**Capítulo 12 (UML 2): Realización de CU**

**Realizaciones de un CU**

Las clases de análisis modelan la estructura estática de un sistema y las realizaciones de CU muestran cómo las instancias de esas clases interactúan para realizar la funcionalidad del sistema.

La realización de CU de análisis, se centra en capturar atributos, operaciones y relaciones claves entre las clases de análisis.

La realización de un CU es cuando se convierte un CU en diagramas de clase y diagramas de interacción (son especificaciones de alto nivel de un sistema). Cada CU tiene exactamente una realización de CU.

En la realización se utilizan los siguientes elementos:

* Diagramas de clase de análisis: interacciones entre las clases para realizar el CU.
* Diagramas de interacción: interacciones entre las instancias que realizan el CU. Son “instantáneas” del sistema que se ejecuta.
* Requisitos especiales: se pueden descubrir nuevos requisitos que se deben capturar.
* Mejora del CU: se puede descubrir nueva información, por lo que el CU original se debe actualizar.

La realización es un proceso de mejora. Va de una especificación general de un comportamiento requerido (diagramas de clases de análisis), a una descripción detallada de las interacciones entre clases y objetos que harán que ese comportamiento sea real (diagramas de interacción).

El modelado orientado a objetos es un proceso iterativo, por lo que es probable ir descubriendo nuevos requisitos o tener que modificar un CU una vez que empieza a modelar en más profundidad. Todo esto es parte de la realización de CU. Se deben mantener los documentos existentes actualizados (modelo de CU, de requisitos y las clases de análisis).

Interacciones

Son sencillas unidades de comportamiento de un clasificador de contexto (proporciona el contexto para la interacción). La interacción (mensajes, líneas de vida), se muestra en análisis, con:

1. **Diagrama de secuencias**: enfatizan la secuencia ordenada en el tiempo de eventos entre líneas de vida. Sencillos de leer, pero tienden a saturarse muy rápidamente.

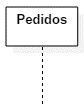
Cuando necesita centrarse en la secuenciación real de los eventos, siempre es mucho más sencillo utilizar un diagrama de este tipo.

1. **Diagrama de colaboración o comunicación**: enfatizan las relaciones estructurales entre objetos, útiles para crear un rápido bosquejo de una colaboración de objeto. En UML, son semánticamente más débiles que los de secuencia.

En diseño:

1. **Diagrama de visión de interacción**: enfatizan relaciones entre interacciones. Muestran lo complejo que se realiza el comportamiento por un conjunto de interacciones sencillas. Caso especial de diagramas de actividad en el que los nodos hacen referencia a otras interacciones. Útiles para modelar el flujo de control dentro de un sistema.
2. **Diagrama de tiempo**: enfatizan aspectos reales de interacciones. Su finalidad: ayudarle a razonar sobre el tiempo.

**Diagramas de secuencia**

Líneas de vida

Una línea de vida representa cómo una instancia de un clasificador participa en la interacción.

Elementos de una línea de vida:

* **Nombre**: Para hacer referencia a la línea de vida dentro de la interacción. Opcional.
* **Tipo**: Nombre del clasificador que representa.
* **Selector**: condición booleana que se puede usar para seleccionar una sola instancia que satisface la condición.

La interacción describe cómo las instancias del clasificador interactúan de una forma general, en lugar de especificar solamente una interacción particular entre un conjunto de instancias particulares.

Para completar la interacción se necesita especificar mensajes que se envían entre las líneas de vida.

Mensajes

Representan un tipo específico de comunicación entre dos líneas de vida en una interacción.

Esta comunicación puede implicar:

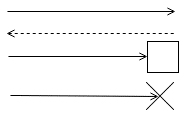
* Invocar una operación mediante un mensaje de llamada.
* Crear o destruir una instancia mediante un mensaje de creación/destrucción.
* Enviar una señal.

Cuando una línea de vida recibe un mensaje de llamada, éste es una petición para la invocación de una operación que tiene la misma firma que el mensaje.

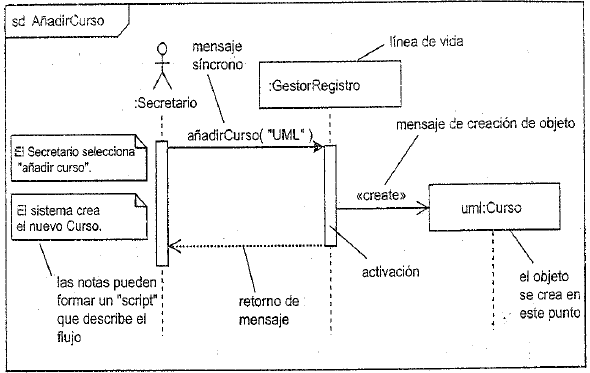
Cuando una línea de vida ejecuta un mensaje, tiene foco de control o activación.

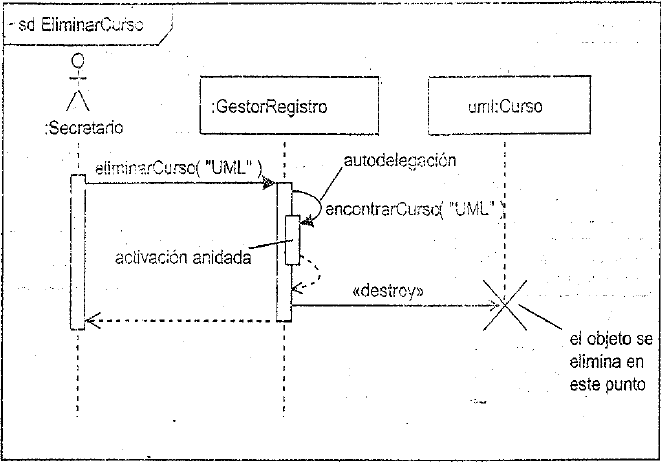
**Flujo de control**: movimiento de la activación entre las líneas de vida a medida que progresa la interacción con el tiempo.

Tipos de mensajes

* Envío de Mensaje
  + Síncrono: el emisor espera a que el receptor acabe de ejecutar la operación solicitada. Indican una secuenciación estricta de llamadas de operación.
  + Asíncrono: el emisor no espera, continúa inmediatamente con el siguiente paso. Posibilidad de concurrencia.
* Retorno de mensaje
* Creación de objeto: Puede mostrar creación de objetos con sólo enviar un mensaje estereotipado <<create>>.
* Destrucción de objeto: La instancia de clasificador a la que hace referencia la línea de vida destino ya no se encuentra disponible para su uso.

En análisis no es importante detallar el tipo de envío de mensaje, sino que el mensaje se envía. Generalmente se muestran todos los mensajes como síncronos. En diseño si se distinguen los tipos de mensajes para poder diseñar los flujos concurrentes de control.

Ejemplo Diagrama de Secuencia



Activaciones

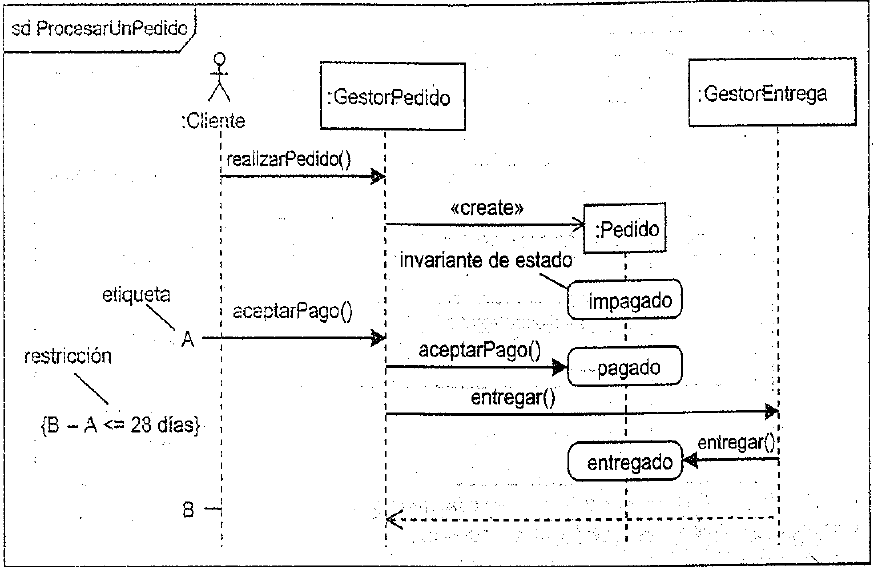
Se pueden poner rectángulos largos en la línea discontinua debajo de la línea de vida para indicar cuándo tiene el foco de control o activación.

Un objeto empieza con el foco e invoca una operación sobre otro objeto anidando el foco de control. Este objeto puede invocar una operación sobre otro objeto anidando aún más el foco de control.

**Autodelegación**: una línea de vida se envía un mensaje a sí misma. Crea una activación anidada. Son operaciones privadas de un objeto.

Invariantes de estado y restricciones

Cuando una instancia recibe un mensaje, puede hacer que cambie de estado. En el diagrama se puede mostrar el estado de las instancias usando invariantes de estado.

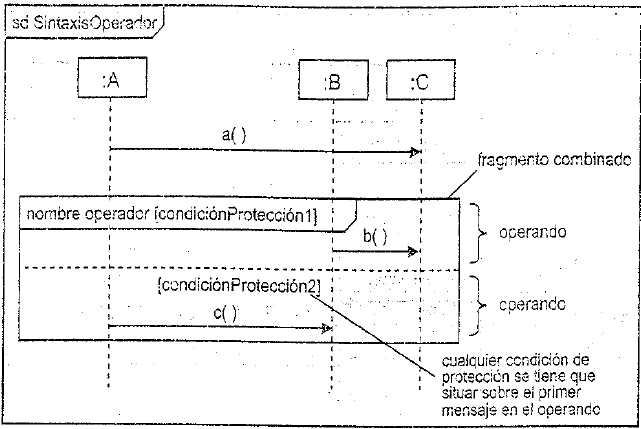


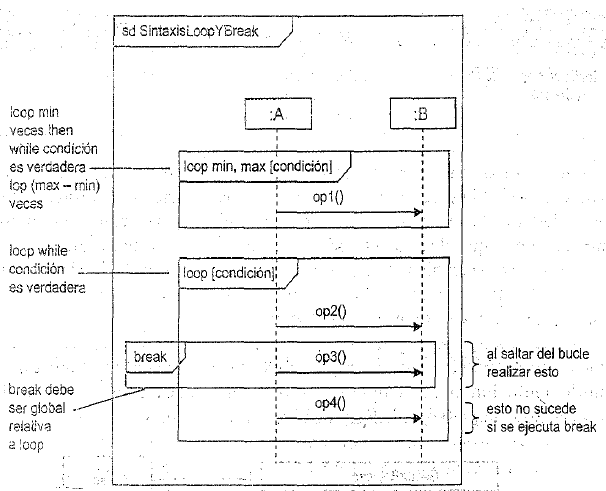
Una restricción sobre la línea de vida indica algo que debe ser verdadero sobre instancias desde ese punto en adelante. Se puede indicar cualquier tipo de restricción. Se debe escribir entre llaves.

Fragmentos combinados y operadores

Los diagramas de secuencia se pueden dividir en áreas denominadas fragmentos combinados. Todo fragmento combinado tiene un operador, uno o más operandos y cero o más condiciones de protección. El operador determina cómo se ejecutan sus operandos.

Las condiciones de protección son expresiones booleanas que determinan si sus operandos se ejecutan. Una sola condición de protección se puede aplicar a todos los operandos, o cada operando puede tener su propia condición de protección única.





Lista de operadores

|  |  |  |
| --- | --- | --- |
| **Operador** | **Nombre** | **Descripción** |
| Opt | Opción | Existe un solo operando que se ejecuta si la condición es verdadera. (if-then) |
| Alt | Alternativas | Se ejecuta el operando cuya condición es verdadera. (if-else) |
| Loop | Bucle | Loop min, max [condición] |
| Break | Saltar | Si la condición de protección es verdadera, el operando se ejecuta, no el resto de la interacción que engloba. |
| Ref | Referencia | El fragmento combinado hace referencia a otra interacción. |
| Par | Paralelo | Todos los operando se ejecutan en paralelo |
| Critical | Crítico | El operando se ejecuta sin interrupción |
| Seq | Secuenciación débil | Todos los operandos se ejecutan en paralelo sujetos a la siguiente restricción: los eventos que lleguen en la misma línea de vida de diferentes operandos, ocurren en la misma secuencia que los operandos. Esto da lugar a una forma de secuenciación débil. |
| Strict | Secuenciación estricta | Los operando se ejecutan en secuencia estricta. |
| Neg | Negativa | El operando muestra interacciones inválidas. Para cuando se quiera mostrar interacciones que deben no suceder. |
| Ignore | Ignorar | Lista mensajes que se omiten intencionadamente de la interacción los nombres de los mensajes ignorados están situados entre llaves en una lista delimitada por comas después del nombre del operador, como {m1, m2, m3). |
| Consider | considerar | Lista mensajes que se han incluido intencionadamente en la interacción. Los nombres están entre llaves separados por coma. |
| Assert | Aserción | El operando es el único comportamiento valido en ese punto en la interacción y cualquier otro comportamiento tendría un error. Utilizar como forma de indicar que cierto comportamiento debe ocurrir en cierto momento. |

Tabla de bucles

|  |  |  |
| --- | --- | --- |
| **Tipo de bucle** | **Semántica** | **Expresión de bucle** |
| while(true)  {cuerpo} | Pasar en bucle sin fin. | loop o loop \* |
| for i = n to m  {cuerpo} | Repetir (m-n) veces. | loop n, m |
| while(expresiónBooleana)  {cuerpo} | Repetir mientras expresiónBooleana es verdadera. | loop [expresiónBooleana] |
| repeat  {cuerpo}  while(expresiónBooleana) | Ejecutar una vez luego repetir mientras expresiónBooleana es verdadera. | loop 1, \* [expresiónBooleana] |
| forEach objeto en colección  {cuerpo} | Ejecutar el cuerpo del bucle una vez para cada objeto en una colección de objetos. | Loop [para cada objeto en colección] |
| forEach objeto de clase  {cuerpo} | Ejecutar el cuerpo del bucle una vez para cada objeto de una clase determinada. | Loop [para cada objeto en NombreClase] |

**Diagramas de comunicación**

Enfatizan los aspectos estructurales de una interacción, es decir, cómo se conectan las líneas de vida.

La sintaxis es similar a los de secuencia, excepto que las líneas de vida no son discontinuas y se conectan mediante vínculos que proporcionan canales de comunicación para los mensajes. Además, se suele numerar los mensajes para indicar cuál es la secuencia y anidamiento dentro de otros mensajes.

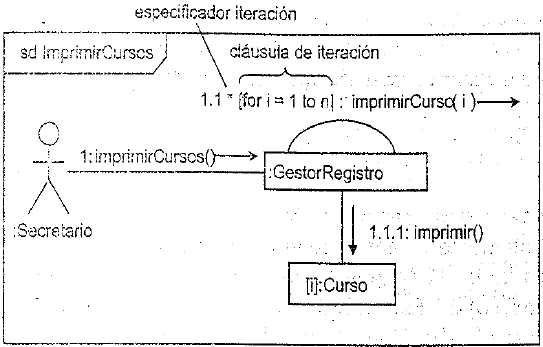
Iteración en diagramas de comunicación

En este tipo de diagramas también puede incluir iteraciones al utilizar una expresión de iteración. Esto implica un especificador de iteración (\*) y una cláusula de iteración (opcional).

Iteración en paralelo 🡪 \*//

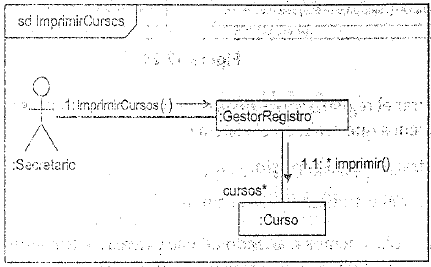
Nota: UML 2 no prescribe ninguna sintaxis en particular para cláusulas de iteración, por lo tanto, se puede utilizar cualquier cosa que tenga sentido. Por lo general, el código o pseudocódigo es una buena opción.

Un buen enfoque podría ser: \*[loop min, max(condición)]



En este ejemplo, se utiliza i como un selector para una instancia Curso específica a la que se envía el mensaje imprimir().

Este enfoque asume que las instancias Curso están almacenadas en algún tipo de colección indexada. Si no quiere realizar esa asunción, puede utilizar el enfoque alternativo.

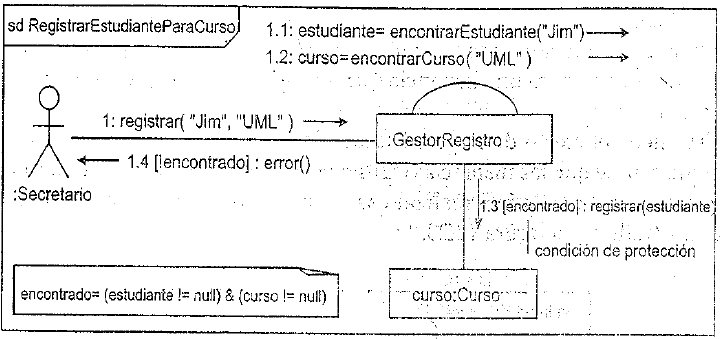


En este enfoque, hemos mostrado el nombre de rol y multiplicidad. Esto indica que :GestorRegistro está conectado a una colección de objetos :Curso por medio del nombre de rol cursos.

Además, hemos añadido el especificador de iteración \* al mensaje imprimir indicando que este mensaje se envía a cada objeto en la colección.

Ramificación

Se puede modelar ramificación al añadir condiciones de protección a los mensajes. Éstos, solamente se enviarán cuando la condición de protección evalúa como verdadero.



Este registro consta de tres pasos:

1. Encontrar el registro correcto de estudiante, no podemos registrar a estudiantes a menos que estén en el sistema.
2. Encontrar el curso correcto.
3. Registrar al estudiante para el curso.

Las condiciones no tienen una sintaxis formal, pero son expresiones que implican variables temporales o atributos de las clases implicadas en la iteración.

Nota: usar ramificación solamente en diagramas sencillos. Suelen volverse complejos rápidamente en diagramas de comunicación (en los de secuencia es más sencillo presentarlas).

**Pressman - Capitulo 8 – Conceptos de Diseño**

Diseño de SW: Proceso a través del cual los requerimientos son trasladados a una representación o modelo para construir el SW. Agrupa el conjunto de principios, conceptos y prácticas que llevan al desarrollo de un sistema o producto de alta calidad.

Comienza luego de analizar y modelar los requerimientos, es la última acción de la ingeniería de SW dentro de la actividad de modelado y prepara la etapa de construcción (generación y prueba de código).

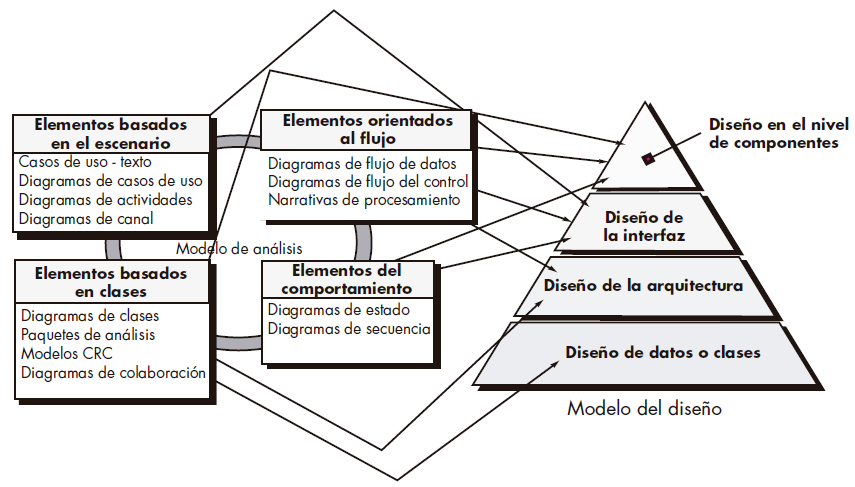
A diferencia del modelo de los requerimientos, proporciona detalles sobre arquitectura del SW, estructuras de datos, interfaces y componentes que se necesitan para implementar el sistema.

Representaciones

El diseño es un proceso iterativo por medio del cual se traducen los requerimientos en un “plano” para construir el SW. El diseño comienza con una representación en un nivel alto de abstracción, pero luego, a medida que tienen lugar las iteraciones, las mejoras posteriores conducen a niveles menores de abstracción.

* Se comienza desde un alto nivel de abstracción (arquitectura).
* Nivel intermedio (patrones, diseño de programas, UI, Datos, etc.).
* Se termina en niveles bajos de abstracción (código, datos).

Del Análisis al Diseño



Representa el flujo de la información durante el diseño del SW. El trabajo de diseño es alimentado por el modelo de requerimientos, manifestado por elementos basados en el escenario, en la clase, orientados al flujo, y del comportamiento. El empleo de la notación y de los métodos de diseño produce diseños de los datos o clases, de la arquitectura, de la interfaz y de los componentes.

* El diseño de datos o clases transforma los modelos de clases en realizaciones de clases de diseño y en las estructuras de datos que se requieren para implementar el SW.
* El diseño de la arquitectura define la relación entre los elementos principales de la estructura del SW, los estilos y patrones de diseño de la arquitectura que pueden usarse para alcanzar los requerimientos definidos por el sistema y las restricciones que afectan la forma en la que se implementa la arquitectura. La representación del diseño de la arquitectura se obtiene del modelo de los requerimientos.
* El diseño de la interfaz describe la forma en la que el SW se comunica con los sistemas que interactúan con él y con los humanos que lo utilizan. Los modelos de escenarios de uso y de comportamiento dan mucha de la información requerida para diseñar la interfaz.
* El diseño en el nivel de componente transforma los elementos estructurales de la arquitectura del SW en una descripción de sus componentes en cuanto a procedimiento. La información obtenida a partir de los modelos basados en clase, flujo y comportamiento sirve como la base para diseñar los componentes.

¿Por qué es importante el Diseño?

Permite modelar el sistema. Es el lugar en el que se establece o introduce la calidad del SW. Un diseño de alta calidad es necesario para alcanzar productos de alta calidad.

¿Qué caracteriza un diseño de alta calidad? ¿Cómo se alcanza? ¿Qué caracteriza una actividad de diseño? ¿Cuál es el producto de diseño?

**Proceso de Diseño**

Características de diseño

* Debe implementar todos los requerimientos explícitos contenidos en el modelo de requerimientos y dar cabida a todos los requerimientos implícitos que desean los participantes.
* Debe ser una guía legible y comprensible para quienes generan el código y para los que lo prueban y dan el apoyo posterior.
* Debe proporcionar el panorama completo del SW, y abordar los dominios de los datos, las funciones y el comportamiento desde el punto de vista de implementación.

Para lograr estas metas de diseño se deben aplicar lineamientos y atributos de calidad.

Lineamientos de calidad de Diseño

1. Debe exhibir una arquitectura.
2. Debe ser modular, es decir, el SW debe estar dividido de manera lógica en elementos o subsistemas.
3. Debe contener distintas representaciones para los datos, arquitecturas, interfaces y componentes.
4. Debe conducir a estructuras de datos apropiadas para las clases que se van a implementar.
5. Debe llevar a componentes funcionalmente independientes (cohesión).
6. Debe llevar a interfaces simples, que reduzcan la complejidad de conexiones entre los componentes y el ambiente externo (acoplamiento).
7. Debe ser realizado utilizando un método repetible y motivado por la información obtenida durante el análisis.
8. Debe ser representado utilizando una notación que efectivamente comunique su significado.

Atributos de calidad

FURPS: funcionalidad, usabilidad, confiabilidad, rendimiento y mantenibilidad. Representan el objetivo de todo diseño de software:

* La funcionalidad se califica de acuerdo con el conjunto de características y capacidades del programa, la generalidad de las funciones que se entregan y la seguridad general del sistema.
* La usabilidad se evalúa tomando en cuenta factores humanos, la estética general, la consistencia y la documentación.
* La confiabilidad se evalúa con la medición de la frecuencia y gravedad de las fallas, la exactitud de los resultados que salen, el tiempo medio para que ocurra una falla (TMPF), la capacidad de recuperación ante ésta y lo predecible del programa.
* El rendimiento se mide con base en la velocidad de procesamiento, el tiempo de respuesta, el uso de recursos, el conjunto y la eficiencia.
* La mantenibilidad combina la capacidad del programa para ser ampliable (extensibilidad), adaptable y servicial, y además que pueda probarse, ser compatible y configurable y que cuente con la facilidad para instalarse en el sistema y para que se detecten los problemas.

Función, Forma y Fabricación

* Un sistema de SW se construye basado en una especificación de su forma, la que satisface una función.
* **Forma:** especificación de diferentes atributos de calidad que a través de la realización en componentes satisfacen la función.
* **Función:** necesidad, propósito, utilidad o uso pretendido del sistema o aplicación.
* **Diseño (arquitectónico):** une la función y la forma.

Forma y Diseño de Interacción

* Puede distinguirse entre el diseño que directamente afecta al usuario final (**diseño de interacción**) del diseño de la aplicación (**diseño de programa**).
* El diseño de interacción pobre contribuye a la ‘fricción cognitiva’: “la resistencia encontrada por el intelecto humano cuando se enfrenta con un sistema complejo de reglas que cambian a la vez que el problema cambia”.
* La interacción con aplicaciones de SW es alta en fricción cognitiva, debido a la gran cantidad de estados que el sistema puede tener, y en base a esto reaccionar.
* Campo de investigación de Interacción Humanos/Computadoras (HCI).

Construcción

* La calidad de un sistema está basada no solo en el diseño, sino también en la elección de la tecnología utilizada en la implementación.
* Un diseño de SW debe ser realizable:
  + Debe ser posible aplicar las prácticas y tecnologías de ingenierías disponibles.
  + Debe ser posible construir el sistema dada la disponibilidad de recursos: tiempo, personal, costos, sistemas existentes (legacy) y componentes.

Problemas, Obstáculos y Soluciones

El diseño es una actividad de encontrar o crear soluciones a problemas dado un conjunto de obstáculos a sobrepasar.

* Problemas ‑> Soluciones (pocas veces sucede).
* Problemas -> Obstáculos -> Soluciones (es lo que mayormente se da).

“El principio de la sabiduría, para un ingeniero de SW, es reconocer la diferencia que hay entre hacer que un programa funcione y lograr que lo haga bien”

**Conceptos de Diseño**

Abstracción

* “concentrarse en un problema al mismo nivel de generalización, independientemente de los detalles irrelevantes de bajo nivel; el uso de la abstracción también permite trabajar con conceptos y términos que son familiares al entorno del problema, sin tener que transformarlos a una estructura no familiar...”
* Ejemplo:
  + **Abstracción de procedimiento:** secuencia de instrucciones que tienen una función específica y limitada. Ejemplo: abrir.
  + **Abstracción de datos:** conjunto de éstos con nombre que describe a un objeto de datos. Ejemplo: ventana.

Arquitectura

“Es la estructura general del SW y las formas en las que ésta da integridad conceptual a un sistema”.

Conjunto de propiedades que deben especificarse como parte del diseño de la arquitectura:

* **Propiedades estructurales**: Define los componentes de un sistema (módulos, objetos, filtros, etc.) y la manera en la que están agrupados e interactúan.
* **Propiedades extrafuncionales**: Descripción de la forma en la que la arquitectura satisface los requerimientos de desempeño, capacidad, confiabilidad, seguridad y adaptabilidad, así como otras características del sistema.
* **Familias de sistemas relacionados**: El diseño arquitectónico debe basarse en patrones repetibles que es común encontrar en el diseño de familias de sistemas similares. En esencia, el diseño debe tener la capacidad de reutilizar bloques de construcción arquitectónica.

Patrones

Describe una estructura de diseño que resuelve un problema particular del diseño dentro de un contexto específico y entre “fuerzas” que afectan la manera en la que se aplica y en la que se utiliza dicho patrón.

El objetivo de cada patrón de diseño es proporcionar una descripción que permita a un diseñador determinar si el patrón es aplicable al trabajo en cuestión, si puede volverse a usar (con lo que se ahorra tiempo de diseño) y si sirve como guía para desarrollar un patrón distinto en funciones o estructura.

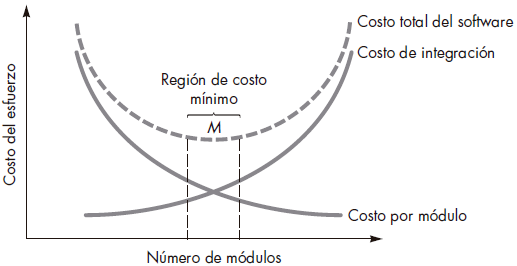
División de problemas

Sugiere que cualquier problema complejo puede manejarse con más facilidad si se subdivide en elementos susceptibles de resolverse u optimizarse de manera independiente. Lleva más tiempo y esfuerzo resolver un único problema complejo.

* (E = esfuerzo requerido)

Modularidad

El SW se divide en componentes con nombres distintos y abordables por separado (módulos), que se integran para satisfacer los requerimientos del problema.



El esfuerzo (costo) de desarrollar un módulo individual de SW disminuye conforme aumenta el número total de módulos. Dado el mismo conjunto de requerimientos, tener más módulos significa tamaños individuales más pequeños. Sin embargo, a medida que se incrementa el número de módulos, el esfuerzo (costo) asociado con su integración también aumenta.

Existe un número M de módulos que arrojarían el mínimo costo de desarrollo, pero no se dispone de las herramientas necesarias para predecir M con exactitud.

Deben aplicarse módulos para que:

* El desarrollo pueda planearse con más facilidad
* Sea posible definir y desarrollar los incrementos del SW
* Los cambios se realicen con más facilidad
* Las pruebas y la depuración se efectúen con mayor eficiencia
* El mantenimiento a largo plazo se lleve a cabo sin efectos colaterales de importancia.

Ocultamiento de Información

* Sugiere que los módulos se “caractericen por decisiones de diseño que se oculten (cada una) de las demás”.
* Deben especificarse y diseñarse módulos, de forma que la información (algoritmos y datos) contenida en un módulo sea inaccesible para los que no necesiten de ella.
* Ventaja: Debido a que la mayoría de los datos y detalles del procedimiento quedan ocultos para otras partes del SW, es menos probable que los errores inadvertidos introducidos durante la modificación se propaguen a distintos sitios dentro del SW.

Independencia Funcional

* Es resultado directo de la separación de problemas y de los conceptos de abstracción y ocultamiento de información.
* Debe diseñarse el SW de manera que cada módulo resuelva un subconjunto específico de requerimientos y tenga una interfaz sencilla cuando se vea desde otras partes de la estructura del programa.
* **Importancia:** la independencia funcional es una clave para el buen diseño y éste es la clave de la calidad del SW.
* La independencia se evalúa con el uso de dos criterios cualitativos:
  + La cohesión es un indicador de la fortaleza relativa funcional de un módulo. Un módulo cohesivo ejecuta una sola tarea, por lo que requiere interactuar poco con otros componentes en otras partes del programa. En pocas palabras, un módulo cohesivo debe (idealmente) hacer sólo una cosa.
  + El acoplamiento es una indicación de la interconexión entre módulos en una estructura de SW, y depende de la complejidad de la interfaz entre módulos, del grado en el que se entra o se hace referencia a un módulo y de qué datos pasan a través de la interfaz. Debe buscarse siempre el mínimo acoplamiento posible.

La conectividad simple entre módulos da como resultado un SW que es más fácil de entender y menos propenso a la propagación de errores.

Refinamiento

Un programa se elabora por medio del refinamiento sucesivo de los detalles del procedimiento. Se desarrolla una jerarquía con la descomposición de un enunciado macroscópico de la función (abstracción del procedimiento) en forma escalonada hasta llegar a los comandos del lenguaje de programación.

Es un proceso de elaboración. Se comienza con un enunciado de un alto nivel de abstracción. Después se trabaja sobre el enunciado original, dando más y más detalles conforme tiene lugar el refinamiento (elaboración) sucesivo.

Rediseño

Actividad sugerida por muchos métodos ágiles y definida como una técnica de reorganización que simplifica el diseño (o código) de un componente sin cambiar su función o comportamiento.

“Es el proceso de cambiar un sistema de SW en forma tal que no se altera el comportamiento externo del código [diseño], pero sí se mejora su estructura interna.”

Cuando se rediseña el SW, se examina el diseño existente en busca de redundancias, elementos de diseño no utilizados, algoritmos ineficientes o innecesarios, estructuras de datos mal construidas o inapropiadas y cualquier otra falla del diseño que pueda corregirse para obtener un diseño mejor.

Resumen

* Una solución al problema de diseño de SW (en una manera simplificada) es la cuidadosa planificación y ejecución sistemática de un proceso de diseño y la producción de artefactos de diseño (modelos y especificaciones).
* El diseño es predominantemente creativo (resolución de problemas, dado un conjunto de obstáculos).
* No hay un método de diseño que sea superior.
* La función requerida de una aplicación es expresada como un problema y, a través del análisis, es dividido en subproblemas menos complejos y más manejables.
* **Diseño Funcional**: diseño de la faceta externa o diseño de interacción. Aspecto percibido por los usuarios.
* **Diseño Estructural**: diseño físico interno del SW. Los métodos de diseño se aplican para descubrir nuevas soluciones y evaluar soluciones existentes.
* **Diseño Detallado**: involucra la especificación del SW para las actividades posteriores de implementación y testing. Debe realizarse en diferentes niveles de abstracción.
* El SW es inherentemente complejo.

**Atributos de Calidad de SW**

Conforman lo que se denomina comúnmente requerimientos no funcionales. Son los requerimientos sobre la forma que debe tener el SW. Condicionan la manera en que las funciones serán llevadas a cabo por el SW.

Categorías de “calidades” de un sistema:

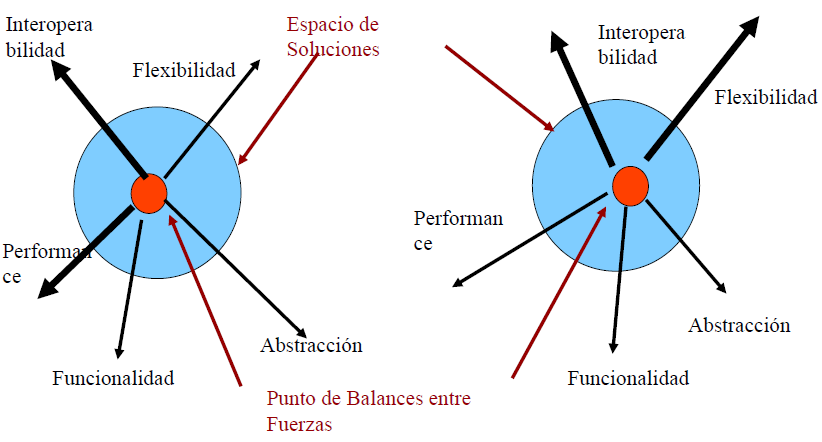
* **Calidad en** **tiempo de ejecución**: pueden ser medidas cuando el sistema ejecuta.
  + ¿Qué tan bien satisface el sistema sus requerimientos de comportamiento?
  + ¿Provee los resultados requeridos?
  + ¿Los provee en el tiempo esperado?
  + ¿Son los resultados correctos o dentro de las tolerancias permitidas?
  + ¿Son las funciones del sistema las deseadas cuando se conecta a otros sistemas?

Estas cuestiones se resuelven en la etapa de diseño.

* **Calidades estáticas**: no pueden observarse cuando el sistema ejecuta.
  + ¿Qué tan fácil es el sistema para integrar, testear o modificar?
* **Calidad de negocio**
  + ¿Qué tan costoso será/fue desarrollarlo?
  + ¿En cuánto tiempo estará operativo?
* **Calidad del diseño/arquitectura**

Interrelaciones

* Alcanzar los atributos de una categoría puede no decir nada acerca de los atributos de otra categoría.
  + Ejemplo: un sistema que alcanza todos sus requerimientos en tiempo de ejecución puede o no ser costoso de desarrollar, o puede o no ser imposible de modificar.
  + Por otra parte, un sistema altamente modificable puede o no producir resultados correctos.
* Alcanzar los atributos dentro de una categoría no dice nada acerca de otros atributos dentro de la misma categoría.
  + Por ejemplo, un sistema que es construido en dos semanas puede tomar años para modificar.
* Alcanzar un atributo de calidad **nunca** puede hacerse de forma aislada.
* Alcanzar cualquier atributo de calidad puede tener un efecto positivo o negativo sobre otros atributos de calidad.
  + Seguridad y tolerancia a fallas existen en un estado de tensión mutua.
    - Un sistema seguro tiene pocos puntos de falla, un kernel de seguridad -> seguro, pero no tolerante.
    - Un sistema tolerante a fallas tiene más puntos de falla. Conjuntos de procesos o procesadores redundantes 🡪 tolerante, pero no seguro.
  + Confiabilidad y performance.
    - Replicar unidades de procesamiento y canales de comunicación 🡪 confiabilidad. Conflicto con performance (ej.: consume más tiempo).
    - Unir procesos críticos para alcanzar performance 🡪 conflicto con confiabilidad (punto de falla simple).



La calidad debe ser considerada en todas las fases de diseño, implementación y distribución. Diferentes cualidades se manifiestan en forma diferente durante estas fases.

Consideraciones:

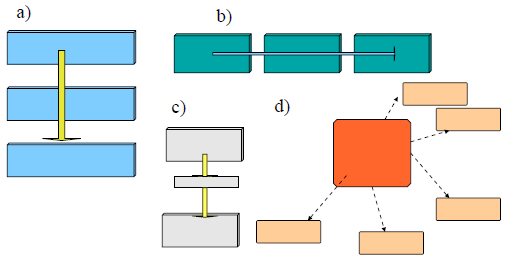
* La arquitectura es crítica para alcanzar muchas de las cualidades de interés, y estas cualidades deben ser diseñadas y evaluadas a nivel arquitectónico.
* Algunas cualidades no son sensitivamente arquitectónicas, e intentar alcanzarlas a través de la arquitectura será infructuoso.

Ejemplos

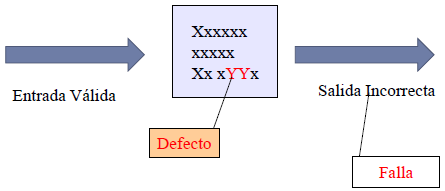
* Usabilidad
  + Hacer la interface de usuario clara y fácil de usar 🡪 no esun problema arquitectónico (**sí** de diseño).
  + Si el sistema debe proveer la capacidad de cancelar las operaciones, deshacer operaciones, o reutilizar datos previamente ingresados 🡪 es un problema arquitectónico.
* Modificabilidad
  + Cómo se divide la funcionalidad en módulos -> Arquitectónico
  + La técnica de codificación de módulos 🡪 No arquitectónico
* Performance
  + Cuánta comunicación existe entre los componentes 🡪 Arquitectónico
  + Cómo son asignados los recursos compartidos 🡪 Arquitectónico
  + La elección de un algoritmo 🡪 No arquitectónico

Atributos de Calidad - Cualidades en tiempo de ejecución: sobre la Funcionalidad

* **Funcionalidad:**
  + Habilidad del sistema de hacer el trabajo para el cual fue creado.
  + Se alcanza a través de la interacción, cooperación y sincronización de los componentes.
  + Funcionalidad es ortogonal al diseño de SW (arquitectura).
  + El diseño del SW al considerar otros atributos de calidad importantes 🡪 condiciona la asignación de funcionalidad a la estructura.
  + Funcionalidad puede ser alcanzada a través de diferentes variantes de estructuras posibles.
  + Funcionalidad es independiente de la arquitectura.



* **Disponibilidad (Availability):**
  + La medida del tiempo en que el sistema está operativo y ejecutando correctamente.
  + Analiza las fallas del sistema y sus consecuencias.
  + Una falla del sistema ocurre cuando este no entrega más un servicio consistente con sus especificaciones.
  + Esta falla es observable por los usuarios del sistema (humano u otros sistemas).
  + Es necesario distinguir entre falla y defecto. La falla es observable, es lo que el usuario ve. El defecto se encuentra en el código. Se dice que hay un defecto latente hasta que se produce una salida incorrecta: la falla.



* + Un defecto llega a ser falla si no es corregido u ocultado.
  + Una falla es observable por el usuario del sistema, un defecto no.
  + Una vez que el sistema falló, es importante el tiempo necesario para repararlo.
  + **Tiempo de reparación**: tiempo hasta que la falla no es más observable.
* **Modificabilidad**: facilidad con la cual el SW puede adecuarse a los cambios.
  + Cualidad estática.
  + Acerca del costo del cambio.
  + Problema netamente arquitectónico: afectado por las características de los componentes de SW.
  + Modificabilidad Mantenibilidad (Maintainability)
  + ¿Qué puede ser cambiado (artefacto)?
    - Cualquier aspecto del sistema puede ser cambiado.
    - Comúnmente las funciones que el sistema realiza.
    - La plataforma del sistema (HW, SO, middleware, etc.) 🡪 Portabilidad.
    - El entorno de operación (los sistemas con los que debe interactuar, protocolos de comunicación, etc.)
    - Las cualidades que el sistema exhibe (performance, fiabilidad, etc.)
    - Su capacidad (cantidad de usuarios soportados, cantidad de operaciones simultáneas, etc.)
    - Los cambios pueden ser Agregar, Eliminar, o Modificar.
  + ¿Cuándo se realiza el cambio y quién lo hace (el entorno)?
    - Cambios durante la implementación (modificando el código fuente).
    - Durante la compilación (utilizando switches en el compilador).
    - Durante la construcción (mediante la elección de librerías).
    - Durante el setup de configuración (ejemplo: seteo de parámetros).
    - Durante la ejecución (seteo de parámetros).
    - Un cambio puede ser realizado por un desarrollador, un usuario final o un administrador de sistema.
  + Categoría de cambios:
    - Extender o cambiar capacidades: agregar nuevas funcionalidades, extender funcionalidades existentes o reparar “bugs”.
      * Extensibilidad: capacidad de adquirir nuevas funcionalidades.
    - Eliminar capacidades no deseadas: simplificar las funcionalidades de un sistema (sistema menos capaz y menos costoso).
    - Adaptarse a nuevos entornos operativos: nuevos procesadores, dispositivos de entrada/salida, dispositivos lógicos, etc. 🡪Portabilidad.
    - Reestructuración: reasignación de servicios del sistema, modularización optimización o creación de componentes reusables.
  + Una vez que el cambio ha sido especificado, debe ser diseñado, implementado, testeado y distribuido. Toma tiempo y dinero, lo que puede ser medido.
  + Portabilidad
    - La habilidad del sistema para ejecutar en diferentes entornos de computación (hardware, SW o ambos).
    - Un sistema es portable cuando logra colocar todas las relaciones con un entorno operativo particular en una componente o un conjunto pequeño de componentes fácilmente intercambiables.
    - Un tipo especial de modificabilidad.
    - Medida: cuánto cuesta en términos de tiempo y dinero portarlo.
* **Performance**
  + Se refiere a las respuestas del sistema, ya sea el tiempo requerido para responder a eventos específicos o la cantidad de eventos procesados en un intervalo de tiempo dado.
  + Normalmente expresado por la cantidad de transacciones por unidad de tiempo o por el tiempo que toma completar una transacción con el sistema.
  + Eventos (interrupciones, mensajes, requerimientos de usuarios, o el paso del tiempo): ocurren y el sistema debe responder a estos.
    - Performance está relacionado con cuánto tiempo le toma al sistema responder cuando un evento ocurre.
    - Las diferentes fuentes de los eventos (usuarios, otros sistemas o dentro del mismo sistema) y los patrones de arribo → Complican el problema de performance.
    - No se ha considerado hasta aquí si el sistema es en red o simple usuario (standalone).
    - No se consideró la configuración del sistema, ni la forma en que se consumen los recursos.
    - Comunicación toma más tiempo que los cálculos.
    - **Performance es una función de cuanta comunicación e interacción hay entre las componentes** (problema arquitectónico).
* **Usabilidad**
  + Qué tan fácil es para un usuario realizar una tarea requerida (Facilidad de uso), y la clase de soporte a usuarios que el sistema provee (entrenamiento y ayuda).
  + Aspectos de usabilidad:
    - **Aprendizaje**: ¿Qué tan fácil y rápido es para el usuario aprender a utilizar el sistema? ¿Qué puede hacer el sistema para hacer el aprendizaje más fácil?
    - **Eficiencia**: ¿el SW responde a la velocidad apropiada a las actividades del usuario? ¿Qué puede hacer el sistema para que el usuario haga en forma más eficiente su trabajo?
    - **Minimizar el impacto de errores**: ¿El sistema anticipa y previene errores comunes de usuarios? ¿Puede el sistema hacer que el impacto de un error sea mínimo?
    - **Manejo de errores:** ¿El sistema ayuda al usuario a recuperar errores?
    - **Adaptar el sistema a las necesidades del usuario:** ¿Cómo puede el usuario (o el sistema mismo) adaptarse para hacer las tareas del usuario más fácil?
    - **Asistencia:** ¿El sistema se comunica bien y asiste al usuario en la resolución de problemas?
    - **Incrementar confianza y satisfacción:** ¿Qué hace el sistema para dar al usuario confianza que las acciones correctas están siendo realizadas?

**UML 2 Capitulo 14 – Diagramas de actividad**

Los diagramas de actividad son diagramas de flujo orientados a objetos que permiten modelar un proceso como una actividad que consta de una colección de nodos conectados por extremos. En UML 2, tienen una semántica completamente nueva basada en Redes de Petri.

Las actividades se anexan a: CU, Clases, Interfaces, Componentes, Colaboraciones y Operaciones. También se pueden utilizar para modelar procesos de negocio y flujos de trabajo.

La esencia de un buen diagrama de actividad es que está centrado en comunicar un aspecto específico de un comportamiento dinámico de un sistema.

Una actividad se puede anexar a cualquier elemento de modelado con la finalidad de modelar su comportamiento. En la práctica, se utilizan más comúnmente en las siguientes formas:

* En el workflow de análisis:
  + Para modelar el flujo en un CU de una forma gráfica que es fácil de entender para los grupos de decisión.
  + Para modelar el flujo entre CU (utiliza una forma especial de estos diagramas denominado diagrama de visión de interacción).
* En diseño:
  + Para modelar los detalles de una operación
  + Para modelar los detalles de un algoritmo.
* En el modelado de negocio:
  + Para modelar un proceso de negocio.

Actividades

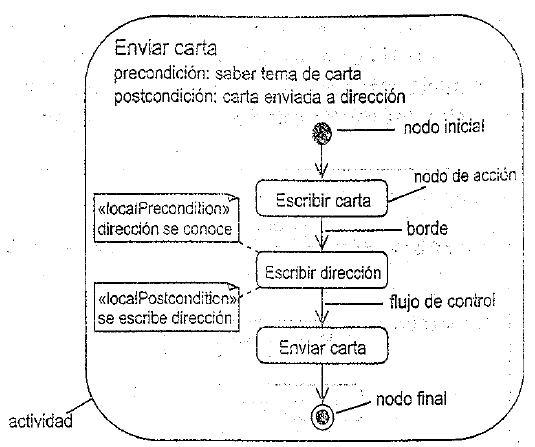
Son redes de nodos conectados por extremos. Existen 3 categorías:

* **Nodos de acción**: representan unidades de trabajo que son atómicas dentro de la actividad (Rectángulo con puntas redondeadas).
* **Nodos de control**: controlan el flujo por medio de la actividad.
* **Nodos de objeto**: representan objetos utilizados en la actividad (Rectángulo con puntas rectas).

Los extremos (flechas) representan flujo por la actividad. Existen 2 categorías:

* **Flujos de control**: representan el flujo de control por medio de la actividad.
* **Flujos de objeto**: representan el flujo de objeto por medio de la actividad.

Las actividades pueden tener precondiciones y postcondiciones (las acciones dentro de la actividad pueden tener también sus propias precondiciones y postcondiciones locales). Las actividades comienzan con un nodo inicial y pueden terminar con uno o más nodos finales.

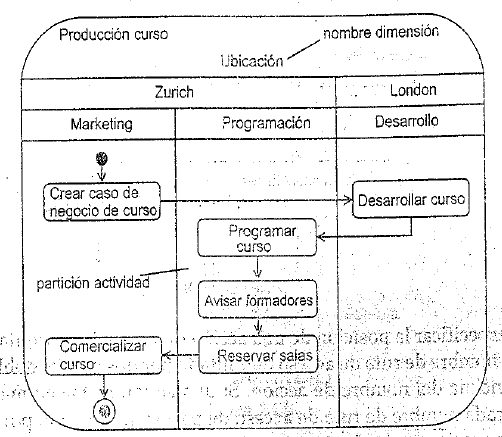


Un uso común de los diagramas de actividad es modelar un CU como una serie de acciones. Los CU expresan comportamiento del sistema como una interacción entre un actor y el sistema, mientras que los diagramas de actividad lo expresan como una serie de acciones. Son vistas complementarias del mismo comportamiento.

Semántica de actividad

Los diagramas de actividad modelan comportamiento utilizando el flujo de tokens. Los tokens fluyen alrededor de la red y pueden representar: el flujo de control, algunos datos o un objeto.

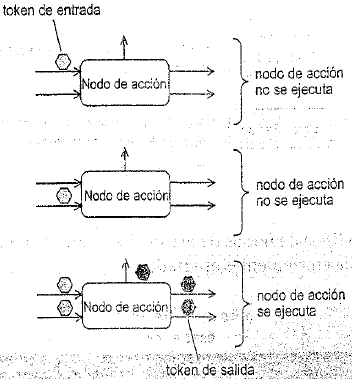
El estado del sistema en cualquier punto en el tiempo está determinado por la disposición de sus tokens.

Particiones de actividad

Se pueden dividir las actividades en particiones para hacer más legible el diagrama. Se utilizan líneas verticales, horizontales o curvas. Cada partición de actividad representa una agrupación de alto nivel de acciones relacionadas.

Se utilizan comúnmente para representar: CU, clases, componentes, unidades organizativas, roles, etc.

Si se necesita mostrar cómo el sistema interactúa con algún otro sistema externo, agregar el estereotipo <<external>> encima del nombre de partición.

Nodos de acción

Son unidades de trabajo atómicas.

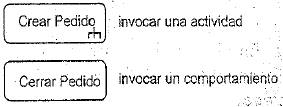
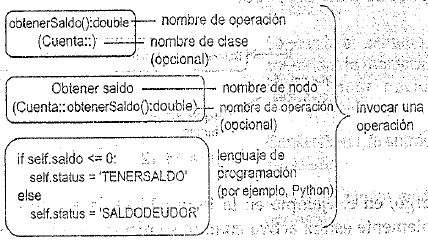
Se ejecutan cuando existe un token simultáneamente en cada uno de sus extremos de entrada AND y se cumplen todas sus precondiciones locales.

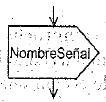
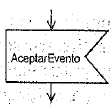
Después de ejecutarse, los nodos de acción ofrecen tokens simultáneamente en todos los extremos de salida cuyas postcondiciones se cumplen. Esto es un fork implícito ya que un nodo de acción puede dar lugar a muchos flujos. A diferencia de los diagramas de flujo convencionales, los diagramas de actividad son concurrentes intrínsecamente.

Puesto que los nodos de acción realizan algo, normalmente se nombran con un verbo o frase verbal.

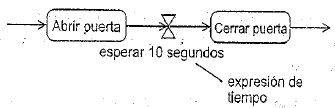
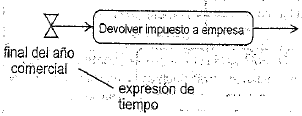
Existen 4 tipo de acción

* **Nodo de acción de llamada**: Es el tipo más común y puede invocar una actividad, un comportamiento o una operación. Puede enviar un nodo de acción de señal, nodo de acción de aceptar evento o nodo de acción de aceptar evento de tiempo.



* Enviar señal
* Nodo de acción de aceptar evento.
* **Nodo de acción de aceptar evento de tiempo**: La actividad de tiempo se activará cuando reciba un “token” en su entrada. Cuando se cumpla la expresión de tiempo, se activará y enviará un “token” a la actividad relacionada. Puede hacer referencia a un punto en el tiempo, un evento en el tiempo o una duración.



Nodos de control

Gestionan el flujo de control dentro de una actividad.

El “token” de entrada se ofrece a todas las salidas, pero atravesará al menos uno de ellos.

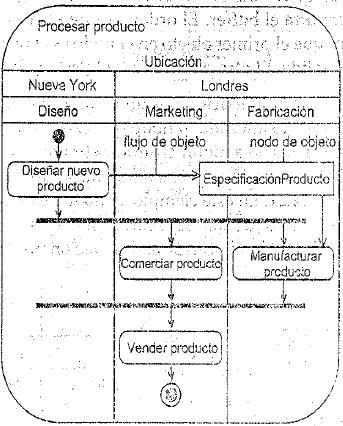
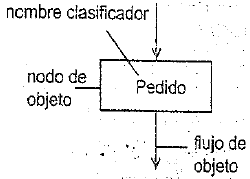
Cada extremo de salida está protegido por una “condición de protección”, que deben ser mutuamente excluyentes.

Define:

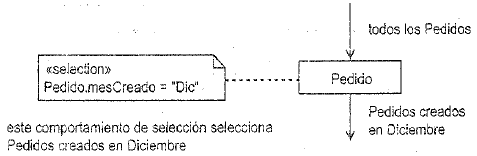
|  |  |  |
| --- | --- | --- |
|  | Nodo inicial | Indica dónde empieza el flujo cuando se invoca una actividad. |
|  | Nodo final de actividad | Terminar termina una actividad. |
|  | Nodo final de flujo | Termina un flujo específico dentro de una actividad; los otros flujos no se ven afectados. |
|  | Nodo de decisión | Se cruza el extremo de salida cuya condición de protección es verdadera. Puede tener opcionalmente una <<decisiónInput>>.  El “token” de entrada se ofrece a todas las salidas, pero atravesará al menos uno de ellos.  Cada extremo de salida está protegido por una “condición de protección”, que deben ser mutuamente excluyentes. |
|  | Nodo de fusión | Copia tokens de entrada en su extremo de salida.  Nota estereotipada: proporciona una condición de decisión. <<>>. |
|  | Nodo fork | Divide el flujo en múltiples flujos concurrentes. |
|  | Nodo join (sincronización) | Sincroniza múltiples flujos concurrentes. Puede tener opcionalmente una especificación de sincronización para modificar su semántica. |

Nodos de objeto

* Son nodos especiales que indican que las instancias de un clasificador especial están disponibles en un punto específico en la actividad.
* Se etiquetan con el nombre del clasificador y representan instancias de ese clasificador o sus subclases. Los extremos de entrada y salida son flujos de objeto. Los propios objetos se crean y consumen por los nodos de acción.
* Cuando un nodo objeto recibe un token de objeto en uno de sus extremos de entrada, ofrece este token en todos sus extremos de salida simultáneamente y estos extremos completan el token. El punto clave es que sigue habiendo un solo token, el token no se replica en los extremos. El primer extremo que acepta el token lo toma.



* Actúan como buffers, en los que se pueden albergar los tokens de objeto mientras esperan a ser aceptados por otros nodos. Por defecto, todo nodo de objeto puede albergar un número infinito de tokens de objeto y el ordenamiento es FIFO. Sin embargo, se puede definir el tamaño máximo del buffer y el tipo de ordenamiento.
* Pueden seleccionar objetos de los extremos de entrada de acuerdo a algún criterio definido. Se especifica por una nota estereotipada <<selection>>.



* Se pueden utilizar para recopilar objetos de múltiples flujos entrantes de objeto o para distribuir objetos a múltiples flujos salientes. En estos casos, está utilizando el nodo pura y exclusivamente para almacenamiento y puede asignarle el estereotipo <<centralBuffer>>.
* Se pueden poner conjuntos de objetos sin duplicados. Para esto, prefijar el nombre del clasificador con “Conjunto de”.
* Pueden representar objetos en un estado determinado. Los parámetros de actividad son entrada o salida de un nodo objeto.
* Se pueden utilizar para proporcionar entradas y salidas de actividades.

**UML 2 Capitulo 16 – Workflow de Diseño**

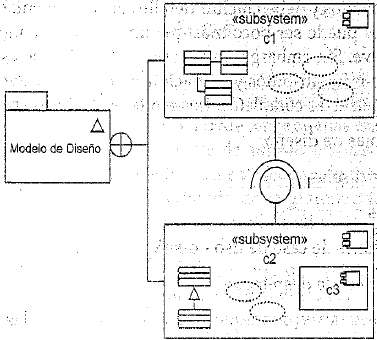
Es la actividad de modelado principal durante la última parte de la fase de elaboración y la primera mitad de la fase de construcción.

En el análisis, el foco estaba en crear un modelo lógico del sistema que capturara la funcionalidad que debía proporcionar para satisfacer los requisitos del usuario. La finalidad en diseño es especificar completamente cómo se implementará esta funcionalidad. Los diseñadores orientados a objetos deciden los aspectos estratégicos de diseño como la persistencia o distribución de objetos y crean un modelo de diseño en consecuencia.

Diseñar artefactos: Metamodelo

El modelo de diseño contiene muchos subsistemas de diseño; estos subsistemas son componentes que pueden contener muchos tipos diferentes de elementos de modelado.

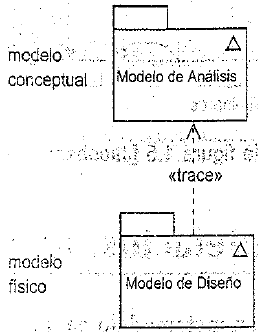
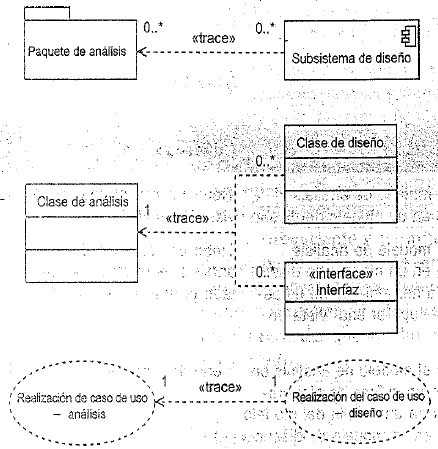
Medamodelo para el modelo de diseño:



Las interfaces tienen un importante rol arquitectónico en diseño.

Existe una sencilla relación <<trace>> entre los modelos de análisis y diseño: el modelo de diseño está basado en el de análisis y se puede considerar una mejora de ello.

El modelo de diseño contiene los mismos elementos que el modelo de análisis, pero todos los artefactos están bien formados y deben incluir detalles de implementación.

Los modelos de diseño constan de:

* Subsistemas de diseño.
* Clases de diseño.
* Interfaces (artefacto clave).
* Realizaciones de CU – diseño.
* Un diagrama de despliegue (muestra cómo se distribuye el sistema de SW sobre nodos computacionales físicos).

¿Debería mantener los dos modelos?

Cuatro estrategias:

1. Convertir el modelo de análisis en uno de diseño.
2. Convertir el modelo de análisis en uno de diseño y utilizar una herramienta de modelado para recuperar una vista de análisis.
3. Convertir una copia del modelo de análisis en uno de diseño.
4. Mantener dos modelos separados.

No existe mejor estrategia, depende de su proyecto.

Si se necesita realizar alguna de las siguientes cuestiones, es necesario conservar una vista de análisis:

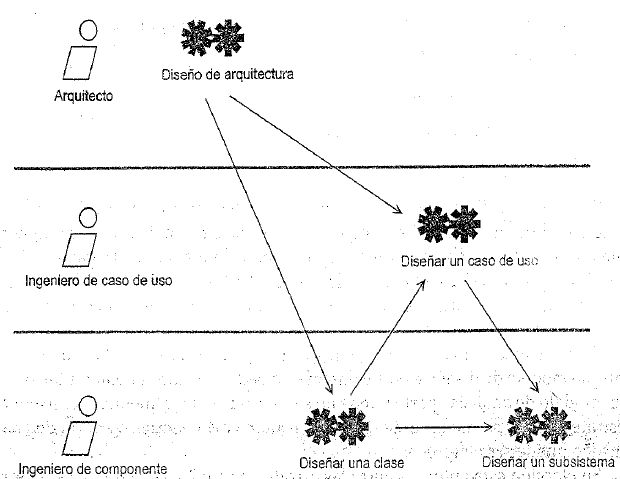
* Incorporar nuevas personas al proyecto.
* Entender el sistema, meses o años después de la entrega.
* Entender cómo el sistema cumple con los requisitos del usuario.
* Proporcionar trazabilidad de requisitos.
* Planificar mantenimiento y mejoras.
* Entender la arquitectura lógica del sistema.
* Externalizar la construcción del sistema.

Qué estrategias elegir:

* Si el sistema es amplio, complejo, estratégico o potencialmente de larga duración:
  + Debería conservar una vista de análisis eligiendo entre las estrategias 3 y 4.
* Si el sistema es pequeño:
  + El propio modelo de diseño es suficientemente pequeño para ser entendible modelo de análisis no necesario.
  + Estrategias 1 y 2.
* Si el sistema no es estratégico o tiene un ciclo de vida proyectado corto:
  + Mantener modelos de análisis y diseño aparte, puede ser peligroso.
  + Estrategias 1 y 2.

Detalle del workflow de diseño

Workflow de UP para el diseño:

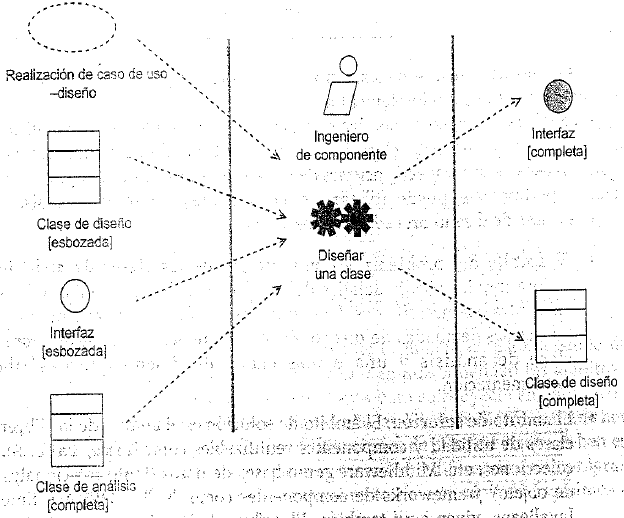


Actividad UP: diseño de arquitectura

En UP, la actividad que lanza todo el proceso de diseño es el diseño de arquitectura. Esta actividad se realiza por uno o más arquitectos y no es normalmente un paso aparte. Recordar que UP es un proceso iterativo por lo que el diseño ocurre al final de la fase de elaboración y principio de la construcción a medida que los detalles de la arquitectura del sistema se desarrollan.

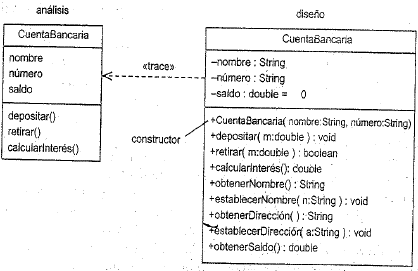
**UML 2 Capitulo 17 – Diseñar Clases**

Después del diseño de la arquitectura, las siguientes actividades son diseñar una clase y diseñar un CU. Estas dos actividades ocurren concurrentemente e iterativamente.



Actividad UP: Diseñar una clase

Una clase de diseño “completa” desde la perspectiva de UP es una que está suficientemente detalla para servir como una base para crear código fuente.

El nivel necesario de detalle depende de su proyecto. Si va a generar código directamente del modelo, sus clases de diseño se tendrán que modelar en gran detalle. Por otro lado, si simplemente van a servir como anteproyecto para programadores, se pueden modelar en menos detalle.

¿Qué son clases de diseño?

Las clases de diseño son clases cuyas especificaciones se han completado hasta tal nivel que se pueden implementar. Tienen todos sus atributos definidos, sus tipos, métodos, operaciones y parámetros.

Clases de Diseño

* Origen
  + Ámbito del problema: a través de una mejora de las clases de análisis.
    - Añadir detalles de implementación.
    - Generar nuevas clases derivadas de una clase de análisis.
    - Evitar clases “navaja suiza” (clases que tratan de hacerlo todo).
  + Ámbito de solución:
    - Librerías de clases de utilidad y componentes reutilizables.
    - Conexiones a BD.
    - Componentes de frameworks (.Net, EJB, etc.).
    - Entornos GUI.
    - Las herramientas técnicas que permiten implementar el sistema.
* Conformación – Anatomía de una clase de diseño
  + Especifican exactamente cómo una clase completará sus responsabilidades
    - Atributos (todos): nombre, tipo, visibilidad y (valor predeterminado).
    - Operaciones (todas): nombre, lista de parámetros y tipo de retorno.
      * Operaciones detalladas = Métodos.
      * Una operación de análisis puede derivar en varias operaciones de diseño.

Diseñar clases

Clases de diseño bien creadas

* **Completa y suficiente**
  + Completitud
    - Proporcionar a otras clases lo que podrían esperar razonablemente.
    - El nombre y descripción de la semántica indican lo que se puede esperar de la clase.
  + Suficiencia
    - Todas las operaciones de la clase están centradas en realizar la finalidad de la clase.
    - Mantener la clase de diseño lo más sencilla y centrada posible.
    - No incorporar operaciones innecesarias para la clase.
  + Regla: una clase debería hacer lo que los usuarios de la clase esperan, ni más ni menos.
* **Sencilla**
  + Las operaciones deben ofrecer un solo servicio sencillo.
  + No debe haber diferentes formas de hacer lo mismo.
  + Las clases deben siempre poner disponible el conjunto de operaciones más sencillo y pequeño posible.
  + **Relajación:** cuando se requiera mejoras en el rendimiento puede haber operaciones diferenciadas (un mismo proceso pero que pueda ser realizado de manera individual y/o en lotes). Solamente debería aplicar esto cuando exista un caso demostrado para hacerlo.
  + **Estadísticas:** las aplicaciones dedican cerca del 90% de su tiempo en el 10% de sus operaciones; estas son las operaciones que debe identificar y optimizar para obtener verdaderos aumentos de rendimiento.
* **Alta cohesión**
  + Toda clase debe modelar un solo concepto abstracto y debería tener un conjunto de operaciones que soporten el propósito de la clase.
  + Las clases cohesivas son por lo general fáciles de entender, reutilizar y mantener.
  + Una clase cohesiva tiene un pequeño conjunto de responsabilidades que están íntimamente relacionadas.
* **Bajo acoplamiento**
  + Una clase debería estar asociada con suficientes clases para permitirle realizar sus responsabilidades, y solamente debería asociar clases si existe un verdadero vínculo semántico entre ellas.
  + El acoplamiento debe ser minimizado tanto como se pueda.
  + Deben revisarse las asociaciones entre clases que surjan por restricciones de implementación o por pretender reutilizar código.
  + Por supuesto, cierto acoplamiento es bueno y deseable. Un alto acoplamiento dentro de un subsistema es generalmente aceptable ya que esto indica una alta cohesión dentro del componente.
  + Solamente compromete la arquitectura cuando el acoplamiento es entre subsistemas. Tiene que tratar de reducir este tipo de acoplamiento.

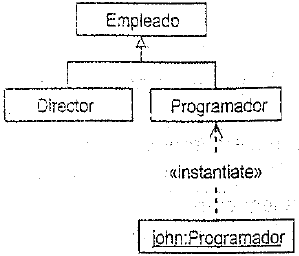
Herencia vs Agregación

La herencia es una técnica muy potente y es un mecanismo clave para generar polimorfismos en lenguajes de programación. Sin embargo, tiene ciertas características no deseables:

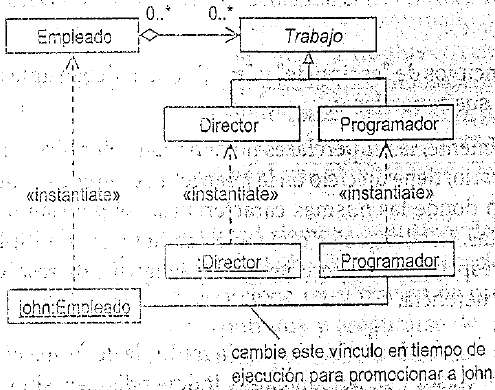
* Es la forma mayor de acoplamiento posible entre dos o más clases.
* La encapsulación es débil dentro de una jerarquía de clase. Los cambios en la clase base provocan cambios en las subclases y tienen un impacto importante sobre otras clases en el sistema.
* Es un tipo muy inflexible de relación puesto que en tiempo de ejecución son relaciones fijas.

Ejemplo en el que se aplica herencia erróneamente:

Error: modelar ‘roles’ como herencia.



Solución: Usar agregación. Los roles son elementos asociados a los empleados.



Herencia múltiple

Heredar de más de un padre.

Características:

* Todas las clases padre implicadas deben estar separadas semánticamente (deben ser ortogonales).
* Los principios de “es tipo de” y “sustitución” deben aplicarse entre la subclase y todas sus superclases.
* Normalmente, las superclases no deberían tener ningún padre en común. De lo contrario, tiene un ciclo en la jerarquía de herencia y puede haber múltiples rutas en donde las mismas características se podrían heredar de clases más abstractas.

Una forma común de utilizar herencia múltiple de forma eficaz es en aquellas clases que no son independientes (“clase mezclada”), sino que están diseñadas específicamente para estar “mezcladas” con otras clases utilizando este mecanismo.

Herencia vs realización de interfaz

* **Herencia**
  + Mediante ésta obtiene:
    - Interfaz: las operaciones públicas de las clases base.
    - Implementación: los atributos, relaciones, operaciones protegidas y privadas de las clases base.
  + Utilizarla únicamente cuanto está preocupado por heredar algunos detalles de implementación (operaciones, atributos y relaciones) de una superclase.
* **Realización de la interfaz**
  + Mediante ésta obtiene:
    - Una interfaz: conjunto de operaciones, atributos y relaciones que no tienen implementación.
  + Es de utilidad siempre que desee definir un contrato, pero no está preocupado por heredar detalles de implementación.
  + No proporciona reutilización de código, sino que, mediante contratos, asegura que la implementación de clases se adapta a los mismos.
  + Mucho más flexible y robusta que la herencia.

**UML 2 Capitulo 18 – Diseño de relaciones**

Transformar las relaciones de análisis en relaciones de diseño. Para crear un modelo de diseño, hay que especificar cómo se van a realizar estas asociaciones. Esta mejora implica varios procedimientos:

* Mejorar asociaciones hacia relaciones de agregación o composición donde sea apropiado.
* Implementar asociaciones uno a muchos.
* Implementar asociaciones muchos a uno.
* Implementar asociaciones muchos a muchos.
* Implementar asociaciones bidireccionales.
* Implementar clases de asociación.

Todas las asociaciones de diseño deben tener:

* Navegabilidad.
* Multiplicidad en ambos extremos.

Además, deberían tener un nombre de asociación, o un nombre de rol en al menos el extremo destino.

Agregación y composición

En diseño, puede convertir una relación de asociación en una relación de agregación o composición.

* **Agregación**
  + Es el tipo de relación más normal entre objetos.
  + Ejemplo: un ordenador y sus periféricos los periféricos pueden estar o no, se pueden compartir entre ordenadores y no son “propiedad” de ningún ordenador en particular.
* **Composición**
  + Es un tipo muy fuerte de relación entre objetos.
  + Ejemplo: un árbol y sus hojas las hojas son propiedad exactamente de un árbol, no se pueden compartir y cuando el árbol muere, las hojas se van con él.

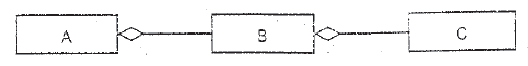
Semántica de agregación

La agregación es un tipo de relación todo-parte en la que el conjunto se compone de muchas partes. Un objeto (el todo; dominante) utiliza los servicios de otro objeto (la parte; pasiva). Si solamente tiene navegabilidad desde el todo a la parte, la parte ni siquiera sabe que es parte de un todo.

* El conjunto puede existir algunas veces independientemente de las partes, algunas veces no.
* Las partes pueden existir independientemente del conjunto.
* El conjunto está en cierto sentido incompleto si falta alguna de las partes.
* Es posible tener propiedad compartida de las partes por varios conjuntos.

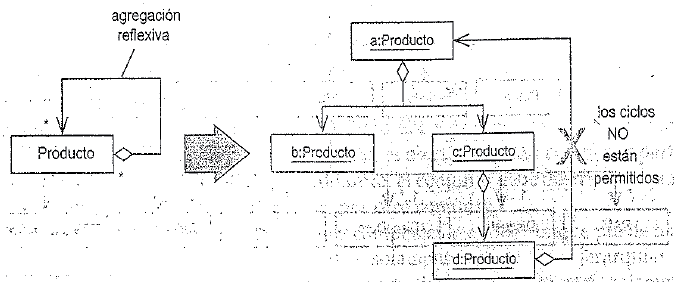
La agregación es transitiva:

* Si C es parte de B y B es parte de A, entonces C es también parte de A.

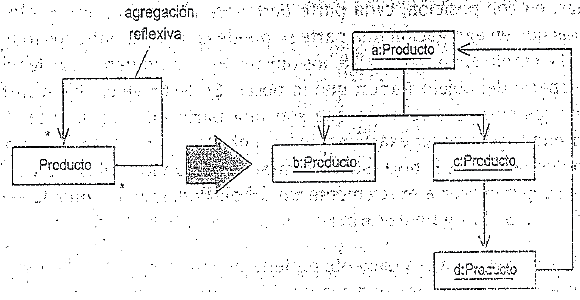


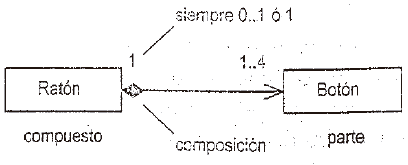
La agregación es asimétrica:

* Un objeto nunca puede ser, directa o indirectamente, parte de sí mismo.
* Los objetos Producto se pueden componer de otros objetos Producto. Esto está bien siempre y cuando sean objetos diferentes y se mantenga la restricción de asimetría.



* + Solución: Asociación reflexiva. Permite que los elementos estén asociados (no agregados) consigo mismos.



Semántica de composición

La composición es una forma más fuerte de agregación y tiene una semántica similar, pero más restringida. Es una relación todo-parte y es tanto transitiva como asimétrica. La diferencia con la agregación es que, las partes no tienen vida independientemente fuera del todo. Además, cada parte pertenece al menos a un y sólo a un todo.

Semántica:

* Las partes pueden solamente pertenecer a un conjunto de cada vez; no existe la posibilidad de propiedad compartida de una parte.
* El conjunto tiene responsabilidad (creación y destrucción) única para la disposición de todas sus partes.
* El conjunto puede liberar partes, siempre y cuando la responsabilidad para ellas se asuma por otro objeto.
* Si se destruye el conjunto, debe destruir todas sus partes o pasar la responsabilidad a algún otro objeto.

Cómo mejorar relaciones de análisis

La mayoría de las asociaciones de análisis terminan como agregación o composición (excepción: asociación reflexiva). Habiendo decidido utilizar alguna de estas dos opciones, debería proceder de la siguiente forma:

* Añadir multiplicidades y nombres de rol a la asociación si están ausentes.
* Decidir qué lado de la asociación es el todo y cuál la parte.
* Examinar la multiplicidad del todo: si es 0..1, o exactamente 1, puede utilizar composición; si no, agregación.
* Añadir navegabilidad del todo a la parte (las asociaciones de diseño deben ser unidireccionales).

Multiplicidad de las asociaciones

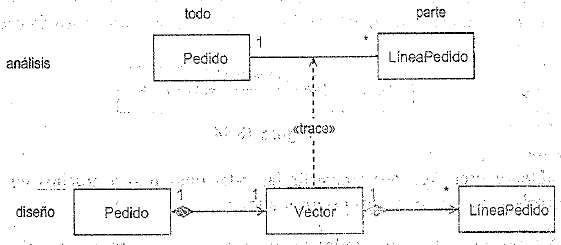
* **Asociaciones uno a uno:** Casi siempre se convierte en composición. De hecho, una asociación de este tipo implica que es mucho más fuerte y a menudo merece la pena ver si se pudieran fusionar en una sola clase sin romper ninguna de las reglas de diseño.
* **Asociaciones muchos a uno:** Existe una multiplicidad de muchos en el lado del todo y una multiplicidad de exactamente 1 en el lado de la parte. No se podría utilizar composición ya que la parte se comparte entre muchos todos 🡪 usar agregación 🡪 Comprobar que no existan ciclos de agregación.
* **Asociaciones uno a muchos:** Existe una colección de objetos en el lado parte de la relación. Para implementarlo se utilizan clases de colección.

Colecciones

Una clase de colección es una clase cuyas instancias se especializan en gestionar colecciones de otros objetos. Todas las clases de este tipo tienen operaciones para:

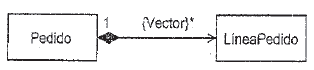
* Añadir y eliminar objetos a la colección.
* Recuperar una referencia de un objeto en la colección.
* Recorrer la colección, es decir, pasar por la colección desde el primer objeto al último.

Para representar la relación de las partes al todo (clase de colección) puede utilizar composición (si el todo es responsable del ciclo de vida de las partes) o agregación.

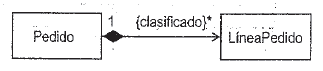


Existen cuatro estrategias fundamentales para modelar colecciones:

* Modelar la clase de colección explícitamente: solo cuando la elección de colección es estratégica, de lo contrario, satura el modelo.
* Indicar a la herramienta de modelado cómo se debería implementar cada asociación específica uno a muchos.



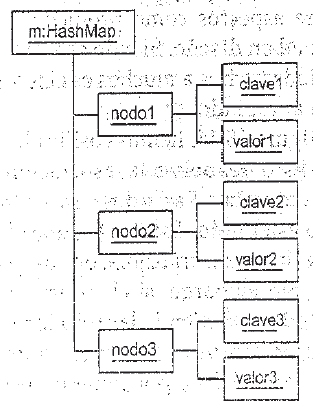
* Especificar la semántica de la colección al añadir una propiedad a la relación.



* {clasificado} Elementos en la colección se mantienen en un orden estricto.
* {sin orden} no existe orden de los elementos
* {único} los elementos en la colección son todos únicos: un objeto aparece en la colección como mucho, una vez.
* {no único} Duplicar elementos está permitido en la colección.

Opciones:

* {sin orden, no único} bag.
* {sin orden, único} set.
* {clasificado, único} ordered set.
* {clasificado no único} sequence.
* No convertir las relaciones uno a muchos en clases de colección; dejarlo a los programadores.

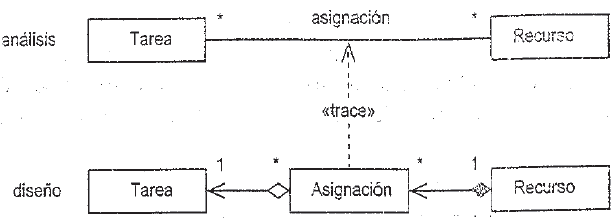
Mapas (Diccionarios): no tiene soporte en UML

* Clases que funcionan como una tabla de BD con dos columnas: clave primaria y valor.
* Cada nodo apunta a dos objetos: el objeto clave y el objeto valor.
* Búsqueda rápida para un objeto clave específico.

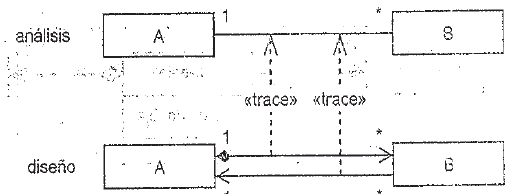
Relaciones cosificadas

Relaciones de análisis que no son implementables directamente y necesita “cosificarlas” (hacer concreto o real). Casos que necesita cosificar:

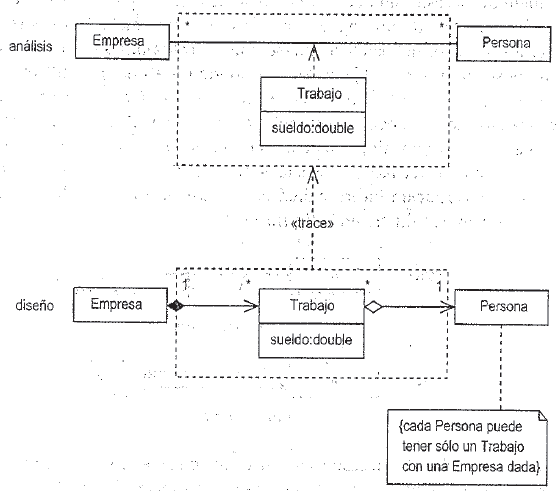
* Asociaciones muchos a muchos.
  + No soportadas por los lenguajes de programación orientados a objetos.
  + Tiene que decidir qué parte es el todo y luego utilizar agregación o composición según sea apropiado.



* Asociaciones bidireccionales.
  + Necesita ser consciente de la restricción asimétrica. Si la clase A tiene una relación de agregación o composición con la clase B, tiene que modelar la relación de B a A como una asociación no mejorada.



* Clases de asociación.
  + Decida qué lado de la asociación es el todo y utilice composición, agregación y navegabilidad en consecuencia.
  + Puede implicar añadir restricciones al modelo.



* Cuando cosifica una clase de asociación, pierde la semántica de la clase de asociación (“los objetos de cada extremo deben formar un único par”) Puede restaurar la semántica al añadir una nota que contenga la restricción apropiada como se muestra en la figura.

**Fundamentos de orientación a objetos**

Los objetos son unidades cohesivas que combinan datos y función. Es una instancia de una clase que define el conjunto común de características (atributos y operaciones) que se comparten por todas las instancias de esa clase.

**Encapsulación:** los datos de un objeto están ocultos y solamente se pueden manipular al invocar una de las funciones del objeto.

Operaciones: especificaciones para funciones de objeto creadas en análisis.

Métodos: implementaciones para funciones de objeto creadas en implementación.

Características:

* **Identidad**: su existencia es única en tiempo y espacio.
* **Estado**: conjunto significativo de valores y relaciones de atributo para el objeto en un punto en el tiempo.
* **Comportamiento**: servicios que el objeto ofrece a otros objetos, modelados como un conjunto de operaciones. Invocar operaciones puede generar una transición de estado.

Los objetos colaboran para generar el comportamiento del sistema. La interacción implica objetos enviando mensajes. Cuando se recibe un mensaje se invoca la operación correspondiente.

Notación de objeto en UML:

Todo icono de objeto tiene 2 comportamientos:

* Comportamiento **superior**: nombre del objeto y/o nombre de clase, que deben estar subrayados. Los nombres del objeto y de las clases se crean poniendo en mayúscula la primera letra de cada palabra. El nombre de objeto se separa del nombre de clase por dos puntos.
* Comportamiento **inferior**: nombres de atributo y valores separados por un signo de igual. Los nombres de atributos se crean poniendo en mayúscula la primera letra de cada palabra. Los tipos de atributo a menudo se omiten para abreviar.

Compartimiento de atributo:

* Visibilidad: se aplica a atributos y operaciones dentro de una clase. Las diferentes son:

|  |  |  |
| --- | --- | --- |
| + | Visibilidad pública | Cualquier elemento que puede acceder a la clase puede acceder a cualquiera de sus características con visibilidad pública. |
| - | Visibilidad privada | Solamente las operaciones dentro de la clase pueden acceder a características con visibilidad privada. |
| # | Visibilidad protegida | Solo las operaciones dentro de la clase o dentro de hijos de la clase pueden acceder a características con visibilidad protegida. |
| ~ | Visibilidad de paquete | Cualquier elemento que esté en el mismo paquete que la clase, o en un subpaquete anidad, puede acceder a cualquiera de sus características con visibilidad de paquete. |

* Nombre: es obligatorio.
* Multiplicidad: indica el “número de elementos” que participan en una relación.
* Valor inicial: permite especificar el valor que tomará un atributo cuando se instancia un objeto desde la clase.
* Tipo: es el tipo de dato, por ej.: si es integer, float, etc.
* Estereotipos y valores etiquetados.

Compartimiento de operaciones: toda operación de una clase debe tener una firma única.

* Visibilidad.
* Nombre.
* Lista de parámetros: nombre y tipo de parámetro. Un parámetro puede tener un valor predeterminado.
* Tipo de retorno.
* Estereotipos y valores etiquetados.

Toda operación tiene una propiedad denominada isQuery. Si se establece en verdadero, es una **operación de consulta.** Esto significa que no tiene efectos secundarios y no cambia el estado del objeto que se invoca.

Ámbito de aplicación:

Los atributos y operaciones de ámbito de instancia pertenecen o funcionan sobre objetos específicos:

* Las operaciones de ámbito de instancia pueden acceder a otras operaciones de ámbito de instancia o atributos la misma.
* Las operaciones de ámbito de instancia pueden acceder a todos los atributos u operaciones de ámbito de clase.

Los atributos y operaciones de ámbito de clase pertenecen y operan sobre toda la clase de objetos.

* Los atributos y operaciones de ámbito de clase solamente pueden acceder a otras operaciones de ámbito de clase.

Construcción y destrucción de objetos.

**Constructores:** operaciones especiales que crean nuevas instancias de clases. Deben tener ámbito de aplicación de clase. Una clase puede tener muchos constructores, todos con el mismo nombre, pero con distinta lista de parámetros.

**Destructores:** destruyen los objetos, esto es, se dejan a un recolector de basura automático.

**UML 2 Capitulo 19 – Interfaces y Componentes**

**Interfaces y componentes – Diseño de subsistemas**

* Desglosar el sistema en partes que sean tan independientes como sea posible *subsistemas*.
* La comunicación entre subsistemas se hace a través de sus interfaces
* Objetivos del diseño de subsistemas
  + Minimizar el acoplamiento al diseñar interfaces apropiadas.
  + Asegurarse que cada subsistema realiza correctamente el comportamiento especificado por sus interfaces.

Interfaz

* Conjunto de características públicas.
* Separa la *especificación* de funcionalidad de su *implementación*.
* No se pueden instanciar. Solo definen un *contrato* que se puede realizar por cero o más clasificadores.
  + El realizador (clase o subsistema) implementa, y acuerda el contrato definido por la interfaz.

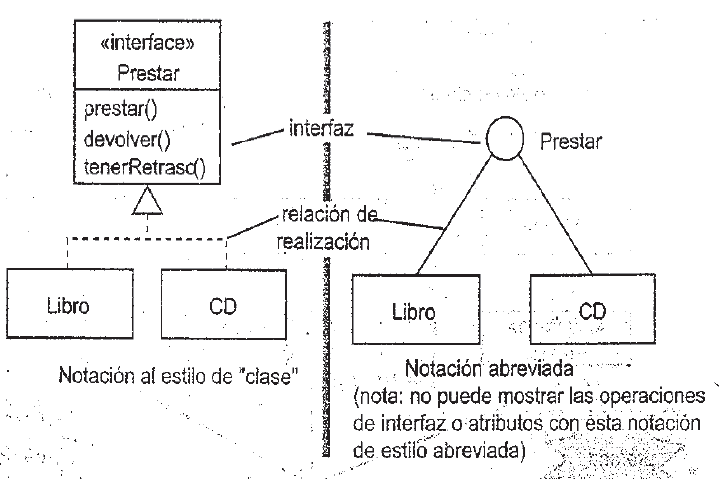
|  |  |
| --- | --- |
| Especifica | Implementador debe |
| **Operación** | Tener una operación con la misma firma y semántica. |
| **[Atributo]** | Tener operaciones públicas para establecer y obtener el valor del atributo. |
| **Asociación** | Tener una asociación con el clasificador destino. También entre clasificadores que implementan interfaces (si una interfaz especifica una asociación con otra interfaz, los clasificadores que implementan estas interfaces deben tener una asociación entre ellos). |
| **Restricción** | Soportar la restricción. |
| **Estereotipo** | Tener el estereotipo. |
| **Valor etiquetado** | Tener el valor etiquetado. |
| **Protocolo** | Realizar el protocolo. |

* Los atributos y operaciones deben estar especificados completamente.
  + Firma completa de las operaciones (nombre, tipos de todos los parámetros y tipo de retorno).
  + Semántica de la operación (texto o pseudocódigo).
  + El nombre y tipo de los atributos
  + Cualquier operación o estereotipo de atributo, restricción y valor etiquetado.
* Una interfaz solamente define una especificación por sus características y nunca implica ninguna implementación en particular.

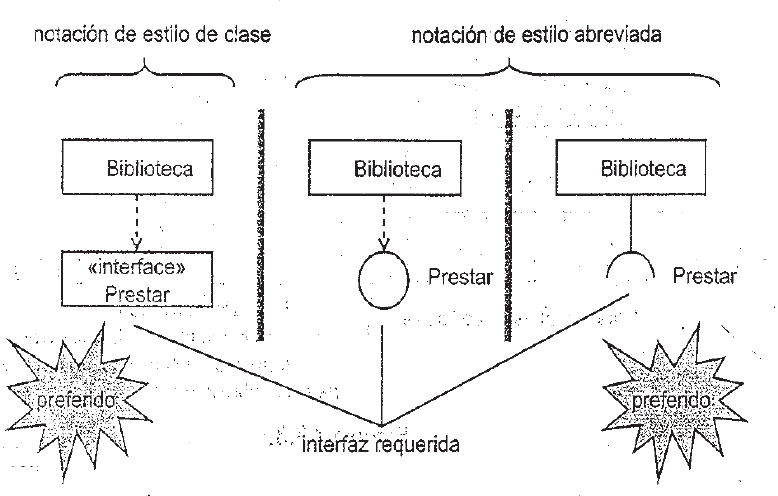
Cambios en la forma de diseño: en vez de relacionar clases (diseñar para una implementación), es más flexible conectar a una interfaz, y que esta interfaz se pueda realizar por un número de clases y otros clasificadores (diseñar para un contrato).

*Interfaz:* define un servicio ofrecido por una clase, subsistema o componente.

Interfaces proporcionadas y requeridas

* Notación de interfaces – 2 variantes:
  + Notación al estilo de “clase” completa
    - Tiene que mostrar los atributos y operaciones.
    - La relación se dibuja como una línea de puntos con una punta de flecha no rellena.
  + Notación abreviada
    - Puede no mostrar los atributos y operaciones.
    - La relación de realización de una interfaz se dibuja como una línea gruesa sin punto de flecha.
* **Interfaces proporcionadas:** conjunto de interfaces realizadas por un clasificador.
* **Interfaces requeridas:** conjunto de interfaces requeridas por un clasificador para su operación.

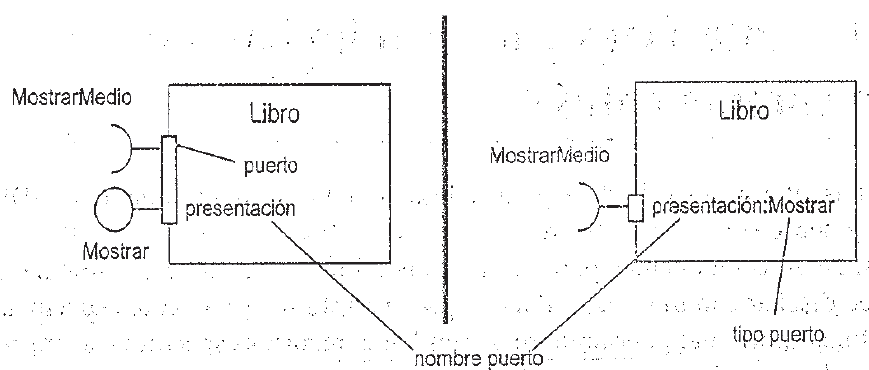
Ejemplo: sistema de gestión de biblioteca. La clase *Biblioteca* mantiene una colección de *Libros* y una colección de *CD*. Cada *Libro* y *CD* realiza la interfaz *Prestar* que especifica el protocolo para un elemento que se puede prestar. Aunque *Libros* y *CD* son semánticamente diferentes, la interfaz permite que se los trate de forma uniforme.

* Se nombran como las clases: primera letra en mayúscula, cada palabra empieza en mayúsculas.
  + Alternativa: Visual Basic y C#: prefijar con ‘I’ IPrestar.
  + Alternativa 2: agregar “able” Prestable.
* Notación de interfaces requeridas y proporcionadas
  + La clase requirente, entiende y necesita el protocolo definido por la interfaz.
  + Sintaxis: puede mostrar una interfaz requerida como una dependencia a una interfaz o como un elemento en el que encaja la interfaz requerida.
  + Interfaz requerida interfaz provista.

Realización de interfaces vs herencia

* La realización de interfaces implica “realiza el contrato especificado por”.
* Herencia implica “es un.”
* Ambos tipos de relación pueden generar *Polimorfismo*.
* Una solución basada en interfaz es más sencilla y tiene mejor semántica.

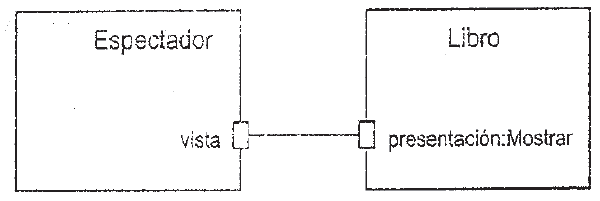
Puertos

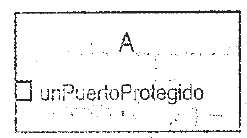
* Agrupa un conjunto semánticamente cohesivo de interfaces proporcionadas y requeridas.
* Puntos específicos de interacción entre un *clasificador* y su entorno.
* Útiles para estructurar las interfaces proporcionadas y requeridas de un clasificador.
* Utilizar puertos es más conciso que mostrar todas las interfaces, pero puede ser más difícil de leer.
* Para conectar los puertos, sus interfaces proporcionadas y requeridas deben coincidir.
* El nombre del puerto es opcional.

Este puerto consta de una interfaz requerida (*Mostrar*Medio) y una interfaz proporcionada (*Mostrar*).

La figura muestra dos variantes de la notación del puerto:

* La parte izquierda muestra el caso normal.
* La parte derecha muestra una alternativa más concisa. Solo es aplicable si el puerto tiene un solo tipo de interfaz proporcionada (puede tener cero o más interfaces requeridas).

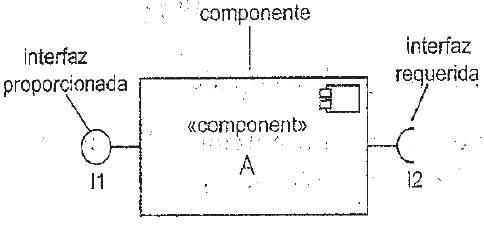


* Visibilidad
  + Visibilidad Pública (+)
    - Se dibuja solapando el límite del clasificador.
    - Las interfaces proporcionadas y requeridas son públicas.
  + Visibilidad Protegida (#) o Privada (-)
    - El rectángulo del puerto se muestra dentro del límite del clasificador.
* Multiplicidad
  + Indican la cantidad de instancias del puerto que puede tener un clasificador/componente.
  + Se puede mostrar entre corchetes detrás del nombre del puerto y el nombre del tipo. Ejemplo: *Mostrar[1]*

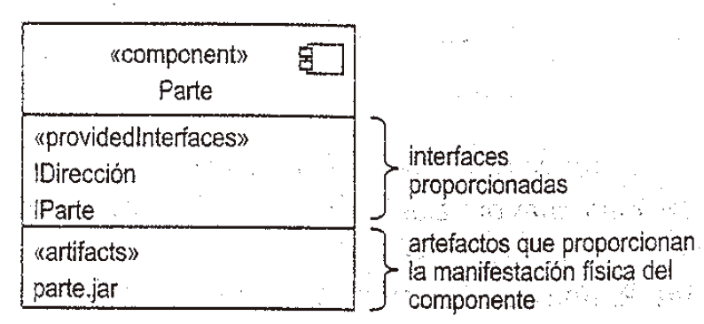
**Desarrollo basado en componentes**

* Las interfaces son la clave para el desarrollo basado en componentes (DBC) Construir software a partir de partes.
* Las interfaces especifican un *contrato* que puede ser implementado por diferentes constructores.

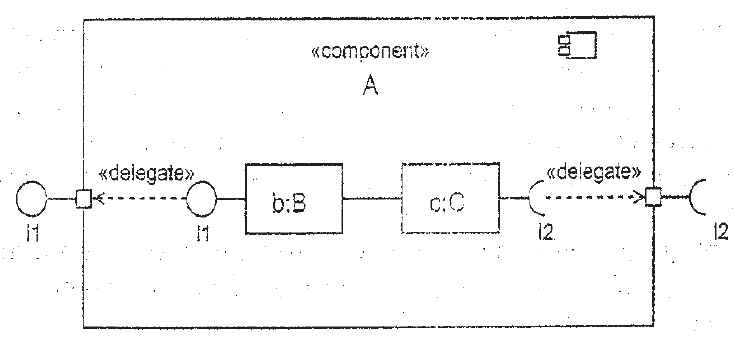
Componente

* “Un componente representa una parte modular de un sistema, que encapsula sus contenidos, y cuya manifestación se remplaza dentro de su entorno.” (UML2)
* Un componente actúa como una caja negra cuyo comportamiento externo está completamente definido por sus interfaces proporcionadas y requeridas. Debido a esto, un componente se puede reemplazar por otro que soporte el mismo protocolo.
* “Una parte modular, desplegable y sustituible de un sistema, que incluye la implantación y expone un conjunto de interfaces.”
* Características
  + Pueden tener atributos, operaciones, pueden participar en relaciones de asociación y generalización.
  + Pueden tener una estructura interna (compuesto por otros clasificadores, partes y conectores).
  + Puede mostrar las partes anidadas dentro del componente o externamente, conectadas a éste por *relaciones de dependencia*.
  + Son reemplazables.
  + Puede tener interfaces proporcionadas/requeridas y puertos.
  + Pueden representar una entidad realizable (componente) o constructor lógico (subsistema; no existe en tiempo de ejecución) o algo que se puede instanciar en tiempo de ejecución.
  + El diagrama de componente puede mostrar componentes, dependencias entre componentes y la forma en la que los clasificadores están asignados a los componentes.
  + Un componente se dibuja como un cuadro con el estereotipo *<<component>>* y/o un icono de componente en su esquina superior derecha.

Componentes compuestos

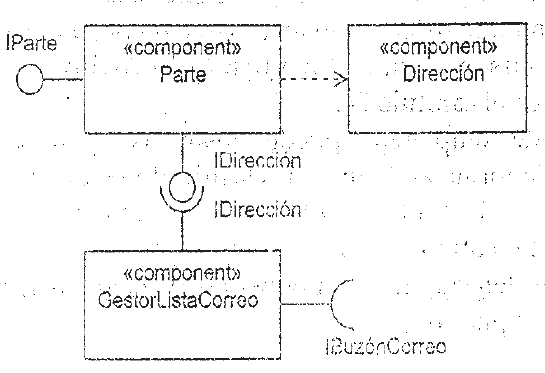
* Los “servicios requeridos y ofrecidos” de la componente estarán delegados a “interfaces de sus subcomponentes/partes.”
* Cuando un componente tiene estructura interna, delegará las responsabilidades definidas por sus interfaces a una o más de sus partes internas.

En la siguiente figura, el componente A proporciona la interfaz I1 y requiere la interfaz I2. Encapsula dos partes de tipo b y c. Delega el comportamiento especificado por sus interfaces proporcionadas y requeridas a b y c respectivamente.



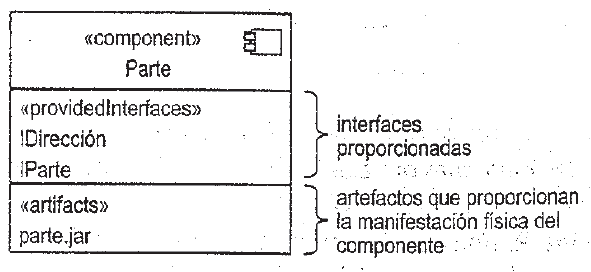
Dependencias entre componentes

* Los componentes pueden depender de otros componentes. SIEMPRE a través de interfaces.

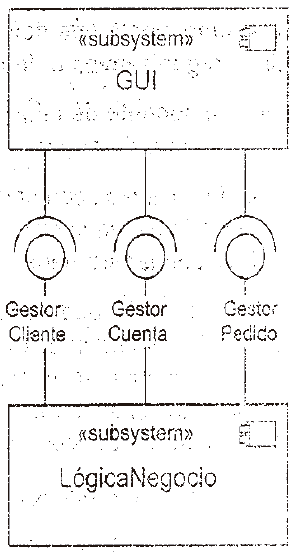


En la figura puede ver lo siguiente:

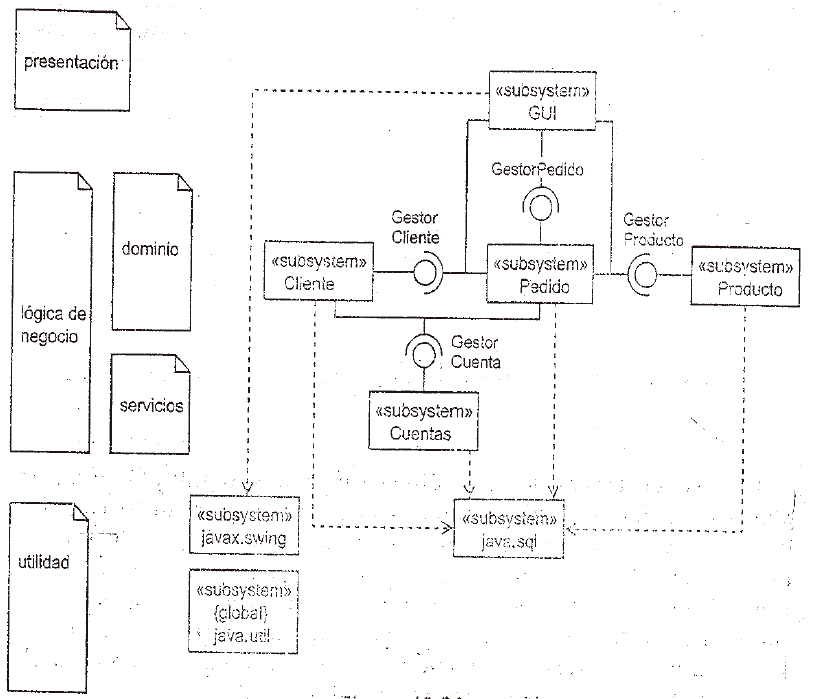
* El componente *Parte* proporciona dos interfaces del tipo *IParte* y *IDirección*. Estas interfaces se representan como círculos.
* El componente *GestorListaCorreo* requiere dos interfaces del tipo *IDirección* e *IBuzónCorreo*. Éstas se representan como sockets.
* Existe un conector de ensamble entre el componente *Parte* y el componente *GestorListaCorreo*. Esto muestra que *GestorListaCorreo* se está comunicando con el componente Parte por medio de la interfaz proporcionada *IDirección*.

Los componentes a menudo se muestran simplemente como cuadros negros con sus interfaces anexadas. Sin embargo, también se puede tener una vista de caja blanca de un componente según se muestra en la siguiente figura. La misma presenta los detalles internos del componente. Puede mostrar cualquier interfaz proporcionada, requerida, realizaciones o artefactos asociados.

Subsistemas

* Es un componente que actúa como una unidad de descomposición para un sistema mayor. Se dibuja como un componente estereotipado subsistem
* Es una **unidad lógica** que se utiliza para descomponer sistemas grandes en bloques manejables.

Arquitectura en capas

* La colección de subsistemas e interfaces constituyen la arquitectura del sistema de software.
* Arquitectura en capas.
  + Organiza los subsistemas e interfaces en capas donde los subsistemas en cada capa son semánticamente cohesivos.
  + Disminuye el *acoplamiento* entre subsistemas.
  + Consideraciones
    - Las dependencias entre capas van en un solo sentido.
    - Las dependencias están mediadas por interfaces.
    - Una capa debería requerir interfaces de la capa inferior, y proveer interfaces a la capa superior.
  + El patrón básico es una división entre las capas de presentación, la lógica de negocio y las capas de utilidad. Es bastante común subdividir aún más la capa de lógica de negocio.

Ventajas de las Interfaces

* Diseñar para contratos libera el modelo de dependencias de implementación, lo que aumenta su **flexibilidad** y **extensibilidad**.
* Reduce el número de dependencias entre clases, componentes y subsistemas controla (y disminuye) el **acoplamiento**.

Desventajas de las Interfaces

* La flexibilidad se logra con **complejidad**.

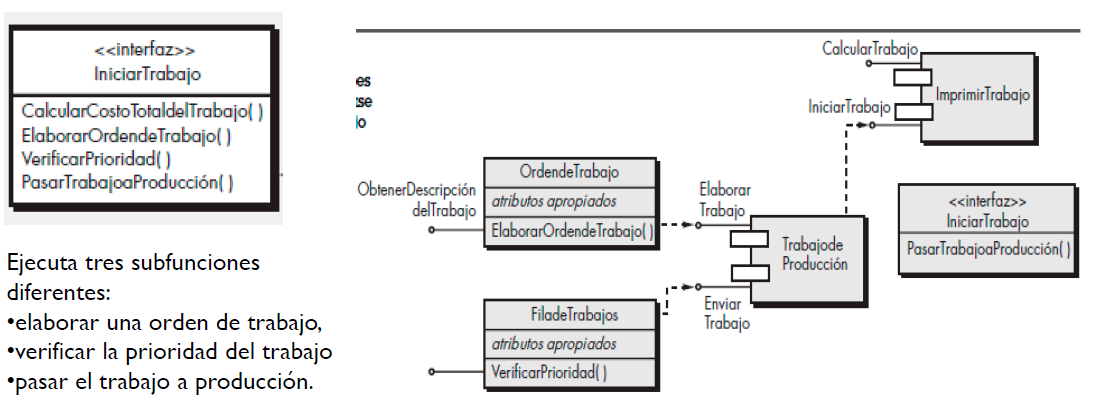
**Encontrar Interfaces**

Se comienza con el modelo de requerimientos y se elaborar clases de análisis (para los componentes que se relacionan con el dominio del problema) y clases de infraestructura (para los componentes que dan servicios de apoyo para el dominio del problema).

* Esta actividad de elaboración se aplica a cada componente definido como parte del diseño de la arquitectura.
* Una vez concluida, se aplica más elaboración a cada atributo, operación e interfaz.
  + Deben especificarse las estructuras de datos apropiadas para cada atributo.
  + Se diseñan los detalles algorítmicos requeridos para implantar la lógica del procesamiento asociada con cada operación.
  + Por último, se diseñan los mecanismos requeridos para implantar la interfaz. Para el software orientado a objetos, esto incluye la descripción de todos los mensajes que se requieren para efectuar la comunicación dentro del sistema.

Proceso (Pressman: Proceso de desarrollo basado en componentes. Se va un poco más allá de lo que el profe desea.)

* Paso 1. Identificar todas las clases de diseño que correspondan al dominio del problema. Con el uso del modelo de requerimientos y arquitectónico, se elabora cada clase de análisis y componente de arquitectura.
* Paso 2. Identificar todas las clases de diseño que correspondan al dominio de la infraestructura.
  + Estas clases no están descritas en el modelo de los requerimientos y con frecuencia se pierden a partir del modelo arquitectónico.
  + Las clases y componentes en esta categoría incluyen componentes de la interfaz gráfica de usuario (con frecuencia disponibles como componentes reutilizables), componentes del sistema operativo y componentes de administración de objetos y datos.
* Paso 3. Elaborar todas las clases de diseño que no fueron adquiridas como componentes reutilizables. Describir en detalle todas las interfaces, atributos y operaciones necesarios para implantar la clase.
* Paso 3a. Especificar detalles del mensaje cuando colaboren clases o componentes
  + El modelo de requerimientos utiliza un diagrama de colaboración para mostrar la forma en la que las clases de análisis colaboran una con la otra.
  + En diseño en el nivel de componentes, es útil mostrar los detalles de estas colaboraciones especificando las estructuras de los mensajes que pasan entre los objetos de un sistema.
  + Aunque esta actividad de diseño es opcional, se usa como precursor de la especificación de interfaces que muestren el modo en el que se comunican y colaboran los componentes del sistema.
* Paso 3b. Identificar interfaces apropiadas para cada componente.
  + Una interfaz UML es un “grupo de operaciones visibles externamente (para el público). La interfaz no contiene estructura interna, ni atributos ni asociaciones…”
  + Las operaciones definidas para la clase de diseño se clasifican en una o más interfaces.
  + Cada operación dentro de la interfaz debe ser cohesiva, es decir, debe tener un procesamiento que se centre en una función o subfunción limitada.
  + Cohesión de interfaz:



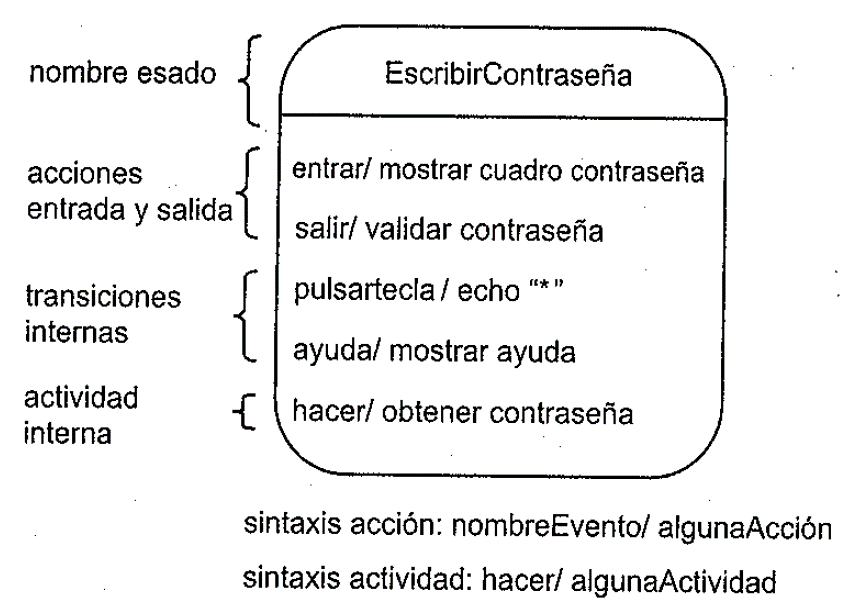
* Paso 3c. Elaborar atributos y definir tipos y estructuras de datos requeridos para implantarlos.
  + En general, las estructuras y tipos de datos usados para definir atributos se definen en el contexto del lenguaje de programación que se va a usar para la implantación.
  + UML: visibilidad nombre:tipo[multiplicidad]=valorinicial
* Paso 3d. Describir en detalle el flujo del procesamiento dentro de cada operación.
  + Esto se logra con el uso de seudocódigo basado en lenguaje de programación o con un diagrama UML de actividades, o diagrama de interacción.
  + Cada componente del software se elabora a través de cierto número de iteraciones que apliquen paso a paso el concepto de refinamiento.
* Paso 4. Describir las fuentes persistentes de datos (bases de datos y archivos) e identificar las clases requeridas para administrarlos.
  + Es normal que las bases de datos y archivos trasciendan la descripción del diseño de un componente individual.
  + En la mayoría de los casos, estos almacenamientos persistentes de datos se especifican al inicio como parte del diseño de la arquitectura. No obstante, a medida que avanza la elaboración del diseño, es frecuente que sea útil dar detalles adicionales sobre la estructura y organización de dichas fuentes persistentes de datos.
* Paso 5. Desarrollar y elaborar representaciones del comportamiento para una clase o componente. Utilizar diagramas de estado UML.
* Paso 6. Elaborar diagramas de despliegue para dar más detalles de la implantación.
  + Los diagramas de despliegue se utilizan como parte del diseño de la arquitectura y se representan en forma de descriptor. De este modo, las funciones principales de un sistema (que con frecuencia se representan como subsistemas) se representan en el contexto del ambiente de computación que las contendrá.
* Paso 7. Rediseñar cada representación del diseño en el nivel de componentes y siempre considerar alternativas.
  + El diseño es un proceso iterativo. El primer modelo en el nivel de componentes que se crea no será tan completo, consistente o exacto como la enésima iteración que se realice. Es esencial rediseñar a medida que se ejecuta el trabajo.

**Máquinas de Estado**

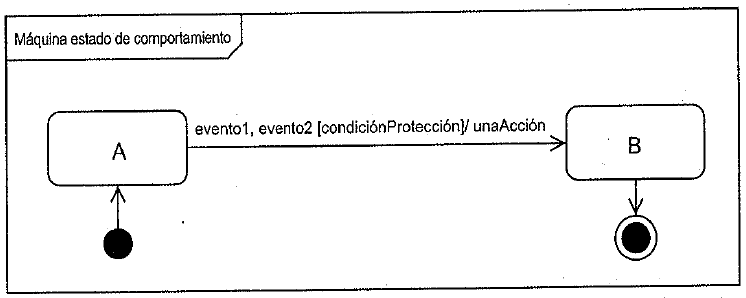
* Modelan el *comportamiento dinámico* de un sistema (como los diagramas de actividad).
* Se utilizan para modelar la *historia del ciclo de vida* de un solo objeto reactivo.
* Número finito de estados. Transiciones de estados en respuesta a eventos.
* Elementos:
  + **Estado**: una condición o situación durante la vida de un objeto durante el cual se cumple alguna condición, realizar alguna actividad o espera la ocurrencia de algún evento.
  + **Evento**: la especificación de una ocurrencia que tiene ubicación en tiempo y espacio.
  + **Transición**: el movimiento de un estado a otro en respuesta a un evento.
* Objeto reactivo: es un objeto que proporciona el contexto para una máquina de estado.
  + Responden a eventos externos.
  + Pueden generar y responder a eventos internos.
  + Tienen un ciclo de vida modelado como una progresión de estados, transiciones y eventos.
  + Pueden tener un comportamiento que depende de comportamiento pasado.
  + Ejemplo: lámpara. Estados: apagada, encendida. Estado final: fundida. Eventos: apagar, encender.
* Pueden usarse máquinas de estado para modelar el comportamiento dinámico de clasificadores como clases, casos de uso, subsistemas y sistemas enteros.

Estado

* Cambia con el tiempo.
* Determinados por:
  + Valores de los atributos
  + Relaciones con otros objetos
  + Actividades que realiza
* Importante:
  + Solo considerar los estados relevantes, los que agreguen valor.
  + Debe existir diferencia semántica que marque una diferencia entre los estados para poder modelarlos.
* Ejemplo:
  + Empleado se actualiza Sueldo no tiene relevancia, ya que no cambia su estado por ello.
  + Empleado en Vacaciones sí tiene relevancia, ya que puede cambiar su estado (a inactivo, por ejemplo). Esta transición sí puede valer la pena.
* Sintaxis:



* + Las acciones de entrada y salida son **instantáneas**, no se pueden interrumpir.
  + Las actividades sí se pueden interrumpir.
  + Transiciones internas: permiten capturar el hecho de que algo que merece modelarse ha ocurrido, pero no provoca una transición a un nuevo estado. Se pueden interrumpir.
  + Dos acciones especiales (entrada y salida) están asociadas a los eventos especiales entrar y salir, que tienen semántica especial. El evento entrar ocurre instantáneamente y automáticamente al entrar en el estado, es lo primero que ocurre cuando se entra en el estado y hace que la acción de entrada asociada se ejecute. El evento salir es lo último que sucede instantánea y automáticamente al salir del estado y hace que la acción de salida asociada se ejecute.

Transiciones

*En evento1 OR evento 2, si condiciónProtección, entonces realizar unaAcción e inmediatamente entrar al estado B.*

Toda transición tiene tres elementos opcionales:

* Cero o más eventos: especifican ocurrencias externas o internas que pueden activar la transición.
* Cero o una condición de protección: expresión booleana que debe evaluar en verdadero antes de que pueda ocurrir la transición.
* Cero o más acciones: parte del trabajo asociado con la transición. Ocurre cuando se activa la misma.

Pseudo estado de unión

Las transiciones se pueden conectar por pseudo estados de unión, que representan puntos donde las mismas se unen o ramifican. Se representan como círculos rellenos con una o más transiciones de entrada y una o más de salida. Cada salida debe protegerse con una condición de protección mutuamente excluyente (de forma tal que solo una pueda activarse).

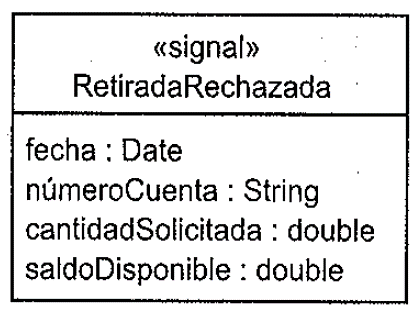
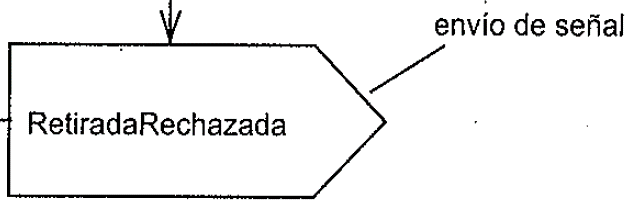
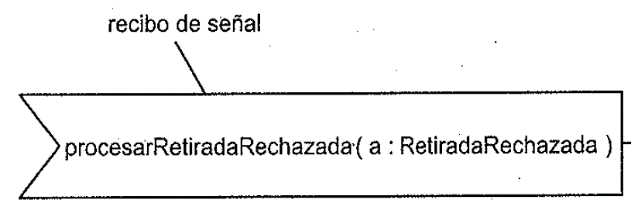
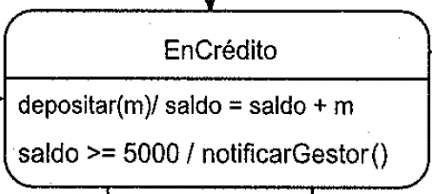
Pseudo estado de opción

Permite dirigir el flujo de acuerdo a condiciones que especifica en sus transiciones de salida. Se representa con un rombo no relleno. Las condiciones en las transiciones de salida deben ser mutuamente excluyentes.

Eventos

Especificación de una ocurrencia destacable que tiene ubicación en tiempo y espacio. Activan transiciones. Se pueden mostrar externamente en transiciones o internamente dentro de estados.

Existen cuatro tipos de evento:

* Evento de llamada
  + Petición de una operación específica a invocarse en una instancia de la clase de contexto.
  + Ejemplo: retirar(m) [saldo >= m].
* Evento de señal
  + Paquete de información que se envía asíncronamente entre objetos. Una señal normalmente no tiene ninguna operación, dado que su finalidad es transportar información solamente.
* Evento de cambio
  + Se especifica como una expresión booleana. La acción asociada con el evento se realiza cuando el valor de la expresión pasa de falso a verdadero.
  + Los eventos se activan cada vez que el valor de la expresión cambia de falso a verdadero. Para volver a activarse, debe pasar a falso y luego de nuevo a verdadero.
  + En la imagen: “saldo >= 5000 / notificarGestor()” es el evento de cambio.
* Evento de tiempo
  + Se indican normalmente por medio de las palabras clave *cuando* y *después*.
  + Ejemplo: “cuando(fecha=07/10/2017)”, “después(3 meses)”

**Implementación – Diagrama de Despliegue**

Implementación

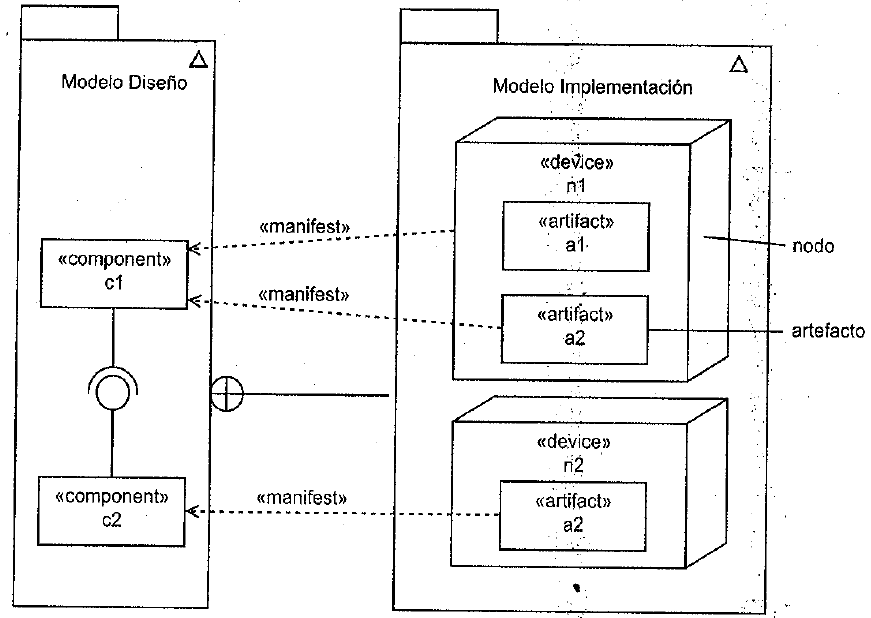
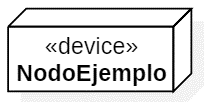
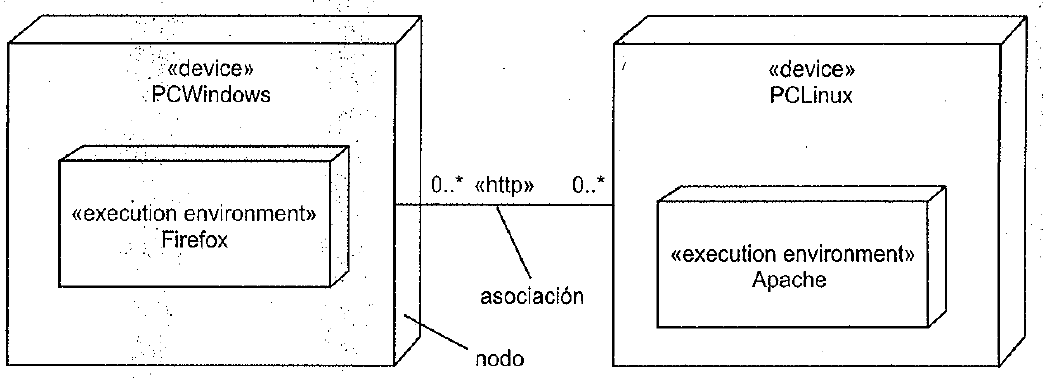
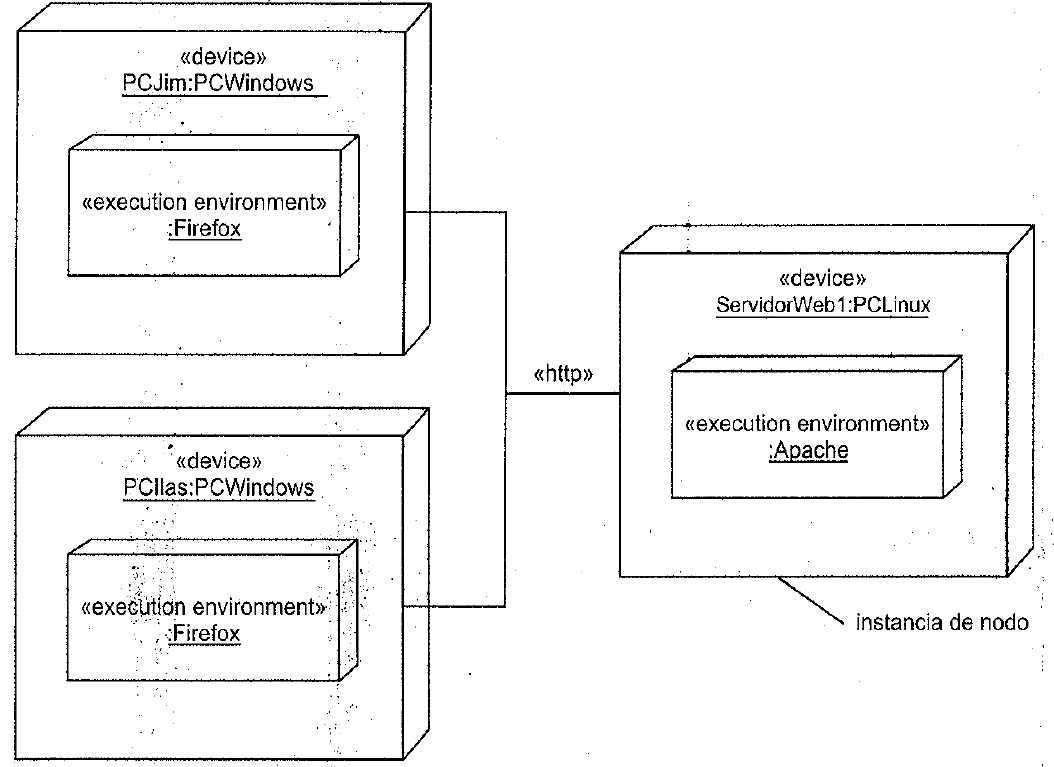
* Transformar un *modelo de diseño* en *código ejecutable* Generar un modelo de Implementación
* Modelo de implementación:
  + Vista de implementación de un modelo de diseño
  + Especifica cómo los elementos de diseño se muestran como artefactos
  + Y cómo estos se despliegan en nodos.
* **Artefactos:** representan las especificaciones de los elementos del mundo real, como archivos de código fuente; elementos que tienen una representación concreta en software.
* **Nodos:** representan especificaciones de hardware o entornos de ejecución en las que se despliegan los artefactos.

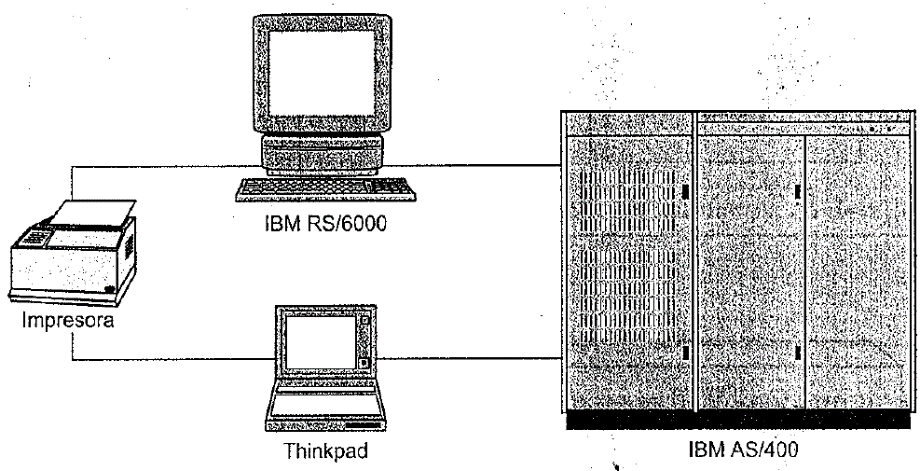
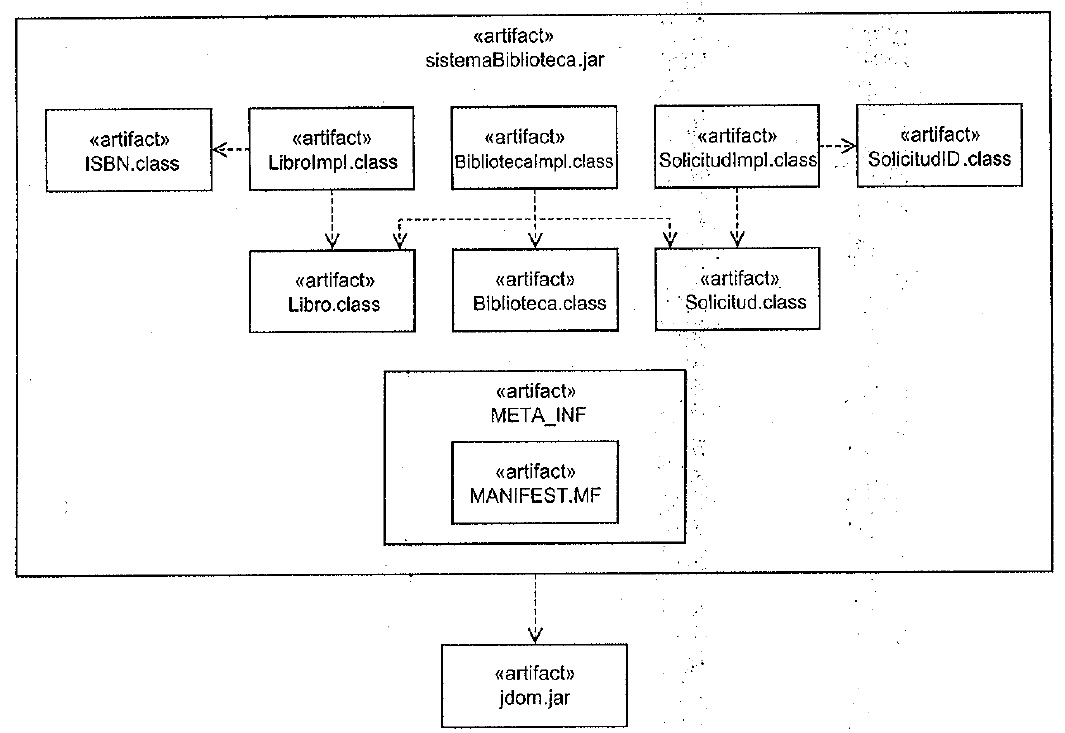
Diagrama de Despliegue

* Muestra cómo el software se desplegará sobre hardware físico y cómo se conecta ese hardware.
* **Despliegue**:
  + Proceso de asignar artefactos a nodos, o
  + Instancias de artefactos a instancias de nodos
* Mapea la arquitectura de software con la arquitectura física que lo ejecuta
* En sistemas distribuidos modela la distribución del software a través de los nodos físicos.
* Formas:
  + **Forma de descriptor**: contiene nodos, relaciones entre nodos y artefactos.
    - Nodo: un tipo de hardware. Ejemplo: PC.
    - Artefacto: un tipo de software físico. Ejemplo: un archivo JAR.
  + **Forma de instancia:** contiene instancias de nodos, relaciones entre instancias de nodo e instancias de artefacto.
    - Instancia de nodo: pieza de hardware identificable y específica. Ejemplo: PC-386.
    - Instancia de artefacto: instancia específica de un tipo de software. Ejemplo. SubSistemaVentas.JAR.
* Proceso:
  + 1. En el workflow de diseño, centrarse principalmente en el nodo o instancias de nodos, y conexiones.
  + 2. En el workflow de implementación, centrarse en asignar instancias de artefactos a instancias de nodo (forma de instancias) o artefactos a nodos (forma de descriptor).
* **Nodos**
  + “Un nodo representa un tipo de recurso computacional sobre el que se pueden desplegar los artefactos para su ejecución.” UML 2.0
* Estereotipos
  + <<device>>: dispositivo físico.
  + <<execution environment>>: tipo de entorno de ejecución para software (servidor Web Apache, Jboss, etc.).
* Nodos anidados
  + Formato Descriptor: son buenos para modelar un tipo de arquitectura física.



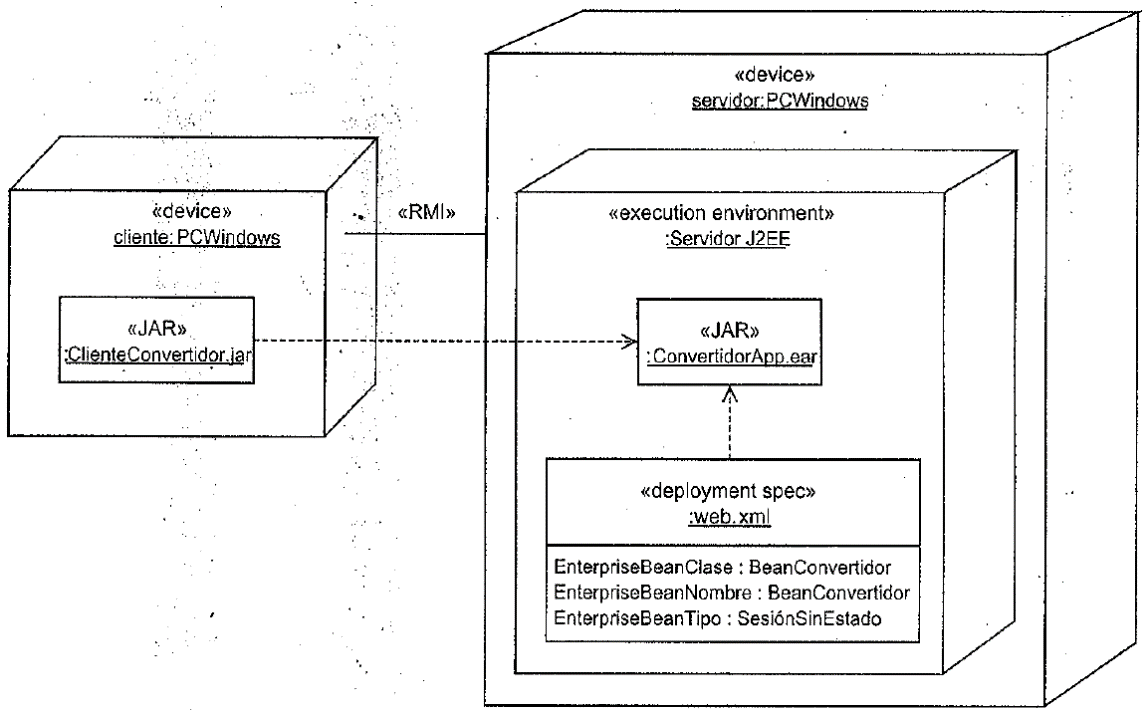
* + Formato Instancia: son buenos para modelar un despliegue real de esa arquitectura en un sitio determinado.



* Diagramas de despliegue estereotipados
  + Utilizar símbolos facilita la lectura del diagrama de despliegue.
  + Se pierde la visión del software que existe dentro de cada nodo.
  + Es útil para ver la infraestructura computacional.
* **Artefactos**
  + Representan la especificación de un elemento concreto del mundo real.
  + Los artefactos se despliegan en nodos.
  + Ejemplos:
    - Archivos fuente
    - Archivos ejecutables
    - Scripts
    - Tablas de Bases de Datos
    - Documentos
* Estereotipos:
  + Los artefactos se etiquetan con el estereotipo “artifact” y pueden tener un ícono de artefacto en su esquina superior derecha.
  + Todo artefacto tiene también un nombre de archivo en su especificación que indica la ubicación física del artefacto.

UML proporciona estereotipos estándar de artefactos que representan diferentes tipos de archivo.

|  |  |
| --- | --- |
| <<file>> | Archivo físico. |
| <<deployment spec>> | Especificación de detalles de despliegue. |
| <<document>> | Archivo genérico que contiene cierta información. |
| <<executable>> | Archivo de programa ejecutable. |
| <<library>> | Biblioteca estática o dinámica, como una DLL o archivo JAR. |
| <<script>> | Script que puede ejecutarse por un intérprete. |
| <<source>> | Archivo fuente que se puede compilar en un archivo ejecutable. |

* Despliegue