases*). 1994 y 1995.

Las segundas versiones de los métodos de Booch y Rumbaugh (Booch 93 y OMT-2), se acercan tanto que es más en lo que se parecen que en lo que difieren:

- Booch 93 toma de OMT las asociaciones y los diagramas de estados (también conocidos como diagramas de Harel)
- OMT-2 toma de Booch los flujos de mensajes, los modelos jerárquicos y los subsistemas, los componentes modelos y esencialmente retira del modelo funcional los diagramas de flujos de datos.

Sin embargo, Booch 93 pone mayor énfasis en la construcción, mientras que OMT-2 se enfoca mayormente en el análisis y la abstracción.

Así que, aunque 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 técnica a la que le llaman *Unified Modeling Language* (**UML,** Lenguaje Unificado de Modelado).

Grady Booch

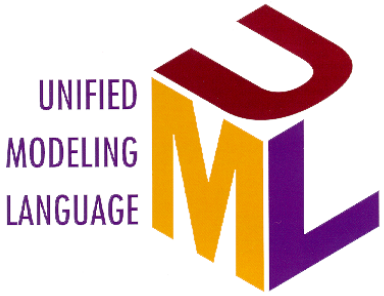
Jim Rumbaugh

Ivar Jacobson.

En conclusión, **UML** unifica esencialmente los métodos de Booch, Rumbaugh (OMT) y Jacobson. Se fijaron cuatro objetivos:

1. Representar sistemas completos con conceptos de objetos.
2. Establecer una relación explícita entre los conceptos y los "artefactos" ejecutables.
3. Tener en cuenta los factores de escala inherentes a los sistemas complejos y críticos.
4. Crear un lenguaje de modelado utilizable tanto por humanos como por máquinas.

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.

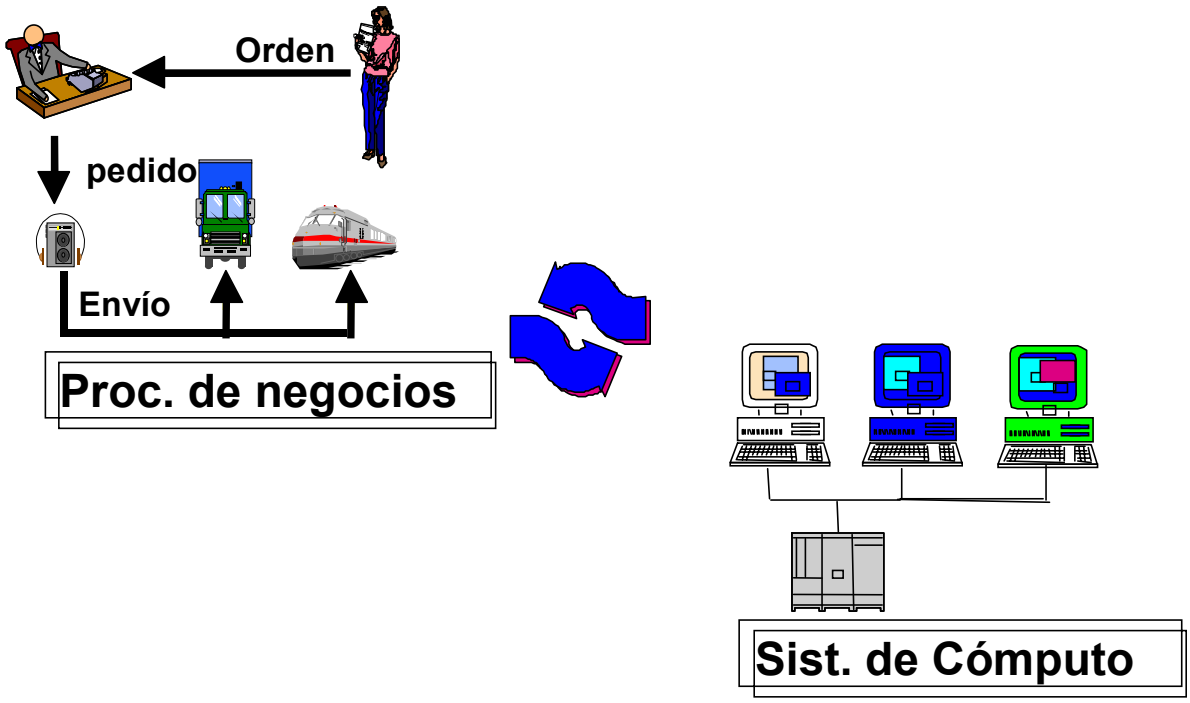
Booch, Jacobson y Rumbaugh trabajaron también en la creación de un proceso unificado llamado anteriormente **Objectory**. Ahora se conoce como *Rational Unified Process* (Proceso Unificado Rational).

Modelado Visual.

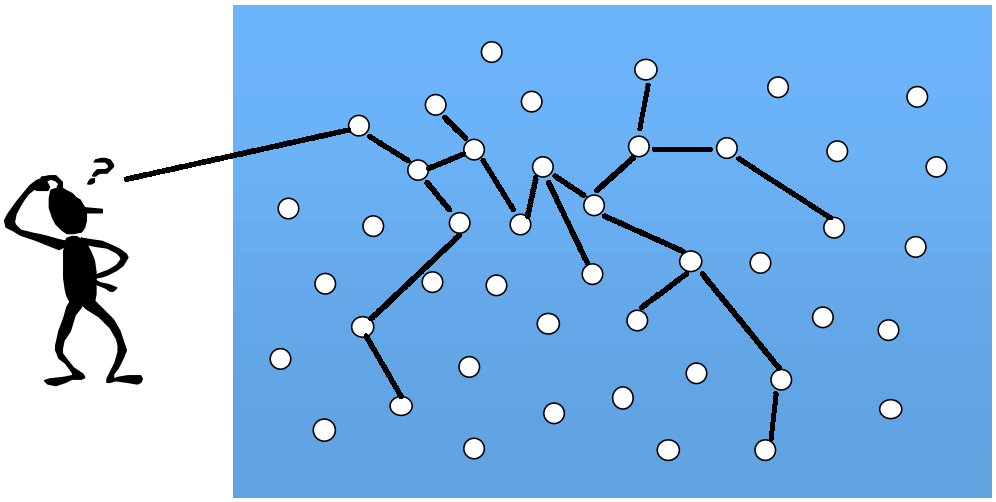
El Modelado Visual es el modelado de una aplicación usando notaciones gráficas.

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

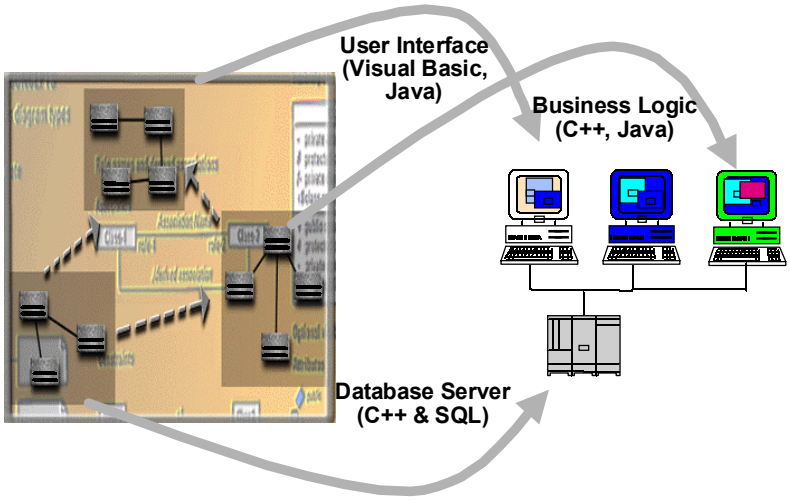
Rumbaugh



El Modelado Visual captura los Procesos de Negocios.



- Es usado 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.

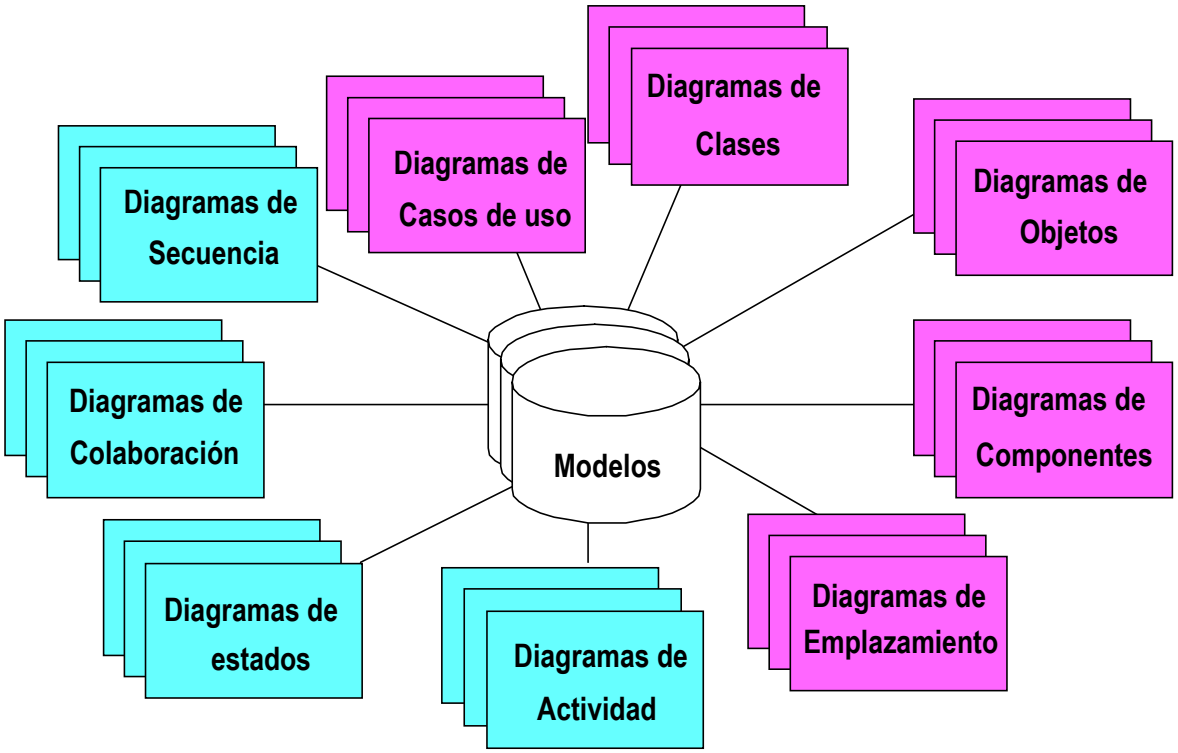
Pregunta obligada: ¿qué es UML?

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

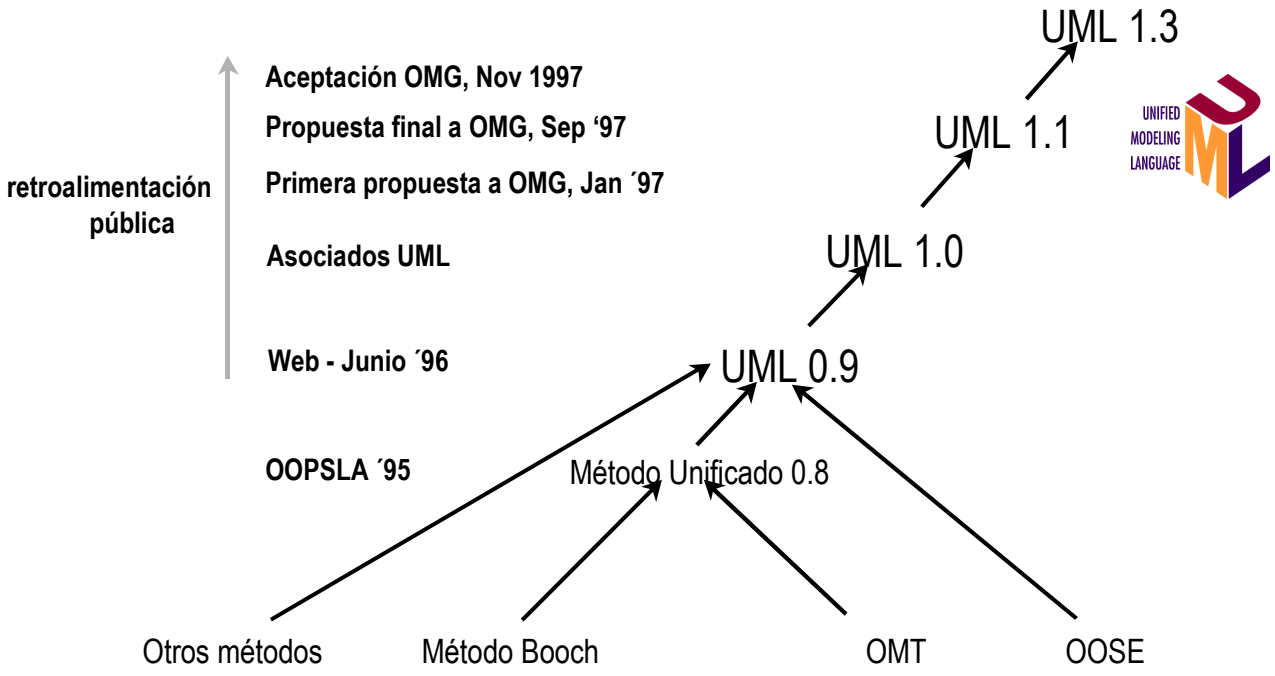
"Un modelo es una descripción completa de un sistema desde una perspectiva particular".

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.

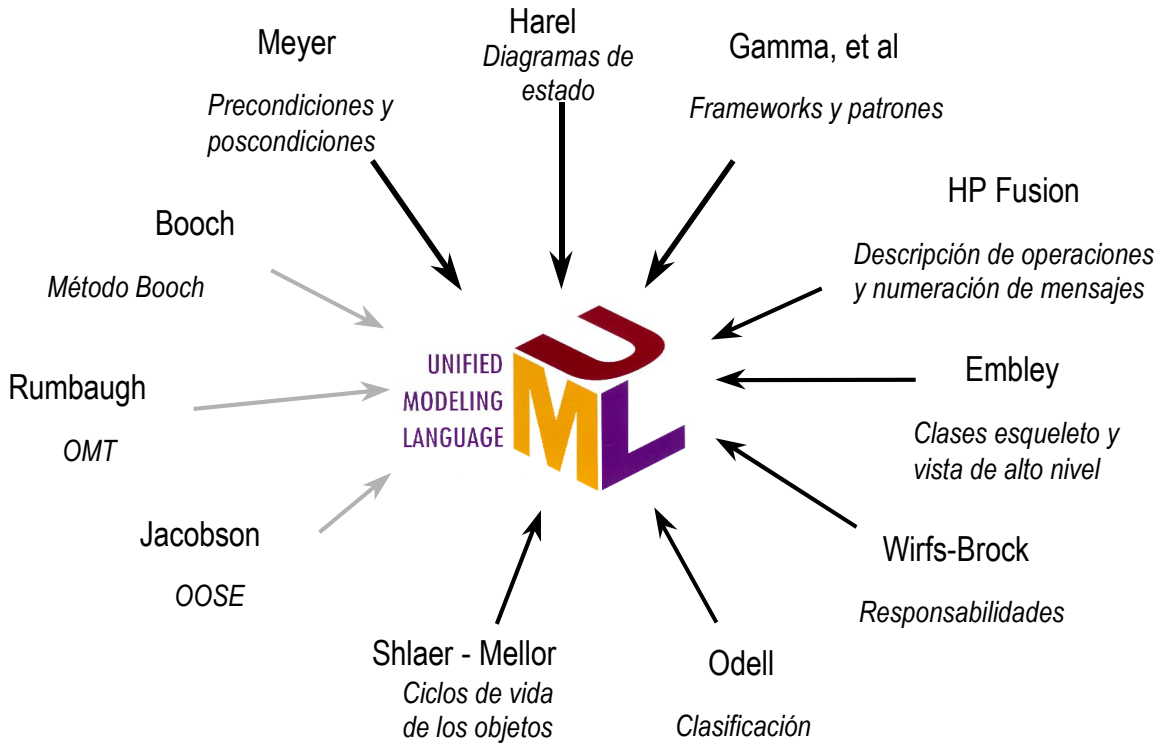
UML conviene por:

- 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.

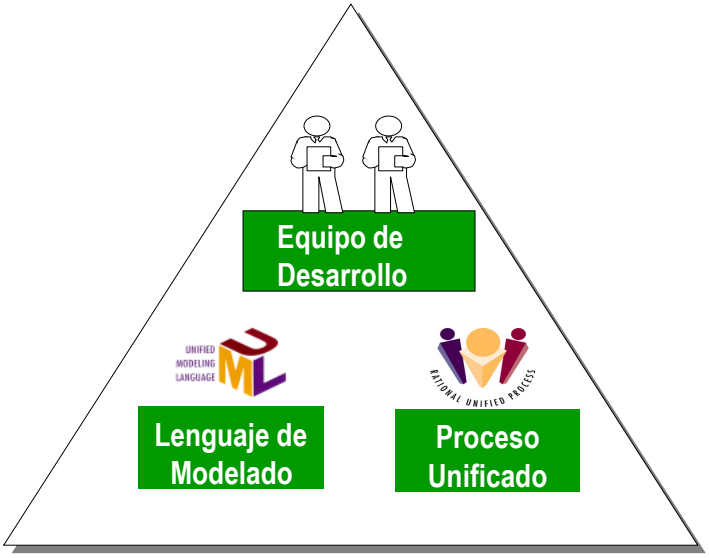


Proceso Unificado Rational.

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.

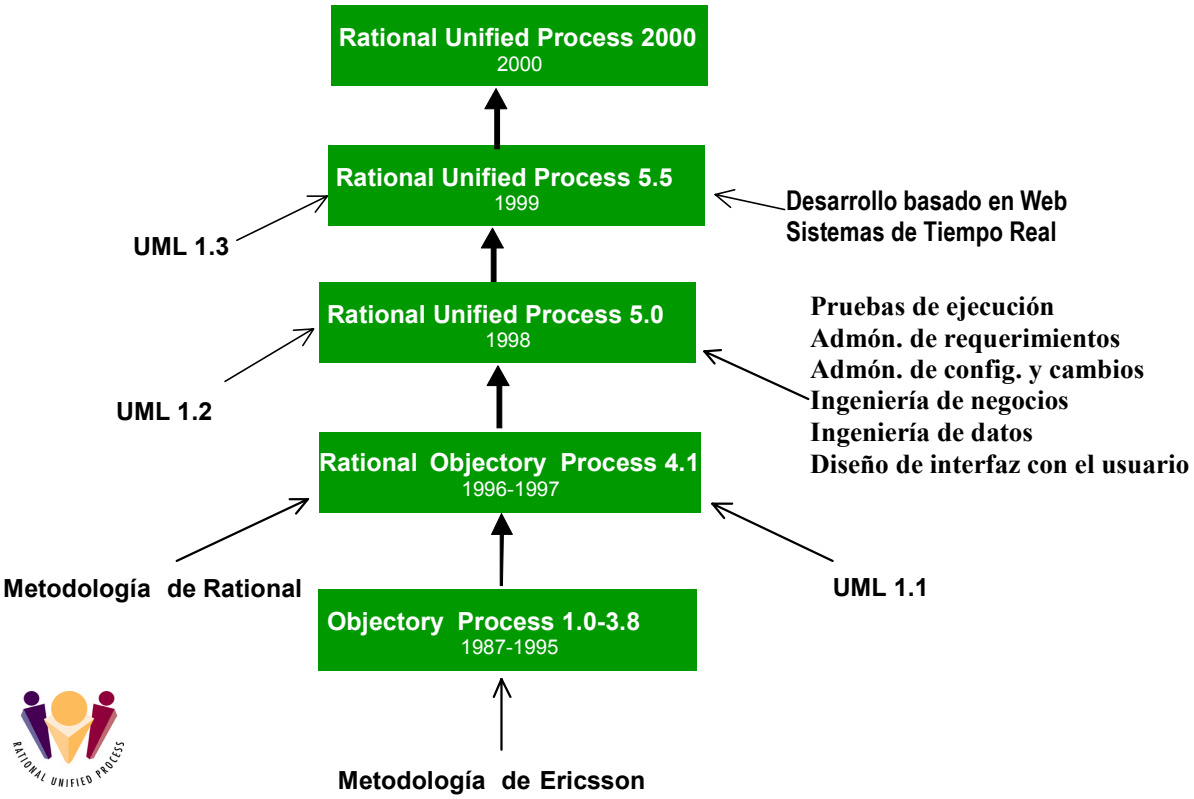
En principio, sabemos que el lenguaje de modelado visual no es suficiente para realizar el modelo de un sistema. Son necesarios tres elementos:



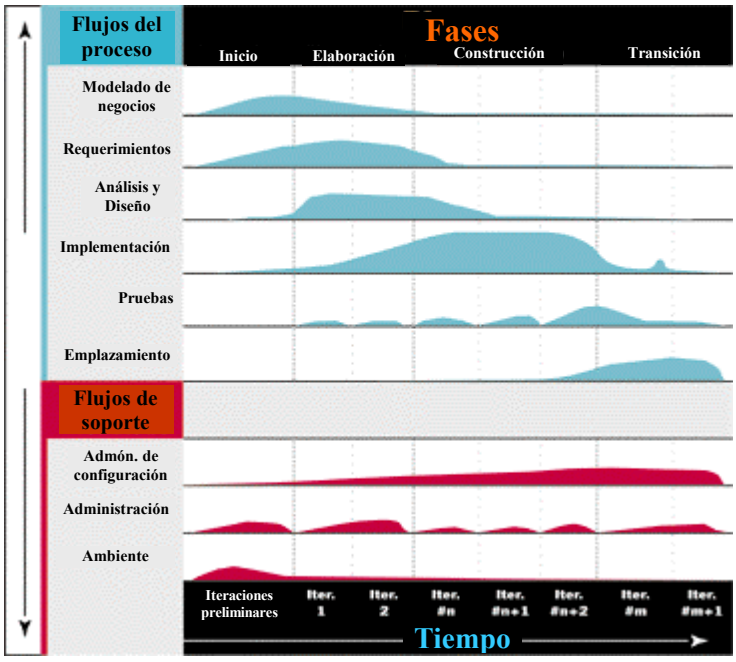
El proceso de desarrollo de software tiene cuatro roles:

1. Proporcionar una guía del orden de las actividades de los equipos.
2. Especificar cuales artefectos deben ser desarrolados y cuando deber ser desarrollados.
3. Dirigir las tareas de desarroladores individuales y equipos como una sola.
4. Ofrecer criterios para monitorear y medir los productos y actividades del proyecto.

Evolución del Proceso Unificado.



Fases del ciclo de vida.



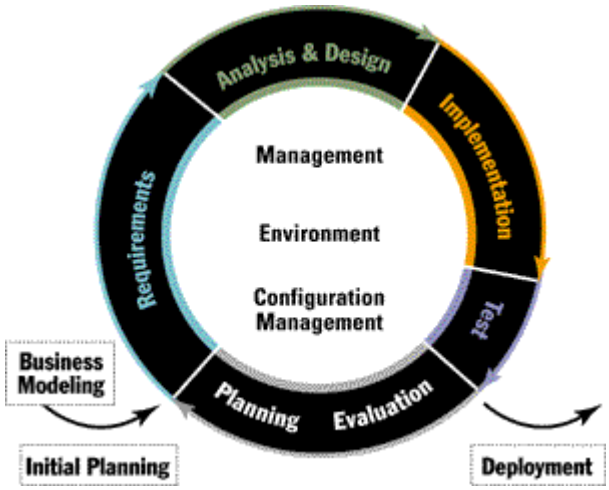
El RUP organiza a los proyectos en términos de flujos de trabajo y fases, las cuales consisten de una o más iteraciones. En cada iteración, el énfasis en cada flujo de trabajo variará a lo largo del ciclo de vida.

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.



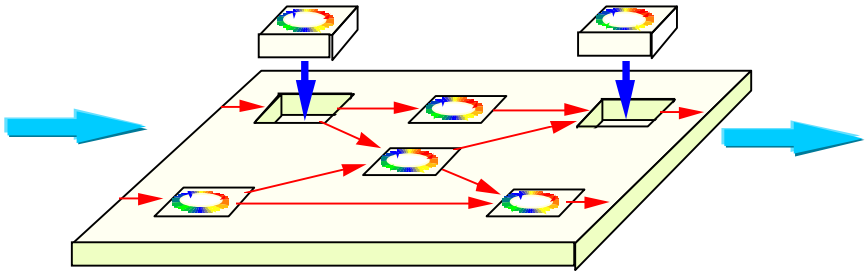
Cada iteración del ciclo del proyecto iniciará con un plan, el cual será llevado a cabo y concluido con una evaluación para ver que tanto se han cumplido los objetivos.

Contempla 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.

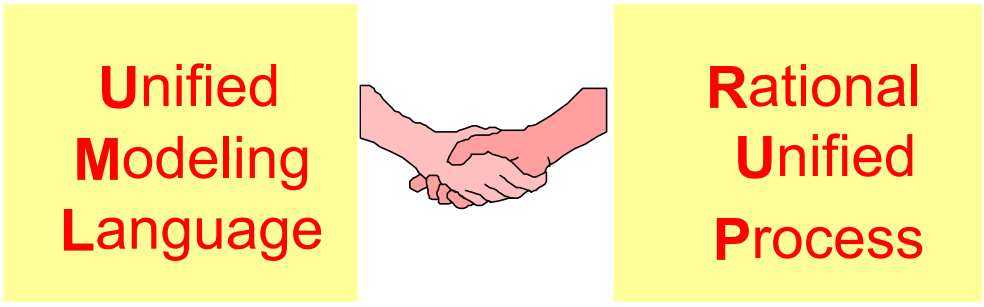
En conclusión el Proceso Unificado:

- 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.



Mejores prácticas en el desarrollo de Software.

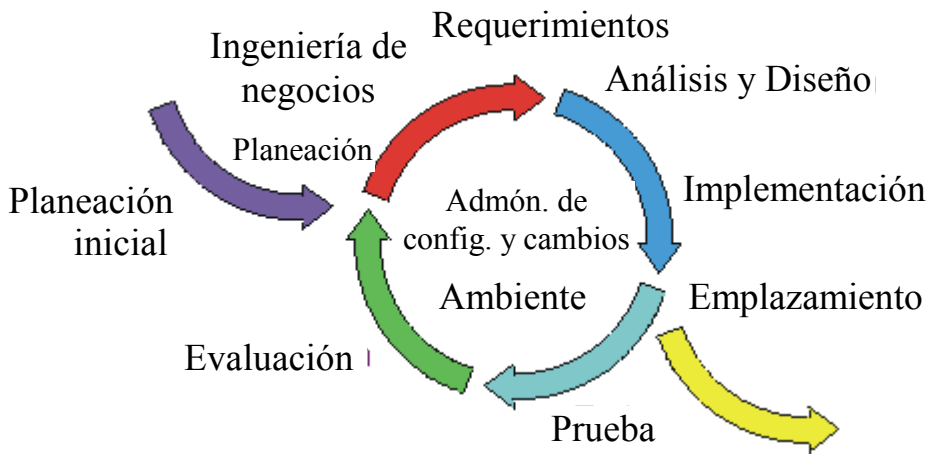
Una forma de tratar de eliminar los errores en la construcción de software es hacer uso de las "mejores prácticas", las cuales son soluciones no cuantificables, pero que han probado tener mayores posibilidades de éxito en la industria. Estás mejores prácticas son:

1. Desarrollo iterativo.
2. Administración de requerimientos.
3. Arquitectura basada en componentes.
4. Modelado Visual.
5. Verificación de la calidad.
6. Control de cambios.

Desarrollo Iterativo.

El desarrollo clásico del software es conocido como desarrollo en cascada, donde de manera secuencial se sigue de la etapa de requerimientos, a análisis, diseño hasta llegar con suerte a la etapa de pruebas.

El desarrollo iterativo propone una planeación inicial y posteriormente entrar a un ciclo en las etapas de desarrollo. Donde para cada iteración resulte une versión ejecutable del sistema.



Para mitigar riesgos, el desarrollo incremental en una forma iterativa.

Beneficios del desarrollo iterativo.

Las principales razones por las que el desarollo iterativo es superior al linea son:

- Tolerable a cambios en los requerimientos.
- Los elementos sonintegrados progresivamente.
- Los riesgos son mitigados en etapas tempranas.
- Permite a la organización aprender e improvisar.
- Facilita el reuso, porque es fácil identificar partes comunes diseñadas o implementadas.
- Resulta un producto más robusto, ya que los errores se van corrigiendo en cada iteración.
- Tolerable a cambios tácticos en el producto.
- El proceso puede ser improvisado y refinado en el desarrollo.

Administración de Requerimientos.

El manejo de los requerimientos de software debe de ser dinámico: debe esperarse que estos cambien durante la vida de un proyecto de software.

La administración de requerimientos es una aproximación sistemática para la búsqueda, documentación, organización y seguimiento de los cambios en los requerimientos de un sistema.

Un requerimiento es una condición o capacidad con el que un sistema debe conformarse.

La administración de requerimientos ofrece soluciones a un buen número de los problemas de raíz de los que sufre el desarrollo de software.

- Las comunicaciones están basadas en requerimientos definidos.
- Los requerimientos pueden ser priorizados, filtrados y trazados.
- Incosistencias son detectadas en etapas más tempranas del desarrollo del proyecto.
- Es posible realizar una evaluación objetiva de funcionalidad y rendimiento.

Arquitectura basada en componentes.

Las actividades de diseño están centradas en la noción de arquitectura. Uno de los principales objetivos de las primeras iteraciones es obtener una arquitectura de software válida, donde en ciclos iniciales de desarrollo formen un prototipo ejecutable de la arquitectura que gradualmente se vaya conviertiendo en el sistema final en las últimas iteraciones.

La arquitectura es importante debido a:

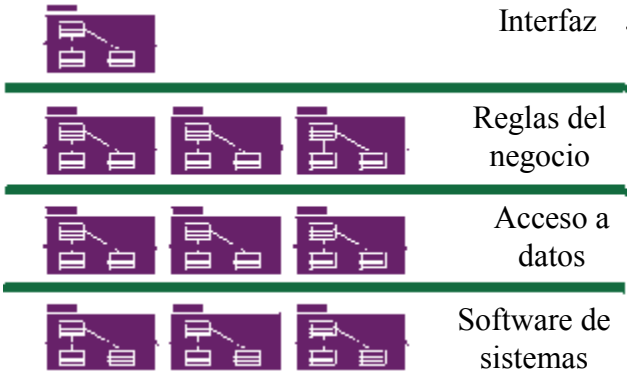
- Permite que el equipo gane y retenga control intelectual sobre el proyecto, para manejar su complejidad, y mantener la integridad del sistema.
- Es la base de un efectivo reuso en gran escala.
- Provee la base para el administrador del proyecto.

El desarrollo basado en componentes (CBD[2]) es una importante aproximación a la arquitectura de software, porque este permite:

- Una arquitectura modular, identificando, diseñando desarrollando y probando componentes individuales, e integrándolos gradualmente al sistema que se está desarrollando.

[2] Component-Based Development

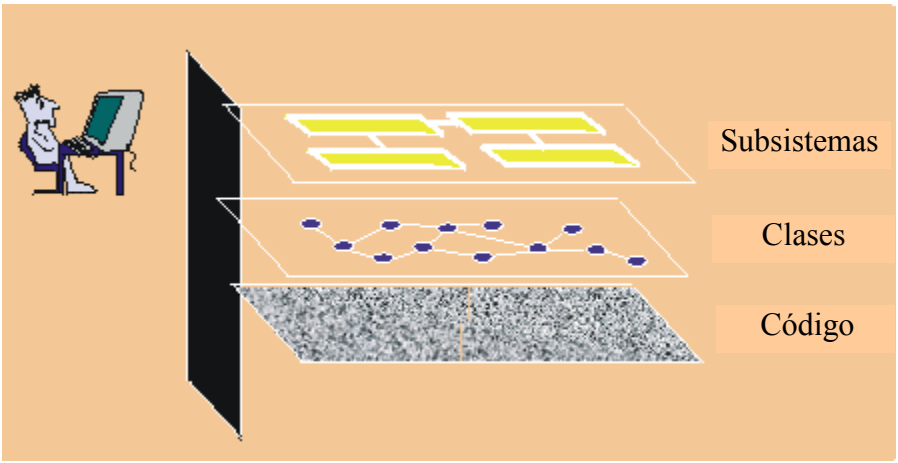
- Algunos de estos componentes pueden ser desarrollados para ser reusables, especialmente los componentes que proporcionan soluciones comunes a un amplio rango de problemas comunes. Estos pueden ayudar a la empresa a formar una biblioteca de componentes, que poco a poco vayan incrementando la productividad y la calidad de desarrollo.
- El aprovechamiento de infraestructuras comerciales como el Modelo Objeto Componente de Microsoft (COM), CORBa de OMG, o JavaBeans de Sun Microsystems.



Modelado Visual.

Un modelo es una simplificación de la realidad que describe completamente un sistema desde una perspectiva particular.

El modelado es importante porque ayuda al equipo a visualizar, especificar, construir y documentar la estructura y el comportamiento de la arquitectura del sistema.

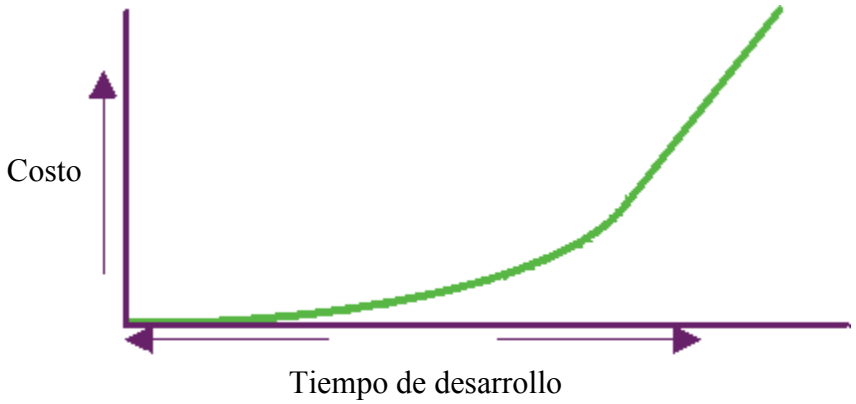


Un Modelo, correctamente diseñado usando tecnología de objetos:

- Es fácil de entender. Claramente corresponde a la realidad.
- Fácil de modificar. Cambios en un aspecto en particular concierne únicamente al objeto que representa ese aspecto.

Control del calidad.

Los problemas del software son de 100 a 1000 veces más difíciles de encontrar y reparar (y por tanto más caros) después del desarrollo. La verificación y administración de la calidad durante el ciclo de vida del proyecto es esencial para lograr mantener los objetivos y el tiempo estimado de desarrollo.



¿Qué es la calidad?

Para los propósitos del Proceso Unificado, la calidad es definida como:

"Es la característica identificada por lo siguiente:

satisface o excede lo acordado sobre el conjunto de requerimientos, y

medidas utilizando las métricas y criterios acordados, y

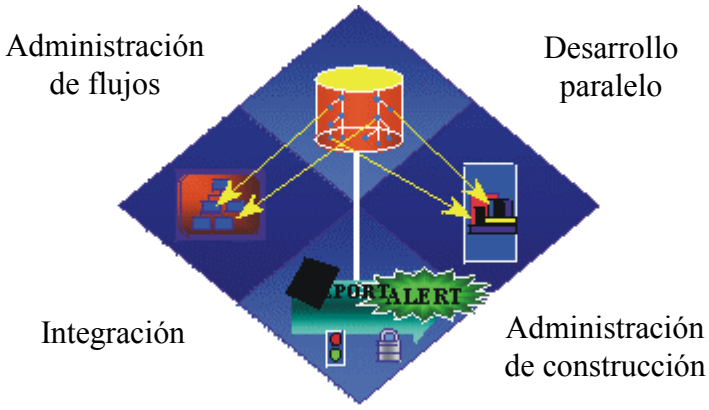
es producido utilizando un proceso acordado de antemano."

Control de cambios.

El desarrollo intensivo de sistemas de software es llevado a cabo por múltiples desarrolladores organizados en equipo, posiblemente en diferentes sitios, trabajando en múltiples iteraciones, versiones, productos y plataformas. Si no existe una disciplina de control, el proceso de desarrollo rapidamente degenera en caos.

La coordinación de las actividades y artefactos de los desarrolladores y equipos, involucra establecer flujos repetibles para administración de cambios al software. Esta coordinación permite una mejor identificación de los recursos básicos en las prioridades y riesgos del proyecto.

La coordinación de iteraciones y versiones involucra establecer y realizar una prueba "baseline" al finalizar cada iteración.



El control de cambios es más que revisar entradas y salidas en los archivos. Este incluye administrar los flujos, el desarrollo paralelo, la integración y la construcción del software.

Elementos del proceso unificado.

Ya se ha dicho que un proceso describe quien hace que, cuando lo hace y como lograrlo. En el RUP, esto es representado usando cuatro elementos primarios:

- Trabajadores: el *quien*.
- Actividades: el *como*.
- Artefactos: el *que*.
- Flujos de trabajo: el *cuando*.

Trabajadores.

Son el concepto central en el proceso. Un trabajador define el comportamiento y las responsabilidades de una persona individual o de un grupo de individuos trabajando todos como un equipo.

El comportamiento es expresado en términos de actividades que los trabajadores realizan, donde cada trabajador es asignado a un conjunto de actividades.

El concepto de trabajador no se refiere a un individuo o persona, sino que hace referencia a el comportamiento y responsabilidades que un o unos individuos tienen en el proyecto de desarrollo. Un trabajador entonces es un rol que una persona puede jugar en un momento determinado.

Algunos trabajadores típicos:

- Analista de sistemas.
- Diseñador.
- Revisor de requerimientos.
- Diseñador de interfaz con el usuario.
- Diseñador de base de datos.
- Integrador.
- Diseñador de pruebas.
- Probador.
- Administrador del proyecto.

Actividades

Los trabajadores tienen actividades, las cuales definen el trabajo que ellos realizan. Una actividad es una unidad de trabajo que un individuo en su rol puede ejecutar, produciendo un resultado significativo en el contexto del proyecto.

Una actividad tiene un propósito muy claro, usualmente expresada en términos de creación o actualización de artefactos, como un modelo, una clase, un plan, etc. La granularidad de una actividad por lo general es de algunas horas a pocos días. El progreso en las actividades me puede dar elementos de planeación y progreso del proyecto.

Una misma actividad puede repetirse de una iteración a otra, refinando y expandiendo al sistema.

Ejemplos de actividades:

- Planear una iteración: ejecutada por el *trabajador* Administrador del proyecto.
- Buscar casos de uso y actores: ejecutada por el *trabajador* Analista de sistemas.
- Revisión del diseño: ejecutada por el *trabajador* Revisor del diseño.

Las actividades son divididas en pasos. Estos se agrupan en tres grandes categorías:

- **Pensando** los pasos. El trabajador entiende la naturaleza de la tarea, reúne y examina los artefactos de entrada, y formula los resultados.
- **Ejecutando** los pasos. El trabajador crea o actualiza algunos artefactos.
- **Revisando** los pasos. El trabajador revisa los resultados bajo ciertos criterios.

Artefactos.

Las actividades tienen artefactos de entrada y salida. Un artefacto es una pieza de información que es producida, modificada o usada por un proceso.

Un artefacto es un producto tangible del proyecto: las cosas que el proyecto produce o usa mientras se trabaja hasta el final del producto. Son usados por los trabajadores como entradas para ejecutar una actividad y son el resultado o salida de alguna actividad.

Ejemplos de artefactos:

- Un **modelo**, como el modelo de casos de uso o el modelo de diseño.
- Un **elemento del modelo**, como una clase, un caso de uso o un subsistema.
- Un documento, como el documento de visión, o el documento de arquitectura.
- Código fuente.
- Ejecutables.

Se hace énfasis en el artefacto y no en el documento. Ya que el artefacto puede estar contenido dentro de alguna herramienta que facilite su manipulación. Un documento es simplemente la forma de presentar el artefacto.

Flujo: Administración del proyecto

"Planear: preocuparse por encontrar el mejor método para lograr un resultado accidental".

Ambrose Bierce

El flujo de trabajo de administración de proyectos de software es el arte de balancear objetivos competitivos, manejo de riesgos y superar las restricciones para generar el producto que satisfaga las necesidades de los clientes y de los usuarios finales.

Los objetivos esenciales de la administración del proyecto son:

- Proveer una estructura para la administración de proyectos de software *intensivos*.
- Proveer guías prácticas para planeación, obtención de personal, ejecución y monitoreo de proyectos.
- Proveer una estructura para la administración de riesgos.

Sin embargo, el RUP no cubre todos los aspectos de la administración de proyectos. Por ejemplo:

- Administración de personal.
- Administración de presupuestos.
- Administración de contratos.

Flujo: Modelado de Negocios.

Los principales objetivos del modelado de negocios son:

- Entender la estructura y la dinámica de la organización en donde un sistema será colocado.
- Entender los problemas actuales en la organización e identificar potenciales mejoras.
- Asegurar que el cliente, usuarios finales y desarrolladores tengan un entendimiento común del objetivo de la organización.
- Derivar los requerimientos del sistema necesarios para soportar el objetivo de la organización.

Para lograr el cumplimiento de estos objetivos, el modelado de negocios describe el desarrollo de una visión del objetivo de la organización y, basado en esta visión, definir el proceso, los roles y las responsabilidades de la organización en un modelo de negocios.

Este modelo comprende el modelo de casos de uso de negocios y el modelo de objetos del negocio.

Complementario a estos modelos, se desarrollan por lo general los artefactos:

- Especificaciones de negocio suplementarias.
- Glosario.

Cuando es necesario el modelado de negocios?

El modelado de negocios no siempre es necesario realizarlo. Este se recomienda mientras el proyecto este más ligado a los procesos de negocios. También se considera si involucra a una mayor cantidad de gente y manejo de información del negocio.

El tamaño del modelado de negocios también debe encontrar sus propios límites, de acuerdo a las áreas de la organización que se vean involucradas.

Trabajadores.

Los principales trabajadores involucrados en el modelado de negocios son:

- **Analista del proceso de negocios**. Este analista encabeza y coordina el modelado de casos de uso de negocios delimitando el modelado de la organización.
- **Diseñador de negocios**. Detalla la especificación de parte de la organización a través de la descripción de casos de uso del negocio. Determina los trabajadores de negocios y entidades de negocios necesarios para realizar los casos de uso, y como estos trabajaran para la realización de los casos de uso.
- **Revisor de negocios**. Revisa los artefactos generados.

Flujo: Requerimientos.

Un proyecto difícilmente puede ser exitoso sin una especificación **correcta** y **exhaustiva** de los requerimientos. Son una descripción de las necesidades o deseos de un producto.

Un requerimiento es descrito como una condición o capacidad que el sistema debe cumplir.

Los objetivos principales del flujo de requerimientos son los siguientes:

- Establecer y mantener acuerdos con los clientes de lo que el sistema deberá hacer.
- Proporcionar a los desarrolladores del sistema una mejor comprensión de los requerimientos del sistema.
- Ofrecer una base para la planeación del contenido técnico de las iteraciones.
- Dar una base para la estimación del tiempo y costo del desarrollo del sistema.
- Ayudar a definir la interfaz del usuario, enfocándose en las necesidades y objetivos de los mismos.

Existen varios atributos en los requerimientos, Robert Grady[3] los

[3] Grady era trabajador de HP cuando clasifico los atributos de un sistema de software en lo que se conoce como FURPS

clasificó bajo el acrónimo FURPS :

- **Functionality**. La funcionalidad, especifica las acciones que un sistema debe ejecutar. Estos requerimientos son los que expresan el comportamiento del sistema a través de la especificación de condiciones de entradas y salida. Estos son descritos a través de los casos de uso.
- **Usability**. La usabilidad, se refiere a requerimientos derivados de factores humanos –estéticos, facilidad de uso y facilidad de aprendizaje- y de consistencia en la interfaz con el usuario, documentación del usuario y materiales de capacitación.
- **Reliability**. La confiabilidad, es la categoría para los requerimientos de frecuencia y severidad de fallas, recuperación, y precisión de la información.
- **Performance**. El rendimiento es el conjunto de requerimientos que impone condiciones sobre los requerimientos funcionales: tiempo de respuesta, velocidad, promedio de transacciones, etc.
- **Supportability**. El soporte y mantenimiento, especifica la capacidad del sistema a ser mantenido y actualizado.

Las últimas 4, son categorías de requerimientos no funcionales, ya que son necesidades que no exhiben un comportamiento directo en el sistema.

Trabajadores involucrados.

Los principales trabajadores involucrados:

- **Analista de Sistemas.** Este dirige y coordina la obtención de los requerimientos y el modelado de casos de uso, delineando la funcionalidad del sistema y delimitándolo.
- **Especificador de Casos de Uso.** Detalla una parte o toda la funcionalidad del sistema, describiendo los aspectos de los requerimientos en uno o varios casos de uso.
- **Diseñador de Interfaz con el Usuario.** Es responsable de seleccionar el conjunto de casos de uso que muestren las interacciones esenciales del usuario con el sistema. Trabajando con los usuarios, el diseñador desarrolla los prototipos de interfaz con el usuario, además de ajustar los casos de uso a el diseño de la interfaz.
- **Arquitecto.** Involucrado principalmente en las primeras iteraciones, junto con el analista de sistemas y especificador de casos de uso, para verificar la integridad de la arquitectura en los casos de uso significativos.

Modelo de 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***

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

Un actor, es un rol que un usuario puede tener dentro del sistema al interactuar con este. Un usuario puede ser una persona o un sistema externo.

Los casos de uso son historias o casos de utilización de un sistema.

- Describen 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, 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.

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. Sin embargo, el Proceso Unificado proporciona algunas consideraciones adicionales.

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.>

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 **flujo normal de los eventos**, paso por paso. El flujo normal también es llamado **flujo básico**

Formato de ejemplo:

Caso de uso: <Nombre del caso de uso>

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

Descripción: **<**Descripción del propósito del caso de uso>

Tipo: <Tipo de caso de uso.>

Flujo 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.

Flujos 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.

Descripción: Capturar una venta y su pago en efectivo. 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.

Flujo 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 captura del producto.	5. Calcula y presenta el total de la venta.

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.

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

11. Registra la venta concluida.

12. El Cliente se marcha con los artículos comprados.

Flujos alternos.

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

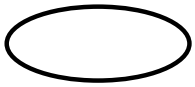
Diagramas de casos de uso.

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

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

Tiene dos componentes esenciales: actores y casos de uso.

Notación para casos de uso:



Caso de Uso

Notación para el actor:[4]

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.

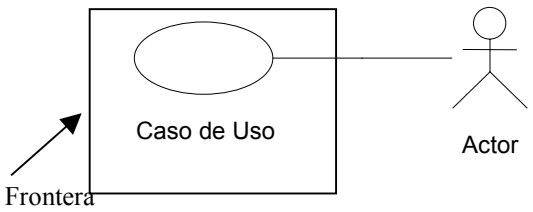
[4] Este es el icono estándar de UML, sin embargo a veces se utiliza un icono de una computadora para diferenciar al actor que es un sistema de cómputo y no una persona.

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:



Relaciones entre Casos de uso.

Aparte de los vínculos entre los actores y los casos de uso, UML define tres tipos de vínculos[5]:

include (incluir)

Una relación *include* ocurre cuando se tiene una porción de comportamiento que es similar en más de un caso de uso y no se quiere copiar

[5] Antes eran dos: *use* y *extend*, pero *use* se cambio por *include* y se agregó *specielize* a partir de UML 1.3

la descripción de tal conducta. Es una técnica similar a realizar factorización a través de subprogramas.

Para *include* 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.

Extend (extiende)

Una relacion *extend* se ocupan cuando se tiene un caso de uso que es similar a otro, pero que hace un poco más. Es usada para incluir comportamiento adicional bajo situaciones particulares.

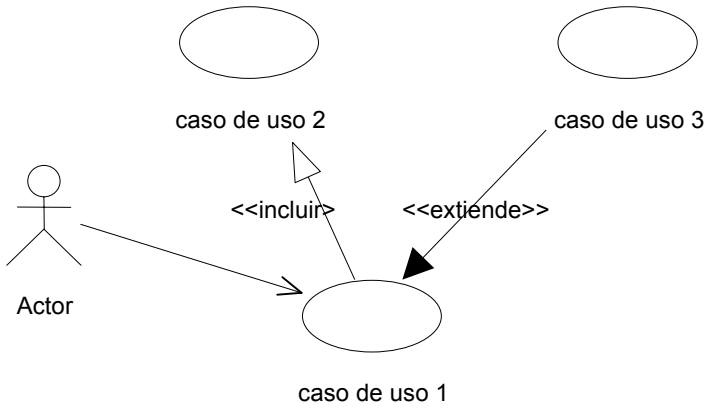
En esta relación, los actores tienen que ver con los casos de uso que se están extendiendo. Un actor entonces se encargará tanto del caso de uso base como de todas las extensiones.

Specialize **(**especializar**)**

Una relación *specialize* a un caso de uso más general. El caso de uso especializado refina el flujo de eventos original.

En conclusión se puede decir que:

- Es factible utilizar *extend* cuando se describa una **variación** de una conducta normal.
- Se ocupa *include* para **repetir** cuando se trate de varios casos de uso y desee evitar redundancia.



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 normalmente 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. Crear el diagrama de casos de uso.
4. Escribir los casos de uso en el formato 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. Solo se elaboran 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 del modelo de casos de uso.

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.

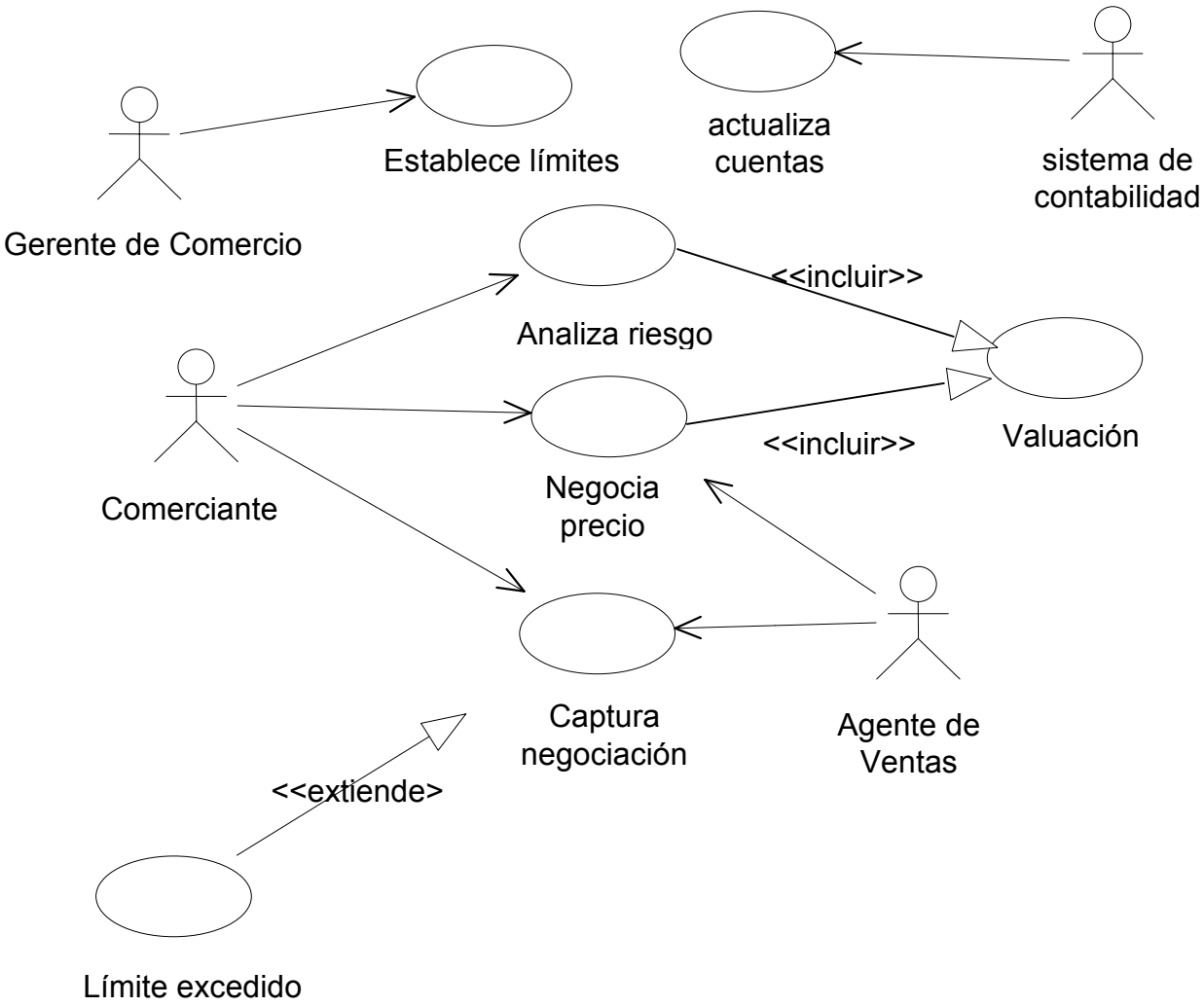
El número de casos de uso puede variar de un sistema a otro y de acuerdo al nivel de granularidad de los diseñadores.

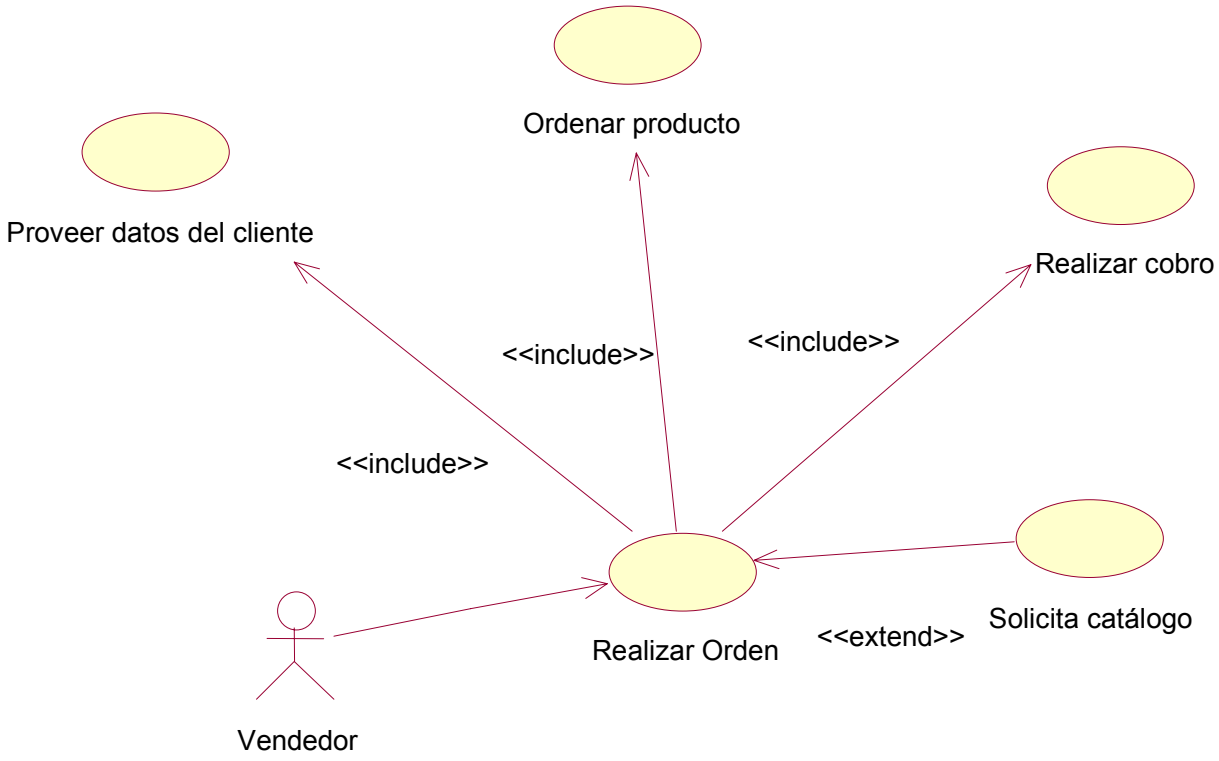
Es posible encontrar el término *script* como sinónimo al de casos de uso, llamados así por el autor Ian Graham.

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. (una instancia 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




Ordenar producto
Proveer datos del cliente
Realizar cobro
<<include>>
<<include>>
<<include>>
Solicita catálogo
Realizar Orden
<<extend>>
Vendedor

Flujo: Análisis y Diseño.

Los principales propósitos del flujo de análisis y diseño:

- Traducir los requerimientos dentro de una especificación que describa como implementar el sistema.
- Establecer una arquitectura robusta para el sistema, que sea fácil de entender, construir y evolucionar.
- Adaptar el diseño al ambiente de implementación.

Diferencia entre análisis y diseño.

El propósito del análisis es transformar los requerimientos del sistema en una estructura que pueda ser convertida en software, a través de un conjunto de clases y subsistemas.

Esta transformación es manejada por los casos de uso y esta complementada por los requerimientos no funcionales. El análisis se enfoca sobre el manejo de los requerimientos funcionales.

Esto se hace para simplificar la representación, de forma que el análisis expresa una imagen cercana a un sistema ideal.

El diseño por su parte, adapta los resultados del análisis a las restricciones impuestas por los requerimientos no funcionales, ambiente de implementación, requerimientos de rendimiento, etc.

El diseño es una refinación del análisis. Este se enfoca en la optimización del diseño del sistema mientras asegura cubrir todos los requerimientos.

Trabajadores y artefactos involucrados.

Los principales trabajadores que son necesarios en el flujo de análisis y diseño se presentan a continuación:

- **Arquitecto de software.** Este dirige y coordina las actividades y artefactos utilizados en esta parte del proyecto. Establece la estructura global de cada vista de la arquitectura: la vista de descomposición, el agrupamiento de elementos, y las interfaces entre los principales agrupamientos.
- **Diseñador.** Define las responsabilidades, operaciones, atributos y relaciones de una o muchas clases y determina como estas serán ajustadas al ambiente de implementación

Opcionalmente pueden ser necesarios los siguientes trabajadores:

- **Diseñador de base de datos**. Es necesario cuando el sistema requiere de una base de datos.
- **Revisor de arquitectura y revisor de diseño**. Son especialista que revisan los artefactos producidos por el flujo de trabajo.

Los artefactos clave del análisis y diseño son:

- El **Modelo de Diseño**, el cual es el mayor documento del sistema en construcción.
- El **Documento de Arquitectura del Sistema**, el cual captura varias vistas de la arquitectura del sistema.

Opcionalmente se genera:

- El **Modelo de Análisis**, el cual contiene la realización de los casos de uso y análisis de clases en un nivel previo al del diseño.

Definir una Arquitectura Candidata

Esta etapa está compuesta de las actividades de *análisis de la arquitectura*, realizada por el arquitecto, y *análisis de casos de uso*, realizado por el diseñador. El propósito de esta etapa es:

- Crear un esqueleto inicial de la arquitectura del sistema, a través de la definición de:
    - Un conjunto inicial de elementos significativos para la arquitectura, para ser usados como base para el análisis,
    - Establecer un conjunto inicial de mecanismos de análisis.
    - Una división y organización inicial del sistema.
    - Las realizaciones de los casos de uso que serán dirigidos en la iteración actual.

- Identificar clases de análisis para la arquitectura a partir de los casos de uso.
- Actualizar las realizaciones de los casos de uso con la interacción de las clases de análisis.

Actividad: Análisis de la Arquitectura

Los propósitos de esta actividad son:

- Definir los patrones arquitectónicos, mecanismos clave y convenciones de modelado para el sistema.
- Definir la estrategia de reuso.
- Ayuda a la planeación del proceso.

El modelo de casos de uso muestra la Vista de Casos de Uso; mientras que el análisis de la arquitectura se enfoca en la Vista Lógica del sistema.

Convenciones de Modelado

En las convenciones se definen los diagramas y elementos del modelado que serán usados, las reglas de uso de los elementos y de los diagramas, convenciones de nombres, etc.

Las convenciones proporcionan un estilo guía para el modelo, definiendo por ejemplo:

- Herencia múltiple no será usada.

- Todos los paquetes deben mostrarse en un diagrama principal mostrando su contenido.
- Los nombres de clases deberán empezar con mayúscula.
- Los nombres de métodos y atributos deberán empezar con minúscula.

Mecanismos de análisis

Los mecanismos de análisis capturan los aspectos clave de una solución en una forma independiente de la implementación. Estos forman parte de los mecanismos de arquitectura, los cuales encierran decisiones estratégicas sobre uso de estándares, políticas y cuestiones prácticas.

Los mecanismos de arquitectura se divieden entonces en:

- Mecanismos de análisis.
- Mecanismos de diseño.
- Mecanismos de implementación.

Los mecanismos de análisis ayudan a definir el comportamiento de una clase o componente, o la cooperación necesaria entre clases y/o componentes.

Algunos ejemplos de mecanismos de análisis:

- Persistencia.
- Comunicación.
- Distribución.
- Manejo de transacciones.

- Seguridad.
- Redundancia.
- Interfaz heredada.
- Base de datos heredada.

Identificación de abstracciones clave.

El propósito de este paso es obtener un primer acercamiento a los elementos clave que colaboran en el sistema para llevar a cabo la solución del problema. Estos pudieron haberse identificado inicialmente en el modelo de negocios (si lo hubiera), pero durante el análisis y diseño, los elementos se centran más en la solución.

La identificación de elementos implica:

- Definir clases de análisis preliminares.
    - Conocimiento del dominio.
    - Requerimientos.
    - Glosario
    - Modelo de dominio o Modelo de negocios
- Definir relaciones entre clases de análisis.
- Modelar clases de análisis y relaciones en diagramas de clases.
- Mapear clases de análisis a los mecanismos de análisis necesarios.

Arquitectura de capas inicial.

En este paso se trata de determinar los paquetes o capas más generales en que podría dividirse al sistema y las relaciones existentes entre estos. Se supone que este sea un primer acercamiento al control de la arquitectura.

Los paquetes, capas y dependencias serán refinadas posteriormente en el diseño de la arquitectura.

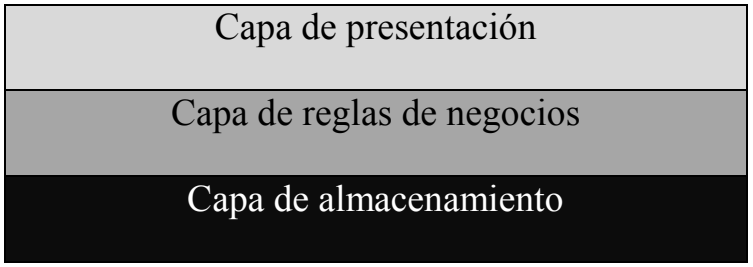
Aunque existen varios modelos para la organización de las aplicaciones, el más popular está basado en capas.

Modelo de capas típico: Tres Capas de Aplicaciones.

- La capa de **Presentación**: la presentación de información de puntos externos al sistema (También conocido como la interfaz de usuario). La capa de la presentación contiene la lógica de la representación de la información a un origen externo y obtiene entrada de que origen. En muchas cosas el origen externo es un humano trabajando en una terminal, aunque el origen externo también puede ser un sistema de robot, un teléfono, o algún otro dispositivo de entrada. La lógica de presentación generalmente provee un menú de opciones para permitir al usuario navegar a través de las partes diferentes de la aplicación, y que manipula la entrada y campos de rendimiento en la representación visual dispositivo. Frecuentemente la capa de presentación también ejecuta una cantidad limitada de validación de datos de entrada.

- La capa de las **Reglas de los Negocios**: Los procesos que implementan e imponen el contexto de los datos dentro del uso especificado de negocios. Un componente de negocio contiene la lógica de aplicación la cual gobierna las funciones del negocio y proceso ejecutado por la aplicación. Estas funciones y procesos son invocados uno u otro por un componente de presentación cuando unas solicitudes de usuario una opción, o por otra función de negocio. Las funciones de negocio generalmente ejecutan algún tipo de manipulación de datos.

- La capa de **Almacenamiento**: Una facilidad para poder manipular, gestar y recuperar la información acerca de los negocios. Unos accesos del componente de datos contienen la lógica de las interfaces con un sistema de almacenamiento de datos tal como sistemas de base de datos o sistemas jerarquías de archivos, o con alguno otro tipo de origen de datos externo tal como un forraje de datos o un sistema de aplicación externo. Funciones de acceso de datos generalmente están invocadas por una función de negocios, aunque en aplicaciones sencillas ellos sean invocados directamente por un componente de presentación.



Artefacto: Diagramas de Paquetes, una introducción.

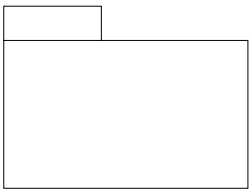
En muchas ocasiones es necesario fragmentar un sistema grande en sistemas más pequeños. En principio la programación orientada a objetos ofrece un gran cambio con respecto a la descomposición funcional. A pesar de esto, a veces tenemos en un sistema demasiadas clases y seria conveniente separarlas.

Otra razón es que la identificación inicial de paquetes ayuda a dividir al software en subsistemas, permitiendo la repartición de tareas y responsabilidades a los trabajadores.

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.

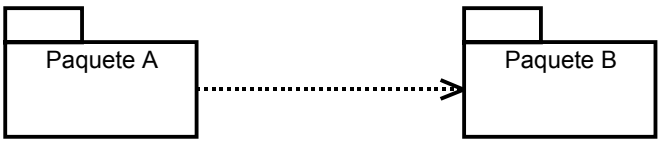
En general la dependencia esta relacionada con la visibilidad de una clase sobre otra.

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)

Actividad: Análisis de Casos de Uso.

En esta actividad es donde son identificadas las clases iniciales del sistema. Los objetivos principales son:

- Identificar las clases que intervienen en el flujo de eventos de los casos de uso.
- Distribuir el comportamiento de los casos de uso entre las clases, identificando las responsabilidades de las clases.
- Desarrollar las realizaciones de casos de uso que modelan las colaboraciones entre instancias de clases identificadas.

El nivel de detalle de la identificación de clases y sobretodo de la determinación de la colaboración entre clases puede variar, ya que se supone que estos modelos se vayan refinando en futuras iteraciones.

Además de el modelo de casos de uso, esta actividad se apoya en la identificación de abstracciones clave, mecanismos de análisis y de la especificación de capas del sistema.

Los pasos del análisis de casos de usos son:

1. Complementar la descripción del flujo de eventos de los casos de uso.
2. Para cada caso de uso:
    - buscar clases a partir de la descripción del comportamiento del caso de uso,
    - distribuir el comportamiento a las clases.

3. Para cada clase de análisis:
   - describir las responsabilidades,
   - describir los atributos y asociaciones,
   - describir los mecanismos de análisis.
4. Unificar clases de análisis.

Complementar la descripción del flujo de eventos de los casos de uso.

El flujo de eventos de los casos de uso se enfoca sobre las interacciones entre el usuario y el sistema, funcionando como una caja negra, donde los detalles internos son omitidos. Esto hace que no sea suficientemente la descripción de los casos de usos tal y como se tiene.

Es necesario entonces convertir esa caja negra en una caja blanca, donde se empiece a describir la manera en que el sistema trabajaría internamente.

Por ejemplo si se tiene un paso en un caso de uso de control escolar de la siguiente forma:

- El sistema despliega una lista de los cursos ofrecidos.

El texto deberá ser expandido posiblemente de la siguiente forma:

- El sistema recupera una lista de los cursos actualmente ofrecidos del catálogo de cursos en la base de datos heredada.

Artefacto: Diagrama de clases, una introducción.

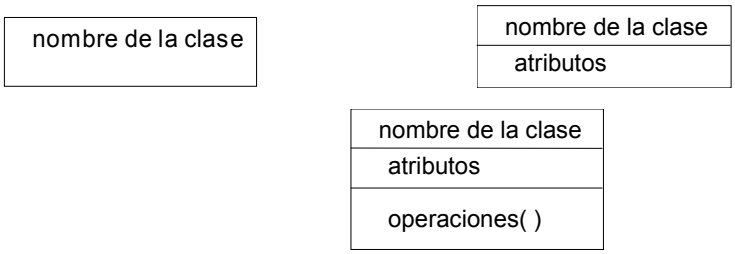
El diagrama de clases existe en diversas metodologías orientadas a objetos, este es muy útil para representar la estructura de un sistema en diversos niveles de comprensión.

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.

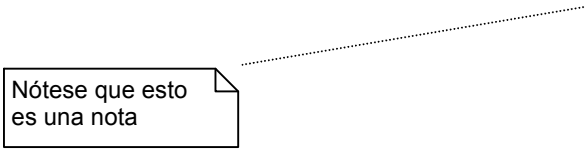
Los diagramas de clase muestran:

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

Notación de UML para representar una clase.



Notación de UML para representar una nota.



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 modo de presentación (de análisis, diseño 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.

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

Identificación inicial de clases.

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

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[6] 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.

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

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** a partir del **catálogo de productos** y a la **transacción de ventas** le agrega la información sobre el **producto**. Se muestran la **descripción** y el **precio del producto** actual.

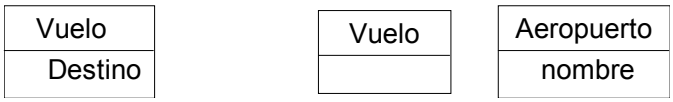
Esta técnica tiene la desventaja de depender del lenguaje utilizado, pues podemos tener diversas frases nominales para un mismo concepto o atributo.

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

Sin embargo, el error más frecuente cuando se comienza a identificar clases 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.

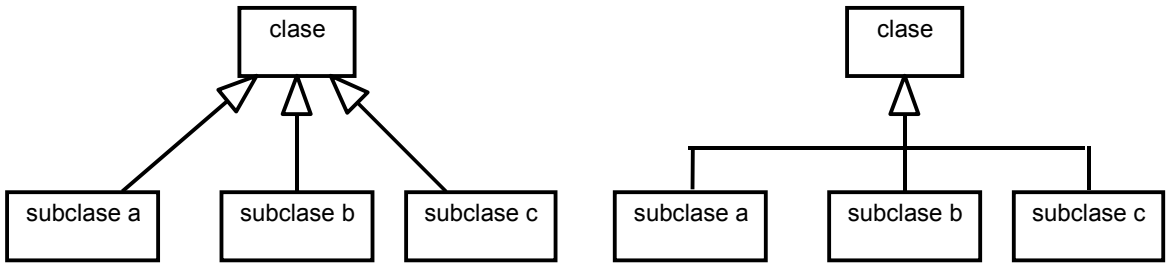
Si existe duda de eliminar o no una clase, es preferible conservarla; ya se eliminará en etapas posteriores si es innecesaria.

Generalización.

Es la actividad de identificar los aspectos comunes de las clases 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 reacciona

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:

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

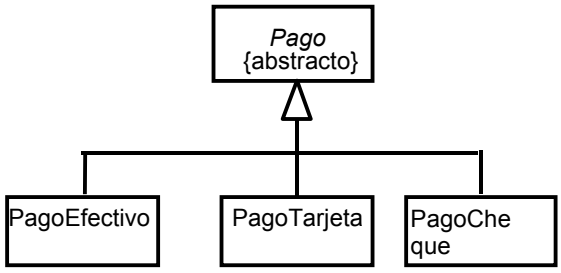
Clases abstractas.

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}**



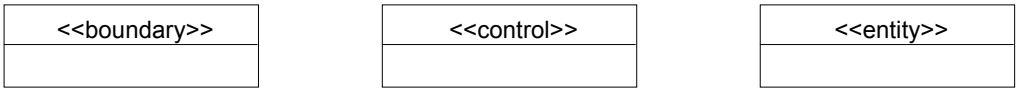
Buscar clases a partir de la descripción del comportamiento del caso de uso

El propósito de este paso es identificar un conjunto de clases de análisis candidatas que sean capaces de ejecutar el comportamiento descrito en los casos de uso.

Las clases identificadas deben de agruparse en tres perspectivas diferentes:

- *Boundary* (límite, frontera)
- *Control*.
- *Entity*.

El uso de estos estereotipos para representar las diferentes perspectivas, resulta en un modelo más robusto.



Clase *Boundary*.

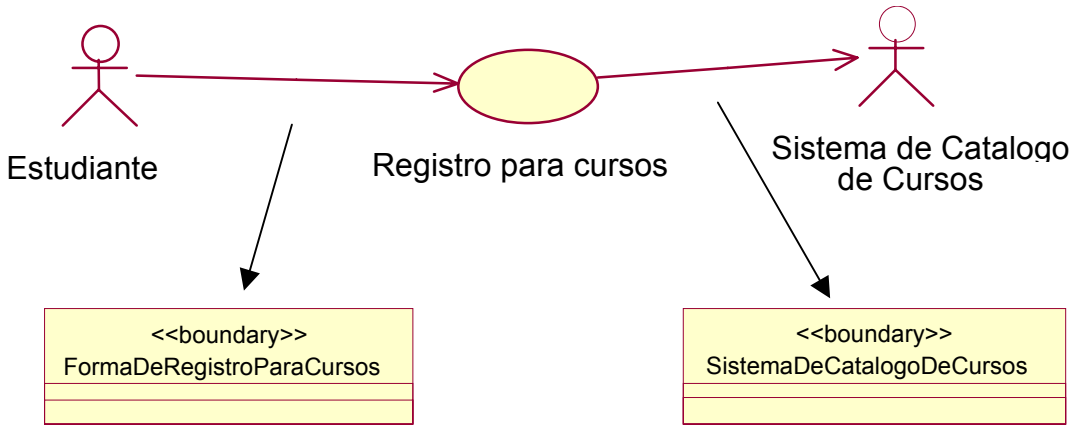
- Son la interfaz o intermediarios entre el sistema y elementos externos al mismo.
- Varios tipos:
  - Clases de interfaz con el usuario.
  - Clases de interfaz con el sistema.
  - Clases de interfaz con dispositivos.

- Existe una clase *Boundary* por cada pareja actor / casos de uso.

Una clase *boundary* muestra la interacción entre el sistema y los actores. Esta interacción involucra transformación y traducción de eventos y notificación de cambios en la presentación del sistema.

Los actores únicamente pueden comunicarse con clases de tipo *boundary*.

Las instancias de clases *boundary* normalmente tienen un tiempo de vida tan largo como una instancia de un caso de uso, pero esto no es forzoso.



Al identificar este tipo de clases es importante concentrarse en las responsabilidades, no irse a los detalles de la interfaz con el usuario o de cómo implementar protocolos de comunicación con sistemas o dispositivos.

Clase *Entity*.

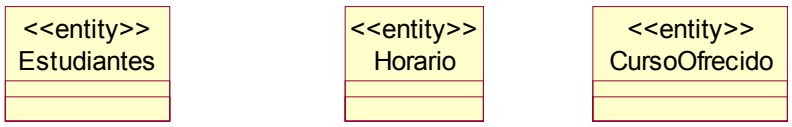
Las clases entidad representan los conceptos clave del sistema. Estas proporcionan otro punto de vista de comprensión sobre el sistema porque muestran la estructura lógica de los datos.

Las clases entidad representan almacenamiento de información del sistema. Las instancias de estas clases son usadas para guardar y actualizar información sobre algún fenómeno, como un evento, una persona, o algún objeto de la vida real. Estos objetos son usualmente persistentes, teniendo atributos y relaciones necesarias por un largo periodo de tiempo, algunas veces durante toda la vida del sistema.

Las principales responsabilidades de una clase entidad son almacenar y administrar información en el sistema.

Una clase entidad por lo general no es específica de un caso de uso; y en algunos casos, no es ni específica del sistema.

Los objetos entidad pueden tener un comportamiento tan complicado como los objetos de los otros estereotipos, pero esta complejidad esta relacionada con el objeto que representa.



Clase *Control*.

Las clases de control proporcionan coordinación sobre el comportamiento del sistema.

El sistema puede ejecutar algunas clases sin clases de control (usando únicamente clases *boundary* y *entity*), particularmente en casos de uso que

involucran simplemente la manipulación de información almacenada. Casos de uso más complejos generalmente requieren una o más clases de control para coordinar el comportamiento de los objetos en el sistema.

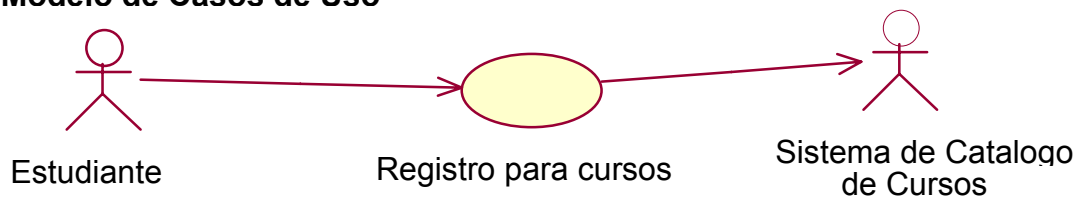
Las clases de control separan los objetos entidad de los objetos *boundary*, haciendo al sistema más tolerante a cambios. También separa el comportamiento de un caso de uso de los objetos entidad, haciendo más reusables los casos de uso.

Una recomendación para la identificación inicial de clases de control es definir una clase de control por caso de uso.

Las clases de control contribuyen a entender al sistema pues estas representan la dinámica del sistema, manejando las principales tareas y flujos de control.

Un objeto de control usualmente muere cuando termina la ejecución de un caso de uso.

Modelo de Casos de Uso



Modelo de Diseño

Distribuir el comportamiento.

El propósito de la distribución del comportamiento a las clases es:

- Expresar el comportamiento de los casos de uso en términos de la colaboración de las clases de análisis.
- Determinar las responsabilidades de las clases de análisis.

Esto que implica que una vez identificadas las clases de análisis, para cada flujo en los casos de uso:

- Asignar las responsabilidades del caso de uso a las clases de análisis.
- Modelar las interacciones entre las clases de análisis con diagramas de interacción.

Asignar responsabilidades.

La asignación de responsabilidades a las clases de análisis es una actividad crucial, ya que limita el alcance de las mismas, separando las tareas y ayudando a la construcción de un sistema más robusto.

Una forma inicial de asignar responsabilidades es a partir de los estereotipos de las clases, pues ya se sabe que de acuerdo a su estereotipo se tienen ciertas limitaciones:

- Clase *boundary*. Comportamiento que involucra comunicación con un actor.
- Clase *entity*. Comportamiento que involucra encapsulamiento y abstracción de datos.

- Clase *control*. Comportamiento específico para un caso de uso o una parte importante del flujo de eventos.

¿Quién tiene la responsabilidad sobre un dato?

- Si una clase tiene el dato, se asigna la responsabilidad de ese dato a la clase.
- Si múltiples clases tienen el dato:
    - Se pone la responsabilidad en una clase y se añaden relaciones a las otras clases.
    - Se crea una nueva clase, se asigna la responsabilidad en esta nueva clase y se agregan relaciones a las clases que necesiten llevar a cabo la responsabilidad.
    - Se pone la responsabilidad en una clase de control, y se añaden relaciones a las clases que requieran esa responsabilidad.

Sin embargo, hay que tener cuidado al agregar asociaciones o nuevas clases. Una nueva clase solo debe de ser creada si no es claro cual clase ya existente debe de ejercer la responsabilidad.

Modelar interacciones.

El modelado de interacciones permite ver la colaboración entre los objetos en un caso de uso en particular. Este paso se lleva a cabo a través de los diagrama de interacción que UML proporciona.

Diagramas de interacción.

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.

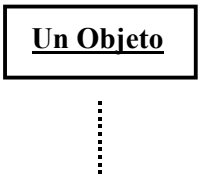
Existen dos tipos de diagramas de interacción:

- Diagrama de secuencia.
- Diagrama de colaboración.

Diagramas de Secuencia.

El diagrama de secuencia, resalta el orden de los eventos que suceden entre un objeto y otro.

Por lo general, se construye un diagrama de secuencia para cada caso de uso dentro de su flujo normal, y un diagrama para cada flujo alterno importante.



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.
- 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.

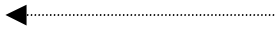
- **Condición. I**ndica que un mensaje se envía sólo si la condición es verdadera.

  Sintaxis: `[condición]`

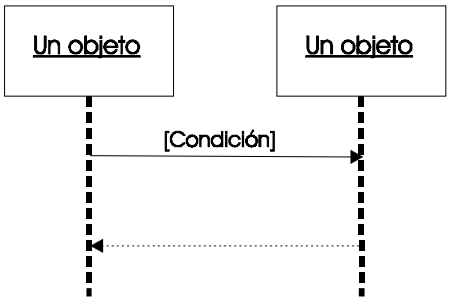
- **Marcador de iteración**. Muestra que un mensaje se envía muchas veces a varios objetos receptores.

  Sintaxis: `*[condición]`

- **Retorno**. 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



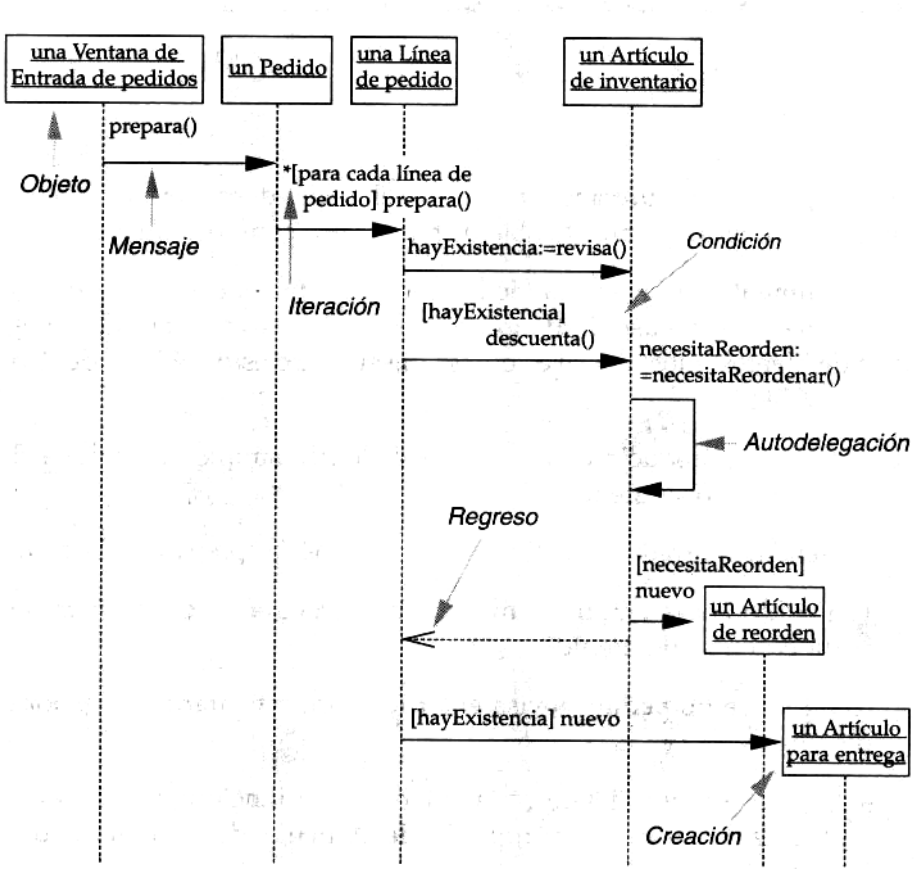
flecha de línea punteada.



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.

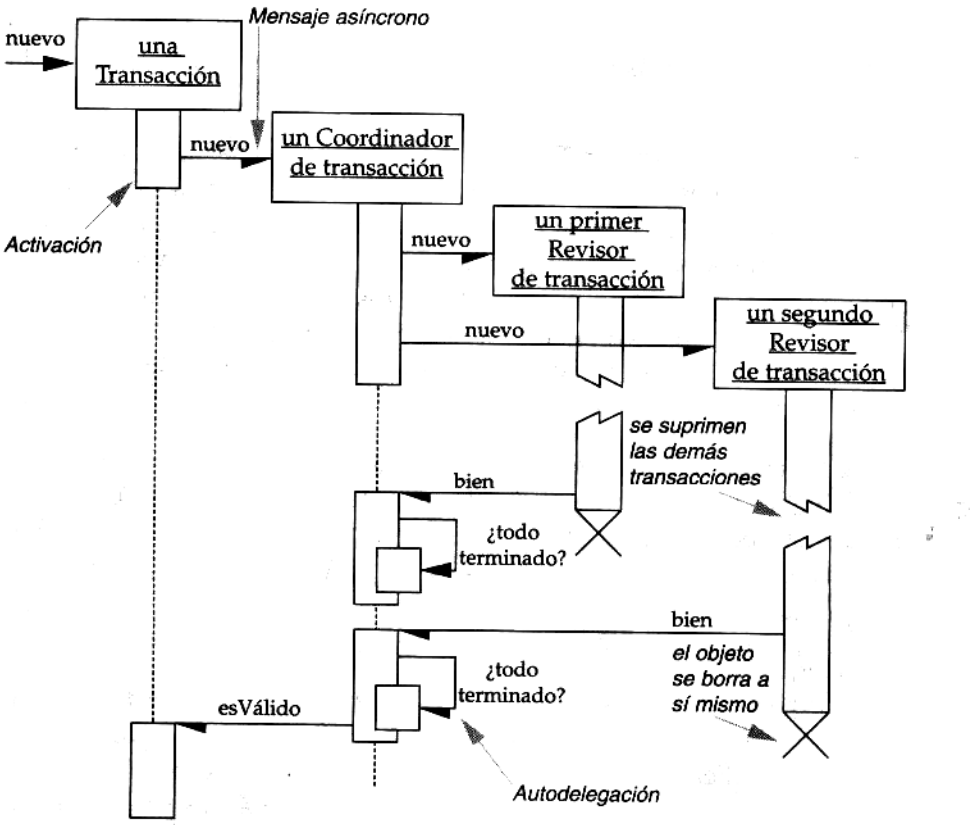
Un mensaje asíncrono puede:

- 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.



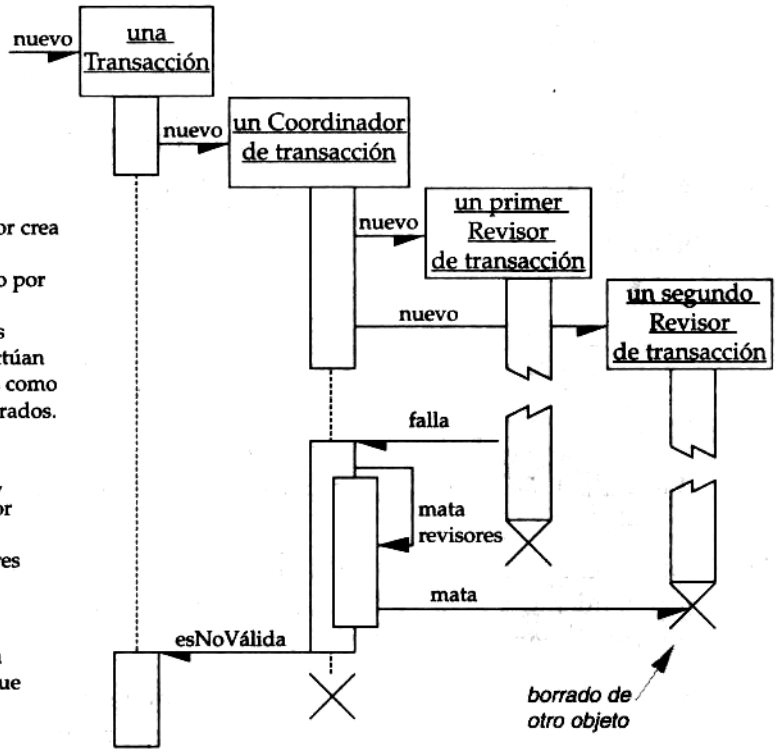
Cuando se crea una transacción...

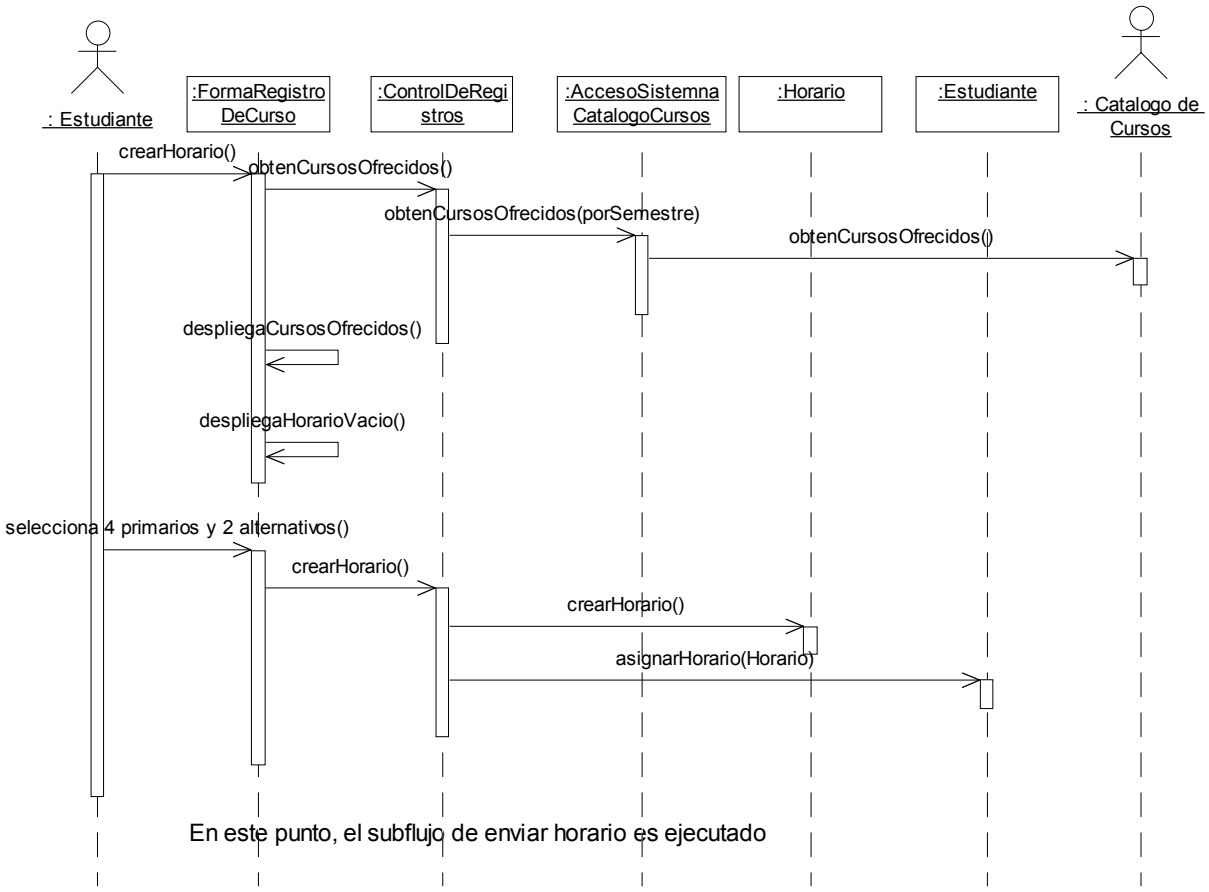
...crea un Coordinador que maneja la revisión.

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




: Estudiante
:FormaRegistro DeCurso
:ControlDeRegi stros
:AccesoSistemna CatalogoCursos
:Horario
:Estudiante
: Catalogo de Cursos
crearHorario()
obtenCursosOfrecidos()
obtenCursosOfrecidos(porSemestre)
obtenCursosOfrecidos()
despliegaCursosOfrecidos()
despliegaHorarioVacio()
selecciona 4 primarios y 2 alternativos()
crearHorario()
crearHorario()
asignarHorario(Horario)
En este punto, el subflujo de enviar horario es ejecutado

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, y estos mensajes se numeran por orden y precedencia.

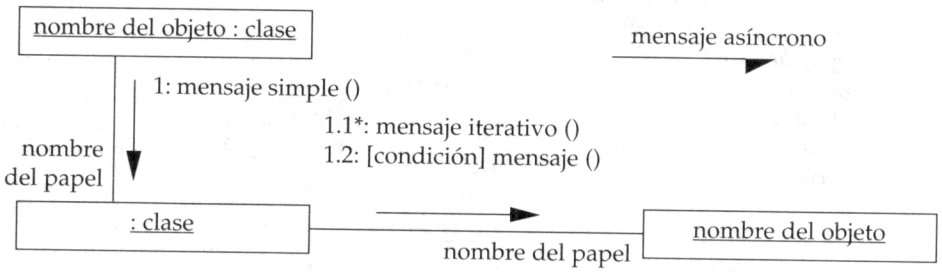
Los diagramas de colaboración describen las interacciones entre los objetos en un formato de grafo o red.

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 del objeto, la clase se debe de conservar los dos puntos como prefijo.

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:

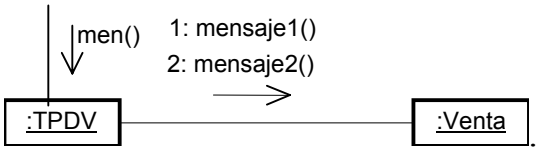
- Los mensajes entre objetos pueden representarse por medio de una flecha con un nombre y situada sobre una línea del vínculo. Puede haber un número indefinido de mensajes.
- Es posible que un objeto se mande un mensaje a si mismo (**autodelegación**).
- Los parámetros de un mensaje pueden anotarse dentro 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:

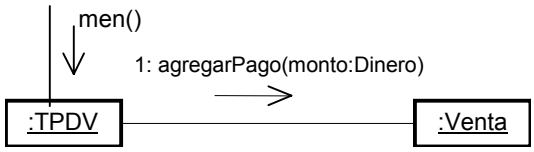
```
Retorno := mensaje (parámetro : tipoParámetro, ...)
: tipoRetorno
```

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

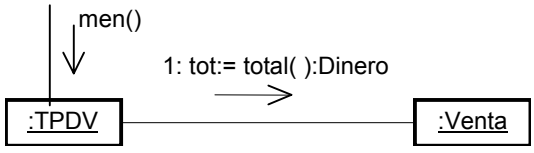
Ejemplo: paso de mensajes.



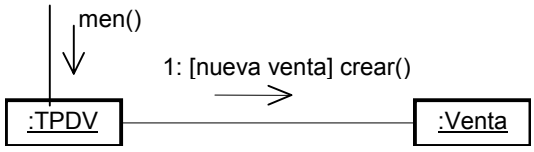
Ejemplo: paso de parámetros



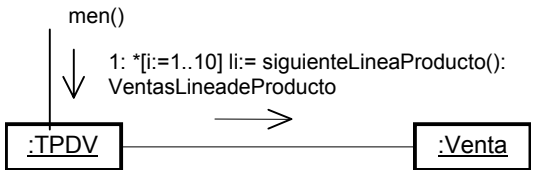
Ejemplo: valor de regreso.



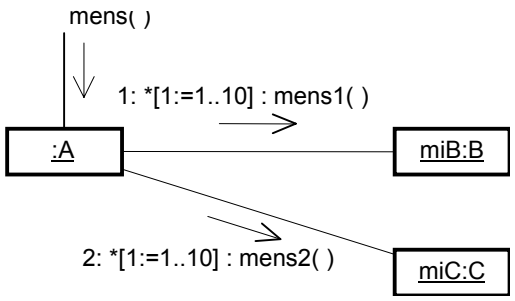
Ejemplo: condición



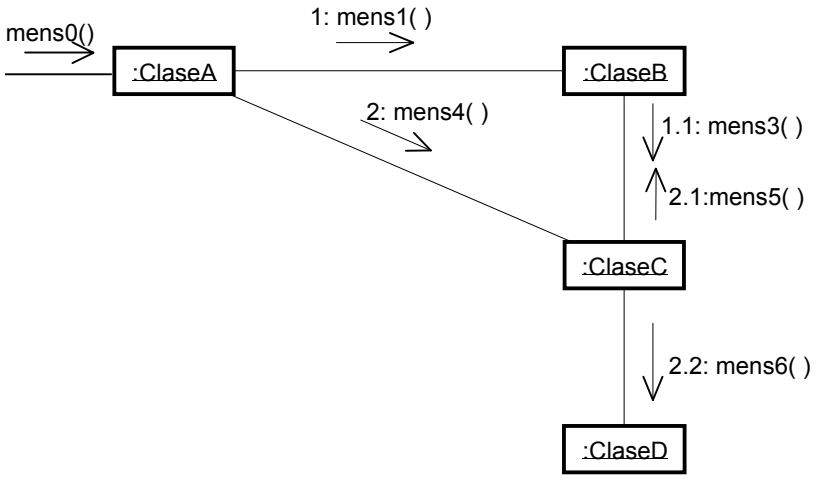
Ejemplo: iteración



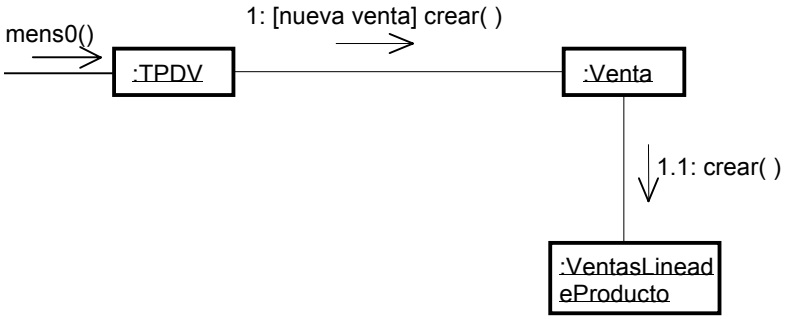
Ejemplo: iteración con mensajes a múltiples objetos de distintas clases.



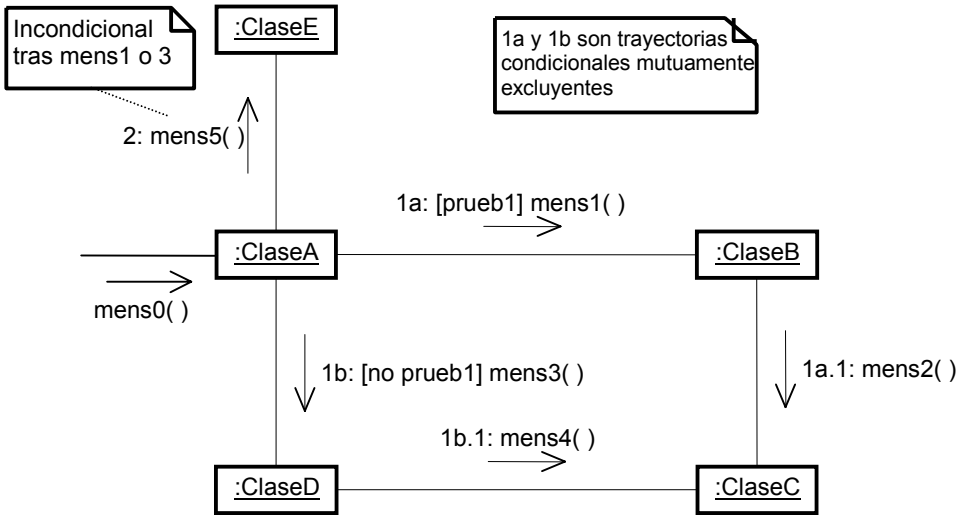
Ejemplo: numeración de secuencias.



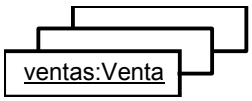
Ejemplo con mensaje condicional



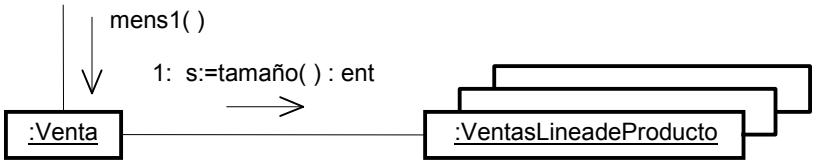
Ejemplo con trayectorias condicionales mutuamente excluyentes:



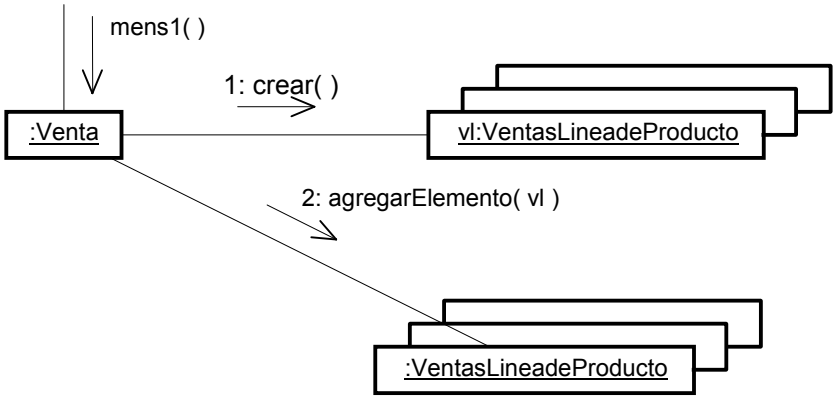
Representación de multiobjetos o conjuntos de instancias.



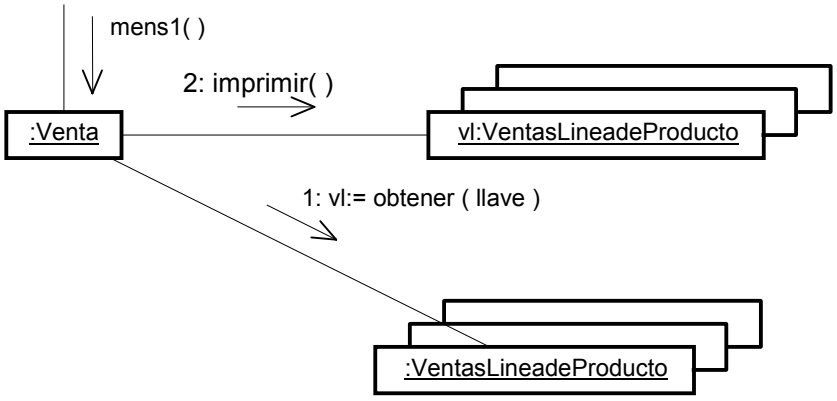
Ejemplo 1:



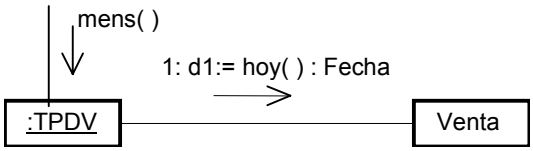
Ejemplo 2:



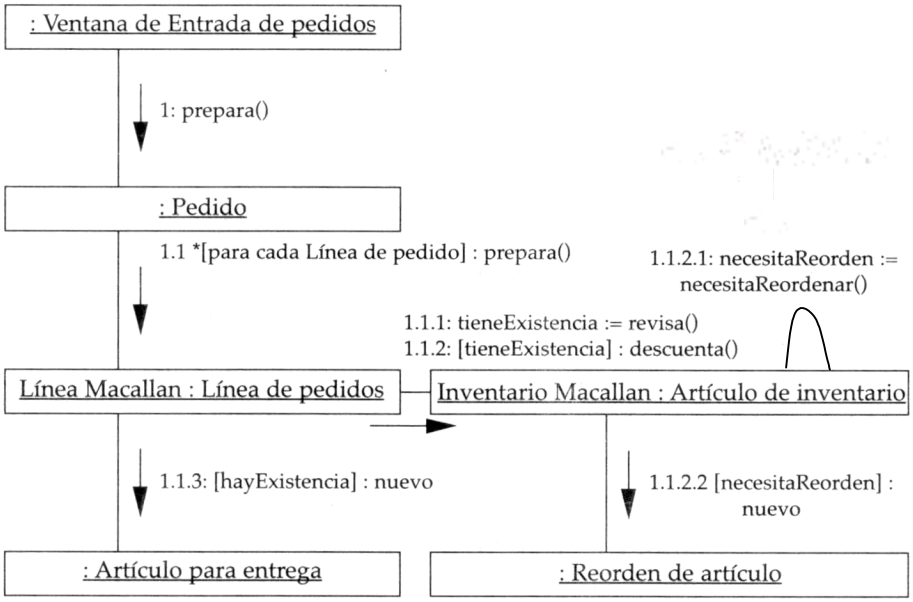
Ejemplo 3:

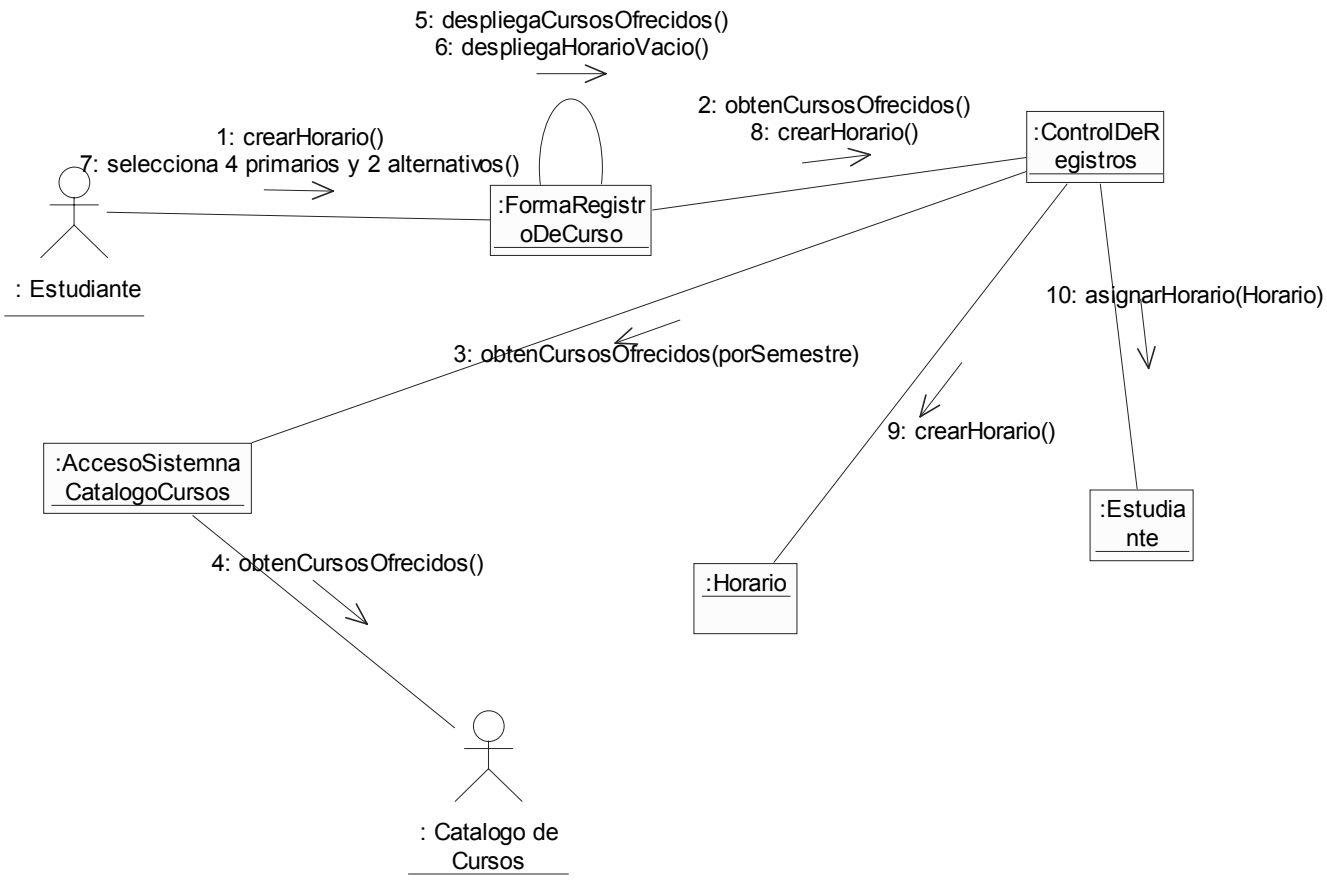


Ejemplo: mensaje enviado a una clase.



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** del mismo.

- 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.

Se debe construir un diagrama de colaboración por cada caso de uso, incluyendo tanto el flujo básico como los flujos alternos.

Sin embargo, si el diagrama es demasiado elaborado se considera que para cada evento externo al proceso que se está modelando, se construye un diagrama de colaboración cuyo mensaje inicial sea el de sus eventos.

Diagramas de Colaboración Vs Diagramas de Secuencia

Los diagramas de secuencia y colaboración expresan información similar, pero la muestran de diferentes formas.

Diagrama de Colaboración	Diagrama de Secuencia
Muestra relaciones adicionadas a las interacciones.	Muestra la secuencia explícita de los mensajes.
Es mejor para la visualización de patrones de colaboración.	Es mejor para visualizar el flujo global.
Es mejor para visualizar todos los efectos sobre el objeto.	La dimensión de tiempo es fácil de leer.
Fácil de usar en sesiones de "lluvia de ideas"	Un largo número de objetos es más fácil de manejar que en un diagrama de colaboración.
Pone énfasis en la colaboración estructural de un conjunto de objetos y proporciona una imagen de las relaciones y control que existe entre los objetos participantes en un casos de uso.	Es mejor para especificaciones de tiempo real y para escenarios complejos.

Ambos, ayudan a identificar objetos, clases, interacciones, y responsabilidades; ayudando a validar la arquitectura.

Describir Responsabilidades.

Para cada clase de análisis identificada se debe describir su responsabilidad.

Una responsabilidad es una orden de algo que se le puede solicitar a un objeto que proporcione.

Las responsabilidades involucran una o más operaciones en clases de diseño; estas pueden ser:

- La acción que el objeto ejecuta.
- El conocimiento que el objeto mantiene y proporciona a otros objetos.

En este nivel, las responsabilidades son derivadas de los mensajes en los diagramas de interacción. Para cada mensaje, se debe examinar la clase del objeto que recibe el mensaje. Si la responsabilidad no existe todavía, deberá crearse una nueva responsabilidad que porporcione el comportamiento solicitado.

Otras responsabilidades pueden derivarse de requerimientos no funcionales. Al definir las responsabilidades se deben tener en cuenta estos requerimientos no funcionales. Puede ser necesario complementar la descripción de alguna responsabilidad o crear una nueva responsabilidad para un requerimiento de este tipo.

Las responsabilidades de las clases de análisis pueden ser documentadas de dos formas:

- Como operaciones de “análisis”. Donde se de nombre a las de operaciones para describir las responsabilidades de las clases de

análisis. Estas operaciones de "análisis" serán probablemente cambiadas o evolucionarán en el diseño.

- Textualmente. Como parte de la descripción de las clases de análisis.

Diagrama de Interacción

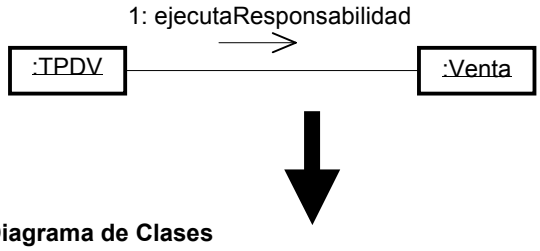
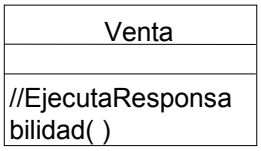
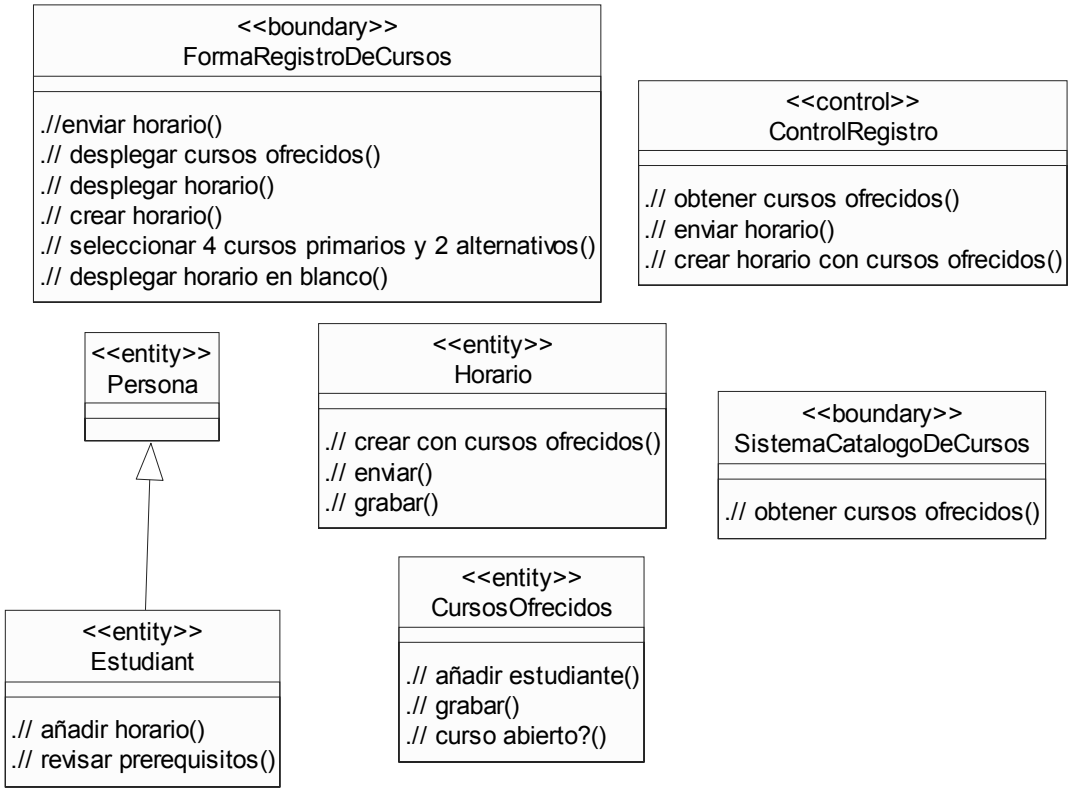


Diagrama de Clases





Mantener Consistencia.

Es importante examinar las clases y revisar, en orden de importancia :

- Clases con responsabilidades redundantes.
- Clases con responsabilidades aisladas.
- Clases con una responsabilidad.
- Clases sin responsabilidades.
- Mejorar la distribución del comportamiento.
- Clases que interactúan con muchas clases.

Describir Atributos y Asociaciones.

Atributos.

Los atributos son usados para almacenar información. Estos deben de considerarse atómicos y sin mayor responsabilidad que contener sus propios datos.

Durante el análisis, los tipos de los atributos pueden indicarse, aunque no es necesario que correspondan con el lenguaje de programación en uso.

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 diseño, 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 esta etapa, 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.

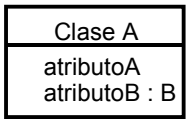
Notación de los atributos en UML

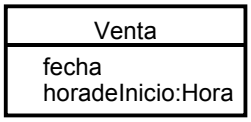
Dependiendo del detalle del diagrama y del modelo, la notación de un atributo puede mostrar:

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

Sintaxis:

visibilidad nombre : tipo = valor por omisión.





Identificación de los atributos

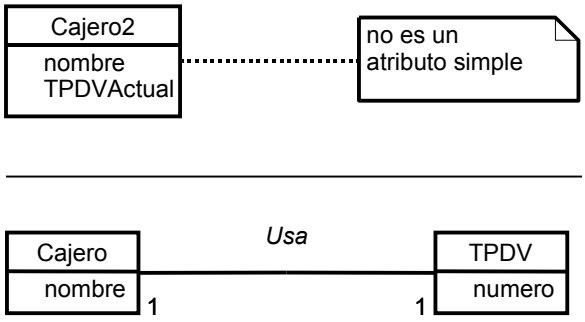
Cuando se reconocieron las clases para los casos de uso, aparecieron los primeros atributos, algunos de los cuales deben de ser agregados a una de las clases definidas.

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.

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 por ejemplo, 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.

Generalidades

Los atributos no deberían servir para relacionar clases en el modelo de análisis. 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.

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.

Se representa 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.

Por ejemplo, suponga en el sistema de ventas que este deba adaptarse a varias monedas al destinarse el software a varios países.

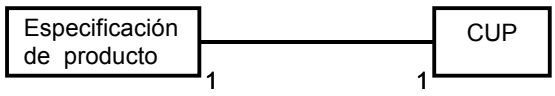
Además, en el diagrama de clases es factible mostrar un atributo compuesto dentro de la clase o como una clase independiente en el diagrama de clases:

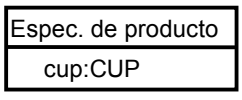
- Se trata de un valor del cual no se requiere tener la identidad de cada instancia, por lo que podría mostrarse en la sección de atributos de la clase.

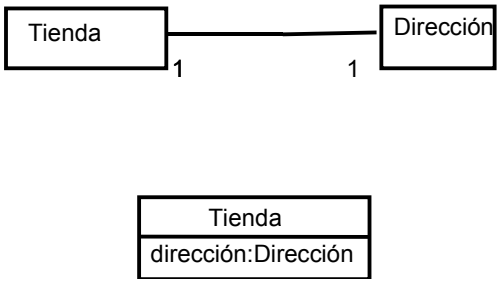
- Por ser un atributo con sus propias características, es factible mostrarlo como una clase independiente.

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 de análisis es una herramienta de comunicación; las decisiones sobre lo que debería mostrarse han de tomarse teniendo eso en cuenta.

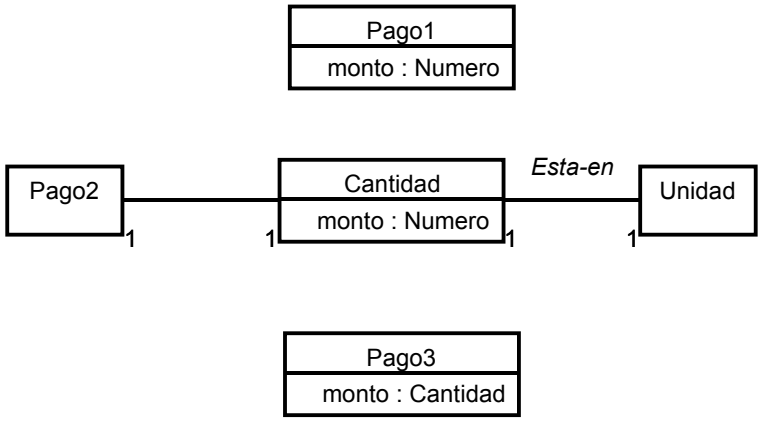
Ejemplo con el sistema de ventas para el código universal de productos y dirección:







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



- Recordar que los atributos son dependientes del dominio, por lo que se incluyen sólo características relevantes para el dominio del problema que se está modelando.
- Los atributos descubiertos deberán ser usados al menos en un caso de uso.
- Fuentes de posibles atributos: conocimiento del domino, requerimientos, glosario, modelo de negocios, etc.

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.

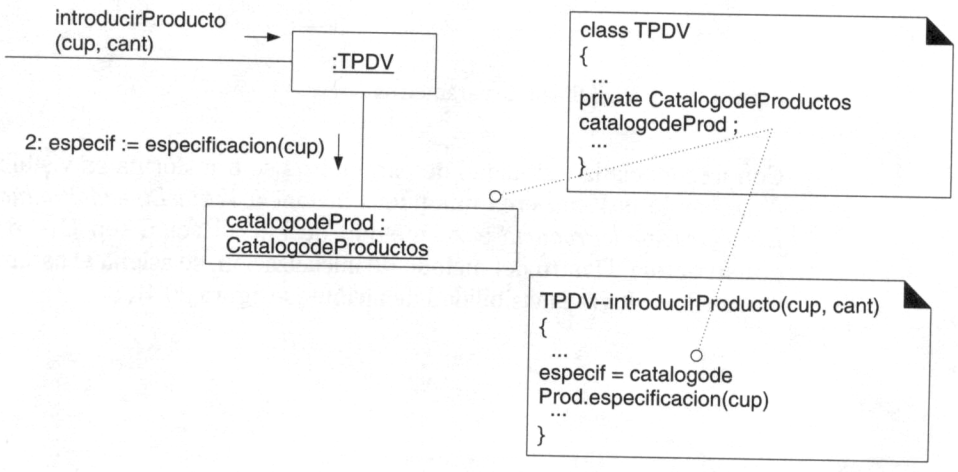
La visibilidad se refiere al alcance o ámbito de los objetos. Existen cuatro formas comunes de visibilidad:

- Visibilidad de atributos.
- Visibilidad de parámetros.
- Visibilidad declarada localmente.
- Visibilidad global.

Visibilidad de atributos.

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

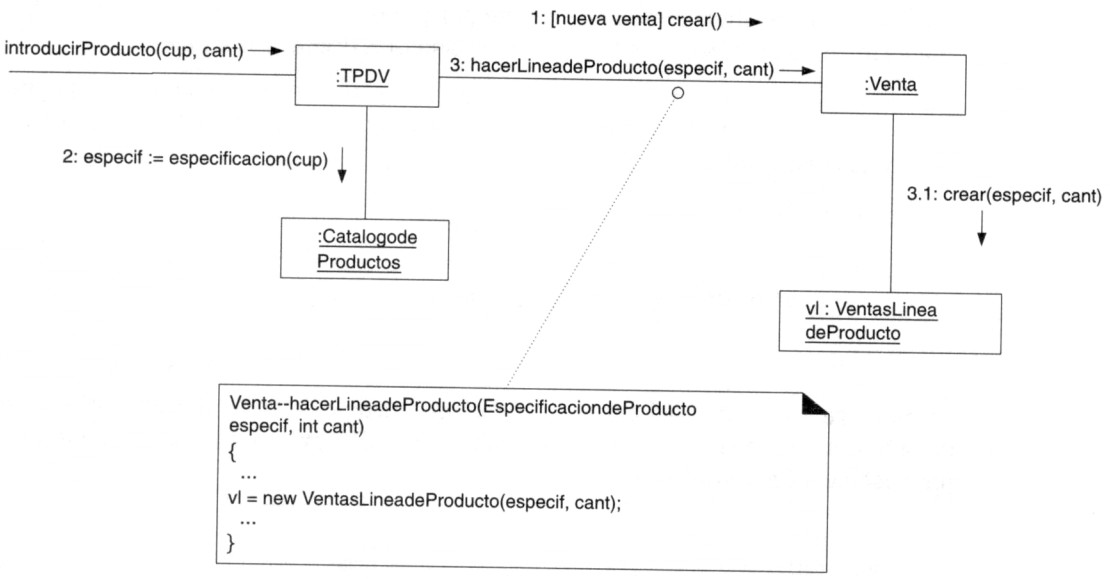
Se trata de una visibilidad relativamente permanente porque persiste mientras existan A y B. Es la forma más común de visibilidad.



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.

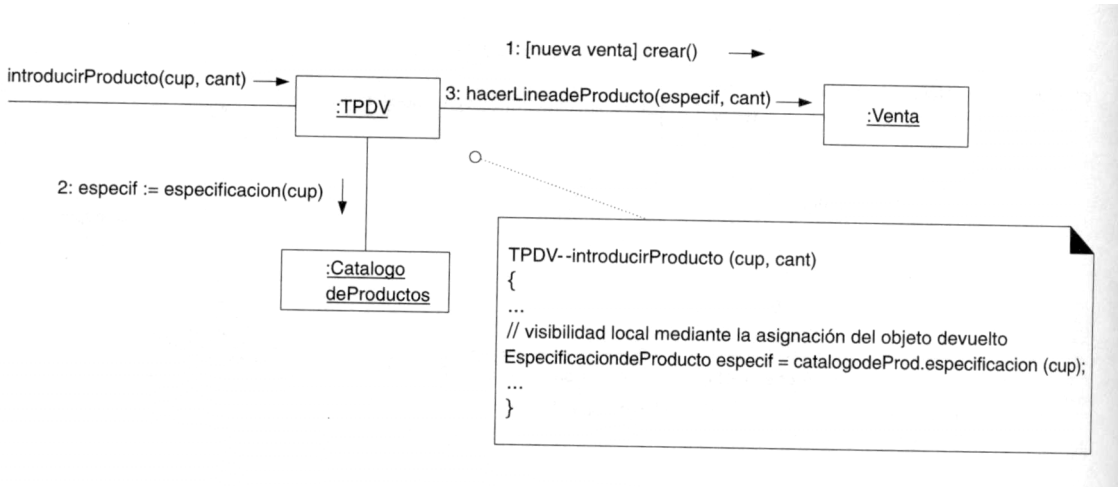


Visibilidad declarada localmente.

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.

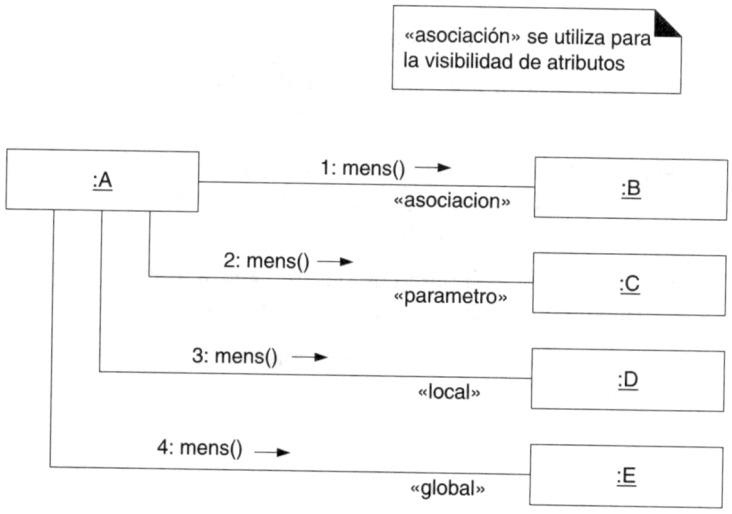


Visibilidad global.

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.



Asociación y Agregación

Asociación.

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

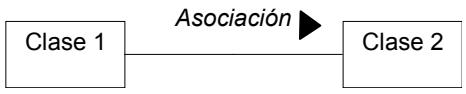
Desde la perspectiva de análisis, las asociaciones representan relaciones conceptuales entre clases.

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.

En el análisis, una asociación se considera abstracta 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.



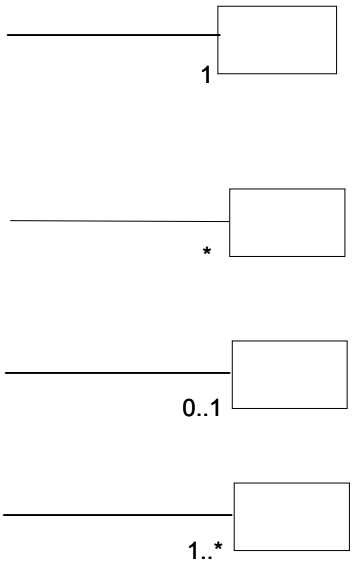
La flecha en el nombre es opcional e indica la dirección en que debe leerse el nombre de la asociación. En ausencia, por convención la asociación se lee de izquierda a derecha o de arriba hacia abajo.

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.

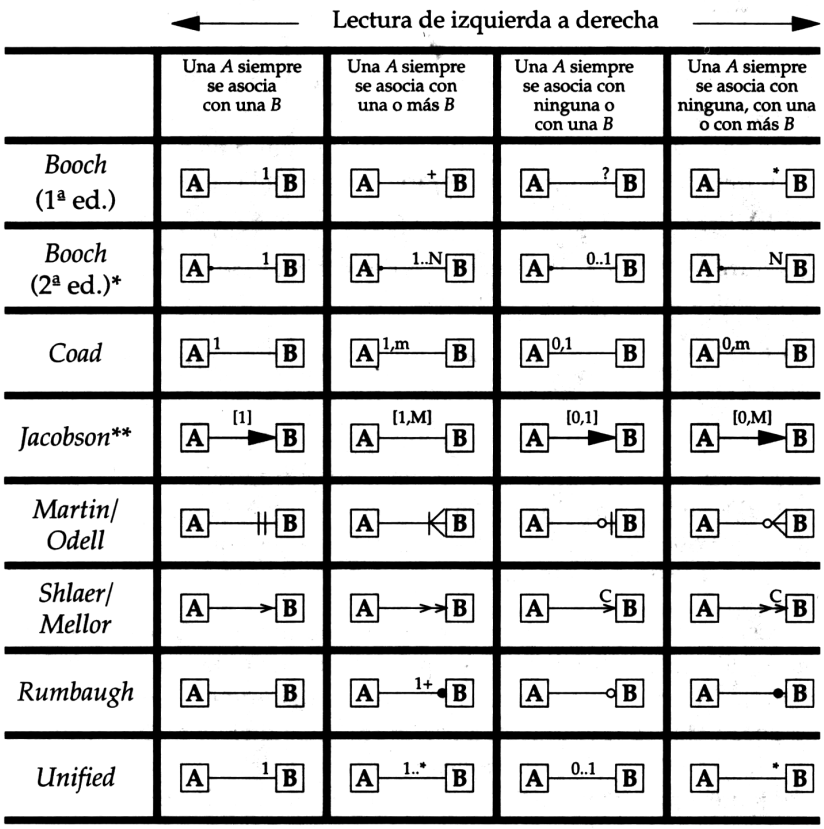
Notación de multiplicidad en UML:



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 (OMT de Rumbaugh).

Diferentes notaciones de multiplicidad o cardinalidad:



* puede ser unidireccional

** unidireccional

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.

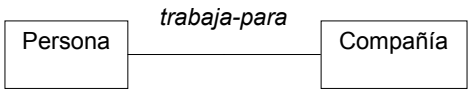
La lista completa se presenta a continuación:

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 transacción B	Pago- Venta 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 de análisis 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**.

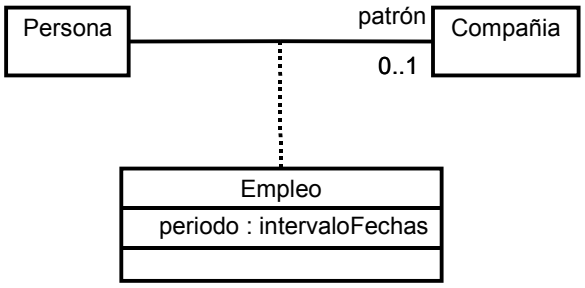
- Un buen modelo ocupa un punto intermedio de un modelo basado en la necesidad mínima de conocimiento y otro que contenga todas las relaciones posibles.

Clase de asociación.

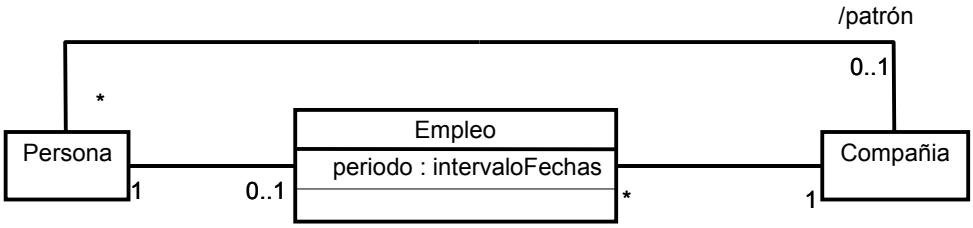
Las clases de asociación permiten añadir atributos, operaciones y otras características a las asociaciones.

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.

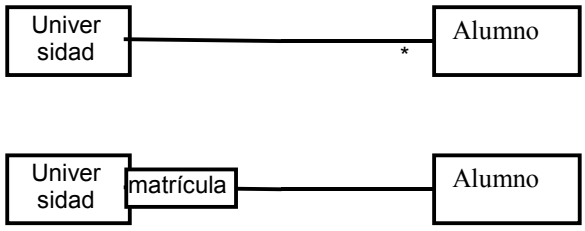
La clase de asociación es también utilizada al tener una asociación de muchos a muchos.

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.

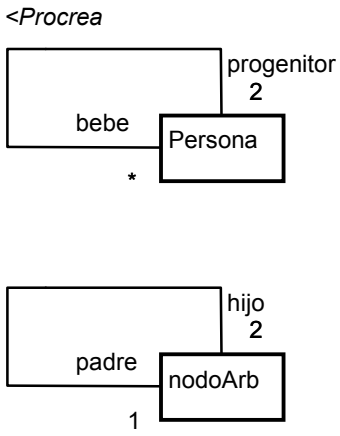
- El calificativo reduce la multiplicidad del lado opuesto de la asociación.
- Se comporta en términos de implementación como un índice de la asociación.

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.



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:

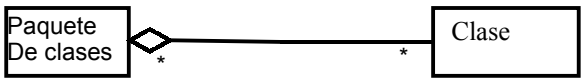
- Peter Coad ejemplificó la agregación como la relación entre una organización y sus empleados.
- Jim Rumbaugh afirmó que una compañía no es la agregación de sus empleados.

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.

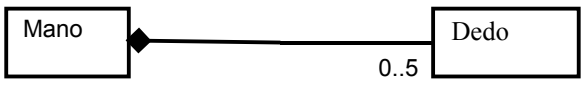


Composición.

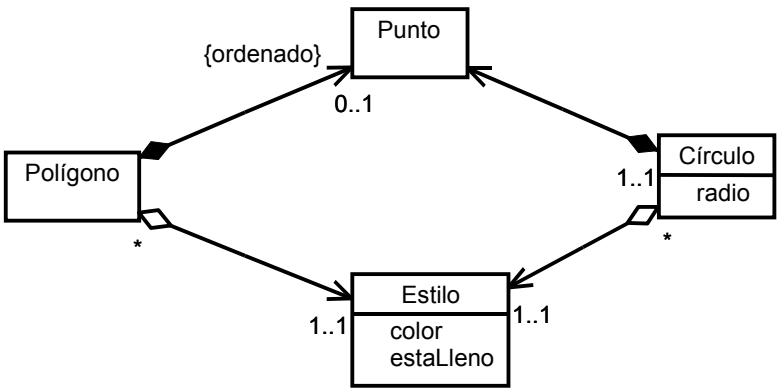
En la composición, el objeto componente puede pertenecer a un todo o compuesto único, y se espera que las partes vivan y mueran con el todo. Nadie más puede contener al componente.

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 a 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.

Descripción de Mecanismos de Análisis.

En al análisis de la arquitectura, los posibles mecanismos de análisis son identificados y definidos:

- Se tienen todos los mecanismos de análisis en una lista.
- Se relacionan todas las clases a los mecanismos:

Clases de Análisis	Mecanismos de Análisis

Los mecanismos de análisis deben de ser identificados y documentados junto con sus características. Estas características son las ayudan a discriminar un cierto rango de diseños potenciales. Las características son en parte funcionales, y en parte de tamaño y desempeño.

No todas las clases tienen mecanismos de análisis asociados. También puede suceder que algunas clases requieran de varios mecanismos.

Ejemplo:

Clases de Análisis	Mecanismos de Análisis
Estudiante	Persistencia, Seguridad
Horario	Persistencia, Seguridad
CursoOfrecido	Persistencia, Interfaz heredada
Curso	Persistencia, Interfaz heredada
ControlDeRegistro	Distribución

Unificar Clases de Análisis

El propósito de este paso es asegurar que cada clase de análisis represente un simple concepto bien, donde estas no estén mezclando responsabilidades.

Antes de que el trabajo del diseño de la arquitectura pueda ser hecho, las clases de análisis necesitan ser filtradas para asegurar que un número mínimo de conceptos nuevos serán creados.

El nombre de la clase de análisis debe capturar la esencia del rol jugado por la clase en el sistema. Estos nombres deben de ser únicos, y no deben existir nombres sinónimos.

Clases que definan un comportamiento similar o un mismo fenómeno deben ser unidas.

Unir también clases entidad que definan los mismos atributos, aún cuando el comportamiento definido sea distinto; agregar o unir el comportamiento en la clase.

Al actualizar cualquier clase, debe de actualizarse cualquier referencia a esa clase en los casos de uso.

Evaluar Resultados

- Verificar que las clases de análisis cubran los requerimientos funcionales del sistema.
- Verificar que las clases de análisis y sus relaciones sean consistentes con las colaboraciones que estas soportan.

Diseño de la Arquitectura.

Los principales propósitos de esta etapa:

- Analizar las interacciones de las clases de análisis para buscar interfaces, clases y subsistemas.
- Refinar la arquitectura, incorporando reusabilidad donde sea posible.
- Identificar soluciones comunes a problemas de diseño comunes encontrados.

El diseño de la arquitectura se concentra sobre la vista lógica el sistema.

Mecanismos de Diseño e Implementación.

En esta sección se definen, a partir de los mecanismos de análisis determinados en la etapa anterior, cuales mecanismos de diseño e implementación serán utilizados.

Un **mecanismo de diseño** asume algunos detalles del ambiente de implementación, pero la tecnología específica no es determinada aún.

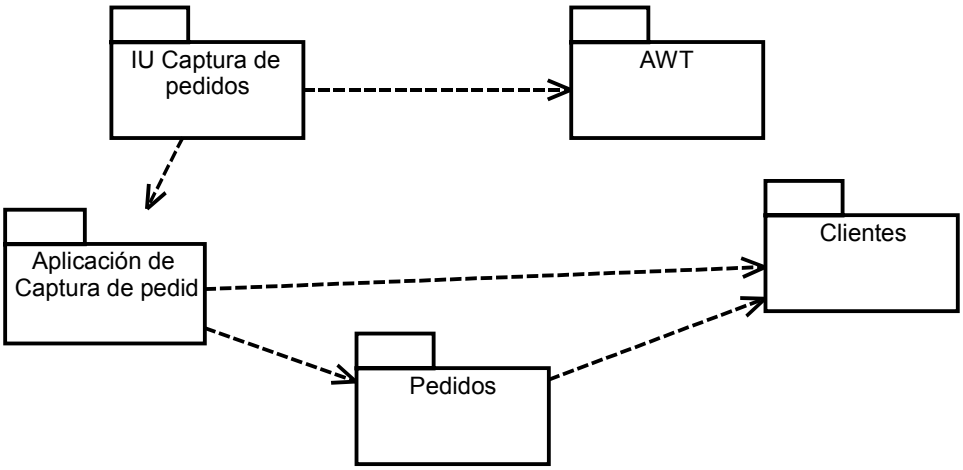
Un **mecanismo de implementación** es utilizado durante el proceso de implementación. Estos son refinamientos de los mecanismos de diseño, y especifica la exacta implementación del mecanismo.

Mecanismos de análisis (Conceptual)	Mecanismos de Diseño (Concreto)	Mecanismos de Implementación (Actual)
Persistencia	SMBDR (Datos heredados)	JDBC
Persistencia	SMBDOO (nuevos datos)	O2
Distribución	RMI	Java 1.2

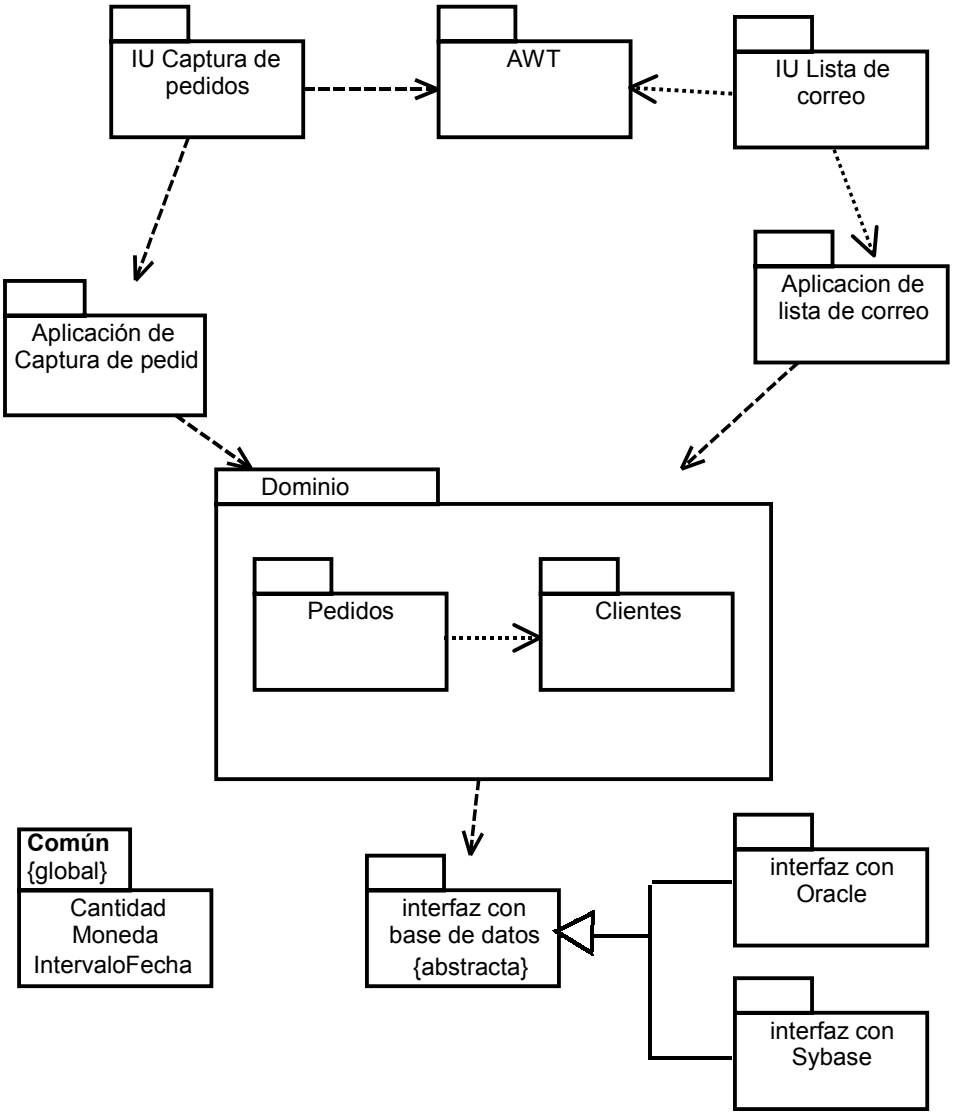
Documentando los Mecanismos de arquitectura

Los mecanismos de arquitectura que se utilizan se deben documentar con información sobre su comportamiento, estructura, clases que afecta, etc. Es común documentar a los mecanismos de análisis como patrones.

Más sobre Diagramas de Paquetes.

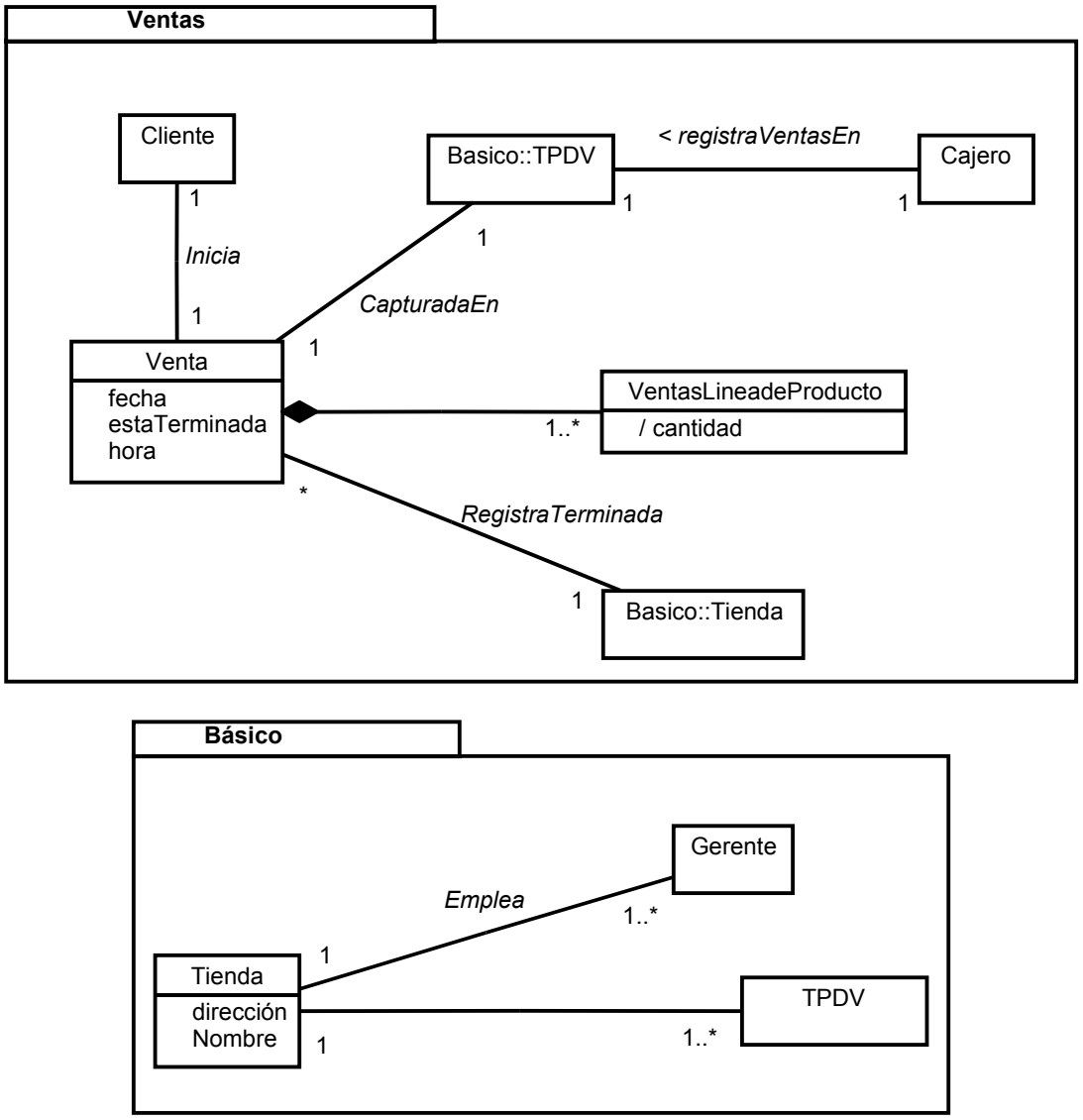


El diagrama anterior muestra los paquetes y su dependencia. Además, puede mostrarse en un diagrama de paquetes de otro nivel el contenido de un paquete, que puede ser una lista de clases, otro diagrama de paquetes o un diagrama de clases.



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.

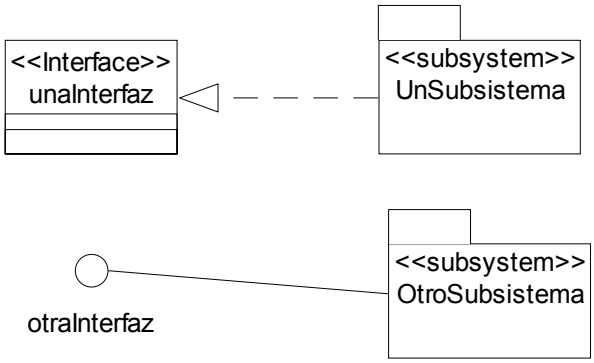
Una forma de eliminar los ciclos es utilizar la generalización de paquetes. Esta implica una dependencia de la subclase a la superclase.

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.

Subsistemas, Interfaces y Paquetes.

Un subsistema es un elemento con la semántica de un paquete, el cual puede contener otros elementos de modelado. Un subsistema realiza una o más interfaces, donde define el comportamiento que puede ser ejecutado.

Un subsistema puede ser representado como un paquete de UML, con el estereotipo <<subsistema>>



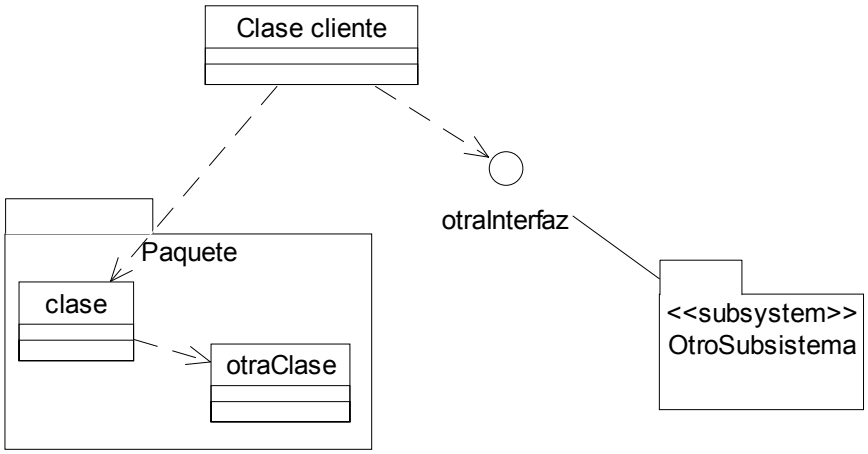
Una interfaz es un elemento del modelado que define un conjunto de comportamientos (a través de operaciones) ofrecidas por un elemento *clasificador* (una clase, subsistema o componente).

Las interfaces son la evolución natural de las clases públicas de un paquete, a abstracciones externas al subsistema. Son vistas fuera del subsistema como un tipo de "antenas", donde el subsistema puede recibir señales desde esas antenas.

Todas las clases dentro de un subsistema son privadas y no son accesibles externamente.

Paquetes o Subsistemas:

- Los subsistemas proporciona comportamiento, mientras que los paquetes no.
- Los subsistemas encapsulan completamente su contenido, los paquetes no.
- Un subsistema es fácilmente reemplazable.



Diseño de Clases y Subsistemas

Es en el Diseño de la Arquitectura donde las clases de análisis identificadas durante el Análisis de Casos de Uso son refinadas en elementos del diseño.

En esta etapa se determina que clases de análisis son realmente clases, cuales son subsistemas y cuales son componentes existentes y no necesitan se diseñadas por completo.

Las responsabilidades de las clases de análisis originales deben de ser

transferidas a los nuevos subsistemas creados. Del mismo modo, los mecanismos de diseño deben ser asociados a los elementos de diseño.

Identificando Clases de Diseño.

Una clase de análisis mapea directamente a diseño si:

- Es una clase simple
- Representa una abstracción lógica sencilla.

Clases de análisis más complejas pueden:

- Dividirse en múltiples clases.
- Convertirse en un paquete.
- Convertirse en un subsistema.

Identificando subsistemas

¿Cómo es posible identificar a los subsistemas? , buscando:

- **Colaboración de objetos**. Si las clases interactúan entre sí para producir un conjunto de resultados.
- **Interfaz de usuario**. Crear subsistemas horizontales (clases boundary y clases entidad relacionadas, ponerlas en subsistemas separados) o subsistemas verticales (clases boundary y entidad relacionadas, ponerlas en el mismo subsistema), dependiendo del acoplamiento de la interfaz con las clases entidad.
- **Actores**. Separar la funcionalidad usada por diferentes actores, donde

cada actor puede cambiar de manera independiente sus requerimientos.

- **Acoplamiento y cohesión de clases**. Organizar las clases altamente acopladas en subsistemas, separando aquellas que tengan un bajo acoplamiento.
- **Distribución**. Si determinada funcionalidad debe residir en un nodo en particular, seguramente la funcionalidad de ese nodo debe quedar en un subsistema.
- **Volatilidad**. Si se tienen áreas de un sistema que pueden cambiar, pueden ser encapsuladas en un subsistema.

Subsistemas candidatos.

Clases de análisis que pueden evolucionar en subsistemas:

- Clases con un comportamiento complejo.
- Clases boundary (interfaz de usuario o interfaz con sistemas externos)

Productos existentes o sistemas externos en el diseño:

- Software de comunicación.
- Soporte de acceso a bases de datos.
- Estructuras de datos.
- Utilerías comunes
- Aplicaciones específicas

Interfaces.

Otro paso es identificar las interfaces de los subsistemas basándose en sus responsabilidades.

Pasos recomendados:

- **Identificar interfaces candidatas**. Organizar las responsabilidades de los subsistemas en grupos de cohesión, con responsabilidades relacionadas.
- **Buscar parecidos en interfaces**. Buscar nombres similares, responsabilidades y operaciones. Extraer esas operaciones similares para una nueva interfaz.
- **Defina las dependencias de la interfaz**. Las relaciones de la interfaz a las clases y/o interfaces deben añadirse.
- **Definir comportamiento para las interfaces**. Si las operaciones de la interfaz deben ser invocadas en un cierto orden, se define una máquina de estados que ilustre los estados visibles (o inferidos) que cualquier elemento tenga que realizar para soportar la interfaz.
- **Empaquete las interfaces**. Las interfaces pueden ser manejadas independientemente de sus subsistemas. Organizar las interfaces si se considera necesario.

Una interfaz se estructura de acuerdo a lo que se espera de esta:

Nombre. Reflejando el rol en el sistema.

Descripción. De acuerdo a sus responsabilidades.

Definición de operaciones. Describir la operación, parámetros y resultado.

Documentación. Diagramas de secuencia y de estados, planes de prueba.

En esta etapa se genera básicamente:

- Un diagrama de clases de contexto del subsistema, el cual contiene al subsistema, sus interfaces, relaciones de realización, y cualquier relación del subsistema con otros elementos del diseño. Puede representarse en lugar del subsistema una clase proxy que implemente el comportamiento de la interfaz, se representa como una clase con el estereotipo <<*proxy* subsistema>>.
- Se construye una lista del mapeo de clases de análisis a elementos del diseño.

Identificar Oportunidades de Reuso

El objetivo es identificar si los subsistemas y/o componentes pueden ser reusados basándose en sus interfaces.

Pasos:

- Buscar interfaces similares.
- Modificar las nuevas interfaces para mejorar el ajuste.
- Reemplazar interfaces candidatas con interfaces existentes.
- Mapear el subsistema candidato al componente existente.

El reuso puede ser de dos formas:

- **Interno**. Reconociendo elementos comunes o cruzados en paquetes y subsistemas.
- **Externo**. Componentes comerciales disponibles, componentes desarrollados previamente.

Diseño de Casos de Uso

El objetivo de esta etapa es refinar las realizaciones de los casos de uso usando los elemento del modelo de diseño, verificando que los elementos sean consistentes para su implementación.

Adicionalmente, cualquier mecanismo de arquitectura aplicable, deberá ser incorporado en las realizaciones de los casos de uso.

Como base para esta etapa se tiene:

- Especificaciones suplementarias.
- Casos de uso.
- Realizaciones de los Casos de Uso.
- Clases de Diseño
- Subsistemas y paquetes de Diseño
- Interfaces.

Como resultado se obtiene una versión refinada de las realizaciones de los casos de uso.

Describir interacciones entre objetos de diseño.

En este paso se reemplazan las clases de análisis de los casos de uso con los elementos de diseño refinados durante el Diseño de la Arquitectura, así como la incorporación de cualquier mecanismo de arquitectura aplicable.

Refinamiento de Realizaciones de Casos de Uso.

Para cada realización de caso de uso:

- Identificar objetos participantes. Estos pueden ser instancias de clases o subsistemas.
- Representar cada objeto participante en los diagramas de interacción. Los subsistemas pueden ser representados como instancias de interfaces de subsistemas.
- Ajustar el envío de mensajes entre objetos, de acuerdo a las modificaciones de los objetos en el diagrama. Un mensaje a una clase de diseño es una operación de clase. Un mensaje a un subsistema, es una operación de interfaz.
- Para cada realización de casos de uso, representar las relaciones de clases que soportan las colaboraciones modeladas en los diagramas de interacción.

Describir comportamiento de persistencia relacionado

La persistencia necesita tener en cuenta varios aspectos, que deben reflejarse en los casos de uso de diseño:

- **Modelado de Transacciones.** Una transacción es un conjunto de instrucciones que se considera atómica. O se realiza todo o nada del conjunto de operaciones de la transacción. Genera consistencia en la base de datos.

  La transacción puede ser modelada textualmente o mediante **mensajes explícitos** en los diagramas de interacción.

Hay que modelar también las posibles condiciones de error. Esto puede requerir diagramas de interacción separados. Deben incluirse las posibles fallas, incluyendo el *rollback*.

- **Escritura de objetos.** Se consideran dos casos, cuando el objeto es grabado por primera vez y posteriormente, cuando el objeto es actualizado.

- **Lectura de objetos.** La recuperación de un objeto de la base de datos implica enviar un mensaje a un objeto que puede consultar la base de datos, recuperar el objeto correcto e instanciarlo (si no lo esta).

- **Borrado de objetos.** Cuando un objeto recibe una orden de borrado permanente, este debe de eliminarse de la base de datos.

Refinar la descripción del flujo de eventos

Se deben de revisar los flujos de eventos de los diagramas buscando aquellas partes en que no sea muy claro el envío de mensajes entre objetos, y considerar añadirle descripciones adicionales.

Estas descripciones pueden incluir anotaciones tipo *script*, notas sobre comportamientos condicionales, o clarificaciones del comportamiento de una operación. El objetivo es lograr que observadores externos puedan comprender los diagramas.

Unificar clases, paquetes y subsistemas.

El propósito de este paso es asegurar que cada elemento del diseño represente un concepto bien definido, sin invadir responsabilidades entre sí.

- Nombres de elementos del modelo deben describir su función.
- Unir elementos que representen un comportamiento similar o que representen el mismo fenómeno.
- Unir clases que representen el mismo concepto o que tengan los mismos atributos, incluso si el comportamiento es diferente.
- Usar herencia para abstraer elementos del modelo.
- Cuando se modifica un elemento, actualizar todas las realizaciones de casos de uso involucradas.

Se recomienda construir un diagrama único mostrando todos los elementos del diseño y sus relaciones. (organizado por capas)

Diseño de los Subsistemas.

Los principales objetivos:

- Definir el comportamiento específico en las interfaces de los subsistemas en términos de colaboración de las clases contenidas.
- Modelar la estructura interna de los subsistemas.
- Determinar las dependencias de los elementos externos al subsistema

Distribuir el comportamiento del subsistema a los elementos del subsistema.

En este paso se pretende especificar el comportamiento interno del subsistema, e identificar nuevas clases o subsistemas necesarios para satisfacer los requerimientos de comportamiento.

Las responsabilidades del subsistema están definidas por las operaciones de interfaz. Estas operaciones pueden ser realizadas por:

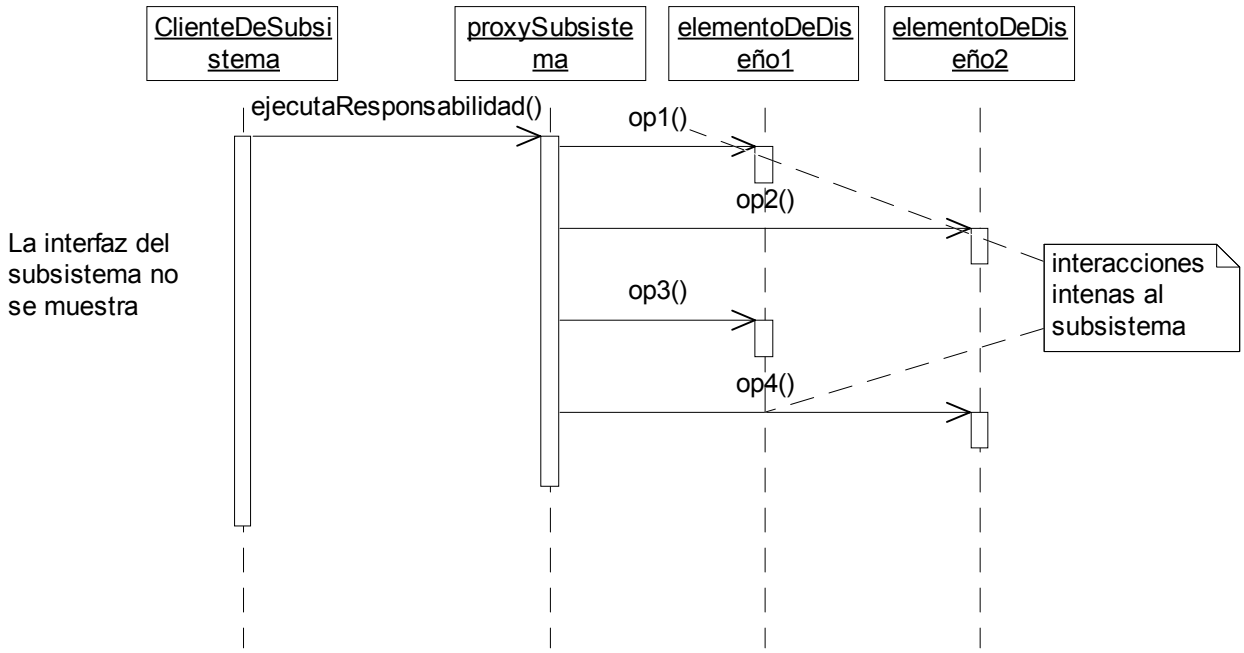
- operaciones de clases internas
- operaciones de subsistemas externos.

La distribución de responsabilidades del subsistema implica:

- Identifica elementos del diseño nuevos o existentes, tales como clases y/o subsistemas).
- Asignar las responsabilidades del subsistema a los elementos del diseño.

- Incorporar los mecanismos de arquitectura aplicables.
- Documentar elementos de diseño en “realizaciones de interfaz”:
    - Uno o más diagramas de interacción por operación de interfaz.
    - Diagrama de clases conteniendo los elementos de diseño requeridos y sus relaciones
- Revisar el Diseño de la Arquitectura. Ajustar los limites de los subsistemas y sus dependencias si es necesario (diagramas de contexto de los subsistemas)

Un esquema general de los diagramas de interacción de los subsistemas se muestra a continuación:



Modelar los elementos del subsistema.

Hasta este punto, las responsabilidades han sido asignadas a loe elementos de los subsistemas y se han construido diagramas de interacción de comportamiento interno del subsistema.

En este punto se debe modelar la estructura estática del subsistema. La estructura interna es dirigida por las colaboraciones necesarias entre los objetos para implementar las interfaces del subsistema.

Se crean uno o más diagramas de clases que muestren los elementos que contiene el subsistema y sus relaciones. Un diagrama puede ser suficiente, pero pueden crearse más para reducir la complejidad y mejorar la legibilidad.

Si existe dependencia hacia o desde algún otro subsistema se puede representar a través de la interfaz o la clase *proxy*.

Adicionalmente, puede ser necesario diagrama de estados para documentar los posibles estados que puede asumir un subsistema

Es importante documentar cualquier dependencia de orden entre las operaciones de las interfaces (si una operación necesita antes de la otra), y esto se logra precisamente con un diagrama de estados.

Describir las dependencias del subsistema.

En este punto se documentan mediante un diagrama de paquetes los elementos externos relacionados con el subsistema, algunas de estas dependencias se pudieron haber generado en el paso anterior y deben ser representadas aquí.

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 o subsistema
- Diagramas de estado de los objetos.

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.

Ejemplos:

Levantar el auricular de un teléfono.

El avión despega.

El disco duro falla.

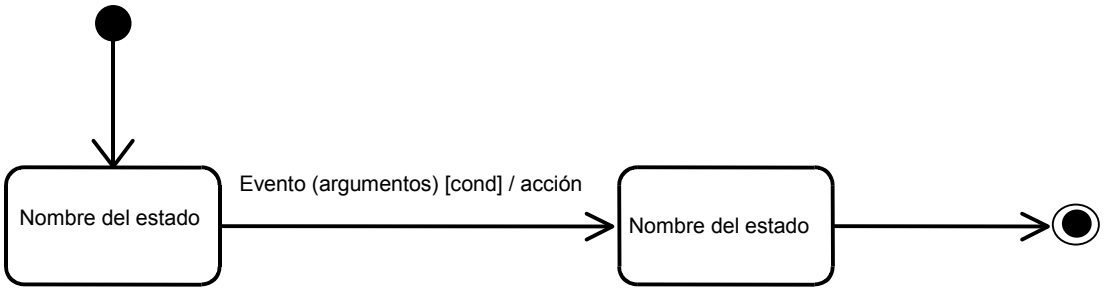
- Un evento es un estímulo de un objeto a otro.
- Puede ser una señal de que un acontecimiento ha ocurrido.
- Un evento separa dos estados.
- La respuesta a un evento depende del estado del objeto que lo recibe.
- Los eventos representan puntos instantáneos en el tiempo.
- Los eventos se derivan de las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

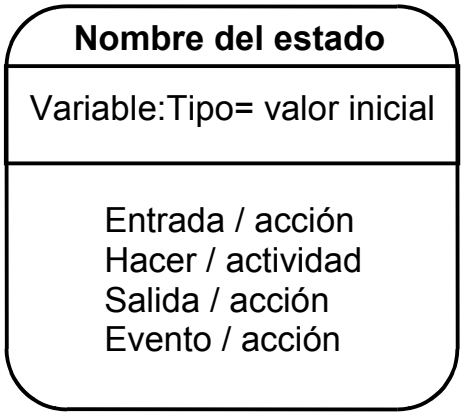
Ejemplo: El evento *levantar el auricular* puede pasar al teléfono de un estado *ocioso* a un estado *activo*.

- Una transición se dispara cuando un evento ocurre.

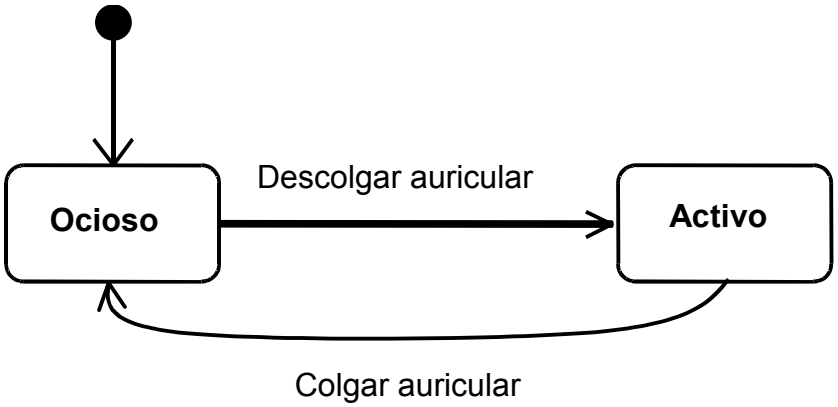
Notación en UML



Representación completa de un estado:



Ejemplo: diagrama de estados para un teléfono:

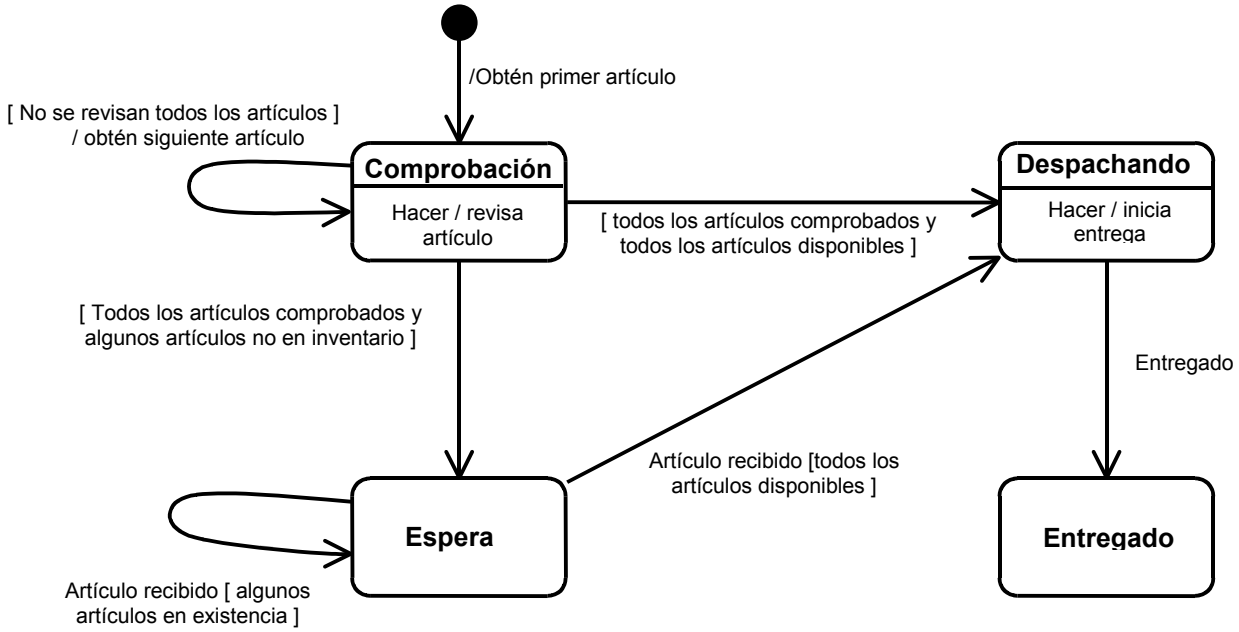


Como en la mayoría de los diagramas de UML, el nivel de detalle puede variar de acuerdo a la información que se quiera representar.

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.

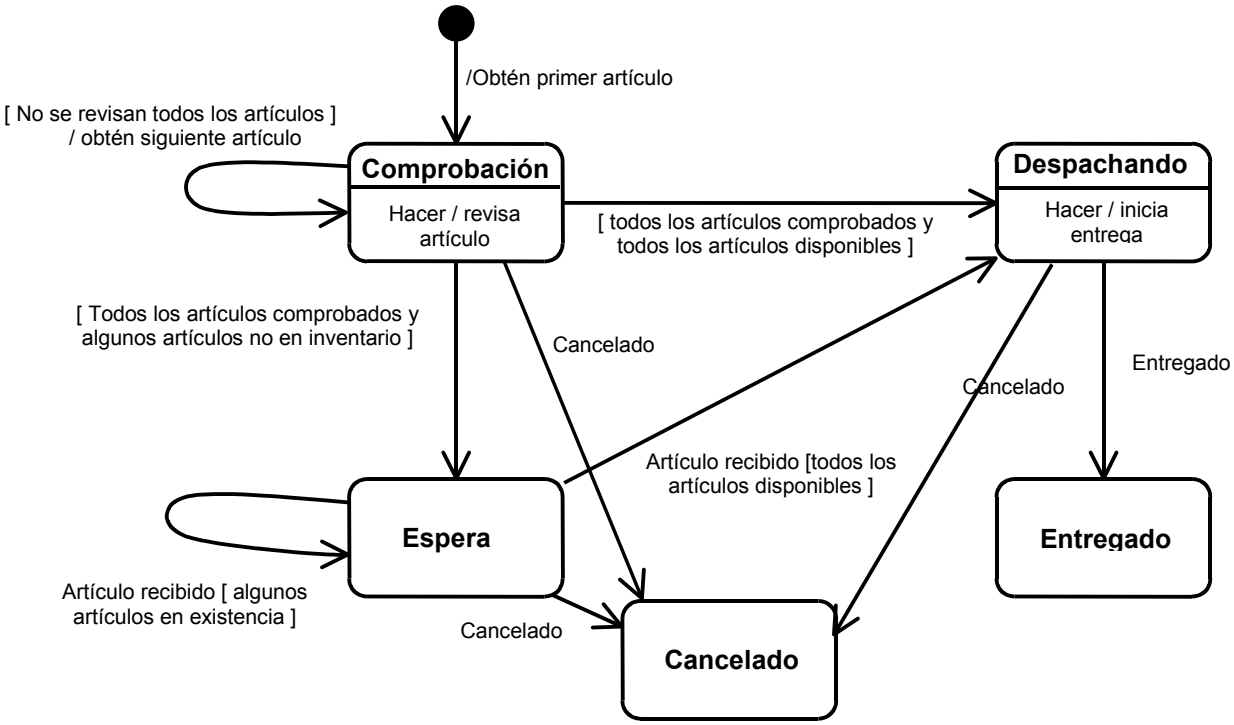
Autotransición. Una transición que vuelve al mismo estado. ¿Qué sucede si ese estado tiene eventos de entrada y/o salida?

Superestado.

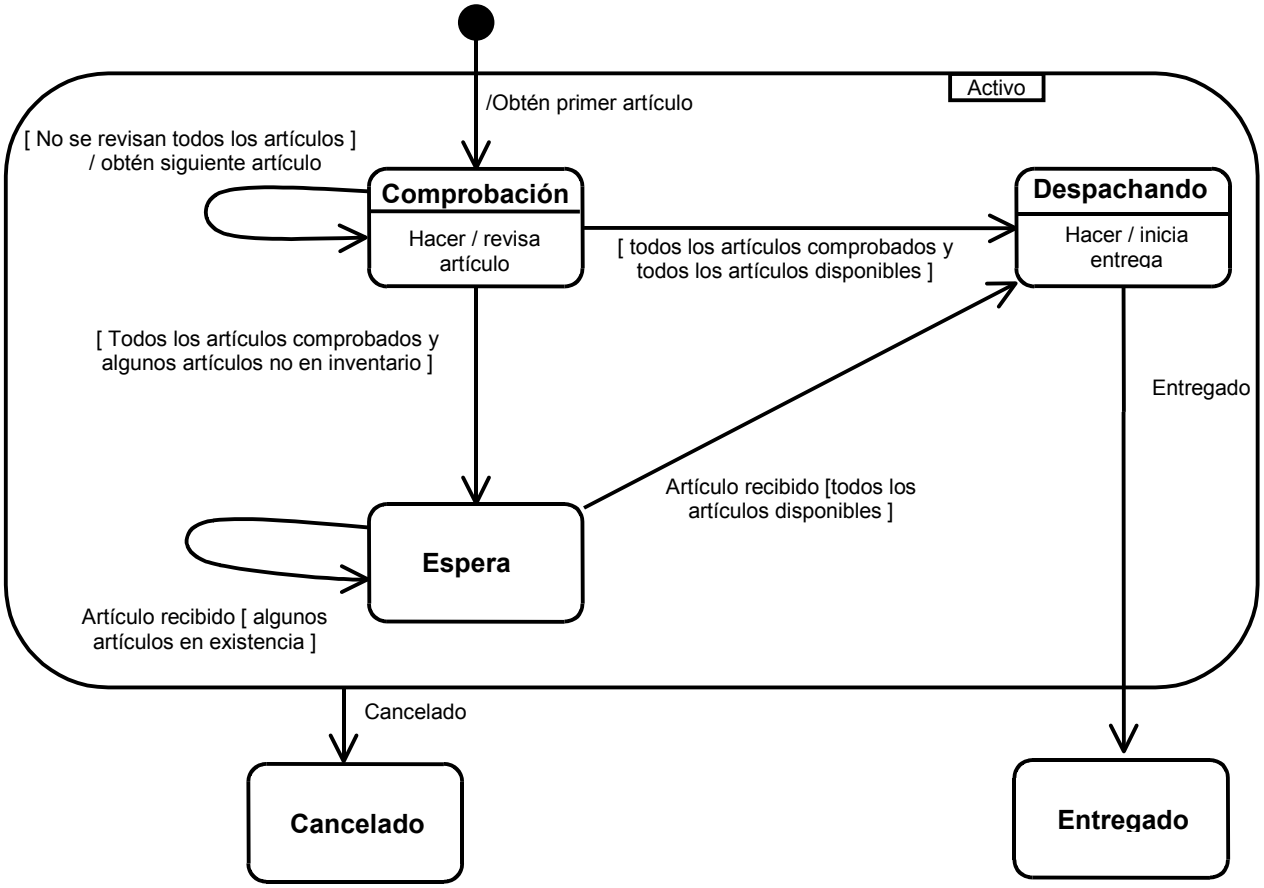
Es posible crear un superestado que contenga a un conjunto de subestados. Los subestados heredan todas las transiciones sobre el superestado.

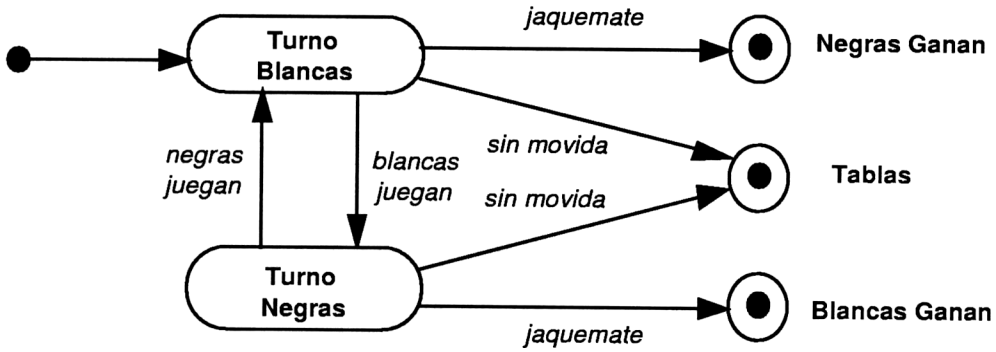
Un superestado nos ayuda a reducir la saturación de transiciones entre estados.

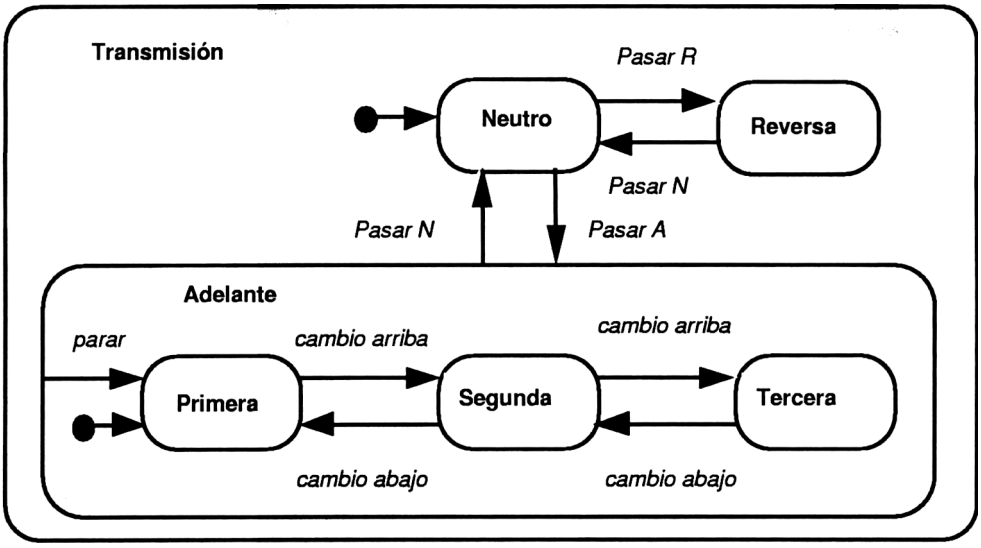
Considere el ejemplo anterior en el que fuera necesario cancelar el pedido desde cualquier punto antes de que sea entregado.



Representado con un superestado:



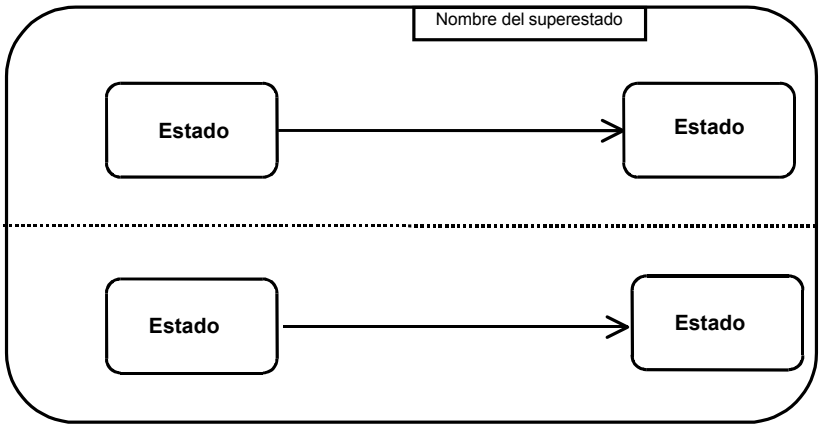




Diagramas de estados concurrentes.

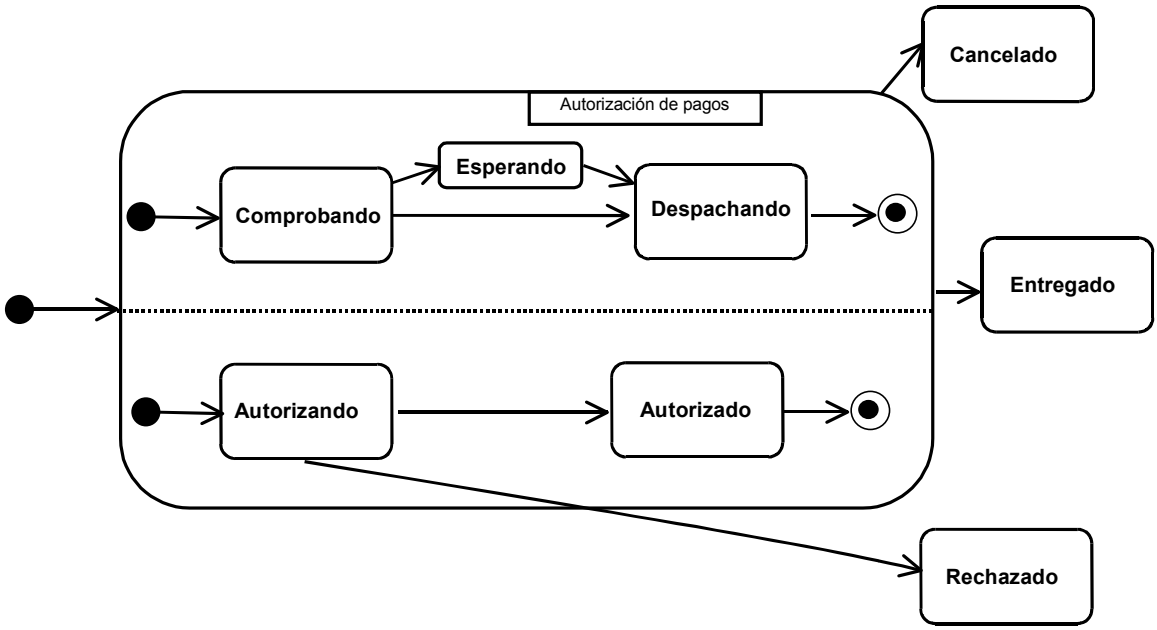
Es posible que sea necesario modelar diagramas de estados concurrentes. Estos son útiles cuando un objeto dado tiene conjuntos de comportamientos independientes.

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:



Clases orientadas al diseño.

Las 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.

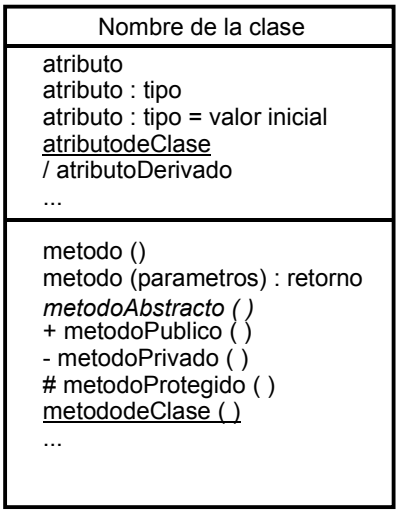
La intención de las clases es describir gráficamente las especificaciones de las clases de software y de las interfaces - como en Java - en una aplicación.

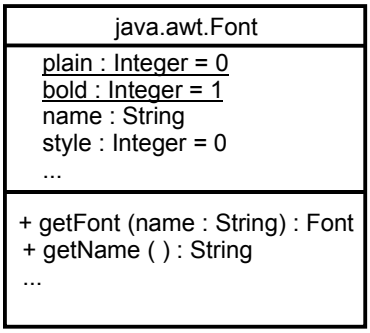
Normalmente contiene la siguiente información:

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

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.





Diseño de Clases

En esta tapa se pretende:

- Identificar clases y relaciones adicionales necesarias para soportar los mecanismos de arquitectura elegidos.
- Identificar y analizar transiciones de estado en objetos de clases "controladas por estados"
- Refinar asociaciones, operaciones y atributos.

Creación inicial de clases de diseño

Se toman las clases de los subsistemas y se propone la creación de clases adicionales que se consideren necesarias para complementar su tarea.

Al identificar una nueva clase, bastará por el momento con encontrar las relaciones iniciales de esta clase con la estructura de clases existente.

Tener en cuenta los mecanismos de arquitectura y los estereotipos de las clases.

Identificar clases persistentes.

En esta parte se especificara completamente el comportamiento de almacenamiento de las clases persistentes. Se describen las clases que intervienen en el manejo de la persistencia, dependiendo de la tecnología elegida.

En este punto también sería importante la coordinación con el diseñador de la base de datos.

Definir operaciones.

En este punto, se mapean las responsabilidades definidas en el análisis a las operaciones que serán implementadas.

Cosas a considerar:

- Nombre de la operación, firma y descripción.
- Visibilidad de la operación.
- Alcance de la operación: operación de clase o de instancia.

Definir visibilidad de clases.

Es importante especificar el tipo de visibilidad sobre las clases contenidas en paquetes. La visibilidad puede ser:

- Pública. La clase puede ser accesada fuera del paquete.
- Privada. La clase sólo puede ser accesada por clases del mismo paquete.

Definir métodos.

Un método describe la implementación de una operación.

En algunos casos la información de la operación no es suficiente para definir como esta será implementada. En esos casos la descripción del método es necesaria.

La descripción del método clarifica cualquier algoritmo especial que será usado, así como los atributos y relaciones que serán implementadas.

Se debe incluir cualquier otro objeto y operación que se ocupe. Puede

modelarse mediante diagramas de interacción.

Definir estados.

Mediante diagramas de estados, modelar el comportamiento de las clases. Este diagrama debe hacerse únicamente para aquellas clases que ofrezcan un comportamiento de estados importante y debe de tener una relación directa con las operaciones de la clase.

Definir atributos.

En este punto se formaliza la definición de los atributos. Se debe considerar:

- Persistencia
- Visibilidad
- Alcance
- Nombre, tipo y valor inicial.

Definir dependencias.

Una dependencia es un tipo de relación entre dos objetos y ayuda a determinar que relaciones estructurales no son requeridas.

- Las asociaciones son relaciones estructurales.
- Las dependencias son relaciones no estructurales.
- La visibilidad determina la relación:

- Visibilidad global, de parámetro y local implican una relación no estructural.
- Visibilidad de atributo implica una relación estructural.

Definir asociaciones.

Se refinan las relaciones de asociación teniendo en cuenta:

- Asociación ó Agregación
- Agregación ó Composición Navegabilidad
- Asociación de clase de diseño
- Diseño de multiplicidad.

Definir generalizaciones

El propósito de este punto es identificar áreas de reuso y refinar la jerarquía de herencia existente para que pueda ser implementada eficientemente. Revisar y/o refinar:

- Clases abstractas o clases concretas
- Problemas de herencia múltiple
- Generalización ó agregación
- Generalización para soportar implementación de reuso
- Generalización para soportar polimorfismo

Resolver colisiones de Casos de Uso

Múltiples Casos de Uso pueden tener acceso simultáneo a objetos. Las opciones comunes son:

- Usar sincronización de mensajes.
- Identificar operaciones para proteger.
- Aplicar mecanismos de control de acceso: cola de mensajes, semáforos u otro mecanismos de bloqueo.

La solución es dependiente del ambiente de implementación.

Manejo de requerimientos no funcionales en general.

En este paso, las clases pueden ser refinadas para incorporar requerimientos no funcionales que no se hayan ya incorporado de algunos mecanismos de diseño. Por ejemplo:

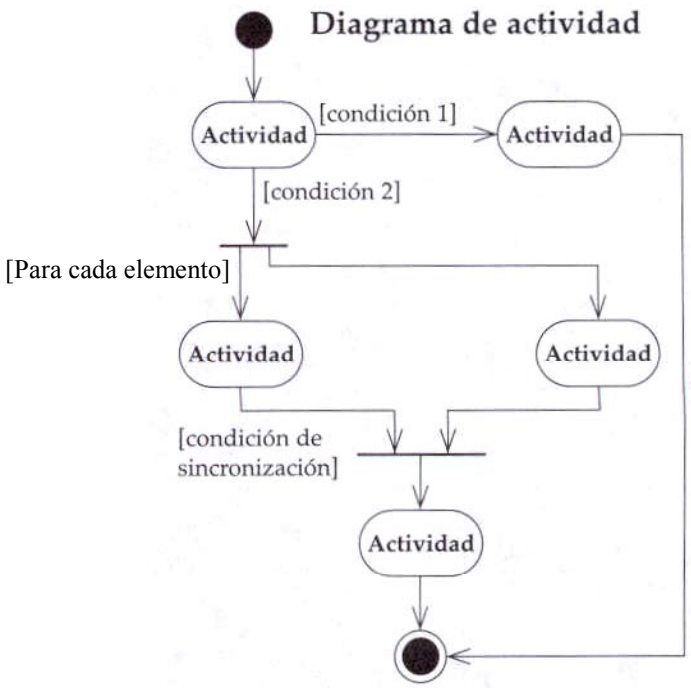
- Adaptaciones al lenguaje de programación.
- Cumplir con un rendimiento aceptable.
- Cumplir con niveles de seguridad.
- Manejo de errores.
- etc.

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.

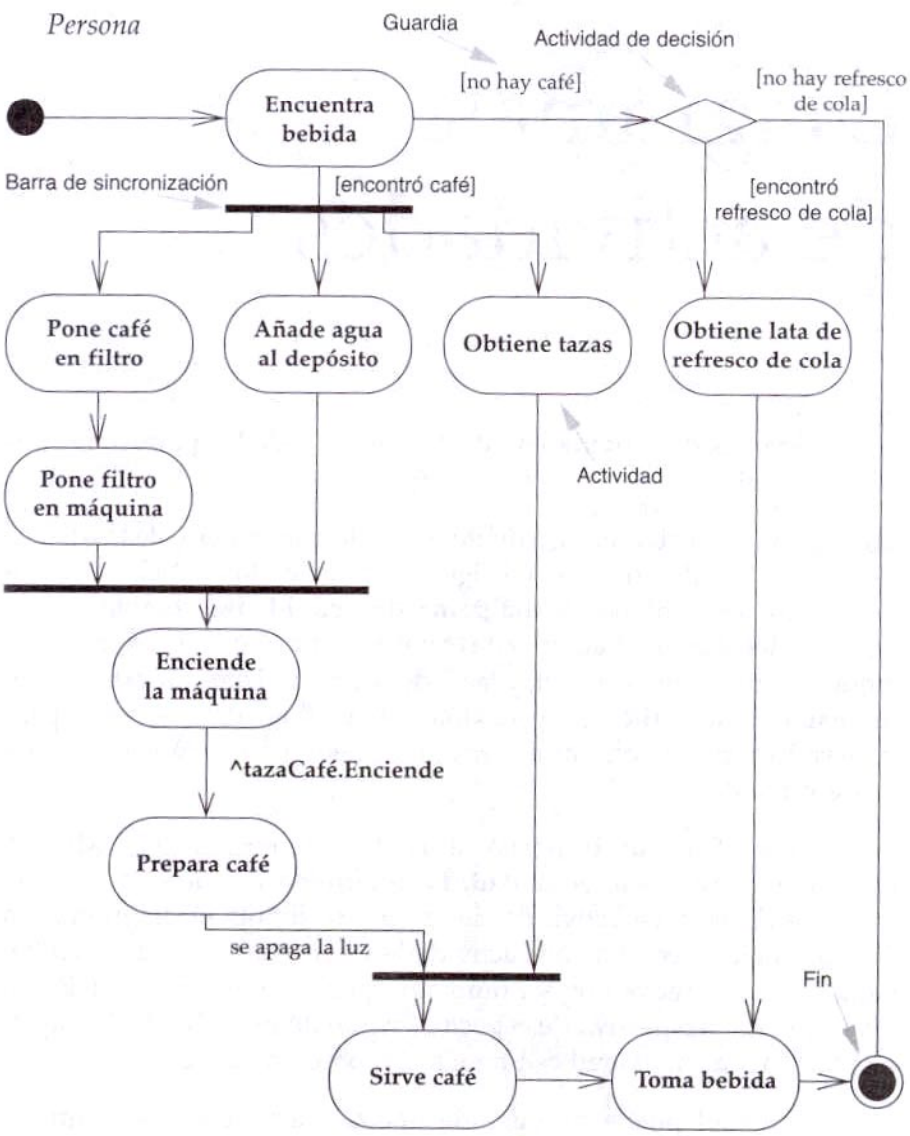
La representación en UML es de la siguiente forma:



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 diseño o implementación, una actividad es una operación 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.



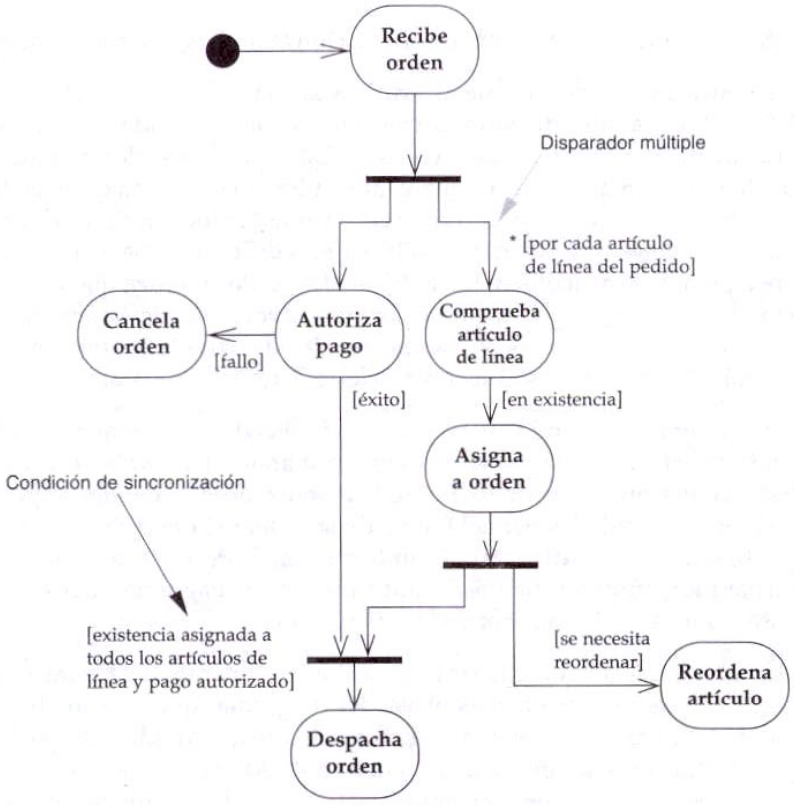
Es evidente que estos diagramas se prestan para modelar programas concurrentes, ya que pueden plantear gráficamente cuáles son los hilos y cuándo necesitan sincronizarse.

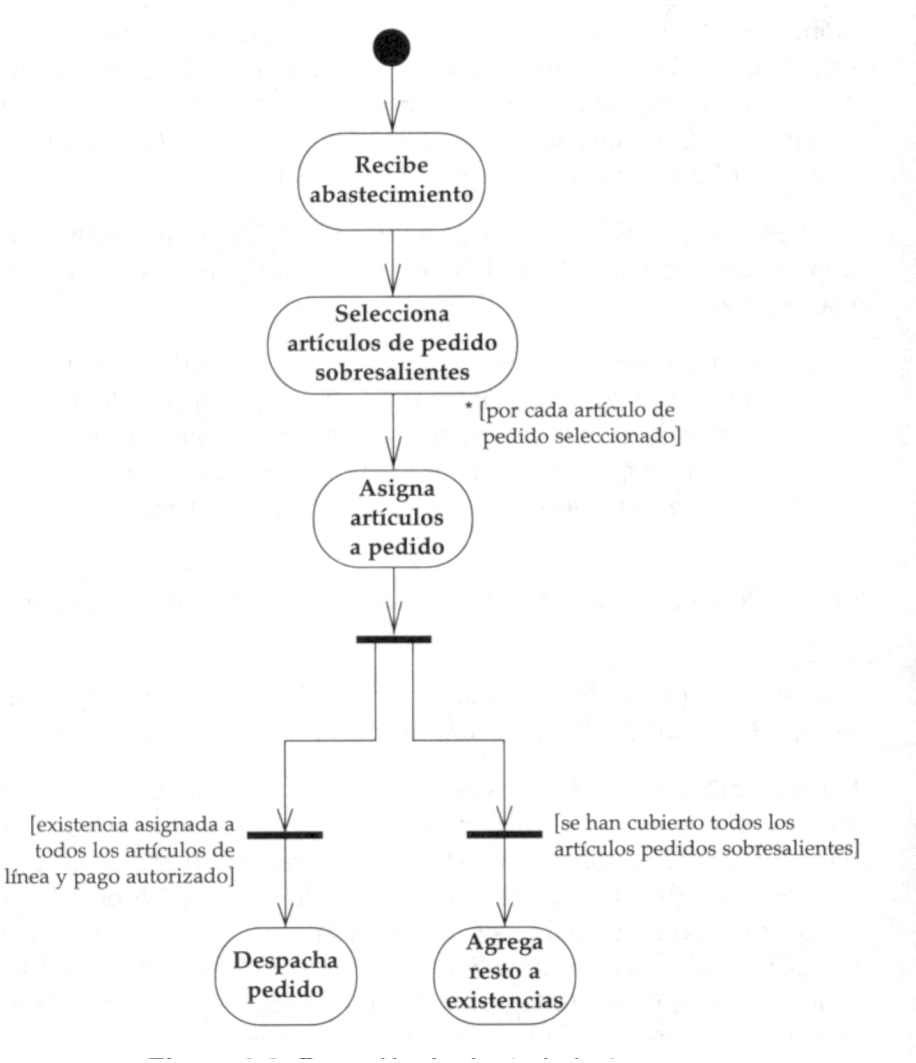
En los procesos concurrentes es necesario especificar las sincronizaciones; para lograr esto, el diagrama de actividades proporciona la barra de sincronización. Una barra de sincronización indica que el *disparo* de salida ocurre sólo cuando se han llevado a cabo todos los *disparos* de entrada.

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.

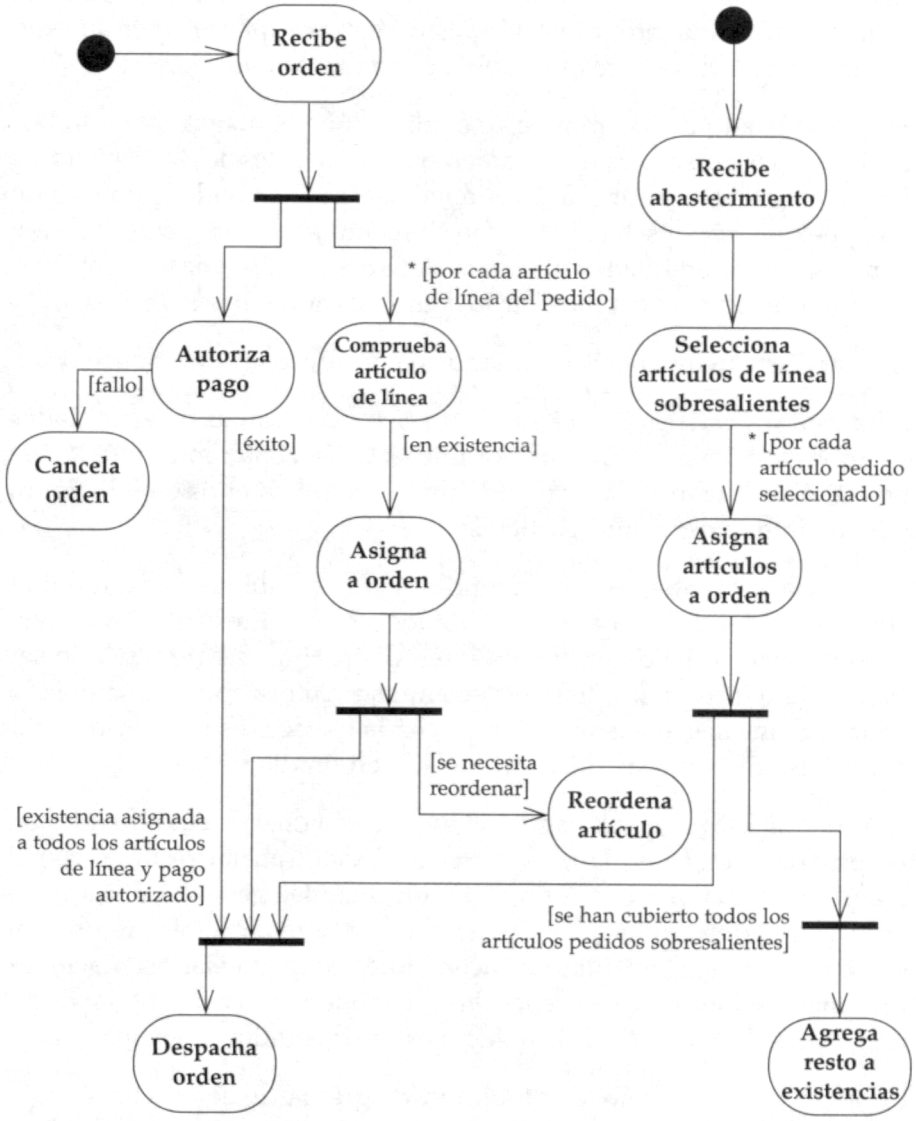
Los diagramas de actividades normalmente se usan para describir métodos complejos o casos de uso potencialmente concurrentes.


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.

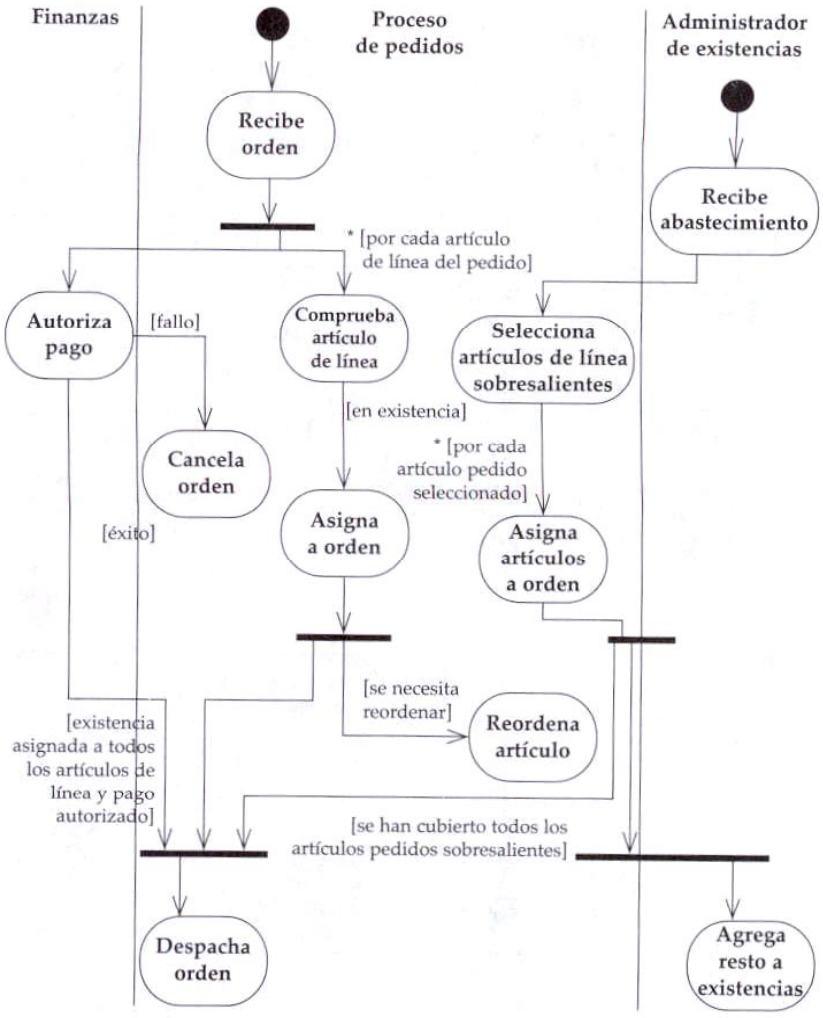


Carriles.

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.



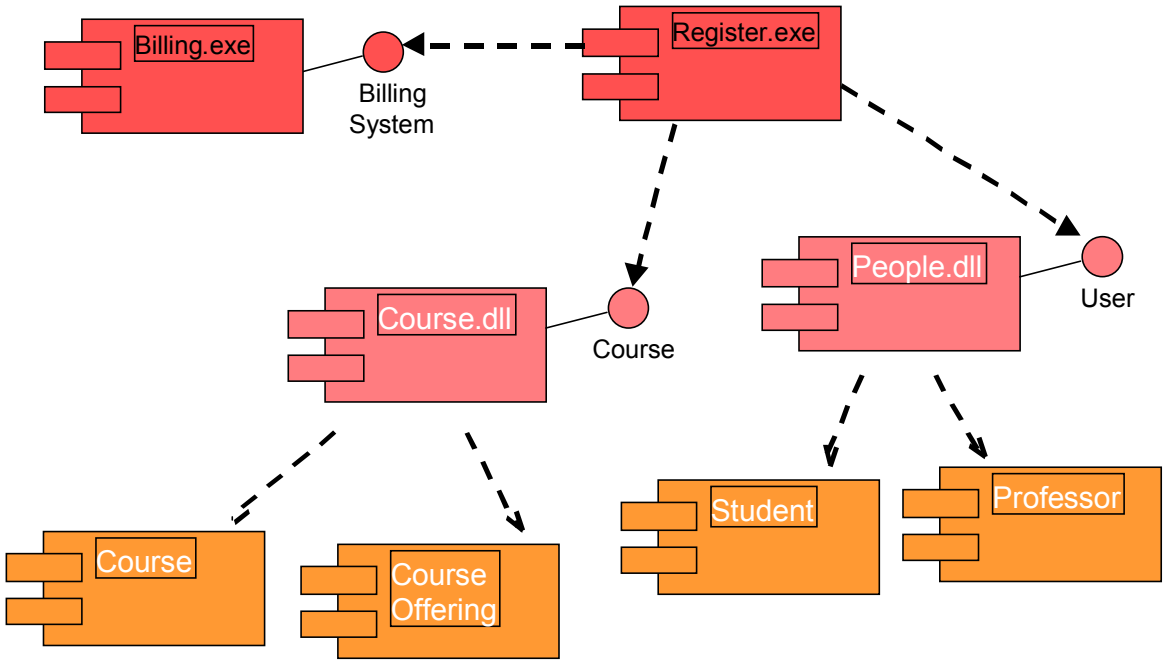
El objetivo de los carriles es combinar la representación lógica de los diagramas de actividades con la representación de las responsabilidades de los diagrama de interacción.

- 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.

Diagramas de componentes y de despliegue.

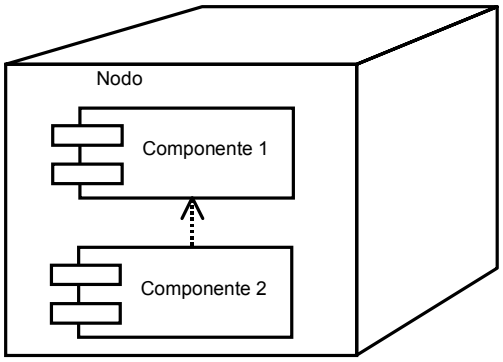
El diagrama de componentes representa las estructura física de la implementación. Forma parte de la arquitectura de la especificación, y ayuda a proponer:

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



Por otro lado, un diagrama de despliegue o 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 despliegue muestra las relaciones físicas entre los componentes de software y de hardware del sistema.

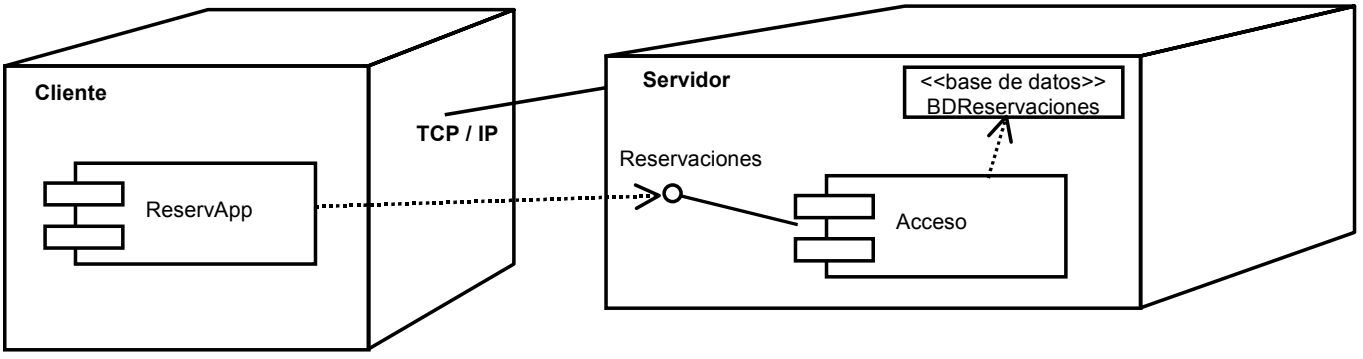


Cada nodo en un diagrama de despliegue representa algún tipo de unidad de cómputo.

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 despliegue 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.



Bibliografía.

- Fowler, Martin. **UML, gota a gota (UML Distilled)**. Addison Wesley. México, 1999.
- Larman, Craig. **UML y Patrones. Introducción al análisis y diseño orientado a objetos (Applying UML and Paterns) .** Prentice Hall. México, 1999.
- **Inside the UML.** Rational Software. U.S.A. 1999.
- Grady **Booch**, Ivar Jacobson, James Rumbaugh. **The Unified Modeling Language User Guide.** Edit. Addison-Wesley. 1998.
- Ivar **Jacobson**, Grady Booch, James Rumbaugh. **The Unified Software Development Process.** Edit. Addison-Wesley. 1999
- Philippe Kruchten. **The Rational Unified Process. An Introduction.** 2a edición. Ed. Addison-Wesley. 2000.

Complementaria

- Rumbaugh, James, y otros, **"Modelado y diseño Orientado a Objetos"**, Prentice Hall, España, 1ª edición. 1996.
- McConnell, Steve. **"Desarrollo y gestión de proyectos informáticos"**. Mc Graw Hill/Microsoft Press. 1ª edición. España. 1997.
- Pressman. Roger. **"Ingeniería de Software. Un enfoque práctico"**. Mc Graw Hill. 3ª edición. 1993.
- Weitzenfeld, Alfredo. **Paradigma Orientado a Objetos.** ITAM. México. 1994.

- Apuntes del curso de "Análisis y Diseño Orientado a Objetos, OMT". Impartido por la Fundación Arturo Rosenblueth. 1997.
- Pierre-Alain Muller. **Modelado de Objetos con UML.** Edit. Gestión 200. España, 1997.
- James **Rumbaugh**, Ivar Jacobson, Grady Booch. **The Unified Modeling Language Reference Manual.** Edit. Addison-Wesley. 1998.

Artículos

1. Arredondo, Eduardo. **La Crisis del Software.** Www.infosistemas.com.mx/arredo12.
2. Booch, Grady. **The Visual Modeling of Software Architecture for the Enterprise.** Rose Architect Magazine. 1999. http://www.rosearchitect.com
3. Sumano López, María de los Ángeles. **Aspectos Humanos de la Ingeniería de Requerimientos. El Caso de México.** ENC. 1999.
4. Aguilar Sierra, Alejandro. **Aplicación de los Patrones de Diseño y el Proceso Unificado al Diseño de un Editor Matemático.** ENC . 1999.
5. Booch, Grady. **The Importance of Teams.** Rose Architect Magazine.1999. http://www.rosearchitect.com
6. Beck; Cunningham . A Laboratory For Teaching Object-Oriented Thinking. En OOPSLA 89. http://c2.com/doc/oopsla89/paper.html
7. Flower, Martin. **Why Use the UML?.** Software Development magazine. 1998.