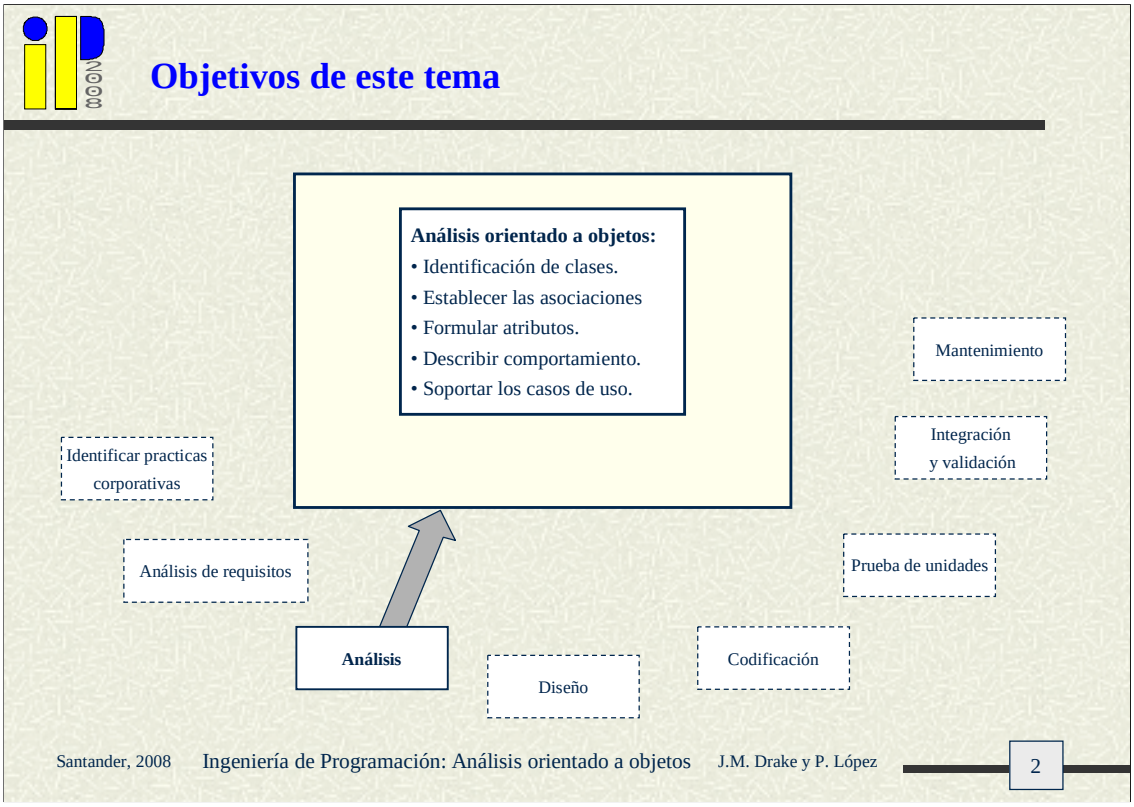
Análisis orientado a objetos

José M. Drake y Patricia López
Computadores y Tiempo Real

Santander, 2008

1



La finalidad del análisis orientado a objetos de una aplicación es establecer su modelo de objetos, que captura la estructura estática del sistema a través de identificar los objetos claves que se utilizan el sistema, formular las relaciones entre los objetos, y caracterizar cada clase de objetos a través de la asignación de los atributos que describen sus características y estado y las operaciones que describen su funcionamiento externo.
El modelo de objetos es el mas importante de una aplicación. En la metodología orientada a objetos se enfatiza el estudio de los objetos en vez de la funcionalidad (análisis estructurado), porque constituye un descripción mas próxima a la realidad y facilita los cambios y adaptaciones posteriores.
El modelo de objetos proporciona una representación gráfica intuitiva que constituye una descripción de la naturaleza de la aplicación y puede ser utilizada como base de comunicación entre diseñadores y de estos con los usuarios, así mismo es la base de la documentación de la estructura del sistema.

Diseño modular del software

Las **herramientas** básicas para **abordar la complejidad** son:

- **Modularización:** Capacidad para descomponer los componentes complejos en otros mas simples.
- **Abstracción:** Capacidad de reducir la información de un componente a la necesaria para manejarlo en un nivel de desarrollo.
- **Herencia:** Capacidad de jerarquizar los componentes del dominio de acuerdo con las características comunes que presentan.

Son las bases de la estrategia que da lugar a la **metodología orientada a objetos** de desarrollo de software .

Las dos herramientas básicas que utiliza el hombre para abordar sistemas muy complejos es la modularización y la abstracción.
La modularización consiste en descomponer un componente complejo en un conjunto reducido de subcomponentes mas sencillos. La estrategia de modularización debe iterarse hasta que la complejidad de los módulos que resultan abordables por el programador.
La abstracción es la capacidad de reducir la información que describe un componente a la justa necesaria para manejarlo dentro de la fase de desarrollo en que se está.
La herencia es la capacidad de agrupar jerárquicamente los componentes de forma que aquellas características que tengan en común muchos de ellos solo se necesite describir una vez a través de la descripción del correspondiente antecesor común.

Criterios de modularización.

- **Descomponibilidad modular**: Permita descomponer sucesivamente cada módulo en otros mas simples que puedan ser abordados.
- **Componibilidad modular**: Genere módulos que puedan ser libremente combinados para generar módulos mas complejos.
- **Comprensibilidad modular**: Cada módulo que se genera puede ser descrito y comprendido por sí y con independencia de otros.
- **Continuidad modular**: La descomposición debe ser tal que pequeñas modificaciones de la especificación del problema introduzca cambios en poco módulos y en proporción de las modificaciones introducidas.
- **Protección modular**: Los errores que se produzcan en un módulo queden confinados y puedan tratarse en él.

Santander, 2008 Ingeniería de Programación: Análisis orientado a objetos J.M. Drake y P. López 4

Criterios que debe satisfacer un método de diseño para ser modular:

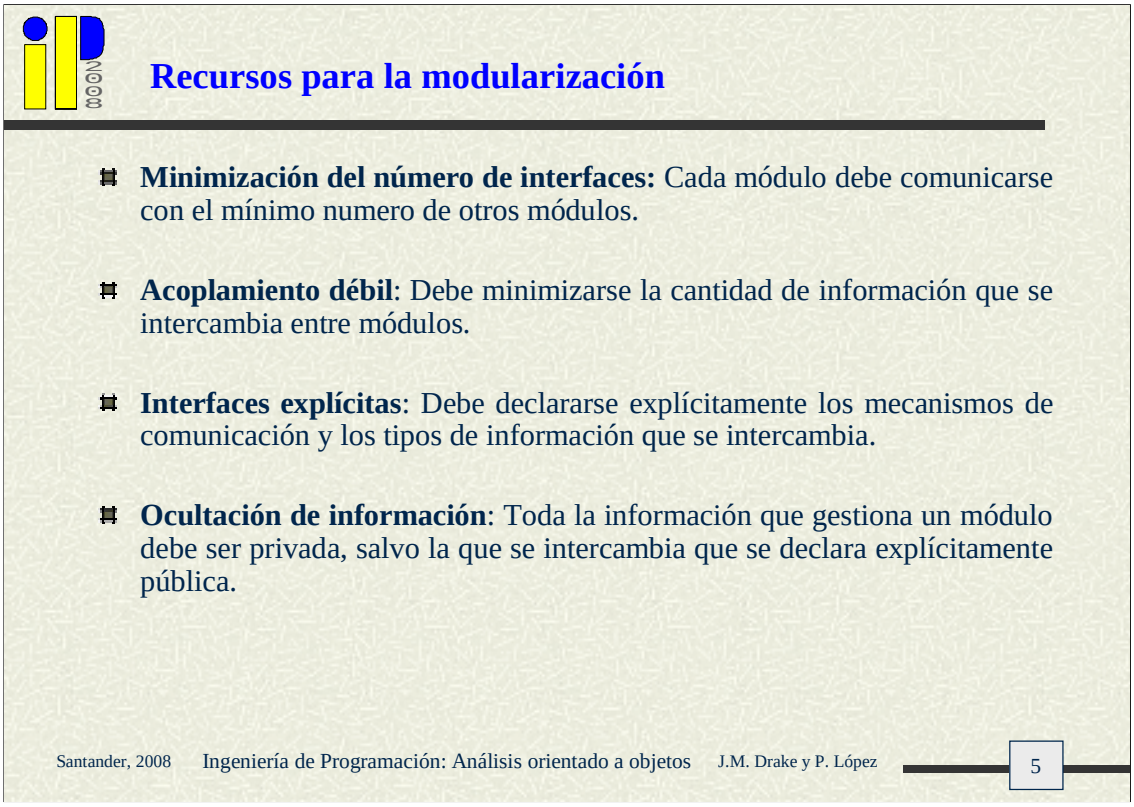
Descomponibilidad modular: El criterio requiere, que los módulos resultantes sean consistentes por sí y puedan ser tratados y manejados de forma independiente por diferentes diseñadores. El método de diseño top-down es el mas utilizado. El diseñador comienza con la descripción mas general del sistema, y luego la refina paso a paso, descomponiéndolo en nuevos subsistemas de complejidad mas simple.

Componibilidad modular: Este criterio tiene como finalidad la reusabilidad. Se busca que el trabajo desarrollado en un proyecto pueda ser aprovechado en los siguientes proyectos. La componibilidad y descomponibilidad son aspectos independientes y, a veces, estan enfrentados. Por ejemplo: Los módulos generados utilizando la estrategia top-down y obtenidos para una aplicación específica suelen ser no componibles, ni reusables.

Compresibilidad modular: Este criterio es esencial para las fases de mantenimiento. En un sistema complejo, se necesita poder comprender, analizar y modificar un módulo individual, sin que se necesite conocer al resto de módulos.

Continuidad modular: Es importante para la extensibilidad y el mantenimiento. Las estructura debe ser tal que una tareas simple de mantenimiento no pueda conducir a situaciones de colapso catastrófico.

Protección modular: Este criterio acepta la existencia de los fallos de código, de plataforma de falta de recursos o de datos no previstos. Busca una estructura modular que haga viable y sencilla la recuperación de los errores imprevistos.



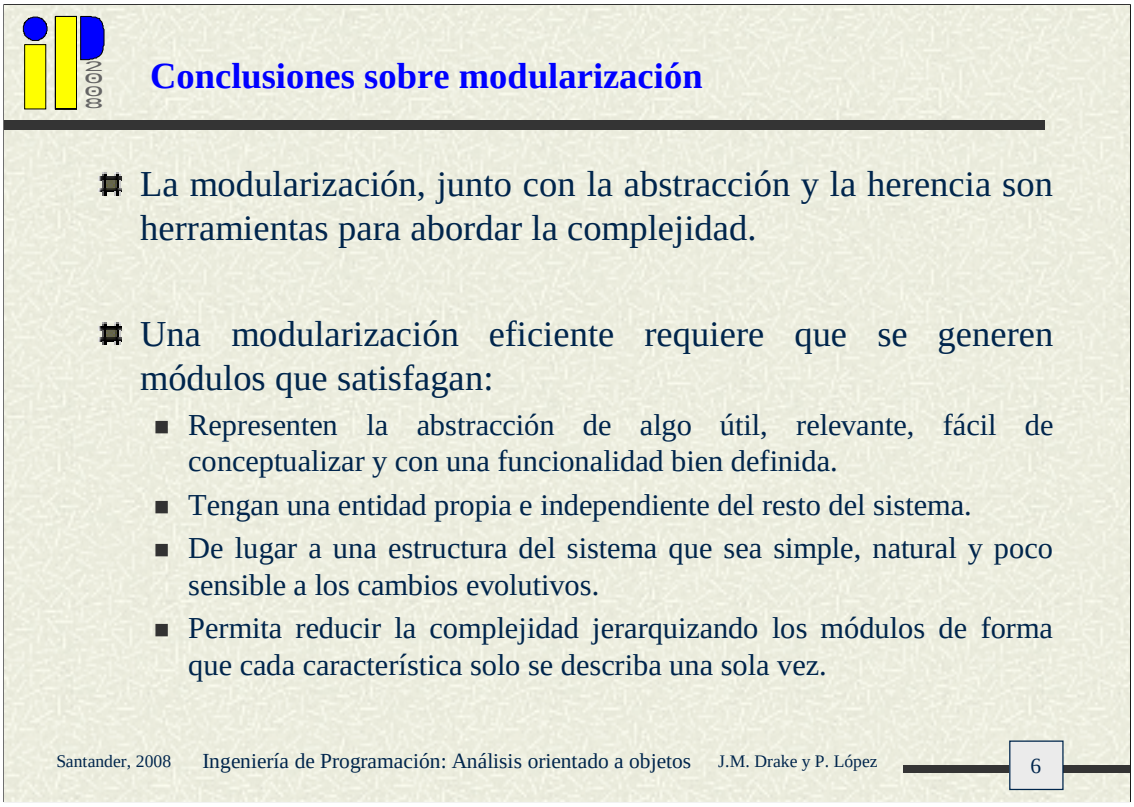
Unidad lingüística: Para conseguir Descomponibilidad, Componibilidad y Protección modular en la modularización de una aplicación se necesita formular el módulo como una unidad sintáctica.

Minimización del número de interfaces: Para conseguir la Continuidad y Protección modular se deben minimizar las interdependencia entre los módulos.

Acoplamiento débil: El intercambio entre módulos de solo los datos que son estrictamente necesarios, reduce la necesidad de revisión ante cambios (Continuidad modular) y la transmisión de condiciones anómalas y de excepción entre ellos (Protección modular).

Interfaces explicitas: Las posibilidades de interconexión entre módulos resulta normalizada (Componibili- dad) y completamente documentada (Comprensibidades).

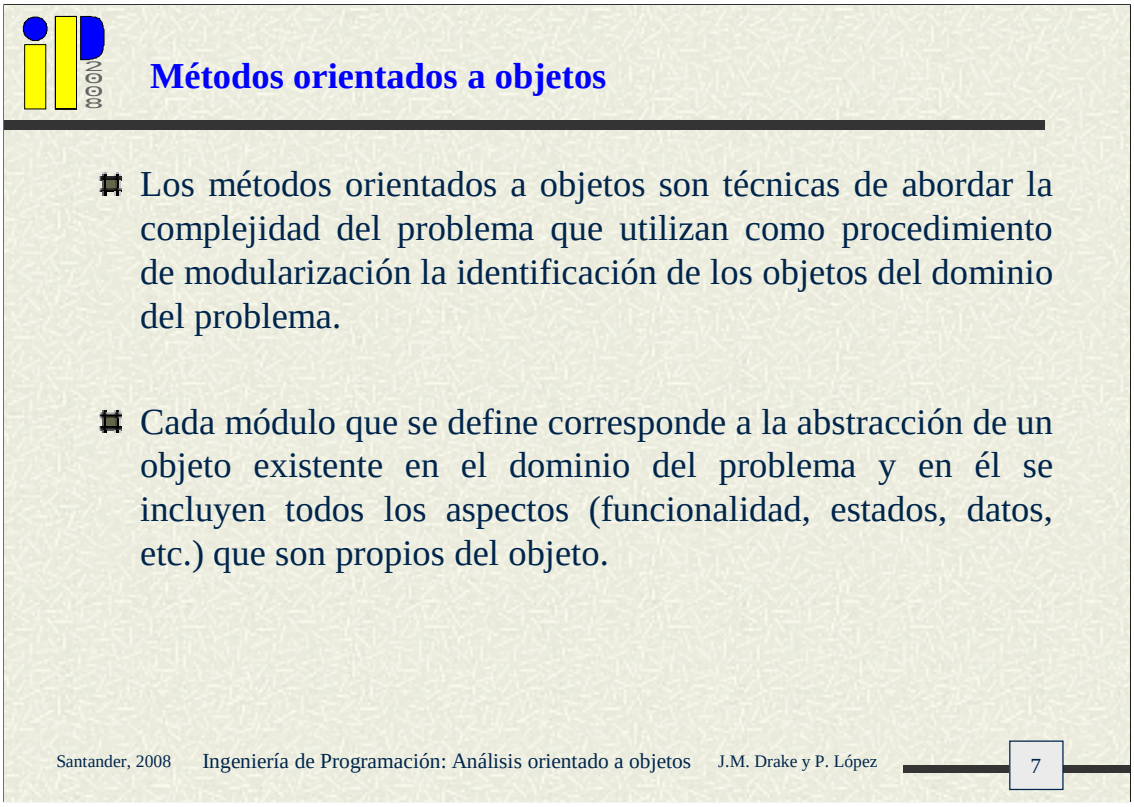
Ocultación de información: Independiza la funcionalidad de la implementación del módulo lo que facilita la Componibilidad, Comprensibilidad y Continuidad modular.



La modularización, la abstracción y la herencia son las principales herramienta que se disponen para abordar la complejidad.

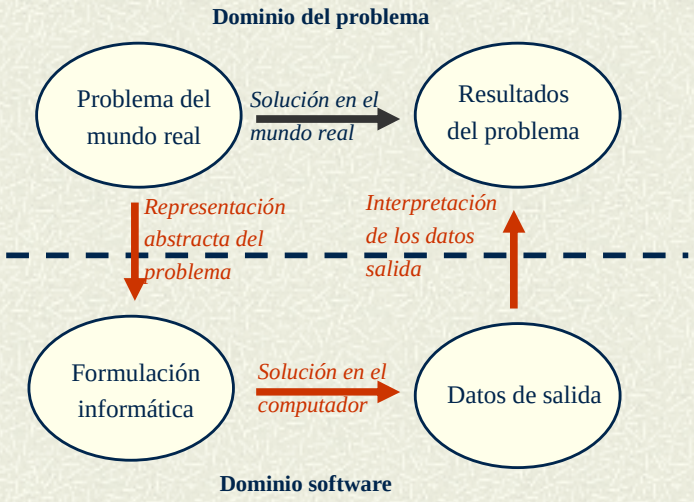
El análisis de un problema o el diseño de la aplicación software que lo resuelve debe llevarse a cabo mediante una eficiente modularización, lo cual requiere que el criterio que se utilice conduzca a la generación de módulos auténticos, esto es,

- Representen la abstracción de algo significativo y útil: y como consecuencia fácil de conceptualizar y de describir mediante una interfaz sencilla.
- Tengan una entidad propia e independiente del resto del sistema: y que por ello, se puedan diseñar, implementar, probar y utilizar de forma independiente y descentralizada.
- De lugar a una estructura del sistema que sea simple, natural y poco sensible a los cambios evolutivos.
- Permita organizar los componentes de forma que no se repita la información común.



La descomposición del problema se realiza utilizando los criterios propios del dominio del problema, y no en función de criterios de representación informática.
En la práctica diaria, los sistemas se conciben como compuestos de objetos interrelacionados. La forma en que describimos y comprendemos un sistema se realiza describiendo de forma independiente los objetos y luego describiendo el sistema en función de ellos y de las relaciones que entre ellos se establecen.
Como el número de objetos puede ser muy grande y sus características muy específicas, se trata de describir los objetos jerarquizados según ciertos tipos abstractos. Esto reduce la necesidad de repetir descripciones muy semejantes, y permite reutilizar las descripciones de los objetos en muchos sistemas diferentes.

Dominio del problema y dominio informático.

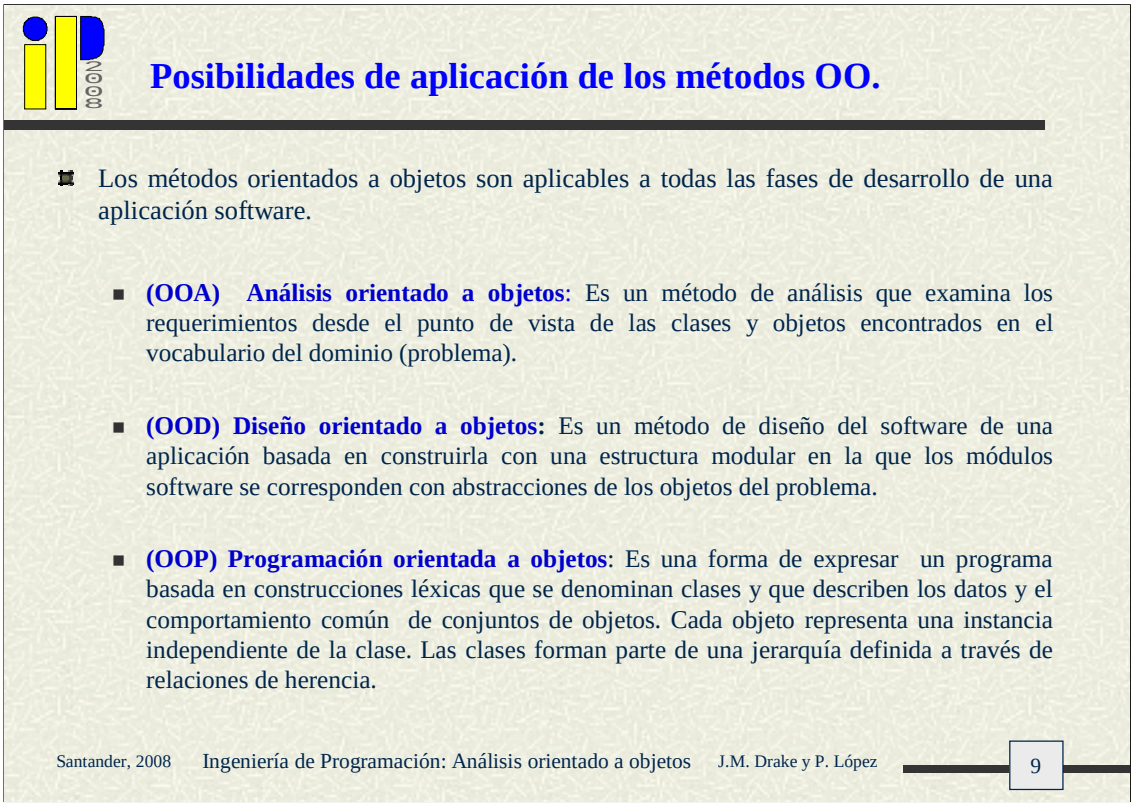


Santander, 2008 Ingeniería de Programación: Análisis orientado a objetos J.M. Drake y P. López 8

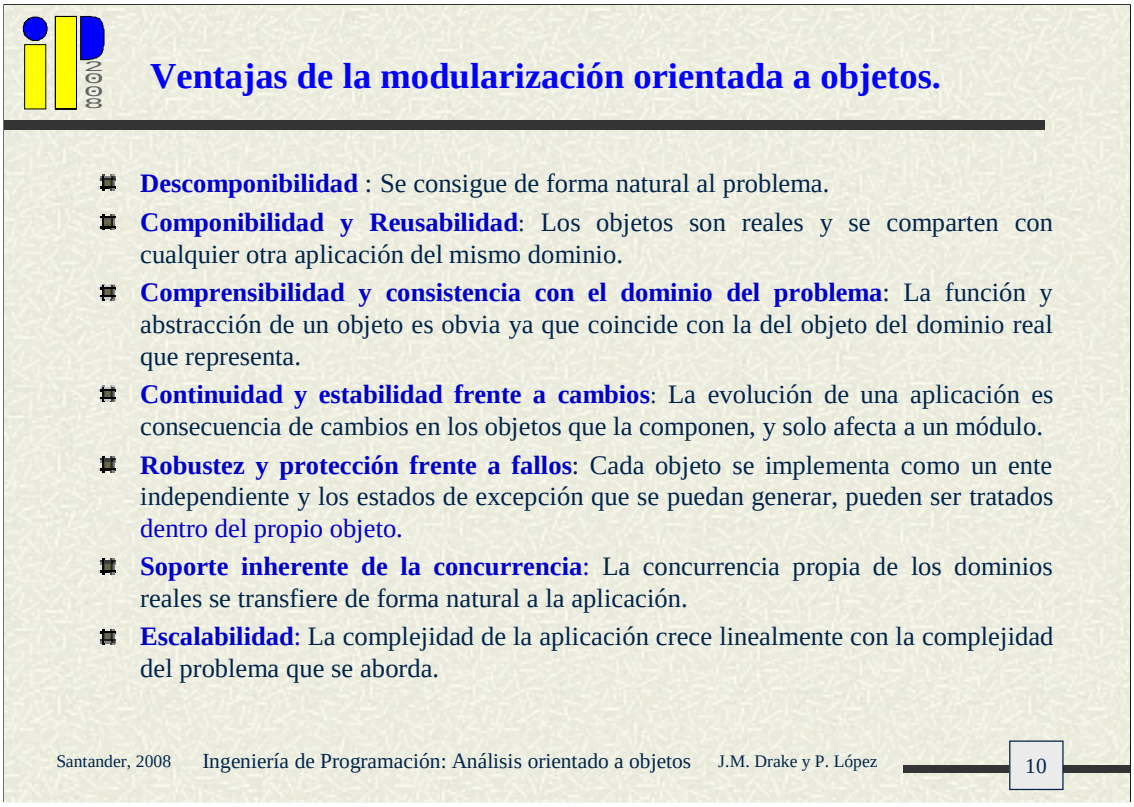
La complejidad de los problemas que se resuelven es consecuencia del desacoplo entre el lenguaje y los conceptos del dominio del problema que se resuelve y los del dominio informático.
El cliente que plantea el problema y el ingeniero que lo resuelve mediante una aplicación software, manejan diferentes interpretaciones del problema:

- Tiene diferente visión de su naturaleza.
- Conciben soluciones diferentes.
- Intercambian entre ellos especificaciones muy complejas.

Pequeños cambios en las especificaciones del problema pueden requerir cambios muy importantes (cualitativamente y cuantitativamente) en la correspondiente formulación informática.



Los métodos orientados a objetos son útiles en todas las fases del desarrollo del software:
Análisis orientado a Objetos (OOA): Es un método de análisis identifica y describe los requerimientos de un problema en función de los objetos que intervienen y en las relaciones de asociación e interacción que existen entre ellos.
Diseño Orientado a Objetos (OOD) : Es un método de diseño del software de una aplicación basada en construirla con una estructura modular en la que los módulos software se corresponden con abstracciones de los objetos del problema.
Programación Orientada a Objetos (OOP): Es un método de generación del código de una aplicación en el que las unidades lingüísticas que se utilizan corresponden a código que representan a los objetos del problema.
El método OO puede utilizarse indiferentemente en una fase independientemente o en las tres fases de forma coordinada.



Con este criterio se satisfacen de forma natural todos los criterios de modularización antes expuestos:

•Descomponibilidad modular: Se consigue descomponer de forma natural al problema. La descomposición de la aplicación coincide con la descomposición que resulta del análisis del dominio

•Componibilidad modular y reusabilidad: Los objetos de un problema son reales y y aparecerán en otros problemas del mismo dominio.

•Comprensibilidad modular y consistencia con el dominio del problema: La funcionalidad y abstracción de un módulo es obvia ya que coincide con la del objeto del dominio real que representa.

•Continuidad modular y estabilidad frente a cambios: Las evolución de un problema es consecuencia de cambios pequeños en los objetos que lo componen, y en consecuencia afecta solo al módulo que se representa.

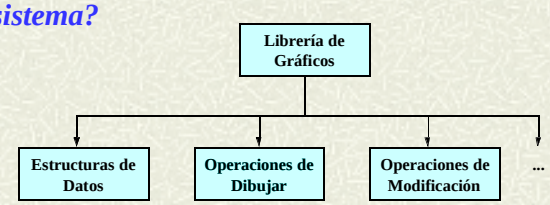
•Robustez y protección frente a fallos: La mayoría de las situaciones anormales que pueden conducir a fallos, son consecuencias de agente externos, y por tanto se pueden definir y tratar a nivel del módulo que corresponde al objeto responsable.

•Soporte inherente de la concurrencia: Los objetos que resultan del análisis de un problema son concurrentes como lo es habitualmente el dominio al que corresponde y pueden transferirse a los objetos que constituyen la aplicación.

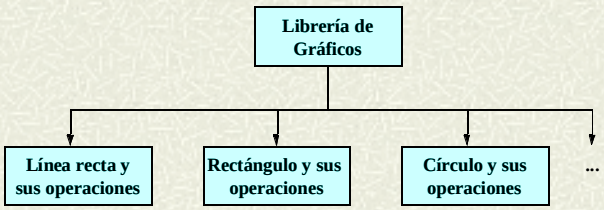
•Escalabilidad: Cuando crece la complejidad del problema, no crece patológicamente la complejidad de la aplicación.

Comparación con los métodos estructurados

- En los métodos estructurados el criterio de modularización es la funcionalidad. ***¿Qué hace el sistema?***



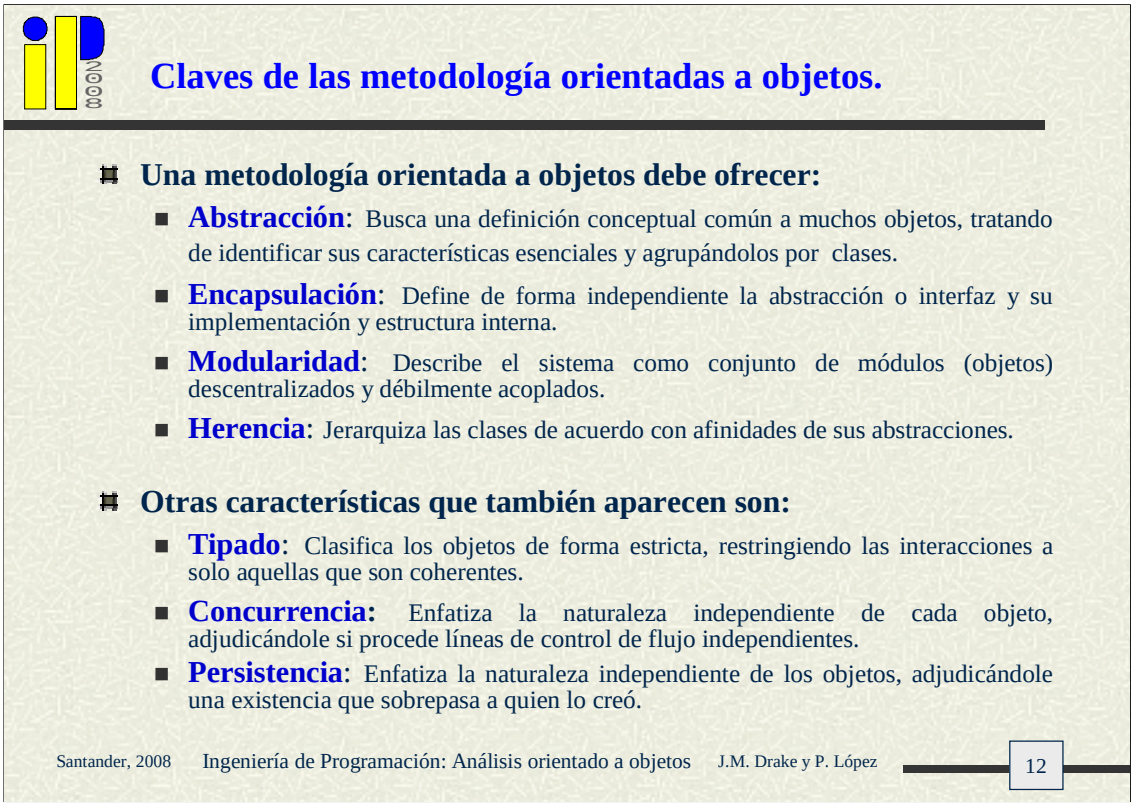
- En los métodos orientados a objetos el criterio de modulari-zación son los elementos que componen el sistema. ***¿Qué componentes tiene el sistema?***



Santander, 2008 Ingeniería de Programación: Análisis orientado a objetos J.M. Drake y P. López 11

En los métodos estructurados el criterio de modularización es la funcionalidad. La pregunta básica que se realiza en este caso, es: *¿Que hace el sistema?*, como respuesta a esta pregunta aparecen las acciones que se identifican con los módulos que son de naturaleza es funcional. La estrategia es típicamente Top-down. Se parte del programa principal y luego se modulariza.

En los métodos orientados a objetos, el criterio de modularización son los componentes del problema. La pregunta básica que se realiza es *¿Que componentes tiene el problema?*, como respuesta se identifican los objetos que intervienen en el problema y la forma que interactúan entre sí. Como consecuencia de su naturaleza y de las interacciones que se producen, resulta la funcionalidad. El programa principal, si existe, es lo último que se aborda.



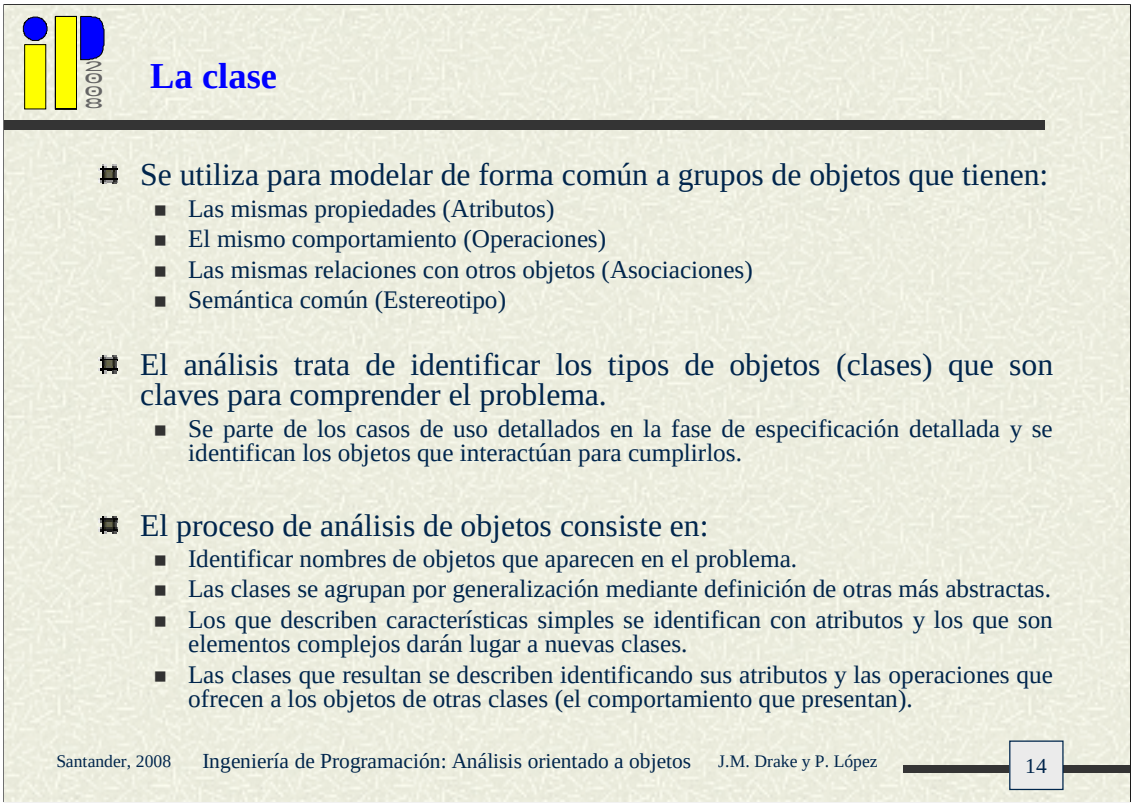
•**Abstracción:** Trata de identificar las características esenciales de los objetos para clasificarlos por clases, y buscarle una definición conceptual común independiente de los detalles.
•**Encapsulación**: Trata de describir de forma separada la abstracción o interfaz que describe su funcionalidad y su capacidad de interacción con otros objetos que es pública, y su implementación y estructura interna que es privada.
•**Modularidad**: Trata de describir al sistema como conjuntos de módulos (objetos) descentralizados y débilmente acoplados.
•**Herencia**: Trata de jerarquizar las clases de objetos de acuerdo con la afinidades de sus abstracciones, a través de relaciones de relaciones de generalización y particularización.
•**Tipado**: Clasifica las características de los objetos de forma estricta, de forma que restringe las interacciones entre los objetos, a solo aquellas que son coherentes.
•**Concurrencia**: Enfatiza la naturaleza independiente de cada objeto, adjudicándole si procede líneas de control de flujo propio, lo que conduce a una actividad concurrente del sistema imagen de la del problema real.
•**Persistencia**: Enfatiza la naturaleza independiente de los objetos, adjudicándole una existencia que puede sobrepasar en el tiempo o en el espacio la del objeto que lo creó.

Elementos de modelado en OO

- Describen los elementos básicos que constituyen el sistema:
  - **Objeto**: Es un concepto de ejecución y es una instancia de datos junto con la descripción de las operaciones que actúan sobre ellos.
    - Los datos del objeto residen en "**atributos**".
    - Las operaciones que actúan sobre los datos son los "**métodos**"
  - **Clase**: Es un concepto de diseño y describe las características de un tipo de objetos, de los que se pueden crear múltiples instancias.
  - **Interfaz**: Representa un conjunto de operaciones que en conjunto representan un servicio. Es un elemento de diseño principalmente orientado hacia la reusabilidad. Aislan al especificación de un servicio de su implementación.

Santander, 2008 Ingeniería de Programación: Análisis orientado a objetos J.M. Drake y P. López 13

Los objetos, las clases y la interfaces son elementos de modelado estructural básico, destinados a representar e incorporar la información que corresponden con los conceptos de bajo nivel que constituyen la aplicación.
Un objeto modela una estructura de datos que se instancia durante la fase de ejecución de la aplicación. El objeto modela tanto la información que lleva asociada y que se describe mediante atributos y las operaciones que pueden operar sobre ella y que se describe mediante métodos. Estos métodos son invocados por otros objetos clientes o por otros métodos del mismo objeto.
Una clase describe durante la fase de diseño a un conjunto de objetos que instancian las mismas estructuras de datos y admiten los mismos tipos de operaciones. Durante la fase de ejecución pueden instanciarse muchos objetos que correspondan a la misma clase, mientras que un objeto es siempre la instancia de una única clase.
Las clases se pueden definir en función de otras clases definidas previamente, bien por especialización, extensión, agregación, asociación, etc. lo que reduce considerablemente la complejidad de la definición del gran número de instancias que requiere una aplicación.
Las interfaces permiten nombrar conjuntos de operaciones que representan un servicio completo y que pueden ser ofertados por diferentes clases que los ofertan. Su función es independizar el servicio de la implementación. Dos clases que realicen una misma interfaz, ofrecen el mismo servicio, lo que significa que si una clase requiere un servicio, lo puede obtener de cualquier clase que realice la misma interfaz.
Una interfaz no es directamente instanciables. Las interfaces se realizan a través de clases. Para ello la clase debe implementar un método concreto, por cada operación de la interfaz.

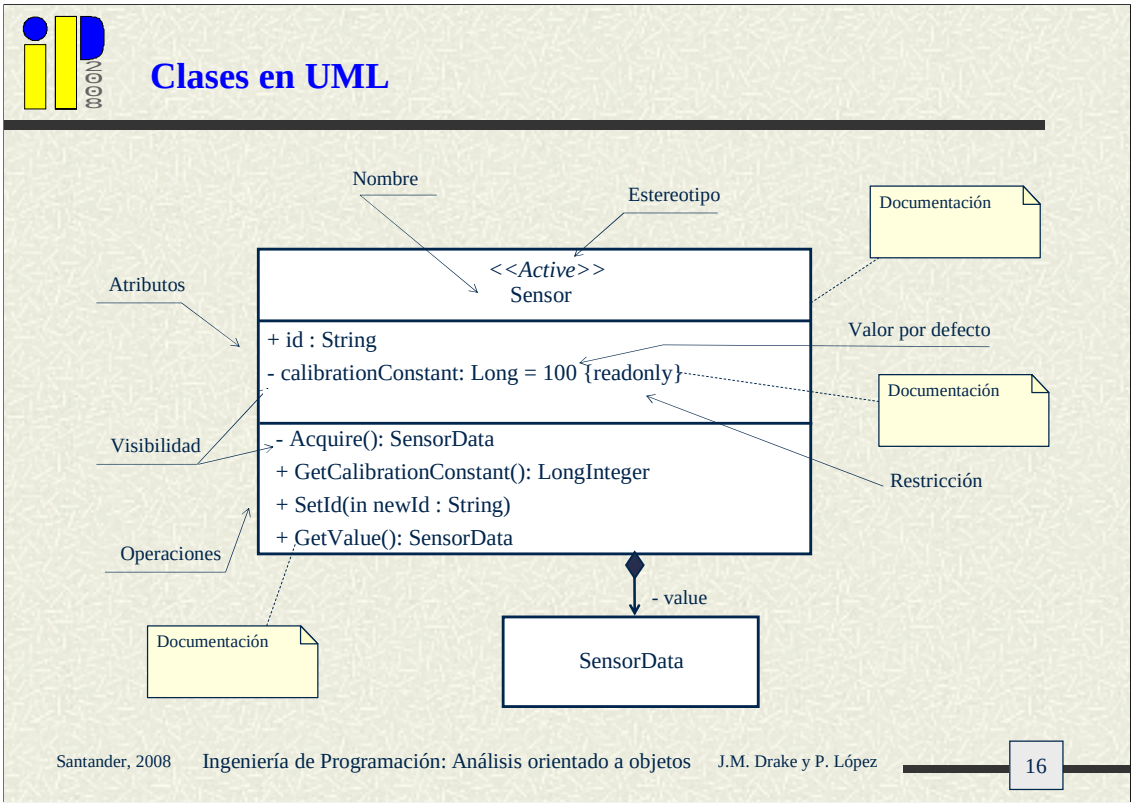


Representa un grupo de objetos que tienen similares propiedades (atributos) , comportamiento (operaciones) y formas semejantes de relacionarse con objetos de otras clases (asociaciones) y una semántica común (estereotipo). Se utilizan como mecanismo de describir de forma unificada todos aquellos objetos que aunque tienen diferente entidad corresponden a una misma descripción.
Todos los objetos tienen una clase, a veces por ser los únicos elementos de la clase la descripción de ambos se hace conjuntamente y sin marcar la diferencias.
En el análisis buscamos el modelado del problema por identificar los objetos que forman parte de él y agruparlos en clases que describen sus características y comportamiento. La clases y objetos están en el enunciado del problema y el análisis trata de descubrirlos, pero también hay una fase de invención ya que tienen que ser abstraídos y dotados de los mecanismos que instrumentan su comportamiento.
Las clases y objetos suelen aparecer del análisis de elementos como:
•Cosas tangibles: Coches, Facturas, etc.
•Personas: Que realizan o llevan a cabo alguna responsabilidad o role (madre, profesor, etc.)
•Eventos: Aterrizaje, interrupción, requerimiento,etc.
•Interacciones: Carga, encuentro, intersección, etc.

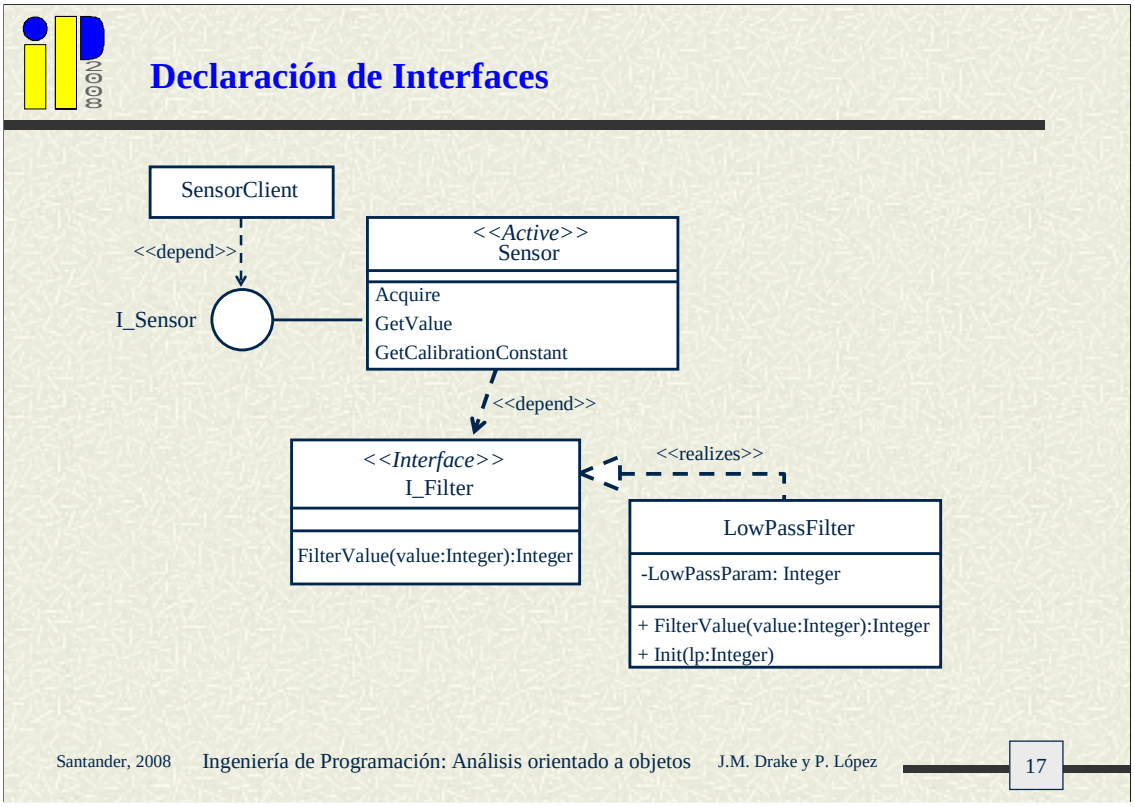
Proceso de análisis de un sistema.

- El análisis de un sistema consiste en:
  - Formular el diagrama de clases que:
    - Identifica las clases.
    - Formula las relaciones entre ellas
    - Asigna los atributos.
    - Describe la operaciones que ofrece.
  - Describir mediante diagramas de estados el modelo interno de las clases.
  - Describir mediante diagramas de actividad los algoritmos que describen las operaciones de la clase.
  - Documentar exhaustivamente el diagrama de clases.
  - Justificar los casos de uso de la aplicación desde los diagramas de objetos (detallandoles para los objetos recien identificados)

Consejos prácticos para realizar el análisis del sistema son:
•Tratar primero de comprender el sistema estableciendo un modelo conceptual que posteriormente se plasma en un modelo de objetos.
•Mantener el modelo de objetos sencillo sin incorporar un exceso de detalles.
•Cuidar especialmente de elegir los nombres adecuados y no ambiguos de las clases, de los atributos y de los roles de las asociaciones. Elegir buenos nombres es uno de los aspectos mas difíciles pero mas importante de un modelo de objetos.
•No introducir atributos de tipo puntero sino formularlos como asociaciones cualificadas que hacen mas intuitivo el modelo.
•No preocuparse de la multiplicidad exacta de la asociaciones al principio, este es un aspecto del que se tratará con mas detalle en la fase de diseño.
•No colapsar los diagramas con excesos de enlaces, reducirse a los fundamentales.
•Intentar evitar excesivos niveles de generalización.
•Inspeccionar y revisar el diagrama de clases las veces que sea necesarias. Plantear un buen modelo de objetos de un sistema es una tarea que requiere múltiples iteraciones para clarificar los nombres, reparar errores, sumar detalles y capturar restricciones estructurales.
•Documentar exhaustivamente los diagramas de clases que resulten. Documentar las clases, los atributos, las operaciones, etc.



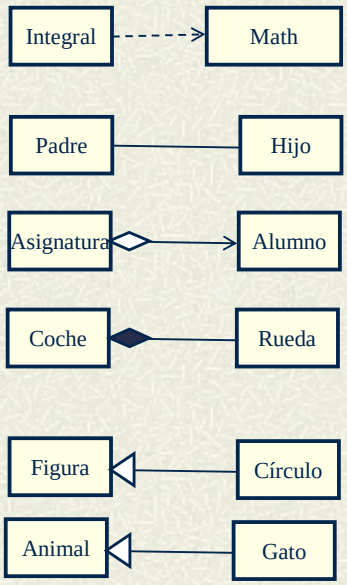
En UML una clase se representa mediante un rectángulo con tres compartimentos, el superior contiene el identificador y en su caso el estereotipo, el medio contiene los atributos, y el inferior, las operaciones o métodos que tienen asociados.
El estereotipo, que puede ser asociado no solo a las clases, sino a cualquier elemento UML, le asocia una semántica especial (que no tiene ningún modo de representación específico en UML)
Los atributos se definen mediante un identificador, y en su caso, su tipo, un valor por defecto y su visibilidad. El tipo puede corresponder a algún tipo primitivo definido en el dominio o cualquier otra clase definida en el sistema. La visibilidad indica quien puede hacer referencia a ellos (privados, públicos, reservados, etc.) y es fuertemente dependiente del lenguaje que se utilice. Los valores de visibilidad que se admiten en UML son :
•Public (+): Accesible desde cualquier otra clase del modelo
•Protected (#) : Accesible desde la propia clase y todas aquellas que heredan de ella, pero nunca desde clases externas a la jerarquía.
•Private (-) : Accesible solo desde la propia clase
•Package (~): Accesible sólo desde clases que se encuentren en el mismo paquete
En orientación a objetos los atributos suelen ser siempre privados o protegidos, si queremos acceder a ellos hay que declarar un método de acceso.
Las operaciones se definen mediante un identificador, un conjunto de parámetros con sus identificadores, tipos y valores por defecto, el tipo del valor de retorno, y la visibilidad.
Las clases pueden estar declaradas como abstractas, en cuyo caso no son directamente instanciables, y solo tienen la función de servir como base para la declaración de nuevas clases derivadas de ellas.
Los atributos y las operaciones también pueden declararse como de clase (estáticos), y ser compartidos por todas las instancias. En ese caso aparecen subrayados.
Cualquier elemento de la clase puede tener asociado un texto de documentación



Las interfaces son conjuntos de operaciones agrupadas bajo un identificador por corresponder a un servicio que pueden ser ofrecidos por múltiples clases. En UML se representan como una clase con el estereotipos <<interface>>.
En un diagrama de clases una interfaz se representa o bien con el símbolo de ordinario de clase o bien mediante un icono consistente en un círculo.
Una clase que realiza una interfaz puede representarse bien asociándole el icono de la interfaz, o bien representando la clase como derivada de la clase que representa la interfaz y asignando a la asociación el estereotipo <<realizes>>.
Cuando una clase requiere el servicio representado por la interfaz para implementar su métodos, se relacionan mediante una asociación de dependencia con el estereotipo <<depend>>. En fase de ejecución la interfaz deberá ser sustituida por una implementación de una clase que realice esta interfaz.

Tipos de asociaciones UML

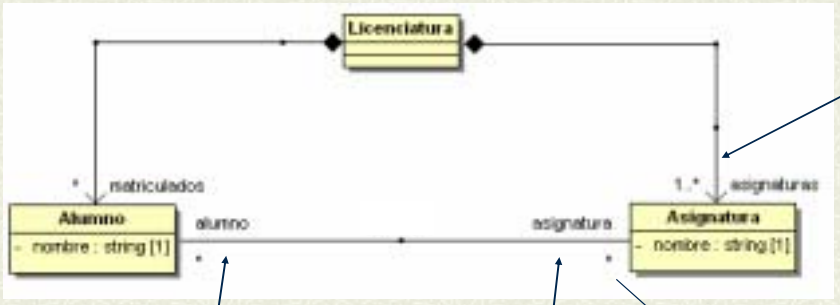
- **Dependencia**: Una clase hace uso, o requiere visibilidad sobre otra
- **Asociación**: Una clase está relacionada o posee una referencia a otra, que puede usar para satisfacer su funcionalidad
- **Agregación**: Una clase está contenida en la otra pero puede ser compartida con otras clases
- **Composición**: Una clase se compone de otras clases a las cuales posee en exclusividad. Es responsabilidad de la clase contenedora la creación y destrucción de las clases contenidas.
- **Generalización:** Una clase es un tipo de otra:
  - **Herencia**: Una clase extiende a la otra con nuevos atributos y métodos
  - **Especialización**: Una clase especializa la funcionalidad de la clase raíz



Santander, 2008 Ingeniería de Programación: Análisis orientado a objetos J.M. Drake y P. López 18

Las asociaciones son los elementos que ofrece UML para representan las relaciones que existen entre los elementos estructurales que se utilizan en los modelos. La asociación es un concepto de la fase de diseño y se establece entre clases, mientras que el enlace (“link”) es su instanciación para los modelos de la fase de ejecución.
En UML se definen muchas clases de asociaciones:
•Dependencia: Una clase requiere visibilidad sobre otra para hacer uso de su funcionalidad. El ejemplo mas simple es la visibiliiad sobre clases de utilidad como librerias matemáticas, de entrada-salida etc
•Asociación : Representa que un clase posee una referencia a otra como parte de sus atributos.
•Agregación: Representa una clase que es parte de otra.
•Composición: Es un tipo de agregación fuerte que significa que una clase se compone de otras clases definidas. La diferencia entre agregación simple y composición esta en la exclusividad (una clase puede estar agregadas en varias otras clases) y de responsabilidad de creación y destrucción (si una clase está relacionada por composición con otra, ésta es responsable de que aquella se cree y se destruya con su creación y destrucción.
•Generalización: una clase es una generalización de otra si es un tipo de ella, esto es, donde pueda estar la clase mas abstracta puede estar la clase mas concreta. Existen dos formas básicas de generalización:
•Herencia: Una clase se crea a partir de otra, incorporando de la antecesora todo lo establecido en la definición de ella mas nuevos aspectos que se añaden en su declaración específica.
•Especialización: Una clase se crea a partir de otra, especializando su funcionalidad de acuerdo con nuevas necesidades.

Caracterización de asociaciones



Asociación direccional

Identificador o "role" de la asociación en Asignatura

Identificador o "role" de la asociación en Alumno

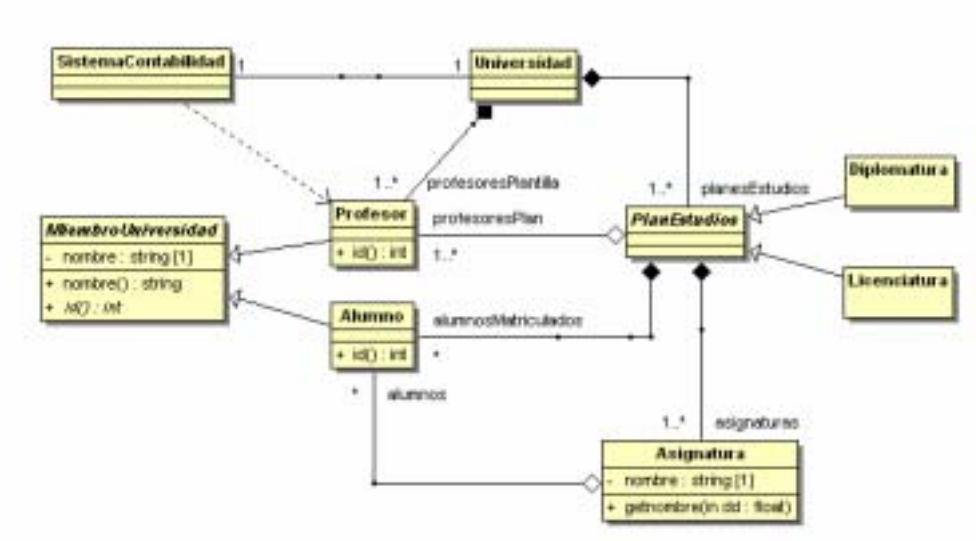
Multiplicidad

1	Una y solo una
0..1	Cero o una
*	Cero o mas
1..*	1 o mas
n..m	De n a m (enteros)
-	Inespecificado

Santander, 2008 Ingeniería de Programación: Análisis orientado a objetos J.M. Drake y P. López 19

Cada asociación lleva una indicación de multiplicidad que muestra cuantos objetos de la clase considerada pueden ser relacionados con un objeto de otra clase.
Un valor de multiplicidad mayor que 1 implica un conjunto de objetos y debe ser implementado con algún tipo de contenedor (array, lista, cola, etc.)
Los valores de multiplicidad implica restricciones relacionadas con el ámbito de la aplicación, válidas durante toda la existencia de los objetos. Las multiplicidades no deben considerarse durante los regímenes transitorios, como en la creación y destrucción de objetos.
La materialización de las asociaciones toma todo su interés para las asociaciones n a n. En las asociaciones 1 a 1, los atributos pueden desplazarse a cualquiera de los objetos. En las relaciones 1 a n pueden asociarse a la clase del lado n, sin embargo en la clase n a n el desplazamiento no es posible, y se requiere asociar una clase (contenedora del atributo) a la propia asociación.
Las asociaciones pueden estar orientadas o no. Una asociación orientada se expresa con una flecha y es solo utilizada por la clase origen para hacer uso de la clase destino. Si no no es orientada se representa sin flecha, y en ese caso es utilizable por ambas clases.
En los extremos de una asociación se puede incluir la cardinalidad, esto es el número de objetos de esa clase con los que se puede estar enlazado en fase de ejecución.
En los extremos de una asociación se pueden incluir un identificador o "role". Para una determinada clase, el role con el que él identifica internamente a la asociación se situa en el extremo opuesto de la asociación.

Representación de asociaciones en diagramas de clases.



Dependencia: Es un tipo de asociación muy débil en la que se indica que para definir o explicar una clase se debe tener en cuenta la definición de la otra. Ejemplos de dependencias son: <<abstraction>> una clase resulta como una concreción de otra mas abstracta; <<bind>> una clase resulta de concretar una clase genérica; <<usage>> una clase hace uso internamente de otra, y en consecuencia, requiere visibilidad sobre ella. <<permission>> una clase tiene derechos especiales de acceso a los recursos de otra.

Una asociación se representa en los diagramas de clases UML como una línea que enlaza las clases que se relaciona.

La agregación se representa mediante un pequeño rombo en el extremo próximo a la clase contenedora. Si la agregación es simple, el rombo está vacío, y si es una composición el rombo esta relleno.

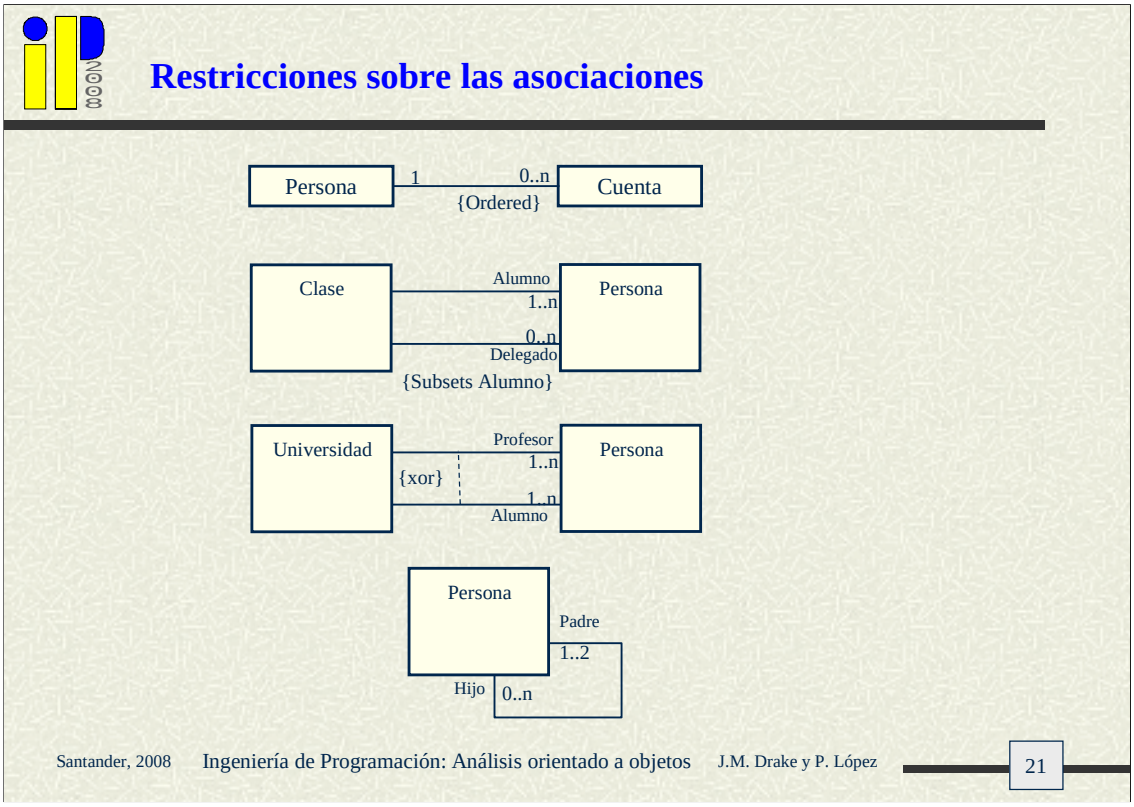
La herencia se representa mediante un pequeño triángulo vacío en el extremo de la clase mas general.

La dependencia se representa mediante una línea punteada.

Las asociaciones pueden estar orientadas o no. Una asociación orientada se expresa con una flecha y es solo utilizada por la clase origen para hacer uso de la clase destino. Si no no es orientada se representa sin flecha, y en ese caso es utilizable por ambas clases.

En los extremos de una asociación se puede incluir la cardinalidad, esto es el número de objetos de esa clase con los que se puede estar enlazado en fase de ejecución.

En los extremos de una asociación se pueden incluir un identificador o “role”.



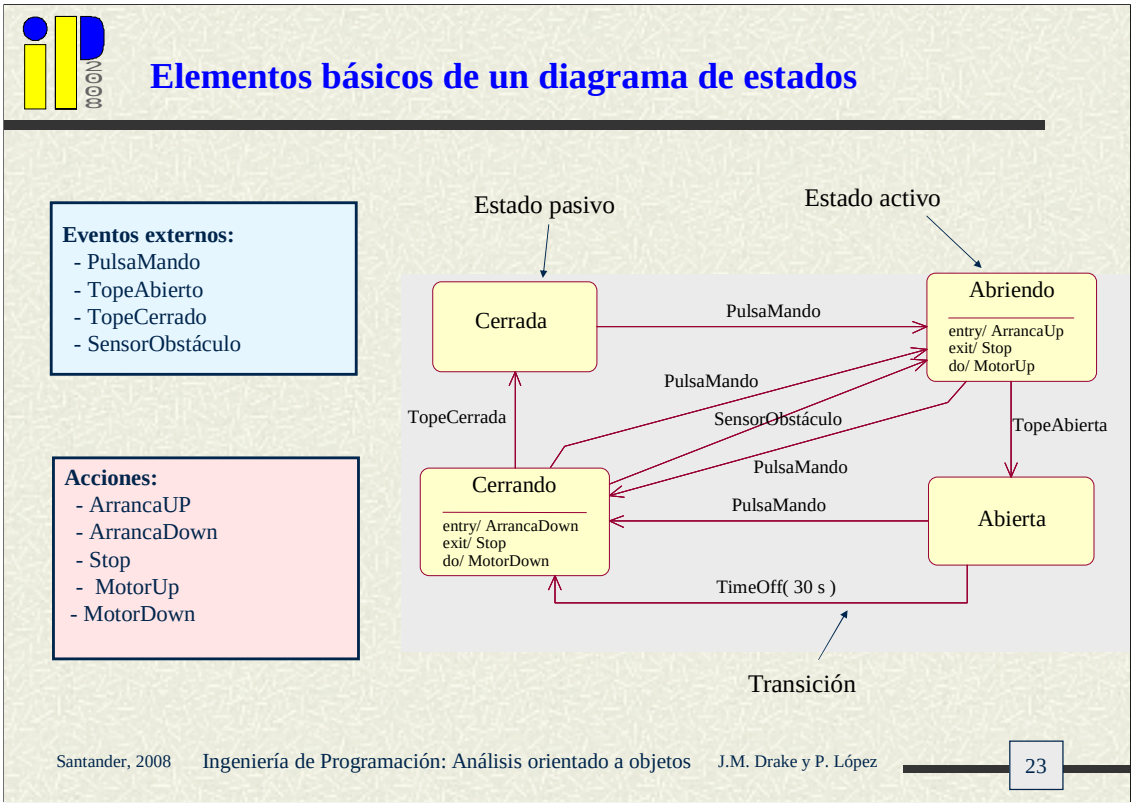
Pueden definirse diferentes tipos de restricciones entre las asociaciones de una clase. Las restricciones se representan en los diagramas por expresiones encerradas entre llaves.
La restricción {ordenada} puede colocarse sobre una asociación para especificar una relación de orden entre los objetos enlazados por la asociación.
La restricción {subconjunto} indica que el conjunto de una asociación está incluido en el conjunto establecida por la otra asociación.
La restricción {O exclusiva} indica que para un objeto dado, es válida una sola asociación entre un grupo de asociaciones.
Las asociaciones pueden también enlazar una clase consigo misma. Este tipo se llama asociaciones reflexivas. El role de las asociaciones toma toda su importancia para distinguir las instancias que participan en la asociación.

Diagramas de estado

- Describe el comportamiento de un elemento estructural o de un método como una máquina de estados finitos, basada en la representación de los estados del elemento y de especificar las causas de transiciones entre ellos.
- Se puede describir comportamiento asociando acciones a:
  - Las transiciones entre estados
  - La entrada a los estados
  - La salida de los estados
- Los diagramas de estados de UML son muy potentes y escalables:
  - Permite agregar un diagrama de estado a un estado.
  - Admite concurrencia a través de estados AND.
  - Permite introducir semántica dinámica a través de pseudoestados.
  - Admite transiciones condicionales y sincronizadas.

Santander, 2008 Ingeniería de Programación: Análisis orientado a objetos J.M. Drake y P. López 22

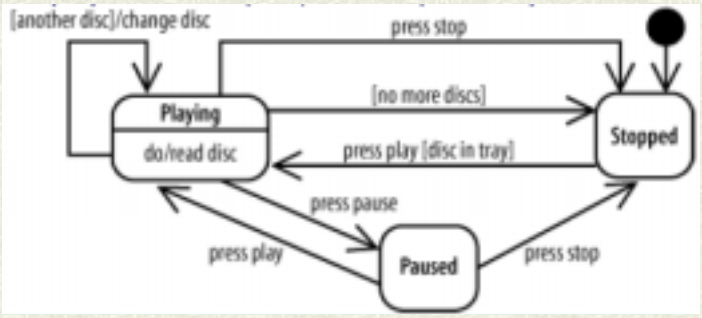
Un diagrama de estados describe una máquina de estados finita basada en un conjunto de estados en que se puede encontrar el sistema, el conjunto de transiciones entre estados disparados por eventos, y los conjuntos de acciones o actividades que se ejecutan en función de las transiciones que se realizan o de los estados en que se encuentra el sistema.
Los diagramas de estado UML admiten:
•Estados formulados como diagramas de estados.
•Máquinas de estados concurrentes sobre un mismo diagrama, y por tanto la definición de estados compuestos AND del elemento.
•Diferentes tipos de psudoestados para organizar la inicialización, acceso, salida o persistencia en las máquinas de estado.
•Admite transiciones condicionales en las que en función de condiciones de guarda las posibles transiciones se habilitan o deshabilitan.
•Admite pseudoestados destinados a la sincronización de eventos concurrente o a la generación simultánea de múltiples eventos.
Permite utilizar eventos asíncronos que son implementados a través colas de eventos en el destino, esto es, eventos que el que los genera los olvida, pero que se mantienen encolados en el destinatario está que está en un estado en el que puede gestionarlo.
También admite eventos síncronos en el que se sigue la política de que el que los envía permanece bloqueado hasta que el evento es gestionado.



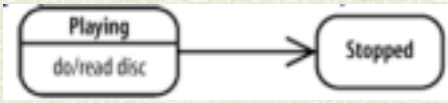
Los estados pueden ser estados pasivos, durante los cuales el sistema no ejecuta ninguna actividad, o activos: durante los cuales el sistema ejecuta alguna actividad
El comportamiento describe las acciones que se producen mientras que el sistema se encuentra en un estado:
entrry/behavior => Acción que se realiza cuando se llega a un estado.
do/behavior => Actividad que se ejecuta mientras se está en un estado.
exit/behavior => Acciones que se ejecuta cuando se abandona un estado.

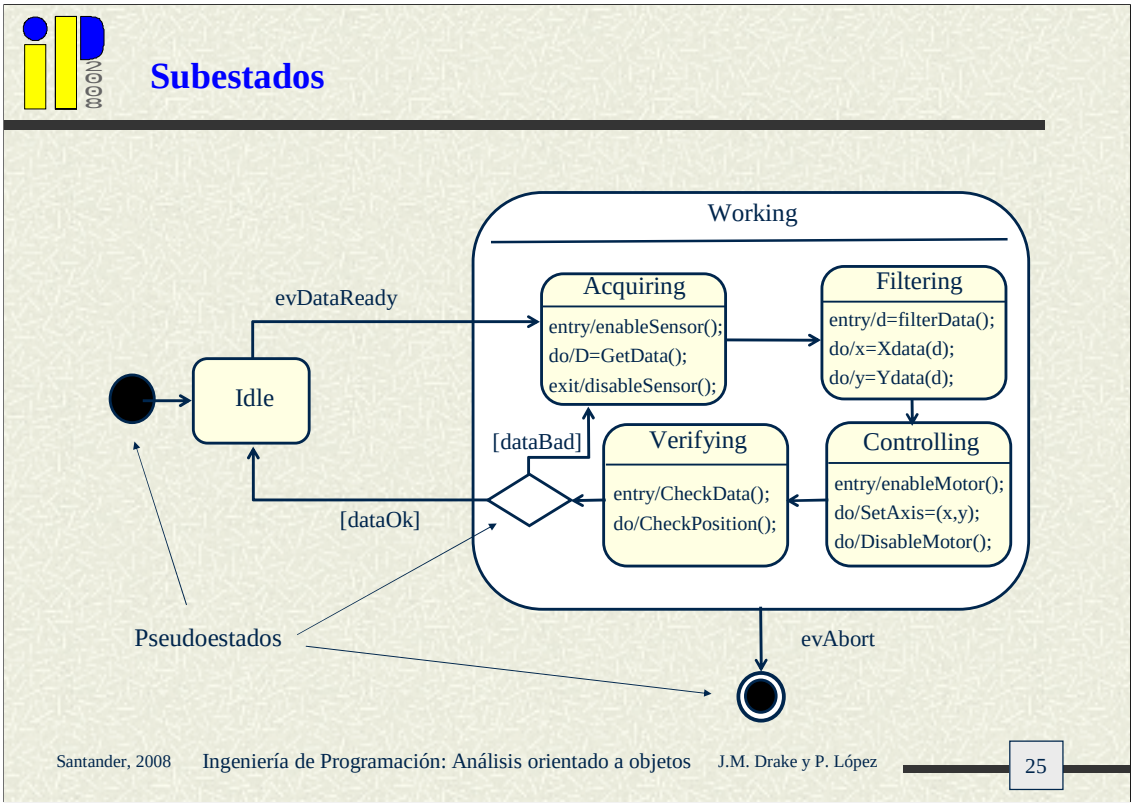
Semántica de las transiciones

- La semántica completa de una transición es: *trigger[condición]/acción*



- Si una transición no tiene trigger ni guarda, la transición se produce por finalización de la actividad asociada al estado.





Dentro de un estado se pueden incluir una sección de diagrama estados, que indican sub estados discernibles dentro de él.
Mediante líneas discontinuas se separan diferentes máquinas de estados finitos que concurren dentro de un mismo diagrama.
Existen definidos muchos pseudoestados para organizar el diagrama:
•Estado inicial: (circulo relleno) Punto de inicio de la actividad en el diagrama.
•Estado salida: (ojo de buey) Punto por el que se abandona la actividad.
•Condicional: (rombo) Se elige una transición de acuerdo con una condición.
•Histórico:(Circulo con H) Se inicia la actividad por el estado en que se abandonó.

Estados concurrentes