¿Qué es un Paradigma?

Un paradigma es una forma de ver la realidad, es una forma de vivir una realidad. Para poder formarse el concepto de lo que es un paradigma las palabras claves, entre otras, son: ***medio ambiente, cultura y reglas de juego***.

El ***medio ambiente*** es la idea de que uno vive en un paradigma y que se ***"acostumbra"*** y ***"se siente cómodo"*** viviendo en él, por ejemplo, estamos viviendo en la tierra y ***"acostumbrarse"*** a vivir en la luna va a costar un poco, esto es debido a que en la tierra es como si tuviésemos ciertas ***"reglas de juego"*** que son diferentes a las de la luna. En realidad lo único que cambió fue que la luna nos ***"atrae"*** con menos fuerza que la tierra y entonces las reglas de juego (la ley de la gravedad) en realidad no cambiaron para nada, pero nosotros percibiríamos que necesitamos ***"acostumbrarnos"*** a vivir en la luna.

La frase ***"reglas de juego"*** nos hace pensar en, por ejemplo, que el ajedrez y el juego de la guerra son juegos que tienen el mismo objetivo y que uno necesita de cierta ***"capacidad"*** (coeficiente mental, astucia) para desenvolverse bien ó en el ajedrez ó en algún juego de guerra. Pero un campeón de algún juego de guerra perdería las primeras partidas con algún experto (no necesariamente un campeón) de ajedrez, esto es porque aquí sí que le cambiaron ***"las reglas de juego".*** Cuando este campeón de juego de guerra ***"se acostumbre"*** a las reglas del ajedrez entonces podrá competir con algún maestro del ajedrez en partidas de ajedrez.

Esto nos conduce a la idea de ***"experiencia".*** Y todo este ***"acostumbramiento",*** necesidad de prácticas para lograr ***"experiencia"*** y adaptación a las nuevas ***"reglas de juego"*** hasta sentirnos ***"cómodos"*** en el nuevo paradigma hace que formemos parte de una ***cultura***, o sea parte de una comunidad de personas que piensa de una determinada manera, que mira las cosas de una determinada forma, adaptada a ciertas reglas de juego, con cierto tiempo de acostumbramiento para lograr cierto nivel de experiencia y consecuentemente de sentirse cómodo en el nuevo paradigma.

Esta definición la necesitamos para poder estudiar qué ventajas tiene el Paradigma Orientado a Objetos con respecto al Paradigma Estructurado.

Entonces analizaremos esas ventajas desde el punto de vista de las ***reglas de juego*** que en este caso se traducen en ***reglas de programación*** ya que tanto en las Metodologías Estructuradas como en las Metodologías Orientadas a Objetos siempre se comenzó enunciando "Qué es un programa de computadora" visto desde el paradigma estructurado ó "Qué es un programa de computadora" visto desde el paradigma orientado a objetos. Luego de enunciar esas ***reglas de programación*** se generaron ideas de cómo diseñar sistemas que vayan a ser

desarrollados en uno u otro paradigma y, finalmente, se definieron las herramientas y técnicas del Análisis. O sea, que históricamente el camino que se siguió para ***"componer"*** una Metodología de Desarrollo de Sistemas (ó Proceso de Desarrollo de Sistemas) fue el siguiente:

Metodologías de Desarrollo:

PARADIGMA ESTRUCTURADO	PARADIGMA ORIENTADO A OBJETOS
✓ Programación Estructurada ✓ Diseño Estructurado ✓ Análisis Estructurado	✓ Programación Orientada a Objetos ✓ Diseño Orientado a Objetos ✓ Análisis Orientado a Objetos

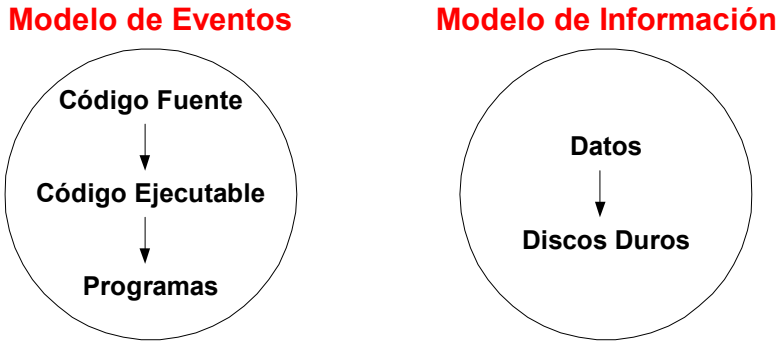
Pero, ¿qué es un programa de computadora?: Es una forma de que la computadora ***"represente"*** la realidad, son cosas que hacemos para que la computadora ***"emule"*** la realidad, para que haga cosas muchísimo más rápido de lo que haríamos en la realidad.

Tipos De Paradigmas de Programación

1. ***Paradigma Estructurado***
2. ***Paradigma Orientado a Objetos***

El Paradigma Estructurado

El Paradigma Estructurado observa a cualquier ente de la realidad como si estuviera dividido en dos partes: datos y acciones. Las acciones exteriorizan un comportamiento, una conducta.
El paradigma estructurado separa los datos (atributos con sus valores) que residen en archivos de computadora, del comportamiento (conducta) que son los módulos de un Diagrama de Estructuras. Los *módulos*, cuando se está ejecutando un sistema, residen en la memoria principal (memoria RAM) mientras que los archivos conteniendo los *datos* se encuentran almacenados en archivos en discos duros (en general).
Entonces es como que los módulos "acceden" a los archivos en discos duros y desde allí "traen" los datos que necesitan procesar a la memoria principal del ordenador. En definitiva hay una clara separación entre *datos* por un lado y *comportamiento* (código fuente decimos nosotros) por el otro.



En la Metodología estructurada que hemos estudiado se representa el plano del software a través de los ***"Diagramas de Estructuras"*** y el plano de los datos mediante los ***"Diagramas de Entidad Relación"*** o ***"Base de Datos".***

Diagramas del Paradigma Estructurado

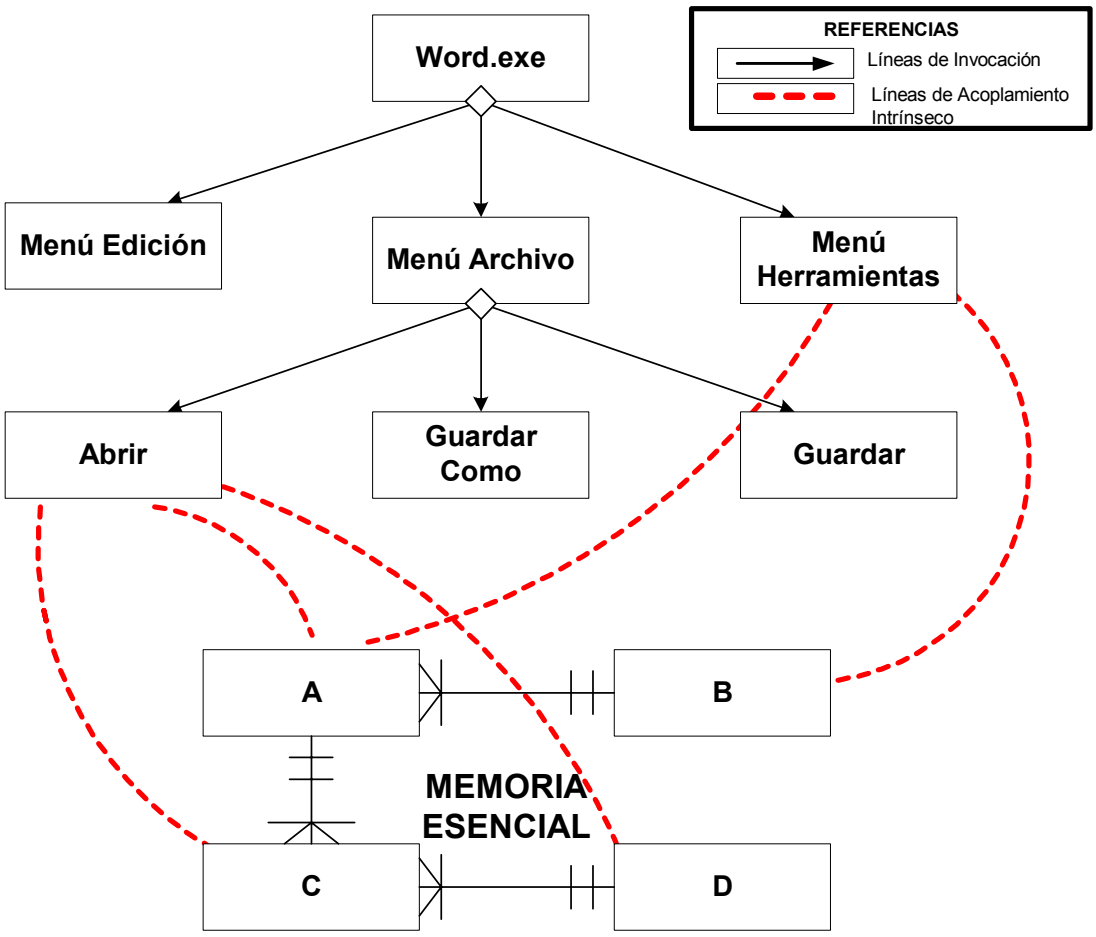
1. ***Diagrama Estructurado***
2. ***DER Normalizado o Base de Datos***

1. Diagrama Estructurado (Plano del Software)

El Diagrama Estructurado se basa en el concepto de *"Módulos*". Un *módulo* es un conjunto de sentencias fuente que se le presenta al compilador y este produce a partir de dichas sentencias un ejecutable. Los módulos se invocan entre sí mediante las líneas de invocación.

2. DER Normalizado o BD (Plano de los Datos)

En el DER los datos están almacenados en la ***"Memoria Esencial",*** que es el conjunto de datos que recuerda el sistema ***"para hacer lo que tiene que hacer".*** Cada tabla relacional está asociada con un conjunto de módulos mediante "líneas de acoplamiento intrínseco de los modelos estructurados".



En la figura anterior y usando al Word como sistema por todos conocido lo graficamos "como si" hubiese sido desarrollado con uso de técnicas de metodologías estructuradas y lo hacemos "relacionar" con las líneas de punto con tablas relacionales que, en el modelo relacional, las representamos con un Diagrama Entidad Relación normalizado (aunque, por supuesto el Word no lee ni guarda datos en entidad relacional alguna).

Definición de Acoplamiento

El acoplamiento es el grado o nivel de dependencia entre los módulos de un Diagrama de Estructuras. Lo ideal es que todos los módulos sean ***"independientes"*** algo que es imposible porque forman parte de un sistema de módulos (que es lo que es un Diagrama de Estructuras), y un sistema sabemos que es un conjunto de elementos (en este caso módulos) interrelacionados. De manera tal que a mayor acoplamiento, **PEOR** diseño.
Lo ideal en todo sistema es reducir el mismo. Constantine definió en su momento diferentes grados o niveles de acoplamiento dando mayor valor al mayor grado de acoplamiento en ciertos casos. Hay diferentes formas en que un módulo se ***"acopla"*** (se relaciona) con otros y lo que se busca es que solamente estén relacionados por ***líneas de invocación*** (cuando un módulo "llama" a otro para que se ejecute y cuando termine que devuelva "el control" al modulo "llamador").

El Acoplamiento Intrínseco del Modelo Estructurado: es el conjunto de las líneas de acoplamiento que se produce entre los módulos y los archivos (tablas relacionales) que usa. Como en el modelo estructurado se separa el software (los módulos) de los datos estas líneas de relación **NO** se pueden eliminar por eso es que se puede llamar acoplamiento ***intrínseco*** del modelo. Si es que se modifica la estructura (cantidad de atributos ó tipos de atributos –numérico, fechas, strings) de una entidad relacional se tiene que analizar el impacto que va a producir en los módulos que hacen uso de esa tabla relacional. Por otro lado si se modifica un módulo hay que analizar qué impacto va a producir en las entidades relacionales con las que ese módulo trabaja y a partir de allí, nuevamente, por cada entidad relacional que sufrió el impacto del cambio se tendrá que hacer el análisis del impacto que puede provocar en los módulos con los que cada tabla relacional está relacionada. Como se puede deducir no es un modelo que esté "preparado" para el cambio y uno de los propósitos de las Metodologías es desarrollar sistemas en los cuales cuando se produzca un cambio (cosa que va a ocurrir con gran probabilidad), ese cambio se haga fácilmente, en poco tiempo, con poco costo.

Definición de Programación Estructurada

Un programa estructurado es un conjunto de líneas de sentencias fuente que se agrupan en alguno de tres bloques o combinación de esos tres bloques.

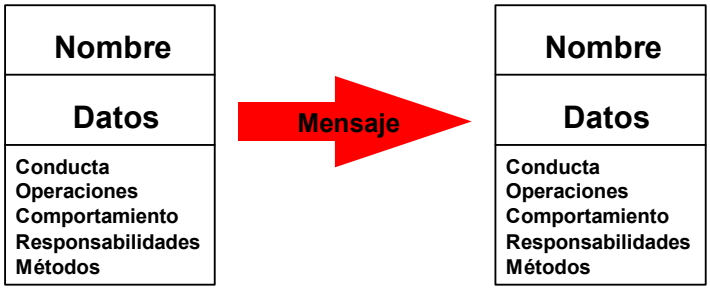
Dichos bloques son:

1. Bloque Secuencial
2. Bloque de Decisión (if-then-else)
3. Bloque Repetitivo (hacer hasta, hacer mientras, for-next)

El Paradigma Orientado a Objetos

El Paradigma Orientado a Objetos no separa los datos del comportamiento. El POO permite representar a entes de la realidad sin dividirlos en el interior de la computadora, sin poner por un lado los datos y por el otro el comportamiento sino que ambas cosas están juntas en lo que denominamos "objeto". Es una forma de ***"representar la realidad más cerca de la realidad"*** (más cerca que el modelo estructurado).

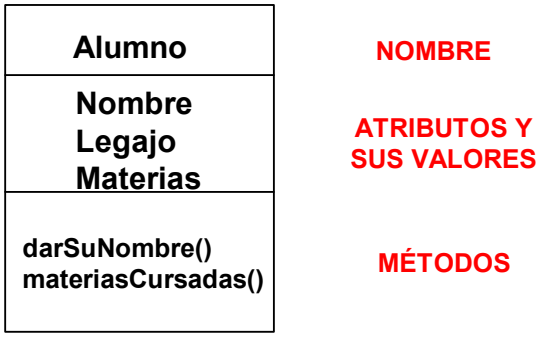
El POO trabaja con ***"Objetos",*** que son entidades con identidad, datos y comportamiento.



Programación Orientada a Objetos

La Programación Orientada a Objetos es un conjunto de objetos que se comunican entre sí enviándose mensajes.

Ejemplo de Objeto:



Nota

Como no hemos visto UML aún no estamos representando los objetos como estrictamente lo exigen las normas UML.

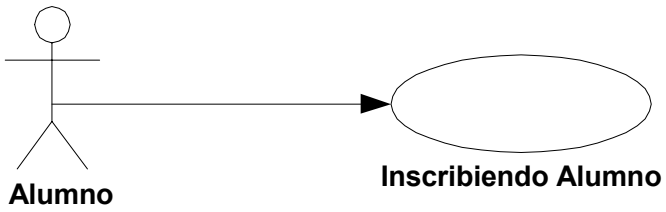
Ventajas: las ventajas de usar POO, con respecto a la Programación Estructurada, o mejor dicho el Paradigma Orientado a Objetos con respecto al Paradigma Estructurado, están dadas por:

- La POO representa "la realidad más cerca de la realidad"
- El paradigma OO elimina el acoplamiento intrínseco del modelo estructurado, lo cual se deduce del hecho de que los datos no están separados de los métodos u operaciones que al ejecutarse exteriorizan una conducta, un comportamiento.

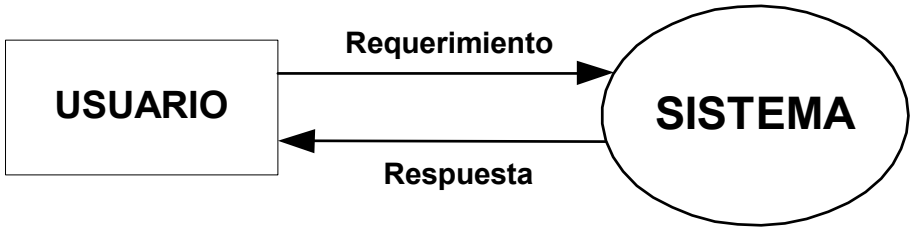
Similitudes entre los "Casos de Uso" y el "Modelo de Eventos"

Los "Casos de Uso" fueron introducidos por Jacobson en 1992. Sin embargo, la idea de especificar un sistema a partir de su interacción con el ambiente fué original de Mc Menamin y Palmer.
Podría definirse a un "Caso de Uso" como el conjunto de todos los eventos que tienen lugar cuando el usuario solicita algún servicio y quiere que el sistema le responda. Un "Caso de Uso" describe las tareas que realiza el sistema cuando un actor realiza alguna acción para obtener algo del sistema.

Ejemplo: caso de uso “Inscribiendo Alumno”



De manera similar, el Modelo de Eventos (MC Menamin- Palmer), define los requerimientos del sistema desde el punto de vista del usuario o agente externo. Es decir, lista los datos que estimulan al sistema para entrar en acción (requerimientos) y los datos que comprenden la respuesta del sistema ante el evento.

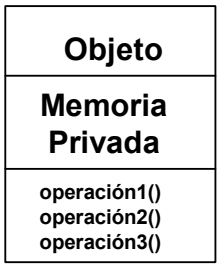


El Paradigma Orientado a Objetos

Como definimos anteriormente, la Programación Orientada a Objetos es un método de implementación en el que los programas se organizan como colecciones de ***"objetos"*** que colaboran entre sí enviándose mensajes. Los componentes básicos de la Programación Orientada a Objetos son los ***"objetos"*** y los ***"mensajes"***

Los Objetos- Definiciones

1. Los objetos son agentes que causan acciones y son sujetos a esas acciones.
2. Un objeto es un conjunto de memoria privada más un set de operaciones



3. Un objeto es la instancia de una clase. Una clase es una descripción de objetos similares.
4. Un objeto hace cosas por sí solo o con la colaboración de otro objeto.
5. Un objeto es un conjunto de conductas y datos.
6. Un objeto es una cosa tangible (se puede percibir con alguno de los sentidos) que exhibe algún bien definido comportamiento.
7. Un objeto es una entidad que combina sus propiedades de procedimiento y datos, ya que realiza cómputos y guardan estados.
8. Los objetos existen, las clases son sus definiciones.
9. En el mundo real los objetos se limitan a existir pero en el interior del programa orientado a objetos cada objeto tiene una identidad única.
10. Un objeto tiene sentido en el contexto de una aplicación (Rumbaugh).
11. Los objetos tienen dos propósitos: promover la comprensión del mundo real y proporcionar una base práctica para una implementación por computadora.
12. La descomposición de un problema en objetos, depende del juicio y naturaleza del problema. No existe una única interpretación correcta.

Tipos de Objetos

- ***Actor***: es el que inicia cualquier acción
- ***Agente***: es un intermediario, un objeto "vago" ya que busca que otro objeto haga las cosas.
- ***Servidor***: es quién presta servicios

Por esto se puede decir que ***"un objeto hace cosas, por sí mismo ó con la colaboración de otro u otros objetos"***. Este concepto de colaboración es central.

Los Objetos según Booch

Según Grady Booch un objeto (instancia de una clase) es algo que puede definirse por 3 de sus características:

1. ***Estado***
2. ***Comportamiento***
3. ***Identidad***

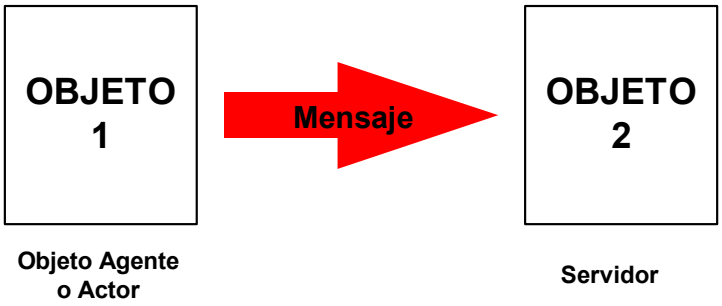
1. **Estado**: Todo objeto posee un estado que releva sus:

   - ***Propiedades Estáticas:*** son variables de instancia cuyos valores no se modifican o que se modifican con muy poca frecuencia.
     Ejemplo: par (Nombre, José)

   - ***Propiedades Dinámicas:*** son variables de instancia cuyos valores si cambian con gran frecuencia.
     Ejemplo: par (Edad, 56)

   Cabe recordar que "dato" es un par (atributo, valor), y que "dato" y "Variable de Instancia" pueden tomarse como sinónimos.

2. **Comportamiento**: es la conducta del objeto. Se exterioriza al ejecutar sus operaciones. Operaciones, métodos, funciones y responsabilidades son sinónimos.



3. **Identidad**: es algo que me define un objeto de manera única, que hace que "ese objeto sea ese y no otro". Pero esto no es nada parecido a la clave primaria de una entidad relacional. Si le mando a la clase "Manzana" el mensaje crear() dos veces, la clase va a "instanciar" dos objetos manzana y cada manzana es diferente de la otra (tal cual pasa en la realidad) sin necesidad de colocarle un numero de serie (que sería definirle una clave primaria). Esto demuestra claramente, otra vez, el acercamiento a la realidad que tiene el Paradigma Orientado a Objetos, acercamiento del que tanto hemos hablado.

Otras definiciones de Objetos:

Un Objeto puede definirse como tal si posee las siguientes características:

1. **Identidad:** (memoria dedicada): un objeto está en una única dirección de memoria y otro objeto está en otra. No puede haber dos objetos en la misma dirección de memoria (para decirlo más correctamente no pueden existir dos objetos que compartan una porción de la memoria reconocible por la dirección de la primer celda de memoria).
2. **Interfaz Pública:** captura la vista externa del objeto, el comportamiento observable del mismo.
3. **Implementación Oculta:** relacionado con los mecanismos internos que consiguen el comportamiento deseado del objeto.
4. **Semántica:** (la semántica es algo relacionado con el significado de las cosas): tiene que ver con el significado de las instancias producidas por cada clase en cada diseño, a pesar de que en dos ó más sistemas tengan la misma funcionalidad. Cabe aclarar que las entidades relacionales en la programación estructurada también hacían uso del concepto de semántica. Así decimos que un sistema de alumnos de una universidad tiene objetos

"Alumnos" y otro sistema de alumnos de otra universidad también tiene objetos "Alumnos". Tal vez dichos objetos tengan semánticas diferentes, en un caso representen alumnos de cualquier tipo (ingresantes, inscriptos, egresados, etc.) mientras que en otro caso representen solamente alumnos que están inscriptos en materias actualmente. Sin embargo, tal vez, también, ambos sistemas tengan funcionalidades parecidas: inscriben alumnos, emiten actas de exámenes finales, responden consultas sobre las notas de alumnos en materias, etc., etc.

Definición de Mensaje

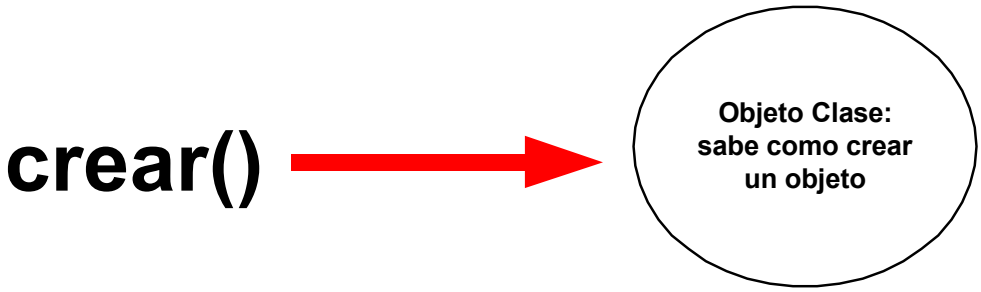
Un mensaje es una petición de un objeto a otro para que este se comporte de una manera determinada, ejecutando uno de sus métodos. Existen diferentes tipos de mensajes, entre ellos:

- ***Modificadores:*** alteran el estado del objeto (o sea modifican el valor de una o varias de sus variables de instancia).
- ***Selectores:*** devuelve una variable de instancia o varias variables de instancias pero dicha devolución no se muestra al graficar el mensaje, solo se muestra que el mensaje es enviado al objeto que devuelve uno o varios parámetros (datos) pero no se muestra ninguna línea ni flecha "de vuelta", esto significa que debe interpretarse que la devolución existe aunque no se muestre gráficamente.
- ***Iteradores:*** es un tipo de operación que permite acceder a varios objetos de una colección. Comunico un mismo mensaje a todos los objetos que componen una colección de objetos.
- ***Constructores:*** crean objetos.
- ***Destructores:*** eliminan un objeto

Creación de un Objeto – Definición de Clase

El objeto creador de objetos se conoce con el nombre de ***"Clase"***, y los objetos creados se denominan ***"instancia"*** o ***"ejemplares de clase".*** Una clase es simplemente un modelo que se utiliza para describir objetos similares, definiendo para ello los atributos y métodos de los objetos instancia de ella.
La clase es, por lo tanto, un objeto o una construcción sintáctica (marco de referencia).
A través de las clases se pueden crear objetos enviándole a la misma (sea un objeto –Smalltalk- o sea una estructura sintáctica –Java-) mensajes "crear ()".

También se puede crear un objeto utilizando métodos como la "clonación" o "copia" de un objeto "padre".



Todos los lenguajes orientados a objetos que usa la industria (del software) son "claseados" o sea que siempre un objeto es la instancia de una clase. No se liberó comercialmente ningún producto que use la "clonación" para crear objetos (lenguaje Self).

Búsqueda de la Operación: es implementada por cada lenguaje de una manera determinada, teniendo en cuenta que:

1. Cada objeto conoce la clase que lo creó.
2. Mientras viva dicho objeto no puede cambiarse de clase.

Observaciones:

- Para todos los objetos de la misma clase, las operaciones están una sola vez en la memoria (memoria principal del tipo Ram). Decir esto equivale a decir que el código (conjunto de líneas de código en realidad) que al ejecutarse desarrolla una operación definida para ese objeto está una sola vez en la memoria para todas las instancias de una determinada clase o sea que es como si la operación "estuviese" en el interior de la clase.
- Los mensajes siempre se envían a un objeto determinado, es decir que un objeto no envía un mensaje del tipo broadcasting que pueda ser "escuchado" por varios objetos sino que si un objeto quiere enviar un mismo mensaje a varios objetos lo debe hacer tantas veces como objetos deban ser destinatarios de ese mensaje. A las clases en general se les envía un solo tipo de mensaje que son los mensajes constructores o sea el mensaje es: "crear ()".
- Al decir que las clases "saben" cómo fabricar objetos agregamos que, de esta manera, estamos "representando conocimiento". Este concepto es muy importante también.
- Un objeto es una parte de la memoria dedicada a contener los valores de los atributos de ese objeto. Pero esa parte de memoria de un objeto NO contiene el código de cada una de las operaciones que puede realizar ese

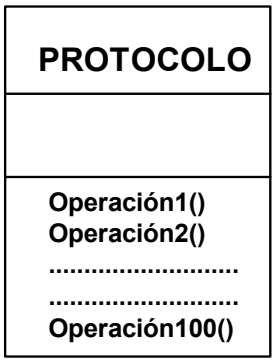
objeto sino que, como ya se dijo más arriba, las operaciones están (es como si estuvieran) en el interior de la clase que instanció al objeto.

Características Esenciales del Modelo de Objetos

Existen 4 elementos fundamentales en este modelo. Al decir fundamentales quiere decir que un modelo que carezca de alguno de ellos no es orientado a objetos. Dichos elementos esenciales son:

1. Abstracción
2. Encapsulamiento
3. Modularidad
4. Herencia- Jerarquía

1. **Abstracción:** es un concepto relacionado con la visión externa del objeto (*lo que es*), exterioriza lo esencial (me permite concentrarme en lo más importante). Un ejemplo de ello es el protocolo. Un protocolo es una lista de funciones u operaciones. El objeto es quien realiza dichas operaciones: operacion1 (), operacion2 (), operacion100 (). El protocolo me indica como invocar estas operaciones sin darme detalles de cómo se realizan las mismas. Es decir que el protocolo me dice cuantas operaciones puede realizar un objeto de una clase determinada y me detalla por cada una de esas operaciones la sintaxis del mensaje que debo enviarle para que sea ejecutada, los parámetros de entrada y los parámetros de salida. En todos los casos debe indicar el tipo de parámetro (string, número entero, fecha, etc.).



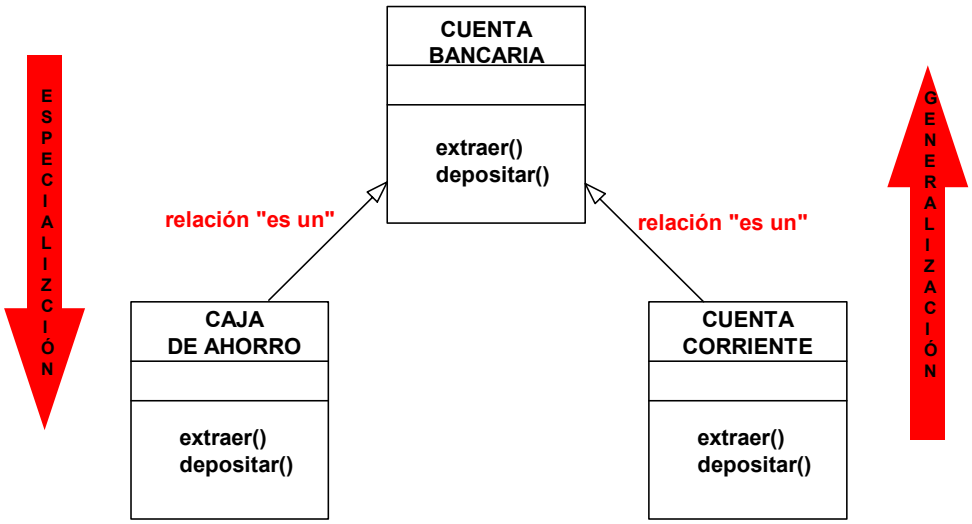
Muy Importante:

El Protocolo no es un objeto sino un manual de cómo "usar" ese objeto.

2. **Encapsulamiento:** puede visualizarse a través del concepto de ***"caja negra"***. El encapsulamiento persigue:
    - Ocultar aspectos internos (todos los datos –para simplificar digamos atributos- y todos los algoritmos que implementan las operaciones deben estar ocultas o encapsuladas).
    - Se debe poder modificar cualquier aspecto interno sin afectar a las aplicaciones que usen a los objetos. Es decir que puedo cambiar los atributos, cambiarles el nombre, eliminar algunos, agregar otros y el resto del sistema no se ve afectado por este cambio. Notar que en el modelo estructurado cuando se cambiaba un atributo había que hacer un análisis del impacto que este cambio podría producir en los módulos del software. También puedo cambiar algún algoritmo que implementa alguna operación que realiza un objeto reemplazándolo por otro algoritmo más eficiente (que haga las cosas más rápido por ejemplo) y, nuevamente, este cambio no afecta para nada al resto de los objetos que solamente envían mensajes al objeto que le estoy cambiando un algoritmo. Otra vez se debe comparar esto con el modelo estructurado en donde cada vez que cambio un módulo (que en el modelo estructurado representa la conducta, el comportamiento) debíamos hacer un análisis del impacto que este cambio puede llegar a producir en otros módulos ó en las entidades relacionales con las que trabaja dicho módulo, ahora eso es totalmente innecesario, ahora el modelo orientado a objetos es algo mejor preparado para enfrentar los cambios en los sistemas informáticos.
    - Reduce el nivel de acoplamiento porque desde el exterior no necesito saber qué hay en el interior, no necesito saber como se llaman los atributos de un objeto, solamente necesito conocer el protocolo entonces las relaciones (acoplamiento) entre el exterior y el objeto se reducen.
    - La "intercambiabilidad" de objetos. Yo puedo quitar del sistema un objeto y reemplazarlo por otro siempre que respete el protocolo del objeto anterior y al resto de los objetos que compone el sistema ese cambio no le hace ningún impacto, es como si no le hubieran cambiado nada porque le siguen mandando a ese objeto los mismos mensajes que le enviaban al "objeto anterior" y ni se deben enterar de que existía un anterior y uno nuevo. Cabe recordar que la única forma de acceder a un objeto debe ser a través de mensajes

3. **Modularidad:** me permite agrupar a los objetos en un paquete o subsistema. Un ***"subsistema"*** es un conjunto de módulos que puedo agrupar de acuerdo a algún criterio. De hecho, un módulo es un programa ejecutable y el menor módulo es aquel producido por la compilación de una sola clase.

4. **Herencia –Jerarquía**: es una relación que se establece entre dos ó más clases (subclases ó clases especializadas) con otra clase (superclase) de la cual heredan sus características. Las clases especializadas heredan todas las características que poseen las superclases (responsabilidades, atributos y algoritmos que implementan esas responsabilidades). Pero, obviamente, no pueden "heredar" la identidad.
   A su vez dichas clases pueden poseer comportamientos propios (especialización). Se dice que la superclase es una "generalización" del comportamiento de las clases hijas.
   Ejemplo:



Nota:

Aunque no se especifiquen en el dibujo de las subclases debe interpretarse que las subclases también tienen los mismos comportamientos y atributos que la superclase. En este caso se han especificado.

Observaciones:

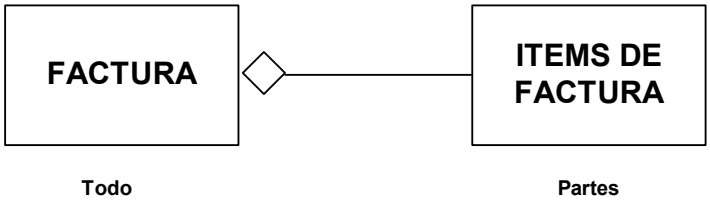
- La relación ***"es un"*** expresa jerarquía y herencia. También indica que eventualmente podría "reemplazar" la superclase con alguna subclase.
- No se pueden crear instancias de una clase abstracta. Las superclases no se pueden instanciar, entonces son clases abstractas.

En el gráfico anterior se puede visualizar con claridad que no es posible crear instancias de la clase "Cuenta Bancaria" porque no estaríamos representando la realidad. En un Banco no podemos pedir "Quiero que me abran una Cuenta Bancaria" aunque todos sabemos (clientes del banco y empleados del banco) qué es una Cuenta Bancaria. Entonces al decir que de las superclases no se pueden crear instancias estamos, nuevamente, representando lo que sucede en la realidad. En cambio, tal como pasa en la realidad, si podemos pedir que nos abran una "Caja de Ahorro" que *es una* "Cuenta Bancaria" y entonces decimos que tanto "Caja de Ahorro" como "Cta.Cte." heredan el comportamiento de una "Cuenta Bancaria", esto es que cualquier "Cuenta Bancaria" sea una Caja de Ahorro, o sea una Cuenta Corriente Bancaria, saben ***extraer ()*** y saben ***depositar ().***

Relaciones Entre Clases:

1. Herencia-Jerarquía
2. Agregación

Ejemplo de Agregación:



En algunos lenguajes como Smalltalk, al "todo" se le denomina "Colección". También se denominan "Contenedores" a objetos que representan el "todo"

El Polimorfismo

El polimorfismo es la obtención de resultados diferentes a partir de enviar un mismo mensaje a varios objetos.

Objetivos del Polimorfismo:

1. Reuso de código.
2. Buscar que los sistemas sean fáciles de modificar

Uno de los propósitos de las Metodologías (sean estructuradas o sean OO) es lograr desarrollar sistemas informáticos en los cuales cuando deban cambiar ese cambio se haga lo más fácilmente posible, en poco tiempo, con poco costo, sin provocar anomalías por el hecho de implementar el cambio.
Los sistemas informáticos cambian con gran probabilidad, con alarmante frecuencia (pensar en productos informáticos y en la gran cantidad de "versiones" de esos productos, cada versión es la implementación de un cambio) y por las siguientes causas:

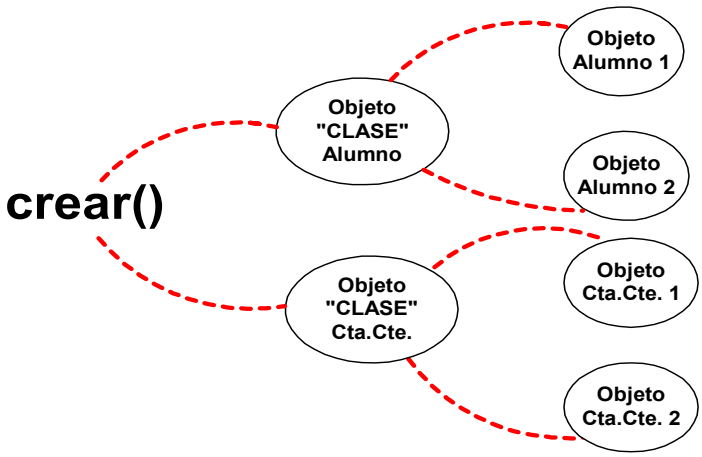
- Por cambios en los requerimientos del usuario (o porque se le quiere añadir nueva funcionalidad a la versión del producto).

- Por errores (ejemplo: se libera el Windows 98 y al poco tiempo se tiene que liberar el Windows 98 SE porque la versión anterior adolecía de una serie de defectos).

- Por cambios en la Tecnología Informática: esta es una de las causas más importantes de cambios: se cambia de plataforma, se cambia de arquitectura (de cliente-servidor se pasa a arquitectura de 3 capas), se cambia de Motor de Base de Datos (de Visual Fox Pro se pasa a Oracle), se cambia de Sistema Operativo (de Windows se pasa a Linux), etc.

O sea que las metodologías proponen hacer sistemas que sean fáciles de modificar, con poco esfuerzo, en poco tiempo, con el menor costo ya que la probabilidad de que los sistemas informáticos cambien es altísima.
El Paradigmas Orientado a Objetos consigue este propósito a través de la implementación de "Polimorfismo"

Ejemplo de Polimorfismo:



Nota:

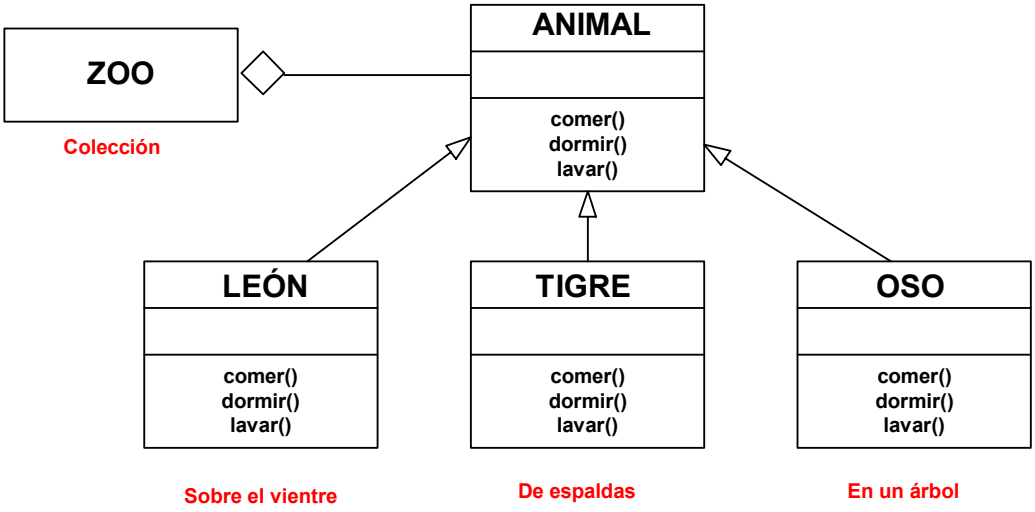
Este gráfico no está hecho respetando normas UML, pero se hace así porque aún no hemos visto UML y porque en UML no hay gráficos que "mezclen" objetos con clases.

"Crear" es un mensaje polimórfico. De acuerdo al objeto (en este caso objetos Clase) al que se le envíe dicho mensaje obtengo respuestas diferentes (comportamientos distintos). ***No debe perderse de vista que lo más importante de un objeto es su comportamiento.*** Es decir que le envío el mismo mensaje (crear()) a dos clases diferentes y obtengo resultados diferentes: se crean objetos "Alumno" y objetos "Cta.Cte."

Diferencias entre los Paradigmas de Programación

Al siguiente diagrama lo vamos a usar para podernos explicar la forma en que el Polimorfismo hace reuso de código en una forma en la que no se puede hacer en el paradigma estructurado.
El Paradigma Orientado a Objetos produce sistemas más controlados, más simples y más pequeños a través de la reutilización de mecanismos comunes (código), proporcionando así una importante economía de expresión (reutilización de código) y haciendo que los sistemas sean fáciles de modificar, que sea fácil la agregación de nuevas funcionalidades.

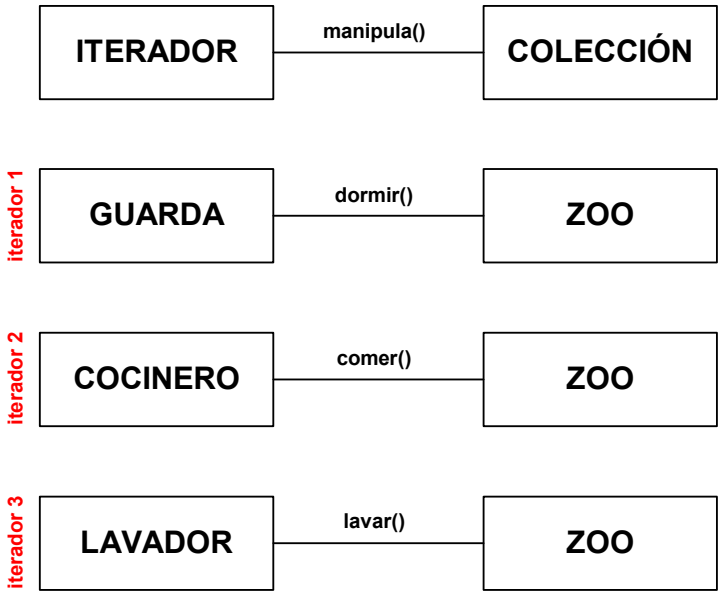
Diagrama de Clases en Paradigma Orientado a Objetos:



Decimos que entre Zoo y Animal hay una relación de agregación, o sea que el Zoo es una colección de animales, ó que el Zoo es un contenedor de animales ó que el Zoo está compuesto por animales.
También existe relación de jerarquía entre la superclase Animal con las subclases León, Tigre y Oso. Todos los animales saben comer, dormir y lavar (lavarse), nada más que cada especie de animal (*especialización*) lo hace en una forma particular. A modo de ejemplificar decimos que el León "duerme sobre el vientre", el Tigre lo hace "de espaldas" y el Oso duerme "en un árbol".

Debe notarse otra vez que no podemos mandarle el mensaje crear() a la superclase Animal ya que en la realidad no le podemos pedir a la naturaleza que produzca un Animal (que todos sabemos qué es lo que es un animal) hasta que no le indiquemos qué especie de animal es la que queremos. Entonces se vuelve a poner en evidencia el concepto de lo ***"abstracto"*** que es una superclase: no puede tener instancias. Por el contrario cualquier clase que si pueda tener instancias se dice que es ***"concreta".***

Iteradores: son objetos que ***"manipulan"*** a objetos ***"colección",*** o sea que le mandan mensajes "iteradores". Esto de "manipular" significa que es como si el objeto iterador le preguntase al objeto colección: ¿cuánto objetos componen tu colección?, la respuesta por ejemplo puede ser 3, entonces el objeto iterador le dice al objeto colección: "a tu objeto 1 mándale el mensaje x, a tu objeto 2 mándale el mensaje x, al objeto 3 mándale el mensaje x". Se ha puesto de manifiesto lo "iterativo" (repetitivo) que puede ser un mensaje.



En el ejemplo la clase "Animal" ha sido definida solo para poder heredar los comportamientos que, en común, tienen todos los animales, esto es que saben lavarse, saben cómo comer y saben como dormir.

Para representar que el zoológico acaba de incorporar animales de una nueva especie, por ejemplo perezosos, en el paradigma orientado a objetos solo tendríamos que:

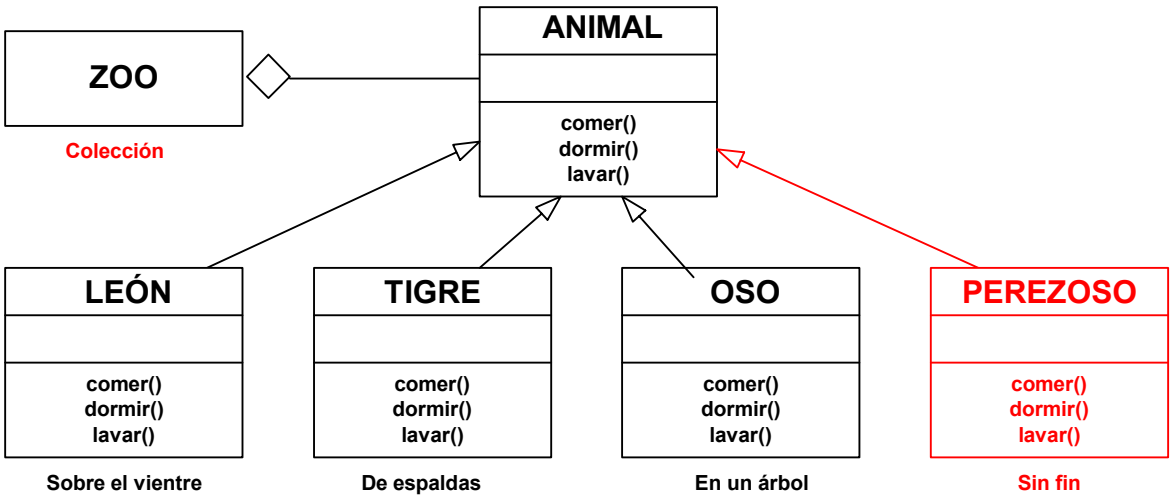
1. Definir una nueva subclase (por ejemplo: perezoso), escribiendo el código para cada comportamiento animal que esta especie implementa, o sea para que sepa como comer, como lavarse y como dormir. En el ejemplo siguiente decimos "sin fin" como forma de dormir que tienen todos los perezosos.
2. Hacer que esta nueva clase herede los comportamientos definidos en la clase madre "Animal", para hacer esto solamente tenemos que decirle al lenguaje orientado a objetos que la subclase perezoso es una subclase de animal.
3. Recompilar el sistema.

Y nada más, a partir de allí todo el sistema debe seguir funcionando sin problemas, sin tener que agregar sentencias en ninguno de los iteradores que eventualmente "manipulen" la colección de animales que tiene un Zoológico. Y ese es el código que estamos "re-usando", el que estaba en los iteradotes sigue funcionando tal cual estaba hasta antes de agregar la nueva especie. Se lo usa de la misma manera en que lo estábamos usando cuando el zoológico tenía solamente leones, tigres y osos, o sea lo re-usamos.

Para decirlo en otras palabras todo funciona como si a los objetos iteradores (Guarda que va jaula por jaula haciendo dormir a los animales del zoo, ó Cocinero

que va jaula por jaula dando de comer a los animales, ó Lavador que va jaula por jaula lavando a cada animal del zoológico) no les interesa que tipo de animal tienen que "atender" ya que cada animal "sabe" como comer, como dormir, como lavarse y entonces al iterador solamente le queda la tarea de ir jaula por jaula y decirle a cada animal: "hora de comer", "es tiempo de lavarse", "es hora de dormir" y cada animal cumple esa orden.

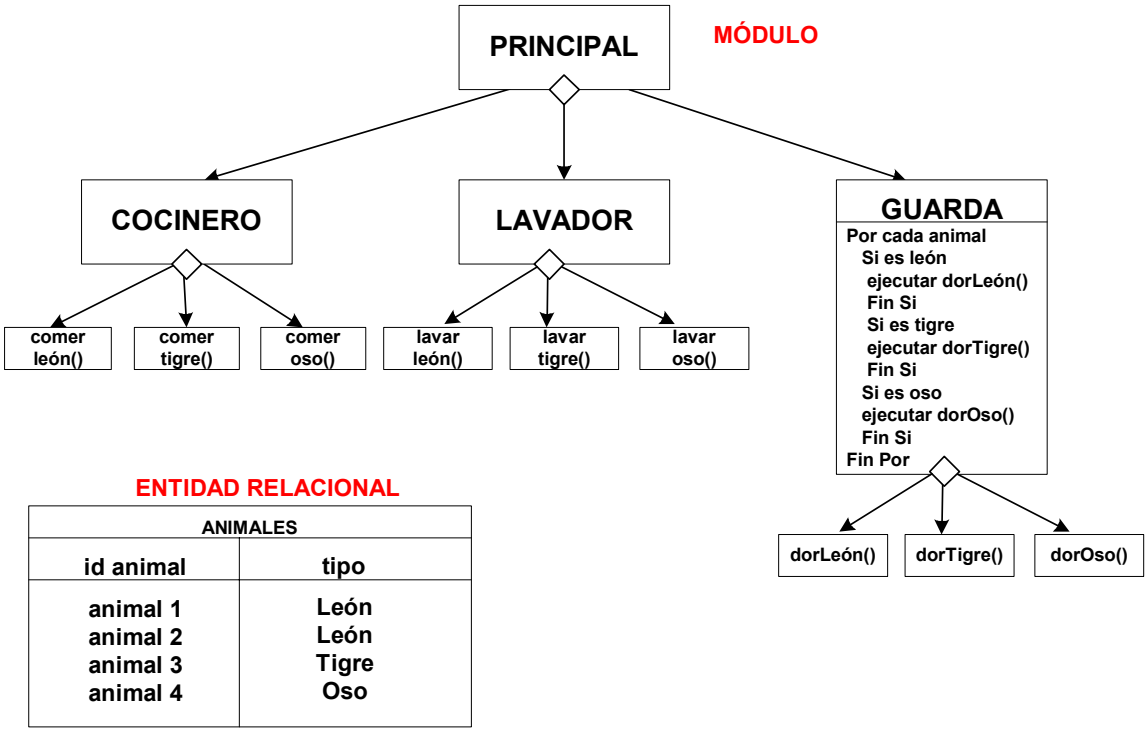
Operación de agregar una nueva especie en POO



La forma en cómo cada animal realiza cada una de las acciones definidas para todos los animales la explicitamos solamente para la operación de "dormir ()" diciendo, para seguir con un ejemplo, que el león "duerme" "*sobre el vientre"*, que el tigre lo hace "*de espaldas"* y que el oso duerme "*en un árbol"*. Al agregar una nueva especie (subclase) lo hacemos diciendo que el perezoso duerme "*sin fin"*.
Todas esas formas de dormir es lo mismo que decir que cada subclase ***"implementa"*** su operación y hay que tener idea de que esto de "implementar" es equivalente a un algoritmo o conjunto de sentencias fuente diseñadas para realizar una determinada operación.

A continuación analizaremos el mismo ejemplo, pero desde el punto de vista de la Metodología Estructurada

Modelo Estructurado (Diagrama de Estructuras y DER):



ENTIDAD RELACIONAL

ANIMALES	
id animal	**tipo**
animal 1	León
animal 2	León
animal 3	Tigre
animal 4	Oso

En este diagrama tenemos el módulo principal que es el que comanda la funcionalidad del sistema. Tenemos también algunos módulos de control (Guarda, Cocinero y Lavador) que "invocan" a módulos de trabajo: dorLeon (), dorTigre (), dorOso (), comerLeon (), comerTigre (), lavarLeon (), lavarTigre () y lavarOso (). Se ha puesto "()" para resaltar lo equivalente con los métodos u operaciones del Modelo OO ya que en el Diagrama de Estructuras no se usa el poner los paréntesis.

En el Diagrama de Estructuras está prohibido que dos ó más módulos tengan el mismo nombre.

Aunque, por razones de simplificación, no se ha escrito el código en lenguaje estructurado en los módulos Cocinero ni Lavador (cosa que sí se ha hecho en el módulo Guarda) se debe suponer que el código en ambos es similar al del Guarda. Decimos que hemos abierto la caja negra que es un módulo en el Diagrama de Estructuras aunque normalmente eso no se hace, lo hemos hecho para poder explicar lo que queremos. También es importante destacar que los nombres elegidos hacen llegar a la conclusión de que existe una total "equivalencia" entre estos 3 módulos de control con los "iteradores" del modelo orientado a objetos.

Es interesante destacar que en este ejemplo puede verse como cada uno de los 3 módulos de control accede a los datos que están en la entidad relacional "Animales" (entidad relacional que representa los datos de todos los animales que tengo en el zoológico). El código en Guarda muestra como el software va "recorriendo" cada instancia de "Animales" y analizando los valores de ciertos atributos descubre de cual especie animal se trata y en función de ese "descubrimiento" manda a ejecutar la función que corresponde a esa especie de animal. Nótese que aquí hay un bloque repetitivo (iterativo) que es el ***Por cada-Fin Por*** y 3 bloques de decisión (***Si-Fin Si***) respetando las reglas de la programación estructurada.

Podemos decir que aquí ***"el programador decide"*** que cosa hacer, a qué modulo de trabajo invocar para que se ejecute una acción determinada, en cambio en el modelo de objetos equivalente podíamos decir que ***"son los objetos los que deciden"*** como ejecutar la operación equivalente.

En el modelo de objetos decíamos que a "los iteradores" no les importaba de qué animal se trataba y sin embargo se lograba con el modelo hacer que los animales coman, duerman ó se laven, ahora en el modelo estructurado se logra el mismo efecto, obviamente, pero con la diferencia de que a los módulos de control (los que serían "los iteradores") ahora **SI** les interesa saber de qué especie de animal se trata.
Si quisiera en este modelo agregar una nueva especie debería:

1. Escribir los nuevos códigos que implementen las operaciones de dorPer (), comerPer () y lavarPer (), o sea escribir los módulos de trabajo "nuevos". Esto es lo mismo que lo apuntado en el punto 1 del modelo equivalente. Hasta aquí no hemos ahorrado nada, no hemos re-usado nada. Ambos modelos son equivalentes.

2. Modificar el código de todos "los iteradores" incorporando el bloque de decisión que analice que **SI** es un perezoso entonces ejecutar dorPer () (ó comerPer () ó lavarPer () según corresponda). Esto no lo tenemos que hacer en el modelo de objetos, esto es lo que nos ahorramos de hacer usando el polimorfismo en el paradigma de objetos.

3. Recompilar el Sistema.

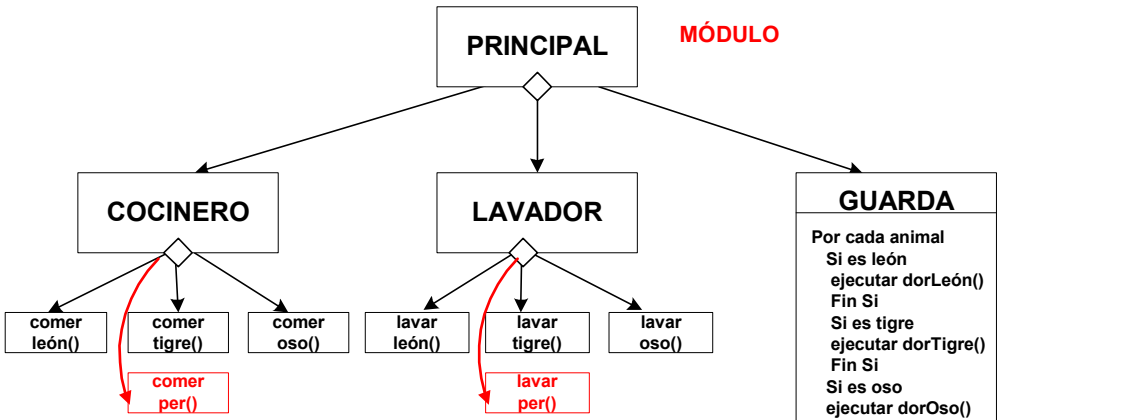
Y si nos llegamos a olvidar de modificar algún "iterador", entonces la modificación podría provocar errores derivados de haber agregado funcionalidad en el sistema. Esta necesidad de "recordar" cuales son todos los "iteradores" que recorren las instancias de la entidad "Animal" no existe en el modelo de objetos y por lo tanto no existe esa probabilidad de error.

Hemos logrado entonces dos cosas: 1) no tener que modificar el código de todos "los iteradores", por lo tanto hemos ahorrado esfuerzo; y 2) no tener necesidad de recordar cuales son todos los iteradores a modificar por lo que hemos reducido la

probabilidad de error. Es decir que hemos obtenido como recompensa por usar un paradigma que nos acerca más a la realidad "***Ahorro de esfuerzo***" (puedo hacer modificaciones en menos tiempo, con menos costo) y "***Eximición de probabilidad de error***".

La operación de agregar una nueva especie implicaría que por cada tabla relacional que tuviera que modificar, debería modificar también a todos los módulos que la utilizan. Esto se debe al acoplamiento intrínseco del modelo estructurado.

Operación de agregar una nueva especie (en el Paradigma Estructurado)



ENTIDAD RELACIONAL

ANIMALES	
id animal	**tipo**
animal 1	León
animal 2	León
animal 3	Tigre
animal 4	Oso
animal5	Perezoso

En el Modelo Estructurado los programadores (el software) deciden qué se debe hacer.
En el Modelo Orientado a Objetos los objetos deciden qué se debe hacer.

¿Qué es un Modelo?

Un modelo es una simplificación de la realidad. Los modelos se construyen para tener una mejor comprensión del sistema que se esta desarrollando. El objetivo del modelado guarda estrecha relación con el concepto de ***"Análisis"***

Un proceso de análisis es una tarea investigativa que tiene como propósitos:

1. Comprender el dominio del problema
2. Definir lo ***"que el sistema tiene que hacer"*** en el ámbito de la aplicación que el usuario haya definido.

El proceso de análisis involucra el estudio de los requerimientos, y este tema es tan importante que se ha formalizado una nueva disciplina que es la ***"Ingeniería de Requerimientos"*** para tratarlo con mayor rigor. Esta ingeniería efectiviza un conjunto de técnicas para la investigación y es una respuesta al hecho de que se ha medido que el 40% de los errores en los sistemas informáticos se *"introducen"* durante la definición de cuales son los requerimientos del sistema a construirse.

Es importante destacar que en los comienzos (aparición de las primeras metodologías) se distinguía una definida mayor importancia relativa por el Análisis (resumible con el QUE hacer) que por el Diseño (sintetizado en el COMO hacerlo). En la actualidad se nota que el proceso de construcción de software o metodologías para el desarrollo de software se concentran más en el diseño que en el análisis, y que, por su parte, dentro del proceso analítico se pone mayor énfasis en el Análisis de Requerimientos.

¿Cuál es el objetivo de modelar?

1. Me ayuda a visualizar un sistema tal cual es o como yo quiero que sea
2. Permite especificar la estructura o el comportamiento del sistema (los diagramas me permiten concentrarme tanto en la estructura del sistema como en su conducta).
3. Me dan una plantilla que me guía en el comportamiento del sistema
4. Documenta las decisiones que hemos tomado

Principios del Modelado

1. La elección de que modelo crear tiene una profunda influencia en como atacamos un problema y como delineamos una solución. La solución o respuesta depende, por tanto, de la complejidad del problema que tengamos en frente.
2. Cada modelo debe poder expresarse a diferentes niveles de precisión. El lenguaje de modelado (UML) permite que se pueda ***"ir refinando"*** el modelo a medida que se avanza en su desarrollo.
3. Los mejores modelos tienen una conexión con la realidad.
4. Un único modelo no es suficiente. Cualquier sistema es enfocado mejor con un conjunto de modelos independientes, pero relacionados. Por ejemplo puede existir un modelo orientado a la estructura del sistema y tener relación con otro modelo orientado al comportamiento, etc.

¿Qué es UML?

UML es un lenguaje de modelado, de propósito general, usado para la visualización, especificación, construcción y documentación de sistemas Orientados a Objetos.
Al ser definido como un lenguaje de propósito general, quiere decir que es un lenguaje que puede aplicarse a muchos tipos de sistemas, como por ejemplo algún sistema de soporte a las tareas de administración.
UML no es una Metodología de Desarrollo, no es un Proceso de Desarrollo de Software.

¿Cómo trabaja UML?

- ✓ A través de métodos y notaciones históricas
- ✓ A través del desarrollo del ciclo de vida del sistema
- ✓ A través de dominios de aplicación
- ✓ A través de lenguajes y plataformas de implementación
- ✓ A través de procesos de desarrollos
- ✓ A través de conceptos internos

¿Por qué UML es un lenguaje?

- ✓ Provee un vocabulario y reglas para combinar los elementos del vocabulario con el propósito de comunicar
- ✓ En un lenguaje de modelado esos vocabularios y reglas se focalizan en representaciones conceptuales y físicas de un sistema.

En, síntesis decimos que UML es un lenguaje porque realiza ***"todo lo que se supone debe realizar un lenguaje***": tiene sintaxis, vocabulario y reglas para combinar estos elementos con el objeto de comunicar.

UML: un lenguaje para documentación…

UML es un lenguaje que contempla a todos los ***"artefactos"*** que se producen en el desarrollo de software, entre ellos:

- Requerimientos
- Arquitectura
- Diseño
- Código
- Planes de Proyecto
- Pruebas
- Prototipos

Artefacto: un artefacto se define como todo aquel "***elemento"*** o ***"cosa"*** que se produce cuando se está trabajando en el proceso de desarrollo.
Ejemplos de artefactos son:

- Un diccionario de datos
- Un diagrama de USE CASE
- Un glosario de términos que se usan en el dominio.

Cabe aclarar que el término ***"artefacto"*** se utiliza para diferenciarlo de ***"objeto".***

Principales Elementos de UML

- ***Bloques Básicos de Construcción***
- ***Reglas para juntar los Bloques Básicos de Construcción***
- ***Mecanismos Comunes para aplicar UML***

1. Bloques Básicos de Construcción: conformado por los elementos, relaciones y diagramas en UML.

1.1. Elementos o Cosas

1.1.1. Elementos Estructurales: Clase, Clase Activa, Interface, Use Case, Colaboración, Nodo y Componente.
1.1.2. Elementos de Comportamiento: Interacción y Máquina de Estados
1.1.3. Elementos de Agrupación: Paquete (Modelo, Subsistema, Entorno)
1.1.4. Elementos de Anotación: Nota

1.2. Relaciones

1.2.1. Dependencia
1.2.2. Asociación
1.2.3. Generalización
1.2.4. Realización

1.3. Diagramas

1.3.1. Diagrama de Clases
1.3.2. Diagrama de Objetos
1.3.3. Diagrama de Componentes
1.3.4. Diagrama de Despliegue
1.3.5. Diagrama de Use Case
1.3.6. Diagrama de Secuencia
1.3.7. Diagrama de Colaboración
1.3.8. Diagrama de Transición de Estados
1.3.9. Diagrama de Actividad

2. Reglas para juntar los Bloques de Construcción:

denominadas también "reglas de combinación de los bloques básicos"

2.1. Reglas Semánticas para

2.1.1. Nombres
2.1.2. Alcance
2.1.3. Visibilidad
2.1.4. Integridad
2.1.5. Ejecución

2.2. Reglas para que los modelos sean

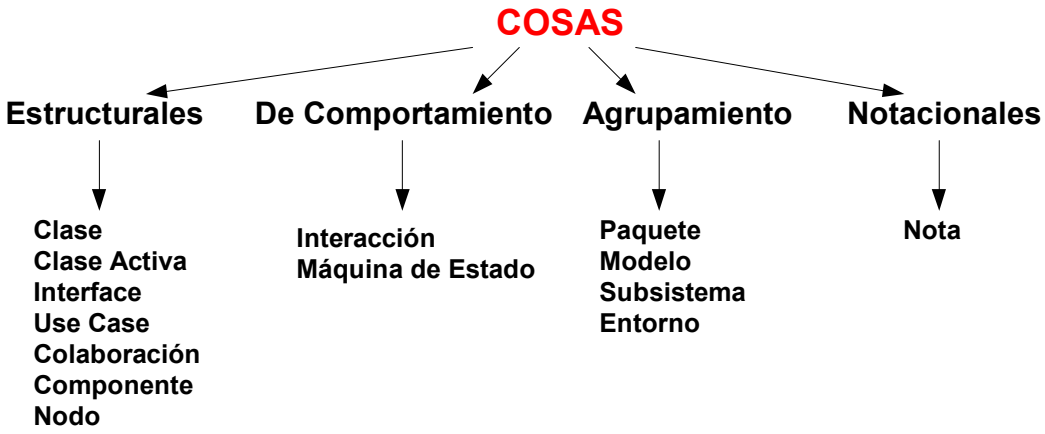
2.2.1. Abreviados
2.2.2. Incompletos
2.2.3. Consistentes

3. **Mecanismos comunes para aplicar UML:** Por ejemplo en todos los modelos los atributos deberían ser privados (por defecto) y los métodos o conductas deberían ser públicos (por defecto).
   3.1. Especificaciones
   3.2. **Adornos:** Pública, Privada, Protegida.
   3.3. **Divisiones Comunes:** Clase y Objeto, Use Case e Instancia de Use Case, Componente e Instancia de componente, Interfaz e implementación.
   3.4. **Mecanismos de Extensibilidad:** Estereotipo, Valores etiquetados, Restricciones.

Bloques Básicos de Construcción de UML

Los Bloques Básicos de Construcción de UML están compuestos por:

1. **Cosas ó elementos estructurales:** se dice que son los "ciudadanos de primera clase en un modelo". Obviamente son abstracciones.
2. **Relaciones:** ligan las cosas ó elementos de UML.
3. **Diagramas:** Agrupan colecciones interesantes de cosas ó elementos relacionados entre sí.



Reglas de UML

Las Reglas están relacionadas con el concepto de semántica (el significado de las cosas). Establece reglas semánticas para:

- **Nombres:** cómo denominar las cosas ó elementos, cómo denominar las relaciones y qué nombres poner a los diagramas.
- **Alcance:** el contexto que da significado específico a un nombre.
- **Visibilidad:** me permite, por ejemplo visualizar solo el nombre de la clase mientras suprimo sus atributos.
- **Integridad:** cómo se relacionan apropiada y consistentemente unos elementos con otros
- **Ejecución***:* son las reglas que se siguen para ejecutar tareas dentro de un UML, o sea qué significa ejecutar o simular un modelo dinámico.

A su vez las reglas de UML, contemplan la construcción de modelos:

- **Suprimidos:** por ejemplo suprimo los atributos en el modelo.
- **Incompletos:** el modelado en UML acepta que no defina "comportamientos". Esto último genera inconsistencia dentro del modelo, ya que como habíamos definido anteriormente ***"lo más importante de un objeto es su comportamiento".*** La forma de lograr un modelo consistente tiene que ver con aplicar el principio del modelado que dice que ***"cada modelo puede expresarse a diferentes niveles de precisión".*** Esto me permite ir eliminando inconsistencia a medida que avanzo en el refinamiento del modelo.
- **Inconsistentes:** misma idea que la explicada en el párrafo anterior. UML permite ***"inconsistencias"*** en los modelos pero eso no quiere decir que UML sostenga que está bien hacer modelos "inconsistentes" sino que es una manera de ir logrando la consistencia que debe lograrse a través de pasos sucesivos.

Mecanismos Comunes de UML

Compuestos por:

- **Especificaciones:** son descripciones textuales de lo que cualquier elemento gráfico de UML expresa. La notación gráfica de UML se utiliza para visualizar un sistema; una especificación de UML se utiliza para enunciar los detalles del sistema.

- **Adornos:** son "complementos" que ayudan a comprender mejor lo que estamos graficando, por ejemplo los adornos más conocidos son:
  - ✓ + Operación pública
  - ✓ # Operación protegida
  - ✓ - Atributo privado

- **Divisiones Comunes 1:** Cuando modelamos con Orientación a Objetos notamos que siempre aparecen ciertas cosas de ***"a pares",*** por ejemplo cuando hablamos de instanciación:
  - ✓ Un objeto es la instancia de una clase
  - ✓ Un use case es una instancia de un use case
  - ✓ Específicamente un escenario (de éxito ó de fallo) es una instancia de use case.
  - ✓ Un componente es una instancia de un componente.
  - ✓ Un nodo es una instancia de un nodo.

- **Divisiones Comunes 2:** Otra "dicotomía" que podemos encontrar es la de Interfaz-Implementación. La interfaz declara un contrato mientras que la implementación representa una realización concreta de ese contrato. Ejemplos:
  - ✓ Casos de Uso con Colaboraciones
  - ✓ Interfaz con Clase
  - ✓ Interfaz con Componente

- **Mecanismos de Extensibilidad:** Estereotipos, Valores Rotulados y Restricciones.

  - ✓ ***Estereotipos:*** es una especie de herencia de conceptos que existían antes del "acuerdo" que, entre metodólogos, significó UML. Por ejemplo Jacobson en su libro OOSE (Object Oriented System Engineer) define ciertas categorías ó tipos de objetos que son: 1) Objetos de entidad; 2) Objetos de Interfaz de un Sistema Informático con el usuario (ojo que no es la Interfaz como conjunto de operaciones que implementa una clase ó un componente) y 3) Objetos de control. Esta categorización resultaba muy conveniente porque eran la base para sustentar un diseño arquitectónico en 3 capas. Otro ejemplo de estereotipo es cuando para representar un actor que representa a un sistema informático no uso el gráfico del "muñequito" sino un rectángulo que representa una clase y coloco el estereotipo <<actor>> dentro del rectángulo para denotar que se trata de eso, de un actor. Por último están los estereotipos <<extiende>> e <<incluye>> usados para poner en evidencia "Relaciones de Dependencia" entre 2 Casos de Uso.

  - ✓ ***Valores rotulados:*** cuando se colocan, por ejemplo, en la ventana donde se pone el nombre de la clase texto que aclara ciertas cosas importantes como por ejemplo en algún caso pongo {versión 3.2} para evidenciar que esa clase ***"se agregó"*** a partir de la versión 3.2 o {persistente} para denotar que esa clase es persistente. Siempre estos valores rotulados se colocan entre llaves.

- ✓ ***Restricciones:*** otra vez se trata de "aclarar" cosas en un diagrama. Por ejemplo pongo la restricción {ordenado} al lado de una operación *agregar()* para poner en evidencia que el agregado debe hacerse respetando cierto ordenamiento.

Bloques Básicos de Construcción de UML

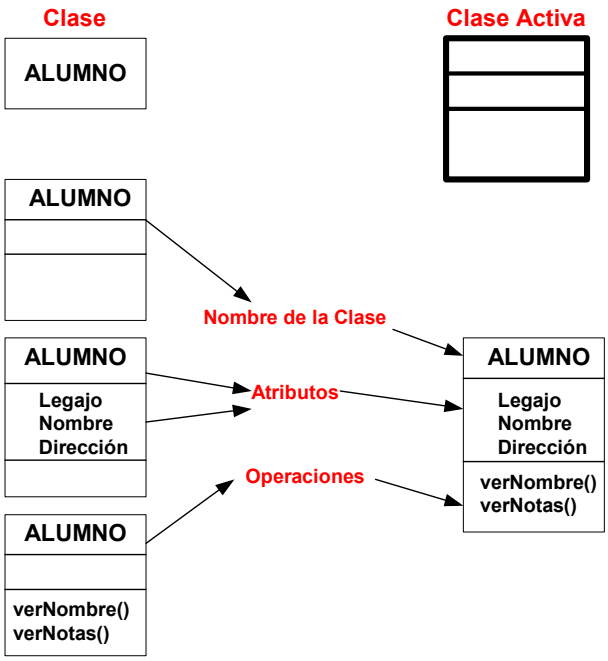
1. **Elementos ó Cosas Estructurales**
   - 1.1 Clase
   - 1.2 Clase Activa
   - 1.3 Use Case
   - 1.4 Colaboración
   - 1.5 Componente
   - 1.6 Interface
   - **1.7** Nodo
2. **Elementos de Comportamiento**
   - 2.1 Interacción
   - 2.2 Estado
3. **Elementos de Agrupación**
   - 3.1 Paquete (es el único a estudiar)
   - 3.2 Modelo
   - 3.3 Subsistema
   - 3.4 Entorno
4. **Elementos de Anotación.**
   - 4.1Nota

Elementos o Cosas Estructurales

Clase: gráficamente se representa con un rectángulo y el nombre de la clase en su interior. Es uno de los 7 elementos básicos de UML. En una clase los atributos deben ser privados (característica relacionada con el encapsulamiento) y los métodos públicos (relacionado con el concepto de abstracción y protocolo).

Clase Activa: es la clase que lleva el hilo de ejecución del programa. Las clases activas representan los módulos de control, o los programas principales (main).

Ejemplo:



En el gráfico anterior se pueden ver diferentes maneras de representar la clase "Alumno", en algún caso solamente está la ventana con el nombre de la clase y están ***"suprimidos"*** los atributos y las operaciones; en otro caso solamente se "suprimen" los atributos; en otro caso se "suprimen" los comportamientos y, por último en otro caso hemos graficado la clase ***"con visibilidad completa"*** es decir con las tres ventanas: la del nombre de la clases, la que contiene los atributos y la que contiene las operaciones.
También hemos dibujado una ***"clase activa"*** como puede ser manejadorDeNotas () en un sistema de alumnos.

Use Case: es un objeto que describe todas las tareas que se realizan alrededor de un proceso. Representa un conjunto de actividades de un proceso de negocio ó de un proceso del sistema de información automatizado (que sería un "proceso de computadora").
Siempre debe distinguirse a los "Procesos de Negocios", de los "Procesos de Computadora".
Un ***Proceso de Negocios*** es un conjunto de tareas que producen como resultado algo que el negocio hace para un determinado cliente, el resultado de ese proceso es "algo de valor" para el cliente. Se concentra en la visión del cliente.
Cuando un USE CASE representa un ***"Proceso de Computadora",*** decimos que ante cierto evento se ***"instancia"*** ó se ***"crea"*** un objeto de tipo "USE CASE". Este evento puede ser, por ejemplo, el hacer clic (ó doble clic) con el Mouse, con la flecha del mismo apoyada sobre un icono de un producto, por ejemplo el Word ó el Internet Explorer. Ese evento produce la ***"instanciación"*** de un Use Case o sea

que con el Word ó con el Internet Explorer voy a poder realizar una serie de tareas que tengan algún valor para mí –usuario del sistema-.

Ejemplo:
El USE CASE ***"Inscribiendo alumno"*** comienza cuando el alumno se dirige a la bedelía y le comunica al empleado que lo atiende su necesidad de inscribirse. Este hecho dispara el USE CASE (proceso de negocios) y que hace que se ejecuten un conjunto de tareas que le permitan al alumno inscribirse (satisfacer el requerimiento del cliente de la universidad, en este caso el alumno). Decimos que en este caso el "sistema" es la universidad.
El USE CASE (proceso de computadora) se inicia en cambio cuando el usuario (en este caso quién inscribe al alumno –el bedel, por ejemplo-) hace que se ***"instancie"*** un Use Case, por ejemplo haciendo doble clic con el mouse u oprimiendo la tecla <Enter> en algún momento dado, para ***"habilitar"*** la ejecución por parte de la computadora de algún proceso que le permite al alumno inscribirse. Decimos ahora que el ***"sistema"*** es la aplicación que se ha desarrollado y que corre en una ***"computadora".***

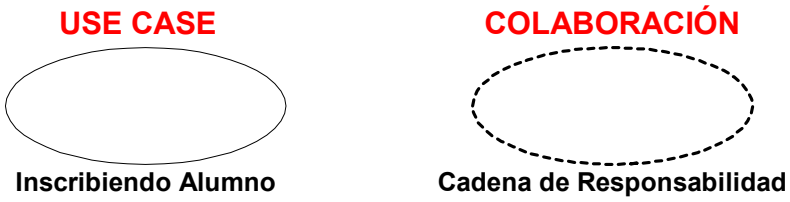
Importante

Los "Procesos de Negocio" son soportados por los Sistemas de Información que por eso se llaman "Sistemas de Información de Soporte a los Procesos de Negocio". Como nosotros trabajamos en construir Sistemas Informáticos o sea Sistemas de Información automatizados, los procesos del Sistema de Información serán "Procesos de Computadora", y todo el Sistema de Información se deriva a partir de los procesos de negocios haciendo uso del Proceso Unificado de Desarrollo con UML.

Un Use Case se relaciona con un actor mediante relaciones de asociación y esa relación es bidireccional para representar la ***"conversación***" entre el actor y el use case que, en nombre del "sistema" lo "atiende". En esta definición el sistema puede ser el negocio ó una aplicación en una computadora.
Un Escenario es una secuencia específica de acciones e interacciones entre los actores y el sistema objeto de estudio, también se lo denomina instancia de use case. Por ejemplo: un Escenario de Éxito es el de la compra de artículos con pago en efectivo que termina cuando el actor (cliente) se lleva una mercadería que ha pagado con dinero en efectivo; y un Escenario de Fracaso ó de Fallo es cuando se intenta pagar con tarjeta de crédito los artículos que se quieren comprar y el <<actor>> ***"Administradora de Tarjetas de Crédito"*** no aprueba la operación.
Se puede decir, entonces, que un use case es una colección de escenarios con éxitos y fallos relacionados, que describe a los actores utilizando un sistema para satisfacer un objetivo.
Los Use Case definen una promesa o contrato de la manera en que se comportará el sistema ante cierto requerimiento por parte de un actor. Se especifica QUE

debe hacer el sistema (idea de definir requisitos funcionales, recordar la frase que dice que Análisis es equivalente a QUE) sin decidir COMO el sistema hará para cumplir esa promesa o contrato (Diseño es equivalente a COMO, a dominio de la solución).
Un use case captura TODO y SOLO el comportamiento relacionado con la satisfacción de los intereses de los actores involucrados. Un Actor, por su parte, es algo con comportamiento, como una persona (identificada por un rol), o un sistema informatizado u organización. Ejemplo de actores: un cajero, un cliente, un sistema de Administración de Tarjetas de Crédito, etc.

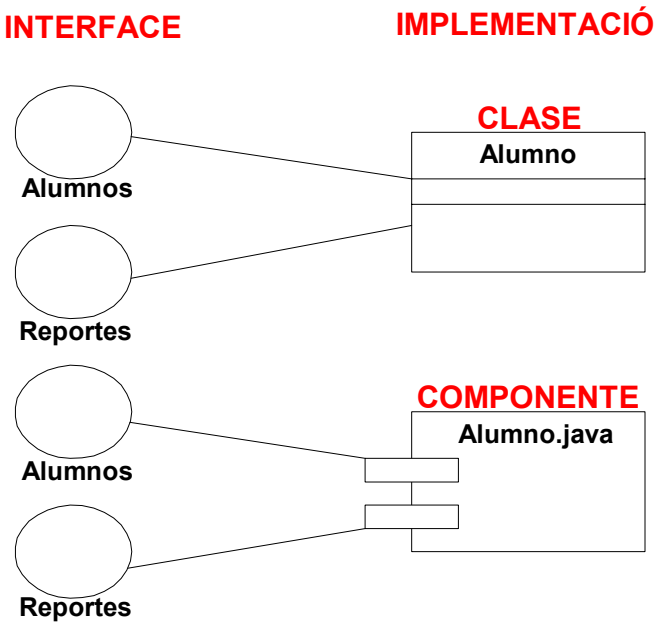
Colaboración: es un conjunto de objetos que se mandan mensajes entre sí y de esta forma realizan todas las tareas descriptas en el USE CASE. A través de los mensajes ***"efectivizan"*** las actividades del mismo.
Esto es así ya que un USE CASE es netamente descriptivo, por lo tanto por cada USE CASE siempre esta presente una colaboración. El conjunto de tareas descriptas en todos los USE CASE que componen un sistema representa toda la funcionalidad del sistema.
La colaboración se representa mediante un óvalo conformado por una línea de puntos.



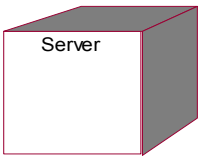
Componente: es una parte física reemplazable que empaqueta su implementación y es conforme a un conjunto de interfaces a las que proporciona su realización. A diferencia de los paquetes que se dice que son algo puramente conceptual (o sea que sólo existen en tiempo de desarrollo), los componentes son cosas físicas y existen en tiempo de ejecución.
Este concepto de "componente" tuvo una gran importancia en algún momento y era la base para pensar que diseñar software era disponer de "componentes" tal como los diseñadores de hardware lo hacen (por ejemplo al diseñar una aplicación como puede ser el diseñar un circuito impreso constituido por chips relacionados por líneas por donde circulan señales o pulsos eléctricos).
Se puede afirmar que el gran éxito de Visual Basic es que puso esta idea en acción (aparte de ser un producto "conducido por u orientado a eventos") y la velocidad de construir software con Visual Basic en muchísimos casos depende de la experiencia del diseñador en buscar "componentes" (DLL's por ejemplo) que realicen determinadas cosas (o sea que Implementen ciertas interfases). Pero atención que no estamos diciendo que Visual Basic es un lenguaje orientado a objetos.

Frederick Brooks escribió un artículo ("No silver bullet: essence and accidents of Software Engeneering") en el que, pesimistamente, pronosticaba que la ***"crisis del software"*** nunca iba a ser superada, que no existía una bala de plata que pueda matar los problemas de desarrollar software sin defectos.
En respuesta a este pensamiento Brad J. Cox escribió "Existe una bala de plata!" (Los "componentes" de los que estamos hablando) en donde dice que para hacer software confiable y con cero defectos había que hacer las cosas tal cual la hacen los diseñadores de hardware: ir a un negocio que venda chips (en nuestra ciudad hay una serie de ellos en la zona de la calle Rivera Indarte y La Rioja), seleccionar de allí los que necesite basándose en los manuales donde se explica qué hacen los chips (sin saber cómo lo hacen) y poniendo en juego este concepto de protocolo o interfase que ya hemos desarrollado en clases.
Incluso Brad Cox fundó una compañía que se dedicó a la "venta de componentes de software", pero no funcionó. Los desarrolladores de software no tienen los mismos puntos de vista que sus colegas del hardware.

Interface: se representa mediante un círculo atado a una clase. Es un subset del protocolo de la clase, es decir contiene solo unas cuantas funciones del mismo.

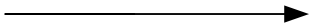


Nodo: sirve para representar "cosas físicas". Representa, por ejemplo, la arquitectura del hardware sobre la que se monta la arquitectura del software.

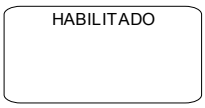


Elementos o Cosas de Comportamiento

Interacción: son líneas que "conectan" los estados. Pueden ser mensajes, links, secuencias de acción, etc. Las líneas de acción hacen que un objeto cambie de un estado a otro.

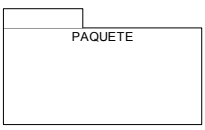


Estado: es el valor de un atributo de un objeto que representa un estado del mismo. Por ejemplo para un objeto "Alumno", este puede tener un atributo "condición", cuyos valores podrán ser: libre, regular o condicional (diferentes estados). Para un terminal de punto de venta el estado puede ser:



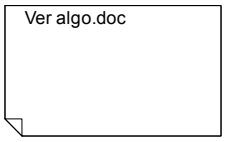
Elementos o Cosas de Agrupamiento

Paquete: es el contenedor de una colección de objetos. Dicha colección puede incluir: entornos (frameworks), modelos, otros subsistemas, archivos ejecutables, archivos.txt, etc., y sirve para organizar elementos en grupos. Una característica de los paquetes es que puedo incluir en su interior lo que desee. Un ejemplo claro de paquete lo constituyen los ***"Service Pack".*** Alguna bibliografía cita que un paquete es puramente conceptual (o sea que sólo existe en tiempo de desarrollo) al contrario de un componente que es físico, pero el Ing. Freddy Díaz opina que no es así, que no solo existe en tiempo de desarrollo y que es algo físico como en el caso del ejemplo citado.



Elementos o Cosas de Anotación

Notas: las cosas "notacionales" sirven para hacer comentarios o notas que agreguen comprensión al tema tratado. Las ***Notas*** se ubican dentro del diagrama de clases (o en cualquier otro diagrama) y me permiten aclarar o realizar consideraciones importantes para el modelo. Las notas pueden ser aclaratorias (texto simple), links a otros documentos, URL embebidas, etc.



Relaciones en UML

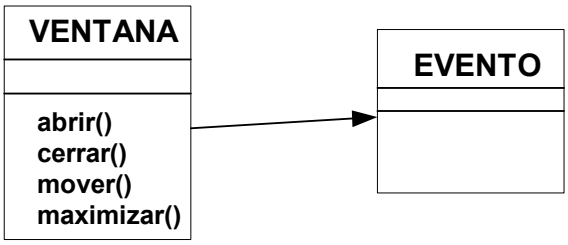
UML soporta cuatro de tipos de relaciones entre clases:

1. ***Dependencia***
2. ***Asociación***
3. ***Generalización (herencia- jerarquía)***
4. ***Realización***

Propiedad de las Relaciones: ***"Si la clase A existe y esta relacionada con la clase B, entonces las instancias de la clase A, también van a estar relacionadas con las instancias de la clase B".*** Es decir que las relaciones entre clases también es algo que "heredan" los objetos como instancias de cada clase. Es como si se dijese que si mi papá es *amigo* del Sr. Pérez y el Sr. Pérez tiene un hijo que se llama Pepe entonces yo también tengo que ser *amigo* de Pepe.

1. **Dependencia:** la ocurrencia o aparición o ejecución de operaciones de un objeto que es instancia de una determinada clase depende de la ocurrencia o aparición o instanciacion de un objeto de la clase evento (en el ejemplo de más abajo). También la dependencia se puede establecer con el cambio de estado no solo con la instanciación. No es muy usado por nosotros en los modelos que hacemos en los trabajos prácticos que propone la cátedra. En el ejemplo que se cita a continuación el hecho de que aparezca el objeto ventana depende de la ocurrencia de un evento en particular.

Ejemplo:



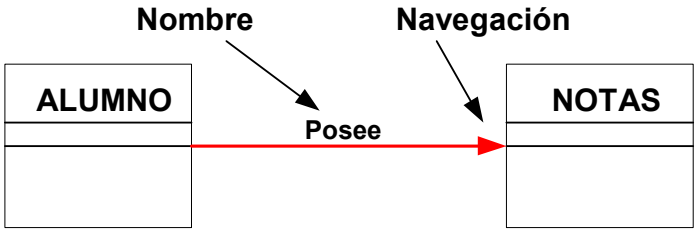
2. **Generalización (herencia- jerarquía):** en este tipo de relaciones las superclases ***"generalizan"*** los comportamientos de las subclases. (De modo inverso se comporta la "especialización", en donde las subclases ***"especializan"*** el comportamiento de las superclases de las que heredan dichos comportamientos) Si hablamos de Herencia debemos distinguir entre:

   - ***Herencia Simple:*** las clases heredan de una única superclase.

   - ***Herencia Múltiple:*** permite la herencia de múltiples clases. Cuando se implementa "Herencia Múltiple" deben enfrentarse problemas como: la herencia de atributos de diferentes clases pero con el mismo nombre. Entonces si se hereda el Atributo A desde 2 superclases tenemos dos casos: 1) si A tiene la misma significancia estaría heredando la misma cosa 2 veces y 2) si A es un número en una superclase y es un string en la otra superclase entonces deberemos de alguna manera hacer que se hereden ambas cosas (porque son diferentes) pero distinguiéndolas de alguna manera. Otro problema es que la relación ***"es un"*** puede no aplicarse, si decimos que un "Vehículo anfibio" tiene herencia múltiple desde "Vehículo terrestre" y desde "Vehículo acuático" entonces la relación "es-un" es cierta para ciertos casos, o sea un "Vehículo anfibio" es-un "Vehículo terrestre" a veces.

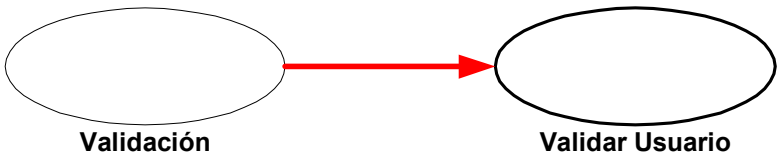
3. **Asociación:** es la relación más usada de todas. Asocia dos objetos mediante una línea que representa dicha relación generalmente en un solo sentido (navegabilidad). Cabe aclarar que en nuestros diseños con OO utiliza solo excepcionalmente las relaciones "ida y vuelta". El concepto de ***"Navegabilidad"*** es similar ó equivalente (parecido pero no igual) al de ***"Cardinalidad"*** en las entidades relacionales. Implica que un determinado objeto A "conoce" a otro determinado objeto "B". La navegabilidad en las relaciones entre A "hacia" B es como decir que A *"conoce"* a B y ¿para qué lo conoce?, para poder enviarle mensajes, para requerir sus servicios.
   En el ejemplo que ponemos a continuación decimos que el Alumno "conoce" sus notas para decirle por ejemplo: "Decime que notas obtuve en los parciales de DSI en el curso 3K7".

Ejemplo:



4. **Agregación:** (ya vista). Es un tipo especial de relación de Asociación. Representa conexiones lógicas o físicas entre una "Colección de Objetos" o "Contenedor de Objetos" y las partes que componen dicha colección. Es una relación todo-partes. Lo único que podemos agregar ahora a lo que ya estudiamos es que cuando el rombo que está del lado del objeto que representa el todo es un rombo "vacío" representa una conexión lógica en cambio cuando el rombo está "relleno, todo pintado de negro" eso indica una conexión física entre el todo y sus partes como por ejemplo un automóvil que está compuesto por partes (carrocería, motor, etc.).

5. **Realización:** no es muy usada, solamente la usamos en los Use Case y está asociada con el concepto de "colaboración". Ya hemos definido lo que es "colaboración" ahora agregamos entonces que la relación que existe entre un Use Case y una Colaboración es una relación de "Realización". En este tipo de relaciones existe una clase que solicita a otra (interface), la realización de una validación especifica.



Nota:

En UML la flecha que representa una "realización" es una línea de puntos.

Diagramas en UML

A la hora de desarrollar sistemas es necesario contar con herramientas que nos permitan visualizar las necesidades del proyecto o la organización desde diferentes puntos de vista, en las diferentes etapas por las que va atravesando un proyecto.
Como es sabido mientras se avanza en un proyecto, surgen cambios, nuevas necesidades o requerimientos, las cuales se hace necesario modelar para comprender mejor el sistema que se está desarrollando, para reducir la complejidad, para simplificar la realidad…
UML permite la construcción de modelos a partir de bloques de construcción básicos tales como clases, interfaces, colaboraciones, componentes, nodos, dependencias y relaciones entre dichos bloques.
Un diagrama es un elemento que permite visualizar los diferentes bloques de construcción, a la vez que combinarlos para adecuarnos a las necesidades de la organización.
Los diagramas se utilizan para visualizar un sistema desde diferentes perspectivas. Como ningún sistema puede ser comprendido completamente desde una perspectiva, UML define varios diagramas que permiten centrarse en diferentes aspectos del sistema independientemente, no obstante cabe aclarar que, al igual que sucedía con el Proceso Unificado de Desarrollo, la elección del conjunto adecuado de diagramas para modelar un sistema depende de los requisitos y requerimientos del proyecto en particular.

Tipos de Diagramas

1. **Estructurales:** ponen en evidencia la estructura del sistema. Existen para visualizar, especificar, construir y documentar los aspectos estáticos de un sistema. Los elementos estáticos de un sistema incluyen la existencia y ubicación de clases, interfaces, colaboraciones, componentes y nodos.

2. **De Comportamiento:** evidencian el comportamiento del sistema. Existen para visualizar, especificar, construir y documentar los aspectos dinámicos de un sistema. Los elementos dinámicos de un sistema involucran cosas tales como el flujo de mensajes a largo del tiempo y el movimiento físico de componentes en una red.

Diagramas Estructurales

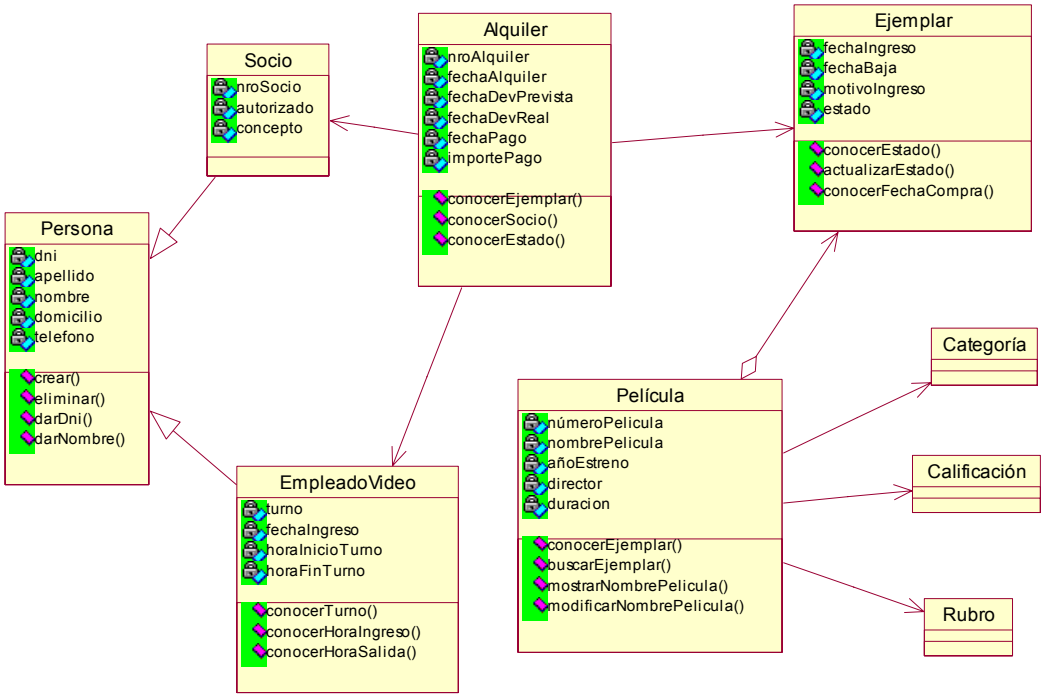
Los diagramas estructurales de UML se organizan en líneas generales alrededor de los principales grupos de elementos que aparecen al modelar un sistema:

1. ***Diagrama de Clases***
2. ***Diagrama de Objetos***
3. ***Diagrama de Componentes***
4. ***Diagrama de Despliegue***

1. Diagrama de Clases

Los diagramas de clases muestran un conjunto de clases, interfaces y colaboraciones y sus relaciones. Me permite conocer cuantas clases tiene el sistema y como se relacionan entre sí. Es un diagrama muy importante y no debería existir ningún proyecto que no cuente con él.
En un Diagrama de Clases se describen todas las clases de todos los use case de un sistema y, consecuentemente, todas las responsabilidades definidas en todos los use cases de un proyecto.

Ejemplo: Diagrama de Clases de un Video Club



Este ejemplo representa un Diagrama de Clases realizado en algún momento del Proceso de Desarrollo de un Sistema de Video Club y que pone en evidencia:

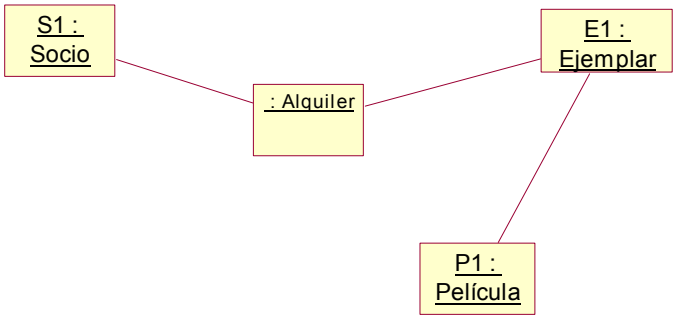
- Que la clase Alquiler es la que "conoce" a casi todas las otras clases, que ocupa una posición "central" en el diagrama y que de esta manera el diseñador está representando, de alguna manera, la esencia del negocio de un video club que es la de alquilar cassettes de video conteniendo películas grabadas.
- Que la mayoría de las relaciones son de asociación.
- Como este diagrama está importado desde el Rational Rose se notan los "adornos" que, por defecto, se colocan a los atributos (privados) y a las Operaciones (públicas).

2. Diagrama de Objetos

Un diagrama de objetos muestra un conjunto de objetos y sus relaciones. Un diagrama de objetos puede visualizarse como "una foto de la memoria", del estado de un objeto en un determinado momento. Es decir, que dichos diagramas me permiten ver como se representan cada una de las instancias de una clase en un momento en particular. Cabe aclarar que es un diagrama menos explícito que un diagrama de clases.
Los diagramas de Objetos contienen comúnmente:

- Objetos
- Conexiones (links)

Ejemplo: Diagrama de Objetos

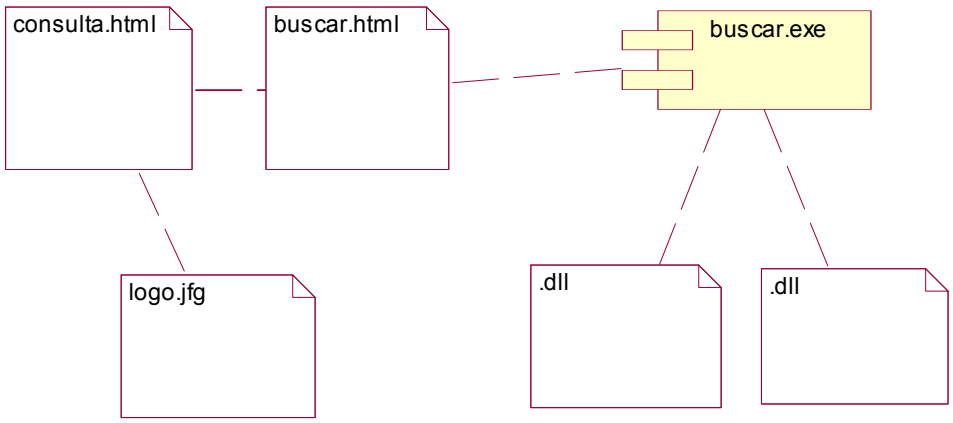


3. Diagrama de Componentes

Un diagrama de componentes muestra un conjunto de componentes y sus relaciones (es similar al concepto de diagrama de clases).
Un componente es una parte física reemplazable que empaqueta su implementación y es conforme a un conjunto de interfaces a las que proporciona su realización. A diferencia de los paquetes que se dice que es algo puramente conceptual (o sea que sólo existen en tiempo de desarrollo), los componentes son cosas físicas y existen en tiempo de ejecución.
Un diagrama de componentes contiene comúnmente:

- Clases
- Interfaces
- Realizaciones de asociación, generalización, realización y/o dependencia.

Ejemplo: Diagrama de Componentes

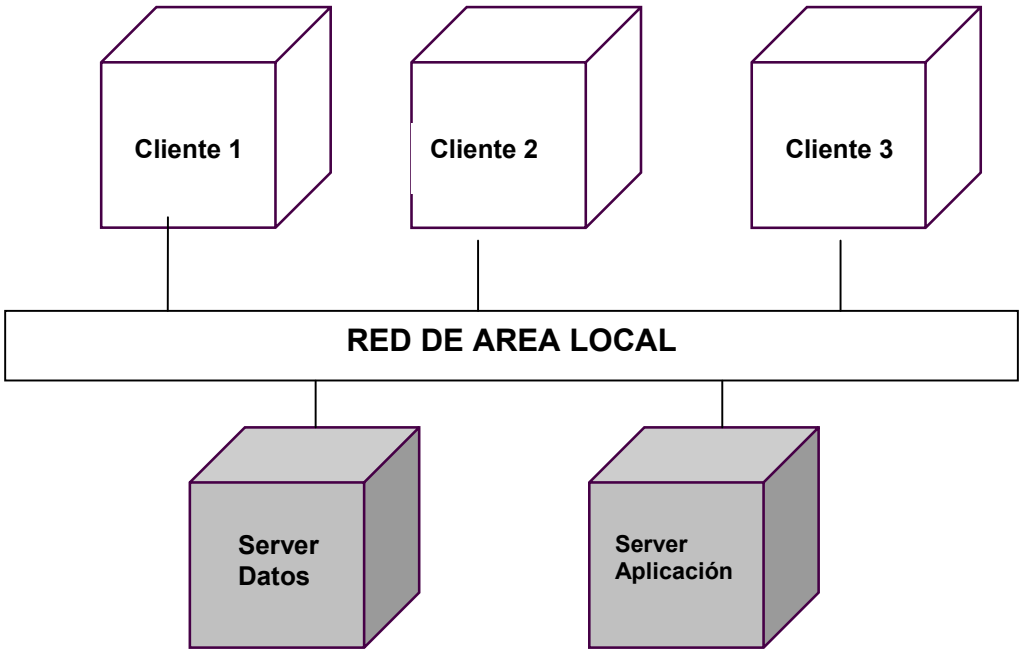


4. Diagrama de Despliegue

Un diagrama de despliegue muestra un conjunto de nodos y sus relaciones. Un nodo es un elemento del diagrama de UML que sirve para representar ***"cosas físicas".*** Representa, por ejemplo, la arquitectura del hardware sobre la que se monta la arquitectura del software.
Un Diagrama de Despliegue contiene comúnmente:

- Nodos
- Relaciones de asociación y/o dependencia

Ejemplo: Diagrama de Despliegue



Síntesis de los Diagramas Estructurales

TIPO DIAGRAMA	DE CLASES	DE OBJETOS	DE COMPONENTES	DE DESPLIEGUE
VISTA	Describe la esencia del negocio, a través de las clases	Describe la vista de diseño estática de un sistema	Describe la vista de implementación estática de un sistema	Describe la vista de despliegue estática de una arquitectura
COMPONENTES	• Clases • Interfaces • Colaboraciones • Relaciones	• Objetos • Conexiones	• Clases • Relaciones • Interfaces	• Nodos • Relaciones

Diagramas De Comportamiento

Los diagramas de comportamiento de UML se organizan en líneas generales alrededor de las formas principales en que se puede modelar la dinámica de un sistema:

1. Diagrama de Use Case
2. Diagrama de Secuencia
3. Diagrama de Colaboración
4. Diagrama de Transición de Estado
5 .Diagrama de Actividad

1. Diagrama de Use Case

Pone en relieve el comportamiento y la funcionalidad del sistema. El Proceso Unificado de Desarrollo es ***"conducido por Uses Cases"*** o sea que a partir de cada Use Case se van desarrollando los otros modelos.
Un diagrama de Use Cases muestra un conjunto de use cases, actores y sus relaciones. Es un objeto que describe todas las tareas que se realizan alrededor de un proceso, de manera que representa un conjunto de actividades de un Proceso de Negocios o de un Proceso del Sistema de Información automatizado (Proceso de Computadora.)
Un diagrama de Use Cases contiene comúnmente:

- Use Cases
- Actores: son quienes "instancian" los Use Case. Es "alguien" o "algo" con comportamiento, como una persona identificada por un rol, un sistema informatizado, o una organización.
- Relaciones: de asociación, generalización y/o dependencia (son las relaciones que "soporta" UML, más las relaciones estereotipadas: exclusión e inclusión).

Casos de Uso de Inclusión: siempre son llamadas desde otro Use Case, es decir, dentro de un Use Case (abstracto), se ejecutan obligatoriamente. En una plantilla de Casos de Uso a "Trazo Fino", se visualizan en la descripción del curso normal del mismo.

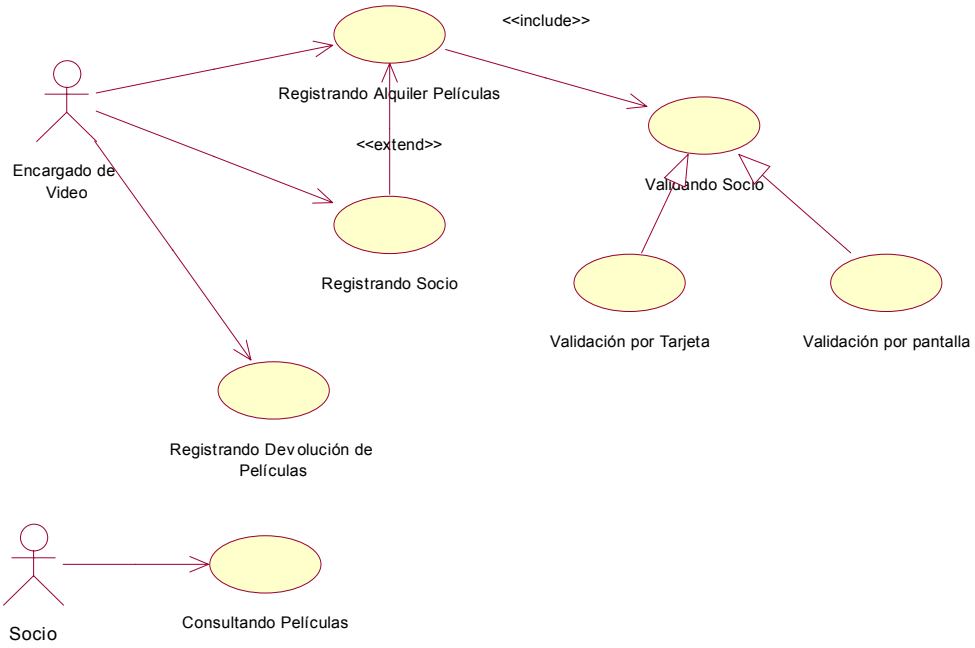
Casos de Uso de Exclusión: Son relaciones a veces llamadas desde un Use Case y a veces son la extensión de otro Use Case. Es decir, pueden ser casos de usos abstractos o concretos. En una plantilla de Casos de Uso a "Trazo Fino", se visualizan en la descripción del curso alternativo del mismo. A

diferencia de las relaciones de inclusión, los casos de uso de exclusión se ejecutan alternativamente.

De acuerdo a como puede ser "instanciado" un Use Case se considera:

- ***Concreto:*** si es instanciado por un actor
- ***Abstracto:*** es instanciado desde otro Use Case

Ejemplo: Diagrama de Use Cases



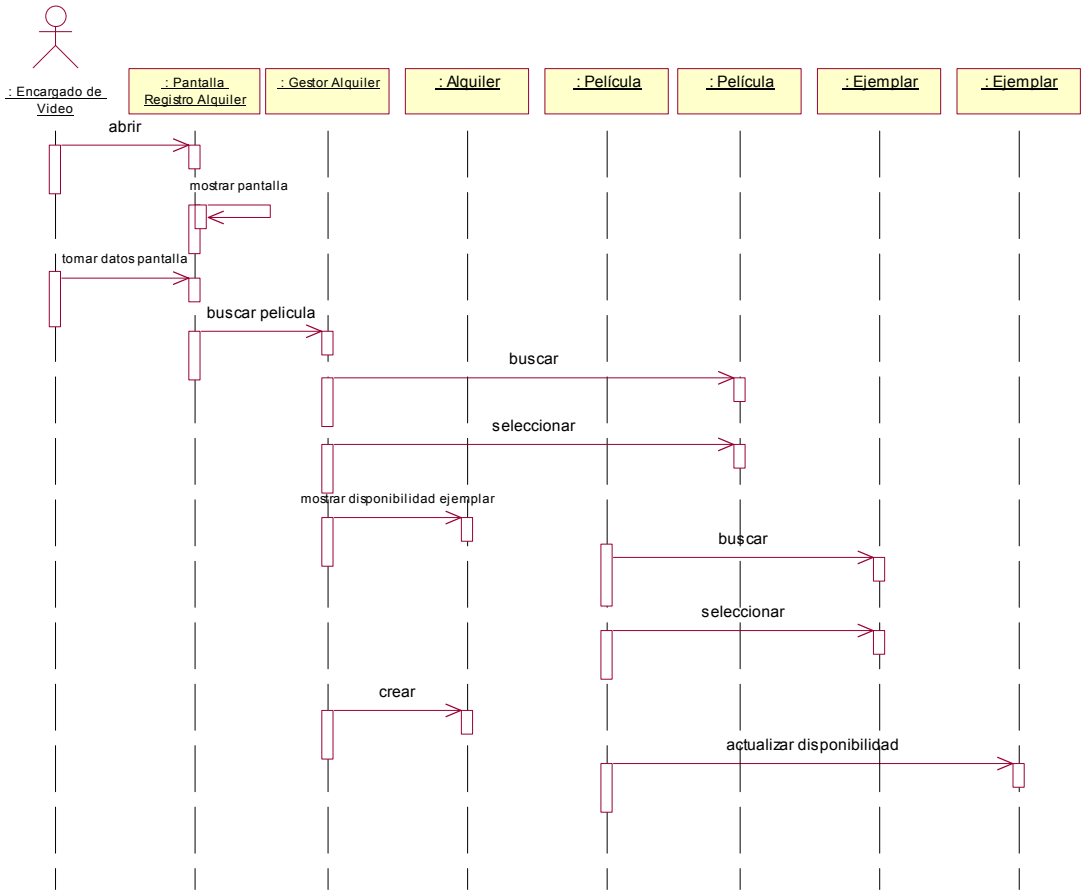
2. **Diagrama de Secuencia**

Explica la secuencia en la que los mensajes se intercambian entre los objetos a través del tiempo. Muestra un conjunto de objetos y los mensajes enviados y recibidos por esos objetos. Podríamos decir que son ***"diagramas derivables"***, esto significa que a partir de un diagrama de secuencia puedo "derivar" o "construir" un diagrama de colaboración o viceversa. De esta manera un diagrama de secuencia representa otra forma de "visualizar" la forma en la que colaboran entre sí los objetos.

Un Diagrama de Secuencia contiene comúnmente:

- Objetos
- Links
- Mensajes

Ejemplo: Diagrama de Secuencia



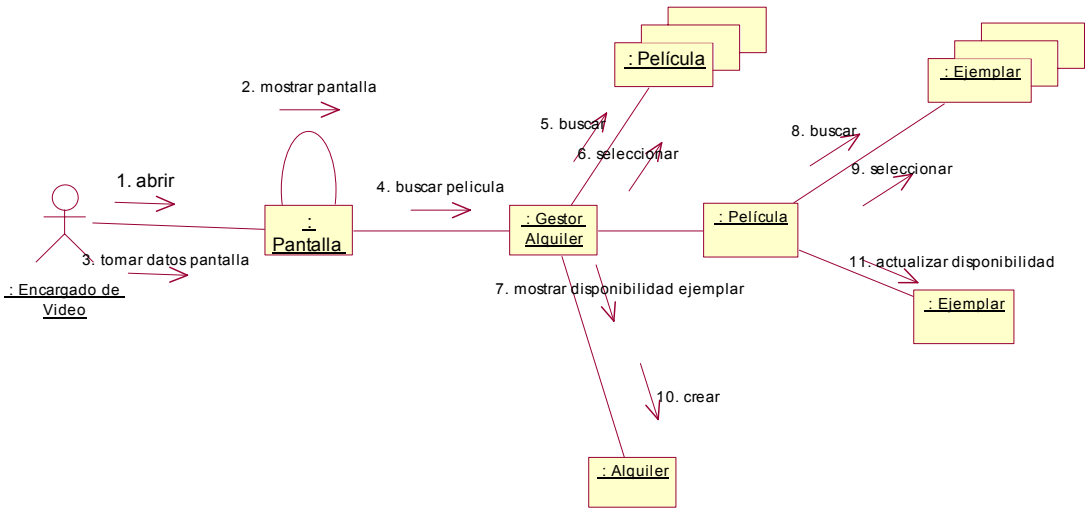
3. **Diagrama de Colaboración**

La colaboración entre objetos permite que estos intercambien mensajes poniendo en evidencia un determinado comportamiento como respuesta a dichos mensajes, que es la definición que hemos hecho de lo que es un programa desde la óptica del paradigma de objetos.
Anteriormente habíamos definido a la colaboración como ***"un conjunto de objetos que se mandan mensajes entre sí"*** y de esta forma realizan todas las

tareas descriptas en el Use Case, es decir, a través de los mensajes efectivizan las actividades del mismo.
Esto es así ya que un USE CASE es netamente descriptivo, por lo tanto por cada USE CASE siempre esta presente una colaboración. El conjunto de tareas descriptas en todos los USE CASE que componen un sistema representa toda la funcionalidad del sistema.
Un diagrama de Colaboración, es entonces, un Diagrama de Secuencia que enfatiza la organización estructural de los objetos. Muestra un conjunto de objetos y conexiones entre esos objetos y los mensajes enviados y recibidos por esos objetos. Contiene comúnmente:

- Objetos
- Links
- Mensajes

Ejemplo: Diagrama de Colaboración



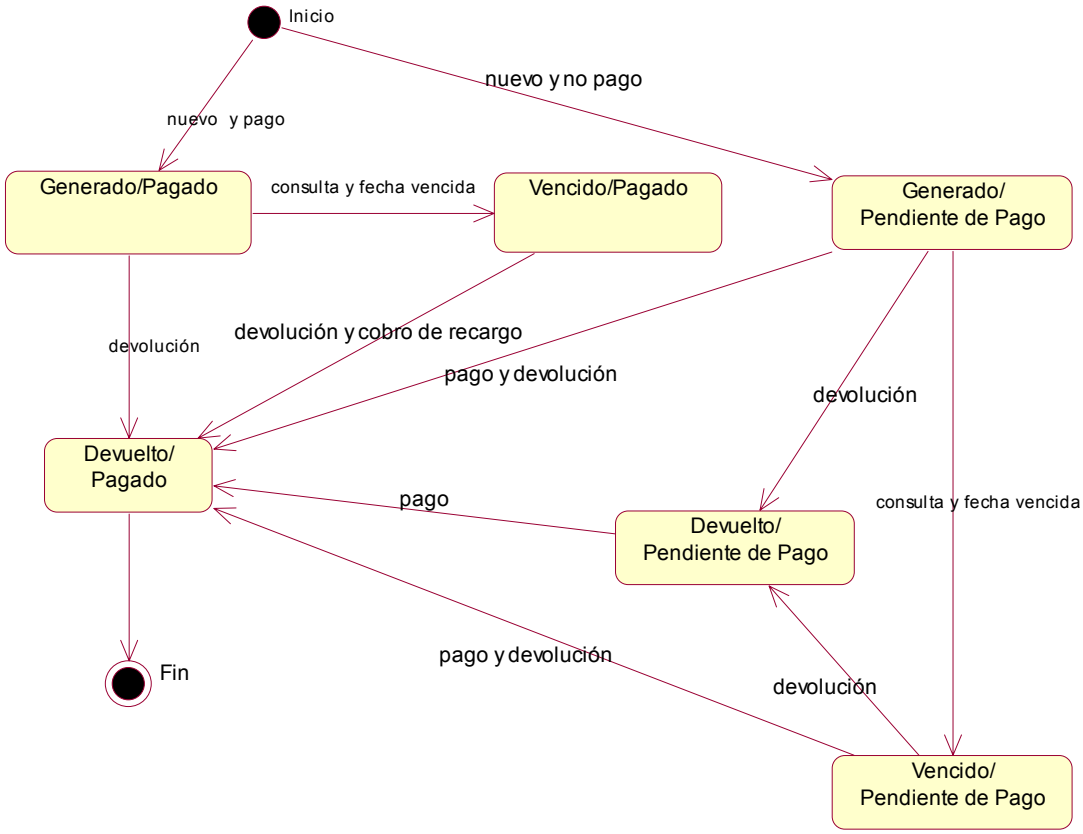
4. Diagrama de Transición de Estados

Un diagrama de Transición de Estado muestra una ***"máquina de estado",*** compuesta por estados, transiciones, eventos y actividades. Una máquina de estados es un comportamiento que específica las secuencias de estado por las que pasa un objeto o una interacción durante su vida en respuesta a eventos, junto son sus reacciones a esos eventos.

Un Diagrama de Transición de Estados contiene comúnmente:

- Estados
- Transiciones
- Eventos

Ejemplo: Diagrama de Transición de Estados



5. **Diagrama de Actividad**

Similar al concepto de colaboración. No visto en clases

Síntesis de los Diagramas de Comportamiento

TIPO DIAGRAMA	USE CASE	COLABORACIÓN	SECUENCIA	TRANSICIÓN DE ESTADO
CENTRADO EN	Organiza los componentes del sistema	Centrado en la organización estructural de los objetos que envían y reciben mensajes	Centrado en la organización temporal de los mensajes	Centrado en el estado cambiante de los objetos de un sistema dirigido por eventos
COMPONENTES	• Use Case • Relaciones • Actores	• Objetos • Relaciones • Mensajes	• Objetos • Relaciones • Mensajes	• Transiciones • Eventos • Estados

Reingeniería de Procesos

Los procesos, y no las organizaciones, son el objeto de la Reingeniería. Las compañías no rediseñan sus áreas o departamentos de ventas o manufactura; sino que rediseñan el trabajo conjunto que realizan las personas empleadas en esas dependencias.

La ***Reingeniería*** es la revisión fundamental, y el rediseño radical de ***procesos*** para alcanzar resultados espectaculares en medidas críticas y contemporáneas de costo, calidad, servicio y rapidez.

Palabras Claves

1. ***Fundamental:*** ataca al fondo del problema. La Reingeniería implica no tener ningún preconcepto. Es preguntarse ¿Porqué estamos haciendo lo que estamos haciendo? ¿Por qué lo hacemos de esta forma? Hay que pensar en **QUE SE DEBE HACER**. Por ejemplo: ¿Cómo podemos hacer en forma más eficiente la Investigación de Crédito? Significa que HAY que hacer la Investigación de Crédito.

2. ***Radical:*** no busca mejorar los procedimientos individuales existentes en una organización, sino que produce cambios profundos, desde la raíz. No busca mejoras incrementales en el negocio sino que reinventa el mismo. Significa volver a empezar, arrancar de cero.

3. ***Espectacular:*** se aplica para lograr algo grande o no se aplica.

4. ***Procesos:*** es el concepto más importante de esta definición. Siempre debo pensar en procesos de negocio, no en tareas, ni áreas, ni oficios.

¿Qué son los Procesos?

Un proceso es un conjunto de tareas que “agregan valor”, que transforman entradas (insumos) en salidas (productos o servicios).

En ASI (Análisis de Sistemas) cuando estudiamos los Procesos en los Diagramas de Flujo de Datos decíamos que un Proceso ó "burbuja" transforma flujos de datos de entrada en flujos de datos de salida.

Proceso de Fabricación ó de Manufactura es un conjunto de tareas ó actividades que se desarrollan cuando ingresan insumos ó partes y esas tareas "transforman" esos insumos en un producto terminado.

Proceso de Venta es un conjunto de tareas ó de actividades que se desarrollan cuando un cliente pide un producto y el negocio de ventas (ejemplo un hipermercado) se lo entrega, aquí no se ve claramente la "transformación" de cosas de entrada en productos de salida pero SI existe un "valor agregado" que es el poner disponible (en estanterías, mostradores, cerca de la casa de cada cliente, con estacionamiento, etc., etc.) el producto para el cliente.

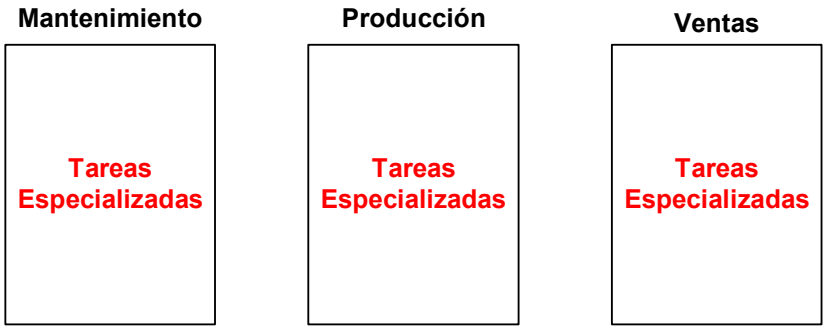
Pero Proceso de Negocio es, en general, un conjunto de tareas ó actividades que "agregan valor" a insumos y dan por resultado un producto un servicio ó un producto para un cliente. El pensar que es un cliente le da mucho sentido a la comprensión del término "proceso de negocio" porque el objetivo del negocio es producir renta a partir de "vender" algo (productos ó servicios) a clientes del negocio.
Luego "ampliamos" ese concepto y decimos por ejemplo que un Proceso de Negocio de Manufactura comienza con el aprovisionamiento y termina en el despacho de fábrica, y el Proceso de Negocio de Desarrollo de Producto comienza con un concepto de producto y termina con el prototipo del mismo.
En una empresa los procesos se corresponden a actividades naturales de los negocios. Existen diferentes "tipos" de procesos dentro de una organización: de negocio, de soporte, de administración. Vamos a ocuparnos solo de los Procesos de Negocio.
Los Procesos de Negocios son aquellos que son centrales y esenciales para la organización u empresa (venta de servicios, venta de productos, etc.). Son los procesos que se describen en los Casos de Uso (sucesión de eventos a partir de la acción de un actor).
En tiempos pasados las empresas dividían sus procesos en "tareas especializadas". Esto traía aparejado una gran tarea de coordinación y, consecuentemente, de un gran número de supervisores o responsables. Cada

supervisor o responsable estaba encargado de un departamento o de una unidad de trabajo, pero nadie tenía asignada la responsabilidad de realizar y controlar todo el ***proceso*** que involucra un conjunto de tareas de varias áreas de la organización. Esta forma de trabajo continúa en muchísimas organizaciones y se ve reflejada en la forma en que están "organizadas" como lo muestra cualquier organigrama que divide a la organización en compartimentos de "especialización de tareas", entonces el Departamento Ventas solamente se dedica a vender, el de Producción solamente a fabricar, el de Finanzas solamente a hacer tareas que tienen que ver con los recursos financieros de la organización y así sucesivamente.

A continuación trataremos de explicar los motivos por los cuales estas formas organizacionales: grupos funcionales, áreas, departamentos, dejaron de tener importancia para esta visión de Procesos que actualmente se tiene en las organizaciones y sobre la cual hace énfasis la reingeniería.

Teoría de Sistemas vs. Teorías Mecanicistas

Antes de implantarse la Teoría de Sistemas, las organizaciones implementaban para su correcto funcionamiento las "*Teorías Mecanicistas*".
Las Teorías Mecanicistas "concebían" a la organización dividida en áreas o departamentos, cada una de las cuales realizaba un conjunto de "tareas especializadas".
Por ejemplo, el área "Organización y Métodos", agrupaba a un conjunto de empleados dedicados a escribir procedimientos y métodos en busca del funcionamiento óptimo de la organización.



Cada área de la organización realizaba un conjunto de "tareas especializadas"

Las Teorías Mecanicistas intentaban solucionar el "problema organización" como si una organización fuese la aplicación simple del principio de causa/efecto: una causa siempre provoca el mismo efecto y un efecto puede ser producido por más de una causa. Aplicando este principio las "tareas especializadas", producían

resultados (efectos), de acuerdo a cada una de las entradas del proceso (causas). El concepto era como buscar para el "problema organización" una simple receta de cocina: dados los ingredientes (entradas), sigo paso a paso las instrucciones de la receta (cómo hacer cada una de las tareas) y elaboro una comida (resultado). Por eso Organización y Métodos trabajaba en hacer "procedimientos" que eficientizaren cada una de las tareas en cada una de las áreas de la organización.
Sin embargo la aplicación de estas teorías no lograba el objetivo perseguido, ya que al concentrarse demasiado en el funcionamiento interno de la organización perdía todo contacto con el medio ambiente. Es decir, la mayoría de las tareas eran ejecutadas para satisfacer exigencias internas de la propia organización y no para satisfacer las necesidades de los clientes.
Otra característica del enfoque mecanicista era su "visión estática" de la organización, a través de una estructura de jerarquías u organigrama, donde cada cargo de dicho organigrama representaba un empleado realizando una tarea especializada.

La Teoría de Sistemas vino a dar algunas ideas más interesantes para solucionar el ***"problema organización":***

- ✓ El reconocer que las organizaciones son cosas complejas y que el simple enunciado de una "receta" (por otra parte nunca completamente encontrada) no produciría los resultados esperados.

- ✓ El reconocer la importancia del "medio ambiente" como algo altamente complejo y fuera de control de la organización. En ese medio ambiente se encuentra el cliente sin el cual no se puede hacer negocios, sin el cual no se puede ganar dinero.

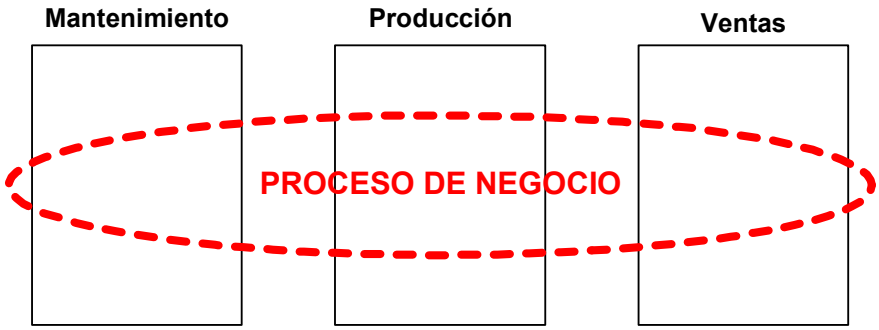
- ✓ El reconocer que la organización es un "sistema probabilístico" y no "mecánico" por lo tanto no es de aplicación el principio de causa/efecto en forma simple.

- ✓ Lo probabilístico de una organización es porque se toman decisiones por parte de personas humanas en el interior de la organización y en el medio ambiente por parte de clientes que también son seres humanos.

- ✓ El enunciar, sin embargo, que la sinergia (solamente esperable en sistemas no mecanicistas) es algo que puede producir resultados mucho mayores que la suma de las partes producidas por cada tarea "especializada" en las que se ha divido el trabajo en las organizaciones.

La Reingeniería, como una expresión práctica de la Teoría de Sistemas, puso en evidencia las falencias del enfoque mecanicista, y propuso la reunificación de las "tareas especializadas" en procesos coherentes denominados ***"Procesos de Negocios".*** Los "Procesos de Negocios", no ocurren en una oficina ó área de la organización en particular, sino que atraviesan varias áreas o departamentos, de

esta forma los pasos del proceso se realizan siguiendo un orden natural, evitando el seguimiento de líneas rígidas e inalterables y el desplazamiento por fronteras organizacionales. La Reingeniería reemplazó los responsables de área o departamento por "Gerentes de Caso", quienes se hacen cargo, dentro de la organización, de la realización de los procesos, del contacto con el cliente, con el mundo exterior.



"No se reingenieriza un área, sino que se hace reingeniería sobre un proceso"

Nota

El alumno no debe interpretar en el gráfico anterior que el Proceso de Negocio es un Use Case sino que el gráfico intenta explicitar que un proceso de negocio "atraviesa" varios Departamentos ó áreas de la organización.

Siempre se "completa" la intelectualización de un concepto con ejemplos prácticos. El hecho de que la reingeniería al concentrarse en Procesos soluciona las cosas y que los Procesos son un conjunto de Tareas y que la concepción "mecanicista" hacía énfasis en las Tareas es difícil, en algunos casos, de conceptualizarse debidamente.
El ejemplo de que el Departamento de Mantenimiento de Aviones de una compañía aérea que está en Córdoba, recibe un pedido de reparación de un avión que en ese momento está haciendo escala en Rosario y el Jefe de Mantenimiento decide enviarlo al día siguiente para no tener que pagar el hotel al técnico que va a hacer la reparación es un buen ejemplo. Es un buen ejemplo porque la organización jerárquica dividida en departamentos mide la eficiencia en esos términos, por departamento, por especialización. Entonces el Departamento Mantenimiento tendrá una buena eficiencia porque tendrá pocos gastos (el Jefe con su decisión "ahorró" costos de hotel para el técnico).
Pero nadie piensa en los pasajeros (clientes) que tienen que pasarse una noche en Rosario esperando que reparen el avión que podría haberse reparado en el mismo día pero no se hizo porque el Jefe de Mantenimiento vió que en ese día su

técnico no tendría vuelo de vuelta a Córdoba para su técnico lo que lo obligaría a pasar una noche en Rosario con el consiguiente costo de hotel.

Se ve claramente como el Departamento Mantenimiento ahorra costos, tiene un buen rendimiento, pero la organización es mal vista por los clientes que con alguna probabilidad se cambiarán a otra organización o sea que la organización como un todo pierde a pesar de que cada departamento, en que jerárquicamente (y con sentido de "especialización" de tareas) se dividió a la organización, pueda tener buen rendimiento. Es como que la organización organizada jerárquicamente como lo dictan los organigramas, por especialización de tareas, no pensara en los clientes que son completamente necesarios para el negocio, para cumplir con su objetivo. Por eso es que la Reingeniería propone la figura del "Gerente de Caso" para que alguien en la organización sea "responsable" de un Proceso de Negocio, para que alguien "hable" en nombre del cliente, para que alguien piense en el cliente.
Este pensar en el Proceso hoy se pone en evidencia cuando en las entidades bancarias, por ejemplo, existen "Ejecutivos de Cuenta" (una especie de "Gerente de Caso" de la Reingeniería) que atienden a un cliente del banco como tal, como cliente del banco. Antiguamente a la aparición de estos conceptos cada cliente era "cliente de cuentas corrientes" ó "cliente de caja de ahorros", etc., y para que lo "atiendan" en cada negocio tenía que hacer una cola, una cola para ser atendido en cuentas corrientes, otra cola para ser atendido en caja de ahorro, etc. Dicho de otro modo el cliente era cliente de cuentas corrientes ó de caja de ahorro pero no cliente del banco como lo es ahora cuando se pone en funciones a un "Ejecutivo de Cuenta" que atienden a un cliente para todas las cuentas que tuviere con el banco.

Objetivos de la Reingeniería

Uno de los objetivos perseguido por la reingeniería es intentar ***"modelizar el Proceso de Negocios"*** desde el punto de vista del cliente. Es por eso que el concepto de "Proceso" es central en la definición de lo ***que es*** la Reingeniería. La Reingeniería no se aplica a toda la empresa en su conjunto o a cada departamento en particular sino a un determinado proceso de negocio.
Sintetizando, la reingeniería persigue:

1. Hacer más horizontal a la estructura organizacional
2. Modelizar el Proceso de Negocios
3. Modificar la forma de trabajar para orientarse al cliente. La aplicación de la reingeniería a los procesos de una empresa debe realizarse a través de los ojos de los clientes.

La reingeniería ofrece grandes ganancias, incluyendo reducciones de costos de producción y mejoras a gran escala de calidad y servicio al cliente. Aun cuando

esto suene tentador para cualquier organización no cualquier empresa está en condiciones de realizar proyectos de reingeniería.

¿A qué empresas reingenierizar?

- ✓ A las que se encuentran en una situación realmente crítica (casi en quiebra).
- ✓ A las que, por el contrario, están en excelentes condiciones pero buscan distanciarse significativamente de la competencia.
- ✓ A las que avizorizan que pueden venir situaciones críticas.

¿A qué empresas no reingenierizar?

- ✓ A las que buscan aumentar sus ventas un 10% o reducir sus costos un 5%. En líneas generales todas aquellas organizaciones que no persigan cambios espectaculares.

Puntos a tener en cuenta en la Reingeniería

Entre un 50% y un 70% de los proyectos de Reingeniería fracasan. Entre las causas del fracaso pueden citarse que no se han tenido en cuenta las siguientes pautas:

1. Considerar que hay que empezar de cero
2. No conformarse con mejoras marginales
3. Distinguir entre transformaciones y mejoras
4. Tener en cuenta las diferencias entre procesos y áreas

¿Qué no es reingeniería?

1. No es hacer con computadoras lo que antes se hacía manualmente. No es solamente hacer más rápidas las tareas con la ayuda de una computadora. La Reingeniería va más allá de la "informatización" o "transformación" en procesos de computadora de todos los procedimientos manuales de una organización. La Reingeniería piensa desde el punto de vista de los "Procesos de Negocios" y su optimización y no solo de la mejora del software.

2. No es Reingeniería de software. La Reingeniería del software estudia el sistema que se aplica en una organización y realiza la documentación del mismo, como si dicho sistema se hubiese realizado con alguna metodología en particular. La Reingeniería de software produce los planos de los datos, la confección de todos los modelos que correspondan (de información, eventos, de contexto si es estructurada la metodología que se aplica ó los modelos de use case, modelo de objetos del dominio del problema, etc., si se usará la metodología orientada a objetos), aplicando una determinada metodología. No desarrolla el sistema, solo lo documenta.
3. No es sólo eliminar burocracia. Burocracia es la excesiva interacción entre los componentes internos de una organización en desmedro de la relación entre la organización y los componentes del medio ambiente.
4. No es reestructurar (hacer menos con menos). La Reingeniería persigue hacer más con menos. Aun cuando con frecuencia los proyectos de Reingeniería implican reducciones de personal, no es un achicamiento, ni implica despidos u otras medidas aisladas.

El Pensamiento Inductivo

En la Reingeniería no hay solución si no se tiene como herramienta a la Tecnología y principalmente a la TI (Tecnología Informática). Esta premisa posibilita el pensamiento inductivo que es aquel en donde se piensa primero en la solución y luego en los problemas que esa solución puede solucionar, al contrario del pensamiento deductivo normal, en donde, ante un problema analizo las diferentes soluciones que puede tener.

Conclusión

Finalmente puede decirse que ***"No hay reingeniería sin tecnología informática"*** y es la tecnología informática la que provee a los especialistas en sistemas de todas las herramientas y soluciones para que estos sean capaces de brindar soluciones a las empresas u organizaciones (pensamiento inductivo de la reingeniería).

Metodologías Orientadas a Objetos

Proceso Unificado de Desarrollo (PUD): es el conjunto de actividades necesarias para transformar los requisitos del usuario en un sistema software.

Es un proceso que:

- Proporciona una guía para ordenar las actividades de un equipo
- Dirige las tareas de cada desarrollador por separado y del equipo como un todo
- Especifica los artefactos que deben producirse
- Ofrece criterios para el control y la medición de los productos y actividades del proyecto

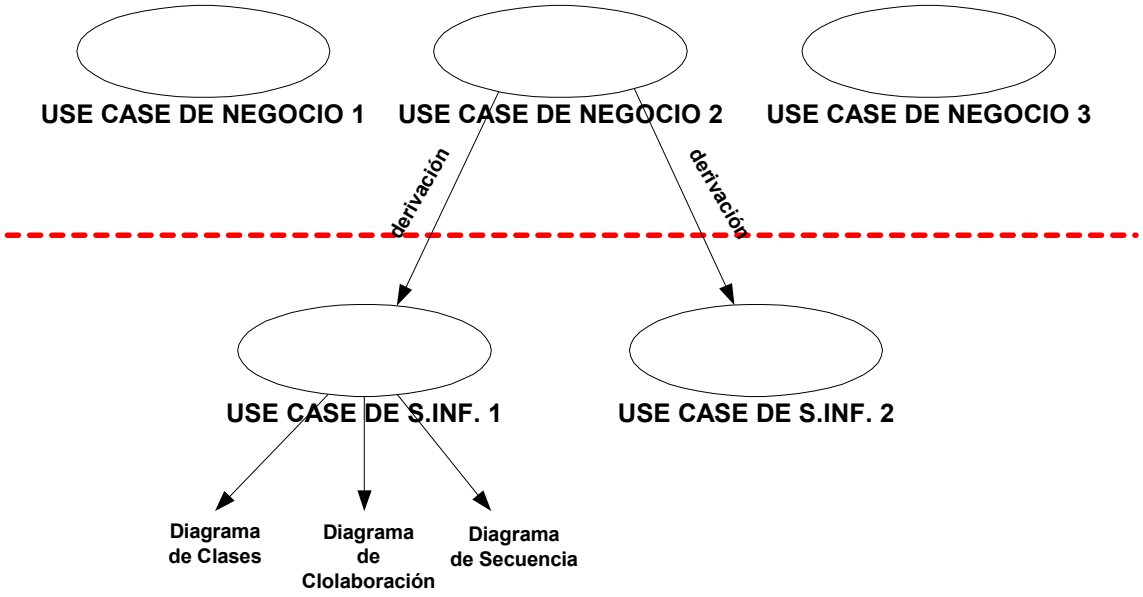
PUD- Características Esenciales

1. ***Dirigido por Casos de Uso***
2. ***Centrado en la Arquitectura***
3. ***Iterativo e Incremental***

1. Dirigido por Casos de Uso

Los casos de uso rescatan la funcionalidad del sistema o lo que tiene que realizar el mismo. Esa es la característica más importante. Sin bien existen requerimientos no funcionales (performance, seguridad, etc.), los uses cases que estudiamos se centran en la descripción de los requerimientos funcionales. Que el PUD sea dirigido por casos de uso significa que por cada caso de uso del proceso de negocio voy a poder derivar todos los otros modelos: de colaboración, de secuencia, de clases, de estado, etc. Esto último provee de ***"rastreabilidad",*** que es la base para que los cambios (que indudablemente van

a producirse) puedan realizarse con facilidad, entonces cada vez que el sistema deba modificarse puedo tener control de todo el ambiente de cambio.



El gráfico anterior muestra los Use Case que describen Procesos de Negocio por encima de la línea de puntos y los Use Case que describen Procesos del Sistema de Información que se derivan de aquellos están por debajo de la línea de puntos. A su vez se trata de explicitar que a partir de cada Use Case del Sistema de Información se producen los Diagramas correspondientes (Diagrama de Colaboración, Diagrama de Secuencia, etc.), lográndose de esta manera la "rastreabilidad" buscada.

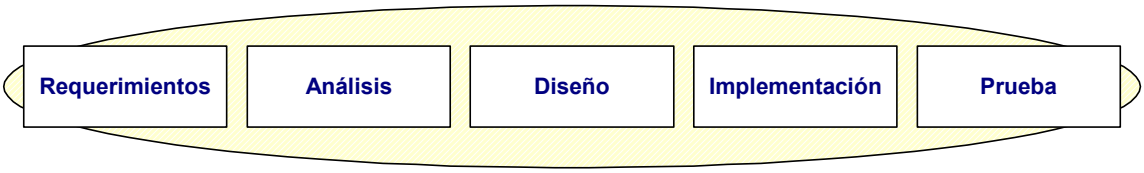
Los Casos de Uso:

- Capturan el valor de los requerimientos, o sea los requisitos funcionales, centrándose en el valor añadido por el usuario.
- Conducen el Proceso de Desarrollo. El Proceso de Desarrollo sigue un hilo, avanza a través de una serie de flujos de trabajo que parten de los casos de uso y que le facilitan a los jefes de proyecto la planificación y el control de las tareas involucradas en el desarrollo.
- Delinean la arquitectura: con una selección correcta de casos de uso. Se desarrollan junto con la arquitectura, la guían y esta influye en la selección de casos de uso.

Los Casos de Uso son:

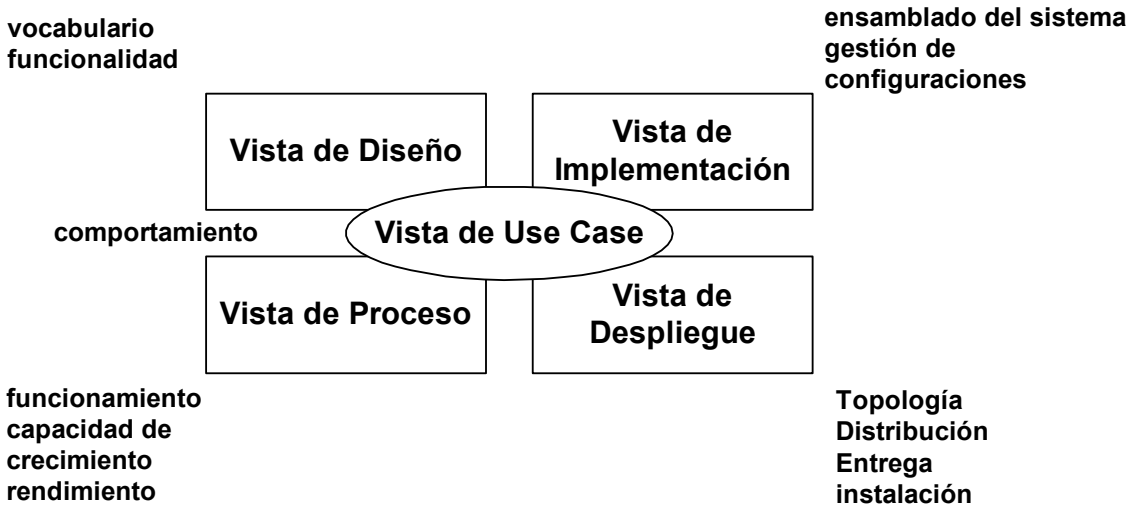
- Especificados en el Modelo de Análisis

- Realizados por el Modelo de Diseño
- Distribuidos por el Modelo de Despliegue
- Implementados por el Modelo de Implementación
- Verificados por el Modelo de Prueba



2. Centrado en la Arquitectura

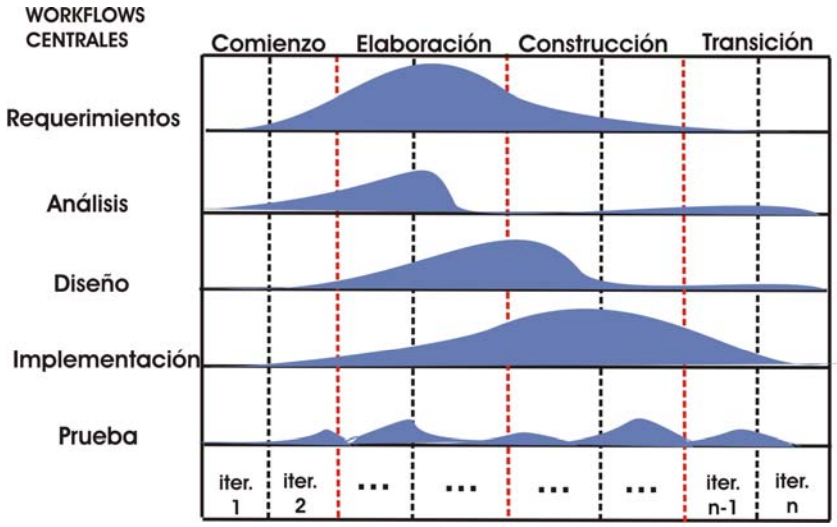
Significa que la arquitectura del sistema se utiliza como un artefacto básico para conceptuar, construir, gestionar, y hacer evolucionar el sistema en desarrollo. La arquitectura es el arte de combinar los elementos, de organizar los componentes. En el caso del PUD, cada una de las formas de organizar los componentes me permite ver a los Casos de Uso con una visión diferente. Cada una de esas visiones se relaciona entre sí y se denominan "Vistas"



3. Iterativo e Incremental

Se dice que es iterativo porque involucra el desarrollo de una serie de pasos en el flujo de trabajo. Se dice que es incremental porque en cada interacción se va integrando la arquitectura del sistema, a su vez en cada iteración, se van incorporando mejoras incrementales sobre el producto.

El PUD puede descomponerse en fases. Cada fase representa un intervalo de tiempo entre dos hitos importantes de un proceso. En el ciclo de vida de desarrollo de software existen cuatro fases:



- Inicio: se analizan las principales funciones para los usuarios más importantes, se identifican arquitecturas candidatas, se hace un plan del proyecto y una estimación del costo.

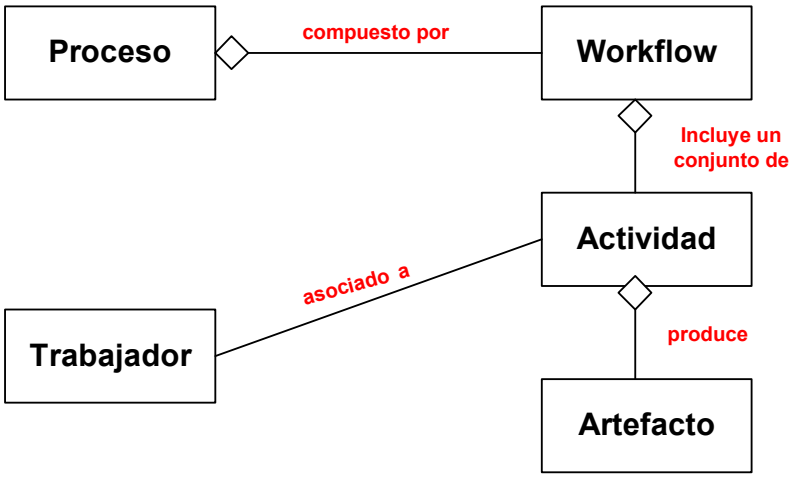
- Elaboración: se especifican en detalle la mayoría de los casos de uso del producto y se diseña la arquitectura del sistema. Podemos decir que la arquitectura es análoga al esqueleto cubierto por la piel pero con muy poco músculo (el software).

- Construcción: creación propiamente dicha el producto. Se especifican todos los casos de uso "faltantes". La arquitectura debe ser estable y garantizar la robustez del producto.

- Transición: actividades concentradas en el testing (ej. Debe producirse una versión beta), y actividades de formación del cliente, proporcionar ayuda en

línea de ayuda y asistencia. Planificación de la implementación del producto final y de las estrategias de mantenimiento para casos de fallas.

Conceptos importantes del PUD



Proceso: son todas las actividades que deben llevarse a cabo para transformar los requisitos de usuario en un producto (no es la ejecución de las actividades). Es una plantilla para crear proyectos, que son el elemento central organizativo del PUD. Como puede verse en el gráfico los procesos están compuestos por Workflows.

Workflow: es una realización (parte de un caso de uso de negocio). Cada workflow involucra un conjunto de actividades que incluyen: trabajadores participantes, actividades que ellos realizan y artefactos que ellos producen). Es una forma de describir procesos de trabajo de un negocio.
Un ejemplo de flujo de trabajo es el flujo de trabajo de los requisitos. En el intervienen los siguientes trabajadores: analista de sistemas, arquitecto, especificador de casos de uso y diseñador de interfaz de usuario y se producen los siguientes artefactos: modelo de casos de uso, casos de uso y otros.

Actividad: es una unidad de trabajo tangible, ejecutada por un trabajador en un workflow, por lo tanto siempre están asociadas a un trabajador. Las actividades producen como resultado artefactos.

Trabajador: es el rol o el papel que un individuo puede desempeñar en el desarrollo de software (arquitecto, desarrollador, diseñador, analista, ingeniero de prueba, etc.).

Artefactos: cualquier tipo de información creada, producida, cambiada o usada por los trabajadores en el desarrollo del sistema. Algunos ejemplos de artefactos son:

- Los diagramas de UML y su texto asociado
- Los bocetos de la interfaz de usuario
- Los prototipos

Modelos que se producen en el Proceso de Desarrollo de Software

1. ***Modelo de Negocio***
2. ***Modelo del Dominio***
3. ***Modelo de Casos de Uso***
4. ***Modelo de Análisis***
5. ***Modelo de Diseño***
6. ***Modelo de Proceso***
7. ***Modelo de Despliegue***
8. ***Modelo de Implementación***
9. ***Modelo de Prueba***

Como ya se ha definido el Proceso Unificado de Desarrollo es un marco de referencia que puede especializarse para gran variedad de sistemas de software, diferentes áreas de aplicación, diferentes tipos de organizaciones y diferentes tamaños de proyectos. Esto significa que el PUD se puede adaptar, modificar, especializar, recortar, para proyectos de mediano o corto alcance, por lo tanto en la mayoría de los casos solo se utilizarán algunos de los modelos que propone.

Tipos de Workflows

Un workflow es una realización de un caso de uso del negocio o parte de él (en nuestro caso en el Proceso Unificado para el desarrollo de software). Puede describirse en términos de diagramas de actividad que incluye: trabajadores, actividades que ellos realizan y artefactos que ellos producen. En cada uno pueden identificarse:

- actividades (que cosas se tienen que hacer),
- rol, trabajador (quien o quienes realizan la actividad),
- producto obtenido (artefacto ó artefactos que se producen)

Existen diferentes tipos de workflow, los siguientes son los “Fundamentales”:

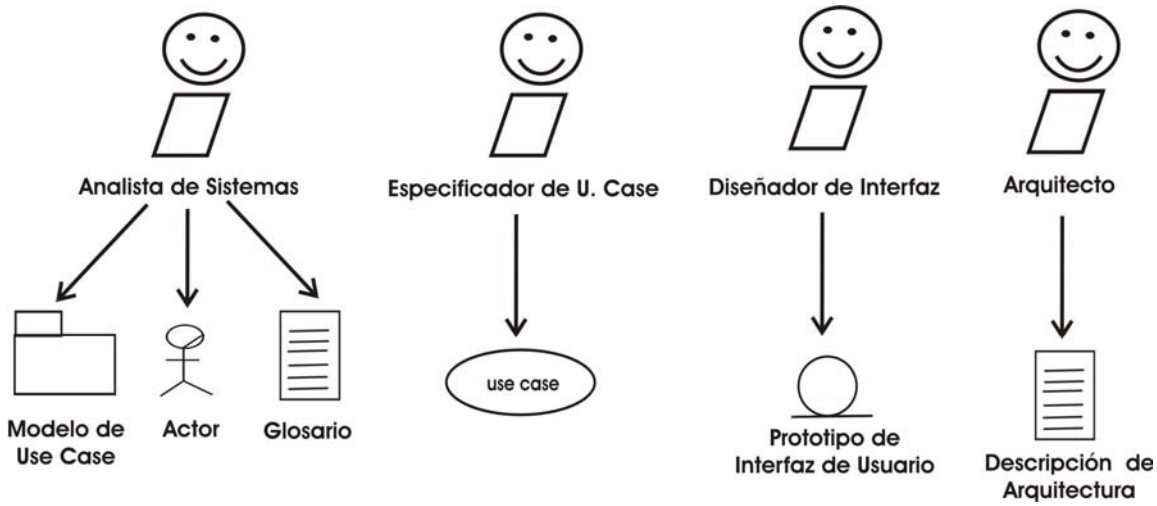
1. ***Workflow de Requerimientos***
2. ***Workflow de Análisis***
3. ***Workflow de Diseño***
4. ***Workflow de Implementación***
5. ***Workflow de Prueba***

1. Workflow de Captura de Requerimientos

El concepto de Requerimiento está relacionado con lo que debe hacer el sistema, con la funcionalidad del mismo. El propósito de la captura de requerimientos incluye:

- Hallar los requerimientos candidatos
- Comprender el contexto del sistema (2 aproximaciones: Modelo de Negocio o Modelo del Dominio)
- Capturar los requerimientos funcionales (produce el modelo de casos de uso)
- Capturar los requerimientos no funcionales (produce casos de uso de soporte: seguridad, permisos, accesibilidad, etc.)
- Cada una de estas actividades es llevada a cabo por un trabajador y produce un artefacto específico.

Trabajadores y Artefactos involucrados en la Captura de Requerimientos:



El rol de los requerimientos en el Ciclo de Vida

1. ***Fase de Comienzo:*** identificar la mayoría de los casos de uso para delimitar el sistema y el alcance del proyecto. Detallar los más críticos (menos del 10%).
2. ***Fase de Elaboración:*** capturar la mayoría de los requerimientos restantes para poder estimar el esfuerzo de desarrollo. En esta etapa se capturan el 80% de los requisitos y se describen la mayoría de los casos de uso.

3. ***Fase de Construcción:*** identificar e implementar los requerimientos restantes
4. ***Fase de Transición:*** casi no hay captura de requerimientos, al menos que se requieran cambios. Esto se debe a que los requerimientos han sido capturados en las etapas anteriores

Workflow de Requerimientos- Actividades- Responsables- Artefactos que se producen

Actividades:

1. ***Encontrar actores y Casos de usos***
   1.1. Encontrar actores
   1.2. Encontrar los U-C
   1.3. Describir brevemente cada U-C
   1.4. Describir el modelo de U-C en su totalidad y preparar Glosario
2. ***Priorizar los U-C***
3. ***Detallar cada uno de los U-C***
   3.1. Estructurar la descripción de los U-C
   3.2. Formalización de la descripción de los U-C
4. ***Prototipar la interfaz de usuario***
   4.1. Crear el diseño lógico de una interfaz de usuario
   4.2. Crear un diseño y un prototipo físico de la interfaz de usuario
5. ***Estructurar el modelo de U-C***
   5.1. Descripción de funcionalidad compartidas
   5.2. Descripción de funcionalidad adicional y opcional.

ACTIVIDAD	REALIZADO POR	RESULTADOS
PUNTO UNO	Analista de sistemas	• Modelo de U-C esbozado • Glosario • Actor
PUNTO DOS	Arquitecto	• Descripción de la Arquitectura • (vistas del modelo de U-C)
PUNTO TRES	Especificador de U-C	• Caso de Uso detallado
PUNTO CUATRO	Diseñador de interfaz de usuario	• Prototipo de interfaz de • de usuario
PUNTO CINCO	Analista de sistemas	• Modelo de U-C (estructurado)

2. Workflow de Análisis

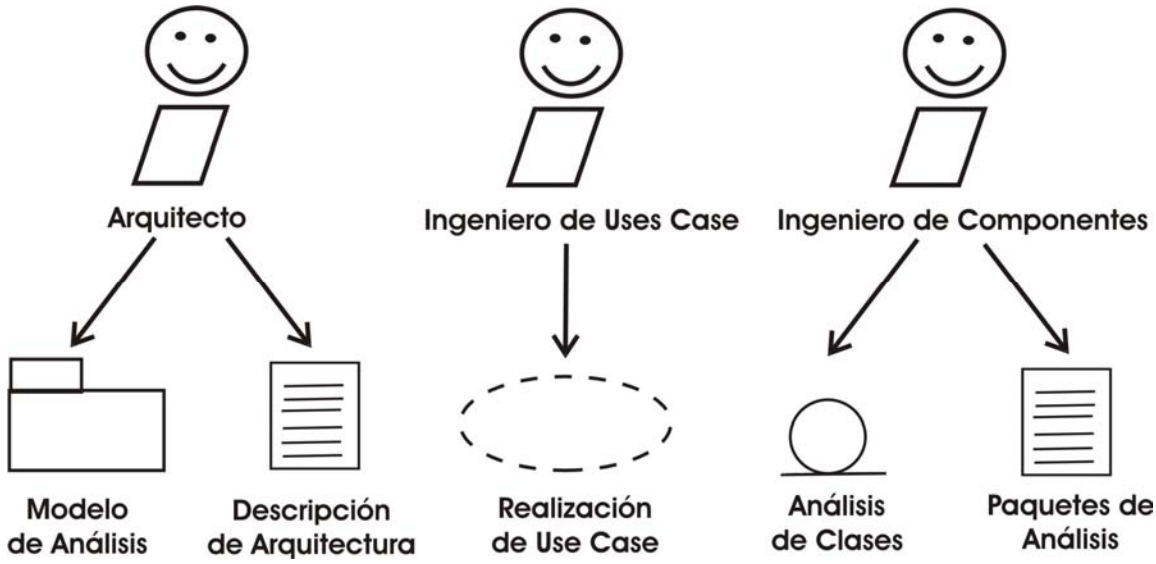
El Análisis es una tarea investigativa cuyos propósitos son entre otros:

1. Comprender el dominio del problema
2. Definir lo "que el sistema tiene que hacer" en el ámbito de la aplicación que el usuario haya definido.

La importancia de este workflow radica en que:

- Permite una especificación más precisa de los requerimientos, a la vez que estructura los mismos
- Se describe usando el lenguaje del desarrollador
- Puede verse como un primer corte del modelo de diseño

Trabajadores y Artefactos involucrados en el Análisis:



El rol del Análisis en el Ciclo de Vida

1. ***Fase de Comienzo:*** contribuye a la planificación de los incrementos, de cada miniproyecto.

2. ***Fase de elaboración y Construcción:*** contribuye a obtener una estructura robusta y estable y facilita una comprensión más profunda de los requerimientos

Workflow de Análisis- Actividades- Responsables- Artefactos que se producen

Actividades:

1. ***Análisis de la arquitectura:*** tiene por objeto esbozar el modelo de análisis y la arquitectura a través de la identificación de:
    1.1. ***Paquetes del análisis:*** que proporcionan un medio para organizar el modelo de análisis en piezas más pequeñas y más manejables.
    1.2. ***Clases de entidad obvias:*** propuesta preliminar de las entidades más importantes (de 10 a 20) basadas en las entidades del negocio que surgieron en la captura de requisitos.
    1.3. ***Requisitos especiales comunes:*** estos aparecen durante el análisis y es importante anotarlos para que sean tratados en las etapas de diseño e implementación, como ser:
    Persistencia, distribución y concurrencia, gestión de transacciones, etc.
2. ***Analizar un Caso de Uso:*** para identificar las clases del análisis que ejecutan el U-C, para distribuir el comportamiento de los objetos y para capturar requisitos especiales del U-C.
    2.1. ***Identificación de clases del análisis:*** se identifican las clases de control, entidad e interfaz para ejecutar el U-C. Poner los nombres, responsabilidades, atributos y relaciones.
    2.2. ***Descripción de interacciones entre los objetos del análisis:*** mediante el uso de diagramas de colaboración.
3. ***Analizar una clase:*** encontrar las responsabilidades de acuerdo a su papel en cada caso de uso, también encontrar los atributos, asociaciones, agregaciones y generalizaciones.
4. ***Analizar un paquete:*** para que los paquetes sean tan independientes como sea posible, para describir las dependencias y así notar los efectos de un cambio futuro.

Workflow de Análisis

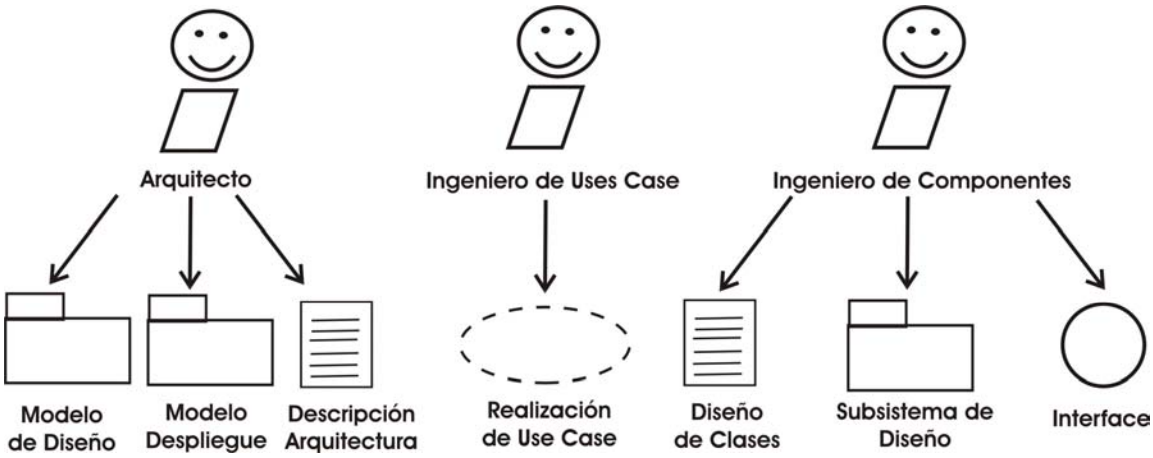
ACTIVIDAD	REALIZADO POR	RESULTADOS
PUNTO UNO: Análisis de la arquitectura	Arquitecto	• Paquete del análisis (esbozo) • Clase del análisis (esbozo) • Descripción de la arquitectura • (vista del modelo de análisis)
PUNTO DOS: Analizar un use case	Ingeniero de casos de uso	• Realización de U-C de análisis • Clase de análisis (esbozo)
PUNTO TRES: Analizar una clase	Ingeniero de componentes	• Clase del análisis (terminado)
PUNTO CUATRO: Analizar un paquete	Ingeniero de componentes	• Paquete del análisis(terminado)

3. Workflow de Diseño

Los propósitos perseguidos por el diseño son:

- Adquirir una comprensión profunda de aspectos relacionados con requerimientos no funcionales y restricciones del entorno de implementación.
- Refinar los requerimientos para subsistemas, clases e interfaces individuales.
- Descomponer el trabajo de implementación en piezas manejadas por diferentes equipos de desarrollo.
- Capturar interfaces entre subsistemas.
- Crear una abstracción de la implementación del sistema.

Trabajadores y Artefactos involucrados en el Diseño:



El rol del Diseño en el Ciclo de Vida

1. ***Fase de Comienzo:*** contribuye a delinear la arquitectura. En esta etapa defino donde pongo cada subsistema, como armo cada paquete de acuerdo a la arquitectura.
2. ***Fase de Elaboración y Construcción:*** contribuye a obtener una estructura más robusta y estable del sistema. Crea el plano para el modelo de implementación

Workflow de Diseño- Actividades- Responsables- Artefactos que se producen

Actividades:

1. Diseño de la arquitectura: esboza los modelos de diseño y despliegue identificando los siguientes elementos:

- ***1.1. Identificación de nodos y configuraciones de red:*** potencia de procesador, capacidad de memoria, tipo de conexión entre nodos, protocolos de comunicación, etc.
- ***1.2. Identificación de subsistemas y de sus interfaces:*** para poder reutilizarlos
- ***1.3. Identificación de clases de diseño:*** es suficiente la identificación de clases significativas para la arquitectura. Podemos partir de las clases del análisis, relevantes para la arquitectura. Para encontrar las clases activas hay que considerar los requisitos de concurrencia y se pueden esbozar a partir los resultados del análisis y del modelo de despliegue.
- ***1.4. Identificación de mecanismos genéricos de diseño:*** al identificarlos (requisitos) hay que decidir como tratarlos, teniendo en cuenta las tecnologías de diseño e implementación disponibles; estos requisitos están relacionados con aspectos como: persistencia, detección y recuperación de errores, gestión de transacciones, etc.

2. Diseño de un Caso de Uso: tiene como objetivo: identificar las clases del diseño y/o los subsistemas necesarios para realizar un U-C, distribuir los comportamientos entre los objetos, definir los requisitos sobre las operaciones de las clases.

- ***2.1. Identificación de clases del diseño:*** son las clases que se necesitan para realizar el U-C
- ***2.2. Descripción de las interacciones entre objetos:*** se debe describir como interactúan los objetos mediante diagramas de secuencia.
- ***2.3. Identificación de subsistemas e interfaces:*** se deben identificar los subsistemas e interfaces necesarios para realizar el U-C
- ***2.4. Descripción de interacciones entre los subsistemas:*** describir como interactúan los objetos de las clases que contiene en el nivel de subsistema. mediante diagramas de secuencia con instancias de actores, subsistemas y transmisores de mensajes.
- ***2.5. Captura de requisitos de implementación:*** se incluyen todos los requisitos identificados en el diseño que se tratarán en la implementación.

3. Diseño de una clase: crear una clase de diseño que cumpla su papel en la realización del U-C y los requisitos no funcionales que se aplican a estos.

- ***3.1. Identificar atributos***
- ***3.2. Identificar operaciones***
- ***3.3. Identificar asociaciones y agregaciones***
- ***3.4. Identificar generalizaciones***
- ***3.5. Describir estados:*** para algunos objetos describir sus estados con diagramas de estados

3.6. Tratar requisitos especiales: requisitos no tratados en pasos anteriores.

4. Diseño de un subsistema: garantizar que los subsistemas sean:

- ✓ Independientes uno del otro y de sus interfaces, tanto como sea posible.
- ✓ Que proporcionen las interfaces correctas
- ✓ Que cumpla su propósito de realizar correctamente las operaciones que definen sus interfaces.

Workflow de Diseño

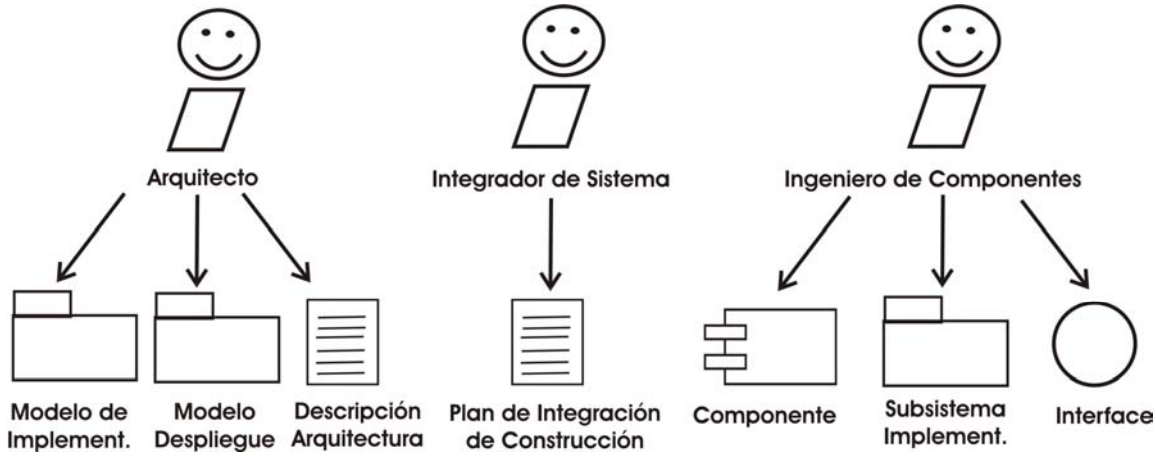
ACTIVIDAD	REALIZADO POR	RESULTADOS
PUNTO UNO: Diseño de la arquitectura	Arquitecto	• Subsistema (esbozado) • Interfaz (esbozado) • Clase de diseño (esbozado) • Modelo de despliegue (esbozado) • Descripción de la arquitectura (vista de los modelos de diseño y distribución)
PUNTO DOS: Diseñar un use case	Ingeniero de casos de uso	• Realización de U-C de Diseño • Clase de diseño (esbozado) • Subsistema (esbozado) • Interfaz (esbozado)
PUNTO TRES: Diseñar una clase	Ingeniero de componentes	• Clase del diseño (terminado)
PUNTO CUATRO: Diseño de un subsistema	Ingeniero de componentes	• Subsistema (terminado) • Interfaz (terminada)

4. Workflow de Implementación

En esta etapa especifico qué componentes y que nodos. La implementación tiene como propósito:

- Planear las integraciones requeridas en cada iteración
- Distribuir el sistema en componentes ejecutables maleables a los nodos del modelo de despliegue
- Implementar el diseño de las clases y subsistemas
- Prueba de unidad de componentes
- Integración de componentes en uno o más ejecutables

Trabajadores y Artefactos involucrados en la Implementación:



El rol de la Implementación en el Ciclo de Vida

- ***Fase de Elaboración:*** crear un artefacto ejecutable de la arquitectura.
- Se focaliza en la Fase Construcción.
- ***Fase de Transición:*** manejar los artefactos creados

Workflow de Implementación- Actividades- Responsables- Artefactos que se producen

Actividades:

1. Implementación de la arquitectura: debe esbozar el modelo de implementación y la arquitectura a través componentes ejecutables y de la asignación de componentes a los nodos

1.1. Identificación de los componentes significativos arquitectónicamente

2. Integrar el sistema: debe crear un plan de construcciones que describa las construcciones necesarias en una iteración y los requisitos de cada construcción.

2.1. Planificación de una construcción

2.1.1*. Identificar la realización del caso de uso-diseño correspondiente en el modelo de diseño.*

2.1.2*. Identificar los subsistemas y clases de diseño que participan en la realización del U-C-diseño.*

2.1.3. Identificar los subsistemas y componentes de implementación del modelo de implementación que son necesarios para implementar el U-C.

2.1.4. *Considerar el impacto de implementar los requisitos de estos subsistemas de implementación y de los componentes y evaluar si éstos son aceptables; si lo son implementar el U-C en la construcción.*

2.2. Integración de una construcción: se hace recopilando las versiones correctas de los subsistemas de implementación y de los componentes, compilándolos y enlazándolos para generar una construcción.

3. Implementar un subsistema: Se debe asegurar que los requisitos (escenarios o U-C) implementados en la construcción y aquellos que afectan al subsistema están implementados correctamente por componentes o por otros subsistemas.

3.1. Mantenimiento de los contenidos de los subsistemas

4. Implementar una clase: el propósito es de implementar una clase de diseño en un componente fichero.

4.1. Esbozo de los componentes fichero

4.2. Generación de código a partir de una clase de diseño

4.3. Implementación de operaciones

5. Realizar prueba de unidad: se debe probar los componentes implementados como unidades individuales.

5.1. Realización de pruebas de especificación: verifica el comportamiento del componente sin tener en cuenta como se implementa dicho comportamiento en el componente (es decir tiene en cuenta la salida que se devuelve cuando se le da una determinada entrada en un estado determinado)

5.2. Realización de pruebas de estructura: acá se verifica que el componente funcione internamente como se quería.

Workflow de Implementación

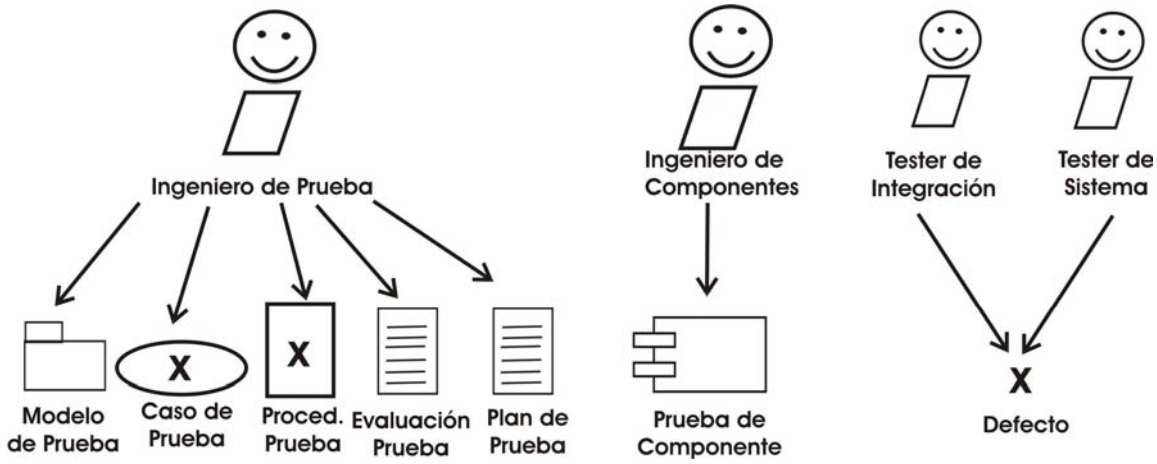
ACTIVIDAD	*REALIZADO POR*	*RESULTADOS*
PUNTO UNO: Implementación de la arquitectura	Arquitecto	• Componente (esbozado y posiblemente asignado a nodo) • Descripción de la arquitectura • (vista de los modelos de diseño y de despliegue)
PUNTO DOS: Integrar el sistema	Integrador de Sistema	• Plan de integración de Integración de construcciones • Modelo de implementación (construcciones anteriores)
PUNTO TRES: Implementar un subsistema	Ingeniero de componentes	• Subsistema de implementación (implementado para una construcción) • Interfaz (implementado para una construcción)
PUNTO CUATRO: Implementar una clase	Ingeniero de componentes	• Componente (implementado)
PUNTO CINCO: Realizar prueba de unidad	Ingeniero de componentes	• Componente (unidades probadas)

6. Workflow de Prueba

Propósitos de la prueba:

- Planear las pruebas requeridas en cada interacción
- Diseñar e implementar las pruebas, diseñando casos de prueba
- Ejecutar los test y actuar en consecuencia con los resultados de los test
- Creación de procedimientos de prueba y componentes ejecutables para la automatización de las mismas

Trabajadores y Artefactos involucrados en la Prueba:



El rol de la Prueba en el Ciclo de Vida

1. ***Fase de Comienzo:*** planificación de pruebas iniciales.
2. ***Fase de Elaboración:*** cuando la arquitectura ejecutable es verificada.
3. ***Fase de Construcción:*** cuando se implementa el sistema.
4. ***Fase de Transición:*** se concentra en defectos detectados durante el uso. En este punto es importante resaltar la existencia de la metodologías ligeras (Ejemplo XP-Extreme Programing), que se oponen a lo que plantea el Workflow de Prueba. En este tipo de metodologías, la planificación de los "lotes de pruebas" es posterior a la definición de los requerimientos.

Workflow de Prueba- Actividades- Responsables- Artefactos que se producen

Actividades:

1. Planificar prueba: se deben planificar los esfuerzos de prueba en una iteración, realizando:

- ***1.1. Una estrategia de prueba***
- ***1.2. Estimar los requisitos*** como ser: recursos humanos y sistemas necesarios.

2. Diseñar prueba:

Identificando y describiendo los casos de prueba para cada construcción
Identificar y estructurar los procedimientos de prueba especificando como realizar los casos de prueba.

- ***2.1. Diseño de los casos de prueba de integración***
- ***2.2. Diseño de los casos de prueba de sistema:*** se usan para probar que el sistema funcione correctamente como un todo, haciendo pruebas con combinaciones de U-C instanciados bajo condiciones diferentes.
- ***2.3. Diseño de los casos de prueba de regresión***
- ***2.4. identificación y estructuración de los procedimientos de prueba:*** para poder reutilizar procedimientos de casos de pruebas, necesitamos identificarlos y estructurarlos para poder luego modificarlos.

3. Implementar prueba: su propósito es de automatizar los procedimientos de prueba, creando componentes de prueba.

4. Realizar pruebas de integración: para cada una de las construcciones creadas en una iteración y se recopilan los resultados de las pruebas

5. Realizar pruebas de sistema: realiza las pruebas de sistemas necesarias en cada iteración y recopila los resultados de las pruebas. Comienzan cuando las pruebas de integración indican que el sistema sastiface los objetivos de calidad de integración fijados en el plan de pruebas de la iteración actual.

6. Evaluar prueba: su propósito es la de evaluar los esfuerzos de prueba en una iteración dada.

Workflow de Prueba

ACTIVIDAD	REALIZADO POR	RESULTADOS
PUNTO UNO: Planificar Prueba	Ingeniero de pruebas	• Plan de Pruebas
PUNTO DOS: Diseñar una prueba	Ingeniero de pruebas	• Caso de prueba • Procedimiento de Prueba
PUNTO TRES: Implementar una prueba	Ingeniero de componentes	• Componente de Prueba
PUNTO CUATRO: Realizar pruebas de integración	Ingeniero de pruebas de Integración	• Defecto
PUNTO CINCO: Realizar Prueba de Sistema	Ingeniero de pruebas de Sistema	• Defecto
PUNTO SEIS: Evaluar prueba	Ingeniero de pruebas	• Evaluación de pruebas (para una iteración)

Derivación del Sistema de Información a partir del modelo de negocios

El Modelo de Objetos del Negocio describe ***la realización*** de los Use Case de negocio. Me muestra cómo necesitan relacionarse los trabajadores y las entidades del negocio y como necesitan colaborar para ejecutar el negocio. ***Describe los uses cases desde el punto de vista interno del negocio***. El modelo de objetos del negocio define como debe relacionarse la gente que trabaja en el negocio con las cosas que maneja y usa -clases y objetos del negocio- para producir los resultados deseados.
Dicho modelo posee:

- **Entidades del Negocio:** son "cosas" manejadas, usadas por trabajadores del negocio, cuando ejecutan un use case de negocio. Solo debo modelar como entidades del negocio aquellos elementos o cosas a los que otras clases en el modelo de objetos van a hacer referencia.

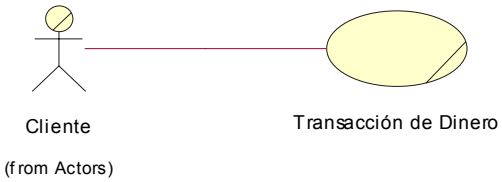
- **Trabajadores del Negocio:** representan el rol o los roles que los empleados que actuaran en el sistema.

Una forma de identificar casos de uso en el Sistema de Información, es a partir de los ***trabajadores del negocio*** del modelo de objetos del negocio. Los empleados actuando como trabajadores usan Sistemas de Información para comunicarse entre sí, y con los actores, para acceder a información acerca de las entidades del negocio. ***Siempre que hay una conexión, también hay potencialmente algún soporte del Sistema de Información.***

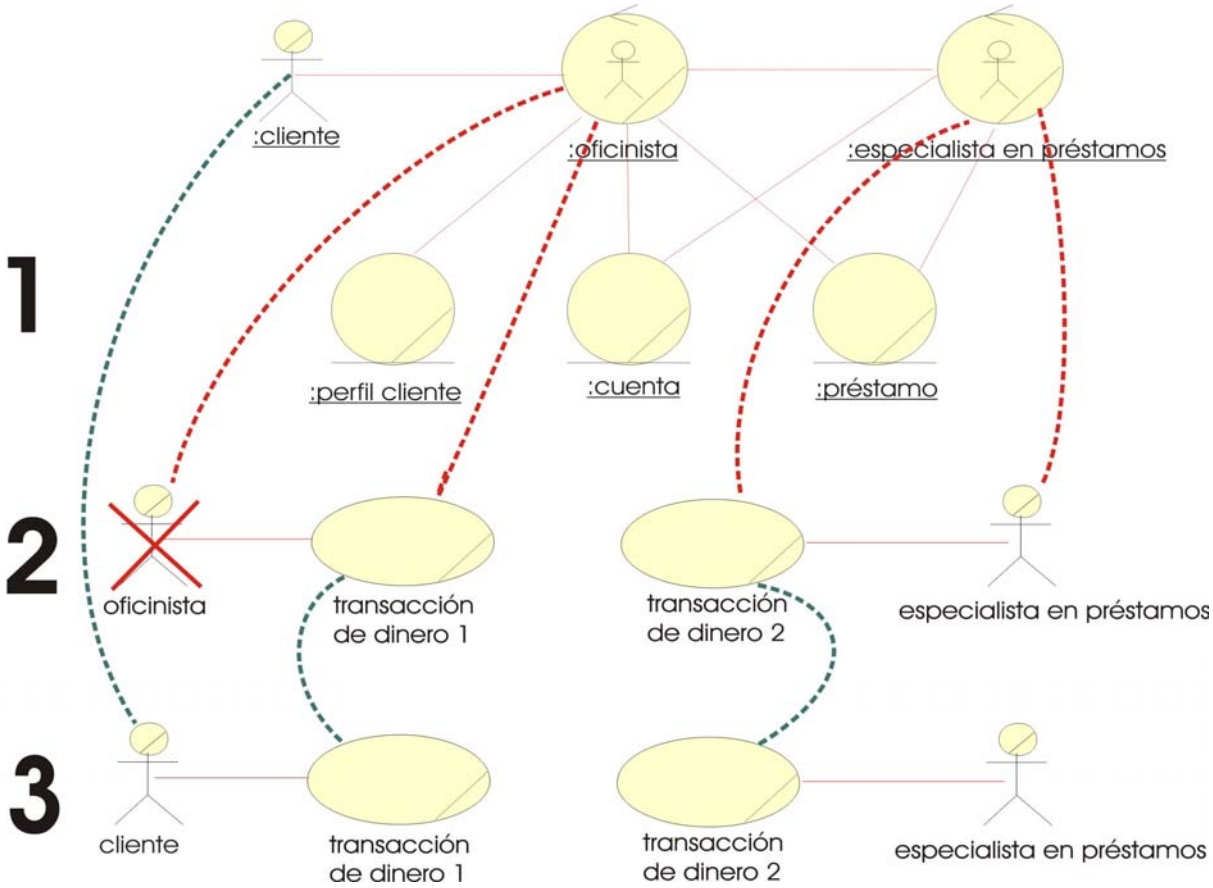
Conceptos Importantes

- Un empleado actuando como cierto trabajador corresponde a un actor del sistema de información.
- Decidir si el trabajador de negocio usará el Sistema de Información
- Si es así, hacer un use case con el actor del Sistema de Información que tenga el mismo nombre que el trabajador del sistema de negocio
- Por cada Use Case del sistema de negocios en que el trabajador participe como actor del Sistema de Información crear un Use Case del Sistema de Información.

A modo de ejemplo trabajaremos con el siguiente Caso de Uso:



En el siguiente caso de Uso identificamos a un actor denominado “Cliente” quién va a realizar una “Solicitud de Préstamo” en un Banco. Como muestra el siguiente gráfico, ***en el ítem 1***, para que dicho caso de uso pueda realizarse se necesitara de la interacción y colaboración de los trabajadores de negocio (Oficinista y Especialista en Préstamos, en nuestro caso) y las clases perfil de cliente, cuenta y préstamo.



En el punto 3 analizamos que sucede cuando el trabajador de negocio es automático. En este caso las responsabilidades del trabajador de negocio se han movido hacía el cliente y el actor ha sido removido del sistema.

Patrones de Análisis y Diseño

Introducción: los patrones describen la forma de utilizar las clases y los objetos a la hora de construir software.
Un patrón es una plantilla que brinda soluciones probadas para determinadas problemáticas, es decir provee soluciones a aquellas situaciones recurrentes que siempre responden de la misma forma. En otras palabras, cada patrón describe un problema que es reiterativo en el entorno, para describir luego una solución a ese problema, de forma tal que esa solución pueda ser utilizada también de manera reiterativa, ya que los propios patrones se reutilizan cada vez que se vuelven a aplicar.

Origen del Concepto:

El concepto o la idea central de los “Patrones de Diseño” provienen del campo de la Arquitectura. A finales de los 70, Christopher Alexander, escribió varios libros acerca de urbanismo y construcción de edificios en donde planteó la reutilización de diseños ya aplicados en otras construcciones que catalogó como modelos a seguir.
Estas ideas luego fueron tomadas y utilizadas en el campo de la programación para desarrollar lenguajes de patrones como guía para los programadores de Smalltalk primero y C++ después. Pero no fue sino hasta los años 1990/1994 cuando Erich Gamma, Richard Helm, Ralph Johnson y Hohn Vlissides (conocidos como el grupo de los cuatro) realizaron el primer catálogo de patrones de diseño, publicado en el libro "Design Patterns:Elements of Reusable Object-Oriented Software".
Este podría considerarse como el aporte más importante sobre “Patrones de Diseños”.

Clasificación de los Patrones

1. Patrones para la Construcción de Modelo de Objetos del Dominio
2. Patrones para la Asignación de Responsabilidades (GRASP)

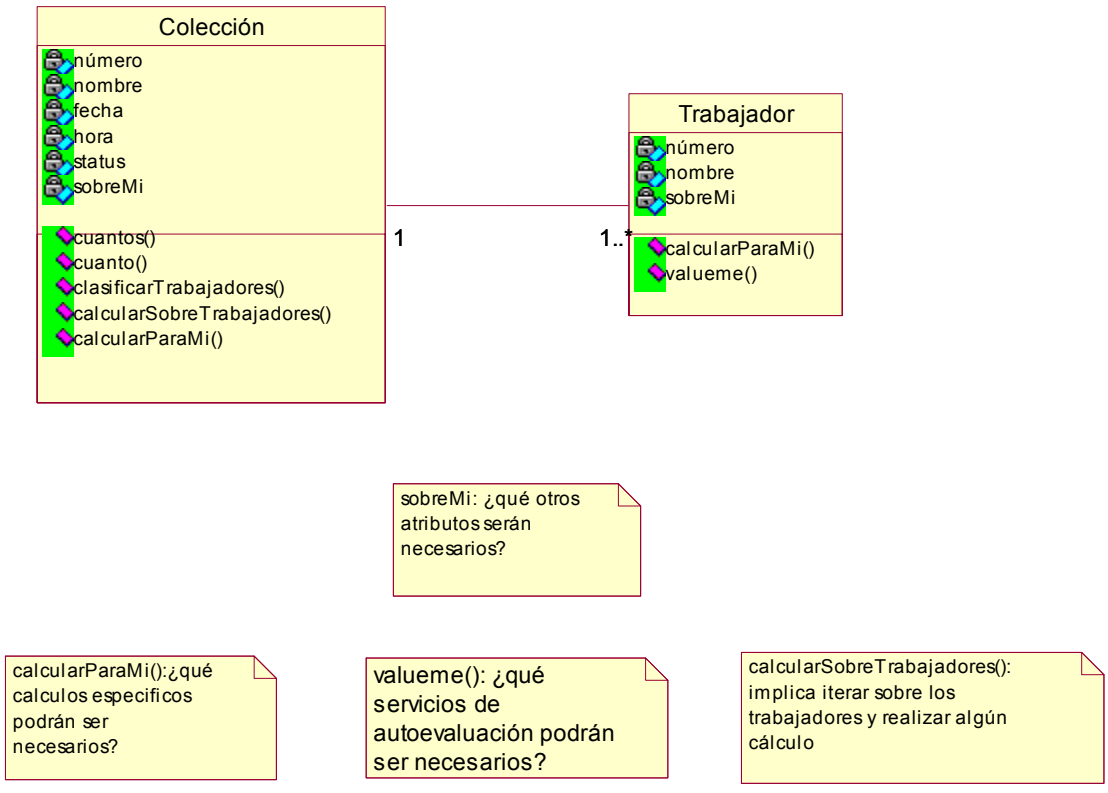
Para utilizar los patrones primero debo analizar las relaciones que existen en mi diagrama de clases, para ello siempre analizo las clases “de a pares” buscando que patrón podría utilizar para resolver la problemática planteada por las mismas

Patrones para la construcción del Modelo de Objetos del Dominio

1. ***Patrón Fundamental***
2. ***Patrones Transaccionales***
3. ***Patrones de Agregación***
4. ***Patrones de Plan***

1. El Patrón Fundamental

Es el esquema seguido por todos los patrones, es el patrón fundamental del modelo de objetos. Se basa en la definición de que los objetos son vagos, lo que un objeto no puede hacer por si mismo lo realiza otro objeto.



Los patrones son de gran ayuda a la hora de definir los atributos de las clases y objetos instanciados de dichas clases. También ayudan a definir los comportamientos, esto último es fundamental, ya que podríamos decir que la

definición de el patrón fundamental se basa en el concepto ya estudiado de que "***un objeto hace cosas por si solo o con la ayuda de otros***".
Al definir el comportamiento del sistema, he encontrado la esencia del mismo, siempre debo buscar los componentes de la conducta que no van a cambiar, la línea de conducta que no va a modificarse.
Cabe recordar que los atributos de las clases y objetos pueden modificarse, cambiar de valor, pero la línea de conducta debe perdurar.

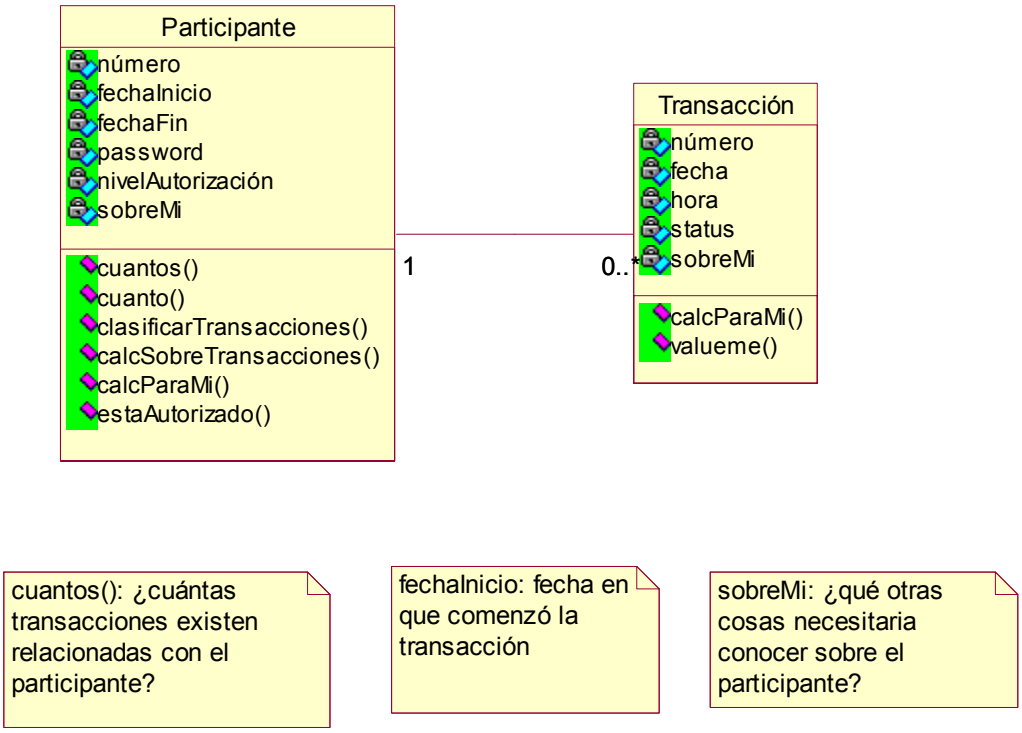
2. Los Patrones Transaccionales

Los patrones transaccionales son aquellos que tienen un jugador de transacción, o un conjunto de jugadores que se relacionan con algún jugador de transacción. Para entender este concepto analizaremos el significado del concepto "Transacción".
Una "Transacción" es la registración de un evento. Un evento es algo que sucede y que es capaz de modificar el estado de un objeto. A nivel Base de Datos la registración de un evento podría ser o bien la modificación de una tupla de la misma o la inserción de una instancia en una tabla relacional. Al grabar algunas de estas transacciones en el disco, recuerdo la ocurrencia de dicho evento.

Los patrones de transacción son:

- actor-participante (ejemplo: persona- cajero)
- participante-transacción (ejemplo: cajero- sesión)
- lugar-transacción (ejemplo: almacén- venta)
- ítem especifico-transacción
- transacción- detalle de transacción (ejemplo: factura- ítem de factura)
- transacción- transacción subsiguiente (ejemplo: venta- pago)
- detalle de transacción- detalle de transacción subsiguiente
- ítem- detalle de transacción
- ítem específico-detalle de transacción
- ítem- ítem específico
- asociación-otra asociación
- ítem específico-jerarquía de ítem

Ejemplo: Participante- Transacción



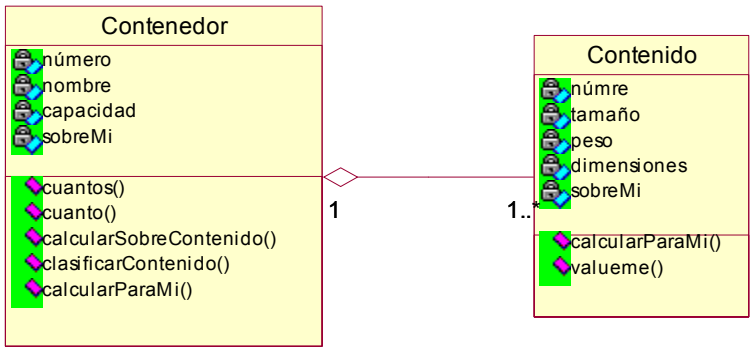
En este tipo de patrón el participante siempre representa roles de personas como: solicitante, comprador, cliente, cajero, oficinista, profesor, trabajador. Las transacciones representan: contratos, acuerdos, entregas, órdenes, pagos, etc.

Observaciones: que la línea sea sin navegabilidad implica que solo le pregunto al participante cosas que sabe por si solo, sin la ayuda de otros objetos.

3. Patrones de Agregación

- contenedor-contenido. (ejemplo: almacén-registradora)
- contenedor-detalle de contenedor.
- grupo-miembro(ejemplo: almacén- artículo)
- todo- parte
- parte de compuesto- parte
- paquete-componente de paquete

Ejemplo: Contenedor- Contenido



4. Patrones de Plan: (no vistos)

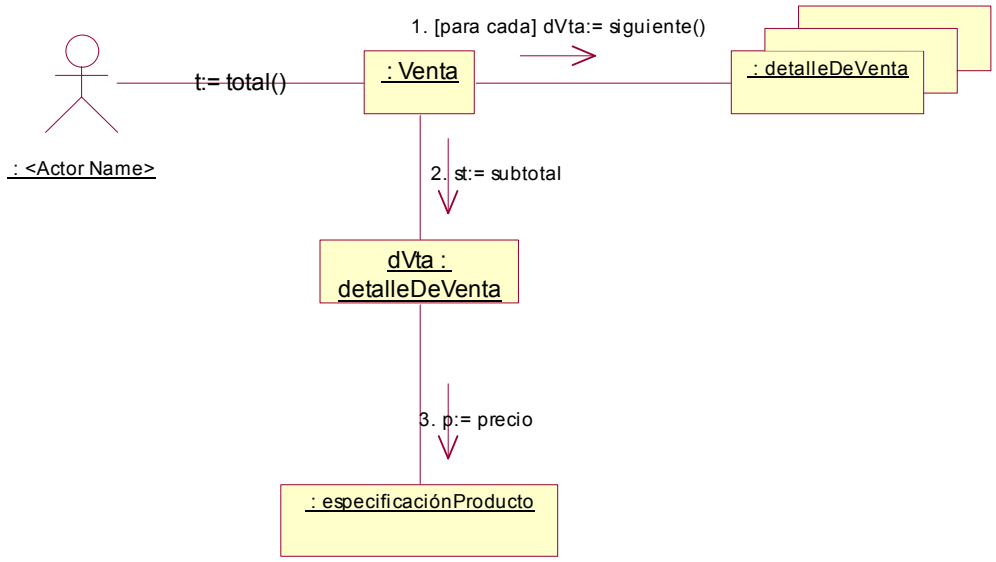
Patrones para la Asignación de Responsabilidades

1. ***El Patrón Experto***
2. ***El Patrones Creador***
3. ***El Patrón Bajo Acoplamiento***
4. ***El Patrón Alta Cohesión***

1. El Patrón Experto

En este tipo de patrón cada objeto es experto en hacer cosas relacionadas con la información que posee. Se basa en el concepto ya estudiado de que "los objetos hacen cosas relacionadas con la información que poseen"

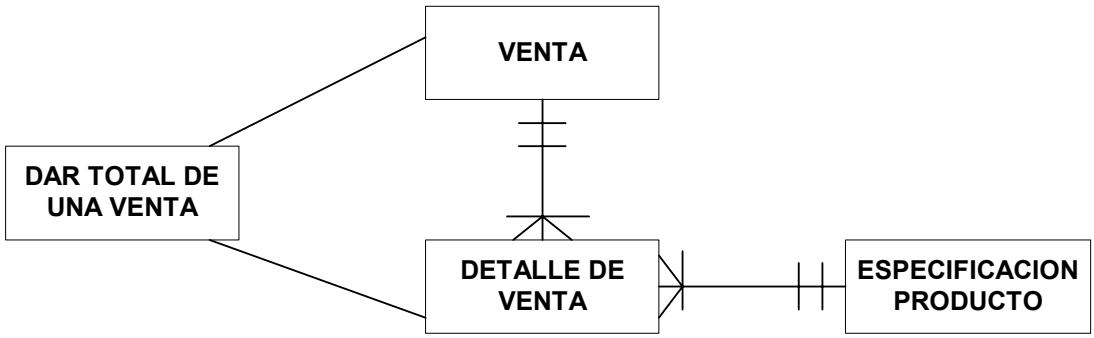
Ejemplo: Sistema de Punto de Venta



Venta: es el responsable de conocer el total de la venta
detalleDeVenta: es el responsable de conocer el subtotal de cada línea de detalle
especificaciónProducto: es el responsable de conocer el precio de cada producto

En el ejemplo anterior está claro como cada objeto hace cosas relacionadas con la información que posee.

Comparación con el Paradigma Estructurado



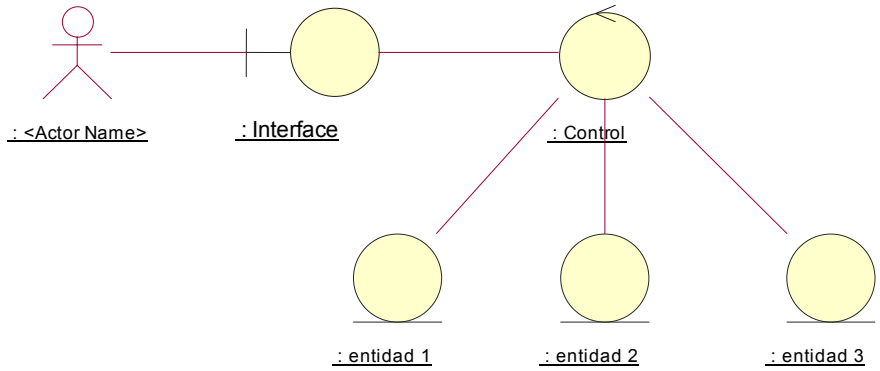
Como estudiamos anteriormente, el Paradigma Estructurado observa a cualquier ente de la realidad como si estuviera dividido en dos partes: datos y acciones. Las acciones exteriorizan un comportamiento y están contenidas en los módulos de un Diagrama de Estructuras, mientras que los archivos conteniendo los *datos* (entidades relacionales) se encuentran almacenados en archivos en discos duros (en general).
En el ejemplo, es el módulo "Dar Total de una Venta" quién contiene el comportamiento, el código con los métodos y tanto "Venta", como "Detalle de Venta" y "Especificación de Producto" son las entidades relacionales, que vistas desde el POO son objetos incompletos, sin conducta.

En forma genérica, el módulo "Dar Total de Una Venta", tendría el siguiente código:

```
Por cada DetalleDeVenta
Donde nventa=X
Por cada Producto
donde CodProd = CodProd
Hacer precio = precioprod
Fin por
Total = cantidad * precio
Fin Por
```

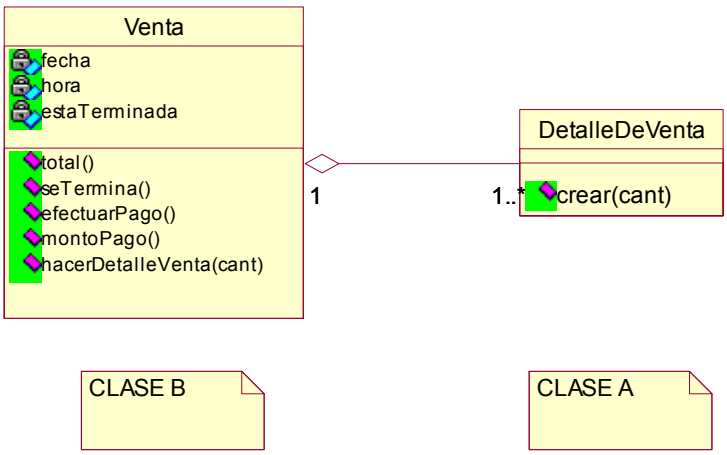
En el Paradigma Estructurado toda la responsabilidad recae sobre el módulo, a diferencia del Paradigmas OO, en donde las responsabilidades están divididas entre los objetos.
Sin embargo esto muchas veces no sucede en POO y toda la responsabilidad recae sobre un mismo objeto de "Control", al igual que en el Paradigmas Estructurado sucede con los módulos. A lo descrito anteriormente se lo denomina "POO desvirtuado"

Ejemplo:



2. El Patrón Creador:

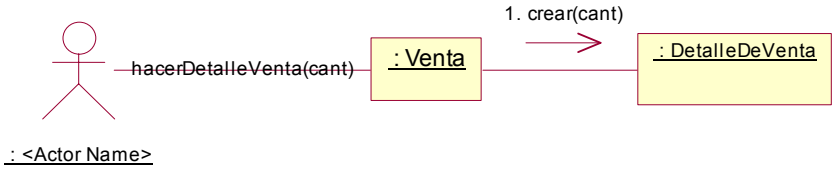
Define quién tiene la responsabilidad de crear un objeto. Plantea resolver el problema de ¿Quién debería ser el responsable de crear una nueva instancia de alguna clase?



En el ejemplo la responsabilidad de crear una instancia de A la tiene B si:

- B agrega los objetos de A.
- B contiene los objetos de A (relación contenedor- contenido)
- B registra las instancias de los objetos de A.
- B utiliza específicamente los objetos de A.
- B tiene los datos de inicialización que serán transmitidos a A cuando sea creado
  (así B es un experto respecto de la creación de A).

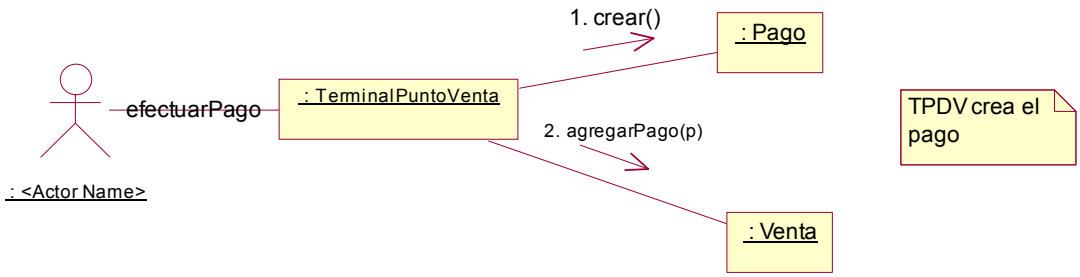
Ejemplo: Sistema de Punto de Venta, ¿quién debe crear el detalle de venta?



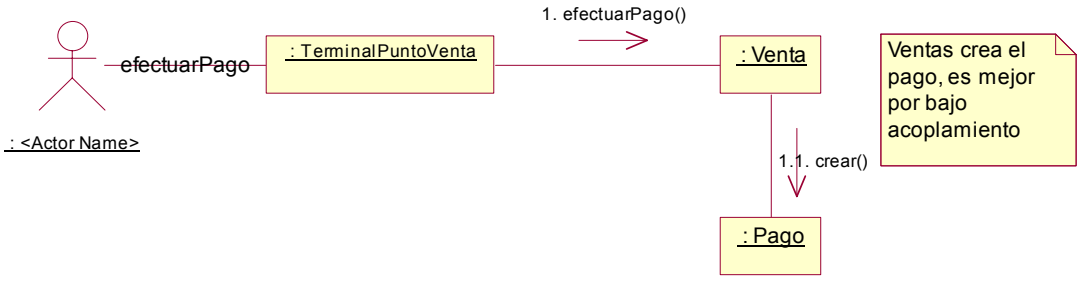
En este caso "Venta" es quién tiene la responsabilidad de crear el DetalleDeVenta.

3. El Patrón Bajo Acoplamiento

Este patrón busca mantener bajo el acoplamiento mediante la asignación de responsabilidades. El término "Acoplamiento" hace referencia a la "cantidad de líneas de relación entre dos clases" y cuan fuerte es dicha relación.
En el siguiente ejemplo la clase "Terminal de Punto de Venta" realiza todas las tareas, esta sobrecargada y acoplada con las clases "Ventas" y "Pago". Las clases con alto grado de acoplamiento son menos reutilizables y ante algún cambio en las clases afines también deben modificarse.

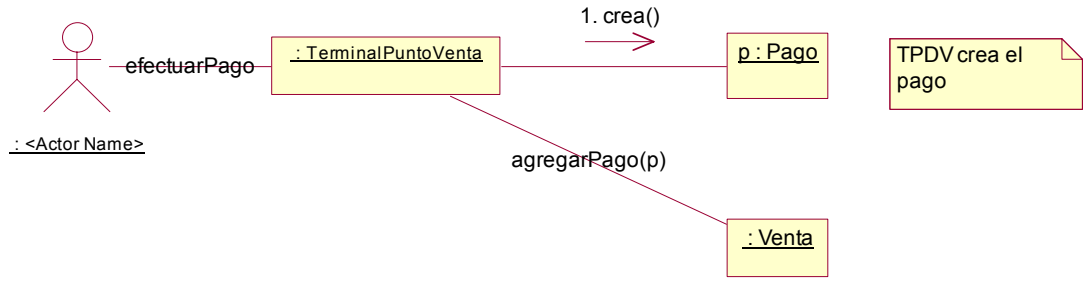


En este segundo caso vemos como las responsabilidades han sido repartidas entre las clases y cada objeto está acoplado con un solo objeto.

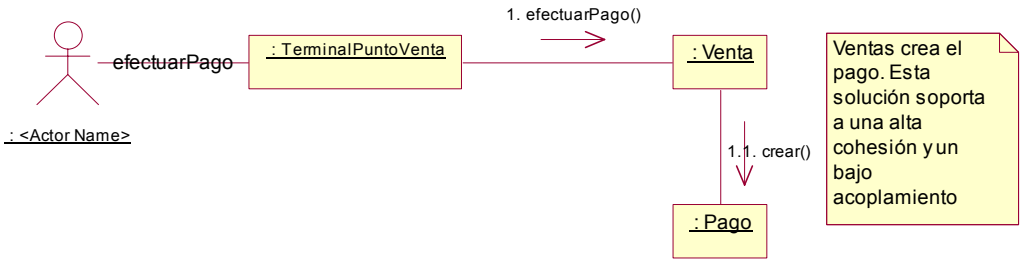


4. El Patrón Alta Cohesión

Este patrón busca que la cohesión siga siendo alta mediante la asignación de una responsabilidad.
El término cohesión está relacionado con todas las responsabilidades que le corresponden a un objeto, es una medida de cuanto están relacionadas las responsabilidades con la semántica de una clase, recordemos que cada clase hace lo que le corresponde hacer, lo que está relacionado con su semántica.



En el primer ejemplo es el objeto “Terminal Punto de Venta” quien crea el pago, en el segundo caso ilustrado es “Venta” quién crea el pago. Esta última solución soporta una alta cohesión y un bajo acoplamiento.



Patrones de Diseño- Introducción

Al estudiar las características del Paradigma Estructurado vimos que los datos y el comportamiento están separados. Los datos son almacenados en tablas relacionales y cada tabla relacional está asociada con un conjunto de módulos (que contienen el comportamiento) mediante "líneas de acoplamiento intrínseco".
Esto no sucede en el POO, donde los objetos poseen atributos y comportamiento y de esta forma ofrecen una manera de ver la realidad "más cercana a la realidad".
Ahora bien, para que esta realidad pueda representarse es necesario que dichos objetos, que la mayoría de las veces poseen información que necesita ser persistente, puedan almacenarse y sobrevivir a las ejecuciones que los crean.
Esto genera el problema (tecnológico) de donde almacenar a los objetos, recordemos que la RAM es volátil y que además solos los datos pueden almacenarse y no el comportamiento.
Existe un patrón que ofrece una solución al problema tecnológico y que permite guardar objetos persistentes en un BD.

El Patrón Wrapper:

Este patrón se basa en que cualquier objeto puede ser envuelto por otro objeto. Como muestra el gráfico siguiente el objeto "Wrapper" envuelve al objeto del dominio del problema e intercepta y analiza los mensajes que este recibe, si el mensaje no modifica el estado del objeto "lo deja pasar". Si sucede lo contrario, guarda el nuevo estado en la BD y de esta forma el objeto (del dominio del problema) no se entera de los cambios que se han realizado.