TEMA 1

Introducción

SOFTWARE

El software son Instrucciones que cuando se ejecutan dan la función y el comportamiento deseados. Son Estructuras de datos que facilitan a los programas el manipular adecuadamente la información y También lo son Documentos que describen la operación y uso de programas.

Características software como elemento lógico : Se desarrolla (no se fabrica), No se estropea, se deteriora por los cambios y La mayoría se construye a medida en vez de ensamblar componentes ya existentes.

Crisis del Software

El desarrollo de software viene con el desarrollo de sistemas informáticos. Sobre mitades de los 70 empieza la crisis.

Causas: Lo que se hacía se hacía muy lento, muchas veces mal, no servía, al poco tiempo dejaba de funcionar…

- Los productos exceden la estimación de costes.
- Prestaciones inadecuadas.
- Modificaciones a precios prohibitivos.
- Notables retrasos en la fecha de entrega.
- Mantenimiento casi imposible.
- Falta de fiabilidad del producto software.

SOFTWARE DE CALIDAD

Software de calidad es aquel que tiene concordancia con:

1. Requisitos : Los requisitos funcionales y de rendimiento establecidos explícitamente.
2. Estándares : Los estándares de desarrollo explícitamente documentados.
3. Profesionalidad : Las características implícitas que se espera de todo software profesional.

Factores de calidad

Tres aspectos importantes de un producto software que se deben medir, a lo largo de todo el proceso de desarrollo:

Características operativas

1. Corrección: ¿Hace lo que quiero?
2. Integridad : ¿Es seguro?
3. Facilidad de uso : ¿Está diseñado para ser usado?
4. Fiabilidad : ¿Lo hace de forma fiable todo el Tiempo?
5. Eficiencia: ¿Se ejecutará lo mejor que pueda?

Capacidad de soportar los cambios

6. Facilidad de : mantenimiento ¿Puedo corregirlo?
7. Flexibilidad : ¿Puedo cambiarlo fácilmente, mejorarlo, añadirle cosas?
8. Facilidad de prueba : ¿Puedo probarlo? cuando cambie algo debo de poder volver a hacer las mismas pruebas

Adaptabilidad a nuevos entornos

9. Reusabilidad : ¿Podré reusar alguna parte del Sw?
10. Portabilidad : ¿Podré usarlo en otra máquina o SO?
11. Facilidad de interoperación: ¿Puedo hacerlo interactuar con otro sistema?

Problemática de la industria del software

Los productos no son de calidad: Poca inversión y esfuerzo en el análisis y la especificación, Uso de lenguajes informales y modelos inadecuados.

Altos costes de desarrollo y mantenimiento : Naturaleza no física de la programación, Poca teoría y poca difusión, Productos ya en el mercado dificultan la innovación.

Grandes retrasos: Artesanía, Trabajo en grupo, Comunicación con usuario, Gestión de por no informáticos.

Soluciones

Formalización: Métodos de razonamiento formal (*lenguajes formales de especificación ejecutables: lógica + álgebra*), nuevos modelos de desarrollo y modificación del ciclo de vida.

Difusión de los avances tecnológicos : Nuevos paradigmas de programación, Arquitecturas, protocolos y modelos de computación

Inversión en herramientas : Entornos de desarrollo modernos, Generadores de documentación.

LA INGENIERÍA DEL SOFTWARE (*INS*)

La INS es una ciencia (disciplina), se considera que hace falta mantenimiento y es Procedimental.

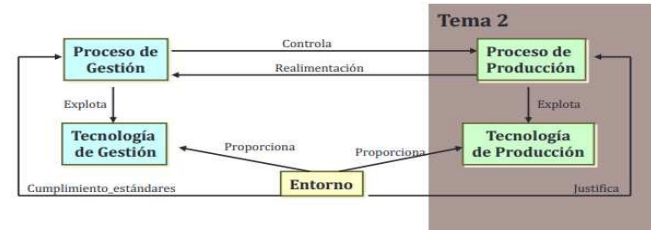
La INS sin Artesanía : La INS es algo más que programar, El proceso de la INS comienza bastante antes de escribir líneas de código y continúa después de que la primera versión del producto haya sido completada. *Esto es NO ser artesanal: no escribir a pelo sino planear antes lo que vas a hacer.*

Son cruciales la planificación, el seguimiento y el control rigurosos de los proyectos software. Para que los productos puedan ser adaptables al entorno ya que este cambia rápido.

El Proceso del Software

Gestión de proyectos software

La administración de un proyecto software es el primer nivel del proceso de ingeniería del software y CUBRE TODO EL PROCESO DE DESARROLLO. Por lo que se requiere una especificación del proceso del desarrollo.



Actividades que incluye la gestión de proyectos : *Han de poder dar respuestas rápidas al cambio de requisitos.*

- Redacción de la propuesta.
- Estimación del coste del proyecto.
- Seguimiento y control del proyecto.
- Planificación del proyecto.
- Selección y evaluación del personal.
- Redacción y presentación de informes.

Ingeniería del Software VS Ingeniería

Similitudes: Se parecen en que muchas técnicas de gestión de proyectos de ING son aplicables a los de INS. De igual forma, problemas complejos de los proyectos de ING aparecen en los de INS.

Diferencias: Por otro lado, a diferencia del producto de la ING, el de la INS es intangible y flexible. El proceso no es estándar y muchas veces se habla de proyectos único (*poca replicación*).

TEMA 2

El Proceso del Software

EL PROCESO DEL SOFTWARE

Establece un marco para el desarrollo de software con 2 partes principales: En general el término "Proceso Software" se asocia al proceso de producción pero que incluye también el proceso de gestión.

Son actividades que se centran en el desarrollo/evolución de software → Ciclo de Vida

Ciclo de vida

Todo esto se conoce también como Ciclo de Vida, el conjunto de actividades que se realizan. Las actividades principales son:

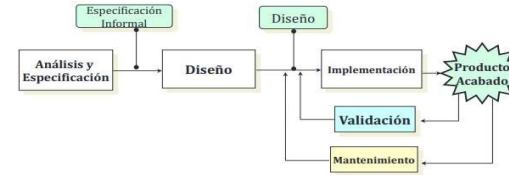
Especificación, Desarrollo, Validación, Evolución, Análisis, Diseño, Implementación, Pruebas y Mantenimiento

Modelos: Según como se organice y se haga esto, se destacan 5 modelos de ciclo de vida.

Codificar y corregir (*code-and-fix*) : Solo se trata de implementar, probar y volver implementar… *Sin análisis, sin diseño, sin mantenimiento...*

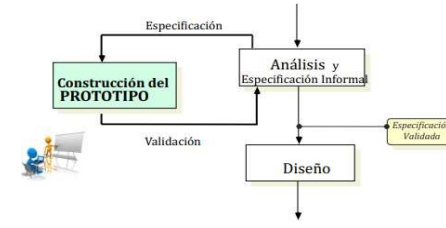
Clásico o en cascada : Hacer secuencialmente las actividades. De cada una de estas actividades se consigue un resultado.
Cuando llega la hora de probarlo, primero se hace parte por parte, después se comprueba todos juntos y se ve si funcionan.
Por último, se prueba por el cliente, para que de feedback y si lo acepta.

Vuelta a tras: Cada una de las pruebas desencadena en una vuelta a fases anteriores para corregir errores. Se puede llegar a invertir mucho tiempo en algo que, al cliente no le gusta. Lo que aumenta los costes y el tiempo de desarrollo.



Muchas veces no se vuelve al punto donde se debería, se ponen parches y tirando. Esto deforma el modelo.

Clásico con prototipado : Vamos a hacer una primera versión del producto (*prototipo*) y se lo enseñas al cliente a ver que tal le parece, así después no tengo que cambiar todo desde abajo, lo voy cambiando poco a poco y hago otro prototipo.

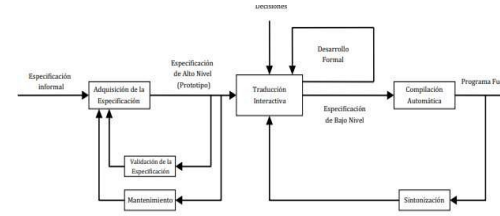


Clases de Prototipos : *Antes de enseñárselos al cliente*

- Vertical: Desarrolla completamente alguna de las facetas concreta del programa.
- Horizontal: Desarrolla en parte todas las facetas a la vez e ir terminándola poco a poco.

Pros / Contras : Permite al cliente ir viendo cómo va el desarrollo y orientar al desarrollador poco a poco hacía lo deseado reduciendo riesgos. *Aun que puede ser más costoso, dar falsas esperanzas (de que ya está hecho) etc.*

Programación Automática: Pretende Introducir la automatización en el proceso de construcción del software. Se da una especificación formal de un prototipo y el bicho se supone que te "se deriva" dando como resultado partes del resultado mediante el uso de lenguajes formales.



Prototipado Clásico	Programación automática
Especificación informal	Especificación formal
Prototipado no usual	Prototipado estándar
El prototipo se crea manualmente	La especificación es el prototipo
El prototipo se desecha	Evoluciona hacia el producto final
Implementación manual	Implementación automática
El código ha de probarse	Sin pruebas
Mantenimiento sobre el código	Mantenimiento sobre la especificación

Pros / Contras: Ayuda a reducir errores humanos, Reduce el coste de desarrollo, sin embargo, es algo muy difícil, *cuesta mucho hacer la traducción a lenguajes formales*. Antecesor del MDE/MDA (*que no el MDMA*)

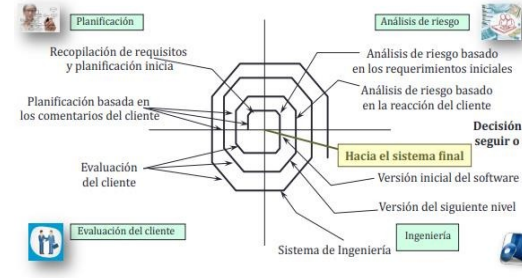
Modelos evolutivos: Es una modelo Adaptable a requisitos cambiantes, Se elaboran versiones cada vez más completas del SW: Se parte de una solución mínima y se van añadiendo cosas de manera interactiva e iterativa. Tenemos dos puntos de vista: Incremental y En espira, *Son parecidos.*

- **Incremental**: Se va por incrementos, se va añadiendo poco a poco funcionalidades.
  Cada adición se pasa por cada prueba análisis, diseño, codificación, pruebas y dejar que el cliente testee y valide. Si algo no está ok lo arreglas antes de añadir algo más, luego sigues.

  Pros / Contras: Bueno si no hay suficiente personal para hacer todo de golpe y es muy interactivo con el usuario. Sin embargo, es difícil saber "como" incrementar en cada iteración.

- **En espira**: Es muy similar al otro, pero este Introduce el análisis de riesgos en el desarrollo.

  Igual que antes se parte de un diseño inicial pequeño, y poco a poco se ban añadiendo cosas. Pero en cada paso se hace un análisis de riesgos, si este me dice que no renta, chapas y a casa. Si te dice de seguir, pues sigues.



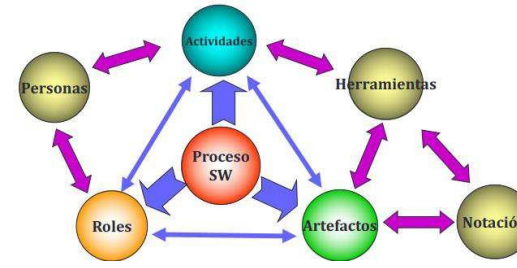
  Pros / Contras: Tiene interactividad con el usuario y cada vez las versiones son más completas.
  Sin embargo, es difícil evaluar los riesgos y asegurar que se avanza hasta el final: *A lo mejor siempre hay algo que mejorar y te quedas en bucle haciendo y probando todo el rato.*

En vez de partir de uno nuevo, siempre mejoras el anterior hasta que llegue a ser la versión final (Por eso puede ser que nunca termines si siempre le estás añadiendo cosas). Por otra parte, en la de prototipos, tu vas desarrollando cosas y cuando quieres enseñar al cliente creas un prototipo nuevo.

METODOLOGÍA

En un proyecto de desarrollo de software, la metodología define: Quién / Qué / Cómo / Cuándo. La metodología Define un proceso explícito de desarrollo de software. Este debe ser repetible, mejorable, medible y optimizable.

- Artefacto: Es lo que creas.
- Roles: Quienes hacen que cosas en cada punto.
- Actividades: Como lo consigo.



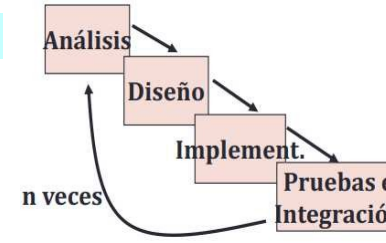
Diferencia entre artefacto y herramienta: Artefacto es lo que creas, herramienta es lo que usas para hacer el artefacto.

No existe una metodología de software universal. Cada cosa requiere algo que se ajuste al producto, a las personas, los recursos…

Metodologías principales Tradicionales: *Rational Unified Process* (*RUP*)

Usa los casos de uso como "marco director" del desarrollo. *Se usan desde la planificación hasta la especificación*.

Es Iterativo e incremental



Vas añadiendo software y mejorando el ya hecho usando estos casos de uso.
Para cada caso de uso se hace un *Análisis, diseño, implementación y las pruebas*.

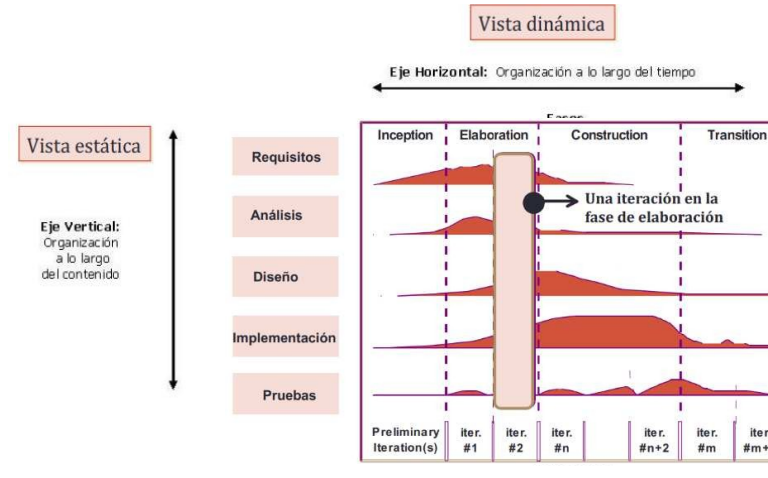
Actividades de la interacción: Planificar la iteración (riesgos), Análisis de Casos de Uso y Escenarios, Diseño Opciones Arquitectónicas, Implementación, Pruebas, Integración, Evaluación de la entrega, Preparación de la entrega.

El conjunto de todos los casos de uso serán el producto final

Visiones

Visión estática: El qué, como, cuando, quién…

Visión dinámica: A lo largo del tiempo. Se centra en 4 fases, y en cada fase le va dando más importancia a la parte que toque.



- Inicio (*Estudio de Oportunidad*): Define las bases del proyecto, el ámbito, objetivos del proyecto, funcionalidad y capacidades del producto.
- Elaboración: Se empieza a hacer el proyecto una vez se diseña y planea todo, funcionalidad deseada, se defina una arquitectura básica. Todo en función de los recursos disponibles.
- Construcción: Análisis diseño e implementación. Es donde se implementa todo a la vez que se testea y se documenta.
- Transición: Cuanto ya está todo desarrollada y se ha de lanzar, se entrega al usuario, se hace el marketing, las instalaciones... Se supone que ha de estar todo pulido (A excepción de cosas que se pueda arreglar después, MANTENIMIENTO).

Metodologías principales Ágiles: *XP*

Estas metodologías se valoran al individuo y las interacciones en el equipo de desarrollo más que a las actividades y las herramientas. Mejor desarrollar buen software y no centrarse tanto en la documentación (*tiene que haber, pero no es main*). El cliente es muy importante, se le ha de casi "*incluir en el equipo*". También se tiene Responder a los cambios más que seguir estrictamente una planificación.

Suelen funcionar mejor porque te deja tener una planificación y tener buena interacción con el cliente mientras que no es 100% restrictivo.

Principios

1. La prioridad principal es satisfacer al cliente mediante tempranas y continuas entregas de software usable.
2. Dar la bienvenida a los cambios. Los procesos ágiles aplican los cambios para que el cliente sea competitivo: *Si algo no se tenia planeado por vemos que le va a ser un buen cambio se hace.*
3. Entregar el software desarrollado frecuentemente con el menor intervalo de tiempo posible entre una entrega y la siguiente: *Que el cliente vea que se van haciendo las mejoras y que va todo bien.*
4. La gente de negocios y los desarrolladores trabajan juntos a en cada proyecto: *Es muy útil que estén juntos.*
5. Construir proyecto empujados por motivaciones personales. Dar el entorno que necesitan las personas y confiar en ello: *Que lo hagan por ellos mismos sin que sean obligados.*
6. El diálogo cara a cara es el método más eficiente y efectivo para comunicar información para un equipo.
7. Desarrollar software es la primera medida de progreso.
8. Promueven un desarrollo llevador. Reduce los problemas entre el personal (patrocinadores, desarrolladores y usuario), Se apoyan entre ellos para llevar a cabo el proyecto con la mejor calidad posible.
9. La simplicidad es esencial. *Aquí hay 3 más que no entiendo y se los va a aprender tu madre :3.*

Metodología Ágil	Metodología No Ágil
El cliente es parte del equipo de desarrollo	El cliente interactúa con desarrollo mediante reuniones
Grupos pequeños (< 10) y trabajando en el mismo sitio	Grupos grande
Pocos artefactos	Más artefactos
Pocos roles	Bastantes roles
Menos énfasis en la arquitectura	La arquitectura es esencial
Heurísticas	Rigurosas
Impuestas internamente (por el equipo)	Impuestas externamente
Proceso menos controlado, con pocos principios	Proceso mucho más controlado, con numerosas normas
No existe un contrato tradicional o al menos es flexible	Existe un contrato prefijado

Extreme Programming (XP) *No se yo que tan importante es esto*

Historias, Iteraciones, Versiones, Tareas y Casos de Prueba

El cliente selecciona la siguiente versión a construir, eligiendo las características funcionales que considera más valiosas (*llamadas Historias*) de un conjunto posible de historias, *siendo informado de los costes y del tiempo que costará su implementación*. Los programadores convierten las historias en tareas a realizar y a continuación convierten las tareas en un conjunto de casos de prueba para demostrar que las tareas se han finalizado. Trabajando con un compañero el programador ejecuta los casos de prueba y evoluciona el diseño intentando mantener su simplicidad.

Se hacen entregas pequeñas, mediante pruebas, metáforas, elección del cliente… Se hacen pequeños grupos de desarrollo dentro del mismo equipo para desarrollar conjuntamente (apoyados por el cliente). Son importantes los estándares de codificación

Tema 3

Arquitectura del software

Programming in the *SMALL/MEDIUM/LARGE*: Cuando los sistemas software crecen en tamaño, se requiere una organización de los mismos en subsistemas que los hagan manejables. Hay muchas maneras de manejar la complejidad, generalmente creando niveles de abstracción. Sin embargo, estas son de bajo nivel, solo es código agrupado. Hacen falta mecanismos con más abstracción → *Separara la aplicación en Bloques funcionales*.

ARQUITECTURA DEL SOFTWARE

La arquitectura de software tiene que ver con el diseño y la implementación de estructuras de software de alto nivel. Es el resultado de juntar elementos arquitectónicos de forma adecuada para satisfacer la mayor funcionalidad y requerimientos de desempeño de un sistema, así como *requerimientos no funcionales*, como la *confiabilidad*, *escalabilidad*, *portabilidad*, y *disponibilidad*

En la descripción de la Arquitectura del Sistema, debemos dotar al sistema de una organización en subsistemas.

Tipos de Sistemas (entre otros muchos…)

- Sistemas Personales: No son distribuidos y están diseñados para ejecutarse en un ordenador personal.
- Sistemas Empotrados: Sistemas informáticos (*hardware + software*), usualmente de tiempo real, integrados en un sistema de ingeniería más general, en el que realizan funciones de control y procesamiento.
- Sistemas Distribuidos: El sistema software en el que el procesamiento de información se distribuye sobre varias computadoras en vez de estar confinado en una única máquina.
    - Arquitecturas Multi-procesador: Sistema con diferentes procesos (*diferentes procesadores o no*).
    - Arquitecturas de objetos distribuidos: conjunto de objetos que interaccionan y cuya localización es irrelevante. *No hay distinción entre un proveedor de servicios y el usuario de estos servicio*
    - Arquitecturas Cliente/Servidor: Conjunto de servicios que se proporcionan a los clientes por parte de los servidores. Los *servidores y los clientes se tratan de forma diferente*. Estos interactúan por paso de mensajes. El servidor suele estar compuesto de varias máquinas para cada tarea.

Cliente Servidor Con 3 y 4-niveles

Arquitectura Multicapa

Capa (layer): Hace referencia a una segmentación lógica de la solución.

Nivel (tier): Se refiere a la segmentación o ubicación física.

Un sistema por capas es un conjunto ordenado de subsistemas, cada uno de los cuales está construido en términos de los que tiene por debajo, y proporciona la base de la implementación de aquellos que estén por encima de él.

Dependencia entre capas: Los objetivos de cada capa se recomienda que sean independientes, aunque entre las capas inferiores y superiores Existe una relación cliente/servidor: *Las inferiores proporcionan unos servicios (básicos de la arquitectura) y las superiores consumen dichos servicios.*

Visibilidad de las capas

Las arquitecturas basadas en capas pueden ser abiertas o cerradas según la dependencia existente entre capas

- Abiertas: Una capa puede utilizar características de cualquier capa a cualquier nivel.
- Cerradas: Una capa sólo utiliza características de su capa inmediatamente inferior.

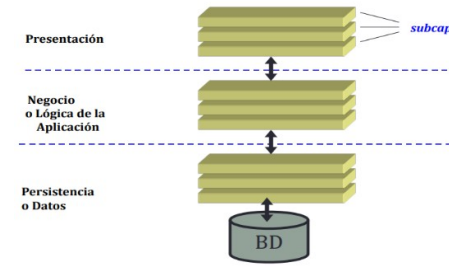
Se recomienda trabajar con arquitecturas cerradas, ya que reducen las dependencias entre capas y permiten que los cambios se hagan con facilidad porque de una de ellas sólo afecta a la capa siguiente.

Ejemplo: *Arquitectura de 3 Capas*

- Presentación: Presentación de los resultados de computación al usuario y recogida de entradas del usuario al sistema.
- Lógica: Proporcionar la funcionalidad de la aplicación
- Datos: Proporcionar persistencia a los datos, a través de bases de datos, ficheros…

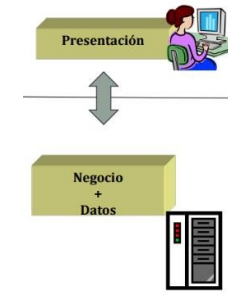
Ventajas

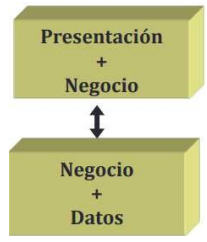
- Aislar la lógica de la aplicación en componentes separados.
- Distribución de capas en diferentes máquinas o procesos (*Cliente /Servidor y diferenciación entre niveles y capas*)
- Posibilidad de desarrollo en paralelo.
- Dedicación de recursos a cada una de las capas.
- REUTILIZACIÓN.



Ejemplo: *Arquitectura de 2 Capas*

- Clientes Ligeros (Thin clients): Son útiles para "Sistemas legados/heredado" (*sistemas anticuados pero que se siguen usando*) donde no se puede separa el procesamiento y la gestión de datos. *Nos dan aplicaciones de manejo de datos (consultas/navegación por una BD) con poco procesamiento.*
- Clientes Gruesos (Fat clients): En esta arquitectura la Parte de la lógica (*validaciones, reglas de negocio*) se pasa al cliente





TEMA 4

Modelado orientado a objetos con UML

UNDEFINED MODELING LANGUAGE: *UML*

Se entiendo Modelo como una simplificación de la realidad. *Esto no ayuda a crear y estudiar cosas.* Se entienden diferentes visiones sobre los sistemas y también tipos de diagramas que nos permitan representar estas realidades: Diagrama de Clase (*incluyendo Diagrama de Objetos*), Diagrama de Casos de Uso, Diagramas de Comportamiento, Diagramas de implementación…

Visiones de un Sistema Software 00

Visión estática :

- Objetos: Representan entidades que existen en el mundo real. Tiene identidad, una estructura y un estado.
- Clases: Describe un conjunto de objetos con las mismas propiedades y un comportamiento común las cuales se pueden relacionar entre sí (*relaciones entre clases).*

Visión dinámica :

- Objetos: Se comunican mediante la invocación de métodos de otros objetos.
- Se describen aspectos de un sistema que cambian con el tiempo: *Interacciones entre objetos, sus estados y las transiciones entre esos, eventos, operaciones/métodos que se ejecutan…*

DIAGRAMA DE CLASES

Muestra la estructura estática del sistema, mostrando las clases y las relaciones entre ellas. *Es la herramienta principal de la mayor parte de los métodos OO*. SE TIENE QUE REPRESENTAR CON UNA CAJITA CON ATRIBUTOS.

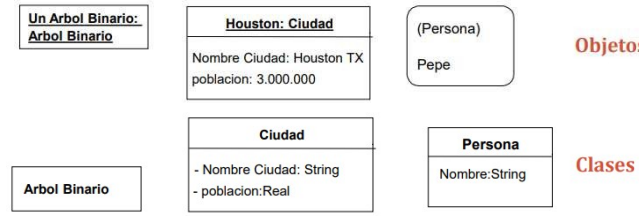
Clase

Son la descripción de un grupo de objetos con estructura, comportamiento y relaciones similares. Tienen atributos que Representan propiedades del objeto, *pero NO representan relaciones ni nada con otros objetos*, *Estas referencias se representan mediante enlaces*.

Los atributos/operaciones pueden ser: (-) *Privados*, (#) *Protegidos* y (+) *Públicos*.

Atributos derivados: Un atributo derivado se representa como $/\boldsymbol{Atributo} : \boldsymbol{Tipo}$.

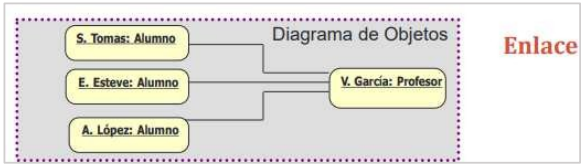
Métodos: Son la implementación de una operación que se realiza sobre o por el objeto de la clase.



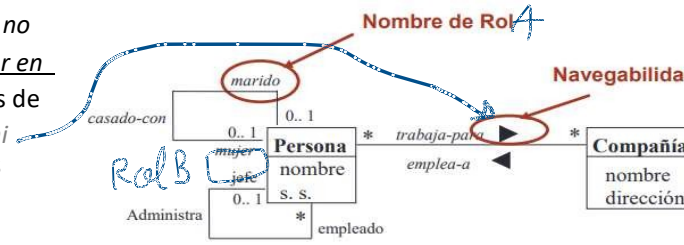
Asociaciones/Enlaces

Un enlace es una conexión física o conceptual entre objetos. Una asociación es una relación estructural que especifica que los objetos de un elemento están conectados con los objetos de otro. Hay que darle nombres, si tengo un objeto "coche" pues un objeto de "persona" es su "propietario": *Esa relación tiene que estar representada*.

Después en la implementación a lo mejor solo es que en coche hay un atributo persona y viceversa, pero aquí es así



Bidireccional: Toda asociación es bidireccional *(a no ser que digan lo contrario)*, es decir, *se puede navegar en los dos sentidos*. Desde objetos de una clase a objetos de la otra (*pero NO hace falta que pongamos la flecha ni especifiquemos las 2 direcciones, no tenemos por qué restringir la navegabilidad ni nada*).



- Nombre: Cada objeto tiene un nombre, y estas relaciones también lo pueden tener, además de cardinalidad.

Multiplicidad/cardinalidad: "1", "0..1", "0..N $(*)$", "1..N", "M..N".Es como lo de 1 a muchos, muchos a muchos 1 a 1… Pero aquí cada cantidad representa algo: *0 es que puede o no haber, min 1 es que Sí o Sí hay…*

- 0..1: Opcionalidad, en el *constructor NO* se pone y es solo 1 atributo/variable normal.
- 1..1: Obligatoriedad, en el *constructor Sí* se pone y es solo 1 atributo/variable normal.
- 0..*: Conectan con muchos, en el *constructor NO* se pone y es una colección de esos objetos.
- 1..*: Conectan con muchos, en el *constructor Sí* se pone y es una colección de esos objetos. Exige al menos 1, *por lo que En el constructor se pasa solo uno de esos elementos y se hace ".add(…)".*

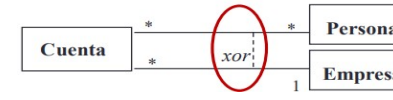
Atributos de Enlace

En una asociación entre clases, la propia relación puede tener propiedades, denominados atributos de enlace Ej: *Cada alumno tiene una nota para cada examen, si le pones el atributo a examen no vale xq es el mismo para todos los alumnos, si se lo pones a alumno no especificas el examen --> nueva clase con todo*.

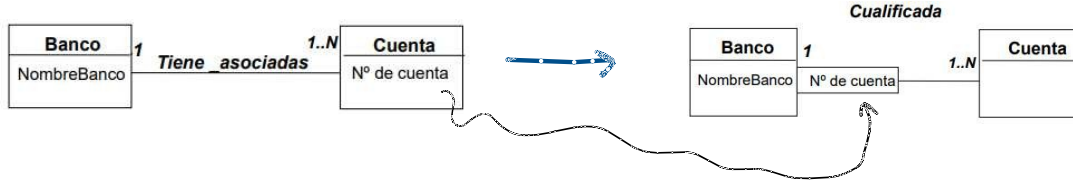
Se Hace una clase a parte (*clase asociación*) que tendrá un atributo de cada uno de los que conecta. Estos conectados, dependiendo si es muchos a muchos, 1 a 1… Tendrán una colección o un atributo de esta nueva clase.

Atributos excluyentes y calificada

Asociación excluyente: Se refiera a una asociación de un objeto con otros 2 de manera que solo puede tener o uno u otro (per eso se excluye). H*ace una XOR.*



Asociaciones cualificadas: Los Cualificadores o calificadores nos sirven para refinar más el modelo, indicando el índice para recorrer la relación.

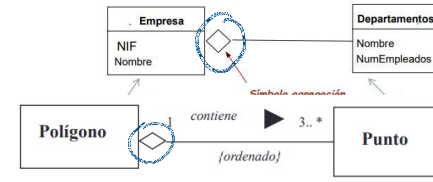


Agregación

Es una asociación con unas propiedades semánticas adicionales. Significa "está formado por". Hay varios tipos

Inclusiva o física: Cada componente puede pertenecer a lo sumo a un compuesto. *La destrucción del compuesto implica la destrucción de las partes*.

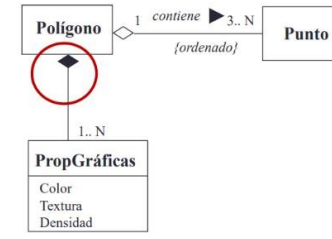
Referencial o de catálogo: los componentes son reutilizables a lo largo de distintos compuestos. *No están relacionados los tiempos de vida*.



- *Una empresa está formada por departamentos*:
- *Un polígono está formado por puntos, un polígono puede enumerar sus puntos, pero no hace falta que el punto diga a quien pertenece (la flecha)*:

Composición

Es una agregación inclusiva o física que significa "está compuesta por". *Es una conexión más fuerte y exclusiva que la Agregación.* No es solo que tenga algo, sino que está compuesto por ese mismo algo. *Si se destruye el objeto también sus partes.*



- *Un polígono está compuesto/tiene sus proporciones Gráficas.*
- *Un coche y sus partes: Este está compuesto por un motor, ruedas…*

Diferencia con Agregación: En una relación de agregación, las partes pueden existir de forma independiente, mientras que en la composición, las partes son completamente dependientes del objeto y no pueden existir sin él.

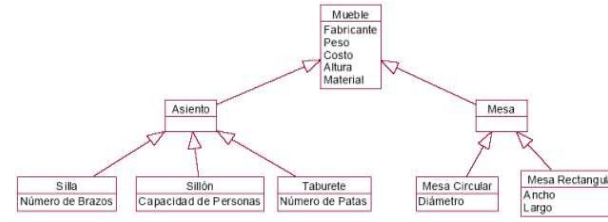
Especialización / Generalización: *Como herencia*

Herencia: Permiten definir jerarquías de clases. *Los atributos que desciendan tienen lo mimos que el padre + algo.*

Generalización: Dado un conjunto de clases, si tienen en común atributos y métodos, *se puede crear una clase más general (superclase) a partir de iniciales (subclases).*

Especialización: Es la relación inversa que representa Una relación de "Es un".

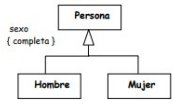
- Una silla es una asiento (tendrá sus características) que a su vez es un mueble (igual).



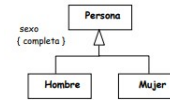
Diferencia Herencia Especialización: La relación de especialización se emplea en la fase de modelado de un sistema, mientras que la relación de herencia se ve como un mecanismo de reutilización de código en la fase de implementación o diseño.

Tipos de restricciones

Completa: Todos los hijos de la generalización se han especificado en el modelo. *Todas las posibles opciones.*



Incompleta: No se han especificado todos los hijos y se permiten hijos adicionales, *por ejemplo, un naranjo entra en árbol pero no en una especificación.*

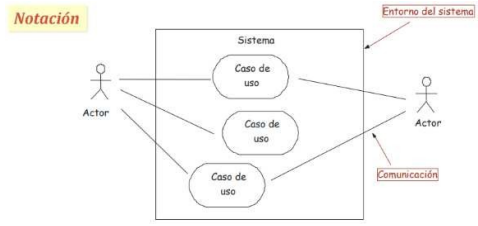


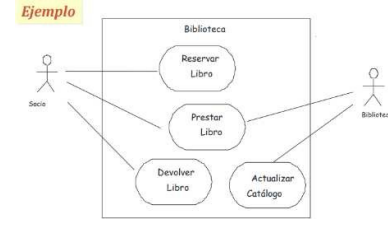
FLECHA: Es como un extends, ROMBO: Tiene/Implementa/Usa/Está Compuesto o Formado POR

CASOS DE USO

Es una técnica para capturar información de cómo un sistema trabaja actualmente, *o de cómo se desea que trabaje*. Se utiliza para capturar los requisitos funcionales del sistema a desarrollar

- **Actor**: Entidad (*Humana, Dispositivo, Otro Sistemas Software*) que intercambia información con el sistema.
- **Caso de Uso**: Contiene una secuencia de transacciones que intercambian los actores y el sistema cuando se desea ejecutar cierta funcionalidad del mismo.



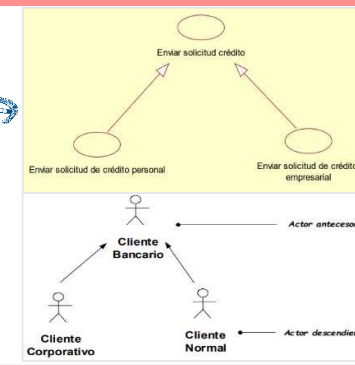


Herencia: *Relaciones entre Actores y Casos de uso*

Casos de uso: Un caso de uso B especializa a un caso de uso A, si el flujo de eventos de B es un refinamiento del flujo de eventos asociado a A.

Similar a la herencia OO que permite separar un patrón de interacción genérico (*caso padre*) de un patrón de interacción más específico (*caso descendiente*).

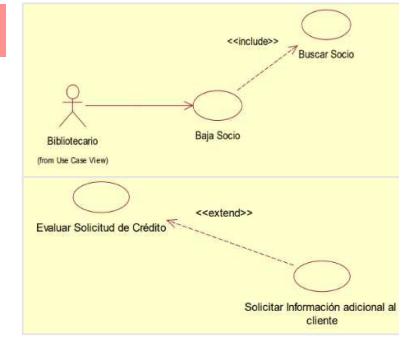
Actores: La relación de herencia entre actores indica que el actor descendiente puede jugar todos los roles del actor antecesor.

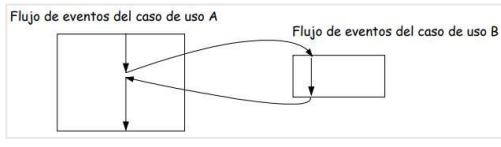


Inclusión y Extensión: *Relaciones entre Casos de Uso*

Inclusión: Un caso de uso A incluye a un caso de uso B, si una instancia de A puede realizar todos los eventos que aparecen descritos en B: *La instanciación de Baja Socio utiliza siempre el flujo de eventos de Buscar Socio.*

Extensión: Un caso de uso B extiende a un caso de uso A, si en la descripción de A figura una condición cuyo cumplimiento origina la ejecución de todos los eventos que aparecen descritos en B. *Igual que inclusión, pero con condición.*





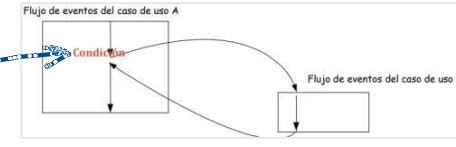
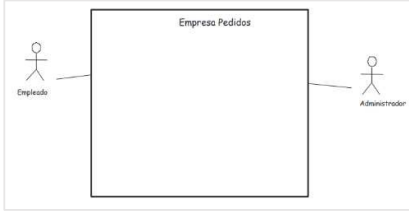
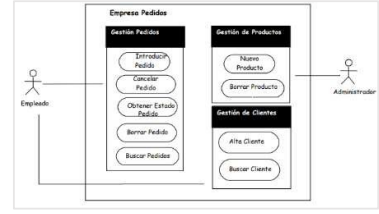
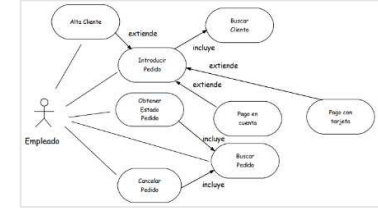


Diagrama de Casos de Uso: *3 Tapas*

1. **Diagrama de contexto y Diagrama inicial**: Muestra límites del sistema y actores que interactuarán con él.
2. **Plantillas de descripción**: Contiene la agrupación jerárquica de los distintos casos de uso
3. **Diagrama estructurado o Modelo de Casos de Uso**: Se muestra un/unos actores, los casos de uso a los que pueden acceder y las relaciones de extensión e inclusión entre estos.





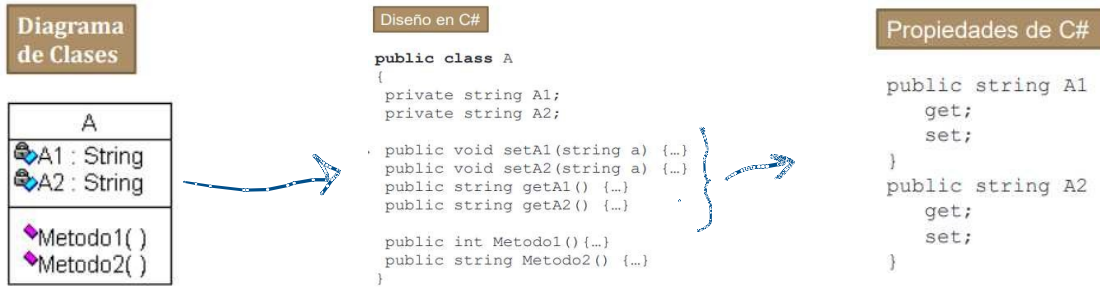


Tema 5

Diseño de la lógica de la aplicación

DISEÑO DE OBJETOS

Clases usando propiedades C#



Asociaciones *sin más atributos*

Cuando un objeto está asociado / relacionado / tiene un objeto de otro tipo, hay que representarlo. La forma es con atributos de ese tipo de objeto.

Asociaciones Uno – Uno (*min 0 o 1 pero MAX 1*)


```
public class A
{
   public B Rb {
        get;
        set;
   }
}
```

```
public class B
{
   public A Ra {
        get;
        set;
   }
}
```

A tiene un objeto tipo B y B tiene un objeto tipo A

Asociaciones Uno – Muchos (*min 0 o 1 pero MAX muchos*)

Si tiene más de uno tendrá una colección del tipo que toque. Además de poner métodos para añadir y eliminar objetos.

```
public class B
{
   private ICollection<A> Ra;
   public void AddA(A a){
        Ra.Add(a);
   }
   public void RemoveA (A a){
        Ra.Remove(a);
   }
   public A GetA(object idA){
        foreach (A a in Ra) if (a.Id == id) return a;
        return null;
   }
   public void RemoveA(object idA){
        RemoveA(GetA(idA));
   }
}
```

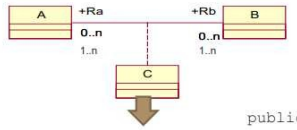
```
public class A
{
 public B Rb {// asociación uno-uno
        get;
        set;
   }

public class B
{
   public ICollection<A> Ra {
        get;
        set;
   }
}
```

Si es muchos a muchos, pues los 2 objetos tienen que tener una colección del tipo del otro.

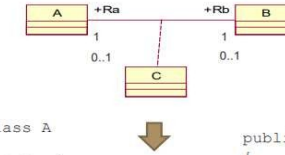
Asociaciones *con + atributos* : Hay que crear clases que tengan: Un objeto de cada tipo y los atributos extra.

1 a 1




```
public class A
{
  public ICollection<C> Rc {
        get;
        set;
        }
}
public class B
{
 public ICollection<C> Rc {
        get;
        set;
        }
```

```
public class C
{
   public A Ra {
        get;
        set;
        }
   public B Rb {
        get;
        set;
        }
}
```



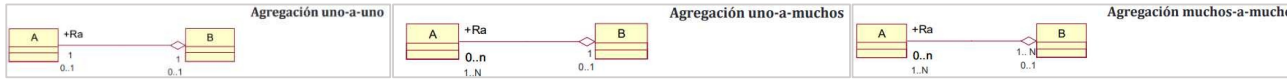
Muchos a Muchos

```
public class A
{
  public C Rc {
        get;
        set;
        }
}
public class B
{
  public C Rc {
        get;
        set;
        }
```

```
public class C
{
   public A Ra {
        get;
        set;
        }
   public B Rb {
        get;
        set;
        }
}
```

Agregación / Composición

Sigue las mismas pautas que las asociaciones, *si es max uno, poner solo una variable, si es a muchos colección…*



Especialización/Generalización : Para la herencia si B hereda de A se pone `public class B : A{…}` y en el constructor se le han de dar LOS MIMOS VALORES QUE a "A" + los propios y pasárselos con :base(arg1)

```
Public B(arg1, arg2...,extra1,extra2...) : base(arg1. arg2....) {this.extra1 = extra1;…}
```

DISEÑO DE CONSTRUCTORES

Inicializar un objeto supone dar valores tanto a sus atributos como a los enlaces con objetos de otras clases, si los hubiere. *La multiplicidad mínima de las asociaciones/agregaciones determina cómo se realiza la inicialización.*

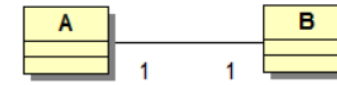
Al menos 1 : Se tienen que pasar en el constructor ese valor o al menos uno de ellos

Al menos 0 : NO hace falta pasarlo, pero *se tiene que inicializar la colección igualmente en caso de que tenga*.

x	y	Declaración en A	Constructor de A
0	1	`public B Rb { get; set; }`	`public A(…){…};`
1	1		`public A(…, B b, …) { this.Rb = b; … }`
0	N	`public ICollection<B> Rb { get; set; }`	`public A(…){ Rb=new List<B>; … }`
1	N		`public A(…, B b, …){ Rb = new List<B>; Rb.Add(b); … }`

Constructores en relaciones uno-uno

Cuando en ambos extremos de una asociación, la multiplicidad mínima es 1, se crea una dependencia circular que no puede resolverse en un paso. Debe implementarse una inicialización en varios pasos: *Relajar un constructor para luego darle al objeto el que le hace falta → Primero se hace un objeto A sin pasarle B, se crea B pasando A y luego le paso a A el B.*



Tenemos que garantizar por código que la restricción se cumple. Ejecutar como un todo

Se pondría en un método para la inicialización.

O se pondría en el B la línea de "A.setEl_B(this)" y se le pase a si mismo.

```
public class A {
 B el_B;
 public A(...)
 {
 ...
 }
 ...
 }
```

```
public class B {
 A el_A;
 public B(... A el_A)
 {
   this.el_A=el_A;
 }
 ...
 }
```

```
...
//se debe ejecutar como un todo
A un_A=new A(...);          // un A
B un_B = new B(un_A);       // un B
un_A.setEl_B(un_B);         // 1..1
...
```

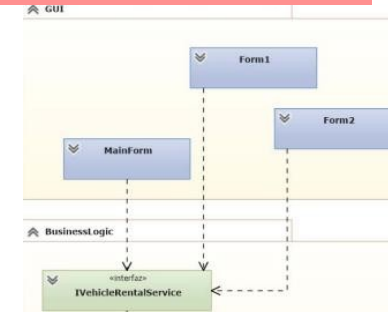
En el examen poner mal los constructores, luego comentar y tachar el atributo en uno de los constructores (*que se sepa que lo hago aposta*) Pero aun que te digan (no implementes los constructores) pones la línea extra A.set(B/this).

DISEÑO ARQUITECTÓNICO

Diseño separado en capas: Presentación (IGU), Lógica de negocio, Persistencia para acceso a la fuente de datos.

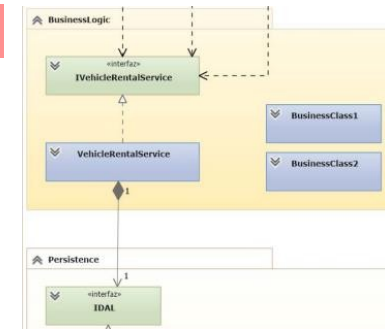
Presentación

Son un conjunto de formularios (uno de ellos el *MainForm*). Todos los formularios accederán a los servicios que ofrece la lógica de negocio (*en el ejemplo, vía VehicleRentalService*). Por lo tanto, el constructor de todos los formularios necesita una referencia a VehicleRentalService.



Reutilización : Definimos una interfaz *IVehicleRentalService* que indica el qué, no el cómo. Así, se podrán adoptar distintas implementaciones y la capa de presentación no se verá afectada. *Cada formulario cumple los mínimos a su manera.*

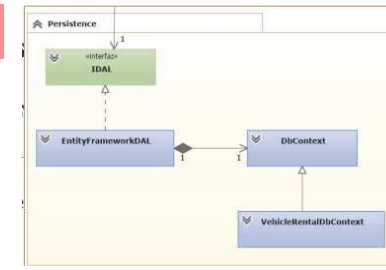
Lógica de negocio / Funcionalidad



Proporciona todos los servicios de nuestra aplicación (*casos de uso*). Estos servicios son identificados en una interfaz (*en el ejemplo IVehicleRentalService*). Se pueden proporcionar distintas implementaciones de los servicios de esa interfaz.

Interacción con el resto de clases : Estas clases trabajarán con el resto de las clases de la lógica: Cada implementación puede trabajar con una capa de acceso a datos distinta (*DAL, Data Access Layer*), modelada como interfaz.

Persistencia



Proporciona el acceso a la fuente de datos (*BD relacional, BDOO, archivo de texto, archivo XML, etc.*). El acceso a datos se identifica mediante una interfaz (*en el ejemplo IDAL*) la cual también se pueden dar diferentes implementaciones.

Tema 6

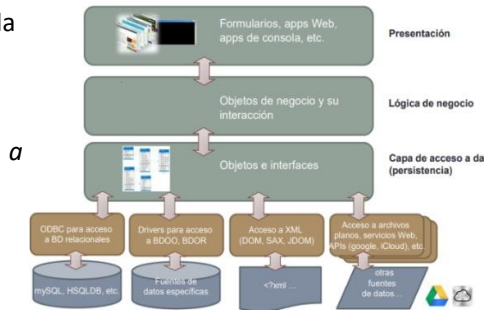
Diseño de la Persistencia

INTRODUCCIÓN

En la mayoría de las aplicaciones el almacenamiento no volátil de la información es esencial. Por lo que tenemos que guardar información:
Ya sea con XML, JSON, BD relacionales u Orientadas a Objetos.

Esta capa está entre la lógica de la aplicación y la capa de acceso a datos.

Las capas de abajo proporcionan Servicios a las de arriba: Implemtamos una serie de clases que nos facilitenel intercambio de información



PATRÓN DE ACCESO A DATOS: *DAO*

Los patrones Van a servir de puente entra la base de datos y la capa de lógica. Comunicar lógica de negocio con la base de datos ocultando las fuentes en las que se guarda toda la información.

Yo le digo: inserta esto o lo otro y lo hace, y a mí me da igual donde y como lo hace.

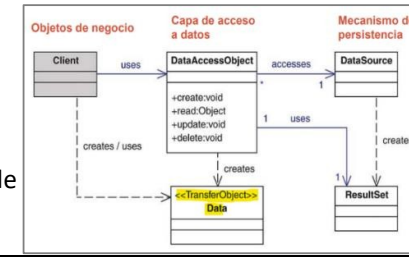
Estructura del patrón DAO

Se usa una capa para gestionar las operaciones de la información que ha de ser persistente.

BusinessObject: Objeto de la capa de negocio que necesita acceder a la fuente de datos, para leer o almacenar.

DataAccessObject (*DAO*): Abstrae la implementación subyacente del acceso a datos a la capa de negocios para conseguir un acceso transparente a la fuente de datos. *El BusinessObject delega en el DAO las operaciones de lectura y escritura de datos. SON LOS SERVICIOS*.

Deben tener métodos para añadir y recuperar los objetos a ser persistido.



DataTransferObject (*DTO*): Representa el objeto portador de los datos.
DAO puede devolver los datos al BusinessObject en un DTO. El DAO puede recibir los datos para actualizar la BD en un DTO.

ESTOS OBJETOS tienen los atributos/información necesaria para poder crear objetos con la info de la base de datos y como añadir cosas en la base de datos a partir de objetos.

DataSource: Implementación de la fuente de datos (SGBDR, SGBDOO, SGBDOR, repositorio XML, etc.)

HACE FALTA UNA CLASES DAO Y OTRA DTO PARA CADA UNA DE LAS CLASES QUE HAYA QUE PERSISTIR

DTO y DAO: La base de datos, necesita saber qué hacer con toda la información que tiene. Cuando crees un objeto de coche, tendrás que pasarle todos sus atributos (matrícula, color…). Cuando crees un objeto propietario, tienes que pasarle todos sus atributos + los necesarios para identificar los objetos que tienen. *Por ejemplo, el propietario tiene un coche, por lo que hay que pasarle algo para identificar al coche (la Matricula por ejemplo).*

Y VICEBERSA. Cuando del objeto quiera rellenar la base de datos, tengo que pasarle los atributos + lo necesario para identificar los atributos que sean clave ajena (*para identificar los objetos a los que esté relacionado*).

Ventajas del Patrón DAO

Encapsulación: Los objetos de la capa de negocio no conocen detalles específicos del acceso a datos.
Migración más fácil: Migrar el sistema a un gestor de datos diferente es cambiar la capa de DAO por otra.
Menor complejidad: Menor complejidad en la capa de negocio al aislarse del acceso a datos.
Centralizado: Centraliza el punto de acceso a datos, la manera en la que accedemos a ellos (librerías y clases).

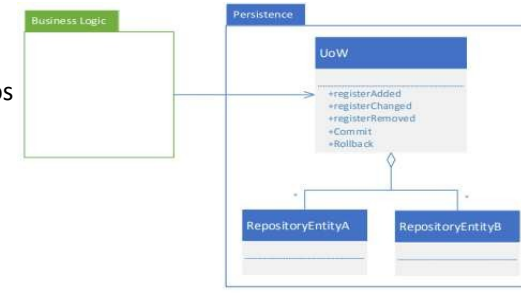
Inconvenientes del patrón DAO

Arquitectura compleja: Arquitectura software ligeramente más compleja.
Más código: Hay que implementar más código para ofrecer ese nivel de indirección adicional.
Menos eficiente: Desde el punto de vista de la eficiencia se puede ralentizar el proceso.

PATRÓN REPOSITORIO + UNIDAD DE TRABAJO

Es el mejor de todos. Ofrece una abstracción para acceder a los datos como si fueran colecciones en memoria. Interfaz con métodos para añadir, borrar, actualizar y buscar objetos del dominio de forma directa, sin necesidad de crear objetos de transferencia (*fuck DTO*)

Va a ofrecer lo mismo que el patrón DAO pero sin implementar los DTOs. Es un puente entre la capa de dominio y la de persistencia (*una BDOR*). Ofrece una interfaz tipo colección para acceder a los objetos de dominio directamente (sin DTOs).



Se hace mediante una interfaz de tipo colección para acceder a la base de datos. El repositorio nos crea métodos para añadir/trabajar a/con colecciones.

Unidad de Trabajo *UoW* (Unit of Work) Mantiene un registro de los objetos afectados durante una transacción (*operación atómica sobre objetos de negocio*) y se encarga de persistir los cambios (*o deshacerlos en caso de abortar la transacción*) y resolver cualquier problema de concurrencia que pudiera surgir

Esta unidad se encarga de ver que objetos han sido modificados/afectados guardando un registro de todos ellos. De manera que cuando haga falta las "persiste" (las guarda) o no según se le indique.

Un repositorio por objeto del dominio

La mejor manera de actuar sobre todo cuando se implanta en cosas viejas, es crear un repositorio por cada objeto del dominio a persistir. Solo se necesita implementar en cada repositorio los métodos que se vayan a utilizar y nada más (*no hace falta hacer recuperar si no lo vas a usar por ej)*

Lo mejor si se va a hacer algo nuevo es hacer un repositorio genérico para todos los objetos.

Repositorio genérico Para todos los objetos *Actualidad*

```
public interface IRepository
{
  void Add<T>(T entity) where T : class;
  void Delete<T>(T entity) where T : class;
  T GetById<T>(IComparable id) where T : class;
  bool Exists<T>(IComparable id) where T : class;
  IEnumerable<T> GetAll<T>() where T : class;
}
```

Se necesita una sola interfaz y una única implementación por cada mecanismo de persistencia que queramos utilizar (y no una implementación por cada clase)

Se hace una interfaz genérica con operaciones sobre colecciones. Y para cada tipo se desarrolla la clase en concreto que haga falta: *EL TIPO MARCA A QUÉ Y SOBRE QUÉ COLECCIÓN SE ACTUA. No hace falta un DAO de cada tipo.*

Esto es posible xq el repositorio junto con la unidad de trabajo se encargan de que los objetos genéricos estos se guarden donde toca. Estos evitan que tengas que crear un DAO por cada uno.

Repositorio vs DAO

El concepto de DAO está más cercano al mecanismo de persistencia: Es un enfoque centrado en los datos. Por eso normalmente se crea un DAO por cada tabla o vista de una base de datos.

El concepto de repositorio se encuentra más cercano a la capa de negocio: Trabaja directamente con los objetos de negocio. Internamente, un repositorio podría usar BDOR, pero también mecanismos de bajo nivel como DAO/DTO

Patrón Unidad de Trabajo

Actualizar la información en la base de datos cada vez que se modifica algo Es poco eficiente, haría muchos cambios tontos y si hay un error a la mierda todo. Da muchos problemas, consistencia, concurrencia…

Una Unidad de Trabajo mantiene un registro de cambios. No realiza los cambios en el mecanismo de persistencia hasta que no se complete la transacción, y si hay que deshacer algo se soluciona en memoria: *Es más eficiente y evita problemas de consistencia en el acceso concurrente*. Además de normal hay frameworks que ya lo implementan.

Clase DbContext en Entity Framework hace el trabajo de pasar de uno a otro (*junta repositorio y unidad de trabajo).*

La unidad de trabajo ES EL QUE HACE LA TRADUCCIÓN DE UN OBJETO A LA TABLA DE LA BD Y VICEVERSA

Objetos en memora → Filas BD. Filas BD → Objetos en memoria

PERSISTENCIA EN BDOR Y BDOO

En vez de una BD relacional se proporciona una BDOO (no relacional). Un mecanismo que persiste los mismos objetos y como están relacionados. *Ahora todos los patrones no hacen falta xq como tal tengo los objetos, por lo que la traducción es directa, se puede guardar y recuperar en memoria directamente*.

Las operaciones se realizan de forma más eficiente. No hace falta manejar distintas tablas ni nada.

Ventajas: Se une la potencia de los objetos con la flexibilidad de SQL. Simplifica el desarrollo de aplicaciones en capas xq no hace falta hacer ni las consultas sql ni saber usar varios modelos, con saber usar uno el otro lo maneja él.

Esto en pocas líneas xq se lo ha saltado como un crack el teacher (*Voy a arrancarme los huevos*).

Resumen ¿? No sé ya que es nada de esto

Los patrones de acceso a datos (DAO/Repository) permiten abstraer el acceso a la capa de persistencia de su implementación.

Es posible aplicar una sencilla correspondencia para derivar un modelo relacional a partir de un modelo OO.

Las BD objetuales simplifican el desarrollo de aplicaciones porque:

- El modelo de datos se puede proyectar (hacia o desde la aplicación) sin necesidad de establecer correspondencias
- Las operaciones son simples operaciones sobre objetos
- En general, también son más simples de usar y más eficientes