

TEORIA INGENIERIA DEL SOFTWARE

Jaime Lorenzo Sánchez

25 de diciembre de 2021

Capítulo 1

INTRODUCCIÓN A LA INGENIERÍA DEL SOFTWARE

Características y evolución del software

Comienza aproximadamente en los años 60, avanzando en distintas etapas con distintas evoluciones en todos los aspectos (software a medida, bases de datos, inteligencia artificial o sistemas expertos).

En sus primeras décadas, se centran en el desarrollo del software y en la reducción de costes de procesamiento y almacenamiento.

A medida que pasan los años, aparecen problemas como sobre-explotación del potencial del software, incapacidad de atender a la demanda e incapacidad de mantener el software existente.

Actualmente, el objetivo es mejorar la calidad de las soluciones software.

El software es el factor decisivo a la hora de elegir entre varias soluciones informáticas disponibles para un problema dado, pero no siempre ha sido así.

Peculiaridades del software

1. El producto software es enteramente conceptual.

2. No tiene características físicas y, en consecuencia, no está sujeto a leyes físicas o eléctricas.
3. Su naturaleza conceptual crea una distancia intelectual entre el software y el problema que resuelve.
4. Es difícil para una persona que entiende el problema entender el software que lo resuelve.
5. Para probar el software es necesario disponer de un sistema físico.
6. El mantenimiento no es sólo una sustitución de componentes.

Naturaleza y problemas del Software

El software como elemento lógico se desarrolla, no se fabrica. El software se "deteriora" con el mantenimiento y tiene un desarrollo a medida, provocando una ausencia de componentes.

Esto dio origen a la crisis del software.

La crisis del software

Se define como el conjunto de problemas que aparecen en el desarrollo del software al desarrollar, mantener y atender la demanda de nuevas aplicaciones.



Causas de la crisis del software

1. La naturaleza lógica del software.
2. Mala gestión de los proyectos.

3. Ausencia de entrenamiento formal en nuevas técnicas.
4. Resistencia al cambio.
5. Mitos del software

MITOS DE LOS DESARROLLADORES

- Programa funcionando = fin del trabajo
- Calidad = el programa se ejecuta sin errores
- Entrega al cliente: programa funcionando

MITOS DEL CLIENTE

- Requisitos establecidos como una declaración general de objetivos
- Flexibilidad del software ante los cambios

MITOS DE GESTIÓN

- Uso de estándares
- Uso de herramientas
- Mala planificación: aumento de programadores

Síntomas de la crisis del software

1. **Productividad** de los desarrolladores baja en relación a la demanda.
2. **Expectativas** de los sistemas no responden a las expectativas.
3. **Fiabilidad**: Los programas fallan a menudo.
4. **Calidad** no es adecuada.
5. **Costes** difíciles de predecir, a menudo sobrepasan lo esperado.
6. **Mantenimiento** costosa y compleja.
7. **Plazos** no se cumplen.
8. **Portabilidad**: Difícil cambiar de plataforma.
9. **Eficiencia**: No hay aprovechamiento óptimo de recursos.

Consecuencias de la crisis del software

1. **Baja productividad**
2. **Baja calidad**

Solución a la crisis del software

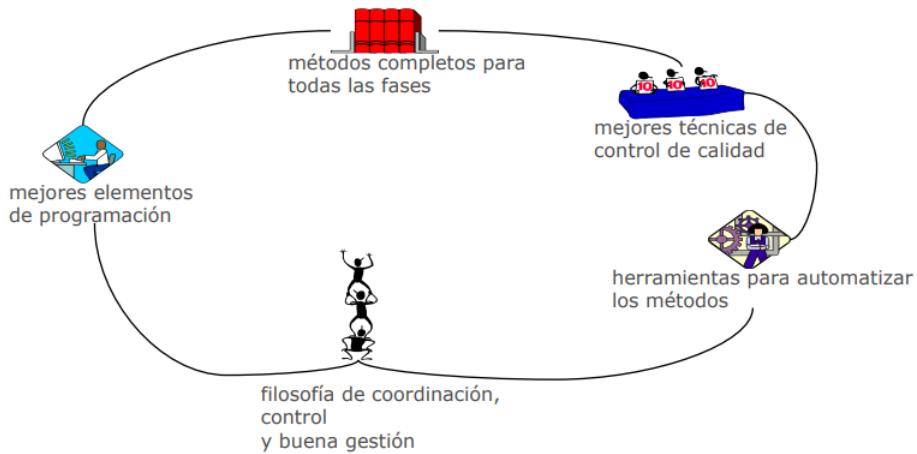
La solución a la crisis del software fue aplicar la Ingeniería del Software en la construcción de sistemas informáticos.

Esta solución fue propuesta en una conferencia de la OTAN en 1968.

Ingeniería del software

La Ingeniería del Software trata de ser la respuesta a la crisis del software.

Combina los siguientes elementos:



Se define la **Ingeniería del software** como el establecimiento y uso de principios de ingeniería para obtener software económico que trabaje de forma eficiente en máquinas reales.

En definitiva, la Ingeniería del Software es aplicar el sentido común al desarrollo de sistemas software.

Direcciones principales de la ingeniería del software

Los esfuerzos realizados para identificar las causas del problema y definir pautas estándar para la producción y mantenimiento del software se han encaminado en tres direcciones principales.

- 1. Identificación de los factores clave que determinan la calidad del software.**
- 2. Identificación de los procesos necesarios para producir y mantener el software.**
- 3. Acotación, estructuración y desarrollo de la base de conocimiento necesaria para la producción y mantenimiento de software.**

El resultado ha sido la necesidad de profesionalizar el desarrollo, el mantenimiento y la operación de los sistemas de software, introduciendo métodos y formas de trabajo sistemáticos, disciplinados y cuantificables.

¿Qué es el sentido común

- 1. Planificar antes de desarrollar.**
- 2. Diseñar antes de programar.**
- 3. Reutilizar diseños que funcionan y sean mantenibles.**
- 4. Utilización de herramientas apropiadas (Herramientas CASE).**

Utilidad de los estándares

- 1. Agrupan lo mejor y más apropiado de las buenas prácticas y usos del desarrollo de software.**
- 2. Engloban los conocimientos".**
- 3. Proporcionan un marco para implementar procedimientos de aseguramiento de la calidad.**
- 4. Proporcionan continuidad y entendimiento entre el trabajo de personas y organizaciones distintas.**

Estándares destacables en la Ingeniería del Software

- 1. SWEBOK:** Define las áreas de conocimiento que la comprenden.
- 2. ISO/IEC 12207:** Define los procesos que intervienen en el desarrollo, mantenimiento y operación del software.
- 3. CMM/CMMI, ISO/IEC TR 15504:** De las mejores prácticas, extraer modelos de cómo ejecutar esos procesos para evitar los problemas de la crisis del software.
- 4. IEEE 830, IEEE 1362, ISO/IEC 14764, etc:** Define estándares menores para dibujar criterios unificadores en requisitos, pruebas, gestión de la configuración, etc.

Participantes

Se definen como **participantes** a todas las personas involucradas en el proyecto.

Los participantes pueden ser:

1. **Cliente**: Encarga y paga el sistema.
2. **Desarrolladores**: Construyen el sistema (analistas, diseñadores, programadores, etc).
3. **Gerente o director del proyecto**: Planifica y calcula el presupuesto, coordina a los desarrolladores y el cliente.
4. **Usuarios finales**: Utilizan el sistema.

Roles o papeles

Se define un papel o rol como el conjunto de responsabilidades en el proyecto o en el sistema.

Está asociado con un conjunto de tareas y se asigna a un participante.

Un mismo participante puede cumplir varios papeles.

Definición de sistema

Se define un sistema como una realidad subyacente.

Definición de modelo

Se define un modelo como cualquier abstracción de la realidad.

Definición de producto de trabajo o Entregable

Se define un Producto de trabajo o Entregable como un artefacto o elemento que se produce durante el desarrollo (documento, fragmento de software, etc).

Puede ser de 2 tipos:

1. **Entregable interno**: Producto para el consumo interno del proyecto (revisión de la estructura de una base de datos, resultados de pruebas para el gerente, etc).

2. **Entrega:** Producto de trabajo para un cliente (especificación de requisitos, manual de usuario, producto final, etc.).

Definición de actividad o fase

Se define una actividad o fase como el conjunto de tareas que se realiza con un propósito específico que puede componerse de otras actividades.

Definición de tarea

Se define una tarea como una unidad elemental de trabajo que puede ser administrada; consumen recursos, dan como resultado productos de trabajo y dependen de productos de trabajo producidos por otras tareas.

Definición de recursos

Se definen los recursos como los bienes que se utilizan para realizar el trabajo.

Definición de objetivos

Se definen los objetivos como principios de alto nivel que se utilizan para guiar el proyecto.

1. Definen los atributos realmente importantes del sistema.
2. A veces hay conflictos entre objetivos que aumentan la complejidad del proyecto.

Definición de requerimientos

Se definen los requerimientos como las características que debe tener el sistema.

Pueden ser de 2 tipos:

1. **Requerimiento funcional:** Área de funcionalidad que debe soportar el sistema.
2. **Requerimiento no funcional:** Restricción que se establece sobre el funcionamiento del sistema.

Definición de notación

Se define una notación como un conjunto de reglas gráficas o de texto para representar un modelo.

Definición de método

Se define un método como una técnica repetible para resolver un problema específico.

Definición de metodología

Se define una metodología como una colección de métodos para la resolución de una clase de problemas.

Capítulo 2

PROCESO DE DESARROLLO DEL SOFTWARE. PARADIGMAS O MODELOS DE DESARROLLO

Definición de proceso

Se define un proceso como un conjunto ordenado de tareas, una serie de pasos que involucran actividades, restricciones y recursos, que producen una salida determinada.

Desde el punto de vista del estándar ISO/IEC 12207 1995, un proceso es un conjunto de actividades y tareas relacionadas, que al ejecutarse de forma conjunta transforman una entrada en una salida.

Definición de proceso de software

Se define un Proceso de software como un conjunto de actividades necesarias para transformar los requisitos de un usuario en un sistema software.

Características de un proceso

1. Tiene una serie de actividades principales.
2. Utiliza recursos, está sujeto a restricciones y genera productos intermedios y finales.

3. Compuesto por subprocessos que se encadenan de alguna forma.
4. Cada actividad tiene sus criterios de entrada y salida, que permiten conocer cuando comienza y termina dicha actividad.
5. Existen principios orientadores que explican las metas de cada actividad.

Los procesos aportan consistencia y estructura sobre el conjunto de actividades, lo que permite realizar la misma tarea correctamente de forma repetida.

Definición de ciclo de vida de un proceso

Proceso que implica la construcción de un producto.

Procesos primarios del ciclo de vida del software

1. **Adquisición:** Proceso global que sigue el adquiriente para obtener el producto.
2. **Suministro:** Proceso global que sigue el suministrador para proporcionar el producto.
3. **Desarrollo:** Proceso empleado por el suministrador para el diseño, construcción y pruebas del producto.
4. **Operación:** Proceso seguido por el operador en el "día a día" para el uso del producto.
5. **Mantenimiento:** Proceso empleado para mantener el producto, incluyendo tanto los cambios en el propio producto como en su entorno de operación producto.

El estándar 12207 identifica los procesos de soporte que pueden ser utilizados desde un proceso primario, o incluso desde otro proceso de soporte.

Procesos de soporte del ciclo de vida del software

1. **Documentación:** Actividades empleadas para registrar información específica empleada por otros procesos.

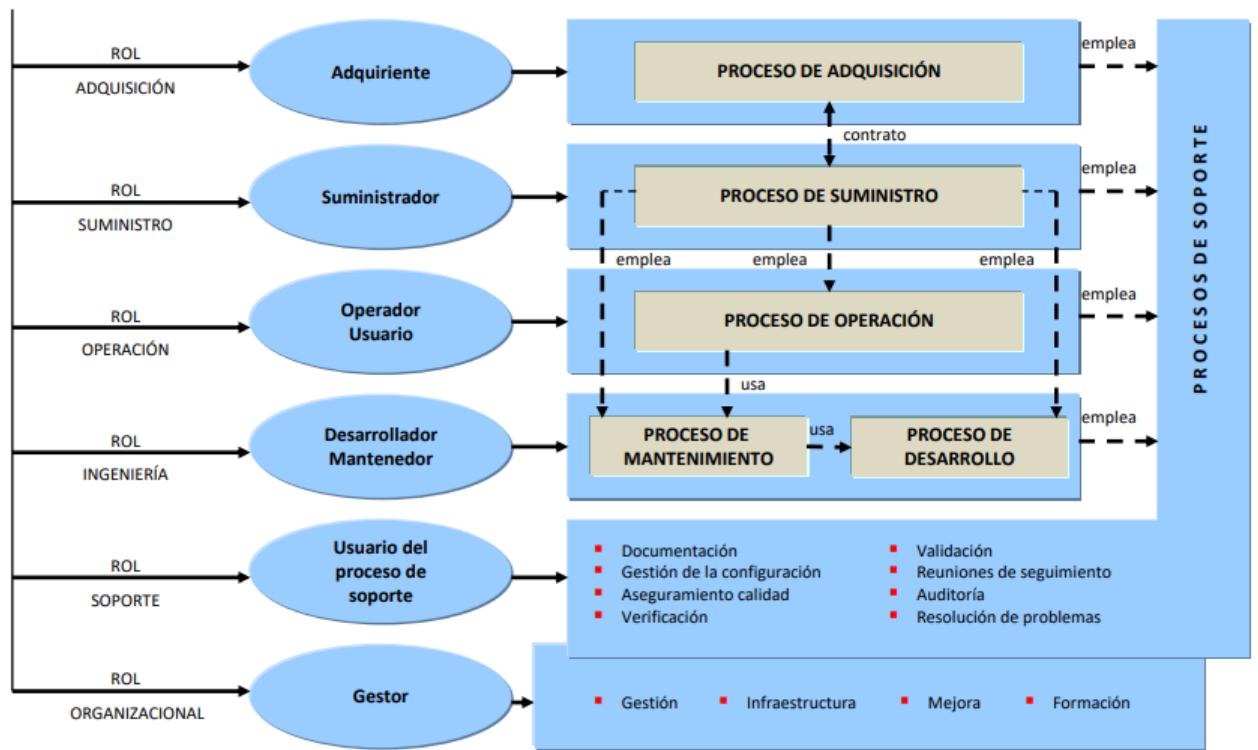
2. **Gestión de la configuración:** Actividades empleadas para mantener un registro de los productos generados en la ejecución de los procesos.
3. **Aseguramiento de la calidad:** Actividades empleadas para garantizar de forma objetiva que el producto y los procesos asociados son conformes a los requisitos documentados y a las planificaciones.
4. **Verificación:** Actividades empleadas para verificar el producto.
5. **Validación:** Actividades empleadas para validar el producto.
6. **Reuniones de Revisión:** Reuniones empleadas por las 2 partes para evaluar el estado del producto y de las actividades.
7. **Auditorías:** Actividades para determinar que el proyecto cumple con los requisitos, planes y contratos.
8. **Resolución de Problemas:** Actividades para analizar y resolver problemas relativas al proyecto, sea cual sea su fuente y naturaleza.

El estándar 12207 identifica los procesos que deben realizarse en el contexto de la organización que va a ejecutar el proyecto.

Procesos organizacionales

1. **Gestión:** Describe las actividades de gestión de la organización, incluyendo también la gestión de proyectos.
2. **Infraestructura:** Actividades necesarias para que puedan realizarse otros procesos del ciclo de vida (incluye el capital y el personal).
3. **Mejora:** Actividades realizadas para mejorar la capacidad del resto de procesos.
4. **Formación**

Visión general de los procesos, relaciones y roles



La aplicación de los procesos, tanto en el desarrollo como en el mantenimiento y operación del software, se dibuja a través de unos "patrones fijos" que configuran el esquema de mapa de situación, relación y continuidad entre los diferentes procesos, actividades y tareas.

Patrones básicos de la etapa de desarrollo

1. Desarrollo en cascada o variante secuencial
2. Desarrollo en espiral
3. Proceso Unificado de Desarrollo

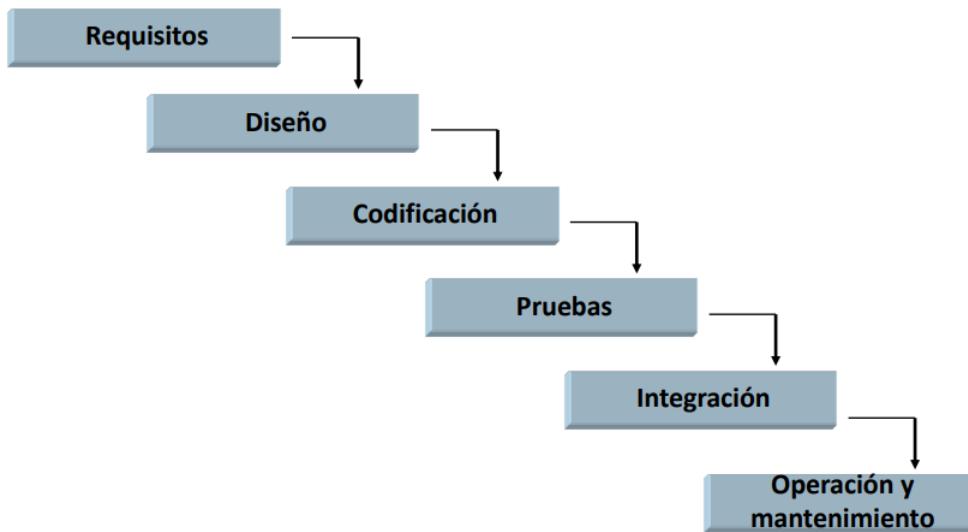
Patrones de desarrollo del ciclo de vida tras el desarrollo de la primera versión

1. Desarrollo incremental del sistema
2. Desarrollo evolutivo del sistema

Modificadores del ciclo de vida

1. Prototipado
2. Concurrencia
3. Componentes comerciales y reutilización

Modelo de ciclo de desarrollo lineal o secuencial



Refleja un desarrollo marcado por la sucesión escalonada de las etapas que lo componen.

Es necesario terminar por completo cada etapa para pasar a la siguiente.

Resulta muy rígido porque cada fase requiere como elemento de entrada el resultado completo de la fase anterior.

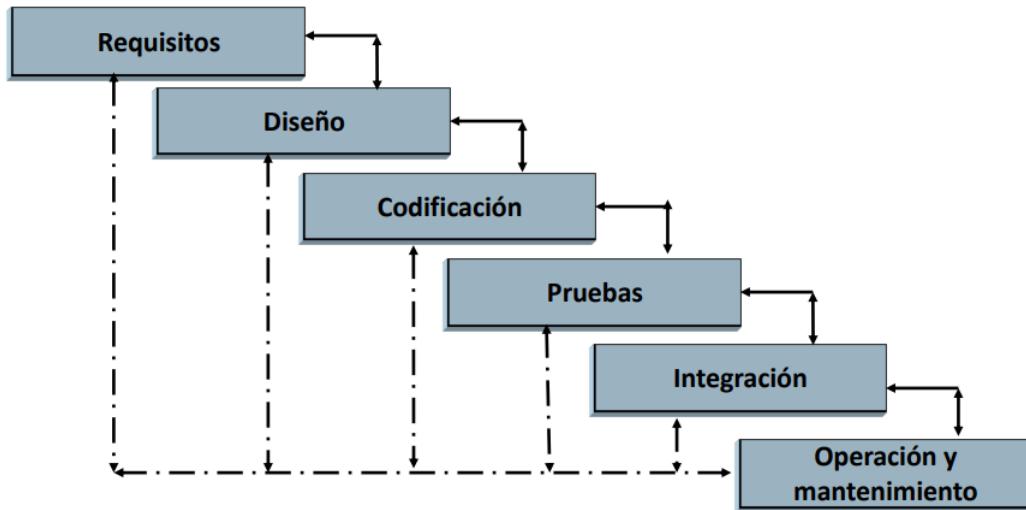
Al aplicarlo en situaciones reales su rigidez genera problemas, pues resulta difícil disponer de requisitos completos o del diseño pormenorizado del sistema en las fases iniciales, creando una barrera que impide avanzar.

Es apropiado su uso en los siguientes casos:

1. Desarrollar nuevas versiones de sistemas ya veteranos en los que el desconocimiento de las necesidades de los usuarios, o del entorno de operación no plantea riesgos.

2. Sistemas pequeños, sin previsión de evolución a corto plazo.

Modelo de ciclo de desarrollo en cascada

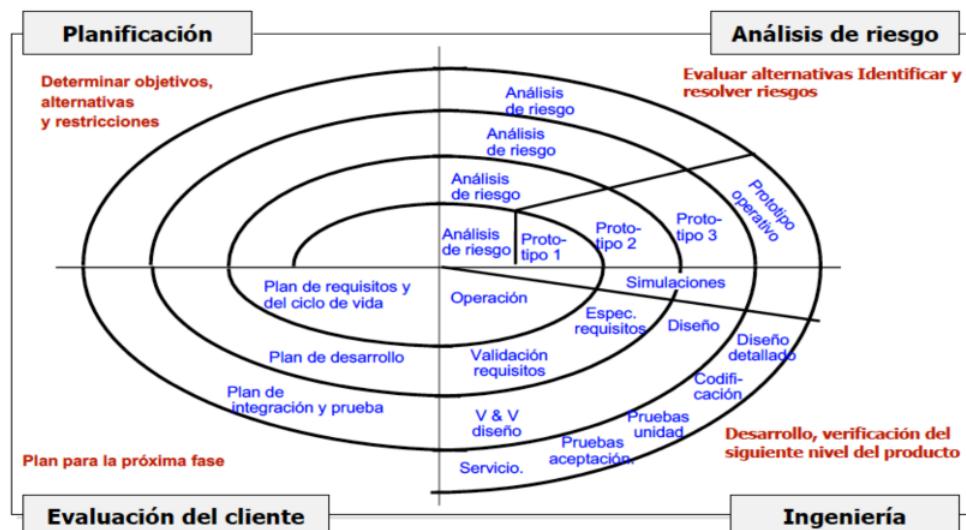


Representa la necesidad impuesta por la realidad de retornar con frecuencia desde una fase hacia las anteriores con la información generada al avanzar el desarrollo.

Es apropiado su uso en los siguientes casos:

1. Desarrollar nuevas versiones de sistemas ya veteranos en los que el desconocimiento de las necesidades de los usuarios, o del entorno de operación no plantea riesgos.
2. Sistemas pequeños, sin previsión de evolución a corto plazo.

Modelos de ciclo de desarrollo en Espiral



Presenta un desarrollo evolutivo e introduce como elemento distintivo la actividad de análisis de riesgo para guiar la evolución del proceso de desarrollo.

El ciclo de iteración se convierte en una espiral, que al representarse sobre ejes cartesianos muestra en cada cuadrante una clase particular de actividad, que se suceden de forma consecutiva a lo largo del ciclo de vida de desarrollo.

La dimensión angular representa el avance relativo en el desarrollo de las actividades de cada cuadrante.

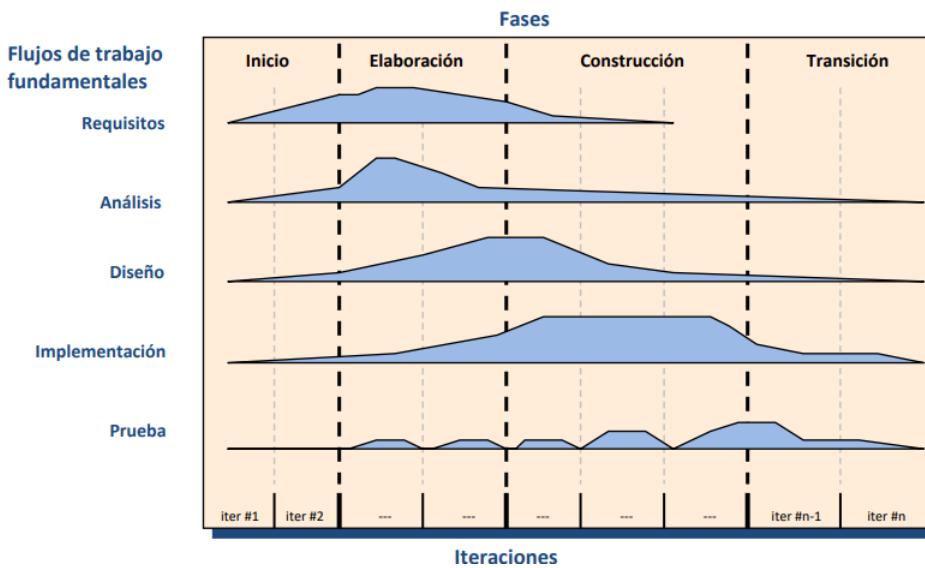
Permite múltiples combinaciones ya que en la planificación de cada ciclo se determina el avance que se va a ejecutar durante la vuelta:

1. **Obtención y validación de requisitos.**
2. **Desarrollo del diseño.**
3. **Diseño junto con la codificación.**
4. **Obtención de un subsistema completo.**

Tipos de actividades del modelo en espiral

1. **Planificación:** Se establece el contexto del desarrollo y se decide qué parte del mismo se abordará en el ciclo siguiente.
2. **Análisis de riesgo:** Evalúan las alternativas posibles para la ejecución de la siguiente parte del desarrollo, seleccionando la más ventajosa y previendo los riesgos posibles.
3. **Ingeniería:** Corresponden a las indicadas en los modelos lineales (secuencial y cascada).
4. **Evaluación:** Analizan los resultados de la fase de ingeniería, tomando el resultado de la evaluación como punto de partida para el análisis de la siguiente fase.

Modelos del ciclo de desarrollo: Proceso Unificado de Desarrollo



Fue propuesto por UML y está basado en componentes interconectados mediante interfaces.

Utiliza UML para desarrollar los esquemas y diagramas de un sistema software.

Principales aspectos definitorios del proceso unificado de desarrollo

1. **Dirigido por casos de uso.**
2. **Centrado en la arquitectura.**
3. **iterativo e incremental.**

Los casos de uso son una herramienta para especificar los requisitos de un sistema: Representan los requisitos funcionales y juntos constituyen el modelo de casos de uso, que describe la funcionalidad total del sistema.

Casos de usos que guían el proceso de desarrollo del modelo

1. Basándose en el modelo de casos de uso, se crean modelos de diseño e implementación.
2. Se revisa cada modelo para que sean conformes al modelo de casos de uso.
3. Se prueba la implementación para garantizar que los componentes del modelo de implementación implementan correctamente los casos de uso.

Los casos de uso no sólo inician el proceso de desarrollo, sino que éste sigue un hilo de trabajo que parte de los casos de uso.

El proceso unificado se repite a lo largo de una serie de ciclos.

Fases de los ciclos del modelo

1. **Inicio:** Descripción del producto final a partir de una idea inicial y análisis de negocio para el producto.

- a) Principales funciones del sistema y usuarios más importantes (modelo de casos de uso).
- b) Posible arquitectura del sistema.
- c) Plan del proyecto, coste, identificación y priorización de riesgos.

2. **Elaboración:** Se especifican en detalle los principales casos de uso.

a) **Se diseña la arquitectura del sistema:** Vistas arquitectónicas del modelo de casos de uso, del modelo de análisis, del modelo de diseño, del modelo de implementación y del modelo de despliegue.

b) Al final se pueden planificar las actividades y estimar los recursos necesarios para finalizar el proyecto.

3. **Construcción:** Se crea el producto añadiendo el software a la arquitectura.

Al final se dispone de todos los casos de uso acordados para el desarrollo, aunque puede incorporar defectos.

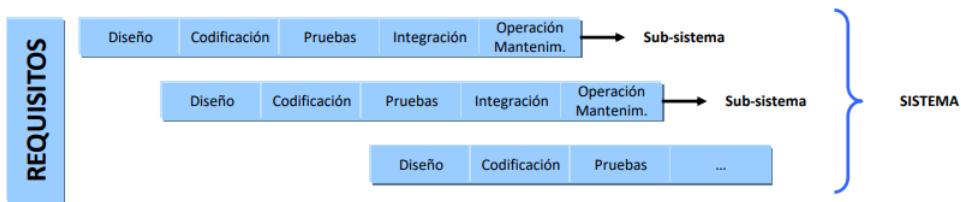
4. **Transición:** Período durante el cual el producto se convierte en versión beta, en la que los usuarios prueban el producto e informan de defectos y deficiencias.

Se corrigen problemas e incorporan sugerencias.

Incluye actividades como la formación del usuario, proporcionar una línea de ayuda y asistencia, etc.

Cada fase se divide a su vez en iteraciones.

Modelos del ciclo de evolución: Incremental



Mitiga la rigidez del modelo en cascada, descomponiendo el desarrollo de un sistema en partes; para cada una de las cuales se aplica un ciclo de desarrollo.

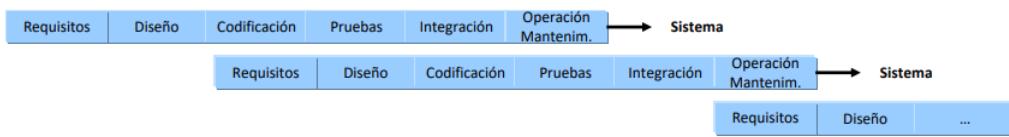
Ventajas del modelo de ciclo de evolución incremental

1. El usuario dispone de pequeños subsistemas operativos que ayudan a perfilar mejor las necesidades reales del sistema en su conjunto.
 2. El modelo produce entregas parciales en períodos cortos de tiempo, comparados con el tiempo necesario para la construcción del sistema en su conjunto, y permite la incorporación de nuevos requisitos que pueden no estar disponibles o no ser reconocidos al iniciar el desarrollo.

Este modelo es apropiado en los siguientes casos de uso:

1. Desarrollo de sistemas en los que el cliente necesita disponer de parte de la funcionalidad antes de lo que costaría desarrollar el sistema completo.
 2. Desarrollo de sistemas en los que por razones del contexto interesa realizar la obtención de los requisitos de forma escalonada a través de subsistemas.

Modelos del ciclo de evolución: Evolutivo



Está compuesto por varios ciclos de desarrollo. Cada uno de ellos produce un sistema completo en el que se operará en el entorno de operación.

La información acumulada en el desarrollo de cada sistema, y durante su fase de operación, sirve para mejorar o ampliar los requisitos y el diseño del siguiente.

En realidad, es un ciclo de vida común a todos los sistemas desarrollados que se mejoran a través de versiones sucesivas.

Este modelo es apropiado en los siguientes casos de uso:

1. Desconocimiento inicial de todas las necesidades operativas que serán precisas, generalmente por tratarse del desarrollo de un sistema que operará en un entorno nuevo sin experiencia previa.
2. Necesidad de que el sistema entre en operación en tiempos inferiores a los que serían necesarios para diseñarlo y elaborarlo de forma exhaustiva.
3. Necesidad de desarrollar sistemas en entornos cambiantes (sujetos a normas legislativas, mejora continua del producto para hacer frente a desarrollos de la competencia, etc.).

Modificadores de los modelos: Prototipado

Consiste en la construcción de modelos de prueba, que simulen el funcionamiento que se pretende conseguir en el sistema.

Tipos de prototipados

1. **Ligeros:** Dibujos de pantallas de interfaz con simulación de funcionamiento por enlaces a otros dibujo, etc.
2. **Operativos:** Módulos de software con funcionamiento propio que se desarrollan sin cubrir las funcionalidades completas del sistema, normalmente en entornos RAD.

Principal objetivo del modificador de modelos prototipado

Esta forma de trabajo previo tiene como principal objetivo la experimentación con un entorno similar al pretendido, para obtener reto-información del usuario o cliente que ayuda a los desarrolladores en la concreción de los requisitos.

Riesgos del uso del prototipado

1. Como puede parecer que se ha desarrollado una interfaz de usuario sofisticado y elaborado, el cliente puede llegar a pensar que ya se ha realizado el grueso del

trabajo.

2. Si se trata de un prototipo operativo, puede empezar a crecer al margen de la planificación, más allá de los objetivos previstos, desbordando agendas y recursos.
3. Si se trata de un prototipo operativo desarrollado fuera de desarrollo, puede mostrar al cliente funcionalidades no implementables.
4. El prototipo puede llegar a ofrecer funcionalidades superiores a lo consegurable, por estar construido en un entorno diferente al de desarrollo, o no incluir toda la funcionalidad del sistema.

Modificadores de los modelos: Concurrencia

La **concurrencia** consiste en el solapamiento de un proceso sobre otro.

Resulta bastante frecuente que aunque se haya planteado un desarrollo en cascada, se comience con una fase sin haber terminado por completo la anterior.

Puede aportar beneficios sobre la planificación de un proyecto de software, o por el contrario ser origen o consecuencia de problemas.

Factores a tener en cuenta al analizar el rendimiento

1. **Índice de concurrencia:** Se produce en un grado reducido, generando un escaso flujo de modificaciones; o por el contrario se da de forma intensiva generando situaciones problemáticas en la planificación o en la distribución del trabajo.
2. **Gestión de la concurrencia:** La concurrencia puede producirse en un proyecto de forma planificada o inducida por las circunstancias. En ambos casos resulta muy importante la labor de gestión del proyecto para tratarla de forma adecuada con el mayor beneficio, o el menor prejuicio a los planes y calidad del proyecto.

Modificadores de los modelos: Componentes comerciales y reutilización

Resulta muy habitual integrar en el desarrollo de un sistema partes "preconstruidas": Pueden ser componentes comerciales o la reutilización de componentes o marcos ya desarrollados por otros sistemas.

Situaciones de surgimiento de esta tendencia

1. Presión competitiva para reducir agenda y costes.
2. Incremento de la complejidad y estandarización de los entornos de operación.
3. Aparición de líneas de producción en las que se desarrollan múltiples sistemas de software reutilizando partes de diseño y componentes.

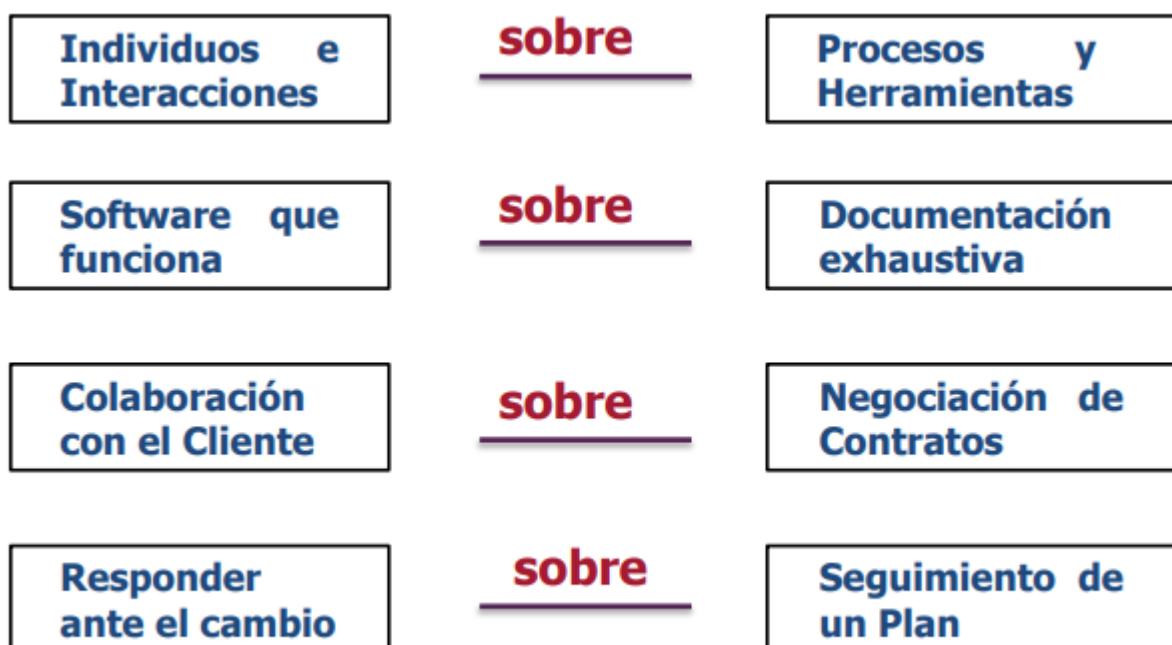
El uso de componentes o partes ya desarrolladas tienen implicaciones en el ciclo de desarrollo, diferentes según las circunstancias.

Si un proyecto va a delegar funcionalidades críticas en un componente comercial, que no ha empleado previamente la organización desarrolladora, es posible que incorpore en su ciclo de desarrollo una fase de pruebas de ese componente, antes del diseño, para obtener la certeza previa de que el componente se comporta como se espera.

Modelos de Procesos Ágiles

Se caracterizan por ser la negación de las características de los ciclos de vida convencionales.

Se basan en el **Manifiesto Ágil de Software**, manifiesto centrado en 4 valores:



Principios en los que se basa el manifiesto

1. Satisfacer al cliente a través de entrega por valor.
2. Aceptamos que los requisitos cambien.
3. Entregamos software funcional frecuente.
4. La gente del negocio y los desarrolladores trabajamos juntos diariamente.
5. Los proyectos se hacen en entornos de individuos motivados.
6. Comunicaciones cara a cara.
7. El software funcionando es la medida principal de progreso.
8. Promover un paso sostenido.
9. Excelente técnica y buen diseño.
10. Simplicidad es esencial.
11. Equipos auto-organizados.
12. Inspeccionar y adaptar.

Situaciones clave de aplicación del proceso ágil

1. Resulta difícil predecir los requisitos del software que persistirán y que cambiarán según los cambios de prioridades del cliente.
2. Es conveniente que el diseño y la construcción de software no son predecibles respecto a la planificación.

La Ingeniería del Software Ágil combina una filosofía y un conjunto de directrices de desarrollo.

Directrices de desarrollo de la Ingeniería del Software Ágil

1. Busca la satisfacción del cliente y la entrega temprana del software incremental.
2. Equipos de proyectos pequeños y con alta motivación.

3. Métodos informales.
4. Mismos productos de trabajo
5. Simplicidad general del desarrollo.

Razones de la importancia del modelo

1. En estos momentos los sistemas basados en computadoras y los productos de software están acelerados y en un cambio continuo.
2. La ingeniería del software ágil representa una alternativa a la ingeniería convencional para ciertas clases de software y para ciertos tipos de proyectos.

Modelos de Procesos Ágiles: Modelo de Desarrollo SCRUM

Es un proceso ágil que nos permite centrarnos en ofrecer el más alto valor de negocio en el menor tiempo.

Nos permite rápidamente y en repetidas ocasiones inspeccionar software real de trabajo (cada 2 semanas o cada mes).

El negocio fija las prioridades. Los equipos se auto-organizan a fin de determinar la mejor manera de entregar las funcionalidades de más alta prioridad.

Cada 2 semanas o cada mes, cualquiera puede ver el software real funcionando y decidir si liberarlo o seguir mejorándolo en otro sprint.

Características del modelo SCRUM

1. Equipos auto-organizados.
2. El producto avanza en "Sprints" de 2 semanas a un mes de duración.
3. Los requisitos son capturados como elementos de una lista de "Product Backlog".
4. No hay prácticas de ingenierías prescritas.
5. Utiliza normas generativas para crear un entorno ágil para la entrega de proyectos.
6. Es uno de los procesos ágiles.

7. Cumple el manifiesto ágil.

Características de un sprint

1. Duración típica de 2 a 4 semanas o a lo sumo un mes de calendario.
2. Duración constante conduce a un mejor ritmo.
3. Producto es diseñado, codificado y testeado durante el Sprint.

SCRUM FrameWork



Creación del Modelo de Ciclo de Vida

Pasos a realizar al iniciar el proyecto

Al iniciar el proyecto, el responsable de la arquitectura de procesos debe realizar los siguientes pasos:

1. Análisis de las circunstancias ambientales del proyecto.
2. Diseño del modelo específico de ciclo de vida para el proyecto (sobre las bases de los diseños más apropiados, para el desarrollo y evolución del sistema de software).
3. Mapeo de actividades sobre el modelo.
4. Desarrollo del plan para la gestión del ciclo de vida del proyecto.

Aspectos a considerar

1. Posibilidad de descomposición del sistema en subsistemas de software, con agendas y entregas diferenciadas.
2. Estabilidad esperada de los requisitos.
3. Novedad del proceso o procesos gestionados por el sistema en el entorno del cliente.
4. Criticidad de las agendas y presupuestos.
5. Grado de complejidad de la interfaz de operación, criticidad de la usabilidad.
6. Grado de conocimiento y familiaridad con el entorno de desarrollo, componentes externos empleados, etc.

Capítulo 3

PLANIFICACIÓN DE SISTEMAS SOFTWARE

Introducción

La planificación de un sistema software se encuentra enmarcada dentro de la Gestión de Proyectos.

La Gestión de Proyectos de Software es una parte esencial de la Ingeniería del Software. Aunque la buena gestión no garantiza el éxito del proyecto, la mala gestión generalmente lleva al fracaso del proyecto, produciendo entregas fuera de plazo, costes mayores que los estimados e incumplimiento de los requisitos.

Definición de un proyecto

Un proyecto es un conjunto de etapas, actividades y tareas que tiene como finalidad alcanzar un objetivo que implica un trabajo no inmediato, a un plazo relativamente largo.

Características de un proyecto

1. Implica un principio y un final.
2. Utiliza diversos recursos finitos y cuenta con un presupuesto.
3. Tiene actividades únicas y esencialmente objetivas.

4. Tiene un objetivo.
5. Requiere un jefe de proyecto y personal de desarrollo cuyos roles y estructura deben definirse y desarrollarse.
6. Tiene que planificarse.
7. Debe medir su progreso frente al plan.
8. Suele coexistir con otros proyectos y competir por los recursos.

Hay que considerar que un proyecto es la división en trabajos más sencillos lo que permite al personal del proyecto dominar la complejidad del proceso para desarrollar el software.

Plan del proyecto

Elementos del plan de proyecto

1. Resumen del proyecto que pueda ser comprendido por cualquier persona. Debe indicar los productos entregables de forma que, cuando se produzcan se pueda comprobar que se ajustan al plan.
2. Una lista de los hitos alcanzables.
3. Procedimientos y estándares que se van a aplicar.
4. Especificación del proceso de revisión que determine quién, cómo y cuándo se puede revisar la planificación del proyecto y con qué objeto.
5. Un diagrama de descomposición del trabajo (WBS).
6. Lista del personal del proyecto y su asignación en relación al WBS.
7. Red de actividades que muestre la secuencia de tareas en el tiempo y su relación entre ellas.
8. Los responsables de todas y cada una de las actividades.

9. Presupuestos de esfuerzos y costes, y los calendarios y plazos para todas las actividades.

Planificación temporal

Principios de la planificación temporal

1. Descomposición del proyecto en un número manejable de tareas.
2. Interdependencia: Se deben determinar las dependencias de cada tarea.
3. Asignación de campo: A cada tarea se le debe asignar un cierto número de unidades de trabajo, una fecha de inicio y una fecha de finalización.
4. Validación del esfuerzo: A medida que se realiza la asignación de tiempo, el gestor del proyecto se tiene que asegurar de que los técnicos necesarios estarán disponible en cada momento.
5. Responsabilidades definidas: Cada tarea que se programe debe asignarse a un miembro específico del proyecto.
6. Resultados definidos: El resultado de cada tarea, normalmente un producto, deberá estar definido. Los productos se combinan generalmente en entregas.
7. Hitos definidos: Todas las tareas o grupos de tareas deben asociarse con algún hito del proyecto. Se considera un hito cuando se ha revisado la calidad de uno o más productos y se han aceptado.

Descomposición en tareas

WBS es un método para representar de forma jerárquica los componentes de un proceso o producto.

Es utilizado para documentar la descomposición de un proceso, un producto o de forma híbrida.

Paralelismo de tareas

Cuando más de una persona está involucrada en un proyecto, puede que varias actividades

puedan llevarse a cabo en paralelo.

Debido a que estas tareas pueden realizarse de forma asíncrona, el planificador debe determinar la dependencia entre las tareas para asegurar el progreso continuo del proyecto hasta su terminación.

Cada referencia o meta parcial del proyecto es colocada a intervalos regulares y se alcanza una vez la documentación ha sido revisada satisfactoriamente.

Distribución de esfuerzos

Una distribución de esfuerzos recomendada es la que se muestra que potencia la fase de análisis y diseño y las pruebas del software.

Esta distribución se debe de tomar como una directiva, siendo las características del proyecto las que marquen la distribución de esfuerzos que se deben asignar.

Pasos en la Planificación Temporal

1. PUNTO DE PARTIDA

1. Disponemos del WBS en tareas del proyecto.
2. Disponemos de las fichas de cada tarea con los recursos y tiempos asignados.

2. CREACIÓN DE UN CALENDARIO ACEPTABLE

1. Creación del calendario y camino crítico.
 - a) Ordenación de las tareas.
 - b) Creación del calendario.
2. Revisión y ajuste del calendario
 - a) En función del uso de recursos.
 - b) Según las necesidades del usuario.
3. Aceptación generalizada del plan

3. ORDENACIÓN DE TAREAS

Se deben identificar y documentar las dependencias: Restricciones, supuestos, dependencias obligatorias, dependencias discretionales y dependencias externas.

Para la ordenación de las tareas, se tienen que organizar las tareas en el orden técnico de ejecución.

Para establecer las precedencias nos podemos plantear las siguientes cuestiones:

¿Qué se puede hacer ahora?, ¿Qué debe haberse hecho antes de esto?, ¿Qué puede hacerse a la vez?, ¿Qué debe seguir a lo que hacemos ahora?

Añadiremos a cada ficha de tarea la lista de tareas precedentes.

4. RESTRICCIONES

Son los factores que limitan las opciones del equipo de desarrollo.

Son impuestas por el cliente o la dirección de la empresa desarrolladora.

5. SUPUESTOS

Son factores que se consideran verdaderos durante la planificación.

Están directamente relacionados con los riesgos del proyecto.

6. DEPENDENCIAS OBLIGATORIAS

Son las inherentes a la naturaleza del trabajo (aspectos técnicos).

Se suelen deber a la necesidad de disponer de un entrega que es punto de partida de la tarea.

7. DEPENDENCIAS DISCRETIONALES

Son dependencias definidas por el equipo del proyecto. Por ello, hay que ser cautelosos, pues pueden condicionar la programación del proyecto en el futuro.

Se basan en las mejores prácticas, se prefiere una secuencia (será más fácil de controlar) y tienen limitaciones en la asignación del personal.

8. DEPENDENCIAS EXTERNAS

Vienen impuestas desde el exterior y se refieren a la interdependencia con otros proyectos o con empresas externas o contratos (con los que no podemos ejercer ninguna presión).

Una actividad no puede comenzar hasta que no se dispone de un producto ajeno.

Métodos de Planificación Temporal

1. MÉTODOS PARA REPRESENTAR VISUALMENTE LA SECUENCIA DE TAREAS

1. Diagrama de Gantt.
2. Diagrama de precedencias o actividades.
3. Diagrama de flechas

Las características comunes de los métodos de planificación son:

1. Identificar las tareas que determinan la duración del proyecto.
2. Establecimiento de estimaciones de tiempo para tareas individuales de acuerdo con modelos estadísticos.
3. Calcular los tiempos límite que definen un espacio temporal para cada tarea.

Método de Gantt

Se representa en un cuadro de doble entrada:

1. Eje horizontal: Tiempo.
2. Eje vertical: Tareas.
3. Cada tarea se representa como un rectángulo situado a la altura de la tarea y que va desde el comienzo a la finalización de la tarea.

Inconvenientes del diagrama de Gantt

1. No muestra explícitamente la relación entre tareas.

2. En proyectos con muchas tareas es complicado de crear.

Ventajas del diagrama de Gantt

1. Es fácil de entender por todo el mundo.
2. Se puede aplicar para representar la utilización de recursos.

Diagrama de precedencias o actividades

Es un modelo que introduce tiempo y precedencias, permitiendo calcular la duración total del proyecto.

Se consideran las actividades y las relaciones de precedencia entre ellas.

Una **actividad** es una parte de un proyecto que se lleva a cabo durante un período de tiempo.

Para cada actividad se deben describir sus precedentes, su duración y su producto.

Un **hito** es un punto en el tiempo que marca el inicio o el fin de una actividad.

Modelo simplificador

Contiene duraciones fijas y no permite indicar comienzo de una actividad en un punto de la ejecución de otra (se soluciona dividiendo las actividades).

Se trata de un grafo totalmente ordenado, donde las tareas se representan como nodos y las relaciones entre tareas como los arcos.

Todos los nodos tienen el mismo tamaño y pueden contener mucha información sobre la tarea.

Los arcos van desde la tarea antecesora a la sucesora, indicándolo con una flecha.

Al utilizar sistemas informáticos para generar los diagramas, se pueden establecer relaciones del tipo: Fin-Comienzo y Comienzo-Fin, Comienzo-Comienzo y Fin-Fin.

Elementos del diagrama de actividades

Etiqueta actividad		Duración
Inicio temprano	DESCRIPCIÓN DE LA ACTIVIDAD	Final temprano
Inicio tardío		Final tardío
Máximo tiempo disponible		Holgura

1. **Descripción de la actividad:** Nombre dado a la actividad.
2. **Etiqueta actividad:** Número que identifica a la actividad.
3. **Duración:** Tiempo que calculamos que se tardará en completar la tarea.
4. **Inicio temprano:** Fecha más temprana en que puede comenzar la tarea.
5. **Final temprano:** Fecha más temprana en que puede finalizar la tarea.
6. **Inicio tardío:** Fecha más retrasada en la que se puede comenzar sin que afecte a la fecha de terminación del proyecto.
7. **Final tardío:** Fecha más retrasada en la que puede comenzar la tarea sin que afecte a la fecha final del proyecto.
8. **Máximo tiempo disponible:** Tiempo máximo que puede durar una tarea en caso de comenzar en su Inicio temprano y concluir en su Final tardío.
9. **Holgura:** Tiempo que disponemos para jugar con el inicio de la tarea, sin afectar al proyecto.

Cálculo de las fechas para cada tarea del proyecto

1. Partimos del diagrama de precedencias.
2. Asignamos como inicio temprano 0 a todas las tareas sin predecesor.
3. El final temprano de cada tarea es el inicio temprano más su duración.

4. Si la tarea tiene predecesoras, y todas estas tienen calculado su final temprano, asignamos como inicio temprano el máximo de todos ellos.
5. Obtenemos la fecha final del proyecto: Partimos de la máxima fecha de final temprano.
6. A todas las tareas que no tengan sucesoras se le asigna esta fecha como final tardío.
7. El inicio tardío se calcula restando al final tardío la duración.
8. Aquellas tareas con sucesoras, se les asigna como final tardío el mínimo de los inicios tardíos de estas.
9. Máximo tiempo disponible = Final tardío - Inicio temprano.
10. Holgura = Máximo tiempo disponible - Duración.

Camino crítico

Se define como el conjunto de tareas con holgura 0, de modo que existe sólo si la duración es mínima.

1. Parte de una tarea sin predecesoras, atraviesa el grafo por tareas con holgura cero y termina en una tarea sin sucesoras.
2. Cuando una tarea del camino crítico se retrasa, se retrasa el proyecto.
3. A las tareas del camino crítico se les llama **tareas críticas** y un retraso en cualquiera de ellas lleva a un retraso del final del proyecto

Diagrama de flechas o PERT

Es una representación dual a la anterior, donde las tareas se representan como arcos.

Los nodos son sucesos puntuales en el tiempo, muestran que se ha alcanzado un estado, al concluir todas las tareas que llegan a él.

Aparecen actividades ficticias para asociar estados parciales.

Los modelos formales para el cálculo de calendarios se basa en él.

Parece menos intuitivo que los otros grafos, debido fundamentalmente al uso de tareas ficticias. Es muy usado en la construcción

Es muy útil el poder ver sólo las tareas que hay asignadas en cada recurso, para comunicar a los participantes el uso de un recurso compartido, verificar que se utilizan de forma equilibrada y verificar que ningún recurso se pretende utilizar más de lo posible.

Se usa el Gantt y el de Cargas.

Análisis del Riesgo

Tiene que ver con la identificación de los riesgos y los planes para minimizar sus efectos en el proyecto.

Categorías de riesgos

1. **Riesgos del proyecto:** Identifican problemas de presupuesto, de agenda, del personal, de los recursos, del cliente y sus requisitos, del tamaño y complejidad del proyecto.

Afectan el coste y duración del proyecto.

2. **Riesgos del producto:** Problemas en el diseño, implementación, interfaz, incertidumbre técnica, problemas de obsolescencia o de utilización de tecnología punta.

Afectan a la calidad del software resultante.

3. **Riesgos del negocio:** Cambio en la dirección, cambios de estrategia del negocio, cambios del mercado, pérdidas en la empresa.

Afectan al equipo de desarrollo y a la realización del proyecto en sí.

Proceso de identificación del riesgo

1. IDENTIFICACIÓN DEL RIESGO

Se identifican los posibles riesgos del proyecto, del producto y del negocio.

Consiste en determinar para cada tipo de situaciones cuáles son los posibles riesgos que pueden afectar al desarrollo del proyecto.

2. ESTIMACIÓN Y EVALUACIÓN DEL RIESGO

Determina la probabilidad y las consecuencias de cada riesgo, y la probabilidad e impacto de cada riesgo identificado.

La probabilidad puede ser expresada de forma cuantitativa o cualitativa: Muy bajo, bajo, moderado, alto o muy alto.

El impacto tiene que ver con sus consecuencias y la duración de las mismas: Catastrófica, seria, tolerable o insignificante

3. PLANIFICACIÓN DEL RIESGO

Se traza un plan para evitar o minimizar la ocurrencia de un riesgo.

Se considera cada riesgo con probabilidad alta/muy alta a partir de un impacto tolerable y moderada a partir de un impacto serio y desarrollar estrategias para gestionar dicho riesgo.

Se diseñan estrategias con tareas específicas para evitar en lo posible la ocurrencia del riesgo.

Se diseñan estrategias con tareas específicas para aplicar a posteriori de la ocurrencia de un riesgo y que tengan como fin minimizar sus consecuencias.

Todas estas tareas se engloban y detallan en el **Plan de Gestión y Supervisión del riesgo**

4. CONTROL DE RIESGO

Controla la ocurrencia de riesgos a lo largo del proyecto.

Asegura el cumplimiento de las tareas para evitar el riesgo y para minimizar su impacto en caso de que ocurra.

Revisa periódicamente cada uno de los riesgos identificados para decidir si su probabilidad de ocurrencia ha aumentado o disminuido.

Revisa también si las consecuencias del riesgo cambian.

Los riesgos considerados deben ser discutidos en las reuniones periódicas que discuten el progreso del proyecto.

Planificación Organizativa

Actividades del jefe del proyecto

1. Resolución de problemas usando el personal disponible.
2. Motivación del personal que forma parte del proyecto.
3. Planificación de las tareas a realizar por cada miembro del proyecto.
4. Estimación del trabajo que puede realizar el personal.
5. Control de las actividades del personal.
6. Organización del modo de trabajo.

Dentro del equipo de desarrollo las comunicaciones son necesarias e inevitables para un grupo de trabajo eficiente. Sin embargo, son improductivas pues mientras dura la comunicación el individuo no está realizando su función.

Por ello, hay que minimizar y acortar las reuniones de comunicación.

Factores que influyen a la comunicación en grupo

1. Tamaño del grupo
2. Estructura del grupo
3. Personalidades implicadas y su categoría profesional
4. Ambiente físico de trabajo

Cuanto mayor es el grupo, mayor es el número de enlaces de comunicación entre sus miembros.

Para disminuirlas, debemos estructurar las comunicaciones de manera que todas pasen por un coordinador central dentro de cada grupo de trabajo, y establecer grupos de comunicación y el mínimo de comunicaciones entre grupos.

Los grupos ideales son de entre 2 y 8 personas:

1. Disminuyen los problemas de comunicación
2. Pueden mantener la continuidad del proyecto si un miembro abandona el grupo
3. El trabajo desarrollado se considera responsabilidad del grupo
4. Existe un mayor consenso hacia como abordar las tareas

Estimación de costes

Se define como la predicción del personal, esfuerzo, costes y planificación requerida para realizar todas las actividades y construir todos los productos asociados con el proyecto.

La **estimación** se define como el proceso que proporciona un valor a un conjunto de variables para la realización de un trabajo, dentro de un rango aceptable de tolerancia.

Uno de los factores críticos de la estimación es determinar su exactitud.

La estimación y la planificación son actividades relacionadas pero difieren en su alcance y propósito.

1. La Estimación normalmente está orientada al proyecto en su conjunto
2. La planificación está dirigida a los individuos.

Una diferencia técnica entre las herramientas de planificación y estimación es que estas últimas son normalmente sistemas expertos, construidos a partir de reglas derivadas de muchos proyectos; mientras que las herramientas de planificación son herramientas utilizadas por expertos.

Los objetivos de la estimación de proyectos son reducir los costes e incrementar los niveles de servicio y calidad.

Midiendo determinados aspectos del proceso de software puede dar una visión de alto nivel de lo que sucederá en el desarrollo:

1. Mediciones de procesos anteriores permiten realizar predicciones sobre los actuales.

2. Mediciones de atributos de proceso en fases iniciales del desarrollo permiten realizar predicciones sobre fases posteriores.

Las predicciones de proceso conducen la toma de decisiones antes del comienzo del desarrollo, durante el proceso de desarrollo, durante la transición del producto al cliente y a lo largo de la fase de mantenimiento.

Métricas del Software

1. Métricas de productividad: Se centran en el rendimiento de procesos de ingeniería de software.
2. Métricas de calidad: Conveniencia del software para su utilización (como se ajusta a los requisitos).
3. Métricas técnicas: Se centran en las características del software.
4. Métricas orientadas al tamaño: Medidas directas del software y proceso por el que se desarrolla. Calcula la productividad y calidad del software.
5. Métricas orientadas a la función: Medidas indirectas del software y del proceso por el que se desarrolla.

Métricas orientadas al tamaño

1. **Productividad** = $\frac{KLDC}{Personas-Mes}$
2. **Coste** = $\frac{Inversión}{KLDC}$
3. **Calidad** = $\frac{Errores}{KLDC}$
4. **Documentación** = $\frac{nPaginas}{KLDC}$

Métricas orientadas a la función

Características del dominio de la Información

1. **Número de entradas de Usuario**: Proporciona diferentes datos orientados a la aplicación, sin considerar peticiones.

2. **Número de salidas de Usuario:** Proporciona información orientada a la aplicación.
3. **Número de Peticiones al Usuario:** Entrada interactiva que produce alguna respuesta del software inmediata de forma interactiva.
4. **Número de archivos lógicos:** Pueden ser parte de una gran base de datos o archivos independientes.
5. **Número de interfaces externas:** Flujo legible por la máquina que transfieren información desde/hacia otros sistemas.

Fórmula de cálculo de los Puntos de Función

$PFA = PF * [0,65 + 0,01 * \sum Fi]$, siendo PFA los puntos de función ajustados, PF los puntos de función sin ajustar y Fi el conjunto de factores de complejidad.

Métodos de estimación

1. JUICIO EXPERTO

1. **Puro:** Un experto estudia las especificaciones y hace su estimación, de modo que se basa fundamentalmente en los conocimientos del experto. Si desaparece el experto, la empresa deja de estimar.
2. **Delphi:** Un grupo de personas son informadas y tratan de adivinar el coste del desarrollo en esfuerzo y duración. Las estimaciones en grupo suelen ser mejores que las individuales.

2. ANALOGÍA

Consiste en comparar las especificaciones de un proyecto con las de otros proyectos. La estimación inicial se ajusta dependiendo de las diferencias entre los proyectos pasado-nuevo.

3. BASADO EN LOS COMPONENTES DEL PRODUCTO O PROCESO DE DESARROLLO

1. **Bottom-up:** Se descompone el proyecto en las unidades lo menores posibles, se

estima cada unidad y se calcula el coste total.

2. **Top-Down:** Se ve todo el proyecto, se descompone en grandes bloques/fases y se estima el coste de cada componente.

4. DISTRIBUCIÓN DE LA UTILIZACIÓN DE RECURSOS EN EL CICLO DE VIDA

Usualmente las organizaciones tienen una estructura de costes similar entre proyectos. Si ya hemos realizado fases del proyecto, los costes deben distribuirse de manera proporcional.

5. MÉTODOS ALGORÍTMICOS

Producen una estimación de coste del proyecto.

6. MÉTODOS DE ESTIMACIÓN

1. **Modelo básico:** Calcula el esfuerzo de Desarrollo en función del tamaño del programa, expresado en líneas estimadas de código.
2. **Modelo intermedio:** Calcula el esfuerzo de desarrollo en función del tamaño del programa y de conductores de coste.
3. **Modelo Avanzado:** Incorpora todas las características del modelo intermedio y lleva a cabo una evaluación del impacto de los conductores de coste en cada fase del proceso de Ingeniería de Software.

Tipo de Proyecto	Requisitos	Tamaño del proyecto	Complejidad	Nº de personas	Experiencia
Modo orgánico	Poco rígidos	Pequeño (<50KLDC)	Pequeña	Pocas	Mucha
Modo semi-acoplado o semi-rígido	Poco/Medio	Medio [50,300] KLDC	Medio	Medio	Medio
Modo empotrado o rígido	Alto	Grande (>300KLDC)	Alta	Alta	Poca

Básico

MODO	ESFUERZO (personas-mes)	TIEMPO DE DESARROLLO (meses)
Orgánico	ESF = 2,4 × (KLDC)^{1,05}	TDES= 2,5 × (ESF)^{0,38}
Moderado	ESF= 3,0 × (KLDC)^{1,12}	TDES= 2,5 × (ESF)^{0,35}
Embebido	ESF = 3,6 × (KLDC)^{1,20}	TDES= 2,5 × (ESF)^{0,32}

Intermedio

MODO	ESFUERZO (personas-mes)	TIEMPO DE DESARROLLO (meses)
Orgánico	ESF = 3,2 × (KLDC)^{1,05} × FEC	TDES= 2,5 × (ESF)^{0,38}
Moderado	ESF= 3,0 × (KLDC)^{1,12} × FEC	TDES= 2,5 × (ESF)^{0,35}
Embebido	ESF = 2,8 × (KLDC)^{1,20} × FEC	TDES= 2,5 × (ESF)^{0,32}

- Se consideran un conjunto de atributos denominados conductores de conste

$$\mathbf{FEC} = \prod \mathbf{FE}_i$$

7. CONDUCTORES DE COSTE

1. **Producto:** Requerimientos de confiabilidad + Tamaño de la base de datos + Complejidad.
2. **Computador usado:** Restricciones de tiempo de ejecución + Restricciones de memoria principal + Velocidad de cambio de los medios de cómputo + Tiempo respuesta del computador.
3. **Personal:** Capacidad de los analistas + Experiencia de los analistas + Capacidad de los programadores + Experiencia en el sistema operativo + Experiencia en el lenguaje de programación.
4. **Proyecto:** Uso de técnicas modernas de programación + Uso de herramientas de software + Requisitos de planificación.

Capítulo 4

ANÁLISIS DE REQUISITOS

Definición de requisitos

Componen la primera fase del ciclo de vida del software en el que se produce una especificación a partir de ideas informales.

Debe obtenerse y documentarse: Requisitos de información, Requisitos funcionales, Requisitos no funcionales, Criterios de medición del grado de su consecución.

El proceso de desarrollo de dicha especificación de requisitos se conoce como **ingeniería de requisitos**.

Se define un **requisito** como un proceso de descubrimiento y comunicación de las necesidades de clientes-usuarios y la gestión de cambios en dichas necesidades.

Según la IEEE, se define un **requisito** como una representación en forma de documento de una condición o capacidad para resolver un problema o lograr un objetivo, o para satisfacer un contrato, norma, especificación u otro documento formal.

¿Qué son los requisitos?

Describen los servicios que debe proporcionar el sistema y sus restricciones operativas.

Pueden ser una simple declaración abstracta de alto nivel o una definición detallada y formal de una función del sistema.

Es necesario separar entre niveles de descripción: Requisitos de usuario y Requisitos de sistema.

La comunicación es uno de los aspectos más destacables en la ingeniería de requisitos, haciendo de esta ingeniería una disciplina compleja al intervenir el factor humano.

¿Qué describe un requisito?

Describe una utilidad para el usuario, una propiedad general del sistema, una restricción general del sistema, cómo llevar a cabo cierto cálculo o una restricción sobre el desarrollo del sistema.

Tipos de requisitos

1. **Requisitos de Usuario:** Descripciones de alto nivel.
2. Requisitos del sistema: Descripción detallada del sistema.

a) Requisitos Funcionales

b) Requisitos no funcionales: Requisitos del producto, requisitos organizacionales o requisitos externos.

Un **requisito de usuario** es una descripción de lo que se espera que el sistema proporcione y las restricciones bajo

Los **requisitos del sistema** establecen con detalle las funciones, servicios y restricciones operativas del sistema.

Deben ser precisos + Definir exactamente lo que se va a implementar + puede ser parte del contrato comprador-desarrollador.

Se emplean dos **tipos de visores apropiados:**

1. **Intención del usuario:** Visores especiales para cada tipo de documento.
2. **Interpretación del desarrollador:** Visor de texto que muestra el contenido del documento.

Principalmente, lo deseable es que los requisitos fueran:

1. **Completos**: Recogen la descripción de todos los servicios esperados por el usuario.
2. **Consistentes**: Sin contradicciones entre estas especificaciones.

Los **requisitos no funcionales** son restricciones de los servicios ofrecidos por el sistema, los cuales se aplican a la totalidad del sistema.

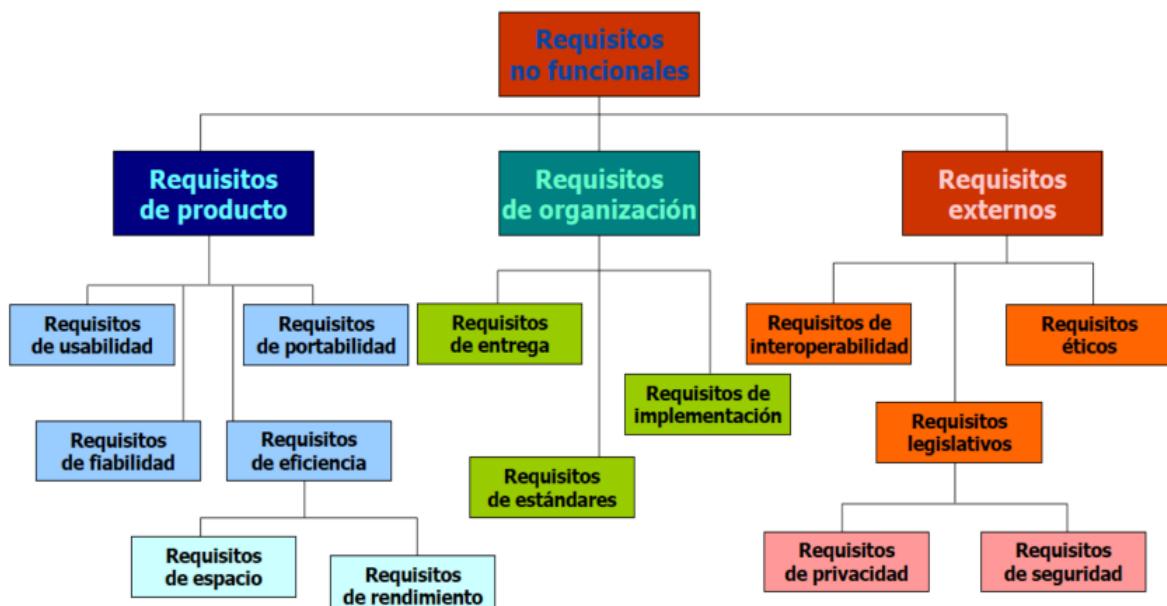
Surgen de las necesidades del usuario, definiendo propiedades y restricciones del sistema.

También pueden referirse al proceso de desarrollo: Uso de una metodología CASE, lenguaje de programación o metodología concreta.

Los **requisitos del producto** especifican el comportamiento del producto.

Los **requisitos organizacionales** derivan de políticas y procedimientos existentes en la organización del cliente y del desarrollador.

Los **requisitos externos** se derivan de factores externos al sistema y a su proceso de desarrollo.



Las metas y requisitos puede ser complicado de precisar cuando se trabaja con requisitos no funcionales.

Se define una **meta** como un propósito general del usuario.

Se define un **requisito no funcional no verificable** como una frase que implica alguna medida objetiva.

Las metas son útiles para los desarrollados porque transmiten los deseos del usuario al sistema.

Requisitos de usuario

Deben describir requisitos funcionales y no funcionales de modo que sean entendibles por usuarios del sistema que carecen de conocimientos técnicos.

Se describen usando un lenguaje natural, tablas y diagramas.

Sin embargo, presenta un problema de falta de claridad.

Se acostumbra a mezclar los requisitos funcionales y no funcionales.

Se utiliza la **fusión de requisitos**: Agrupan varios requisitos en la definición de uno solo.

Requisitos de interfaz

Describen el formato con el que la aplicación se comunica con su entorno.

Gestión de requisitos

Los requisitos deben ser:

1. Incluidos en un catálogo
2. Analizados, con el objetivo de detectar inconsistencias, ambigüedades, duplicidad o escasez de información.
3. Validados por los clientes/usuarios.

Se deben realizar actividades que conformen que los requisitos se cumplen.

Características de un requisito

1. Identificador
2. Autor
3. Tipo de requisito
4. Descripción
5. Prioridad: Alta, media o baja
6. Estado: Propuesto, aprobado o incorporado
7. Fecha de creación/Revisión

Se deben organizar: Por subsistemas, por tipo o jerárquicamente.

Actividades a realizar en la Especificación de Requisitos

Principales actividades

1. Extracción de requisitos: Proceso por el cuál los clientes descubren, revelan y comprenden los requisitos que desean.
Se emplean técnicas de recogida de la información como la observación, entrevistas o grupos de trabajo.
2. Análisis de requisitos: Proceso de razonamiento sobre los requisitos obtenidos en la etapa anterior, detectando y resolviendo posibles inconsistencias o conflictos, coordinando los requisitos relacionados
3. Especificación de requisitos: Proceso de redacción o registro de los requisitos.
Para el análisis y especificación de requisitos existen técnicas gráficas como los Diagramas de Flujo de datos y los Casos de Uso
4. Validación de requisitos: Proceso de confirmación por parte de los usuarios/clients de los requisitos especificados en el que se comprueba su validez, consistencia, etc.
Suele recurrirse listas de comprobación junto con técnicas de revisión y comprobación.

Especificación de requisitos

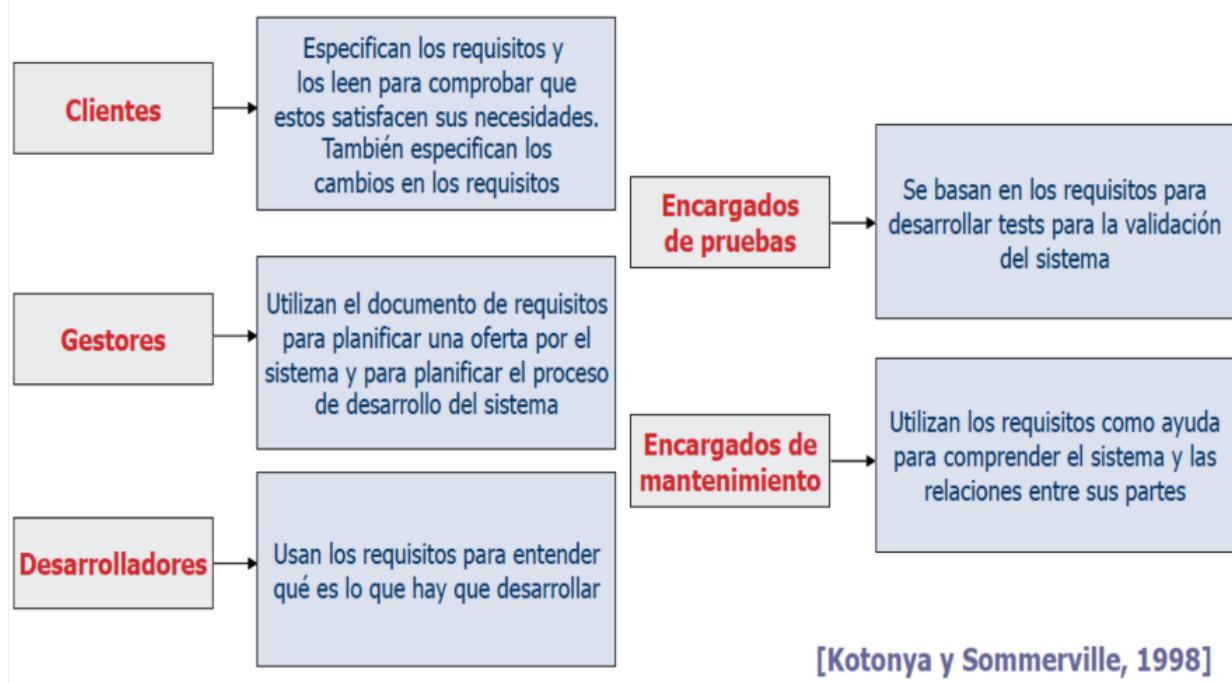
Es un documento que define, de forma completa, precisa y verificable los requisitos que debe cumplir el sistema tanto funcionales como no funcionales, así como las restricciones aplicables al diseño.

Características deseable en una buena especificación de requisitos

1. No ambigua
2. Completa
3. Fácil de verificar
4. Consistente y coherente
5. Clasificada por importancia o estabilidad
6. Fácil de modificar
7. Fácil identificación del origen y consecuencias de cada requisito.
8. Fácil utilización durante la fase de explotación y mantenimiento.

El objetivo es obtener un documento de especificación de requisitos de Software.

Usuarios de una ERS



Especificación de requisitos ágiles

Historias de usuario

Son utilizadas en los métodos ágiles para la especificación de requisitos.

Son una descripción breve de una funcionalidad software tal y como la percibe el usuario.

Describen lo que el cliente/usuario quiere que se implemente y se escriben en 1-2 frases utilizando un lenguaje común del usuario.

Cada historia de usuario debe ser limitada, pudiendo ser memorizable fácilmente y escribirse sobre una tarjeta.

Epics

Se define como una etiqueta que aplicamos a una historia grande, cuyo esfuerzo es demasiado grande para completarla de una sola vez o en un solo sprint.

Son superhistorias de usuario que se distinguen por su gran tamaño, teniendo una alta granularidad.

Suelen tener un flujo asociado por el cual se puede dividir en historias de usuario.

A medida que aumenta su prioridad y se acerca su implementación, el equipo la descomponen en historias de usuario con un tamaño más adecuado para ser gestionada con los principios y técnicas ágiles:

1. Estimación
2. Seguimiento cercano: Normalmente diario.

Temas

Representan una colección de epics y/o historias de usuario relacionados para describir un sistema o subsistema en su totalidad.

Tareas

Son resultado de la descomposición por parte del equipo de las historias de usuario en unidades de trabajo adecuadas para gestionar y seguir el avance de su ejecución.

En las metodologías ágiles, una pila del producto puede contener tanto historias de usuario como epics.

La pila del producto debe estar ordenada por prioridad y el nivel de detalle de cada elemento debe ir en relación a la posición del mismo dentro de la pila.

Información en una historia de usuario

Los **campos de una historia de usuario** son:

1. ID: Identificador de la historia de usuario, único para la funcionalidad o trabajo.
2. Descripción: Debe responder a 3 preguntas (¿quién se beneficia?, ¿qué se quiere? y ¿cuál es el beneficio?)
Como [rol del usuario], quiero [objetivo] para poder
3. Estimación: Esfuerzo necesario en tiempo ideal de implementación de la historia de usuario.

4. Prioridad: Permite determinar el orden en que las historias de usuario deben ser implementadas.

Capítulo 5

TÉCNICAS DE ESPECIFICACIÓN Y MODELADO

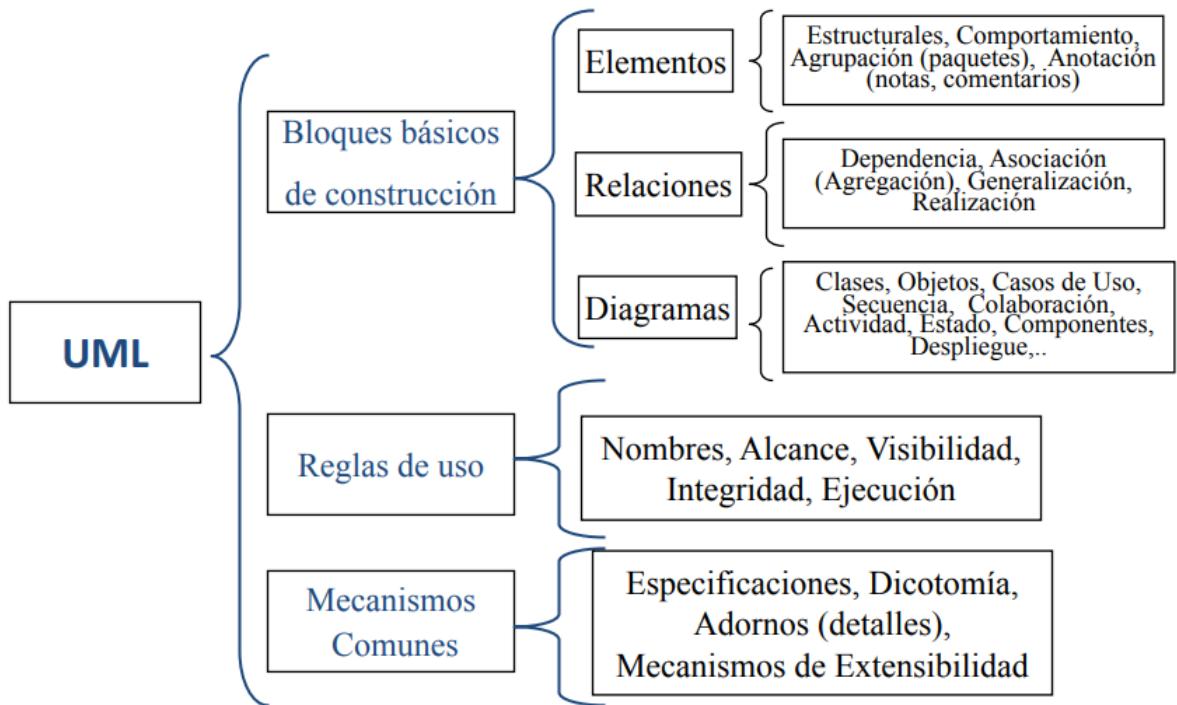
Técnicas orientada a objetos

UML es un lenguaje para visualizar, especificar, construir y documentar los artefactos (modelos) de un sistema que involucra una gran cantidad de software, desde una perspectiva orientada a objetos.

UML es una notación, no un proceso.

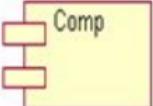
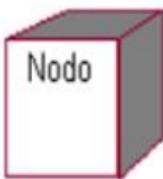
Modelo conceptual UML

1. Elementos: Bloques básicos de construcción orientada a objetos
2. Relaciones: Ligan los diferentes elementos entre sí.
3. Diagramas: Representación gráfica de un conjunto de elementos y sus relaciones entre sí.

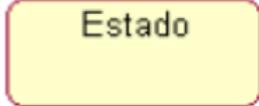
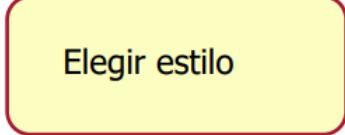


Elementos estructurales de UML

CLASE		Es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica.
CASO DE USO	RealizarCompra	Es una descripción de un conjunto de secuencias de acciones que un sistema ejecuta y que produce un resultado observable de interés para un actor particular.
INTERFAZ		Es una colección de operaciones que especifica un servicio de una clase o componente. Define un conjunto de especificaciones de operaciones, pero nunca sus implementaciones.

COMPONENTE		Es empaquetamiento físico de diferentes elementos lógicos como clases, interfaces.
ARTEFACTO		Parte física y reemplazable, representa el empaquetamiento físico de código fuente o información en tiempo de ejecución.
NODO		Un nodo es un elemento físico que existe en tiempo de ejecución, representando un recurso computacional que, por lo general, dispone de algo de memoria y con frecuencia capacidad de procesamiento.

Elementos de comportamiento UML

Interacción		Comprende un conjunto de mensajes que se intercambian entre un conjunto de objetos, para cumplir un objetivo específico.
Máquinas de estados		Especifica la secuencia de estados por los que pasa un objeto o una interacción, en respuesta a eventos.
Actividad		Es un comportamiento que especifica la secuencia de pasos que ejecuta un proceso computacional. Un paso de una actividad se denomina acción.

Elementos de agrupación de UML

Son las partes organizativas de los modelos UML

El elemento de agrupación principal son los **paquetes**: Mecanismos de propósito general para organizar elementos en grupos.

Un paquete es puramente conceptual (sólo existe en tiempo de desarrollo).

También podemos encontrar dentro de los paquetes los FrameWorks, modelos y subsistemas

Elementos de notación de UML

Son las partes explicativas de los modelos UML.

Hay un tipo principal denominado **nota**: Son comentarios que se pueden aplicar para describir, clarificar y hacer observaciones sobre cualquier elemento de un modelo.

Relaciones en UML

Permiten modelar el enlace entre diferentes elementos estructurales, mostrando información adicional como multiplicidad y nombres de roles.

UML maneja **4 clases de relaciones**:

Relación de dependencia: Una dependencia es una relación semántica entre dos clases en la cual un cambio de un elemento (independiente) puede afectar la semántica de otro (dependiente). Gráficamente una dependencia se representa como una línea discontinua, dirigida



Relación de asociación: Una asociación es una relación estructural que describe un conjunto de enlaces, las cuales representan conexiones a través de objetos. La **agregación** es una clase especial de asociación que representa una relación de estructura entre un conjunto y sus partes. Gráficamente una asociación se representa como una línea continua, que puede incluir multiplicidad

Relación de generalización: La generalización es una relación de especialización/generalización en la cual los objetos de un elemento especializado (hijos) son consistentes con los objetos de un elemento generalizable (el padre). De esta forma, los hijos comparten la estructura y comportamiento del padre. Gráficamente una generalización se representa como una flecha con la punta vacía dirigida al padre



Relación de Realización: Una realización es una relación semántica entre clasificadores, en donde un clasificador especifica un contrato que otro clasificador garantiza llevar a cabo. Se pueden encontrar realizaciones en dos partes: entre interfaces y las clases o componentes que las realizan, y entre casos de uso y las colaboraciones que los realizan. Gráficamente es la combinación de una dependencia y una generalización

Diagramas en UML

Un **diagrama** es la representación gráfica de un conjunto de elementos conectados entre sí.

Son en forma de grafos conectados donde los vértices representan elementos y los arcos relaciones.

Los diagramas sirven para visualizar un sistema desde diferentes perspectivas.

Un mismo elemento puede aparecer en varios diagramas, en ninguno o en sólo algunos.

Diagramas UML

1. Estructurales: Sirven para visualizar, especificar, construir y documentar los aspectos estáticos de un sistema.
2. De comportamiento: Sirven para visualizar, especificar, construir y documentar los aspectos dinámicos de un sistema.

Modelados UML

1. Modelado de Casos de Uso: Diagramas de Casos de Uso.
2. Modelado Estructural: Diagramas de clases.
3. Modelado de comportamiento: Diagramas de Interacción (secuencia y comunicación) y Diagramas de Estado.
4. Modelado de flujos de actividades: Diagramas de actividades.
5. Modelado Implementación: Diagramas de componentes

6. Modelado de despliegue: Diagramas de artefactos y Diagramas de Despliegue

Reglas UML

1. Los bloques de construcción UML no se pueden combinar de cualquier manera.
2. UML tiene un número de reglas que especifican a qué debe parecerse un modelo bien formado.
3. UML tiene reglas semánticas para nombres, alcance, visibilidad, integridad y ejecución.

Mecanismos comunes de UML

1. Especificaciones

Proporcionan una base semántica que incluye a todos los modelos de un sistema.

Cada elemento está relacionado con otros de manera consistente.

La notación gráfica de UML se utiliza para visualizar un modelo.

La especificación de UML se utiliza para enunciar los detalles del sistema.

2. Adornos

La mayoría de los elementos de UML tienen una única y clara notación gráfica que proporciona una representación visual de los aspectos más importantes del elemento.

A estas notaciones se les pueden agregar detalles que aclaren o complementen la información que se quiere mostrar (**adornos**).

3. Divisiones comunes

En el modelo orientado a objetos, se pueden ver las cosas desde la generalidad (abstracción) y/o lo particular (concreto).

Casi todos los objetos de construcción presentan esta posibilidad.

4. Mecanismos de extensibilidad

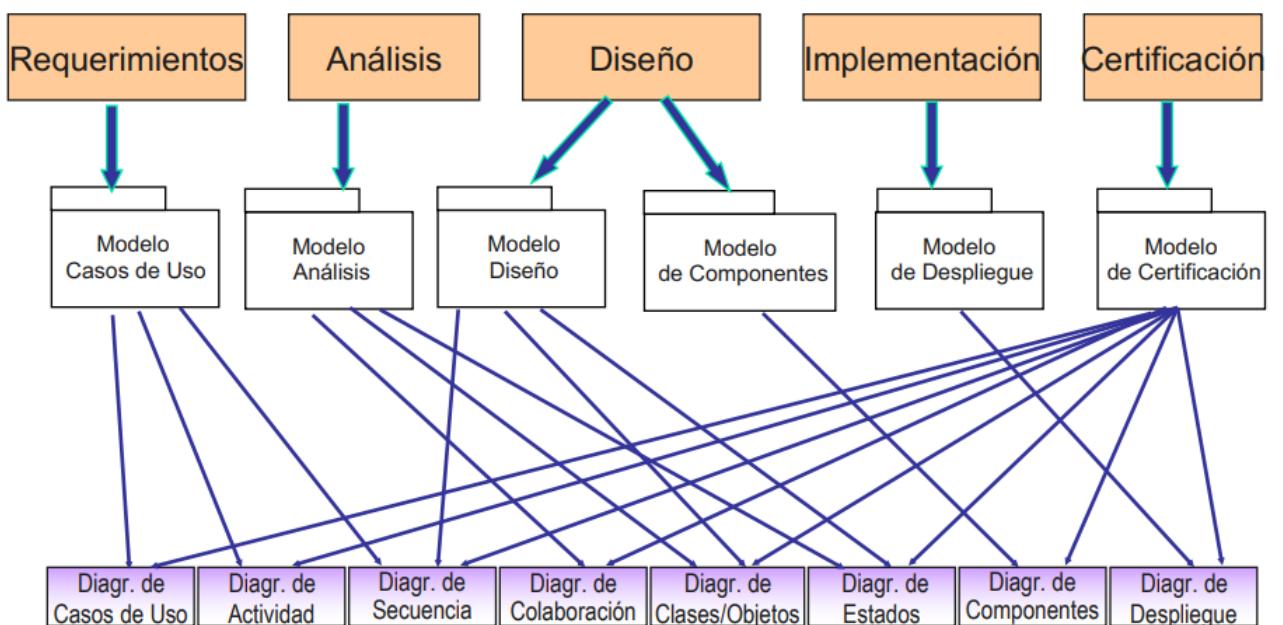
UML proporciona un lenguaje estándar para escribir planos de software, pero es posible que no sea suficiente para cubrir todos los matices de todos los modelos en todos los dominios y en todos los modelos.

Los mecanismos de extensibilidad permiten configurar y extender UML para las necesidades de un proyecto.

Los mecanismos que permiten extender el lenguaje UML son:

1. Estereotipos: Permiten crear nuevos tipos de bloques de construcción que deriven de los existentes, pero que sean específicos a un problema.
2. Valores etiquetados: Extiende las propiedades de un bloque de construcción, permitiendo añadir nueva información en la especificación de un elemento.
3. Restricciones: Extiende la semántica de un bloque de construcción de UML.

Metodología de Desarrollo en UML



En UML la arquitectura de un sistema con gran cantidad de software puede describirse mejor a través de 5 vistas interrelacionadas.

Cada vista es una proyección de la organización y la estructura del sistema, centrada en un aspecto particular de ese sistema.

Vistas interrelacionadas en UML



1. **Vista de casos de uso:** Describe el sistema tal y como es percibido por los usuarios finales, analistas y encargados de las pruebas.

Se utilizan para los aspectos estáticos los casos de uso y los aspectos dinámicos mediante diagramas de interacción, de estados y de actividades.

2. **Vista de diseño:** Comprende las clases interfaces y colaboraciones que forman el vocabulario del problema y su solución.

Soporta principalmente los requisitos funcionales del sistema.

Los aspectos estáticos se realizan mediante diagramas de clases y de objetos y los dinámicos se capturan en los diagramas de interacción, de estados y de actividades.

3. **Vista de procesos:** Comprende los hilos y procesos que forman los mecanismos de sincronización y concurrencia del sistema, considerando la capacidad de crecimiento, rendimiento y funcionamiento del sistema.

Los aspectos estáticos y dinámicos se muestran con los mismos diagramas que la vista de diseño pero haciendo hincapié en las clases activas que representan estos hilos y procesos.

4. **Vista de implementación:** Comprende los componentes y archivos que se utilizan para ensamblar y hacer disponible el sistema físico.

Se ocupa principalmente de la gestión de configuración de las distintas versiones de un sistema a partir de componentes y archivos un tanto independientes y que pueden ensamblarse de diferentes formas para producir un sistema en ejecución.

Los aspectos estáticos nos los dan los diagramas de componentes y los dinámicos con los diagramas de interacción, de estados y actividades.

5. Vista de despliegue: Contiene los nodos que forman las topologías hardware sobre la que se ejecuta el sistema.

Se ocupa de la distribución, entrega e instalación de las partes que constituyen el sistema físico.

Los aspectos estáticos se capturan con los diagramas de despliegue y los dinámicos con los diagramas de interacción, de estado y de actividades.

Cada una de estas vistas puede existir por sí misma y a su vez también pueden interactuar entre sí.

Modelo del comportamiento en UML

Interacciones

Una **interacción** es un comportamiento que incluye un conjunto de mensajes intercambiados por un conjunto de objetos dentro de un contexto para lograr un propósito.

Las interacciones se utilizarán para modelar el comportamiento dinámico de las colaboraciones.

En el contexto de las interacciones podemos encontrar clases, interfaces, componentes, nodos y casos de uso.

Los aspectos dinámicos se visualizan, especifican, construyen y documentan como flujos de control que pueden incluir simples hilos secuenciales a través de un sistema, así como flujos más complejos que implican bifurcaciones, iteraciones, recursión y concurrencia.

Un **mensaje** es la especificación de una comunicación entre objetos que transmiten información con la expectativa de que se encadenará una actividad.

Al pasar un mensaje, la acción resultante es una instrucción ejecutable que constituye

una abstracción de un procedimiento computacional.

Tipos de acciones

1. Llamada: Invoca una operación sobre el objeto, que puede enviarse un mensaje a si mismo lo que resulta en la invocación local de una operación.
2. Retorno: Devuelve un valor al invocador.
3. Envío: Envía una señal a un objeto.
4. Creación: Crea un objeto.
5. Destrucción: Destruye un objeto.

Secuenciación

Cuando un objeto envía un mensaje a otro, este a su vez puede enviar un mensaje a otro objeto (este flujo de mensajes forma una secuencia).

Cualquier secuencia debe tener un comienzo que tiene su origen en algún proceso o hilo y continuará mientras dicho proceso/hilo que lo contiene exista.

Cualquier proceso/hilo dentro de un sistema define un flujo de control separado y en él los mensajes se ordenan en secuencia según se suceden en el tiempo.

Para visualizar mejor dicha secuencia se puede expresar explícitamente la posición de un mensaje con relación al inicio de la secuencia, precediéndolo de un número de secuencia separado por 2 puntos.

Cuando se modelan interacciones que involucran varios flujos de control es importante identificar el proceso/hilo que envió un mensaje particular.

En UML se distingue un flujo de control de otro escribiendo el número de secuencia de un mensaje precedido del nombre del proceso/hilo que es el origen de la secuencia.

Además de mostrar los parámetros reales enviados junto a una operación en el contexto de una interacción también se pueden mostrar los valores de retorno de una función.

Diagramas de casos de uso

Un **caso de uso** especifica el comportamiento de un sistema o una parte del mismo, y es una descripción de un conjunto de secuencias de acciones que ejecuta un sistema para producir un resultado observable de valor para un actor.

Se utilizan para capturar el comportamiento deseado del sistema en desarrollo, sin tener que especificar cómo se implementa ese comportamiento.

Permiten la comprensión del sistema a los desarrolladores, los expertos en el dominio y los usuarios finales.

Son un medio para ayudar a validar la arquitectura y verificar el sistema mientras evoluciona en el desarrollo.

Los casos de uso bien estructurados muestran sólo comportamientos esenciales de un sistema o subsistema, no siendo ni demasiados genéricos ni demasiados específicos.

Los casos de uso se utilizan como base para establecer casos de pruebas para esos elementos mientras evolucionan durante el desarrollo:

1. Los casos de uso aplicados a subsistemas son una fuente excelente de pruebas de regresión.
2. Los casos de uso aplicados al sistema completo son una fuente excelente de pruebas de sistema y de integración.
3. Permiten definir los límites del sistema y las relaciones entre el sistema y el entorno.
4. Cubren la carencia existente en métodos previos en cuanto a la determinación de requisitos.
5. Se representan gráficamente mediante una elipse con un nombre que lo distinga de los otros casos de uso.

Dicho nombre puede ser simple o nombre de camino cuando se indica el nombre del paquete en el que se encuentra.

Tipos de actores

1. Principales: Personas que usan el sistema.

2. Secundarios: Personas que mantienen o administran el sistema.
3. Material externo: Dispositivos materiales imprescindibles que forman parte del ámbito de la aplicación y deben ser utilizados.
4. Otros sistemas: Sistemas con los que interactúa el sistema.

La misma persona física puede interpretar varios papeles como actores distintos.

El nombre del actor describe el papel desempeñado.

Los actores se representan como monigotes.

Se pueden distinguir categorías generales de actores y especializarlos mediante relaciones de generalización.

Los actores se conectan a los casos de uso a través de asociaciones que indican la comunicación entre ellos y cada uno de puede enviar y recibir mensajes.

El **comportamiento** de un caso de uso se especifica describiendo un flujo de eventos de forma textual, de forma lo suficientemente clara.

En este flujo de eventos se debe incluir cómo y cuándo empieza y acaba el caso de uso, cuando interactúa con los actores y qué objetos se intercambian.

Además, se deberá especificar el flujo básico y los flujos alternativos del comportamiento.

Los casos de uso se determinan observando y precisando, actor por actor, las secuencias de interacción, los escenarios, desde el punto de vista del usuario.

Un **escenario** es una instancia de un caso de uso y se representa mediante los diagramas de interacción.

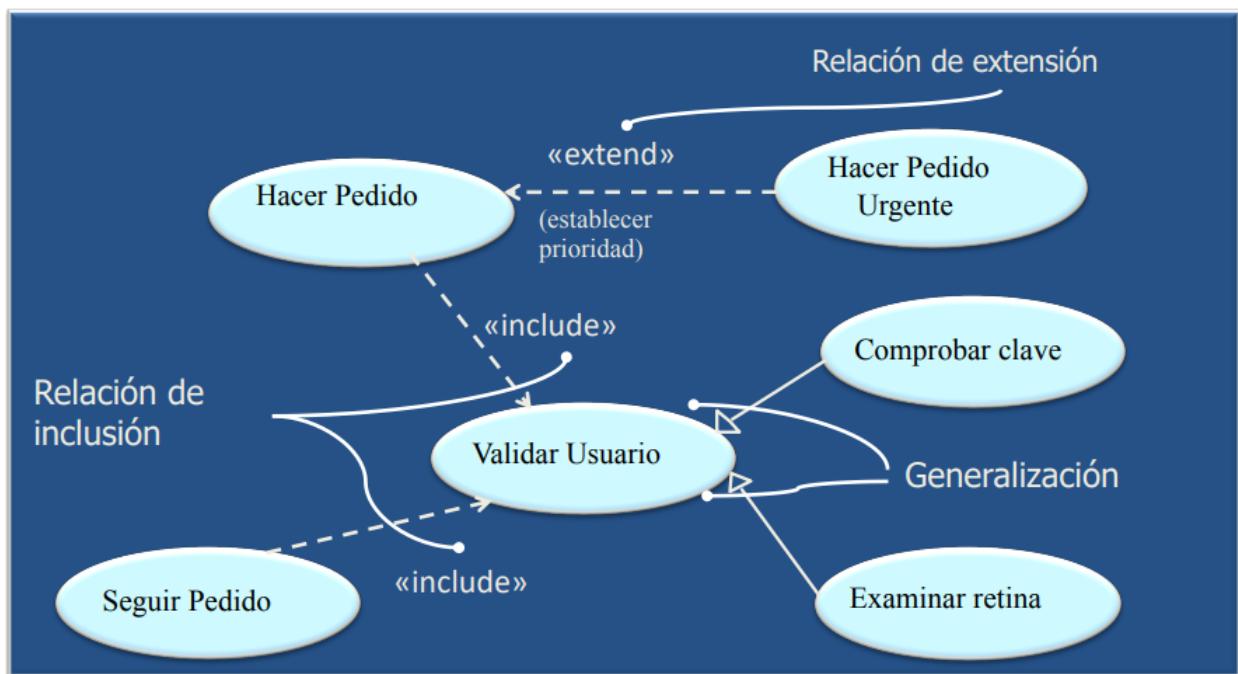
La **realización** de un caso de uso se puede especificar explícitamente mediante una colaboración, aunque en la mayoría de los casos no es necesario mostrar dicha colaboración.

El objetivo de la estructura de un sistema es encontrar el conjunto mínimo de colaboraciones bien estructurado que satisfacen el flujo de eventos, especificado en todos los casos

de uso del sistema.

Tipos de relaciones en los diagramas de casos de uso

1. Comunicación: Se produce entre los actores y el sistema y determina las entradas y salidas de la especificación mostradas en el diagrama.
2. Herencia: El caso de uso origen hereda la especificación del caso de uso destino y posiblemente la amplía y/o modifica.
3. Inclusión: El caso de uso base incorpora explícitamente el comportamiento de otro caso de uso en el lugar especificado en caso de uso o bien, una instancia del caso de uso origen incluye también el comportamiento descrito por el caso de uso destino.
4. Extensión: El caso de uso origen extiende el comportamiento del caso de uso destino, por lo tanto indica que un caso de uso base incorpora implícitamente el comportamiento de otro caso de uso en el lugar especificado indirectamente por el caso de uso que extiende la base.



Modularización de casos de uso

Es un mecanismo que nos permite organizar una especificación que utiliza casos de uso pero evitando redundancias y facilitando la comprensión mejor del sistema.

Para ello utilizaremos los siguientes **artefactos de UML**:

1. Relaciones de extensión: Muestran una parte de la funcionalidad del caso de uso que no siempre ocurre y no necesariamente provienen de un error o una excepción.
2. Relaciones de uso: Representamos una funcionalidad común que es accedida por un conjunto de casos. El caso es usado siempre que el caso que lo usa es ejecutado.

Tipos de casos de uso

1. Esenciales o de trazo grueso: Se ignoran detalles sobre la forma de interacción entre el actor y el sistema.
Sólo se incluyen las alternativas más relevantes y no se entra en detalle sobre las acciones que realiza el sistema cuando el usuario interactúa con él.
2. Implementación o de trazo fino: Completan todos los detalles que no se han especificado anteriormente, completamos las alternativas de todo tipo y especifica con más detalle el comportamiento interno del sistema.
3. Temporales: El inicio del caso de uso está determinado por el paso del tiempo.
4. Primarios: Se corresponden con los procesos de negocio.
5. Secundarios: Son necesarios para que el sistema funcione normalmente.

Proceso de análisis de requisitos con casos de uso

1. Identificar los actores
2. Identificar los principales casos de uso de cada actor.
3. Identificar nuevos casos a partir de los existentes.
4. Crear descripciones de casos de uso de trazo grueso.
5. Definir prioridades y seleccionar casos de la primera iteración.
6. Escribir los casos de trazo fino y crear prototipos de interfaces.

Un caso de uso es una descripción de las posibles secuencias de interacción entre el sistema

y actores externos en relación a el objetivo de un actor particular.

Fases del mecanismo de ejecución de un caso de uso

1. El actor envía al sistema una petición y los datos necesarios para llevarla a cabo
2. El sistema valida la petición y los datos
3. El sistema altera su estado interno
4. El sistema devuelve el resultado al actor

Diagramas de Casos de Uso

Un **diagrama de casos de uso** es un diagrama que muestra un conjunto de casos de uso, actores y sus relaciones, que pueden ser de dependencia, generalización y asociación. Al igual que los demás diagramas pueden contener notas y restricciones.

Los diagramas de casos de uso pueden contener paquetes , que se emplean para agrupar elementos del modelo en partes mayores.

Se emplean para modelar la vista de casos de uso estática de un sistema, pudiendo utilizarlos normalmente para modelar el contexto de un sistema o para modelar los requisitos de un sistema.

Los diagramas de casos de uso pueden ser vistos como un mapa general de las funcionalidades más importantes del sistema.

También pueden utilizarse en ingeniería inversa.

Especificación de los diagramas de casos de uso

Para completar eficientemente los diagramas de casos de uso se utiliza las denominadas **plantillas** en las que en realidad se incluye un cuestionario sobre diferentes aspectos.

Se utilizan para dar un formato uniforme a la explicación textual de los casos de uso.

Diagramas de interacción

Modelan el comportamiento dinámico del sistema.

Describe la interacción entre objetos; los objetos interactúan a través de mensajes para cumplir ciertas tareas.

Las interacciones proveen un “comportamiento” y típicamente implementan un Caso de Uso.

Existen dos tipos de diagramas de interacción en UML: **Diagramas de Secuencia (dimensión temporal)** y **Diagramas de Comunicación o Colaboración (dimensión estructural)**

Los elementos que contiene son: objetos, enlaces y mensajes, pudiendo contener al igual que los demás diagramas notas y restricciones.

Un **diagrama de secuencia** es un diagrama en el que se destaca la ordenación temporal de los eventos.

Gráficamente es una tabla que representa a objetos, dispuestos a lo largo del eje X, y mensajes, ordenados según se suceden en el tiempo, a lo largo del eje Y.

Un **diagrama de colaboración o comunicación** destaca la organización estructural de los objetos que envían y reciben los mensajes.

Gráficamente es una colección de nodos y arcos.

Son semánticamente equivalentes.

Los objetos interactúan para realizar colectivamente los servicios ofrecidos por las aplicaciones.

Los diagramas de interacción muestran cómo se comunican los objetos en una interacción.

Los **diagramas de secuencia** se suelen asociar a los casos de uso, mostrando como estos se realizan a través de interacciones entre sociedades de objetos.

Los diagramas de secuencia tienen dos características que los distinguen de los diagramas de colaboración o comunicación:

1. **Línea de vida:** Línea discontinua vertical que representa la existencia de un objeto a lo largo de un periodo de tiempo.

Pueden crearse objetos durante la interacción y su línea de vida comienza con la recepción del mensaje estereotipado como «create».

Los objetos también pueden destruirse durante la interacción y su línea de vida acaba con la recepción del mensaje estereotipado como «destroy», añadiendo además una señal en X.

2. **Foco de control:** Rectángulo delgado y estrecho que representa el periodo de tiempo durante el cual un objeto ejecuta una acción, bien directamente o a través de un procedimiento subordinado.

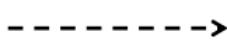
También puede mostrarse el anidamiento de un foco de control colocando otro foco de control ligeramente a la derecha de su foco padre.

Tipos de mensajes de los diagramas de interacción



Mensaje Simple / Sincrónico

No se dan detalles de la comunicación cuando no son conocidos o no son relevantes.



Respuesta / Resultado



Mensaje Asincrónico

Sintaxis del mensaje:

Número de secuencia [condición] * [expresión iteración]
valor de retorno := nombre del mensaje (parámetros)

Operadores de Control y/o Marcos de Interacción

Un **marco de interacción** es una parte del diagrama de secuencia asociado a una etiqueta, la cual contiene un **operador de control** que determina el modo de ejecución de esa secuencia.

Principales modalidades de ejecución

1. **Ejecución Opcional:** El cuerpo del operador de control se ejecuta si una condición de guarda es cierta cuando se entra en el operador.

2. **Ejecución Condicional:** La etiqueta es alt el cuerpo del operador de control se divide en varias subregiones con líneas discontinuas horizontales, representando cada una una rama de la condición.
3. **Ejecución Bucle (iterativa):** La etiqueta es loop . Comportamiento repetitivo, indicando min y max y la condición de guarda. El cuerpo del bucle se ejecuta mientras la condición de guarda sea cierta antes de cada iteración.
4. **Ejecución Paralela:** La etiqueta es par el cuerpo del operador de control se divide en varias subregiones con líneas discontinuas horizontales, representando cada una una computación paralela (concurrente).

Los **diagramas de colaboración o comunicación** destacan la organización de los objetos que participan en una interacción.

Es un grafo en el que los nodos son los objetos y los arcos los enlaces.

Los arcos se etiquetan con los mensajes que envían y reciben los objetos.

Dan una visión del flujo de control en el contexto de la organización de los objetos que colaboran.

Características que los distinguen de los diagramas de secuencia

1. **Camino:** Para indicar como se enlazan los objetos se utilizan estereotipos en los extremos de los enlaces (local, global y self).
2. **Número de secuencia:** Para indicar la ordenación de los mensajes se utiliza la numeración decimal de Dewey (1, 1.1,.....).

Es un diagrama de interacción que enfatiza la organización estructural de los objetos que participan en una interacción y se muestran: objetos / clases, relaciones/uniones entre ellos y mensajes.

Los mensajes son numerados para mostrar secuencias.

Recordar que tanto los diagramas de colaboración como los de secuencia son proyecciones del mismo modelo de los aspectos dinámicos de un sistema.

Para modelar la dinámica del sistema global se utilizan varios diagramas de interacción, así como las de sus subsistemas, operaciones, clases, casos de uso y colaboraciones.

El objetivo principal en su desarrollo debe ser la simplicidad.

Son útiles para ver el comportamiento de varios objetos en un caso de uso.

Diagramas de actividades

Los **Diagramas de actividades** se pueden considerar un caso especial de máquina de estados en la que: La mayoría de los estados son actividades, y la mayoría de las transiciones se disparan implícitamente por la terminación de acciones.

Diferencias con un diagrama de estados

1. Un diagrama de actividad se usa para modelar una secuencia de acciones en un proceso.
2. Un diagrama de estados para modelar los estados discretos de un objeto a lo largo de su ejecución.

Un **diagrama de actividades** es esencialmente un diagrama de flujo que destaca la actividad que tiene lugar a lo largo del tiempo.

Una **actividad** es una ejecución no atómica en curso, dentro de una máquina de estados.

Las actividades producen finalmente alguna acción , que está compuesta de computaciones atómicas ejecutables que producen un cambio en el estado del sistema o la devolución de un valor.

Las **acciones** incluyen llamadas a otras operaciones, envío de señales, creación o destrucción de objetos o simples cálculos como la evaluación de una expresión.

Gráficamente es una colección de nodos y arcos.

Los diagramas de actividades normalmente contienen Estados de actividad y estados de acción, Transiciones y Objetos.

Los **estados de acción** son computaciones ejecutables y atómicas ya que son estados del sistema y cada una representa la ejecución de una acción.

Se considera que la ejecución de un estado de acción conlleva un tiempo insignificante.

Los **estados de actividad** pueden descomponerse, representando su actividad con otros diagramas de actividades.

No son atómicos y por lo tanto pueden ser interrumpidos, considerando que se invierte algún tiempo en completarse.

Pueden tener partes adicionales como acciones de entrada y salida.

Las **transiciones** muestran el camino de un estado de actividad o acción al siguiente.

Las transiciones secuenciales son frecuentes aunque no son el único camino que se necesita para modelar un flujo de control, así se puede incluir una **bifurcación** que especifica caminos alternativos elegidos según el valor de alguna expresión booleana.

La bifurcación puede tener una transición de entrada y dos o mas de salida. En cada transición de salida se coloca la expresión booleana que se evalúa sólo una vez al entrar.

Las guardas de las transiciones de salida no deben solaparse aunque deben cubrir todas las posibilidades para que el flujo de control no se vea interrumpido.

Cuando se modelan flujos de trabajo de procesos de negocios es posible encontrar en los diagramas de actividades flujos **concurrentes**.

Se utiliza una **barra de sincronización** para especificar la división y unión.

Una **división** puede tener una transición de entrada y dos o más de salida, cada una representando un flujo de control independiente.

La **unión** representa la sincronización de dos o más flujos de control concurrentes, puede tener dos o más transiciones de entrada y una de salida de estos flujos. Los flujos concurrentes se sincronizan.

Las **calles o Swimlanes** son útiles cuando se modelan flujos de trabajo de procesos de negocio.

Representan una división de actividades en grupos, normalmente asignados a objetos o subsistemas.

Cada calle tiene un nombre único en un diagrama.

Una calle realmente no tiene una semántica profunda sino que puede representar una entidad del mundo real, representando una responsabilidad de alto nivel de una parte de la actividad global de un diagrama de actividades, pudiendo ser implementada por una o más clases.

Los diagramas de actividades se utilizan para modelar los aspectos dinámicos de un sistema.

Su uso más generalizado es en el contexto de un sistema global, un subsistema, una operación o una clase.

Otra alternativa de utilización es asociando diagramas de actividades a los casos de uso, para modelar un escenario y a las colaboraciones para modelar aspectos dinámicos de una sociedad de objetos.

Usos más extendidos de los diagramas de actividades

1. **Modelar un flujo de trabajo:** Se hace hincapié en actividades tal y como las ven los actores.
2. **Para modelar una operación:** Se utilizan como diagramas de flujo, para modelar los detalles de una computación.

Diagramas de estado

En UML un **evento** es la especificación de un acontecimiento significativo que ocupa un lugar en el tiempo y en el espacio.

En el contexto de las máquinas de estados modelan los estímulos que producen un cambio de estado.

Tipos de eventos:

1. **Síncronos:** llamadas o invocación de operaciones.

2. **Asíncronos:** señales, paso del tiempo y el cambio de estado, representan sucesos que pueden ocurrir en cualquier instante.

Tipos:

- **Externos:** entre el sistema y sus actores (interrupción, pulsación de una tecla,...)
- **Internos:** entre los objetos del sistema
- Se pueden modelar 4 tipos de eventos:
 - Señales
 - Eventos de llamada
 - Paso del tiempo (evento de tiempo)
 - Cambio de estado (evento de cambio)

Señales

- Son objetos con nombre enviados (lanzados), asíncronamente por un objeto y capturados por otro.
- El tipo más común de señales internas son las excepciones, tal y como son soportadas por los lenguajes de programación.
- Son una clase en sentido general:
 - Pueden existir instancias
 - Pueden existir relaciones de generalización
 - Pueden tener atributos y operaciones
- Se generan como resultado de una transición en una máquina de estados de un objeto

Eventos de Llamada

- Representan la invocación de una operación de un objeto
- Son eventos **síncronos**
- Cuando un objeto invoca una operación sobre otro objeto que tiene una máquina de estados:
 - el control pasa del emisor al receptor
 - el evento dispara la transición
 - la operación acaba
 - el receptor pasa a un nuevo estado y el control regresa al emisor
- No existen señales visuales para distinguir un **evento de señal** de un **evento de llamada**, sólo el contexto del modelo
- La operación aparecerá declarada en la lista de operaciones del objeto receptor
- Una señal será manejada por la máquina de estados y un evento de llamada por un método

Eventos de tiempo y cambio

- Un evento de tiempo representa el paso del tiempo
- Se modela con la palabra **after** seguida de una expresión
- El tiempo se mide desde el instante en el que se entra en el estado
- Un evento de cambio representa un cambio en el estado o el cumplimiento de alguna condición
- Se modela con la palabra **when** seguida de una expresión booleana

Máquina de estados

- Una máquina de estados especifica la secuencia de estados por los que pasa un objeto durante su vida
- La evolución se produce a causa de eventos, bien internos, bien enviados desde otro objeto
- También se pueden utilizar para modelar el comportamiento dinámico de otros elementos de modelado (instancias de una clase, un caso de uso o un sistema completo)
- Relación con otros diagramas:
 - 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

Estados

- Un estado es una condición o situación en la vida de un objeto durante la cual
 - Satisface alguna condición
 - Realiza alguna actividad
 - Espera algún evento
- Un objeto permanece en un estado durante una cantidad de tiempo finita
- Un estado tiene varias partes como
 - **Nombre:** es una cadena de texto que distingue al estado de otros estados, en ocasiones puede ser anónimo
 - **Acciones de entrada/salida:** son las acciones ejecutadas al entrar y salir del estado
 - **Transiciones internas:** son las que se manejan sin provocar un cambio en el estado
 - **Subestados:** estructura anidada de un estado que engloba subestados disjuntos (activos secuencialmente) o concurrentes (activos concurrentemente)
 - **Eventos diferidos:** son una lista de eventos que no se manejan en este estado sino que se posponen y se añaden a una cola para ser manejados por el objeto en otro momento

Transiciones

- Una transición es una relación entre dos estados que indica que un objeto que esté en el primer estado realizará ciertas acciones y entrará en el segundo estado cuando ocurra un evento especificado y si se cumplen las condiciones especificadas
 - Cuando se produce un cambio de estado se dice que la transición se ha disparado
 - Hasta que se dispara la transición se dice que el objeto está en el estado origen, después de dispararse se dice que el objeto está en el estado destino
 - Se representa como una línea continua dirigida desde el origen hasta el destino
 - Una auto transición es aquella cuyo estado origen y destino es el mismo
 - Puede tener múltiples orígenes representando una unión (join) de muchos estado concurrentes o múltiples destinos que representan una división (fork) a múltiples estados concurrentes
-
- Si un objeto está en un estado y ocurre un **evento** de los que etiquetan una de sus **transiciones**, el objeto entra en el estado destino de la transición y se dice que la transición se ha disparado
 - Si más de una **transición** abandona un estado, el primer evento que ocurra causa que se dispare la transición correspondiente. Si ocurre un evento que no corresponde a una transición que abandona el estado actual, se ignora el evento
 - Una **secuencia de eventos** corresponde a un camino a través del grafo
 - Un estado está **activo** cuando se alcanza mediante una transición y se convierte en **inactivo** cuando se abandona a través de una transición

- **Evento de disparo**
 - Un evento en el contexto de las máquinas de estado, es la aparición de un estímulo que puede disparar una transición de estado y que pueden incluir señales, llamadas, el paso del tiempo o un cambio en el estado
 - Una señal o una llamada pueden tener parámetros cuyos valores estén disponibles para la transición, incluyendo expresiones para la condición de guarda y la acción
 - Puede ser polimórfico
- **Condición de guarda**
 - Se representa como una expresión booleana entre corchetes y se coloca tras el evento de disparo
 - Sólo se evalúa después de ocurrir el evento de disparo de la transición
 - Es posible tener muchas condiciones de guarda desde el mismo estado origen y con el mismo evento de disparo, siempre que las condiciones no se solapen
 - Se evalúa sólo una vez por cada transición, aunque puede ser evaluada de nuevo si la transición se vuelve a disparar
- **Acción**
 - Es una computación atómica ejecutable
 - Pueden incluir llamadas a operaciones sobre el objeto que contiene la máquina de estados o sobre otros objetos visible
 - La creación o destrucción de otro objeto
 - Envío de una señal a un objeto
 - No puede ser interrumpida y se ejecuta hasta su terminación

Un diagrama de estado que describe los eventos globales del sistema y su secuencia en un caso de uso es un **diagrama de estado para casos de uso**.

Acciones de entrada y salida

- Se utilizan cuando al entrar o salir de un estado se requiere realizar una acción
- Son independientes de la transición por la que se llega o por la que se abandona el estado
- Se puede lograr el mismo efecto con una máquina de estados plana, pero sería necesario especificar estas acciones en cada entrada y salida
- No pueden tener parámetros, ni guardas
- Se representan con **entry / acción, exit / acción** dentro del estado

Transiciones internas

- Son transiciones como respuesta a eventos que deben ser manejados sin abandonar el estado
- Son distintas de las **autotransiciones**:
 - En las autotransiciones, se abandona el estado y se vuelve a él y se tienen que ejecutar las acciones de entrada y salida.
 - Pueden tener eventos con parámetros y guardas
 - Son básicamente interrupciones
- Se representan mediante **transición/acción**, dentro del estado

Actividades

- Cuando un objeto está en un estado, puede no estar ocioso, sino ejecutando alguna tarea. Estas tareas son las actividades y se ejecutan de forma continua hasta que se recibe un evento
- Para representarlas se utiliza la transición especial `do/actividad`
- Se pueden especificar secuencias de acciones `do/a1;a2;a3`
- En este caso las acciones no se interrumpen, pero la actividad se puede interrumpir entre acciones

Eventos diferidos

- En UML, sólo los eventos especificados son tratados. Si un evento llega y no está especificado el comportamiento en ese estado, se pierde
- Si se quiere conservar un evento, pero no se quiere tratar, hay que especificarlo como diferido. Existe una cola interna donde se almacenan estos eventos
- Son tratados tan pronto como el objeto entra en un estado que no difiere estos eventos
- Se especifica `nombre_evento / defer`, dentro del estado

Subestados

- Los subestados son características de la máquina de estados que ayudan también a simplificar los comportamientos complejos
- Un subestado es un estado anidado dentro de otro
- Un estado simple no tiene una subestructura
- Un estado con subestados se dice que es compuesto
- Un estado compuesto puede contener subestados concurrentes (ortogonales) y secuenciales (disjuntos)
- Los subestados se pueden anidar a cualquier nivel

Subestados secuenciales

- Sólo un estado dentro del subestado está activo al mismo tiempo
- Dado un estado compuesto una transición de origen externo puede activar el estado compuesto o puede activar un subestado
 - si el destino es el estado compuesto, la máquina de estados anidada debe incluir un estado inicial, al cual se pasa el control después de entrar en el estado compuesto y ejecutar la acción de entrada si la hay
 - Si el destino es el estado anidado el control pasa a éste después de ejecutar la acción de entrada del estado compuesto y la del subestado si la tiene
- Se pueden especificar transiciones comunes a todos los subestados (cancelar en el ejemplo)
- Pueden tener o no un estado inicial
- Cómo máximo pueden tener un estado inicial y otro final
- Una transición que salga de un estado compuesto puede tener como origen el estado compuesto o un subestado, ejecutando con anterioridad las correspondientes acciones de salida si las hay tanto del estado anidado como del compuesto

Subestados concurrentes

- Permiten especificar dos o más submáquinas que se ejecutan en paralelo en el contexto del objeto que las contiene
- Para describir un subestado concurrente es necesario especificar un estado por cada submáquina
- En lugar de dividir el estado de un objeto en estados concurrentes se pueden definir dos objetos activos, con su propia máquina
- Si el comportamiento de uno de los objetos depende del estado del otro es mejor usar subestados
- Si los comportamientos dependen sólo de los mensajes, es mejor utilizar objetos activos
- Si hay poca o ninguna comunicación es indiferente usar uno u otro, aunque en general es más claro usar objetos activos

- La diferencia entre los subestados secuenciales y los subestados concurrentes es que
 - Dados dos o más subestados secuenciales al mismo nivel un objeto estará en uno y sólo uno de esos subestados
 - Dados dos o más subestados concurrentes al mismo nivel un objeto estará en un estado secuencial de cada uno de los subestados concurrentes
- Cada máquina de estados anidada alcanza su estado final antes o después. Si un subestado concurrente alcanza su estado final antes que el otro, el control en ese subestado espera en su estado final
- Cuando ambas máquinas de estado alcanzan su estado final el control de los subestados concurrentes se vuelve a unir en un único flujo
- Cada vez que se produce una transición hacia un estado compuesto el control se divide en tantos flujos como subestados concurrentes

Modelo estructural en UML

Los diagramas estructurales de UML existen para visualizar, especificar, construir y documentar los aspectos estáticos de un sistema.

Los aspectos estáticos de un sistema son aquellos que cubren la parte más estable del sistema, por lo que lo podríamos ver como su esqueleto y su andamiaje.

Los aspectos estáticos de un sistema software incluyen la existencia y ubicación de clases, interfaces, colaboraciones, componentes y nodos

Diagrama de clases

Son los más utilizados en el modelado orientado a objetos y nos muestran un conjunto de clases , interfaces y colaboraciones, así como sus relaciones.

- Son los más utilizados en el modelado orientado a objetos y nos muestran un conjunto de clases , interfaces y colaboraciones, así como sus relaciones
- Están basados en la consideración de que el mundo real puede ser visto desde diferentes abstracciones (subjetividad), existiendo diferentes mecanismos de abstracción como
 - Clasificación / Instanciación
 - Composición / Descomposición
 - Agrupación / Individualización
 - Especialización / Generalización
- La clasificación es uno de los mecanismos de abstracción más utilizados

- La clases consideran el ámbito de definición de un conjunto de objetos
- Cada objeto pertenece a una clase
- Los objetos se crean por instantiación de las clases
- Los diagramas de clases se utilizan para modelar la vista estática de un sistema incluyendo:
 - Modelar el vocabulario del sistema
 - Modelar las colaboraciones, o
 - Modelar esquemas lógicos de bases de datos
- Son la base para un par de diagramas relacionados como los diagramas de componentes y los diagramas de despliegue
- Los diagramas de clases se pueden considerar desde diferentes perspectivas(Cook y Daniels, 1994)
 - **Conceptual**, en la que se representan los conceptos del dominio estudiado. Estos conceptos se relacionan de forma natural con las clases que lo implementan, pero con frecuencia no hay una correspondencia directa. Estos modelos se deben considerar independientes del lenguaje
 - **Especificación**, se representan los tipos que representan una interfaz la cual puede tener cualquier implementación. Un tipo representa una interfaz que puede tener muchas implementaciones distintas, debido por ejemplo al ambiente de implementación, características de desempeño o al proveedor.
 - **Implementación**, en la que se representan las clases tal y como serán implementadas, esta es la perspectiva más empleada aunque es en muchos casos mejor emplear la de especificación

1. Atributos de los diagramas de clase

[**visibilidad**] nombre [: tipo] [= valor_inicial] [{propiedades}]

visibilidad	$\left\{ \begin{array}{l} + = \text{pública} \\ \# = \text{protegida} \\ - = \text{privada} \\ \sim = \text{paquete} \end{array} \right.$
nombre:	nombre del atributo
atributo derivado :	/
tipo:	tipo del atributo
valor_inicial:	valor inicial o por defecto
propiedades:	{frozen} {addOnly}

2. Operaciones de los diagramas de clase

[visibilidad] nombre [(lista_parametros)] [: tipo_retorno] [{propiedades}]

visibilidad {
+ = pública
= protegida
- = privada
~ = paquete

nombre: nombre de la operación

lista_parámetros:

tipo retorno: tipo de valor devuelto por la operación

propiedades: {isQuery}, {sequential}, {concurrent}

- **IsQuery**, la ejecución de la operación no cambia el estado del sistema, por tanto es una función pura sin efectos laterales
- **Sequential**, los invocadores deben coordinarse para que en el objeto solo haya un único flujo al mismo tiempo
- **Guarded**, La semántica y la integridad se garantizan en presencia de múltiples flujos de control por medio de la secuenciación de todas las llamadas a todas las operaciones con guarda de objeto
- **Concurrent**, la semántica e integridad del objeto se garantizan en presencia de múltiples flujos de control, tratando la operación como atómica

Observaciones sobre operaciones

- En ocasiones es útil hacer distinción sobre aquellas operaciones que cambian el estado de una clase y aquellas que no lo hacen
- Una **consulta** es una operación que obtiene un valor de una clase sin que cambie el estado observable de la clase
- Un **modificador** es una operación que cambia el estado observable de un objeto
- Una observación es que las consultas se pueden realizar en cualquier orden pero las secuencia de los modificadores tiene gran importancia
- También habrá que distinguir entre **método de observación** que es aquel que devuelve el valor de un campo sin hacer nada más y el **método de establecimiento** que pone un valor en un campo y nada más
- Distinguiremos también entre **operación** que es algo que se invoca sobre un objeto (la llamada de procedimiento) y **método** que es el cuerpo del procedimiento

- La **multiplicidad** define cuántas instancias de un tipo A pueden asociarse a una instancia del tipo B en determinado momento. Las expresiones de multiplicidad son las siguientes:
 - 0..* cero ó más, muchos
 - 0..1 cero ó 1
 - 1 exactamente 1
 - 1..* uno o más
 - 1..40 de uno a cuarenta
 - 5 exactamente cinco
 - 2,4,6 exactamente dos, cuatro o seis

Relación Dependencia

- Un cambio en la especificación de un elemento afecta a otro. Se deben utilizar cuando se quiera representar que un elemento utiliza a otro (línea discontinua dirigida al elemento del que se depende)

Estereotipos para dependencias

- **bind:** entre una clase genérica y una instancia
- **friend:** dependencia de clase amiga
- **refine:** relación de refinamiento
- **use:** relación de uso
- **import:** un paquete importa los elementos de otro
- **extend:** para casos de uso
- **include:** para casos de uso
- **instanceOf:** especifica que el objeto origen es una instancia del clasificador destino
- **Instantiate:** especifica que el origen crea instancias del destino

Generalización

- Es una relación entre un elemento general y un tipo más específico de ese elemento
- La herencia simple es suficiente en la mayoría de los casos
- Se define como “es un tipo de”

- Para especificar adornos UML define un estereotipo y cuatro restricciones para las generalizaciones
 - **Implementación**, especifica que el hijo hereda la implementación del padre, pero no hace público ni soporta sus interfaces, violando en este caso la semántica de la sustitución. Se utiliza para modelar la herencia privada, como la que aparece en C++
 - **Complete**, especifica que todos los hijos en la generalización se han especificado en el modelo y no se permiten hijos adicionales
 - **Incomplete**, indica que no se han especificado todos los hijos en la generalización y que se permitirán hijos adicionales
 - **Disjoint**, especifica que los objetos del padre no pueden tener más de uno de los hijos como tipo (se utiliza en la herencia múltiple)
 - **Overlapping**, especifica que los objetos del padre pueden tener más de uno de los hijos como tipo (se utiliza en la herencia múltiple)
- **Asociación**, es una relación estructural que especifica que los objetos de un elemento se conectan a los elementos de otro
- **Agregación**, es un caso especial de asociación, representa la relación estructural “parte de”

Asociación calificada

En UML se modela este esquema como un calificador, que es un atributo de una asociación cuyos valores partitionan el conjunto de objetos relacionados a través de una asociación

El objeto origen, junto a los valores de los atributos del calificador, devuelven un objeto destino, si la multiplicidad del destino es como máximo uno o un conjunto de objetos si la multiplicidad del destino es muchos

Navegabilidad

- Posibilidad de limitar la navegación a una sola dirección
- Determina si una clase de la asociación tiene “conocimiento” de la otra.
- Nivel de especificación o implementación

Visibilidad

- Pública: + propietario
- Protegida: #propietario
- Privada: - propietario
- Paquete: ~ propietario

Composición

- Es un forma de agregación con fuerte pertenencia y un tiempo de vida coincidente entre las partes y el todo.
- Las partes con una multiplicidad no fija pueden ser creadas después del propio compuesto, pero una vez creadas viven y mueren con él
- Tales partes también pueden ser eliminadas explícitamente antes de la eliminación del compuesto

Clase Asociación

- Una clase asociación añade una restricción:
“Sólo puede existir una instancia de la asociación entre cualquiera par de objetos participantes”
- No podríamos modelar que una persona tiene diferentes contratos para una misma compañía a lo largo del tiempo.

Realización

- Es una relación semántica entre clasificadores, en la cual un clasificador especifica un contrato que otro clasificador garantiza que cumplirá
- Se representa como una línea dirigida discontinua, con una punta vacía que apunta al clasificador que especifica el contrato
- Este tipo de relación es lo suficientemente fuerte como para ser tratada como un tipo independiente de relación
- Semánticamente es una mezcla entre dependencia y generalización
- Se utiliza en el contexto de las interfaces y en el de las colaboraciones
- Teniendo en cuenta esto podemos decir que un diagrama de clases bien estructurado es aquel que
 - Se centra en comunicar un aspecto de la vista del sistema
 - Contiene sólo aquellos elementos que son esenciales para comprender ese aspecto
 - Proporciona detalles consistentes con el nivel de abstracción
 - Debe tener un nombre que comunique su propósito
 - Hay que intentar no mostrar demasiados tipos de relaciones

Los requisitos

- Los requisitos son una descripción de las necesidades o deseos de un producto
- Se recomienda aquí definir al menos los siguientes puntos:
 - Panorama general
 - Metas
 - Funciones del sistema

Técnicas Estructuradas

Visión de las técnicas de especificación

- Según Yourdon no existe un criterio definitivo por el que podamos organizar las técnicas de la metodología estructurada. Aunque si podemos considerar dos enfoques:
- Según el enfoque de **representación** pueden ser: gráficas, textuales, matriciales y marcos (frame)
- Según el enfoque de **modelado** bajo las que se crean modelos del sistema y distinguimos los basados en la función (estudian el sistema observando principalmente su funcionalidad), en la información (estudio del sistema respecto a los datos que maneja) y en el tiempo (estudio de las respuestas del sistema ante la aparición de eventos temporales)

Clasificación según la forma de representación

- **Gráficas:** cada técnica utiliza una serie de iconos que representan una serie de componentes de un aspecto particular del modelo. Se combinan con los otros tipos de técnicas
- **Textuales:** se utilizan para especificar con más detalle los componentes definidos mediante las técnicas gráficas haciendo uso de una gramática definida más o menos formal
- **Marcos o Plantillas:** especifican la información relativa a un componente de un modelo que ha sido declarado en un diagrama o en otro marco. Se representan mediante un formulario
- **Matriciales:** no se consideran por si solas técnicas de definición, sino de comprobación entre modelos, comprobación de la exactitud y compleción de unos modelos respecto de otros, ya que permiten estudiar las referencias cruzadas entre componentes

Clasificación según el enfoque de modelización

- **Dimensión de la Función**
 - Los DFD se utilizan para mostrar las funciones del sistema y sus interfaces. Se solapa con la dimensión de la información. Las técnicas en las que se apoya son el Diccionario de Datos y la Especificación de Procesos
- **Dimensión de la Información**
 - Los diagramas Entidad-InterRelación se utilizan para describir las entidades existentes en el sistema y las relaciones entre las mismas
- **Dimensión del Tiempo**
 - Las listas de eventos se utilizan para describir cualquier cosa que ocurra y sobre la que el sistema deba responder

Plano Información-Tiempo

- Se utilizan los diagramas de la Historia de la vida de Entidades para mostrar el efecto del tiempo sobre una entidad de datos. También se utiliza las Matrices Entidad- Evento que especifican las relaciones entre la entidades e interrelaciones de un diagrama E-R y un conjunto de eventos

Plano Información-Función

- DFD, mediante la cual se describe el uso de la información por un conjunto de funciones del sistema
- Matrices Entidad-Función que nos proporcionan referencias cruzadas entre las entidades definidas en el diagrama E-R accedidas por cada una de las funciones

Plano Función-Tiempo

- Redes Petri y los Diagramas de Transición de Estados que permiten mostrar el efecto del tiempo sobre un conjunto de funciones del sistema

Especificación y Modelado de la información

Objetivo

- Se intenta mantener el dominio de lo que será mantenido por el sistema, especificando cada uno de los items de datos individuales que son aceptados por el sistema, los cambios a los que son sometidos por el procesamiento y lo que el sistema devuelve como salida

Especificación y Modelado de Función

Objetivo

- Se intentan especificar las funciones, procesos o transformaciones que el sistema ejecuta u opera sobre la información que maneja.

Diagrama de Flujo de Datos

- Técnica gráfica que representa como la información se va moviendo a través del sistema sometida a sucesivas transformaciones
- Esta técnica es ampliamente utilizada con independencia de las características del sistema
- Define fácil e intuitivamente la funcionalidad del sistema
- Una simplificación lo constituyen los diagramas de flujo, los cuales solamente muestran la secuencia de ejecución del sistema

Objetivo

- Se intenta describir el momento en el cual se produce la instancia o ejecución de las funciones sobre la información que maneja el sistema.

Lista de Eventos

- Se trata de una relación de los eventos (cosas que ocurren en el mundo real y que producen un cambio en los datos). Se intenta describir los procesos elementales que realizan las funciones descritas en el DFD a mayor nivel de refinamiento

Redes Petri

- Los DTE pueden considerarse como un tipo de RP. En esta técnica de modelado se representan las transformaciones a que son sometidos los datos en el tiempo

Diagramas de Transición de Estados

- Se utiliza para representar el comportamiento del sistema en tiempo real, en el que el software debe responder a sucesos del mundo real en un tiempo muy limitado

Matriz Evento/ Entidad

- Técnica tabular que proporciona información sobre las entidades que son afectadas por cada uno de los eventos descritos mediante las otras técnicas

Diagramas de la Historia de la Vida de la Entidad

- Muestran el ciclo de proceso de una entidad desde su creación hasta su desaparición. Utiliza los diagramas Jackson

	Información	Función	Tiempo
Información	<ul style="list-style-type: none"> • Especificación de Entidad • Especificación de Interrelaciones • Especificación de TAD 		
Función		<ul style="list-style-type: none"> • Diccionario de Datos • Especificación de procesos • Especificación de Entidades Externas 	
Tiempo		<ul style="list-style-type: none"> • Definición de Funciones 	<ul style="list-style-type: none"> • Especificación de eventos

Clasificación de las principales técnicas de Especificación

	Información	Función	Tiempo
Información	<ul style="list-style-type: none"> • Diagramas E-R • DED • Matriz E/E 		
Función	<ul style="list-style-type: none"> • DFD • Matriz F/E 	<ul style="list-style-type: none"> • DFD • DDF • Diagramas de Estructuras 	
Tiempo	<ul style="list-style-type: none"> • Diagramas HVE • Matriz V/E 	<ul style="list-style-type: none"> • DTE 	<ul style="list-style-type: none"> • Lista Eventos • DTE

Clasificación de las principales técnicas de Modelado

Diagramas de Fluxos de Datos

Es la técnica más difundida dentro del análisis estructurado.

- Un DFD (diagrama de flujo de datos) es una representación en forma de red que refleja el flujo de la información y las transformaciones que se aplican sobre ella al moverse desde la entrada hasta la salida del sistema
- Se utiliza para modelar las funciones y los datos del sistema a distintos niveles de abstracción
- Para ello el sistema se modela mediante un conjunto de DFD nivelados, donde los niveles superiores definen las funciones del sistema de forma general y los niveles inferiores lo hacen de manera mas detallada

Los componentes son: Procesos, Almacenes, Entidades Externas y Flujo de Datos

Procesos

- Representan un componente que transforma los flujos de datos de entrada en uno o varios flujos de salida
- El termino proceso a veces puede dar lugar a confusión puesto que no hay que considerarlo como un programa en ejecución, sino como una función que tiene que realizar el sistema
- El proceso debe ser capaz de generar los flujos de datos de salida a partir de los flujos de datos de entrada mas una información (constante o variable) del proceso, lo que se conoce como "la regla de conservación de los datos"
- Cuando al proceso no le llegan todos los datos necesarios para obtener los datos de salida diremos que hay un "error de conservación de datos". Por lo tanto se habrá olvidado incluir ciertos datos de entrada
- También puede ocurrir el caso contrario, que seria cuando el flujo de datos o algún componente suyo, muere dentro del proceso, por lo que no se utiliza para generar flujo de salida y se denomina "perdida de información"
- Los procesos deben ir numerados y nominados
- Deben ser los mas representativos posible de la función que especifica
- El nombre debe englobar a toda la función y no a parte de ella y ser breve
- El nombre y numero del proceso deben ser únicos en el conjunto de DFD que representan al sistema
- Cuando se realizan los DFD lógicos, los procesos deben estar desligados de cualquier connotación física

Almacenes de Datos

- Representan información del sistema almacenada de forma temporal, representando datos que se encuentran "en reposo"
- Se trata de dispositivos lógicos de almacenamiento y por lo tanto pueden representar a cualquier dato temporalmente almacenado, independientemente del dispositivo utilizado
- Su representación grafica depende de la metodología utilizada
- Todos los almacenes de datos deben llevar su nombre, que debe ser lo mas representativo posible de los datos almacenados en el mismo
- Un almacén de datos se puede representar varias veces en el mismo DFD si con ello se mejora la legibilidad del diagrama
- Dentro del conjunto de DFD nivelados, el almacén se situara en el nivel mas alto de los que sirven de interconexión entre dos o mas procesos en el que se representan todos sus accesos y además se representara en los niveles inferiores
- Si en un DFD hay un almacén que solo tiene conexión con un proceso, se dice que el almacén es local y por tanto no debe aparecer en ese nivel. Dicho almacén se representara en el DFD en que se especifique dicho proceso
- Un almacén se dice que tiene una estructura simple cuando es de tipo registro, es decir, esta formado por una sucesión de atributos en el que uno o varios identifican una ocurrencia en el almacén
- El contenido de los almacenes debe definirse en el DD
- El contenido de un almacén con una estructura compleja es conveniente representarlo mediante un Diagrama Entidad InterRelación

Entidades Externas

- Representan un generador o consumidor de información del sistema y que no pertenece al mismo
- Puede representar un subsistema, una persona, departamento, organización, etc., que proporcione datos al sistema
- Su representación gráfica varía con la metodología
- Al ser externas al sistema, los flujos que fluyen en ellas o los que llegan a ellas definen la interfaz entre el sistema y el mundo exterior
- Su nombre debe ser representativo y puede aparecer varias veces en un mismo DFD si con ello se le aporta claridad y legibilidad
- Normalmente solo aparecen en el, DFD de mayor nivel denominado Diagrama de Contexto, aunque se puede incluir en otros diagramas de nivel inferior

Flujo de datos

- Se define como “Un camino a través del cual viajan datos de composición conocida de una parte del sistema a otra”
- Se representan por arcos dirigidos
- Según su persistencia en el tiempo pueden ser discretos o continuos
- Los flujos de datos **discretos** representan datos en movimiento en un momento determinado (como por ejemplo, la petición de datos al autoanalizador)
- Los flujos de datos **continuos** son un caso específico de los flujos de datos discretos que representan los flujos de datos persistentes en el tiempo (como por ejemplo, un proceso que comprueba continuamente el estado anómalo de los resultados de los análisis o el estado del autoanalizador por si este está listo para enviar información)
- Existen solo determinadas conexiones permitidas por medio de los flujos de datos
- Se observa que existen dos tipos de conexiones que relacionan las entidades externas y los almacenes (*) que indica que solo en el caso en que el almacén de datos externo sirva de interfaz entre el sistema y la entidad externa será permitido este tipo de conexión y solo aparecerá en el diagrama de contexto
- La conexión directa entre dos procesos mediante un flujo de datos es posible siempre y cuando la información sea síncrona, es decir que el proceso destino comience en el momento en que el proceso origen finaliza su función

Las diferentes conexiones que se pueden hacer entre los procesos y almacenes son:

- **Flujo de Consulta**, muestra la utilización de la información del almacén por el proceso para una de las siguientes opciones:
 - Utiliza los valores de uno o mas atributos de una de las ocurrencias del almacén
 - Comprueba si los valores de los atributos seleccionados cumplen unos criterios determinados
- **Flujo de Actualización**, indica que el proceso va a alterar la información mantenida en el almacén para.
 - Crear una nueva ocurrencia de una entidad o interrelación existente en el almacén
 - Borrar una o mas ocurrencias de una entidad o interrelación
 - Modificar el valor de un atributo
- **Flujo de Dialogo**, representa como mínimo un flujo de consulta y uno de actualización que no tienen relación directa
 - Tienen que tener un nombre representativo del contenido de la información que fluye a través de él
 - No tiene que estar nominados aquellos flujos que entran o salen de almacenes que tienen una estructura simple, siendo en estos casos su estructura la misma que la del almacén
 - El contenido del flujo de datos puede ser de varios tipos:
 - **Elemento**, es el que contiene un dato elemental, un elemento de información indivisible
 - **Grupo**, es un flujo de datos discreto que contiene varios campos de datos
 - **Par de dialogo**, Solo se puede especificar este tipo en un flujo de doble flecha en el que se incluyen dos nombres, uno es el iniciador y el otro la respuesta asociada el primero
 - **Múltiple**, esta formado por un conjunto de flujos de datos
 - Los flujos de datos no indican el control de ejecución de los procesos, ni cuando va a comenzar y terminar un proceso

Niveles de Abstracción en los DFD

- Cuando representamos un modelo de un sistema grande es conveniente hacerlo por capas, realizando una aproximación descendente (top-down) en el que cada nivel proporciona una aproximación mas detallada de una parte definida en el nivel anterior. Esta forma de estudio del sistema nos proporciona una serie de ventajas como:
 - Ayuda a construir la especificación de arriba a abajo
 - Los distintos niveles pueden ir dirigidos a personas diferentes
 - Facilita el trabajo de los analistas que pueden trabajar paralelamente modelando funciones del sistema
 - Facilitan la documentación del sistema
- El DFD de un sistema es realmente un conjunto de DFDs dispuestos jerárquicamente
- La raíz de la jerarquía es el “Diagrama de Contexto” que es el mas general de todos
- **Diagramas de contexto** (también conocido de nivel 0). Delimita la frontera del sistema con el mundo exterior y define sus interfaces es decir los flujos de entrada y salida con el entorno, es decir el contexto
- **Diagrama de niveles.** Representan las funciones principales que debe realizar el sistema y sus relaciones
- **Procesos primitivos.** DFD que ya no se descomponen en más diagramas de nivel inferior. Para cada función primitiva habrá una especificación que la describa
- La decisión de hasta que nivel realizar la descomposición es del analista y por lo tanto es subjetiva, podemos establecer una serie de reglas para la guía en este proceso:
 - Cuando un requisito funcional se puede especificar en menos de una pagina mediante un lenguaje de especificación (formal o no)
 - Cuando los procesos del diagrama tiene pocos flujos de datos de entrada y salida
 - Cuando al descomponer una función de un nivel determinado se pierde el significado de lo que tiene que hacer esa función, así pues tendremos:
 - ✓ Nivel 0, Diagrama de contexto
 - ✓ Nivel 1, Diagrama de subsistemas
 - ✓ Nivel 2, Diagramas de funciones de los subsistemas
 - ✓
 - ✓ Nivel N, Diagramas de los procesos necesarios para el tratamiento de cada subfunción

- También es necesario comprobar la consistencia entre los distintos niveles del DFD, es decir, que la información que entra y sale de un proceso representado en un DFD de nivel N, sea consistente con la información que entra y sale del DFD en el que se descompone.
- Para ello se debe seguir la **regla del balanceo** entre los diferentes niveles, la cual se enuncia de la siguiente forma:

"Todos los flujos de datos que entran en un diagrama hijo deben estar representado en el diagrama padre por el mismo flujo de datos entrando en el proceso asociado. Las salidas del diagrama hijo deben ser las mismas salidas del proceso padre asociado, exceptuando el caso de los rechazos triviales"

- La numeración de los DFD se realiza de la forma siguiente:
 - Cada diagrama recibe el numero y nombre del proceso que se descompone "proceso padre"
 - El Diagrama de contexto se numera como 0
 - Los Diagramas de nivel 1, se numeran desde el 1 hasta el N
 - El resto se numeran considerando la numeración anidada (Dewey)
- En general hay que tratar de evitar una partición desigual
- Aplicar criterios de independencia funcional para que el sistema sea ampliable y modificable sin dificultad.

Capítulo 6

INTRODUCCIÓN AL DISEÑO

¿Qué es el Diseño?

- Es el proceso de aplicar distintas técnicas y principios con el propósito de definir un dispositivo, un proceso o un sistema con suficiente detalle como para permitir su realización física
- Sus objetivos son:
 - Implementar todos los requisitos explícitos
 - Ser la guía para quienes construyan, prueben y mantengan el código
 - Proporcionar una idea completa del software
- Proceso iterativo a través del cual se traducen los requisitos en una representación del software
- El diseño estructurado es un enfoque disciplinado de la transformación que es necesaria para el desarrollo de un sistema, a cómo deberá ser hecha la implementación

Actividades del Diseño

- Diseño de Datos: diseño de archivos / tablas
- Diseño Arquitectónico: define una estructura modular
- Diseño de Interfaz: define cómo el software se comunica consigo mismo, con los sistemas que operan con él y con los operadores que lo emplean
- Diseño procedural o de las funciones: diseño de los algoritmos

Principios del diseño

- “El principio de la sabiduría de un ingeniero de software es *reconocer la diferencia entre conseguir que funcione un programa y hacerlo bien*” Jackson
- El proceso de diseño implica la selección de la mejor alternativa
- Un buen diseño está formado por decisiones justificadas
- Se tiene que poder fácilmente recorrer las etapas (**traceability**: marcar una traza, seguir un rastro) o sea que un requerimiento debe estar expresado en el diseño de alguna forma
- Las decisiones de diseño deben estar soportadas por un requerimiento o una conveniencia de Diseño
- Aplicar la **reusabilidad** siempre que sea posible. Reusar módulos (funciones)
- Debe ser **uniforme** definiendo normas de estilo y formato a cumplir por los implicados
- Debe ser **integrado**, definiendo claramente el acoplamiento entre módulos
- **Diseñar no es escribir código** y escribir código no es diseñar. El nivel de abstracción del diseño es diferente al de la codificación y las decisiones de diseño no se deben hacer en la etapa de codificación; solamente se deben tomar decisiones de implementación
- Se debe valorar la **calidad** del diseño mientras que se crea y no una vez finalizado
- Se debe estructurar para **admitir cambios**
- **Minimizar la distancia intelectual** entre el software y el problema
- Se debe revisar para eliminar errores

El proceso del diseño

- La fase de desarrollo de un sistema software absorbe el 75% del coste del ciclo de desarrollo, sin considerar el mantenimiento, ya que aquí en estas etapas es donde se toman decisiones que afectan al éxito de la implementación del sistema y a su facilidad de mantenimiento, por lo tanto se asienta la calidad del software
- Desde el punto de vista de Gestión:
 - **Diseño Preliminar**, se refiere a la transformación de los requisitos en datos y arquitectura del software
 - **Diseño Detallado**, se enfoca hacia los refinamientos de la representación arquitectónica que conduce a una estructura de datos detallada y a representaciones algorítmicas del software

Conceptos fundamentales del Diseño



Abstracción

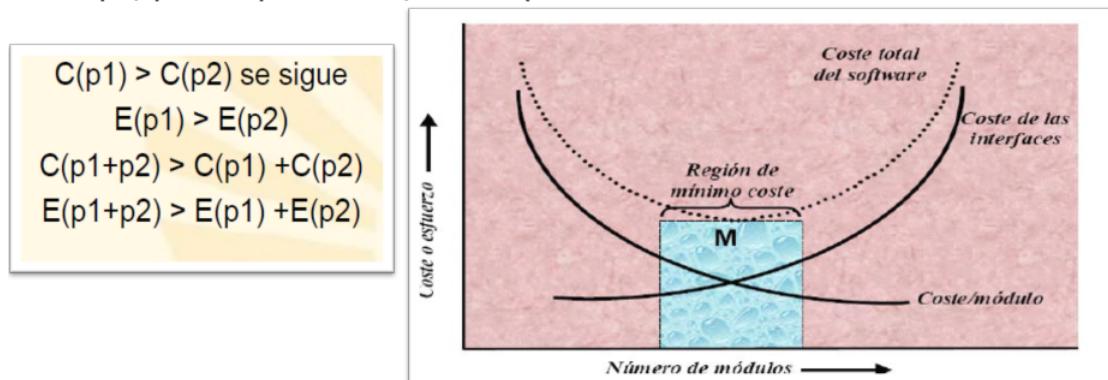
- Herramienta intelectual que permite trabajar con los conceptos independientemente de las instancias particulares de estos, permitiendo la separación de los aspectos conceptuales de los procedimientos
- Los tipos que podemos considerar son:
 - Funcional, incluye el uso de subprogramas parametrizados
 - De datos, especificación de datos o de los objetos por medio de especificar operaciones permitidas sobre los objetos. Aquí son eliminados los detalles de la representación y manipulación
 - De control, es utilizada para establecer un efecto deseado sin necesidad de definir el mecanismo exacto del control
- En los diferentes niveles del diseño, se crean abstracciones procedimentales, de datos y de control

Refinamiento

- Es un proceso de elaboración que permite ampliar una declaración original, dando cada vez más detalles
- El refinamiento paso a paso (Wirth, 1971) es una estrategia de diseño descendente donde la arquitectura de un programa se desarrolla refinando sucesivamente niveles de detalle procedural
- Se empieza con un enunciado de función o descripción de información definida a un alto nivel de abstracción sin proporcionar ninguna información sobre los procesos internos de la función o la estructura interna de la información
- Ayuda al diseñador a revelar detalles de bajo nivel a medida que progresá el diseño

Modularidad

- Atributo del software que permite que sea intelectualmente manejable al estar dividido en componentes que se integran para satisfacer los requisitos
- La arquitectura del software conlleva modularidad; el software se divide en componentes identificables y tratables por separado, denominados módulos, que están integrados para satisfacer los requisitos del programa
- Módulo: Componente bien definido de un sistema software. Autónomo y autocontenido
- Sistema modular = Σ módulos
- Los beneficios que tiene es facilitar los factores de calidad del software
- Es un enfoque ampliamente aceptado en todas las disciplinas, reduciendo la complejidad, facilitando los cambios y permitiendo el desarrollo en paralelo
 - Sea $C(x)$ una función que define la complejidad percibida de un problema
 - $E(x)$ una función que define el esfuerzo (en tiempo) requerido para solucionar el problema x
 - Y p_1, p_2 dos problemas, se cumple:



Jerarquía de Control

- La estructura del programa representa la organización de los componentes e implica una jerarquía de control.
- La jerarquía de control representa dos características de la arquitectura del programa:
 - La **visibilidad** indica el conjunto de componentes que pueden ser invocados o sus datos ser utilizados por un componente dado (incluso indirectamente)
 - La **conectividad** indica el conjunto de componentes a los que directamente se invoca o sus datos son utilizados en un determinado módulo
- Se pueden realizar diferentes medidas de la estructura
- No representa aspectos procedimentales del software, tales como secuencia de procesos, ocurrencias, orden de decisiones o repetición de operaciones

- La profundidad es el nº de niveles de control
- La anchura es la amplitud global del control
- El grado de salida es el nº de módulos controlados por otros módulos
- El grado de entrada es el nº de módulos que controlan directamente a un módulo dado

Procedimientos Software

- Se centra en los detalles de procesamiento de cada módulo individualmente
- Debe proporcionar una especificación exacta del procesamiento, incluyendo la secuencia de acontecimientos, puntos exactos de decisión, operaciones repetitivas e incluso la organización o estructura de los datos
- Existe una relación entre la estructura de programa y el procedimiento del software; el procesamiento indicado para cada módulo debe incluir una referencia a todos los módulos subordinados al módulo que se describe

Ocultación de la Información

- Capacidad de un componente para que la información (procedimiento y datos) contenida dentro del mismo sea inaccesible a otros componentes que no la necesitan
- Implica que se puede conseguir una modularidad eficaz definiendo un conjunto de módulos independientes que se comunican entre ellos sólo la información necesaria para conseguir la función del software
- Define y refuerza las restricciones de acceso tanto al detalle procedural dentro del módulo como a cualquier estructura de datos local empleada por el módulo
- Proporciona ventajas en el mantenimiento del software, ya que los errores son más difíciles de propagar al estar ocultos la mayoría de los datos y los procedimientos a otras partes del software

Concurrencia

- Los sistemas software pueden ser clasificados en secuenciales y concurrentes. En unos solo una porción esta activa. En otros los procesos son independientes y activados de forma simultanea
- En los sistemas concurrentes existen problemas específicos como:
 - Condición de bloqueo: condición indeseable que ocurre cuando todos los procesadores de un sistema se quedan esperando a otros
 - Exclusión mutua: necesaria para evitar la actualización de los mismos componentes de un sistema compartido
 - Sincronización: requerida para que los procesos concurrentes puedan comunicarse en los puntos adecuados

Verificación

- Es el puente entre los requisitos y la instrumentación que satisface a esos requisitos
- Un diseño es verificable si puede demostrarse que el diseño generará el producto que satisface los requisitos del cliente. Y se desarrolla generalmente en dos pasos:
 - Verificando que la definición de los requisitos de programación satisface las necesidades del usuario (verificación de requisitos)
 - Verificando que el diseño satisface la definición de los requisitos (verificación del diseño)

Estética

- Las consideraciones estéticas son fundamentales para el diseño
- La simplicidad, elegancia y claridad de un propósito distinguen a los productos de alta calidad de los mediocres
- Un producto estéticamente agradable es fácilmente reconocido

El Diseño Modular

- Un diseño modular reduce la complejidad, facilita los cambios y da como resultado una implementación más fácil, posibilitando el desarrollo paralelo de diferentes partes del sistema
- La abstracción y la ocultación de la información se usan para definir módulos dentro de una arquitectura del software. Externamente estos conceptos son traducidos en características operacionales del modulo como:
 - Historia del tiempo de incorporación (lenguaje fuente)
 - Mecanismo de activación (call/interrupciones)
 - Camino de control, describe la forma en que se ejecuta internamente (secuencial/reentrante)

Tipos de Módulos

- **Módulos secuenciales.** Se referencia y ejecuta sin interrupción aparente. Son los que se presentan mas frecuentemente. Macros en tiempo de compilación, subprogramas convencionales, subrutinas, funciones o procedimientos
- **Módulo incremental (corrutina).** Puede ser interrumpido antes de la terminación por software de la aplicación. Mantiene un puntero que permite al modulo restablecer el punto de interrupción. Tales módulos son utilices en sistemas conducidos por interrupciones
- **Módulos paralelos (conrutas).** Se ejecuta simultáneamente con otro módulo en entornos de multiprocesadores paralelos. Se encuentran en cálculos de alta velocidad, como el procesamiento en casada que necesita de dos o mas CPUs trabajando en paralelo

Características de los Módulos

- Contienen instrucciones, lógica de proceso y estructura de datos
- Pueden ser compilados independientemente
- Pueden quedar incluidos dentro del programa
- Los segmentos de un módulo pueden ser utilizados por medio de invocar un nombre con algunos parámetros
- Los módulos pueden usar a otros módulos

Independencia funcional

- Se consigue desarrollando módulos con una función única y una disminución de la interacción entre los módulos
- Tendemos a definir módulos que trate una subfunción específica de los requisitos y tenga una sencilla interfaz. Se mide usando dos criterios cualitativos:
 - Cohesión: es una medida de la fuerza relativa funcional de un módulo
 - Acoplamiento: es una medida de la interdependencia relativa entre los módulos

Cohesión

- Es una medida de la fuerza funcional estática de un módulo. Es una extensión del concepto de ocultación de la información
- Un modulo coherente ejecuta una tarea sencilla en un procedimiento software y requiere poca interacción con procedimientos que se ejecutan en otras partes del programa
- La escala de cohesión se representa mediante un espectro que no es lineal, es decir una cohesión baja es mucho peor que una media, que ya es buena
- El objetivo de la cohesión es diseñar servicios robustos y altamente cohesionados cuyos elementos estén fuerte y genuinamente relacionados entre si
- Las ventajas son que favorece la comprensión y el cambio de los sistemas

Niveles de Cohesión

- **Coincidental**, un módulo ejecuta un conjunto de tareas que están relacionadas con otras débilmente
- **Lógica**, se da cuando un módulo contiene actividades de la misma categoría
- **Temporal**, las tareas están relacionadas por tenerse que producir a un mismo tiempo
- **Procedimental**, los elementos de procesamiento de un modulo están relacionados y deben ejecutarse en un orden específico
- **Comunicación**, cuando todos los elementos de procesamiento se concentra en un área sobre una estructura de datos
- **Secuencial**, ocurre cuando la salida de un elemento es la entrada del siguiente
- **Funcional**, realiza una única tarea

Niveles de Cohesión y Acoplamiento

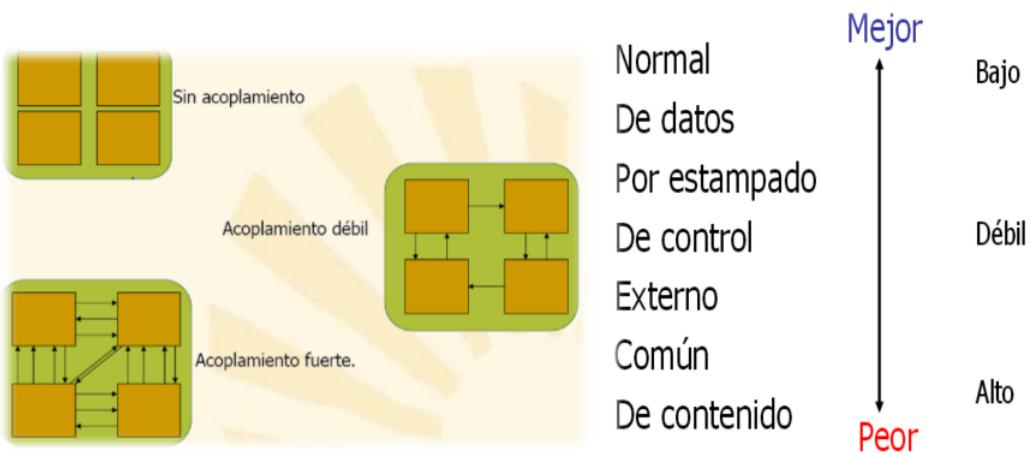
	Niveles de cohesión	Niveles de acoplamiento
Bajo ↓ Alto	Coincidental Lógica Temporal Procedimental Comunicativa Secuencial Funcional	Sin acoplamiento directo De datos Por estampado De control Externo Común Del contenido

Niveles de Cohesión

- Una alta cohesión se caracteriza por un módulo en el que se ejecuta una tarea procedural distinta
- No es necesario determinar el nivel preciso de cohesión, en vez de ello, es importante buscar una cohesión alta y reconocer la baja de forma que el diseño pueda modificarse
- Según Stevens, propone estimar la cohesión de un modulo mediante su descripción por medio de una sentencia, y ver:
 - Si es compuesta, contiene comas o mas de un verbo, es probable que ejecute mas de una función (C. secuencial o de comunicación)
 - Si contiene palabras relativas al tiempo, tales como: primero, a continuación, entonces, después, cuando, etc. (C. secuencial o temporal)
 - Si el predicado no contiene un objeto específico sencillo después del verbo es probable una C. lógica
 - Palabras como inicializar, limpiar, etc. (C. temporal)

Acoplamiento

- Es una medida de la independencia relativa entre los módulos, siendo una medida de la interconexión entre módulos en una estructura de programa
- Se busca el acoplamiento mas bajo posible para evitar el efecto "onda" en la propagación de errores
- El acoplamiento depende de la complejidad de la interfaz entre los módulos, la decisión que hace referencia a otro modulo y los datos que pasan a través de la interfaz
- Se produce una situación de acoplamiento cuando un elemento de un diseño depende de alguna forma de otro elemento del diseño
- El objetivo es conseguir el acoplamiento lo mas bajo posible, que nos indica un sistema bien dividido y puede conseguirse mediante la eliminación o reducción de relaciones innecesarias
- Un bajo acoplamiento implicará que un cambio en un componente no implicará un cambio en otro



Tipos de Acoplamiento

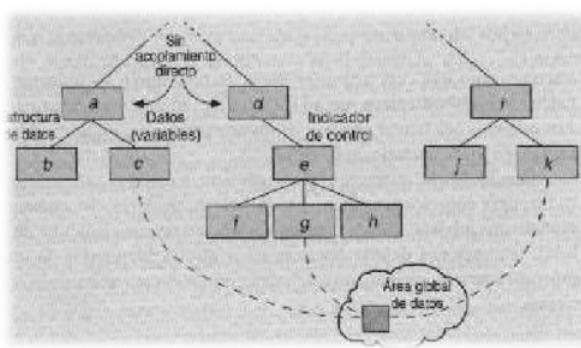
- **Acoplamiento normal (N. bajo)**
 - Dos módulos A y B están normalmente acoplados si: un módulo A llama a otro B, y B retorna el control a A.
 - Dos módulos están acoplados normalmente cuando no se pasan ningún parámetro entre ellos, sólo existe la llamada de uno a otro
- **Acoplamiento de datos (N. bajo)**
 - Los módulos se comunican mediante parámetros
 - Cada parámetro es una unidad elemental de datos
 - Todas las E/S al y desde el módulo llamado son argumentos de datos y no de control
 - Reducir el número de parámetros
- **Acoplamiento por estampado (N. bajo)**
 - Dos módulos están acoplados por estampado si ambos hacen referencia a la misma estructura de datos
 - Evitar estructura de datos globales
 - No es deseable si el modulo que recibe esa estructura de datos necesita solo parte de los elementos que se le pasan
- **Acoplamiento de control (N. medio)**
 - Paso de parámetros de control entre módulos
 - Influencia de un módulo en la ejecución de otro, donde una variable controla las decisiones en un módulo superior o subordinado
- **Acoplamiento externo (N. medio)**
 - Los módulos están ligados a un entorno externo al software. Por ejemplo la E/S acopla un módulo a dispositivos, formatos y protocolos de comunicación
 - Este acoplamiento es esencial, pero deberá estar limitado a unos pocos módulos
- **Acoplamiento común (N. alto)**
 - Los módulos acceden a datos en un área de datos global (un área de memoria accesible por ejemplo). Comparten una estructura de datos global
 - Viola los principios de encapsulamiento y modularidad
 - El diagnostico de problemas en estructuras con acoplamiento común es costoso en tiempo y difícil de realizar

- Acoplamiento de contenido (N. alto)

- Se da cuando un módulo hace uso de datos o de información de control mantenidos dentro de los límites de otro módulo

- ✓ Un módulo modifica algún elemento en el otro módulo
- ✓ Un módulo utiliza una variable local del otro
- ✓ Desde un módulo se salta a otro, pero la sentencia a la que se pasa no está definida como punto de entrada
- ✓ Dos módulos comparten los mismos contenidos

- Inaceptable



Heurísticas de diseño para una modularización efectiva

- Evaluar el planteamiento inicial de la jerarquía de control para reducir el acoplamiento y mejorar la cohesión
- Intentar minimizar las estructuras con muchos grados de salida e intentar concentrarlas a medida que aumenta la profundidad
- Mantener el alcance del efecto de un módulo dentro del alcance del control de ese módulo
 - El ámbito del efecto de un modulo E, son todos los módulos afectados por la decisión que se toman en E
 - El alcance de control son todos los módulos subordinados a el
- Evaluar las interfaces de los módulos para reducir la complejidad, la redundancia y mejorar la consistencia
- Empaquetar el software basándose en las restricciones de diseño y los requisitos de portabilidad

Capítulo 7

INTRODUCCIÓN A LAS PRUEBAS DEL SOFTWARE

Introducción

- Las prueba de software es un elemento crítico para la Garantía de la calidad del producto de programación y representa un ultimo repaso de las especificaciones, del diseño y de la codificación
- La creciente aparición del software como un elemento mas de muchos sistemas y la importancia de los costes asociados a un fallo del mismo están motivando la creación de pruebas minuciosas y bien planificadas, dando lugar, hasta en ocasiones, al 40% del esfuerzo total del desarrollo de un producto de programación
- La prueba representa una interesante anomalía para el ingeniero del software. Durante las fases de definición y desarrollo el ingeniero de software intenta construir el software partiendo de un control abstracto y llegando a una implementación tangible. Después en la prueba el ingeniero crea una serie de casos de prueba que intentan demoler el software que ha sido construido

- Algunas de las definiciones que se utilizan en esta fase del ciclo de vida son según el estándar IEEE, 1990:
 - Pruebas (*test*): es una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan y registran y se realiza una evaluación de algún aspecto
 - Caso de prueba (*test case*): un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular
 - Defecto (*defect, fault, «bug»*): un defecto en el software como, por ejemplo, un proceso, una definición de datos o un paso de procesamiento incorrectos en un programa. Se produce cuando una persona comete un error. Es una vista interna, lo ven los desarrolladores
 - Fallo (*failure*): La incapacidad de un sistema o de alguno de sus componentes para realizar las funciones requeridas dentro de los requisitos de rendimiento especificados. Es un desvió respecto del comportamiento esperado del sistema, puede producirse en cualquier etapa. Es una vista externa, lo ven los usuarios. Es un resultado incorrecto
 - Error: Es una acción humana que conduce a un resultado incorrecto

Objetivos de las Pruebas

- La prueba exhaustiva del software es impracticable (no se pueden probar todas las posibilidades de su funcionamiento) ni siquiera en programas sencillos
- El objetivo de las pruebas es la detección de defectos en el software (descubrir un error es el éxito de una prueba)
- Mito
 - Un defecto implica que somos malos profesionales y que debemos sentirnos culpables → todo el mundo comete errores
- El descubrimiento de un defecto significa un éxito para la mejora de la calidad

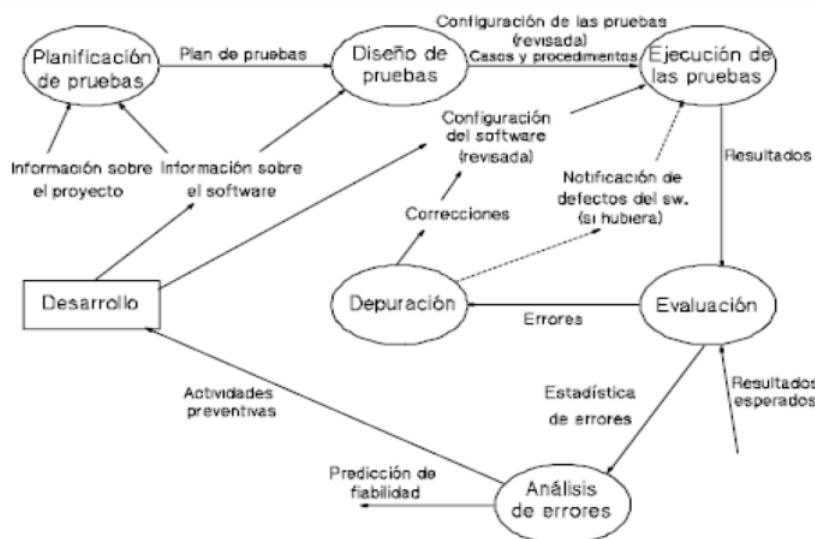
Así, para la realización de las pruebas se debe seguir la siguiente filosofía o intención de trabajo:

- Cada caso de prueba debe definir el resultado de salida esperado que se comparará con el realmente obtenido
- El programador debe evitar probar sus propios programas, ya que desea (consciente o inconscientemente) demostrar que funcionan sin problemas
- Además, es normal que las situaciones que olvidó considerar al crear el programa queden de nuevo olvidadas al crear los casos de prueba
- Se debe inspeccionar a conciencia el resultado de cada prueba, y así, poder descubrir posibles síntomas de defectos
- Al generar casos de prueba, se deben incluir tanto datos de entrada válidos y esperados como no válidos e inesperados

- Las pruebas deben centrarse en dos objetivos (es habitual olvidar el segundo):
 - Probar si el software no hace lo que debe hacer
 - Probar si el software hace lo que no debe hacer, es decir, si provoca efectos secundarios adversos
- Se deben evitar los casos desecharables, es decir, los no documentados ni diseñados con cuidado. Ya que suele ser necesario probar muchas veces el software y por tanto hay que tener claro qué funciona y qué no
- No deben hacerse planes de prueba suponiendo que, prácticamente, no hay defectos en los programas y, por lo tanto, dedicando pocos recursos a las pruebas → siempre hay defectos
- La experiencia parece indicar que donde hay un defecto hay otros, es decir, la probabilidad de descubrir nuevos defectos en una parte del software es proporcional al número de defectos ya descubierto
- Las pruebas son una tarea tanto o más creativa que el desarrollo de software. Aunque siempre se han considerado las pruebas como una tarea destructiva y rutinaria

Es interesante planificar y diseñar las pruebas de manera sistemática para poder detectar el máximo número y variedad de defectos con el mínimo consumo de tiempo y esfuerzo

Proceso de prueba



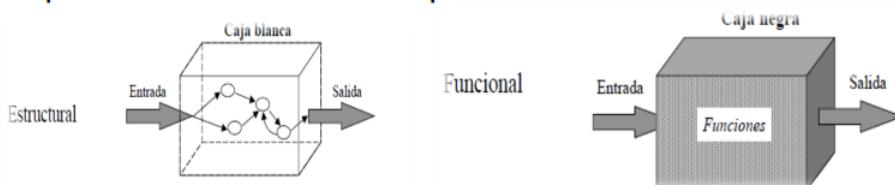
- En el proceso de pruebas se realizan una serie de etapas o pasos, que son:
 - Generación del [plan de pruebas](#) en base a la documentación sobre el proyecto y la documentación sobre el software a probar
 - Posteriormente se [diseñan las pruebas](#) específicas
 - Para la [ejecución de las pruebas](#) se considera la configuración del software (para verificar que es la versión apropiada del software) y se ejecutan sobre ella los casos. En algunas ocasiones se trata de reejecuciones de pruebas, por lo que es conveniente tener constancia de los defectos ya detectados aunque aun no corregidos
 - Posteriormente con la salida obtenida se pasa a la [evaluación](#) mediante comparación con la salida esperada, pudiendo a partir de este momento realizar dos actividades:
 - ✓ La depuración (localización y corrección de defectos)
 - ✓ El análisis de la [estadística de errores](#). Sirve para realizar predicciones de la fiabilidad del software y para detectar las causas más habituales de error y por tanto mejorar los procesos de desarrollo
- A medida que se van recopilando y evaluando los resultados de la prueba se puede tener una medida cualitativa de la calidad y fiabilidad del software.
- Si se encuentran con frecuencia errores serios, que requieren modificaciones en el diseño podríamos considerar que la calidad y fiabilidad del software queda en entredicho y sería necesario considerar pruebas posteriores
- Si por el contrario, el funcionamiento del software parece ser correcto y los errores que se encuentran son fácilmente corregibles, podemos considerar una de las siguientes conclusiones:
 - La calidad y fiabilidad del software son aceptables
 - Las pruebas son inadecuadas para descubrir errores serios

Técnicas de diseño de Casos de Prueba

- Ya que el diseño de casos de prueba está mediatisado por la imposibilidad de probar exhaustivamente el software, el objetivo de dichas técnicas es conseguir una confianza aceptable en que se detectaran los defectos existentes sin necesidad de consumir una cantidad excesiva de recursos
- La idea fundamental para el diseño de casos de prueba consiste en elegir algunas de ellas que, por sus características, se consideren representativas del resto
- Por tanto podremos asumir que si no se detectan defectos en el software al ejecutar dichos casos podemos tener un cierto nivel de confianza en que el programa no tiene defectos
- La dificultad estará en saber elegir los casos de prueba adecuados en cada situación

Existen tres enfoques principales para el diseño de casos:

- El enfoque **estructural** o de caja blanca. Se centra en la estructura interna del programa (analiza los caminos de ejecución)
- El enfoque **funcional** o de caja negra. Se centra en las funciones, entradas y salidas
- El enfoque **aleatorio** consiste en utilizar modelos (en muchas ocasiones estadísticos) que representen las posibles entradas al programa para crear a partir de ellos los casos de prueba



- Estos enfoques no son excluyentes entre si, por lo que se pueden combinar para conseguir una detección de defectos mas eficaz

Pruebas Estructurales en las Técnicas de Diseño de Casos de prueba

- El diseño de casos de prueba tiene que estar basado en la elección de caminos importantes que ofrezcan una seguridad aceptable de que se descubren defectos (un programa de 50 líneas con 25 sentencias if en serie da lugar a 33,5 millones de secuencias posibles), para lo que se usan los criterios de cobertura lógica
- Para ello se pueden derivar casos de prueba que:
 - Garanticen que se ejecutan por lo menos una vez todos los caminos independientes de cada módulo
 - Se ejecutan todas las decisiones lógicas en sus caras verdaderas y falsas
 - Se ejecutan todos sus bucles en sus límites
 - Se ejecutan las estructuras de datos internas para asegurar su validez
- La importancia de estas pruebas esta justificada por la propia naturaleza de los defectos del software, ya que :
 - Los errores tienden a reproducirse en nuestro trabajo cuando diseñamos e implementamos funciones, condiciones o controles que se encuentran fuera de lo normal
 - A menudo creemos que un camino lógico tiene pocas posibilidades de ejecutarse, cuando en ocasiones se puede ejecutar de forma normal

- Conviene señalar que estas técnicas no requieren el uso de ninguna representación gráfica específica de software, aunque es habitual tomar como base los llamados **Diagramas de Flujo de Control**. Que se realizan siguiendo los siguientes pasos:
 - Se señala sobre el código o pseudocódigo cada condición de cada decisión, tanto en sentencias **IF-THEN**, como **CASE-OF** como en los bucles **WHILE** o **UNTIL**
 - Se agrupa el resto de las sentencias en secuencias situadas entre cada dos condiciones según los esquemas de representación de las estructuras básicas
 - Se numeran tanto las condiciones como los grupos de sentencias, de manera que se les asigne un identificador único. Es recomendable alterar el orden en el que aparecen las condiciones en una decisión multicondicional, situándolas en orden decreciente de restricción. El objetivo es facilitar la derivación de casos de prueba una vez obtenido el grafo

Pruebas del camino básico en las Técnicas de Diseño de Casos de prueba

- La cobertura de caminos (secuencia de sentencias) es el criterio mas importante a considerar en este tipo de pruebas el primero a realizar. Se define un camino como:

La secuencia de sentencias encadenadas desde la sentencia inicial del programa hasta su sentencia final

- Como ya hemos comentado el numero de caminos a probar puede ser impracticable, incluso para programas pequeños. Para reducirlo se introduce el concepto:

Camino de prueba, que es un camino del programa que atraviesa, como máximo, una vez el interior de cada bucle que encuentra

- McCabe propuso la prueba del camino básico como un tipo de prueba de la caja blanca
- Es un indicador del numero de caminos independientes que existe en un grafo
- El numero de caminos independientes viene dado por la **Complejidad Ciclomática** $V(G)$

- La Complejidad Ciclomática aporta el límite superior para el número de pruebas que se deben realizar para asegurar que se ejecuta cada sentencia al menos una vez

Un camino independiente es cualquier camino del programa que introduce por lo menos un nuevo conjunto de sentencias de procesamiento o una nueva condición. En términos del grafo de flujo de control, un camino independiente se debe de mover por lo menos por una arista que no haya sido recorrida anteriormente

La complejidad de McCabe se puede calcular de cualquiera de estas 3 formas

- $$\left\{ \begin{array}{l} 1. V(G) = a - n + 2, \text{ siendo } a \text{ el número de arcos o aristas del grafo y } n \text{ el número de nodos.} \\ 2. V(G) = r, \text{ siendo } r \text{ el número de regiones cerradas del grafo.} \\ 3. V(G) = c + 1, \text{ siendo } c \text{ el número de nodos de condición.} \end{array} \right.$$

- Un **nodo predicado** es aquel que contiene una condición y se caracteriza porque de él emergen dos o mas aristas
- El criterio de prueba de McCabe es: Elegir tantos casos de prueba como caminos independientes (calculados como $V(G)$)
- La experiencia en este campo asegura que:
 - $V(G)$ marca el límite mínimo de casos de prueba para un programa
 - Cuando $V(G) > 10$ la probabilidad de defectos en el módulo o programa crece mucho → quizás sea interesante dividir el módulo
- El procedimiento para derivar los casos de prueba es:
 - Usando el diseño o el código como base se dibuja el grafo de flujo de control
 - Se determina la complejidad ciclomática de dicho grafo
 - Se determina un conjunto de caminos linealmente independientes
 - Se preparan los casos de prueba que forzaran la ejecución de cada camino del conjunto básico
- Es posible mecanizar la determinación del conjunto básico de caminos. Para ello se utilizará una matriz del grafo, consistente en una matriz cuadrada igual al numero de nodos y cuyas entradas corresponden a las aristas, es decir las conexiones entre los nodos, recibiendo también el nombre de **matriz de conexiones**
- El proceso para el calculo de la complejidad ciclomática es el siguiente:
 - Se genera una nueva columna en la matriz cuyos elementos son el sumatorio de los elementos de la fila correspondiente menos una unidad
 - El sumatorio de los elementos de esta columna mas una unidad es la complejidad ciclomática
- Esta matriz de conexiones es muy útil, ya que los elementos de la misma pueden contener información añadida de interés, como:
 - La probabilidad de que una arista sea ejecutada
 - El tiempo de ejecución del procedimiento asociado al recorrido del enlace
 - La memoria requerida o los recursos requeridos durante el recorrido del enlace
- Una vez calculada la complejidad ciclomática por los distintos métodos se procede a preparar los casos de prueba que forzaran la ejecución de cada camino del conjunto básico

Pruebas de Condiciones en las técnicas de diseño de Casos de Prueba

- Tipos de errores que pueden aparecer en una condición:
 - Existe un error en un operador lógico
 - Existe un error en un paréntesis lógico
 - Existe un error en un operador relacional
 - Existe un error en una expresión aritmética

Prueba de los bucles en las técnicas de diseño de Casos de Prueba

- Es una técnica de prueba estructural que se centra exclusivamente en la validez de las construcciones de bucles
- Consideramos cuatro diferentes construcciones de bucle:
 - Pruebas para bucles simples
 - Pruebas para bucles anidados
 - Prueba para bucles concatenados
 - Pruebas para bucles no estructurados
- **Pruebas para Bucles simples.** (n es el número máximo de iteraciones permitidos por el bucle)
 - Pasar por alto totalmente el bucle
 - Pasar una sola vez por el bucle
 - Pasar dos veces por el bucle
 - Hacer m pasos por el bucle con $m < n$
 - Hacer $n-1$, n y $n + 1$ pasos por el bucle
- **Pruebas para Bucles anidados**
 - Comenzar en el bucle más interior estableciendo los demás bucles en sus valores mínimos.
 - Realizar las pruebas de bucle simple para el más interior manteniendo los demás en sus valores mínimos.
 - Avanzar hacia fuera confeccionando pruebas para el siguiente bucle manteniendo todos los externos en los valores mínimos y los demás bucles anidados en sus valores típicos.
 - Continuar el proceso hasta haber comprobado todos los bucles.
- **Pruebas para Bucles concatenados**
 - Siempre que los bucles concatenados sean independientes se puede aplicar lo relativo a bucles simples/anidados
 - En caso de ser dependientes se evaluarán como bucles anidados
- **Pruebas para Bucles no estructurados**
 - Siempre que sea posible esta clase de bucles se debe rediseñar para que se ajuste a las construcciones de la programación estructurada

Pruebas Funcionales en las técnicas de diseño de Casos de Prueba

- La prueba funcional o de la caja negra se centra en el estudio de la especificación del software, análisis de las funciones que debe realizar, de las entradas y de las salidas
- La prueba exhaustiva de la caja negra también es impracticable, por lo que también es necesario fijar un conjunto de criterios que permitan elegir un subconjunto de casos de prueba cuya ejecución aporte una cierta confianza en detectar los posibles defectos en el software
- Myers, aporta las dos definiciones siguientes para realizar esta elección:
 - El que reduce el número de otros casos necesarios para que la prueba sea razonable. Esto implica que el caso ejecute el máximo número de posibilidades de entrada diferentes para así reducir el total de casos
 - El que cubre un conjunto extenso de otros posibles casos, es decir, nos indica algo acerca de la ausencia o la presencia de defectos en el conjunto específico de entradas que prueba, así como de otros conjuntos similares
- Estas pruebas intentan descubrir defectos diferentes a los de la caja blanca, por lo que se consideran métodos complementarios y no alternativos
- Las categorías de errores que se intentan descubrir son:
 - Funciones incorrectas o ausentes
 - Errores de Interfaz
 - Errores en estructuras de datos o acceso a bases de datos externas
 - Errores de rendimiento
 - Errores de inicialización o terminación

Prueba de Partición o Clases de Equivalencia

Las cualidades que deben definir a un buen caso de prueba son:

- Cada caso debe cubrir el máximo número de entradas
- Debe tratarse el dominio de valores de entrada dividido en un número finito de clases de equivalencia, que cumplan la siguiente propiedad:

La prueba de un valor representativo de una clase permite suponer razonablemente que el resultado obtenido (existen defectos o no) será el mismo que el obtenido probando cualquier otro valor de la clase

- El método de diseño de casos consistirá entonces en:
 - Identificación de las clases de equivalencia
 - Creación de los casos de prueba correspondientes
- Para identificar las posibles clases de equivalencia se deben seguir los siguientes pasos:
 - Identificación de las condiciones de las entradas del programa, es decir, restricciones de formato o contenido de los datos de entrada
 - A partir de ellas, se identifican clases de equivalencia que pueden ser:
 - ✓ De datos válidos
 - ✓ De datos no válidos o erróneos

La identificación de las clases se realiza basándose en el principio de igualdad de tratamiento:

Todos los valores de las clases deben ser tratados de la misma manera en el programa

- Existen algunas reglas que ayudan a identificar las clases de equivalencia, como son:
 - Regla 1. Si se especifica un rango de valores para los datos de entrada, por ejemplo el numero estará comprendido entre 5 y 25, se creará una clase válida ($5 \leq \text{número} \leq 25$) y dos clases no válidas ($\text{número} < 5$) y ($\text{número} > 25$)
 - Regla 2. Si se especifica un numero de valores (por ejemplo se pueden registrar de uno a cuatro titulares de una cuenta bancaria), se creará una clase válida ($1 \leq \text{titulares} \leq 4$) y dos no válidas ($\text{titulares} < 1$) y ($\text{titulares} > 4$)
 - Regla 3. Si se especifica una situación del tipo «debe ser» o booleana (por ejemplo, «el primer carácter debe ser una letra»), se identifican una clase válida («es una letra») y una no válida («no es una letra»)
 - Regla 4. Si se especifica un conjunto de valores admitidos y se sabe que el programa trata de forma diferente cada uno de ellos, se identifica una clase válida por cada valor y una no válida (cualquier otro caso)
 - Regla 5. En cualquier caso, si se sospecha que ciertos elementos de una clase no se tratan igual que el resto de la misma, deben dividirse en clases menores
- El último paso del método es el uso de las clases de equivalencia para identificar los casos de prueba correspondientes. Este proceso consta de las siguientes fases:
 - Asignación de un número único a cada clase de equivalencia
 - Hasta que todas las clases de equivalencia válidas hayan sido cubiertas por (incorporadas a) casos de prueba, se tratará de escribir un caso que cubra tantas clases válidas no incorporadas como sea posible
 - Hasta que todas las clases de equivalencia no válidas hayan sido cubiertas por casos de prueba, escribir un caso para una única clase no válida sin cubrir
- Habría que diseñar casos de prueba que cubran todas las clases de equivalencia, tanto válidas como inválidas, y para las inválidas en casos de prueba distintos

Prueba de Análisis de Valores Límites

- La experiencia indica que los casos de prueba que exploran las condiciones límite de un programa producen un mejor resultado para detectar defectos
- El AVL es una técnica de diseño de casos de prueba que complementa a la de particiones de equivalencia
- Podemos definir las condiciones límites como las situaciones que se hayan directamente arriba, abajo y en los márgenes de las clases de equivalencia
- Las diferencias son las siguientes:
 - Más que elegir «cualquier» elemento como representativo de una clase de equivalencia, se requiere la selección de uno o más elementos tal que los márgenes se sometan a prueba
 - Más que concentrarse únicamente en el dominio de entrada (condiciones de entrada), los casos de prueba se generan considerando también el espacio de salida
- El proceso de selección es también heurístico, aunque existen ciertas reglas orientativas
- **Regla 1 (entrada).** Si una condición de entrada especifica un rango de valores ($[-1.0, +1.0]$) se deben generar casos para los extremos del rango ($-1.0, 1.0$) y casos no válidos para situaciones justo más allá de los extremos ($-1.001, 1.001$) por ejemplo
- **Regla 2 (entrada).** Si la condición de entrada especifica un número finito y consecutivo de valores (el fichero de entrada tendrá de 1 a 255 registros), hay que escribir casos para los números máximo, mínimo, uno más del máximo y uno menos del mínimo de valores indicados (0, 1, 255, 256)
- **Regla 3 (salida).** Usar la regla 1 para la condición de salida (el descuento máximo aplicable en compras al contado será del 50% y el mínimo será del 6%). Se escribirán casos para intentar obtener descuentos del 5.99%, 6%, 50%, 50.01%.
- **Regla 4 (salida).** Usar la regla 2 para cada condición de salida (el programa puede mostrar de 1 a 4 listados) se escribirán casos para intentar generar 0, 1, 4 y 5 listados
- En la Regla 4 al igual que en la Regla 3 debe considerarse que:
 - Los valores límite de entrada no generan necesariamente los valores límite de salida
 - No siempre se pueden generar resultados fuera del rango de salida (pero es interesante considerarlo)
 - Si la entrada o la salida de un programa es un conjunto ordenado (por ejemplo, una tabla, un archivo secuencial, etc.), los casos se deben concentrar en el primero y en el último elemento

- En las pruebas aleatorias simulamos la entrada habitual del programa creando datos de entrada en la secuencia y con la frecuencia con las que podrían aparecer en la práctica (de manera repetitiva)
- Para ello habitualmente se utilizan generadores automáticos de casos de prueba que se alimentan con una descripción de las entradas y las secuencias de entradas posibles, así como la probabilidad de ocurrir en la práctica
- Este tipo de pruebas es muy común en los compiladores, generando aleatoriamente códigos de programas que sirven de casos de prueba para la compilación
- Sin embargo, esta forma de generar casos de prueba es menos utilizada que las técnicas de caja blanca y caja negra

Estrategia de Aplicación de las Pruebas

- Pretende integrar el diseño de los casos de prueba en una serie de pasos bien coordinados a través de la creación de distintos niveles de prueba con diferentes objetivos
- También permite la coordinación del personal del desarrollo, del departamento de garantía de la calidad y del cliente, definiendo los papeles que deben desempeñar cada uno de ellos y la forma de llevarlos a cabo. Se seguirán las siguientes etapas:
 - Comenzar las pruebas a nivel de módulo
 - Continuar hacia la integración del sistema completo y a su instalación
 - Culminar con la aceptación del producto por parte del cliente
- Se comienza en la prueba de cada módulo, que normalmente la realiza el propio personal de desarrollo en su entorno
- Con el esquema del diseño del software, los módulos probados se integran para comprobar sus interfaces en el trabajo conjunto ([prueba de integración](#))
- El software totalmente ensamblado se prueba como un conjunto para comprobar si cumple o no tanto los requisitos funcionales como los requisitos de rendimientos, seguridad, etc. ([prueba funcional o de validación](#)).
- El software ya validado se integra con el resto del sistema (por ejemplo, elementos mecánicos, interfaces electrónicas, etc.) para probar su funcionamiento conjunto ([prueba del sistema](#))
- Por último, el producto final se pasa a la prueba de aceptación para que el usuario compruebe en su propio entorno de explotación si lo acepta como está o no ([prueba de aceptación](#))

La Prueba Unidad

- Se trata de las pruebas formales que permiten declarar que un módulo está listo y terminado (no las informales que se realizan mientras se desarrollan los módulos)
- Hablamos de una unidad de prueba para referirnos a uno o más módulos que cumplen las siguientes condiciones [IEEE, 1986a]:
 - Todos son del mismo programa
 - Al menos uno de ellos no ha sido probado
 - El conjunto de módulos es el objeto de un proceso de prueba
- La prueba de unidad puede abarcar desde un módulo hasta un grupo de módulos (incluso un programa completo)
- Estas pruebas suelen realizarlas el propio personal de desarrollo, pero evitando que sea el propio programador del módulo

La Prueba Integración

- Están totalmente ligadas a la forma prevista de integrar los distintos componentes del software hasta contar con el producto global que debe entregarse
- Implican una progresión ordenada de pruebas que parte desde los componentes (módulos) y culmina en el sistema completo
- Su objetivo es la prueba de las interfaces (flujos de datos) entre componentes o módulos
- El orden de integración elegido influye en una gran variedad de aspectos como:
 - La forma de preparar casos
 - Las herramientas necesarias
 - El orden de codificar y probar los módulos
 - El coste de la depuración
 - El coste de preparación de casos

Tipos fundamentales de integración:

- **Integración incremental.** Se combina el siguiente módulo que se debe probar con el conjunto de módulos que ya han sido probados. Se incrementa progresivamente el numero de módulos hasta formar el programa completo. En función del orden elegido dentro de la jerarquía modular o de llamadas, se distinguen dos tipos de integración:
 - Ascendente. Comienzan por los nodos hoja
 - Descendente. Se comienza por el módulo raíz
 - Y a su vez pueden ser en profundidad y en anchura
- **Integración no incremental.** Se prueba cada módulo por separado y luego se integran todos de una vez y se prueba el programa completo. Se denomina también Big-Bang por que el número de módulos crece instantáneamente en la construcción del programa
- Habitualmente las pruebas de unidad y de integración se solapan y mezclan en el tiempo

Proceso Incremental Ascendente:

- Se combinan los módulos de bajo nivel en grupos que realicen una función o subfunción específica (o también se puede trabajar con los módulos individualmente) de este modo reducimos el número de pasos de integración
 - Se escribe para cada grupo un módulo impulsor o conductor (driver), que es un módulo escrito "ex profeso" para permitir simular la llamada a los módulos, introducir los datos de prueba a través de los parámetros de llamada y recoger los resultados a través de los de salida
 - Se prueba cada grupo mediante su impulsor
 - Se eliminan los módulos impulsores y se sustituyen por los módulos de nivel superior en la jerarquía. Volviendo a construir nuevos impulsores hasta alcanzar en este paso la raíz de la jerarquía de módulos
-
- **Ventajas:**
 - Se detectan antes los fallos que se produzcan en la parte inferior del sistema
 - La definición de los casos de prueba es más sencilla, puesto que los módulos inferiores desempeñan funciones más específicas
 - Es más fácil observar los resultados de la prueba, ya que es en los módulos inferiores donde se elaboran los datos (los módulos superiores suelen ser coordinadores)
 - **Desventajas:**
 - Se requieren módulos impulsores que deben codificarse
 - El programa o sistema completo no se prueba hasta que se introduce el último módulo

Proceso Incremental Descendente:

- El módulo raíz o principal se prueba primero. Todos sus subordinados se sustituyen por módulos ficticios
- Una vez probado el módulo raíz (estando ya libre de defectos) se sustituye uno de sus subordinados ficticios por el módulo implementado correspondiente
- Cada vez que se incorpora un módulo se efectúan las pruebas correspondientes
- Al terminar cada prueba, se sustituye un módulo ficticio por su correspondiente real
- Conviene repetir algunos casos de prueba de ejecuciones anteriores para asegurarse de que no se ha introducido ningún error nuevo

Módulos Ficticios

- La creación de módulos ficticios subordinados es más complicada que la creación de impulsores
 - Módulos que sólo muestran un mensaje de traza
 - Módulos que muestran los parámetros que se les pasa
 - Módulos que devuelven un valor que no depende de los parámetros que se pasen como entrada
 - Módulos que, en función de los parámetros pasados, devuelven un valor de salida que más o menos se corresponda con dicha entrada
- Ventajas:
- Los defectos en los niveles superiores del sistema se detectan antes
 - Una vez incorporadas las funciones que manejan la entrada/salida, es fácil manejar los casos de prueba
 - Permite ver antes una estructura previa del programa, lo que facilita hacer demostraciones
- Desventajas:
- Se requieren módulos ficticios que suelen ser complejos de crear
 - Antes de incorporar la entrada/salida es complicado manejar los casos de prueba
 - A veces no se pueden crear casos de prueba, porque los detalles de operación vienen proporcionados por los módulos inferiores
 - Es más difícil observar la salida, porque los resultados surgen de los módulos inferiores
 - Pueden inducir a diferir la terminación de la prueba de ciertos módulos, ya que puede parecer que el programa funciona

Integración no Incremental

- Cada módulo que tiene que ser probado necesita lo siguiente:
 - Un módulo impulsor, que transmite o «impulsa» los datos de prueba al módulo y muestra los resultados de dichos casos de prueba
 - Uno o más módulos ficticios que simulan la función de cada módulo subordinado llamado por el módulo que se va a probar
- Una vez probado cada módulo por separado, se ensamblan todos de una vez y se prueban en conjunto

Comparación de los distintos tipos de integración

- Ventajas de la no incremental:
 - Requiere menos tiempo de máquina para las pruebas, ya que se prueba de una sola vez la combinación de los módulos
 - Existen más oportunidades de probar módulos en paralelo
- Ventajas de la incremental:
 - Requiere menos trabajo, ya que se escriben menos módulos impulsores y ficticios
 - Los defectos y errores en las interfaces se detectan antes, ya que se empieza antes a probar las uniones entre los módulos
 - La depuración es mucho más fácil, ya que si se detectan los síntomas de un defecto en un paso de la integración hay que atribuirlo muy probablemente al último módulo incorporado
 - Se examina con mayor detalle el programa, al ir comprobando cada interfaz poco a poco

La Prueba del Sistema

- Es el proceso de prueba de un sistema integrado de hardware y software para comprobar lo siguiente:
 - Cumplimiento de todos los requisitos funcionales, considerando el producto software final al completo en un entorno de sistema
 - El funcionamiento y rendimiento en las interfaces hardware, software, de usuario y de operador
 - Adecuación de la documentación de usuario
 - Ejecución y rendimiento en condiciones límite y de sobrecarga
- Fuente de diseño de casos de prueba del sistema
 - Casos basados en los requisitos gracias a técnicas de caja negra aplicadas a las especificaciones
 - Casos necesarios para probar el rendimiento del sistema y de su capacidad funcional (pruebas de volumen de datos, de límites de procesamiento, etc.). Este tipo de pruebas suelen llamarse pruebas de sobrecarga (*stress testing*)
 - Casos basados en el diseño de alto nivel aplicando técnicas de caja blanca a los flujos de datos de alto nivel (por ejemplo, de los diagramas de flujo de datos)

La Prueba de Aceptación

- Es la prueba planificada y organizada formalmente para determinar si se cumplen los requisitos de aceptación marcados por el cliente
- Sus características principales son las siguientes:
 - Participación del usuario
 - Está enfocada hacia la prueba de los requisitos de usuario especificados
 - Está considerada como la fase final del proceso para crear una confianza en que el producto es el apropiado para su uso en explotación
- Recomendaciones generales
 - Debe contarse con criterios de aceptación previamente aprobados por el usuario
 - No hay que olvidar validar también la documentación y los procedimientos de uso (por ejemplo, mediante una auditoría)
 - Las pruebas se deben realizar en el entorno en el que utilizará el sistema (lo que incluye al personal que lo maneja). Si se trata de un sistema contratado, la prueba se realiza bajo el control de la organización de desarrollo en el entorno de trabajo real ayudando al usuario (**pruebas Alfa**). En casos de productos de interés general se entregan versiones a usuarios de confianza, sin control directo, que informaran de su opinión sobre la aplicación (**pruebas Beta**)