AESM – RESUMEN

TEMA 2.1 – ANÁLISIS Y CAPTACIÓN DE REQUISITOS

El objetivo de los requisitos es servir de guía para convertir los conceptos en valor. Los **requisitos** de un sistema son la descripción de los servicios proporcionados por el sistema y sus restricciones operativas. El proceso de descubrir, analizar, documentar y verificar estos servicios y restricciones se denomina Ingeniería de Requisitos.

Las **características de los requisitos** son los siguientes:

1. **Volátiles**: pueden ser inconstantes.
2. **Mutantes**: presentan alteraciones que se transmiten a otros requisitos.
3. **Emergentes**: surgen al analizar el sistema en profundidad.
4. **Colaterales**: surgen al incluirse otros requisitos.
5. **Por compatibilidad**: se añaden para adaptar el sistema a su entorno.

Los **requisitos cambian** porque las **necesidades** de **los clientes varían**, por cambios tecnológicos, porque el entorno evoluciona, por el mercado, porque el problema cambia…

Los **requisitos** pueden calificarse en las siguientes dimensiones:

1. **Ámbito**: indica en qué **contexto** se debe entender el requisito.
2. **Característica** que define: requisitos funcionales o no funcionales.
3. **Audiencia**: indica a **quién** está dirigido el requisito.
4. **Representación**: establece la **forma** de los requisitos. (formal, informal…)

Los **tipos de requisitos** que podemos encontrar son:

1. De **Producto**: son del **software** en sí a desarrollar.
2. De **Proceso**: son sobre la **manera** en que se desarrolla el software.
3. **Funcionales**: especifica las **funciones** que deben ser capaz de llevar a cabo.
4. **No funcionales**: especifica los **aspectos técnicos** que debe incluir el sistema. Por ejemplo, facilidad de uso, capacidad para recuperarse de fallos… Hay tres tipos:
    a. De **producto**: especifican el comportamiento del producto. (**portabilidad**, tiempo de respuesta, fiabilidad…)
    b. **Organizacionales**: Métodos de diseño, **estándares**, herramientas a usar…
    c. **Externos**: éticos, legislativos, **seguridad**, privacidad…

Para la **captación de requisitos** es importante seguir los siguientes pasos:

1. **Determinar** el **objetivo** y comprender el problema que requiere una solución sw.
2. **Comprender** el **contexto** en el cual se desarrollará el sw.
3. **Comprender** el **impacto** que el cliente espera.
4. **Comprender** la motivación y **expectativas** del cliente.

Por medio de **entrevistas**, **cuestionarios**, sesiones de grupo, brainstorming, prototipados…

REQUISITOS FUNCIONALES (CAPACIDADES)

- Son declaraciones de los servicios que debe proporcionar el sistema, la forma en que debe reaccionar a las entradas y cómo se debe comportar en situaciones particulares
- Especifica una **función** que un sistema o componente de un sistema debe ser capaz de llevar a cabo
- También pueden declarar explícitamente lo que el sistema no debe hacer
- Describen las interacciones entre el sistema y su entorno independientemente de la implementación.
- Especifican el comportamiento esperado del sistema.

Clasificación:

- Requisitos de usuario: declaraciones en lenguaje natural y en diagramas, de los servicios que se espera que el sistema proporcione y de las restricciones bajo las cuales debe funcionar. Suele ser más abstractos.
- Requisitos del sistema: establecen con detalle las funciones, servicios y restricciones operativas del sistema. Se debe definir exactamente qué se va a implementar. Añaden detalles y explican los servicios y funciones que el sistema debe proporcionar.

REQUISITOS NO FUNCIONALES (RESTRICCIONES)

- Son restricciones de los servicios o funciones ofrecidos por el sistema
- Especifican **aspectos técnicos** que debe incluir el sistema
- Incluyen restricciones de tiempo, sobre el proceso de desarrollo y estándares
- Normalmente se aplican al sistema en su totalidad
- Pueden estar relacionados con propiedades emergentes del sistema: *fiabilidad, tiempo de respuesta, capacidad de almacenamiento, escalabilidad,…*
- Suelen ser requisitos más críticos que los requisitos funcionales
- Pueden describir restricciones externas del sistema
- Los clientes o usuarios establecen requisitos no funcionales como metas generales, tales como: *facilidad de uso, capacidad para recuperarse de los fallos o la respuesta rápida al usuario*

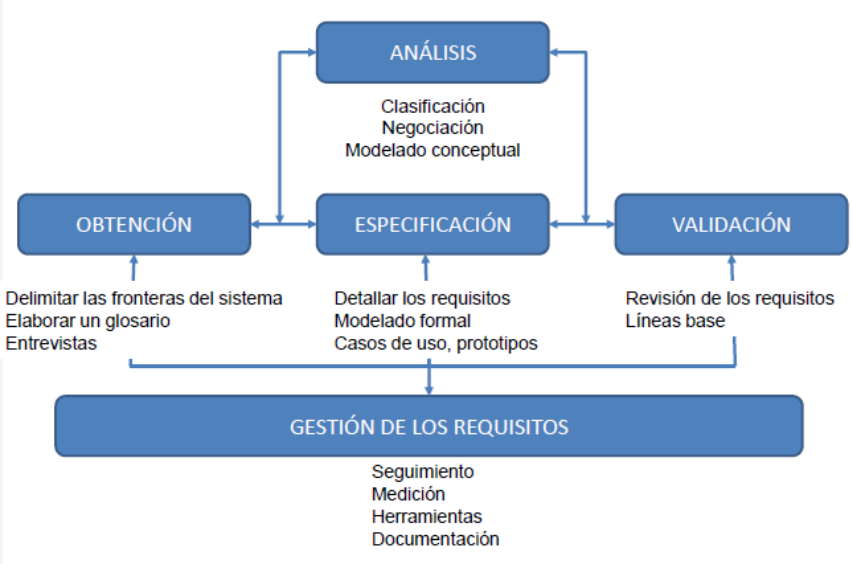
Clasificación:

- Requisitos de producto: especifican el comportamiento del producto. *Tiempo de respuesta, memoria requerida, portabilidad, usabilidad, fiabilidad, rendimiento…*
- Requisitos organizacionales: se derivan de las políticas y procedimiento existentes en la organización del cliente y en la del desarrollador. *Estándares de proceso, herramientas a usar, métodos de diseño, estándares de documentación, requisitos de entrega,…*
- Requisitos externos: factores externos al sistema y de su proceso de desarrollo. *Interoperabilidad con otros sistemas, éticos, legislativos, privacidad, seguridad,…*

TEMA 2.2 – ANÁLISIS Y CAPTACIÓN DE REQUISITOS

El **documento de definición** del sistema establece los **requisitos de alto nivel**, dirigiéndose a los clientes. Detalla los requisitos generales e incluye la lista de participantes, la definición de los conceptos, los requisitos funcionales y no funcionales…
El **documento de especificación** de requisitos (SRS) **define** con detalle todo **lo que el sw debe realizar** y lo que no debe realizar. Debe ser completa, no ambigua, fácil de verificar, consistente, fácil de modificar y trazable.

Las distintas fases para la captación de Requisitos son las siguientes:



1. **Obtención de Requisitos**: El objetivo es determinar cuáles son los requisitos del sistema. A menudo se elabora un glosario de términos. En primer lugar se deben determinar las fuentes de información de las que se obtendrán los requisitos. En segundo lugar, deben establecerse las técnicas de requisitos a utilizar.
2. **Análisis de Requisitos**: El objetivo es delimitar los requisitos y definir exactamente cada uno de ellos. Detectar y resolver conflictos entre requisitos. Elaborar los requisitos del sistema para obtener los requisitos del software a desarrollar.
    a. **Clasificación de requisitos**: establecer un conjunto de categorías y situar cada requisito en ellas.
    b. **Modelado conceptual**: El objetivo es facilitar la comprensión de los requisitos mediante su representación en un lenguaje o notación que comprendan los que los van a tratar.
    c. **Situación de los requisitos en la arquitectura del sistema**: Se debe establecer qué elementos del sistema software van a satisfacer las demandas de los requisitos.
    d. **Negociación**
3. **Especificación de Requisitos:** Descripción completa de los requisitos del sistema a desarrollar. Los requisitos se plasman en un documento denominado Especificación de Requisitos de Software.
4. **Validación de Requisitos:** El objetivo es determinar si los documentos de requisitos definen el software que los usuarios esperan. Revisión de los requisitos, prototipado, validación del modelo y pruebas de aceptación.
5. **Gestión de Requisitos:** Implica la recolección, almacenamiento y mantenimiento de grandes cantidades de información.

DOCUMENTO DE ESPECIFICACIÓN DE REQUISITOS DE SOFTWARE (SRS):

- Define explícitamente todo aquello que el software debe realizar y sus restricciones
- Debe incluir todo aquello que el software no debería realizar.
- Debe ser no ambigua, completa, fácil de verificar, consistente, fácil de modificar, clasificada por orden de importancia o estabilidad, facilidad de traza.

TEMA 3.1 – INTRODUCCIÓN A UML

UML es un lenguaje de modelado visual, independiente del proceso de desarrollo. No se considera una metodología. Ofrece vocabulario y reglas para crear y leer modelos bien formados, y que constituyen los planos de un sistema software. En definitiva, el modelo UML de un sistema consiste en:

- Un conjunto de **elementos de modelado** que definen la estructura, el comportamiento y la funcionalidad del sistema y que se agrupan en una **base de datos única.**
- La presentación de esos conceptos a través de múltiples **diagramas** con el fin de introducirlos, editarlos, y hacerlos comprensibles.
- Los diagramas pueden agruparse en **vistas,** cada una enfocada a un aspecto particular del sistema
- La gestión de un modelo UML requiere una **herramienta específica** que mantenga la **consistencia** del modelo

Ventajas: Es estándar, está basado en un metamodelo con una semántica bien definida, se basa en la notación gráfica concisa y fácil de aprender y utilizar, es fácilmente extensible.

Un diagrama es la representación gráfica de un conjunto de elementos de modelado (parte de un modelo).

- **Diagramas Estáticos:** Diagramas de Casos de Uso, Diagrama de Clases, Diagrama de Objetos, Diagrama de Componentes, Diagramas de Implantación.
- **Diagramas Dinámicos:** Diagramas de Secuencia, Diagramas de Colaboración, Diagramas de Estados, Diagramas de Actividad.

Vistas Arquitecturales: Se requiere que el sistema sea visto desde varias perspectivas. Cada vista puede existir de forma independiente, pero interactúan entre sí.

1. **Vista de Casos de Uso**: captura la funcionalidad del sistema tal y como es percibido por los usuarios finales, analistas y encargados de pruebas. *Diagramas de casos de uso. Diagramas de Interacción, de Estados y de Actividades.*
2. **Vista de diseño:** Captura las clases, interfaces y colaboraciones que describen el sistema. *Diagramas de clases y de objetos. Diagramas de Interacción, de Estados y de Actividades.*
3. **Vista de interacción:** Captura el flujo de control entre las diversas partes del sistema, incluyendo los posibles mecanismos de concurrencia y sincronización. Abarca en especial requisitos no funcionales como rendimiento, escalabilidad y capacidad de procesamiento. *Diagramas de clases y de objetos. Diagramas de Interacción, de Estados y de Actividades.*
4. **Vista de Implementación:** Captura los artefactos que se utilizan para ensamblar y poner en producción el sistema de software real. Define la arquitectura física del sistema. *Diagrama de Componentes y de Estructura Compuesta. Diagramas de Interacción, de Estados y de Actividades.*
5. **Vista de Despliegue:** Captura las características de instalación y ejecución del sistema. Contiene los nodos y enlaces que forman la topología HW sobre la que se ejecuta el sistema SW. *Diagramas de despliegue. Diagramas de Interacción, de Estados y de Actividades.*

TEMA 3.2 – DIAGRAMA DE CASOS DE USO

Diagrama de Casos de Uso: Modelado del sistema desde el punto de vista de los usuarios. Permite definir los límites del sistema y las relaciones entre el sistema y el entorno. Describen qué hace el sistema pero no cómo lo hace.

Un Caso de Uso sirve para capturar el comportamiento deseado del sistema sin tener que especificar cómo se implementa y como medio de comprensión del sistema para desarrolladores, usuarios finales y expertos del dominio.

- Un caso de uso representa un **requisito funcional**.
- Un caso de uso es un comportamiento del sistema que produce un resultado de interés para algún actor.
- Son siempre iniciados por un actor.
- No describe únicamente una funcionalidad, sino también una interacción entre un actor y el sistema bajo la forma de un flujo de eventos.
- Se representan en UML como una elipse.
- Los actores están fuera del sistema y los casos de uso dentro del sistema.
- Hay un actor que inicia un caso de uso (situado a la izquierda del caso de uso) y otro que recibirá algo de valor (situado a la derecha del caso de uso).
- Los límites del sistema se representan mediante un rectángulo.

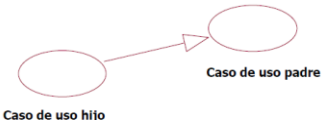
Los actores pueden ser una agrupación uniforme de personas, dispositivos HW u otros sistemas que interactúen con el sistema. Un usuario no es exactamente lo mismo que un actor, un usuario puede acceder al sistema como distintos actores (Un actor es una clase de rol, un usuario es una persona que cuando usa el sistema asume un tipo de rol). Otro sistema que interactúa con el que estamos construyendo también es un actor. Un actor y un caso de uso se pueden comunicar a través de una asociación en donde cada uno de ellos puede enviar y recibir mensajes

Identificación de los Casos de Uso:

- Un caso de uso es una descripción de un proceso de principio a fin relativamente amplia y suele abarcar muchos pasos o transacciones.
- Método para identificarlos basado en los actores: Se identifican los actores relacionados con un sistema y para cada uno de ellos, se identifican los procesos que inician o en los que participan.
- El nombre de un caso de uso se identifica con un verbo seguido del objeto o entidad afectado por el caso (*Realizar pedido, Obtener listado de clientes,…*).
- Están expresados desde el punto de vista del actor. Son iniciados por un único actor.
- Describen tanto lo que hace el actor como lo que hace el sistema cuando interactúa con él.
- Se documentan con texto informal. En general, se usa una lista numerada de los pasos que sigue el actor para interactuar con el sistema.
- La **regla general** es: "*una función del sistema es un caso de uso si se indica explícitamente al sistema que uno quiere acceder a esa función*". No debe seguirse al pie de la letra.
- Alternativas: errores o excepciones que aparecen durante la ejecución de un caso de uso
- Características: representan un error o excepción en el curso normal de un caso de uso. No tienen sentido por sí mismas fuera del contexto en el que ocurren.

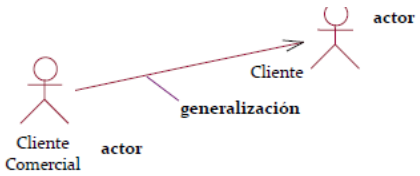
Tipos de Relaciones entre Casos de Uso: Muestran cómo se puede organizar una especificación que utiliza casos de uso para evitar redundancia y facilitar su comprensión.

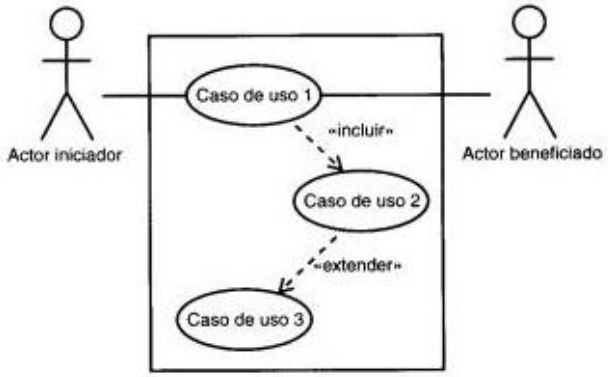
- **Generalización:** el caso hijo hereda el comportamiento y significado del caso de uso padre. El hijo puede añadir o redefinir el comportamiento del padre. El caso hijo hereda la especificación del padre.

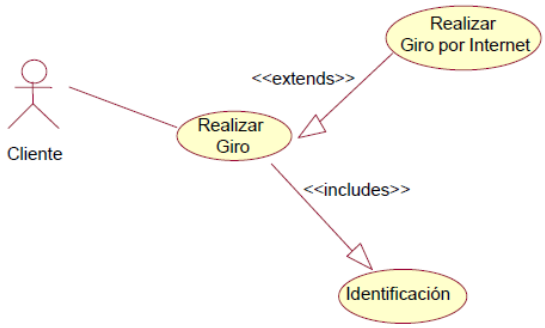


- **Inclusión:** Un caso de uso base incorpora explícitamente el comportamiento de otro caso de uso en el lugar especificado en el caso base. Se usa para evitar describir el mismo flujo de eventos repetidas veces, poniendo comportamiento común en un caso de uso aparte. Aparecen como funcionalidad común. El caso es usado siempre que el caso que lo usa es ejecutado. <<include>> o <<uses>> **(***La funcionalidad buscar pedido puede ser accedida desde: la toma de pedidos, las consultas,…*).

- **Extensión:** Son opcionales. Significa que a un caso de uso base (original) se le añade implícitamente el comportamiento de otro caso de uso. El nuevo caso de uso (opcional) extiende al original al agregar otros pasos a la secuencia del caso de uso original. Se usa cuando se tiene un caso de uso que es similar a otro, pero que hace un poco más. En la extensión, la funcionalidad de un caso de uso incluye un conjunto de pasos que ocurren solo en algunas ocasiones. No necesariamente provienen de un error o excepción. Regla aproximada: Si algo opcional debe ser expresado con más de un paso, seguramente es una extensión y no una alternativa.

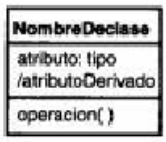






TEMA 3.3 – DIAGRAMA DE CLASES

El **diagrama de clases** captura el vocabulario del sistema. Se crea en las primeras fases del modelado y se va refinando a lo largo del proceso de desarrollo. Su principal propósito es: Nombrar y modelar conceptos del sistema, Especificar colaboraciones y Especificar esquemas lógicos de bases de datos.

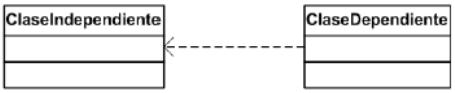


Una clase es una categoría o grupo de cosas que tienen atributos y acciones similares. Determinan el ámbito de definición de un conjunto de objetos. Cada clase se representa con un rectángulo con tres apartados: Nombre de la clase, Atributos de la clase y Operaciones de la clase.

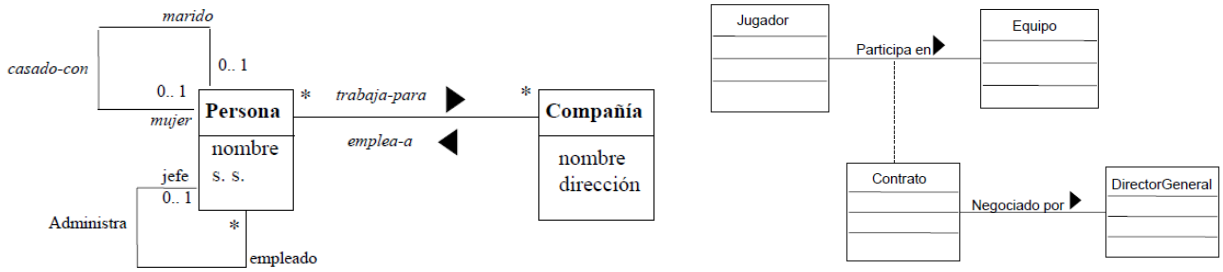
Un atributo es una propiedad característica de una clase. Describen un rango de valores que podrán tomar los objetos. Una operación es algo que la clase puede realizar o que otras clases pueden hacer con ella. Los atributos de una clase no deberían ser manipulables directamente por el resto de objetos (encapsulación).

Relaciones entre clases:

- Dependencia: Es una relación semántica entre dos elementos en la cual un cambio de un elemento (el elemento independiente) puede afectar a la semántica de otro elemento (elemento dependiente).

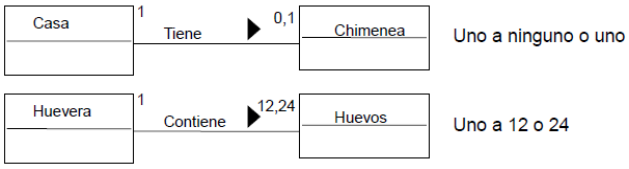


- Asociación: Expresa una conexión bidireccional entre objetos. Es una abstracción de la relación existente en los enlaces entre los objetos. Se visualiza con una línea que une a ambas clases con el nombre de la asociación junto a la línea, es útil indicar la dirección de la relación (con triángulo relleno que apunte a la dirección apropiada). Se puede caracterizar: "participa en", "emplea a", etc.

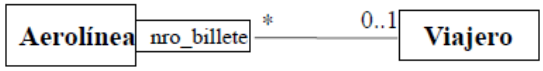


Una asociación puede tener atributos y operaciones, se concibe de la misma manera que una clase estándar. Se utiliza una línea discontinua para conectarla a la línea de asociación, solo puede existir una instancia de la asociación entre cualquier par de objetos participantes.

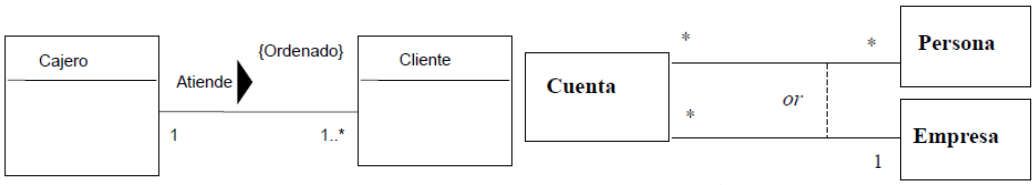
- Multiplicidad: Se define como la cantidad de objetos de una clase que se relacionan con un objeto de la clase asociada.
  - 1 Uno y solo uno
  - 0..1 Cero o uno
  - M..N Desde M hasta N (enteros naturales)
  - * Cero o muchos
  - 0..* Cero o muchos
  - 1..* Uno o muchos (al menos uno)



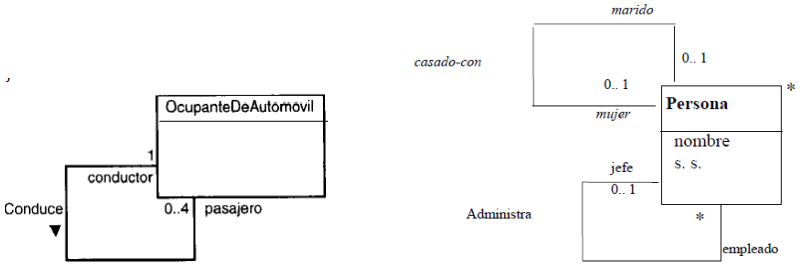
- Asociación Calificada: Si la asociación es de 1 a muchos. Si un objeto de una clase tiene que seleccionar un objeto particular de otro tipo para cumplir con un papel en la asociación. La primera clase debe tener a un atributo para localizar al objeto adecuado. (*Por ejemplo, cuando se realiza la reserva de un hotel, el hotel le asigna un número de confirmación*).



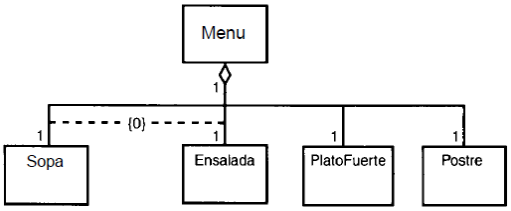
- Restricciones: Cuando una asociación entre dos clases debe seguir ciertas reglas necesita restricciones. Se indica junto a la línea de la asociación. Un tipo de restricción es la "Asociación Excluyente" conocida como {or}.



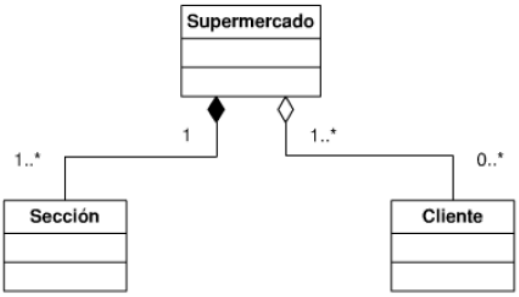
- Asociación Reflexiva: A veces, una clase es una asociación consigo misma. Ocurre cuando una clase puede jugar diversos papeles. (*Un ocupante de un automóvil puede ser Conductor: puede llevar ninguno o más ocupantes, o Pasajero*).



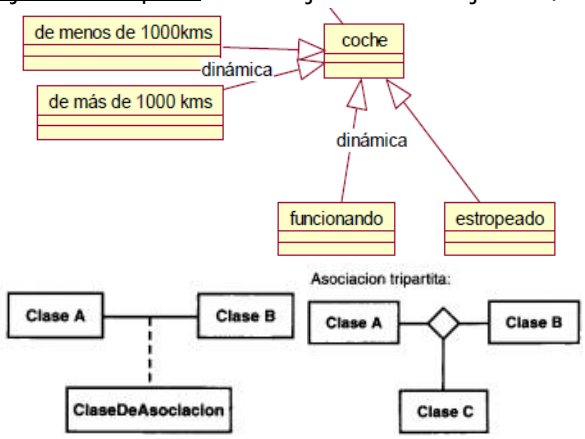
- Agregación: En ocasiones una clase consta de otras clases. Los componentes y la clase que constituyen son una asociación que conforma un todo. Si el objeto base desaparece no desparecen los objetos incluidos. Representa una relación **parte_de** entre objetos. Se utiliza un rombo sin relleno en la parte todo y se une con una línea a los componentes. A veces, el conjunto de componentes se establece dentro de una relación O.



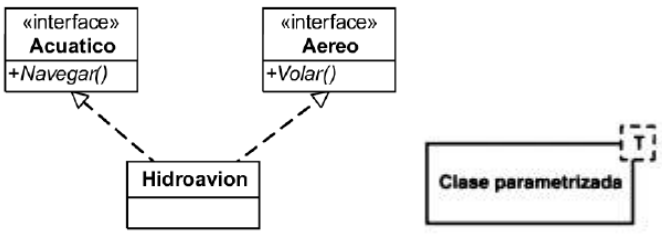
- Composición: Es un tipo representativo de una agregación. Cada componente dentro de una composición puede pertenecer solamente a un todo. El tiempo de vida del objeto incluido está condicionado por el tiempo de vida del que lo incluye. El objeto incluido solo existe mientras exista el objeto base. El objeto base se construye a partir de los objetos incluidos pero no podría existir sin ellos. Se representa con un rombo relleno.



- Generalización/ Especialización: Permiten gestionar la complejidad mediante un ordenamiento taxonómico. La generalización consiste en factorizar las propiedades comunes de un conjunto de clases en una clase más general. Las subclases heredan características de las superclases. Se puede particionar el espacio de objetos (clasificación estática) o se puede particionar el espacio de estados de los objetos (clasificación dinámica), en ambos casos se recomiendan generalizaciones disjuntas.
    - Generalización Total/ Parcial: Todos/no todos los objetos pertenecen a una clase.
    - Generalización Disjunta/ Solapada: Los conjuntos son disjuntos / no disjuntos.



- Herencia múltiple: Se presenta cuando una subclase tiene más de una superclase. Debe manejarse con precaución: problemas con conflicto de nombre y conflicto de precedencia.
- Herencia/ Polimorfismo: Se refiere a que una característica de una clase puede tomar varias otras. Representa la posibilidad de desencadenar operaciones distintas en respuesta a un mismo mensaje. Cada subclase hereda las operaciones pero tiene la posibilidad de modificar localmente el comportamiento de estas operaciones.
- Interfaces: Proporcionan un conjunto de operaciones que especifican cierto aspecto de la funcionalidad de una clase. Permite que clases que no están estrechamente relacionadas entre sí deban tener el mismo comportamiento. La implementación de una interfaz es un contrato que obliga a la clase a implementar todos los métodos definidos en la interfaz. Se declaran mediante las clase estereotipada <<interface>>



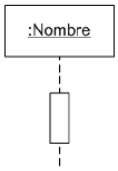
TEMA 3.4 – DIAGRAMAS DE INTERACCIÓN

Los **diagramas de interacción** proporcionan dos notaciones para un mismo objetivo: ilustrar el modo en que los objetos interaccionan por medio de mensajes. Permiten modelar la vista dinámica. Engloban dos tipos de diagramas: De secuencia y de Colaboración / Comunicación.

DIAGRAMAS DE SECUENCIA

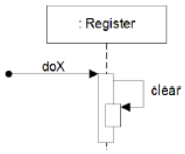
Más adecuados para observar la perspectiva cronológica de las interacciones. Muestran la forma en que un objeto interacciona con otros. Idea Principal: Que las interacciones entre los objetos se realicen en una secuencia establecida y que esta secuencia se tome el tiempo necesario para ir del inicio al fin.

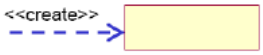
- **Objetos**: Se representan en la parte superior como rectángulos y con el nombre subrayado. Debajo de cada objeto hay una línea discontinua "*línea de vida*". Junto con esta línea se encuentra un pequeño rectángulo conocido como "*activación*", que representa el período de tiempo en el cual el objeto está realizando una operación. La longitud del rectángulo se interpreta como la duración.

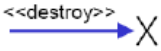


Pueden ser elementos concretos (Instancias) que representan algo del mundo real, o elementos prototípicos (Roles) que representan cualquier elemento de cierto tipo.

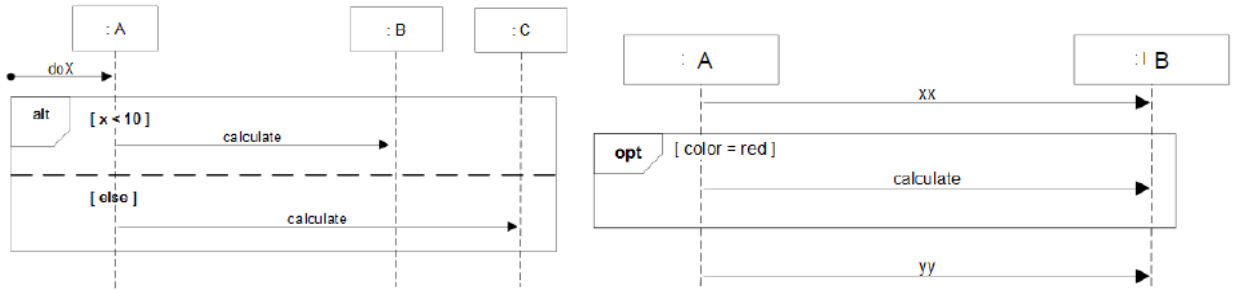
- **Mensajes**: Representan la forma de comunicación entre objetos. Un mensaje va de un objeto a otro. Pasa de la línea de vida de un objeto a la de otro. Representados gráficamente por líneas continuas con una punta de flecha. Un objeto puede enviarse un mensaje a sí mismo.



  - Síncronos: El objeto que envía el mensaje espera la respuesta a tal mensaje antes de continuar su trabajo. 
  - Asíncronos: El objeto que envía el mensaje no esperará una respuesta antes de continuar. 
  - Retorno: Devuelve un valor al emisor. 
  - Creación: Crea un objeto. 

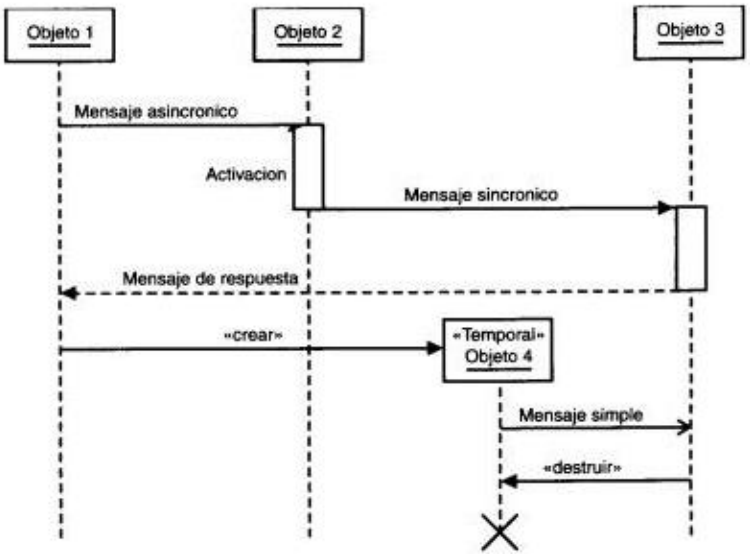
  - Destrucción: Destruye un objeto. Un objeto puede destruirse a sí mismo. 

- **Tiempo:** Representado en una progresión vertical. Se inicia en la aprte superior y avanza hacia la parte inferior. El diagrama de secuencia tiene dos dimensiones: la dimensión horizontal es la disposición de los objetos y la dimensión vertical muestra el paso del tiempo.
- **Fragmentos:** Mecanismo a partir del cual se pueden realizar la especificación de bloques repetitivos, opcionales o alternativos, entre otros.
  - **Alt:** indica que el fragmento de diagrama es una alternativa
  - **Loop:** Indica que el fragmento se ejecuta repetidas veces
  - **Opt:** Indica que el fragmento es opcional.
  - **Par:** indica que el fragmento incluye varias hebras.



Los diagramas de secuencia muestran las interacciones entre objetos en un escenario concreto o de un caso de uso general. Si se modela un escernario concreto podemos hablar de diagramas de secuencia de instancias.

Si se tienen en cuenta todos los escenarios posibles para crear el diagrama de secuencia, se trataría de un **diagrama de secuencia genérico**. Para representar una condición en la secuencia, tal condición se colocará en un "si" entre corchetes. Cada condición provocará una bifuración en rutas distintas. Como cada ruta irá al mismo objeto, la bifuración caursará una "ramificación" del control en la línea de vida del objeto receptor y separará las líneas de vida en rutas distintas.



Otro concepto importante es la creación de objetos. Con frecuencia se da el caso de que un programa orientado a objetos debe crear un objeto. Cuando se crea un objeto se representará con un rectángulo con nombre pero no se colocará en la parte superior, sino en el instante en el que se cree.

ERRORES:

- No hacer un diagrama de secuencia para cada caso de uso.
- No identificar todos los objetos necesarios
- No proveer de texto a las flechas de mensajes
- Dar más importancia a funciones get y set en lugar de enfocarse en métodos importantes
- Las flechas no invocan a operaciones de clases

DIAGRAMAS DE COLABORACIÓN

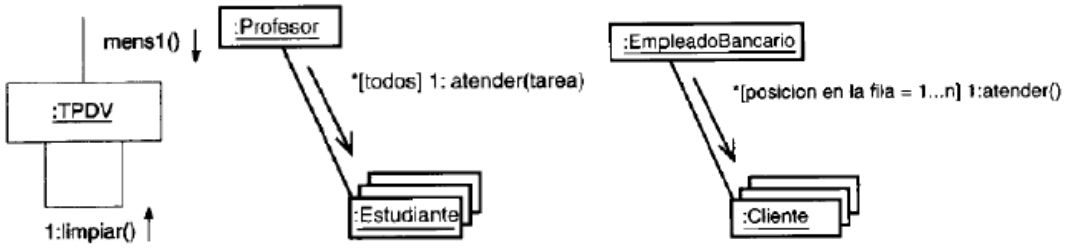
Los **diagramas de colaboración** muestran la forma en que los objetos colaboran entre sí, al igual que ocurre en el diagrama de secuencia. La diferencia entre ambos diagramas es:

- Los diagramas de secuencia destacan la sucesión de las interacciones (organizado respecto al tiempo).
- Los diagramas de colaboración destacan el contexto y organización general de los objetos que interactúan (organizado respecto al espacio).

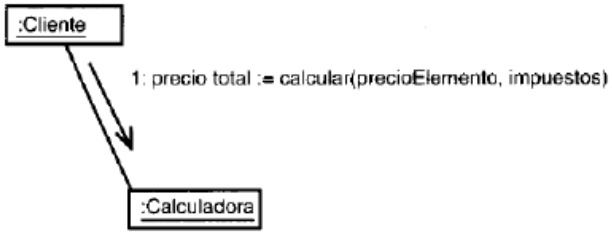
Un diagrama de colaboración es una extensión de un diagrama de objetos. Además de las relaciones entre objetos, el diagrama muestra los mensajes que se envían los objetos entre sí. Para representar un mensaje se dibuja una flecha cerca de la línea de asociación (la flecha apuntará al objeto receptor). El mensaje indicará al objeto receptor que ejecute alguna de sus operaciones.

Son útiles en la fase exploratoria para identificar objetos. La distribución de los objetos en el diagrama permite observar adecuadamente la interacción de un objeto respecto de los demás. La estructura estática viene dada por los enlaces, la dinámica por el envío de mensajes por los enlaces.

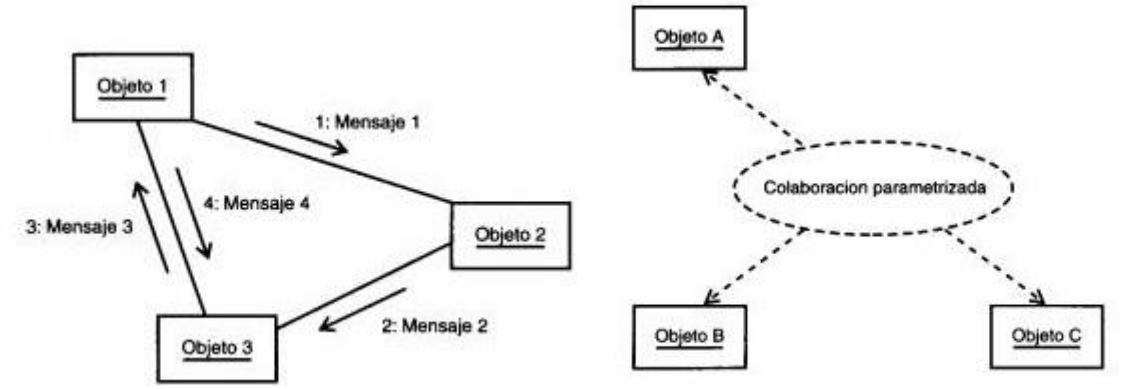
- Un objeto puede enviarse un mensaje a sí mismo.
- En ocasiones un objeto envía un mensaje a diversos objetos de la misma clase. Se representan mediante una pila de rectángulos. *Un profesor pide a un grupo de estudiantes que entregue una tarea.*
- En algunos casos el orden del mensaje enviado es importante. *Por ejemplo, un cajero atenderá a cada cliente en orden de llegada.* Esta opción se representará como un mientras cuya condición implicará el orden.



Un mensaje podría ser una petición a un objeto para que realice un cálculo y devuelva un valor. Para representar este caso se deberá escribir una expresión que tenga el nombre del valor devuelto a la izquierda, seguido de ":=", a continuación el nombre de la operación y las cantidades con que opera para producir el resultado. *Un objeto cliente podría solicitar a un objeto calculadora que calcule el precio total de un producto con el impuesto asociado.*



- Sincronización: un objeto solo puede enviar un mensaje después de que otros mensajes hayan sido enviados. El objeto debe sincronizar todos los mensajes en el orden adecuado.
- Un mensaje desencadena una acción en un objeto destinatario. Un mensaje se envía si han sido enviados los mensajes de una lista. A.1, B.3 / 1:Mensaje
- Un mensaje se envía iterada y secuencialmente a un conjunto de instancias: 1 *[i:=1..n] : Mensaje
- Un mensaje se envía iterada y concurrentemente a un conjunto de instancias: 1 *| | [i:=1..n] : Mensaje
- Un mensaje se envía de manera condicionada: [x>y] 1: Mensaje
- Un mensaje que devuelve un resultado: 1: distancia:= mover(x,y)



TEMA 3.5 – DIAGRAMAS DE ESTADO Y DE ACTIVIDAD

DIAGRAMAS DE ESTADO

Los **diagramas de estado** se utilizan para modelar el comportamiento, están orientados a eventos. Muestran cómo las partes de un modelo UML cambian con el tiempo. Al pasar el tiempo y según suceden los acontecimientos hay cambios que afectan los objetos que nos rodean. Las interacciones de un sistema con los usuarios y con otros sistemas provocan una serie de cambio en los objetos que lo conforman.

Presenta los estados en los que puede encontrarse un objeto. También incluye las transiciones entre estados. Muestra los puntos inicial y final de una secuencia de cambios de estado. Muestra las condiciones de un solo objeto.

Son útiles solo para los objetos con un comportamiento significativo. El resto de objetos se puede considerar que tienen un único estado. Cada objeto está en un estado en cierto instante. El estado está caracterizado parcialmente por los valores de los atributos del objeto. El estado en el que se encuentra un objeto determina su comportamiento.

- Diagramas de interacción: modelan el comportamiento de una sociedad de objetos, mientras que la máquina de estados modela el comportamiento de un objeto individual.
- Diagramas de actividades: se centran en el flujo de control entre actividades, no en el flujo de control entre estados. El evento para pasar de una actividad a otra es la finalización de la anterior actividad.

Representación gráfica:

- Un estado se representa como un rectángulo con bordes redondeados.
- Las flechas indican una transición de estado. La punta de flecha apunta hacia el estado donde se hará la transición.

- El estado inicial se representa con un círculo relleno. El estado final con un círculo relleno con borde.
- Un estado también puede tener tres áreas: nombre del estado, variables del estado y actividades.

Actividades internas:

- Se puede especificar el hacer una acción como consecuencia de entrar, salir o estar en un estado.

```
entry: acción por entrar
exit: acción por salir
do: acción mientras en estado
```

- Se puede especificar el hacer una acción cuando ocurre en dicho estado un evento que no conlleva salir del estado.

```
on evento_activador( arg1 )[ condición ]: acción por evento
```

Un **estado** es una situación en la vida de un objeto caracterizada por satisfacer una condición: esperar un evento (estática) o realizar una actividad (dinámica). Cada estado tiene un nombre. El estado de un objeto está relacionado con los valores de sus atributos, los enlaces a otros objetos y las actividades que esté realizando.

Un **evento** representa la ocurrencia de un suceso, dentro o fuera del objeto, que provoca un cambio de estado (dispara una transición).

Una **transición** es una relación entre dos estados: indica que cuando ocurre un evento, el objeto pasa de un estado a otro. Un estado tiene duración, una transición es inmediata. Pueden tener varios eventos vinculados. Tiene tres partes opcionales:

- Evento: suceso en el tiempo
- Condición o guarda: autoriza a la transición si se cumple la condición
- Acción: operación atómica que se ejecuta antes de que la transición alcance el nuevo estado

```
Evento[ condición ] / acción
```

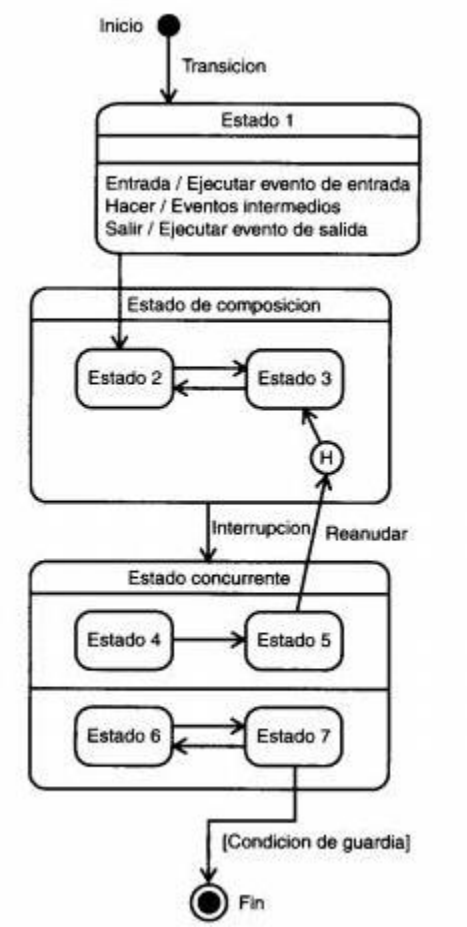
```
Evento( arg1, arg2 )[ condición ] / ^otro_objeto.evento(arg2)
```

Tipos de estados:

- Simple: sin estructura interna
- Compuesto: tiene estructura interna con varios estados interiores (subestados)
    - Estado ortogonal (subestados concurrentes): se dividen en dos o más regiones. Cuando el estado está activo significa que lo está uno de los subestados de cada región.
    - Estado no ortogonal (subestados secuenciales): contiene uno o más subestados directos. Cuando el estado está activo significa que lo está uno y solo uno de los subestados.
- Estado inicial: indica el punto de comienzo por defecto para la máquina de estados o para el subestado.
- Estado final: indica que la ejecución de la máquina de estados o estado que lo contiene, ha finalizado. Si la máquina tiene uso infinito, puede no tener estado final, pero siempre tendrá estado inicial.
- Una transición desde fuera de un estado compuesto puede apuntar a:
    - El estado compuesto: la máquina de estados anidada debe incluir un estado inicial, al cual pasa el control al entrar al estado compuesto.

    - Un subestado anidado: el control pasa directamente a él.
- Subestados concurrentes: Las regiones ortogonales permiten especificar dos o más máquinas de estados anidadas que se ejecutan en paralelo en el contexto del objeto que las contiene. El estado compuesto acaba mediante una sincronización de las regiones ortogonales: las regiones que alcanzan sus estados finales quedan a la espera hasta que todas las regiones acaban, y entonces concluye el estado compuesto. Cada región ortogonal puede tener un estado inicial, un estado final y un estado de historia.
- Transición de División (FORK): El control pasa de un estado simple a varios estados ortogonales, cada uno de una región ortogonal diferente. Las regiones para las que no se especifica subestado destino toman como tal, por defecto, el estado inicial de la región.
- Transición de Unión (JOIN): Varias entradas, cada una de un subestado de una región ortogonal diferente, pasan el control a un único estado simple. Puede tener un evento disparador. La transición ocurre si todos los subestados origen están activos. El control sale de todas las regiones ortogonales, no solo de las que tienen subestado de entrada a la unión.

La **destrucción de objetos** es efectiva cuando el flujo de control del autómata alcanza un estado final no anidado. La llegada a un estado final anidado implica la "subida" al superestado asociado, no el fin del objeto.



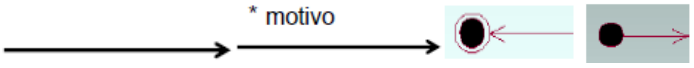
DIAGRAMAS DE ACTIVIDAD

Los **diagramas de actividad** se utilizan para describir la lógica de los procesos, los procesos de negocio y el flujo de trabajo. Muestran el flujo de control entre actividades. Son similares a los diagramas de flujo, pero su principal diferencia es que los diagramas de actividad soportan actividades en paralelo.

Cada diagrama representa una actividad que puede estar formada por otras actividades más pequeñas. Mientras un diagrama de interacción muestra objetos que pasan mensajes, uno de actividades muestra las operaciones que se pasan entre objetos. Sirven para modelar: la dinámica de un conjunto de objetos, el flujo de control de una operación o un caso de uso; o un proceso de negocio o un flujo de trabajo.

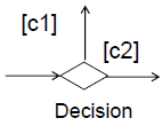
Nodos:
- Acciones: nodos de actividad atómicos
- Actividades: nodos de actividad con estructura interna
- Nodos de control: controlan el flujo
- Objetos de valor: objetos o datos utilizados

Elementos: el diagrama de actividad muestra los pasos de la computación (transición entre actividades)
- Cada paso define un "estar haciendo algo" (estado de actividad)
- El paso de una entidad a otra (transición) se realiza mediante un disparador, que normalmente es el fin de la actividad.
- Un transición se puede disparar n veces (iteración)
- Símbolos especiales para determinar el inicio y el fin de la actividad.

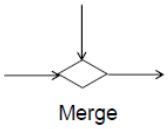


Flujos de control: cuando se completa una acción o un nodo de actividad, el flujo de control pasa a la siguiente acción o nodo de actividad. Debe haber un inicio y una terminación (pueden ser varios).
- **Bifurcaciones** (Decision Node): representan caminos alternativos, elegidos en función del valor de una expresión booleana. Se representan con un rombo. Pueden tener un flujo de entrada y dos o más de salida. En cada flujo de salida se coloca una guarda (expresión booleana), que se evalúa al entrar en la bifurcación
  - Las guardas no deben solaparse (para que solo una sea cierta a la vez)
  - Pero deben cubrir todas las posibilidades (para que siempre haya una cierta).
  - Se puede usar "else" para marcar un flujo de salida alternativo

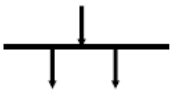
  Se usa cuando en una actividad no hay lógica involucrada, sino sólo comparación.



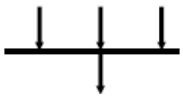
- **Fusiones** (Merge Node): Los caminos antes separados se puede juntar a un rombo con varias entradas y una salida. Aquí no hay guardas.



- **Divisiones** [Fork Node]: Representan la separación de un flujo de control sencillo en dos o más flujos de control concurrentes. Tienen una transición de entrada y dos o más de salida, cada una de las cuales representa un flujo independiente. Las actividades de cada camino después de la división continúan en paralelo.



- **Uniones** [Join Node]: representan la sincronización de dos o más flujos de control concurrentes. Cada flujo de entrada espera hasta que todos han alcanzado la unión. Tiene dos o más transiciones de entrada y una de salida.



Los diagramas de actividades describen lo que sucede pero no quién hace qué. Esto no acarrea grandes problemas, ya que, a menudo interesa saber únicamente qué es lo que se hace. Si se desea mostrar quién hace cada cosa, se puede dividir un diagrama de actividad en particiones. Cada partición mostrará qué acciones se llevan a cabo por una clase o una unidad de la organización.

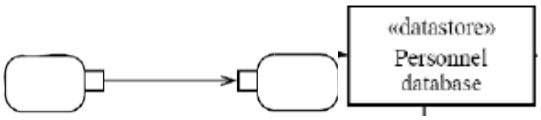
Los flujos de objetos son flujos en los cuales se ven involucrados objetos. Los objetos se representan como nodos objeto conectados con flechas que los crean o los consumen. También se puede mostrar cómo cambia el estado del objeto (se muestra el estado entre corchetes debajo del nombre del objeto).

Object flow (without activity nodes)

Two object flow edges linking object nodes and actions

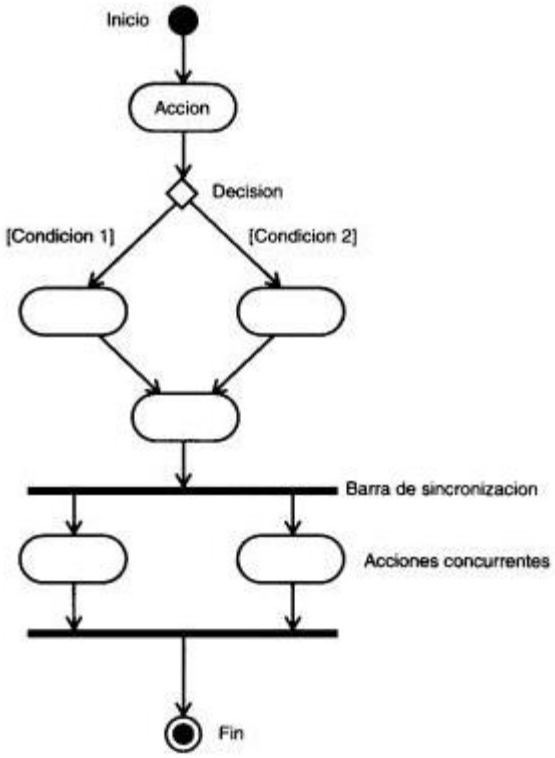
Los **nodos objeto** indican que una instancia de una clase (opcionalmente en cierto estado) está disponible en un punto particular de una actividad. Los **parámetros de acciones** [Pin] son una manera especial de flujo de objetos que representan elementos que proveen valores de entrada para las acciones o valores resultantes de ellas. Se representan como pequeños cuadrados en el borde del símbolo de la acción.

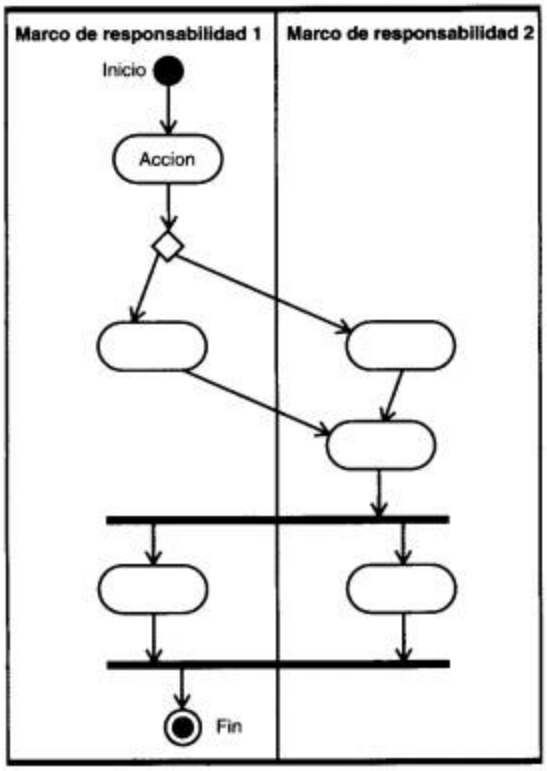
Un **almacén de datos** [Data Store] es un tipo especial de nodo objeto que representa un repositorio persistente de información.



Relación con el caso de uso:

- El DCU describe el sistema exclusivamente desde el punto de vista de la funcionalidad esperada por el usuario. El DA, cuando se asocia a un caso de uso, permite definir el funcionamiento interno de cada requisito funcional.
- Si el DCU se ha completado con descripciones de flujo principal y alternativo, los estados de actividad que describen el caso de uso pueden ser descubiertos a partir de esta descripción.
- Cada DA puede incorporar muchos casos de uso o sólo parte de un caso de uso, en función de cómo los hayamos modelado previamente.





TEMA 4.1 – METODOLOGÍAS TRADICIONALES

Una **metodología** es un proceso para producir software de manera organizada empleando una colección de técnicas y convenciones de notación predefinidas.

Distinguimos **dos tipos de metodologías** en cuanto a su **filosofía**: **ágiles** y **tradicionales**. En cuanto a las **notaciones** usadas para especificar artefactos: **estructuradas** y **orientadas** a objetos.

Se debe seleccionar la **metodología** que **más** se adapte al **tamaño** y **estructura** de la organización y el tipo de **aplicación** a desarrollar.

Algunas **características deseables de una metodología** pueden ser: cobertura total, planificación y control, comunicación efectiva, herramientas CASE, fácil formación…

- Metodologías tradicionales: mayor énfasis en la planificación y control del proyecto, en la especificación precisa de requisitos y modelado.
- Metodologías Ágiles: orientadas a la generación de código con ciclos muy cortos de desarrollo, se dirigen a equipos de desarrollo pequeños e involucran activamente al cliente en el proceso.

Las **metodologías estructuradas** representan **procesos, flujos** y **estructuras** de **datos**. En el análisis se identifican las funciones, en el diseño los datos. Se basan en **métodos descendentes**, se analiza desde una visión global hasta un nivel concreto. Los **componentes de esta metodología** son:

1. **Diagrama de flujo de datos.**
2. **Diccionario de datos:** listado organizado de todos los datos existentes en el sistema
3. **Especificación de procesos:** descripción de qué sucede en cada burbuja primitiva del nivel más bajo de un DFD. Define lo que debe hacerse para transformar las entradas en salidas.
4. **Diagramas Entidad-Relación.**

Se utilizan **herramientas** como tabla de decisiones**,** lenguaje estructurado diagramas de flujo… Podemos destacar las siguientes metodologías estructuradas concretas:

1. **Metodología de DeMarco**:
   a) Construir el modelo físico y lógico actual.
   b) Derivación del nuevo modelo lógico.
   c) Crear un conjunto de modelos físicos alternativos.
   d) Estimar los costes y tiempos de cada opción.
   e) Seleccionar un modelo y empaquetar la especificación.
2. **Metodología de Gane y Sarson**:
   a) Construir el modelo lógico actual.
   b) Construir el modelo del nuevo sistema.
   c) Seleccionar un modelo lógico.
   d) Crear el nuevo modelo físico del sistema y empaquetar la especificación.
3. **Metodología de Yourden/Constantine**:
   a) Realizar el DFD del sistema y el diagrama de estructuras.
   b) Evaluar el diseño y prepararlo para la implantación

Las metodologías estructuradas **no son adecuadas para aplicaciones de tamaño medio o pequeño por la sobrecarga de trabajo**, se invierte más tiempo en cómo se va a desarrollar que en el desarrollo en sí, hay que hacer mucho rediseño y redocumentación…

Las **metodologías orientadas a objetos** examinan el dominio del problema como un **conjunto de objetos que interactúan entre sí**. Estos objetos agrupan datos y operaciones. Sus atributos no son públicos y sólo son accesibles a través de las operaciones. Los objetos son descritos en función de sus **atributos** (datos) y su **comportamiento** (procesos).

La metodología OO es un **proceso unificado** (UP):

1. Es un solo marco de proceso: se adapta a las características del proyecto.
2. Está dirigido por casos de uso.
3. Se centra en la arquitectura: prioritaria de principio a fin.
4. Es iterativo e incremental.

Se hacen **iteraciones de entre 2 y 12 semanas**, y en cada una de ellas el resultado es un sistema funcionando pero no necesariamente listo para distribución. Se busca el feedback del usuario y los riesgos guían la funcionalidad.

Las **fases del UP** son:

1. **Inicio del proyecto**: **definir** contexto, riesgos, factibilidad y objetivos.
2. **Elaboración**: se especifican los casos de uso y la arquitectura.

3. **Construcción**: **desarrollar** el producto iterativamente.
4. **Transición**: **distribuir** el producto en versión beta, los usuarios lo prueban e informan los defectos para corregirlas posteriormente.

Por último, las metodologías OO son **adecuadas para sistemas interactivos pero no para sistemas en tiempo real con requisitos severos**.

TEMA 4.2 – METODOLOGÍAS ÁGILES

Las **metodologías ágiles** simplifican la complejidad de otras metodologías haciendo que la carga de gestión y control sea más liviana. Algunas de sus principios son:

1. Su mayor prioridad es **satisfacer** al **cliente** mediante la **entrega temprana** y continua de SW de valor.
2. Aceptan que los **requisitos cambien** y entregan SW funcional frecuentemente.
3. Los implicados **trabajan juntos de forma cotidiana** y se comunican en **conversación de cara a cara**.
4. La **simplicidad** es esencial.
5. El **SW funcionando** es la medida principal de progreso.
6. Promueven el desarrollo sostenible. Promotores, desarrolladores y usuarios deben ser capaces de mantener un **ritmo constante** de forma indefinida.

• Según las características de un proyecto:

	Ágil	Prescriptivo
Prioridad del negocio	Valor	Cumplimiento
Estabilidad de requisitos	Entorno inestable	Entorno estable
Rigidez del producto	Modificable	Difícil de modificar
Coste del prototipado	Bajo	Alto
Criticidad del sistema	Baja	Alta
Tamaño del equipo	Pequeño	Grande

Tipos de metodologías ágiles más conocidos**: XP, SCRUM y Kanban**.

XP – EXTREME PROGRAMMING

El **eXtreme Programming (XP)** se centra en potenciar las **relaciones interpersonales** para tener éxito. Para ello debe existir una realimentación continua entre cliente y equipo de desarrollo.
Esta metodología **es adecuada para proyectos con requisitos imprecisos,** muy **cambiantes** y donde existe un **alto riesgo técnico**.

Los **roles en XP** pueden ser:
- Programador: escribe las pruebas unitarias y produce el código del sistema.
- Cliente: escribe las historias de usuario y las pruebas funcionales para validar su implementación. Asigna prioridades centrándose en aportar mayor valor al negocio.
- Encargado de pruebas (Tester): ayuda al cliente a escribir las pruebas funcionales. Ejecuta las pruebas regularmente y difunde los resultados en el equipo.
- Encargado de Seguimiento (Tracker): proporciona realimentación al equipo. Debe verificar el grado de acierto entre las estimaciones y el tiempo real dedicado.

- Entrenador (Couch): responsable del proceso global. Guía al equipo para seguir el proceso correctamente.
- Consultor: Miembro externo del equipo con un conocimiento específico necesario para el proyecto. Guía al equipo para resolver un problema específico.
- Gestor (Big Boss): dueño del equipo. Es vínculo entre clientes y programadores. Labor principal es la coordinación.

El **ciclo de desarrollo** se mide en **iteraciones,** y dentro de una iteración se siguen estos pasos:

1. El **cliente define el valor** del negocio a implementar.
2. El **programador estima el esfuerzo** para su implementación.
3. El **cliente selecciona** qué construir.
4. El **programador construye** ese valor de negocio.

El **ciclo de vida ideal** de XP son 6 fases:

1. **Exploración:** Clientes plantean a grandes rasgos las historias de usuario (de pocas semanas a pocos meses).
2. **Planificación de entrega:** El cliente establece prioridades y los programadores estiman el esfuerzo (unos pocos días). Se puede planificar en tiempo (nº iteraciones por velocidad del proyecto) o en el alcance (se divide la suma de puntos de las historias entre la velocidad del proyecto).
3. **Iteraciones:** Compuesto por iteraciones de no más de tres semanas. En la primera iteración se puede establecer una arquitectura del sistema y al final de la última iteración el sistema estará listo para entrar en producción.
4. **Producción:** Requiere pruebas adicionales y revisiones de rendimiento. Tomar decisiones sobre la inclusión de nuevas características.
5. **Mantenimiento:** Mientras la primera versión se encuentra en producción, XP debe mantener el sistema en funcionamiento al mismo tiempo que desarrolla nuevas interacciones. Puede requerir nuevo personal en el equipo o cambios en su estructura.
6. **Muerte del proyecto:** El cliente no tiene más historias para incluir en el sistema. Se deben satisfacer las necesidades del cliente en otros aspectos como rendimiento y confiabilidad del sistema. Se genera la documentación final del sistema y no se realizan más cambios en la arquitectura. También puede ocurrir cuando el sistema no genera los beneficios esperados o cuando no hay presupuesto para mantenerlo.

Las **historias de usuario en XP** deben poder implementarse en un par de semanas como máximo. Cada una tiene una **prioridad** definida por el cliente y un **tiempo** definido por el equipo de desarrollo. Cada historia debe tener una **prueba de validación**. Una historia puede valer de **entre 1 y 3 puntos** (un punto equivale a una semana ideal de programación). Deben responder a tres preguntas: Quién se beneficia, qué se quiere y cuál es el beneficio.

Prácticas en XP:
- Entregas pequeñas y frecuentes, al menos una vez cada 2 o 3 meses.

- Diseño simple, énfasis en diseñar la solución más simple posible. Complejidad innecesaria es eliminada inmediatamente.
- El desarrollo del sistema es conducido por pruebas continuas. El propósito real del código no es cumplir un requerimiento sino pasar las pruebas.
- Refactorización, mejora la estructura del código sin alterar su comportamiento.
- Programación por pares
- Propiedad colectiva del código, cualquier persona puede cambiar cualquier parte del código.
- Integración continua, el sistema integra y reconstruye muchas veces al día.
- Ritmo sostenible, se trabaja semanas de 40 horas.
- Todo el equipo trabajando junto

Algunos de sus **principios básicos** son: **simplicidad, comunicación, retroalimentación** y **valentía**.

SCRUM

El **SCRUM** es una metodología **iterativa e incremental** con un equipo auto-dirigido que hace reuniones diarias e iteraciones de máximo 4 semanas. Al final de una iteración (**sprint**) se tiene una demo para el cliente. En SCRUM los **equipos de desarrollo** hacen un **poco de todo al mismo tiempo** (diseño, código, testing…).

Los **roles en SCRUM** pueden ser:
- Los **comprometidos:**
    - Product Owner: Responsable de que el desarrollo iterativo del producto cumpla las necesidades del cliente. Define las funcionalidades del producto.
    - Equipo de desarrollo: el equipo encargado de desarrollar el producto. Típicamente de 5 a 9 personas. Multi-funcional.
    - Scrum Master: encargado de que el proceso Scrum se realice correctamente. Elimina impedimentos.
- Los **implicados**: Usuarios de aplicación, clientes, managers, marketing,…

SCRUM establece **unas reuniones obligatorias** con un límite de tiempo para que sean eficientes.
Podemos destacar los siguientes tipos de reuniones:

1. La **planificación del Sprint**: definen el **objetivo** del sprint y seleccionan qué **historias** con su estimación se pueden hacer en él. El resultado de cada Sprint debe ser una pieza de SW funcionando. Se obtiene: la meta y el backlog del Sprint. La reunión es conducida por el Scrum Master a la que deben asistir el Product Owner, todo el equipo u opcionalmente otros implicados. No se recomienda extender esta planificación más allá de un día laboral completo.
    - Parte1: El dueño del producto prioriza las tareas obteniendo el backlog del producto
    - Parte2: El equipo divide las tareas creando el backlog del Sprint
2. El **SCRUM Diario**: cada día, todos los miembros del equipo durante 15 minutos, **responden a preguntas** (qué hiciste ayer, qué vas a hacer hoy, qué impedimentos tienes para lograrlo). Sincronización de actividades y actualización de estado del proyecto.
3. La **revisión del Sprint**: se realiza al final de cada sprint. Se hace una **demostración** de la funcionalidad del SW. El equipo de desarrollo realiza las pruebas de funcionalidad para cada

uno de los requerimientos del Sprint. Se añade al backlog del producto el feedback que de nuevo es priorizado por el dueño del producto.

4. La **retrospectiva del Sprint**: se evalúan los **resultados** durante el Sprint y cómo se puede **mejorar**. Se realizan mejoras en la forma de trabajo.
5. **Sprint de Release:** Tras varios Sprints el equipo obtiene un producto preparado para ser puesto en producción.

SCRUM obliga a tener **3 tipos de documentos**:

- El **product backlog**: requisitos del cliente. Inventario de funcionalidades, mejoras, tecnología y corrección de errores que deben incorporarse al producto ordenadas por prioridad. Se deben especificar los criterios de aceptación para cada ítem del Backlog del producto para considerar cumplido el requisito. Es un documento vivo, en constante evolución durante el desarrollo del sistema.
- **Gráfico Burn-Up**: herramienta de planificación y seguimiento del Product Owner. Muestra el plan general de desarrollo del producto y la traza de su evolución.
- **Sprint del Backlog**: definición y estimación de las tareas de programación para cumplir los requerimientos del Sprint. Si una tarea implica menos de 4 horas debería considerarse como parte de otra. Si una tarea implica más de 12 horas debería considerarse dividirla en sub-tareas. El objetivo es desglosar tareas a las que un miembro del equipo se puede comprometer a terminar de un Scrum diario a otro. Para cada tarea de la pila del Sprint se indica la persona que la tiene asignada y el tiempo de trabajo previsto.
- **Gráfico Burn-Down:** permite visualizar cuánto tiempo le queda al equipo para finalizar el sprint en curso. Este gráfico se actualiza diariamente durante la reunión Diaria de Scrum.

Algunos de sus **principios básicos** son: **compromiso**, **enfoque**, **honestidad**, **respeto** y **valentía**.

Esta metodología es **adecuada para**: software **comercial**, aplicaciones **financieras**, **videojuegos**, software de control satélite, sitios **web**…

KANBAN

El **Kanban** se basa en ideas muy simples: **visualiza el flujo de trabajo, limita el trabajo en curso, mide el flujo de trabajo e identifica mejoras.** En Kanban **no existen roles ni iteraciones de tiempo fijo**. Kanban se ajusta a nuestras necesidades si nuestras prioridades cambian todos los días y no nos gusta fijar compromisos. Esta metodología proporciona una **evolución más gradual** y una forma de practicar el desarrollo ágil sin necesidad de iteraciones de tiempo preestablecido, además de poder los cuellos de botella más fácilmente.

Algunos conceptos para entenderlo mejor:

1. Limita el WIP (ajusta carga a la capacidad).
2. Mide y mejora el tiempo de entrega.
3. Empezar menos cosas y acabar más cosas…
4. No se construyen funcionalidades que no se necesiten en el momento.
5. No se escribe más código del que se puede testear…
6. Liberar recursos en caso de bloqueos.

Es **adecuada para**: aplicaciones **multimedia** de vídeo, de web, de radio, de periódicos… y de **producción de videojuegos** porque ayuda a gestionar el WIP y el flujo de trabajo rápidamente.

Scrum	Kanban
Duración **iteraciones** preestablecida	Iteraciones **opcionales.**
El equipo asume un **compromiso** a realizar una cantidad de trabajo en cada iteración	Compromiso **opcional**
Se utiliza la **velocidad** como métrica para planificar y mejorar el proceso	Se utiliza el **Lead time** como métrica por defecto para planificar y mejorar el proceso
Equipos **multi-funcionales**	Equipos multi-funcionales **opcionales.** Se permiten **especialistas**
Los ítems se deben **descomponer** para que sean completados en 1 sprint	**No hay un tamaño** particular **prescrito**
Gráfica burndown prescrita	**No hay** ningún tipo de diagrama prescrito
WIP limitado indirectamente (vía plan del sprint)	**WIP limitado directamente** (por estado del flujo de trabajo)
Estimación prescrita	Estimación **opcional**
No se pueden añadir ítems una vez la iteración ha comenzado	**Se pueden añadir nuevos ítems** si hay capacidad disponible
Una **pila de sprint** pertenece a un **equipo en particular**	Una **pizarra kanban** se puede compartir por **múltiples equipos** o individuos
Prescribe **3 roles** (PO, SM, Team)	No prescribe **ningún rol**
La **pizarra** de scrum se **resetea** después de cada sprint	Una **pizarra** kanban es **persistente**
Prescribe una **pila de producto priorizada**	La priorización es **opcional**

Scrum vs XP

En un primer lugar, lo que más nos llamó la atención de la metodología XP fue que las tareas se asignan a parejas de programadores, lo que permite que el que no está escribiendo piense desde un punto de vista más estratégico y realice lo que podría llamarse revisión del código en tiempo real

Sin embargo, finalmente nos decantamos por la metodología SCRUM por las siguientes razones:

- El SCRUM permite desglosar las historias en tareas más pequeñas, lo que nos permite dividir el trabajo entre los distintos integrantes del equipo.
- Ofrece una demo al cliente al final de cada iteración, por lo que podemos ir comprobando si el cliente está satisfecho con el resultado obtenido.
- El equipo es auto-dirigido y auto-organizado.
- Además, gracias a las reuniones diarias podemos llevar un mejor seguimiento de las tareas que se están llevando