

試験問題が含まれています!

ENSO の概要

今より多くの尾を持ちます!



タボアーダ特集 2016-2017

Índice

I El Software	7
1. Características	7
2. Atributos deseables en el software	8
2.1. Aplicaciones del software	8
3. Software heredado	9
4. Principales problemas asociados a la producción del software	9
II El proceso	10
1. Conceptos	10
2. Procesos para la construcción del software	10
2.1. Norma IEEE 1074 – 2006	11
2.2. Norma ISO 12207–1	12
2.2.1. Procesos principales	12
2.2.2. Procesos de soporte	13
2.2.3. Procesos de la organización	14
2.3. Norma ISO/IEC TR 15504–2 (2003)	14
2.4. Descripción de procesos en la Norma ISO 12207 (2008)	14
3. Evaluación del proceso software	15
3.1. Capability Maturity Model Integration (CMMI)	15
III Ciclos de vida	18
1. Ciclo de vida en cascada	18
1.1. Fases	19
1.2. Aportaciones	20
1.3. Problemas	20
2. Construcción de prototipos	20
2.1. Sistemas que resultan ser buenos candidatos	21
2.2. Beneficios de los prototipos	21
2.3. Formas de prototipos	21
2.4. Construcción de un prototipo	22
2.5. Problemas	22
3. Ciclo de vida incremental	22
3.1. Sistemas que resultan ser buenos candidatos	23
3.2. Ventajas	23
3.3. Desventajas	23
4. Técnicas de cuarta generación	23
4.1. Críticas más habituales	24

5. Modelo en espiral	24
5.1. Descripción del modelo	24
5.2. Ventajas	25
5.3. Desventajas	25
5.4. Análisis de riesgos	25
5.4.1. Definiciones	25
5.4.2. Anticipación a los riesgos	26
6. Metodologías ágiles	27
6.1. Variabilidad del entorno	28
6.2. Fragilidad por las debilidades humanas	28
6.3. Diferentes modelos de proceso	28
6.4. Modelado ágil	28
6.5. Programación extrema	29
IV Análisis de requisitos	31
1. Introducción	31
1.1. Requisito	31
1.1.1. Clasificación de los requisitos	31
1.2. Análisis de requisitos	31
1.3. Personal implicado	31
1.4. Razones por las que el análisis de requisitos es difícil	32
2. Obtención de requisitos	32
2.1. Técnicas de comunicación y recogida de información	32
2.2. TFEA	33
3. Análisis de requisitos	33
3.1. Análisis de requisitos del sistema	33
3.2. Análisis de requisitos del software	35
3.2.1. Principios del análisis de requisitos del software	35
4. Especificación	36
4.1. Especificación del sistema	36
4.2. Especificación del software	36
4.2.1. Principios de especificación	37
4.2.2. Características del documento de especificación	37
5. Verificación y Validación de requisitos	38
5.1. Estrategias	38
5.2. Técnicas	38
6. Administración de requisitos	38
6.1. Clasificación cualitativa	38
6.2. Clasificación cuantitativa	39
6.3. Etapas	39
V Análisis estructurado	40

1. Introducción	40
1.1. Problemas del análisis clásico	40
1.2. Soluciones al análisis clásico	40
2. Técnicas de especificación y modelado	40
2.1. Diagramas de flujo de datos (DFD)	41
2.1.1. Elementos	42
2.1.2. Niveles	42
2.2. Especificaciones de proceso (PSPEC)	44
2.3. Diagramas de flujo de control (DFC)	44
2.3.1. Construcción de un DFC	45
2.4. Especificaciones de control (CSPEC)	45
2.5. Diagramas entidad–relación (DER)	47
2.6. Comprobaciones a realizar sobre una especificación estructurada	47
2.7. Técnicas matriciales	47
3. Metodología del análisis estructurado	48
3.1. Fases	48
3.2. Modelos complementarios	48
VI Pruebas del software	49
1. Conceptos	49
2. Filosofía de las pruebas del software	50
2.1. Recomendaciones	50
3. Jerarquía de la documentación de diseño de pruebas	51
4. Técnicas de diseño de casos de prueba	52
4.1. Pruebas de caja negra (funcionales)	52
4.2. Pruebas de caja blanca (estructurales)	53
4.2.1. Criterios de cobertura lógica	53
4.2.2. Prueba de bucles	54
4.2.3. Utilización de la complejidad ciclomática de McCabe	54
5. Enfoque práctico recomendado para el diseño de casos	55
5.1. Depuración	55
5.2. Análisis de errores a análisis causal	56
6. Estrategia de aplicación de las pruebas	56
7. Pruebas en desarrollos orientados a objetos	57
7.1. Desde el punto de vista de diseño de casos de pruebas	57
7.2. Desde el punto de vista del nivel	57
7.3. Cuestiones a tener en cuenta	57
8. Ejemplo práctico de cálculo de la complejidad ciclomática de un método	58
VII Preguntas de examen	60

1. Resolución de preguntas cortas	60
2. Preguntas largas	68
2.1. Enumera y comenta las principales características del software asociadas a su naturaleza “inmaterial”	68
2.2. Enumera y comenta los principales problemas crónicos del software	68
2.3. Norma 12207-1: Procesos de soporte. ¿Cuál es su diferencia fundamental con la 1074? ¿Qué aporta la 15504-2 que no contempla la 12207 en los procesos de la organización?	68
2.4. Describe en qué consiste el ciclo de vida en espiral, cuales son sus ventajas e inconvenientes. Describe la metodología para el análisis de riesgos según Sommerville	68
2.5. Para cada uno de los siguientes diagramas, señala los elementos que contienen y describe su función en el diagrama	68
2.6. Comenta el proceso de elaboración de un plan de prueba seguido por el estándar IEEE 829 y los documentos asociados al proceso ¿Cuál es el estándar de ejecución de las pruebas? ¿Qué documentación implica?	69
2.6.1. Describe en qué consisten las pruebas funcionales, para qué sirven, en base a qué se construyen. Describe en detalle 3 técnicas de construcción de este tipo de pruebas.	69
3. Caso práctico 2014	70
3.1. Detalla formalmente las siguientes funcionalidades, según el modelo empleado en las prácticas	70
3.1.1. Requisitos	70
3.2. Diseña el Modelo de Contexto y el DFD de primer nivel de forma que englobe las funcionalidades del apartado anterior.	73
3.3. Para la funcionalidad RF1, haz un DFD de nivel 2, con su DFC asociado	74
3.4. En el siguiente contexto ¿cuál sería el modelo de ciclo de vida más adecuado? justifica tu respuesta.	74
3.5. Establece el proceso de admisión de riesgos, según la metodología de Sommerville, que contempla solo 2 riesgos, para el desarrollo de software en el siguiente contexto	75
3.6. Pensando en un proceso de pruebas, establece las clases de equivalencia para las siguientes entradas, asumiendo que en la pantalla táctil se puedan introducir los datos por pantalla con la simulación de un teclado qwerty	76
4. Caso práctico 2014	77
4.1. Detalla formalmente las siguientes funcionalidades, según el modelo empleado y la plantilla que empleasteis en las prácticas	77
4.2. Diseña el modelo de contexto, el DFD de primer nivel de forma que solo englobe las funcionalidades del apartado anterior	78
4.3. Para el RF3, haz el DFD de nivel 2, con su correspondiente DFC asociado	78
4.4. En el siguiente contexto, ¿cuál sería el modelo de ciclo de vida más apropiado? Motiva tu respuesta:	78
4.5. Establece un proceso de administración de riesgos, siguiendo la metodología de Sommerville, que contempla solo 3 riesgos, para el desarrollo del software en el siguiente contexto:	78
4.5.1. Identificación y análisis de riesgos	78
4.5.2. Planificación de riesgos	79

- 4.6. Pensando en el proceso de pruebas, establece las clases de equivalencia para las siguientes entradas: 79

Tema I

El Software

Software Pressman define el software como el conjunto de **código, estructuras de datos y documentación** asociados a un sistema computacional. Es habitualmente, el componente de un sistema que presenta mayores problemas durante el desarrollo.

Ingeniería del Software Es aquella disciplina que se encarga de establecer un orden en el desarrollo de sistemas de software.

Crisis del software Durante los primeros años del desarrollo del software, al no utilizarse metodología ninguna, los programas contenían numerosos errores e inconsistencias que obligaban a continuas modificaciones. Al final, se hacía más rápido y, por lo tanto barato, empezar de cero a realizar un software, que acabaría presentando los mismos problemas.

1. Características

Principalmente, el software se diferencia al presentar una **naturaleza lógica** (es inmaterial); por ello, se dice que el **software se desarrolla, no se fabrica** en sentido estricto. A pesar de muchas similitudes (fases de análisis, diseño, etc.), el software se diferencia de los productos de otras ingenierías por lo siguiente:

1. **Costes de replicación negligibles:** En el caso del software, la mayor parte de la inversión se concentra en las fases de ingeniería previas a la producción, dado que, al ser un producto inmaterial, la replicación del producto no presenta problemas técnicos, no requiere un control individualizado, y el coste unitario resulta prácticamente nulo.

	Ingeniería	Producción o Desarrollo	Coste unitario / 100 unidades	Coste unitario / 100.000 unidades
Hardware	1000	50 c.u.	60	50.01
Software	1000	2000	30	0.03

Figura 1: Influencia de los costes de ingeniería en el coste total del producto.

2. **Curva de fallos con respecto al tiempo diferente:** El software no se estropea con el tiempo; sin embargo es común aplicar ciertos cambios al mismo durante su ciclo de vida debidos al mantenimiento. Por ello, es probable que se introduzcan nuevos errores, que se acumulan con el tiempo, degradando la calidad. **El software no se estropea, pero se deteriora.**
3. **Baja reutilización de las partes:** Por lo general el software se construye a medida como un conjunto, provocando que la reutilización sea baja; consecuentemente, se perjudica el desarrollo del software. Existe una tendencia al alza en la reutilización de artefactos (no tienen por qué ser código) gracias a:
 - La elaboración de librerías y frameworks.
 - Aplicación de técnicas de programación estructurada, modular y orientada a objetos.
 - Aplicación de patrones de diseño, que supone el poder reutilizar diseños, el lugar de solo código.

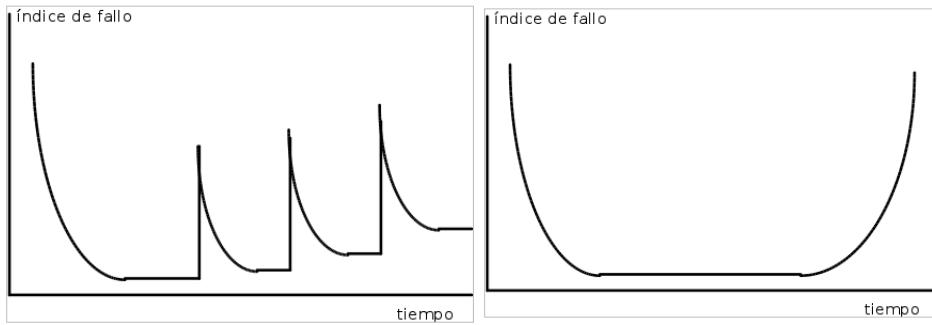


Figura 2: Curva de fallos del Software (Izquierda) y del Hardware (Derecha). Obsérvese como en el caso del software el índice de fallos se va acumulando. Idealmente, el software solo presentaría fallos al principio de su vida por errores no detectados durante el desarrollo.

Una empresa debería identificar los artefactos reutilizables con los que cuente, para que sus empleados los utilicen.

2. Atributos deseables en el software

Todo software debería contar con las siguientes características:

- **Mantenibilidad:** Facilidad para realizar cambios.
- **Confiabilidad:** Capacidad para seguir funcionando de manera segura y correcta.
- **Eficiencia:** Utilización de la mínima cantidad necesaria de recursos.
- **Usabilidad:** Facilidad con la que las personas utilizan el software.

2.1. Aplicaciones del software

Es difícil clasificarlas y carece de mucho valor hacerlo. *Pressman* lo hace de la siguiente forma:

- **Software de sistemas:** Sirven a otros programas.
- **Software de aplicación:** Programas independientes que resuelven una necesidad específica.
- **Software científico y de ingeniería:** “Devoradores de números”.
- **Software empotrado:** Reside en una memoria de sólo lectura de pequeños chips, y con funciones limitadas y muy específicas.
- **Software de línea de productos:** *Middleware*; cuentan con una capacidad específica que muchos clientes pueden usar (procesador de textos, por ejemplo).
- **Aplicaciones web.**
- **Inteligencia artificial:** *Redes neuronales* (máquinas de estados **NO**).

El objetivo de todo software es desempeñar una determinada función cumpliendo una serie de requisitos.

3. Software heredado

Se trata de software **desarrollado hace décadas** que además **ha ido sufriendo cambios** a lo largo del tiempo para adaptarse a los requisitos cambiantes del negocio. Estos dos factores hacen que sea muy difícil, y por lo tanto caro, de tratar, y usualmente es crítico para los negocios (COBOL). Como es de esperar, probablemente se sustente sobre procesos de desarrollo desfasados, y malas prácticas (falta de documentación, por ejemplo). *Pressman* aconseja tocarlo lo mínimo posible, al menos mientras no sea necesario.

4. Principales problemas asociados a la producción del software

Muchos expertos argumentan que la crisis del software nunca se ha solucionado, y es que esta ingeniería arrastra desde hace tiempo los siguientes **problemas crónicos**, causados por las propias características del software y por los errores de quiénes intervienen en su producción:

1. **Imprecisión en la planificación, y estimación de costes temporales y monetarios:** Es frecuente que surjan imprevistos al abordar proyectos de cierta complejidad, dando lugar a desviaciones. Además, sin una planificación detallada, es imposible hacer una estimación de costes e identificar las tareas conflictivas. Entre las causas podemos citar la falta de recogida de datos de proyectos anteriores (no acumular experiencia), y la tradicional falta de experiencia de los gestores de proyecto (gente que no tiene ni idea de software gestionando proyectos o viceversa).
2. **Baja productividad:** Los proyectos de software tienen a una duración mayor a lo esperada y, por lo tanto, mayores costes, y menor productividad y beneficios. Entre los múltiples motivos, se encuentran las especificaciones ambiguas o incorrectas, la poca comunicación con el cliente hasta la entrega, con sus consecuentes cambios de última hora, y la falta de documentación. Este problema culmina con que sea más costoso realizar una modificación sobre un programa que el rehacerlo.
3. **Mala calidad:** El que las especificaciones sean ambiguas o incorrectas, junto con falta de realización de pruebas exhaustivas, propicia la entrega de software con muchos errores, y por lo tanto incrementa los costes durante el mantenimiento.
4. **Insatisfacción del cliente:** Debida a los problemas anteriores, los clientes suelen quedar poco satisfechos con los resultados.

Tema II

El proceso

1. Conceptos

Ingeniería del Software Aplicación de un enfoque sistemático, disciplinado y cuantificable hacia el desarrollo, operación y mantenimiento del software; es decir, la aplicación de la ingeniería al software. (IEEE). Es necesaria para identificar y conocer las causas de los problemas en el desarrollo del software, combinando las metodologías necesarias, dado que no van a desaparecer de la noche a la mañana. Está implícita la existencia de fases, de modo que la ingeniería del software proporciona una metodología que indica en cada momento los pasos a seguir, así como permite identificar en qué parte del proceso de ingeniería estamos y cuán bien lo estamos haciendo.

Tarea Cualquier acción que transforma una entrada en salidas, el objetivo debe de ser pequeño y bien definido. *Ej: Ejecución del caso de prueba.*

Actividad Conjunto de tareas que producen un producto importante del trabajo. *Ejemplo: desarrollo de un plan de pruebas.*

Proceso Conjunto de actividades, y tareas que se ejecutan para llevar a cabo algún producto de trabajo. Este conjunto es **adaptable** a las necesidades del que lo utiliza. *Ejemplo: Validación.*

Modelo de los procesos Descripción de los procesos involucrados en el desarrollo del software sin especificar cuando se desarrolla: *Ejemplos: IEEE 1074, ISO 12207-1 e ISO/IEC TR 15504-2.*

Métodos y procedimientos Determinan el modo en el que se *ejecutan las tareas*, determinando qué técnicas se utilizan en cada fase y cómo. *Ejemplo: IEEE 1008.*

Técnica Cualquier recurso utilizado para llevar a cabo una tarea. Normalmente hablamos de gráficos con apoyos textuales. *Ej: Cobertura de caminos.*

Herramienta Cualquier software que nos ayude en cualquier etapa del desarrollo. *Ejemplo: JUnit.*

2. Procesos para la construcción del software

La construcción del software incluye una serie de actividades que se empiezan a estandarizar. Pressman divide la construcción de software en **3 fases**:

1. **Fase de definición:** Se centra en el **qué**, intentando identificar la información a procesar, la función y rendimiento esperados, las restricciones de diseño, las interfaces a utilizarse, los sistemas operativos y de hardware a utilizar, y los criterios de validación. Se identifican 3 actividades:
 - **Análisis del sistema:** Define el papel de cada elemento relacionado con el sistema informático a desarrollar.

- **Análisis de requisitos del software:** Proporciona el ámbito del software y su relación con el resto de componentes del sistema.
 - **Planificación:** Organización de las tareas que se llevarán a cabo en el proyecto. Es análisis y definición de requisitos es una tarea que debe ser llevada a cabo conjuntamente por el desarrollador de software y el cliente. Esta etapa produce el documento de especificación de requisitos del software.
2. **Desarrollo:** Se intenta definir **cómo** han de diseñarse las estructuras de datos, cómo ha de implementarse la función dentro de una arquitectura software, cómo han de implementarse los detalles procedimentales, cómo han de caracterizarse las interfaces, cómo ha de traducirse el diseño en un lenguaje de programación y cómo han de realizarse las pruebas. Se definen 3 actividades:
- **Diseño del software.**
 - **Codificación.**
 - **Pruebas.**
3. **Mantenimiento:** Se centra en el cambio asociado a la corrección de errores, a las adaptaciones requeridas a medida que evoluciona el entorno del software, y a los cambios debidos a las mejoras producidas por los requisitos cambiantes del cliente. Se definen 4 actividades:
- **Adaptación** (cambio del entorno).
 - **Corrección** (errores).
 - **Mejora** (cambios en los requisitos).
 - **Prevención** (mejor ingeniería).

Nota: *Estas actividades se aplican de forma iterativa según la última versión del Pressman. Por ejemplo un proceso de mantenimiento como puede ser la corrección de un error grande encajaría como una iteración de este modelo y no como una actividad concreta.*

2.1. Norma IEEE 1074 – 2006

Proporciona el conjunto de actividades que constituyen los procesos que son necesarios para el correcto desarrollo y mantenimiento del software. Los procesos se dividen en 4 secciones lógicas:

1. **Modelo del Ciclo de Vida Software.** Actividades para seleccionar el modelo de ciclo que **mejor se adapte** al proyecto; es necesario escoger uno. Además, la versión de 2006 requiere que se evalúe el riesgo.
2. **Grupos de Actividades de Gestión del Proyecto.** Son los procesos que inician, supervisan y controlan los proyectos de software a lo largo de su ciclo de vida.
 - **Iniciación del proyecto:** Se crea el ciclo de vida y se definen las métricas para la gestión del proyecto.
 - **Planificación y control del proyecto:** Se desarrollan planes para la evaluación, gestión de la configuración, instalación, integración, etc.
 - **Monitorización y control:** Actividades de gestión de riesgos, gestión de proyecto, mejoras del ciclo de vida, recolección y análisis de métricas, y cierre del proyecto.

3. **Grupos de Actividades Orientadas al desarrollo.** Comprenden los procesos que se realizan antes, durante y después del desarrollo software.

- Pre-desarrollo:
 - Análisis de la necesidad del sistema.
 - Asignación de requisitos del sistema al software y al hardware (solo si se trabajará con ambos).
- Desarrollo:
 - Análisis de requisitos.
 - Diseño de arquitectura, BBDD, interfaces...
 - Implementación (codificación, documentación, integración, gestión de versiones...).
- Post-desarrollo:
 - Instalación.
 - Operación y Soporte.
 - Mantenimiento.
 - Retirada.

4. **Grupos de Actividades de Soporte:** Necesarios para asegurar terminación y **calidad de los procesos** y, por lo tanto, del proyecto.

- Verificación y Validación (aseguramiento de la calidad).
- Gestión de Configuración Software.
- Desarrollo de documentación.
- Formación.

2.2. Norma ISO 12207-1

Define una serie de actividades que se realizan en la construcción del software, agrupadas en **cinco procesos principales, ocho de soporte, y cuatro de la organización**. No fomenta ningún modelo concreto de ciclo de vida, gestión del software o método de ingeniería, ni prescribe cómo realizar las actividades o cómo organizarlas.

2.2.1. Procesos principales

Los procesos principales resultan útiles a las personas que inician o realizan el desarrollo, la explotación o el mantenimiento del software durante su ciclo de vida. Consisten en:

1. **Proceso de adquisición:** Contiene las actividades realizadas por el cliente para adquirir un software. *Ej: Solicitud de ofertas, o selección del suministrador.*
2. **Proceso de suministro:** Contiene las actividades que realiza el suministrador. Incluye la preparación de la oferta para responder a una solicitud, y la identificación de los recursos y procedimientos para garantizar el éxito del proyecto.
3. **Proceso de desarrollo:** Contiene numerosas actividades:
 - Análisis de requisitos del sistema: funcionales y no funcionales.
 - Diseño de la arquitectura del sistema: principales componentes de hardware y software.

- Análisis de los requisitos del software: se definen y documentan (pruebas de aceptación).
 - Diseño de la arquitectura del software: se transforman los requisitos en una estructura de alto nivel en la que se pueden identificar los componentes principales. Se elabora una versión previa de los manuales, se define qué deben cumplir las pruebas de estos componentes, y se planifica la integración.
 - Diseño detallado del software: se diseña de forma detallada cada componente de software. Se actualizan los manuales, se define qué deben cumplir las pruebas de estos componentes, y se planifican las pruebas unitarias.
 - Codificación y prueba: se desarrollan y documentan los componentes software, y se prueba cada uno. Se actualizan los manuales.
 - Integración del software: se integran los componentes del software y se prueban según sea necesario. Se actualizan los manuales.
 - Prueba del software: en función de los requisitos especificados para él.
 - Integración del sistema: integración de elementos software y hardware.
 - Prueba del sistema: de acuerdo con los requisitos de cualificación especificados para el sistema.
 - Instalación: en el entorno de explotación final.
 - Soporte del proceso de aceptación: apoyo a la revisión de aceptación y prueba por parte del comprador.
4. **Proceso de explotación:** Contiene la explotación del software y el soporte operativo a los usuarios.
5. **Proceso de mantenimiento:** Modificación del código o documentación del software, como consecuencia de errores o deficiencias, mejoras, adaptaciones, y migraciones o retiradas del sistema.

2.2.2. Procesos de soporte

Sirven de apoyo al resto y se aplican en cualquier punto del ciclo de vida.

1. **Proceso de documentación:** Registra la información producida por cualquier proceso o actividad del ciclo de vida.
2. **Proceso de gestión de la configuración:** Procesos administrativos para contar una línea base de los elementos configurables del software, hacer el control de cambio sobre estos elementos, registrar su estado y peticiones de cambio, asegurar su consistencia y corrección si es necesaria, y controlar su almacenamiento y distribución.
3. **Proceso de aseguramiento de la calidad:** Asegurar que los procesos y productos cumplen con los requisitos especificados y se ajustan a los planes establecidos.
4. **Proceso de verificación:** Determina si los requisitos de un sistema o del software están completos y son correctos, y si los **productos software de cada fase** del ciclo de vida cumplen los requisitos o condiciones impuestos en fases previas.
5. **Proceso de validación:** Sirve para determinar si el **sistema o software final** cumple con los requisitos previstos para su uso.
6. **Proceso de revisión conjunta:** Evaluación del estado del software y sus productos en su conjunto.

7. **Proceso de auditoría:** Permite determinar, mediante hitos predeterminados, si se han cumplido los requisitos, los planes y el contrato.
8. **Proceso de resolución de problemas:** Analiza y elimina los problemas descubiertos durante el desarrollo, la explotación, el mantenimiento u otro proceso.

2.2.3. Procesos de la organización

Ayudan a conseguir que la organización sea más efectiva. Se llevan a cabo fuera del ámbito de proyectos y contratos específicos.

1. **Proceso de gestión:** Planificación, seguimiento y control, evaluación ... (actividades genéricas para la gestión de procesos).
2. **Proceso de infraestructura:** Dota a los demás procesos de la infraestructura necesaria, incluyendo hardware, software, normas, herramientas...
3. **Proceso de mejora:** Referente a los procesos del ciclo de vida.
4. **Proceso de formación:** Mantener al personal formado.

2.3. Norma ISO/IEC TR 15504-2 (2003)

Se puede considerar una ampliación a la norma ISO 12207 – 1 ya que amplía o añade procesos. Además de ello, también expresa la capacidad que una empresa ha logrado en el desarrollo de un proceso. Los procesos que se añaden son los siguientes:

- Procesos Principales: **Obtención de requisitos** (necesidades y requisitos del cliente).
- Procesos de Organización: **Gestión de recursos humanos** (proporcionar a la organización y proyectos los individuos con las habilidades y conocimiento efectivos), **alineamiento de la organización** (visión, cultura y compresión de los objetivos común), **medida** (recogida y análisis de datos relativos a los productos desarrollados) y **reutilización**. **El proceso de gestión de los procesos de la organización se divide** ahora en:
 - Gestión general: Iniciación y realización de cualquier proceso.
 - Gestión del **proyecto**: Asegurar que un proyecto produzca el resultado esperado, cumpliendo con los requisitos.
 - Gestión de **calidad**: Con el objetivo de satisfacer al cliente.
 - Gestión de **riesgos**: Identificar y reducir continuamente riesgos en un proyecto a lo largo de su ciclo de vida.

2.4. Descripción de procesos en la Norma ISO 12207 (2008)

1. **Identificador:** Categoría y número secuencial.
2. **Título.**
3. **Propósito.**
4. **Resultados observables.**
5. **Actividades y tareas.**

3. Evaluación del proceso software

Como respuesta a los problemas generados por desarrollar software sin seguir un proceso o una metodología orientados a la calidad, las empresas se interesaron en encontrar y seguir procesos que les garantizasen los resultados. Sin embargo, en muchos casos se aplicaban de forma incompleta e inconsistente. De ello, surgen una serie de métodos de evaluación y mejora de los procesos de la ingeniería del software.

Objetivos:

- De cara al contratante de la empresa de software proporcionar una medida de **cuán confiable** es que la empresa ofrezca sus servicios de creación y mantenimiento de software en tiempo y forma.
- De cara a la desarrolladora pretenden guiarla en la **mejora continua de sus procesos** de ingeniería.



Figura 3: Esquema de la evaluación del proceso software.

3.1. Capability Maturity Model Integration (CMMI)

Es un modelo que centra su evaluación en un total de 22 **áreas del proceso** categorizadas en:

- Gestión de procesos.
- Gestión de proyectos.
- Ingeniería.
- Soporte.

Existen dos versiones del modelo:

Modelo continuo Define el nivel de capacidad de cada una de las áreas del proceso (es decir, de manera independiente). Los niveles de capacidad son los siguientes:

0. **Incompleto:** El área del proceso aún no se realiza, o todavía no alcanza todas las metas.
1. **Realizado:** Todas las metas del área (específicas) han sido satisfechas.

2. **Gestionado:** Además, el trabajo del área se ajusta a una política organizacional definida (control, revisión y evaluación), y se implica al cliente cuando sea necesario.
3. **Definido:** Además, el proceso contribuye al proceso organizacional.
4. **Cuantitativamente gestionado:** Además, el área se controla y mejora mediante mediciones cuantitativas.
5. **En optimización:** Ademas, el área se adapta y mejora mediante medios cuantitativas a las necesidades cambiantes del cliente.

Cada área del proceso consta de un conjunto de **metas específicas** a alcanzar en ella, así como las **prácticas específicas** requeridas para alcanzar dichas metas (las prácticas convierten una meta en un conjunto de actividades); mediante estas metas, se pretende establecer las características que deben existir para que las actividades del área del proceso sean efectivas. Además de las metas y prácticas específicas, CMMI define **metas genéricas y prácticas genéricas** que se aplican a todas las áreas.

Esta versión del modelo cuenta con cinco metas y prácticas genéricas, cada una correspondiente a un nivel de capacidad; por ello, para alcanzar una nivel de capacidad particular, se debe alcanzar la meta genérica para dicho nivel:

1. Alcanzar las metas específicas.
2. Institucionalizar un proceso de gestión.
3. Institucionalizar un proceso definido.
4. Institucionalizar un proceso manejado en forma cuantitativa.
5. Institucionalizar un proceso de mejora.

Modelo discreto Define el nivel de madurez global de los proyectos y de la organización. Aquí, surge la principal diferencia entre las dos versiones, dado que el modelo discreto establece cinco niveles de madurez, en vez de cinco niveles de capacidad:

1. Nivel por defecto.
2. Se deben cumplir las metas generales 2 para cada para siete áreas del proceso.
3. Se deben cumplir las metas generales 2 y 3 para 18 áreas del proceso.
4. Solo añade 2 áreas del proceso, que deben cumplir las metas generales 2 y 3.
5. Solo añade 2 áreas del proceso, que deben cumplir las metas generales 2 y 3.

A diferencia del modelo continuo, del modelo discreto solo define las metas generales 2 y 3, yal y como se habían descrito.

Standard CMMI Appraisal Method for Process Improvement (SCAMPI) Método oficial mediante el cual realizar análisis de calidad en CMMI, con el objetivo de identificar los puntos fuertes y débiles de procesos, revelar riesgos, y determinar los niveles de capacidad y madurez.

Nota: *la diferencia entre las dos versiones es meramente organizacional, y los contenidos equivalentes. En el modelo continuo, CMMI recomienda un orden para la mejora de los procesos a nivel de cada área, dando la libertad de escoger el orden que mejor se ajuste a la organización. En el modelo discreto, cada área tiene un único objetivo específico, y el alcanzarlo supone haber mejorado todas las actividades asociadas a él.*

Tema III

Ciclos de vida

Ciclo de vida Sucesión de etapas por las que pasa el software desde que se concibe hasta que se deja de utilizar.

Etapa del ciclo Lleva asociada una serie de tareas y de documentos (*en sentido amplio: software*) de **salida** que serán la **entrada** de la fase siguiente.

Elección de un ciclo de vida Se realiza de acuerdo con la naturaleza del proyecto, los métodos a usar, y los controles y entregas requeridos.

1. Ciclo de vida en cascada

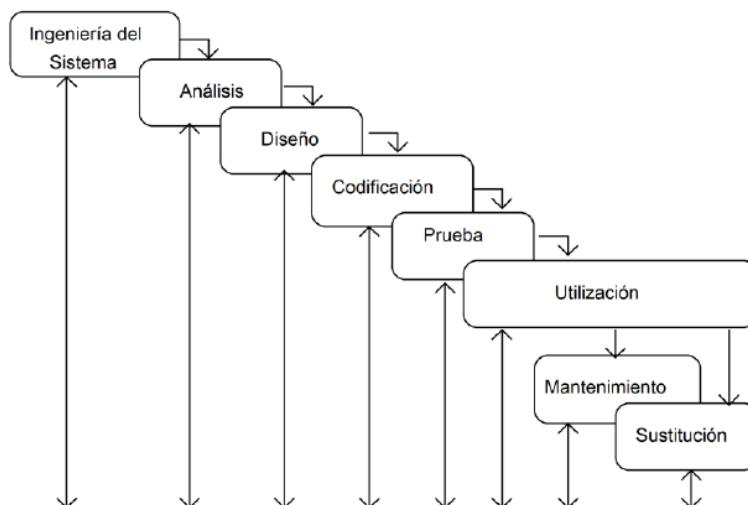


Figura 4: Etapas del ciclo de vida en cascada.

El ciclo de vida en cascada es el más antiguo, surgido directamente de la copia del ciclo convencional de una ingeniería. Exige un enfoque sistemático y **secuencial** del desarrollo de software. Se dice que el modelo en cascada está guiado por documentos, ya que **nunca empieza la siguiente fase antes de que presente el documento de la anterior** (Tabla 1).

Proceso	Documentos Producidos
Especificaciones del sistema.	Especificación funcional. Arquitectura del sistema
Análisis de requisitos	Documento de requisitos
Diseño de la arquitectura del software	Especificación de la arquitectura
Diseño de interfaces	Especificación del diseño
Codificación	Código de programa
Prueba de unidades	Informe de pruebas de unidad
Prueba de módulos	Informe de pruebas de módulo
Prueba de integración	Informe de prueba de integración y manual de usuario final
Prueba del sistema	Informe de prueba del sistema
Prueba de aceptación	Sistema final más la documentación

Tabla 1: Ejemplo de documentos producidos en un ciclo de vida en cascada. *No tiene que tener exactamente las mismas fases que el ciclo en cascada por defecto.*

1.1. Fases

1. **Ingeniería y análisis del sistema:** Define las interrelaciones del software con otros elementos del sistema más complejo en el que esté englobado; es decir, se asignan funciones del sistema al software. Por tanto, comprende los requisitos globales a nivel de sistema, mediante un análisis y diseño a alto nivel.
2. **Análisis de requisitos del software:** Análisis detallado de los componentes del sistema implementados mediante software incluyendo **datos** a manejar, **funciones** a desarrollar, **interfaces**, y **rendimiento** esperado.
Nota: *Los requisitos, tanto del sistema como del software, deben documentarse y revisarse con el cliente.*
3. **Diseño:** En cuanto a estructura de los datos, arquitectura de las aplicaciones, estructura interna de los programas, y las interfaces. Es un proceso que traduce los requisitos en una representación del software que permita conocer la arquitectura, funcionalidad e incluso la calidad antes de codificar.
4. **Codificación:** Traducción del diseño a un formato que sea legible para la máquina.
Nota: *Como se puede observar, estas primeras fases del ciclo de vida consisten básicamente en una traducción de documentos.*
5. **Prueba:** Verificar que no se hayan producido errores en alguna de las fases de traducción anteriores. Deben probarse todas las sentencias de todos los módulos, y no solo los casos “normales”.
6. **Utilización:** Entrega del software al cliente y comienzo de su vida útil. Es una etapa que se solapa con las posteriores.
7. **Mantenimiento:** El software sufrirá cambios a lo largo de su vida útil debido a que el cliente detecte errores, que se produzcan cambios en alguno de los componentes del sistema, o que el cliente requiera modificaciones funcionales. Estos cambios requieren volver atrás en el ciclo de vida (etapas de codificación, diseño o incluso análisis), en función de la magnitud del cambio.

Nota: *El modelo en cascada, a pesar de ser lineal, contiene flujos que permiten la vuelta atrás. Además del mantenimiento, se puede volver desde cualquier fase a la anterior si se detectan fallos. Estas vueltas atrás no son controladas y quedan reflejadas en el modelo.*

8. **Sustitución:** Cualquier aplicación acaba siendo sustituida. Es una tarea que debe planificarse cuidadosamente y, si es posible por fases:

- a) Trasvase de la información que maneja el sistema viejo al formato del nuevo.
- b) Mantención de los dos sistemas funcionando en paralelo para comprobar que el nuevo funciona correctamente.
- c) Sustitución completa del sistema antiguo.

1.2. Aportaciones

- Es el más simple, conocido y fácil de usar.
- Los procesos definidos se **formalizan** en normas, *sabes lo que tienes que hacer en cada momento*.
- Permite generar software eficientemente, y de acuerdo a las especificaciones: no sobrepasar fechas de entrega ni costes esperados.
- Al final de cada fase, los interesados pueden comprobar el estado del proyecto.

1.3. Problemas

- En realidad los **proyectos no siguen un ciclo de vida estrictamente secuencial**, sino que siempre hay iteraciones. Un claro ejemplo de ello es la fase de mantenimiento, aunque es frecuente detectar errores en cualquiera de las otras fases.
- No siempre se pueden establecer los requisitos desde el primer momento, sino lo normal es que se vayan aclarando y refinando a lo largo de todo el proyecto. Es habitual que los clientes no tengan el conocimiento de la importancia de la fase de análisis, o bien no hayan pensado en detalle qué quieren del software.
- Hasta que se llega a la fase final, no se dispone de una versión operativa del programa. Esto no resulta óptimo, dado que la mayor parte de los fallos suelen detectarse una vez el cliente puede probar el programa; al final del proyecto, es cuando más costosos son de corregir, y cuando más prisas hay.
- Se producen estados de bloqueo, dado que una fase no puede comenzar sin los documentos de salida de la anterior.

Nota: *A pesar de ello, es mucho mejor desarrollar software siguiendo el modelo de ciclo de vida en cascada, que hacerlo sin ningún tipo de guías.*

2. Construcción de prototipos

Este ciclo de vida paliaba directamente las deficiencias del ciclo de vida en cascada de que:

- Sea difícil tener claros todos los requisitos al inicio del proyecto.
- No se disponga de una versión operativa del programa hasta las fases finales.

2.1. Sistemas que resultan ser buenos candidatos

- Resulta idóneo en **proyectos con un nivel alto de incertidumbre**, donde los requisitos no están nada claros en primera instancia, ya que se provee un **método para obtener los requisitos de forma fiable** a través del feedback del usuario. Por lo tanto, debería omitirse ante problemas bien comprendidos.
- También es de especial interés en aplicaciones que presenten mucha interacción con el usuario, o algoritmos evolutivos.
- La aplicación no debe contar con una gran complejidad, o las ventajas de la construcción de prototipos se verán superadas por el esfuerzo de un prototipo que habrá que desechar o modificar mucho.
- El cliente debe estar dispuesto a probar un prototipo.

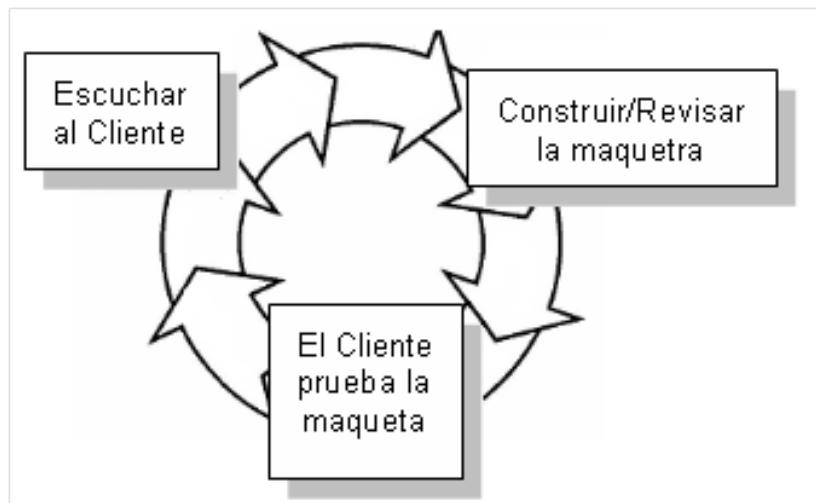


Figura 5: Etapas del paradigma de la construcción de prototipos.

2.2. Beneficios de los prototipos

- Permiten probar la eficiencia en condiciones similares a las que existirán durante la Utilización del sistema.
- Permiten mostrar al cliente la E/S de datos en la aplicación, para detectar problemas en la especificación.
- Gran parte del trabajo realizado puede ser utilizado en el diseño final. El diseño se parecerá cada vez más al del producto final, mientras que los componentes codificados deberán ser generalmente optimizados.
- Los prototipos permiten analizar **alternativas y viabilidades**, refinando el resultado final o abortándolo antes de invertir demasiado dinero en el mismo.

2.3. Formas de prototipos

- En papel o ejecutable, que describa la interacción con el usuario.
- Que implemente subconjuntos de la funcionalidad requerida, para evaluar el rendimiento.

- Que realice todo o en parte, pero con características que todavía deban ser mejoradas.

2.4. Construcción de un prototipo

1. Realizar un modelo del sistema a partir de los requisitos conocidos. No es necesario realizar una definición completa de estos.
2. Diseñar rápidamente el prototipo, centrándose en la arquitectura del sistema y en las interfaces, antes que en aspectos procedimentales.
3. Construir el prototipo, mediante una codificación rápida que facilite el desarrollo incremental. Es irrelevante la calidad obtenida.
4. Presentar el prototipo al cliente para que lo pruebe y sugiera modificaciones.
5. Con estos comentarios, proceder a construir un nuevo prototipo, y así sucesivamente, hasta que los requisitos queden completamente formalizados y se pueda comenzar a desarrollar el producto final.

Nota: *Como se puede comprobar, el prototipado es una técnica que sirve fundamentalmente para la fase de análisis de requisitos.*

Nota: *El prototipo no es el sistema final.*

2.5. Problemas

- En muchas ocasiones, el prototipo, que carece de calidad, pasa a ser parte del sistema final por presiones, o bien por que los técnicos se hayan acostumbrado a él.
- **Es posible que nunca se llegue a la fase de construcción** por falta de recursos, lo cual da lugar a los problemas característicos de un software construido sin seguir procesos de ingeniería al emplear el prototipo como sistema final.
- Es imposible una **predicción de costes** fiable.

3. Ciclo de vida incremental

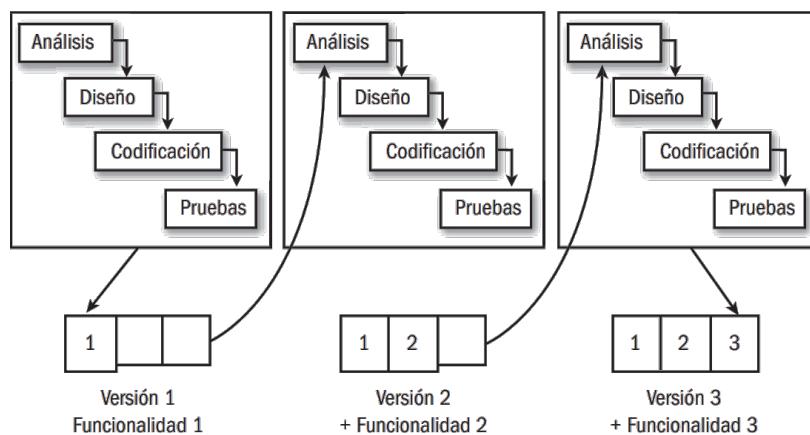


Figura 6: Etapas del ciclo de vida incremental.

Combina elementos del **modelo lineal** en cascada con la **filosofía iterativa** de construcción de prototipos. Para ello, se va creando el sistema software mediante incrementos, aportando nuevas funcionalidades o requisitos. De este modo el software ya no se ve como un producto con una fecha de entrega, sino como una **integración de sucesivos refinamientos**.

A diferencia del modelo por prototipos, el incremental se centra en obtener un producto operativo en cada iteración; los primeros incrementos simplemente son versiones incompletas del producto final.

Incrementos Componentes funcionales del sistema. Cada incremento es una parte completa y funcional del sistema por sí misma.

3.1. Sistemas que resultan ser buenos candidatos

- Entornos con incertidumbre.

Nota: *Nótese que se adapta bien a entornos de incertidumbre, mientras que el modelo basado en prototipos es más propio de entornos de alta incertidumbre.*

3.2. Ventajas

- Soluciona parte de los problemas del modelo en cascada, al mejorar la **comunicación con el cliente**, y al existir **detecciones de errores** con más atelación.
- Favorece la modulación del software al ser un modelo iterativo.

3.3. Desventajas

- Dado que la parte de requisitos no se encuentra en la fase iterativa, es difícil ver si estos son válidos, y se siguen detectando relativamente tarde, al haberse construido ya productos operativos en cada iteración (mismos problemas que con el ciclo en cascada).

4. Técnicas de cuarta generación

Son un conjunto diverso de métodos y herramientas con el objetivo de facilitar el desarrollo de software, al permitir especificar características del software a alto nivel, y generar código fuente automáticamente a partir de estas especificaciones:

- Acceso a bases de datos utilizando lenguajes de consulta de alto nivel, sin ser necesario conocer la estructura de esta.
- Generación de código.
- Generación de pantallas y entornos gráficos.
- Generación de informes.

Con esta automatización, se reduce la duración de las fases del ciclo de vida clásico, especialmente de la codificación.

A pesar de ello, estas técnicas no han obtenido los resultados previstos cuando se comenzaron a desarrollar, con intenciones de eliminar la necesidad de la codificación manual y de incluso analistas. Por lo menos, consiguen eliminar las tareas más repetitivas y tediosas.

4.1. Críticas más habituales

- No son más fáciles de utilizar que los lenguajes de tercera generación.
- El código fuente que producen es ineficiente.
- Solo son generalmente aplicables al software de gestión.

5. Modelo en espiral

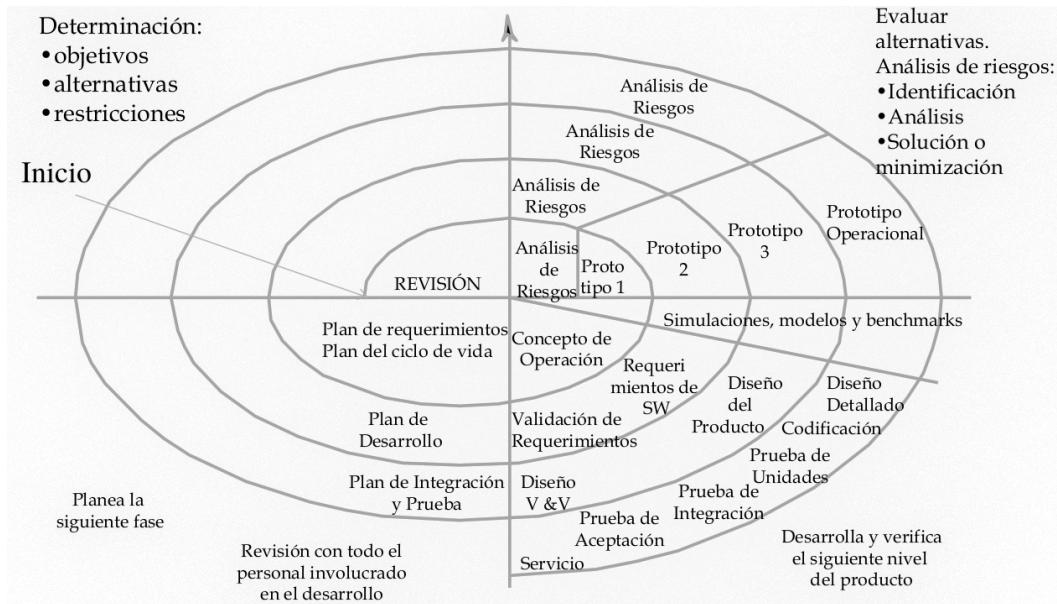


Figura 7: Etapas del modelo en espiral.

Es un **modelo iterativo** que combina las principales ventajas del **modelo de ciclo de vida en cascada** y el **modelo de construcción de prototipos**. Su principal característica es incorporar el **análisis de riesgos** en el propio ciclo de vida, de modo que los prototipos son usados para reducir el riesgo, incluso permitiendo finalizar el proyecto antes de embarcarse en el desarrollo final si no se considera viable.

Nota: Mientras que el modelo iterativo toma la filosofía iterativa del desarrollo de prototipos, el modelo en espiral toma directamente las ventajas de realizar prototipos.

5.1. Descripción del modelo

El proceso se representa como una espiral, en donde cada ciclo representa una fase del proceso, en función de lo mejor se ajuste al proyecto; por ejemplo, el ciclo más interno puede relacionarse con la factibilidad, el siguiente con la definición de requisitos, el siguiente, con el diseño, etc. Siempre se actúa en función a los riesgos que el proyecto presente, y por ello se incluyen las actividades de la gestión de riesgos para reducirlos.

Se definen un total de cuatro sectores:

1. **Definición de objetivos:** Objetivos de la fase actual y restricciones con las que deben alcanzarse. Finaliza con la identificación de diferentes alternativas que permitirían alcanzar estos objetivos.

2. **Análisis de riesgos:** Se identifican los riesgos de las diferentes alternativas, así como tratarlos, con el objetivo de escoger una de las alternativas.
3. **Desarrollo y validación:** Se realiza el análisis de viabilidad de las alternativas y, si se ha reducido el riesgo a niveles aceptables, se realizan las actividades de ingeniería que construyen el producto; es decir, se general entregables de los procesos clásicos, siguiendo, por ejemplo, un ciclo en cascada.
4. **Revisión y planificación:** Todas las personas implicadas, incluso clientes, revisan los productos desarrolladores y se planificaría la ejecución del siguiente ciclo. Se finaliza con la toma de decisión de finalizar o continuar el proyecto.

5.2. Ventajas

- Es mucho más realista que el ciclo de vida clásico.
- Permite la utilización de prototipos en cualquier etapa de la evolución del proyecto.
- Existe un reconocimiento explícito de las alternativas para realizar un proyecto.
- Existe una identificación de los riesgos, junto con las diferentes maneras de tratarlos, eliminando las ilusiones de avance.
- Se adapta a cualquier tipo de actividad, incluso alejadas de un ciclo de vida tradicional, como la consulta a asesores externos.

5.3. Desventajas

- El modelo en espiral se adapta bien a los desarrollos internos, pero necesita un ajuste para la contracción de software, al no contar con la flexibilidad para ajustar el desarrollo etapa por etapa (por ejemplo, aplazar decisiones, pararse en miniespirales para realizar secciones críticas con precaución, etc.).
- Requiere expertos en la evaluació de riesgos, para realizar evaluaciones apropiadas y no abocarse al desastre.
- Resulta difícil de controlar, y de convencer al cliente de que es manejable.

Nota: *En todo caso, podría plantearse la realización de un contrato como una iteración del ciclo en espiral.*

5.4. Análisis de riesgos

5.4.1. Definiciones

Activos Elementos de un sistema de información que soportan los objetivos de la organización (presentan un cierto valor). Por ejemplo: equipamiento, redes de comunicación, instalaciones, personas, etc.

Amenazas Qué puede pasarles a los activos, perjudicándolos consecuentemente. Pueden ser de origen natural, del entorno de trabajo, por defectos en el producto, y causados por las personas de forma accidental o deliberada.

Contramedidas Medidas de protección desplegadas para que las amenazas no causen tanto daño.

Riesgo Estimación del grado de exposición a que una amenaza se materialice sobre uno o más activos, causando perjuicios a la organización.

Análisis de riesgos Proceso para estimar la magnitud de los riesgos a los que está expuesta una organización.

Proceso de gestión de riesgos Proceso destinado a modificar un riesgo, para reducir su impacto.



Figura 8: Elementos del análisis de riesgos potenciales.

5.4.2. Anticipación a los riesgos

Consta de cuatro actividades principales:

1. **Identificar los riesgos** Se identifican los activos relevantes, a qué amenazas están expuestos, y los riesgos asociados. *Sommerville* clasifica los riesgos en:
 - Del negocio: Afectan a la organización (riesgos de mercado, mala reputación...).
 - Del proyecto: Afectan a la calendarización y los recursos.
 - Del producto: Afectan a la calidad del producto.
2. **Análisis de riesgos**: Se evalua, empleando una escala a intervalos y según la experiencia de los expertos, la probabilidad de que cada riesgo ocurra (baja, moderada y alta, por ejemplo), así como sus consecuencias o impacto (catastrófico, grave, tolerable y e insignificante, por ejemplo). A pesar de emplear una escala a intervalos, cada uno de estos debe sustentarse sobre medidas objetivas; por ejemplo, un riesgo tolerable será aquel cuyo impacto no excede los márgenes temporales o económicos del proyecto. Esta actividad concluye con la elaboración de una lista ordenada de riesgos por importancia.
3. **Planificación de riesgos**: Tras escoger aquellos riesgos más prioritarios, se establecen estrategias para gestionarlos, en base a la experiencia de nuevo:
 - Estrategias de prevención.
 - Estrategias de minimización.
 - Planes de contingencia: estar preparados para lo peor.

- Transferencia: subcontratación de expertos que gestionen y se responsabilicen de la gestión del riesgo.
4. **Gestión de riesgos:** Supervisar el desarrollo del proyecto continuamente, de forma que se detecten los riesgos tan pronto como aparezcan. Para ello, se definen indicadores cuantificables para cada riesgo, así como sus umbrales de alerta.

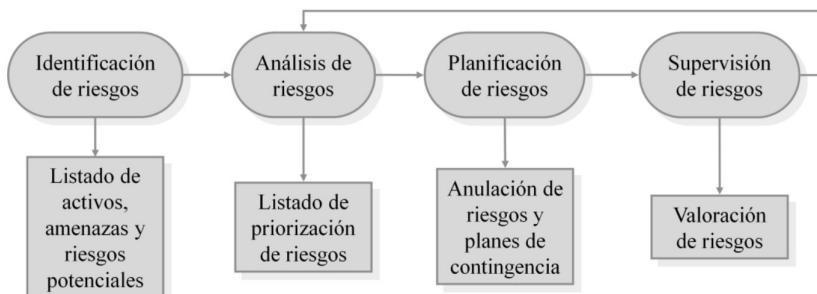


Figura 9: Administración del riesgos.

6. Metodologías ágiles

Cabe destacar, en todo lo visto hasta el momento, el esfuerzo puesto en:

- Documentar los proyectos.
- Buscar posibles fallos y seguir las correcciones y modificaciones que suponen, en lugar de centrarse en construir el software en sí.
- Garantizar la calidad del producto mediante un seguimiento riguroso de los procesos de construcción.
- Modificar los documentos resultantes, dado que no permanecen inmutables en el tiempo.

Como se puede ver, estas metodologías pesadas presentan una aproximación sistemática y disciplinada, con procesos fuertemente orientados a la documentación: la documentación no es un objetivo sino el medio para lograrlo y con calidad, y el formalismo da calidad a pesar de ralentizar el desarrollo.

El desarrollo ágil defiende la renuncia a utilizar estos modelos perfectos, se basa únicamente en que sean lo suficientemente buenos. Tratan de **centrar los esfuerzos en presentar un incremento software ejecutable**, restando importancia a los productos de trabajo intermedio, que no siempre es bueno.

En el manifiesto para el desarrollo ágil, firmado por *Kent Beck* entre otros, se establecía la valoración de:

- Individuos e intenciones sobre los procesos.
- software en funcionamiento sobre la documentación.
- Colaboración con el cliente sobre el contrato.
- Respuesta al cambio sobre el seguimiento de un plan.

Como se puede ver, estos modelos resultan ser una corriente opuesta a la tradicional, no con el objetivo de eliminarlos, sino de solventar sus limitadas capacidades de adaptación al cambio, dada la fragilidad de los requisitos y la variabilidad del entorno, y la disciplina de trabajo exigida a las personas.

6.1. Variabilidad del entorno

Las metodologías ligeras proponen una aproximación incremental en donde se valora la entrega frecuente; por ejemplo, desarrollando incrementos tras un par de semanas o de meses. Estos son valorados por el cliente, entrando a ser partícipe del proceso de software, para que ofrezca el *feedback* necesario, como nuevos requisitos o prioridades.

6.2. Fragilidad por las debilidades humanas

En contraposición a exigir una gran disciplina al personal para asegurar que los procesos se desarrollan correctamente, las metodologías ligeras aceptan tolerancia en la aplicación de los modelos, permitiendo que estos se adapten al equipo, y no al revés.

Algunos de los rasgos con los que las personas deberían contar son: competencia, enfoque común (entregar un incremento a tiempo), colaboración entre todos los *stakeholders*, habilidades y técnicas para la toma de decisiones, capacidad de resolución de problemas ambigüos (debido al cambio), confianza y respeto mutuo, y organización propia para lograr los objetivos.

6.3. Diferentes modelos de proceso

- Programación Extrema (PE).
- Desarrollo Adaptativo del Software (DAS).
- SCRUM.
- Cristal.
- Desarrollo conducido por características (DCC).
- Modelado Ágil.

6.4. Modelado ágil

A alto nivel, es una **colección de valores, principios y prácticas** (en resumen, buenas prácticas) que poder aplicar para realizar modelado y documentación de forma efectiva y ligera en un proyecto de desarrollo de software. Por ejemplo, es habitual la práctica de *Test-Driven Development*, que consta a su vez de las dos prácticas de *Test-First Development* y de refactorización.

Al final, el modelado ágil puede ser tomado más como una filosofía que como una metodología:

- Modelar con un propósito: Trabajar siempre con un objetivo en mente.
- Conocer los modelos y notaciones disponibles, y usar el más apropiado para cada situación.
- Viajar ligero: Conservar sólo la documentación o modelos que proporcionarán valor a largo plazo, y que deberán recibir por lo tanto un mantenimiento.

- El contenido es más importante que la representación: Solo es necesario que un modelo pueda ser interpretado por la audiencia a la que está dirigido.
- Adaptar a las necesidades del equipo.

6.5. Programación extrema

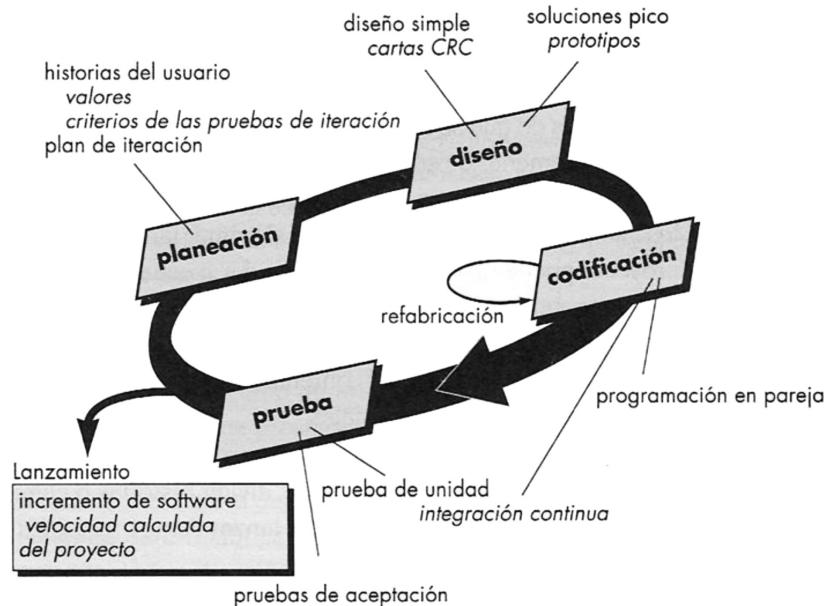


Figura 10: Etapas de la programación extrema.

Utiliza, preferiblemente, un enfoque orientado a objetos, y abarca un conjunto de reglas y prácticas que ocurren las siguientes cuatro actividades:

1. Planificación:

- **Creación de historias de usuario:** Describen características y funcionalidades requeridas (permiten establecer requisitos); cada historia se asemeja a un caos de uso, es escrita por el cliente, y este le asigna una prioridad.
- **Estimación del tiempo de implementación:** Los miembros del equipo evalúan cada historia y asignan un coste en semanas de desarrollo; si es superior a tres semanas, se segmenta la historia.
- **Nota:** *Pueden escribirse historias en cualquier momento.*
- **Plan de construcción:** Los clientes y el equipo se reúnen para decidir qué historias hay en el siguiente incremento del software, priorizando siguiendo una de las siguientes vías:
 - Todas las historias serán implementadas de un modo inmediato.
 - Las historias con **valor más alto** se implementarán al **principio**.
 - Las historias de **mayor riesgo** se implementarán al **principio**.
- **Cálculo de la velocidad del proyecto:** Tras cada lanzamiento, se recoge el número de historias implementadas, con la intención de:
 - Ayudar a estimar futuras fechas de entrega y programaciones.
 - Determinar si, hasta el momento, ya se ha realizado algún compromiso no viable.

2. **Diseño:** Sigue el principio KISS (*Keep It Simple, Stupid*), de modo que solo se implementa lo requerido, y se apoya en:

- **Tarjetas CRC** (Colaborador–Responsabilidad–Clase).
- **Soluciones de Pico:** Si se presenta un problema difícil, se recomienda construir un prototipo para probar dicha parte y analizarlo, de modo que se reduzca el riesgo a la hora de comenzar la verdadera implementación.
- **Refabricación (Refactoring):** Cambiar un sistema de software de tal manera que no altere el comportamiento externo del código y que mejore la estructura interna (mejorar el diseño del código después de que haya sido escrito).
- **Diseño como artefacto:** Se permite la modificación del diseño durante todas las fases de la construcción.

3. Codificación:

- Después de realizar el trabajo de diseño preliminar, se prefiere crear las **pruebas de unidad que ejerciten cada una de las historias, antes que el código** de las clases. Así, el desarrollador se centra en implementar lo necesario para pasar la prueba de unidad y, una vez escrito el código, puede evaluarse de inmediato.
- Un concepto clave durante la codificación es la programación en pareja, con el objetivo de mejorar la capacidad de resolución de problemas y asegurar la calidad. En la práctica, cada persona tiene un papel sutilmente diferente.
- Una vez unos desarrolladores finalizan su trabajo, el código escrito debe integrarse con el trabajo de otros; mediante esta estrategia de integración continua, se evitan mayores problemas de compatibilidad e interfaz. Se realizan “pruebas de humo”, que son pruebas de integración poco exhaustivas de todo el sistema con las que detectar los errores más importantes.

4. Pruebas:

- **Pruebas diarias:** Comprenderían, principalmente, las pruebas de unidad e integración, y deberían organizarse de forma que puedan ser automatizadas. Así, si se realizan a diario, se proporciona al equipo un indicador de progreso, así como una alarma fiable en caso de que las cosas vayan mal.
- **De aceptación o del cliente:** Estando especificadas por el cliente, se enfocan en las características generales y la funcionalidad del sistema, siempre sobre aspectos que el cliente pueda revisar. Se derivan de las historias de usuario implementadas.

Tema IV

Análisis de requisitos

1. Introducción

1.1. Requisito

Condiciones o capacidades de debe cumplir un sistema para satisfacer un contrato, norma o especificación.

1.1.1. Clasificación de los requisitos

- **Requisitos funcionales:** Declaraciones de los servicios o funcionalidades que implementará el sistema. *Ejemplo: El usuario tendrá la posibilidad de buscar por tamaño.*
- **Requisitos no funcionales:** Restricciones sobre los servicios o funciones ofrecidos por el sistema. Pueden ser temporales, de fiabilidad, robustez, de ajuste a estándares... Deben expresarse de forma cuantitativa. *Ejemplo: La duración de la sesión ha de ser superior a 30 minutos.*
- **Requisitos del dominio:** Provienen del entorno en el que se realiza la aplicación y pueden ser funcionales o no funcionales. *Ejemplo: usar el formato DICOM en medicina es a la vez un requisito no funcional y de entorno.*

1.2. Análisis de requisitos

Proceso de **estudio de las necesidades de los usuarios** para llegar a una definición de los requisitos obtenidos, incluyendo el proceso de refinamiento de los mismos.

Encontramos este proceso en todos los modelos de ciclos de vida, compartiendo las siguientes características comunes:

1. Realizan una abstracción de las características del sistema.
2. Representan el sistema de forma jerárquica.
3. Definen las interfaces del sistema, tanto externas como internas.
4. Sirven de base para etapas posteriores del diseño e implementación.
5. Se plantean por niveles: usuario, de sistema, esenciales y de implementación.
6. No prestan demasiada atención a la representación de las restricciones o criterios de validación. (excepto los métodos formales).

Rabenso dice: Por norma general, se debería dedicar un 40 % del tiempo de desarrollo de un proyecto al análisis y diseño, un 20 % a la codificación, y un 40 % a la fase de pruebas.

1.3. Personal implicado

1. **Analista:** Es el encargado de la comunicación con los stakeholders y los desarrolladores, sirviendo de puente entre ambos. Para este labor debe de tener conocimiento multidisciplinar, buenas dotes de comunicación y ser capaz de traducir ideas vagas de necesidades de software en un conjunto concreto de funciones y restricciones.

2. **Stakeholders:** personal involucrado en el proyecto, incluyendo a los usuarios finales, los ingenieros trabajando en el proyecto, el administrador de negocio y los expertos del dominio del sistema. Podemos distinguir entre involucrados y comprometidos.

1.4. Razones por las que el análisis de requisitos es difícil

Sommerville establece las siguientes razones:

- Los stakeholders a menudo no conocen realmente lo que desean obtener del sistema.
- Los stakeholders expresan los requisitos con sus propios términos. Los ingenieros de requisitos deben comprenderlos.
- Diferentes stakeholders tienen requisitos distintos que expresan de varias formas. Los ingenieros de requisitos tienen que descubrir todas las fuentes potenciales de requisitos así como las partes comunes y en conflicto.
- Puede haber condicionantes políticos y de la organización.
- El entorno económico y de negocios en el que se lleva a cabo el análisis es dinámico, y cambia durante el proceso de análisis, por lo que la importancia de ciertos requisitos puede cambiar.

2. Obtención de requisitos

El proceso de análisis de requisitos tiene las siguientes fases:

- **Identificar las fuentes de información** relevantes para el proyecto.
- **Realizar las preguntas apropiadas** para comprender sus necesidades.
- **Analizar la información recogida** para detectar aspectos poco claros.
- **Confirmar con los usuarios** lo que se ha comprendido de los requisitos.
- **Sintetizar los requisitos** en un documento de especificación apropiado.

2.1. Técnicas de comunicación y recogida de información

Aparecen a raíz de uno de los problemas más comunes: cómo poner en contacto a usuarios y técnicos para establecer unos requisitos entendibles y aceptados por todos.

1. **Entrevistas.**
2. **Desarrollo conjunto de aplicaciones:** Se crean equipos de usuarios y analistas que determinan conjuntamente las características que debe tener el software. *Ejemplo TFEA.*
3. **Prototipado.**
4. **Observación:** Consiste en analizar *in situ* cómo funciona la unidad o el departamento a informatizar.
5. **Estudio de documentación:** recopilar muestras de los impresos que se utilizan en la organización.
6. **Cuestionarios.**

7. ***Brainstorming***: Consiste en reuniones de entre 4 y 10 personas en las que se sugiere toda clase de ideas sin juzgarse su validez, para luego analizar cada propuesta de modo detallado.

8. ***Casos de uso***.

2.2. TFEA

Son un conjunto **Técnicas de Facilitación de Especificación de la Aplicación**.

1. Redacción de un documento inicial como resultado de reuniones entre analista y cliente:
 - a. Descripción del problema.
 - b. Objetivos.
 - c. Propuesta de solución (si es posible).
2. Creación de listas individuales de:
 - a. Objetos del sistema y del entorno.
 - b. Operaciones que relacionan objetos.
 - c. Restricciones.
 - d. Rendimiento.
3. Creación de una lista conjunta a partir de las individuales, sin eliminar nada.
4. Discusión de la lista conjunta (deben estar representados el analista, el cliente y el equipo de desarrollo).
5. Redacción de miniespecificaciones.
6. Redacción de un borrador de la especificación.

Este enfoque presenta numerosas ventajas, como la comunicación multilateral de todos los involucrados en el proyecto, el refinamiento instantáneo y en equipo de los requisitos y la obtención de un documento de base para el proceso de análisis.

3. Análisis de requisitos

3.1. Análisis de requisitos del sistema

El objetivo es **conseguir representar un sistema en su totalidad**, incluyendo hardware, software y personas, mostrando la relación entre diversos componentes y **sin entrar en la estructura interna** de los mismos.

El análisis de sistemas tiene las siguientes fases:

1. **Identificación de las necesidades del cliente**:

Recogida y clasificación de requisitos. Distinguimos entre:

- Requisitos prescindibles e imprescindibles.
- Requisitos normales, esperados e estimulantes (lo que el cliente pide, lo que el cliente necesita y los que le resultarían satisfactorios respectivamente).

2. Análisis de alternativas:

Estudio de las diferentes alternativas que solucionen el problema o parte de él, incluyendo las ya implementadas por terceros. Para ello podemos utilizar técnicas como el **análisis paramétrico**, diagramas **DAFO** (Debilidades, Amenazas, Fortalezas y Oportunidades) o el **valor monetario esperado** (EMV).

	Alternativa 1	Alternativa 2	Alternativa 3
Inversión inicial	Nula	Moderada	Grande
Coste funcionamiento	Grande	Moderado	Moderado
Tiempo amortización	Nulo	Moderado	Grande
Fiabilidad	Moderada	Pequeña	Grande
Mantenimiento	Nulo	Grande	Moderado
Flexibilidad	Alta	Nula	Alta

Tabla 2: Análisis paramétrico de un sistema de clasificación de paquetes.

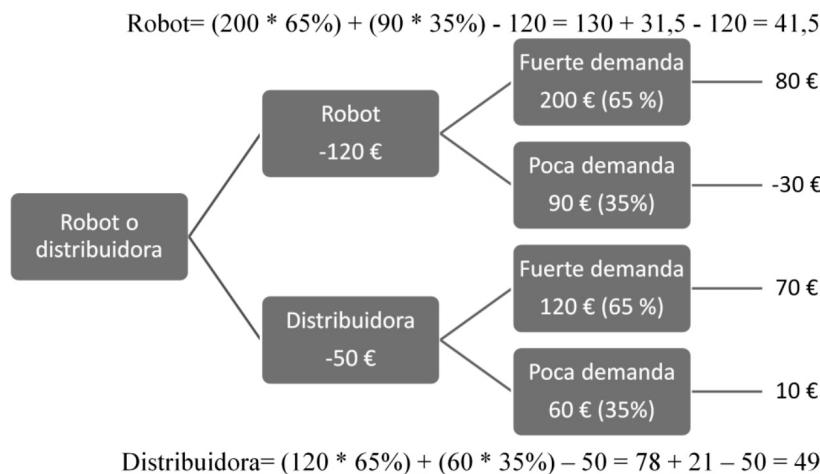


Figura 11: EMV de un sistema de clasificación de paquetes.

3. Estudio de viabilidad del sistema:

Pretende determinar si merece la pena la construcción del software: si el sistema puede implantarse con las restricciones de coste y tiempo con la tecnología actual y con un alcance que le sea útil al usuario.

Para ello se hace un estudio de viabilidad desde el punto de vista económico, técnico, legal, operativo y de plazos.



Figura 12: La calidad del software está restringida en mayor medida, pero no únicamente por el tiempo, el coste y el alcance.

4. **Definición del sistema para que sea la base del trabajo posterior:** Para ello tenemos los siguientes diagramas:

- **Diagrama de contexto:** Representa el sistema en relación con su entorno. Define los límites del sistema y muestra todos los productores y consumidores de información. El sistema se relaciona con los agentes externos mediante flujos de información.

Pressman recomienda realizar este diagrama separado en secciones: Entrada, Salida, interfaz de usuario, función y control de sistema y mantenimiento y diagnóstico.

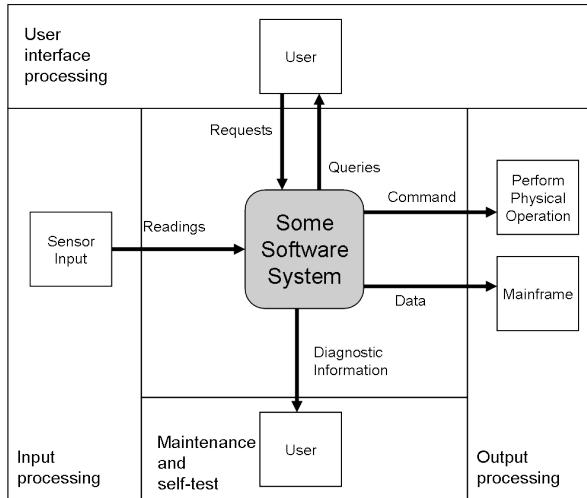


Figura 13: Ejemplo de diagrama de contexto siguiendo la plantilla.

- **Diagrama de flujo:** Determinan el flujo de entre los diferentes subsistemas a través de diagramas de flujo convencionales.

3.2. Análisis de requisitos del software

El análisis de requisitos del software tiene como objeto **desarrollar una representación del software que pueda ser revisada y aprobada por el cliente** que permitirá valorar la calidad del software una vez construido.

Desde el punto de vista del analista, esta fase define con mayor precisión las funciones y rendimiento del software, las interfaces con otros componentes y las restricciones que debe cumplir.

3.2.1. Principios del análisis de requisitos del software

1. **Identificar y representar el ámbito de información del sistema:** La tarea que realiza el software consiste en procesar información, representada tanto por **datos** como por **sucesos o eventos**. El ámbito de información admite dos puntos de vista: flujo de datos y flujo de control.
2. **Modelar la información, la función y el comportamiento del sistema:** Se emplean **modelos de datos** (información que transforma el software), **modelos de procesos** (procesos que transforman la información) y **modelos de control** (comporta-

miento del sistema).

Utilidades de los modelos:

- Ayudan al analista a entender la información, función y comportamiento del sistema.
- Sirven de base para el trabajo del diseñador.
- Sirven para validar el producto una vez desarrollado.

3. **Descomponer el problema de forma que se reduzca la complejidad.**

4. **Avanzar desde lo más general a lo más detallado.**

4. Especificación

4.1. Especificación del sistema

Documento que describe la función, rendimiento y restricciones que el sistema debe cumplir, limitando cada uno de sus componentes.

Es labor del analista realizar una evaluación inicial, determinando si:

1. Se ha delimitado correctamente el ámbito.
2. Se han definido correctamente las funcionalidades, el comportamiento, las interfaces y el rendimiento.
3. Las necesidades de usuario y el análisis de viabilidad justifican el desarrollo del proyecto.
4. Las percepciones acerca del objetivo del proyecto del cliente y el analista coinciden.

El analista deberá hacer además una evaluación técnica, comprobando:

1. Todos los detalles técnicos están bien definidos.
2. La especificación sirve como base para las siguientes fases.
3. Las estimaciones de riesgos, tiempos y costes se corresponden con la complejidad del proyecto.

4.2. Especificación del software

Documento que define, de forma completa, precisa y verificable, los requisitos, el diseño, el comportamiento u otras características de un sistema o componente del mismo.

Es el documento que culmina el análisis de requisitos, conteniendo:

1. Descripción detallada del ámbito de información.
2. Funciones y comportamiento asignados al software.
3. Restricciones de rendimiento y diseño.
4. Pruebas de aceptación.

4.2.1. Principios de especificación

1. Debe modelar el dominio del problema.
2. Es necesario separar funcionalidad e implementación.
3. El lenguaje de especificación debe estar orientado al proceso.
4. Debe abarcar todo el sistema del que el software es parte.
5. Debe abarcar también el entorno del sistema.
6. La especificación debe ser operativa.

4.2.2. Características del documento de especificación

1. **Preciso:** Cada requisito debe tener una única interpretación. Evitar el lenguaje natural.
2. **Completo:** Una especificación de requisitos software (ERS) está completa si:
 - Incluye todos los requisitos significativos del software.
 - Define la respuesta software a todas las entradas en todas las situaciones.
 - Está conforme con cualquier estándar de especificación que se deba cumplir.
 - Están etiquetadas y referenciadas todas las tablas, figuras y diagramas del texto, y están definidos todos los términos y unidades de medida.
 - No contiene la expresión *TBD* (por determinar). A veces es necesario utilizarla, y debe acompañarse de:
 - Una descripción de las condiciones que han causado el *TBD*.
 - Una descripción de qué hay que hacer para eliminar el *TBD*.
3. **Fácil de verificar.** Cualquier requisito al que hace referencia se puede verificar mediante algún procedimiento finito y efectivo en coste.
4. **Consistente:**
 - Ningún conjunto de requisitos descritos en la especificación deben describir el mismo objeto real pero utilizar distintos términos para designarlo.
 - Las características especificadas de objetos reales no pueden estar en conflicto.
 - Ningún conjunto de acciones determinadas deben entrar en conflicto lógico o temporal.
5. **Fácil de modificar.**
6. **Facilidad para identificar el origen y las consecuencias de cada requisito:** se consigue estableciendo la trazabilidad.
7. **Facilidad de utilización durante la fase de explotación y mantenimiento.**

5. Verificación y Validación de requisitos

5.1. Estrategias

Encontramos diferentes estrategias para la verificación y validación de requisitos:

1. **Comprobación de validez.** ¿Responde a todas las necesidades?
2. **Comprobación de consistencia.** ¿Hay incongruencias en los requisitos?
3. **Comprobación de totalidad.** ¿Nos falta algo por especificar?
4. **Verificaciones de realismo.** ¿Es posible implementar la funcionalidad requerida?
5. **Verificabilidad.** ¿Soy capaz de crear pruebas que determinen si se pasa cada requisito?

5.2. Técnicas

Existen varias técnicas de validación de requisitos:

1. **Revisiones de requisitos** Proceso manual de verificación del documento de requisitos que busca encontrar anomalías y omisiones.
2. **Construcción de prototipos.**
3. **Generación de casos de prueba.** Los requisitos deben poder probarse. Si una prueba es difícil o imposible de diseñar, suele significar que los requisitos serán difíciles de implementar, por lo que deben ser reconsiderados.
4. **Análisis de consistencia automática.** Si los requisitos se expresan como un modelo del sistema en una notación estructurada o formal, las herramientas CASE(Computer Aided Software Engineering) deben verificar la consistencia del modelo.

6. Administración de requisitos

La especificación de requisitos es un proceso iterativo, por lo tanto hace falta un proceso formal para gestionar la realización de modificaciones a los requisitos sin descuidar la calidad de la documentación de requisitos.

Para ello debemos llevar a cabo un registro de modificaciones

6.1. Clasificación cualitativa

1. Requisitos duraderos

2. Requisitos volátiles

- **Cambiantes:** Cambian debido a los cambios en el ambiente en que opera la organización.
- **Emergentes:** Surgen al incrementar la comprensión del cliente en el desarrollo del sistema.
- **Consecutivos:** Son resultado de la introducción del sistema de cómputo.
- **De compatibilidad:** Dependen de sistemas particulares o procesos de negocio dentro de la organización.

6.2. Clasificación cuantitativa

Asignándole un valor de estabilidad(baja, media y alta).

6.3. Etapas

El proceso de administración de requisitos tiene dos etapas:

1. Planificación:

En esta etapa debemos de decidir el nivel de detalle estableciendo:

- a. **La identificación de requisitos.**
- b. **Un proceso de administración del cambio.**
- c. **Políticas de rastreo:** Requisito – Fuente, Requisito – Requisito, y Requisito – Diseño.
- d. **Ayuda de herramientas CASE:** Establecemos las herramienta de Computer Aided Software Engineering que utilizaremos para aumentar la productividad durante el desarrollo.

2. Administración del cambio:

Se aplica para todos los cambios en los requisitos, teniendo las siguientes sub-etapas:

- a. **Análisis del problema y especificación del cambio.**
- b. **Análisis del cambio y costeo.**
- c. **Implementación del cambio.**

Tema V

Análisis estructurado

1. Introducción

Modelo Descripción simplificada del sistema que se utiliza en análisis de requisitos como herramienta sobre la que trabajar con el cliente para construir un sistema adecuado a sus necesidades.

1.1. Problemas del análisis clásico

Consistía en redactar especificaciones funcionales, en forma de documentos de texto. Problemas:

1. **Monolíticas:** Había que leerlos de principio a fin.
2. **Redundantes.**
3. **Ambiguas:** Causado por el uso del lenguaje natural.
4. **Imposibles de mantener o modificar:** Al ser redundantes, cualquier modificación de una parte podía provocar una inconsistencia, obligando a leer todo el documento debido a que era una “unidad monolítica”.

1.2. Soluciones al análisis clásico

Surgieron nuevos métodos de análisis para obtener especificaciones:

1. **Gráficas:** comentadas, únicamente material de referencia.
2. **Particionadas:** Facilitan la lectura de partes individuales de la especificación.
3. **Mínimamente redundantes.**
4. **Transparentes:** Fáciles de leer y comprender.

2. Técnicas de especificación y modelado

El análisis estructurado **propone la descripción de los sistemas según 3 puntos de vista:**

1. **Punto de vista de los datos (información):** Información que utiliza el sistema, haciendo explícitas las relaciones entre datos.
 - Diagramas Entidad–Relación (**DER**).
 - Diagramas Estructura de Datos (**DED**).
2. **Punto de vista del proceso (función):** Se centra en qué hace el sistema. Conjunto de operaciones de proceso de información.
 - Diagramas de Flujo de Datos (**DFD**).
 - Especificaciones de Procesos: término genérico que engloba la definición de como un sistema transforma unas entradas en salidas mediante pseudocódigo, lenguaje natural, diagramas de flujo...

3. **Punto de vista del comportamiento (tiempo):** Se centra en cuándo sucede algo en el sistema. Sucesión de estados o modos de funcionamiento. Se indican las condiciones o eventos que hacen que el sistema pase de un modo a otro.

- Diagramas de Flujo de Control.
- Especificaciones de Control.
- Diagramas de Estados.

Utilizando estos modelos en conjunto podremos obtener una descripción detallada del **mismo sistema**.

	Información	Función	Tiempo
Información	D. entidad-relación D. de estructura de datos Matriz entidad/entidad Diagramas de clases		
Función	D. de flujo de datos Matriz función/entidad Diagrama de clases D. de colaboración	D. de flujo de datos D. de casos de uso D. de estructura de datos Tarjetas CRC D. de componentes D. de despliegue D. de actividad	
Tiempo	H ^a de vida de la entidad D. de estados D. de secuencia	Redes de Petri D. de estados D. de secuencia D. de actividad	D. de flujo de control D. de estados

Tabla 3: Métodos de modelado según la dimensión del sistema que modelan

2.1. Diagramas de flujo de datos (DFD)

Representa la programación desde un **punto de vista funcional**, esto es, las **entidades básicas son funciones o procesos** que transforman unos datos de entrada en salidas. El sistema se describe como el flujo de información a través de estas funciones.

1. Las funciones del sistema.
2. Las interacciones entre funciones: a dónde va la información de salida de una determinada función.
3. Las transformaciones de datos del sistema: donde se almacena la información después de pasar por las funciones.
4. Las transformaciones entrada-salida: por donde entra esa información **al sistema** y a dónde sale.

Los DFD **NO** representan el **comportamiento**, sólo dicen lo que hace el sistema pero **NO** indican:

1. **Cuando se hace.**
2. **En qué secuencia se hace.**

2.1.1. Elementos

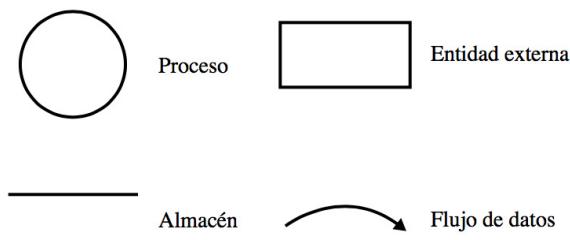


Figura 14: Notación estándar utilizada para representar los elementos del DFD.

1. **Procesos:** Representan elementos software que transforman información.
2. **Entidades externas:** Representan elementos del sistema informático o de otros sistemas adyacentes que producen o consumen información transformada por el software. Los flujos de datos que comunican el sistema con las entidades externas representan las interfaces del sistema. **Sólo aparecen en el diagrama de contexto.**
3. **Almacenes de datos:** Representan información almacenada que puede ser utilizada por el software. En la mayoría de los casos, utilizaremos almacenes de datos cuando los procesos intercambien información pero no estén sincronizados.
4. **Flujos de datos:** Representan datos o colecciones de datos que fluyen a través del sistema. Conectan los procesos con otros procesos, con entidades externas o con almacenes de datos. Contienen información de las 3 dimensiones, aunque normalmente sólo representan las dimensiones de Función e Información.
Según su **dimensión temporal**:
 - a. **Discretos** (*Flecha con una cabeza*): Representan movimiento de datos en un instante determinado de tiempo.
 - b. **Continuos** (*Flecha con con dos cabezas*): Transmisión continua de información.

2.1.2. Niveles

Los **DFD** permiten la representación del sistema en múltiples niveles de abstracción, de esta forma se pueden representar gráficos con un número reducido de procesos (máximo 7 ± 2). Deberíamos usar 7-8 niveles como mucho.

1. **Nivel 0: Diagrama de Contexto**
Representa el sistema como un único proceso, el **proceso 0**. Se representan también las entidades que interactúan con el mismo, dejando claro los **límites del sistema**.
2. **Nivel 1: Diagrama 0**
Denominado así porque *descompone el proceso 0* en sus funciones principales. Es interesante que éstas sean *independientes* y que procesen *las mismos flujos* que el de nivel 0.
3. **Niveles $2 \dots n - 1$**
Continuamos descomponiendo los procesos en subprocesos, recogiendo las interfaces (flujos) de nivel superior y asignándoselas a subprocesos.

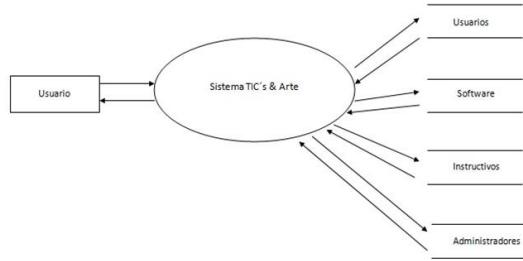


Figura 15: DFD de nivel 0 de un sistema.

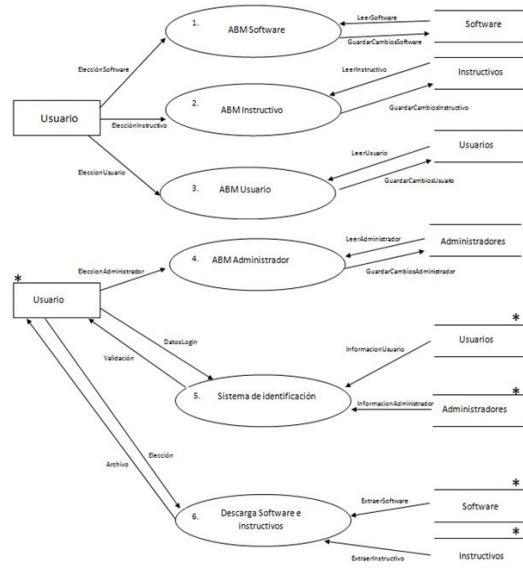


Figura 16: DFD de nivel 1 de un sistema.

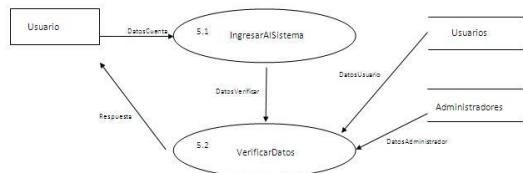


Figura 17: DFD de nivel 2 del proceso 5 del sistema.

4. Nivel n

Contiene los procesos primitivos, que no podemos descomponer. Si queremos describirlo debemos utilizar técnicas de especificación de procesos.

Los procesos en el DFD tienen un **identificador numérico**:

1. Nivel 0: solo tiene un proceso: el **“0”**
2. Nivel 1: se le asigna a cada proceso un numero secuencialmente: **{1, 2, 3... }**
3. Nivel 2: a cada descomposición de cada proceso se le asigna un número secuencial

después de un punto. Si descomponemos el *proceso 1* en 3 tendríamos los procesos: **{1.1, 1.2, 1.3}**

4. Niveles 2..N: a partir de aquí vamos concatenando la versión que descomponen con el número de secuencia **sin utilizar punto**. Descomponiendo el proceso *1.1* en 2 tendríamos: **{1.11, 1.12}**

2.2. Especificaciones de proceso (PSPEC)

El **Process Specification** es un documento que describe textualmente los detalles de un proceso, indicando cómo es el proceso de transformación de una información de entrada en otra de salida.

Debe ser **breve** (menor a una página) e incluir siempre **precondiciones** y **postcondiciones**. Para su elaboración, se utilizan diferentes técnicas de especificación:

1. Lenguaje natural.
2. Diagrama de flujo.
3. Lenguaje estructurado (Pseudocódigo).
4. Árboles de decisión.
5. Tablas de decisión.

2.3. Diagramas de flujo de control (DFC)

Los **Diagramas de Flujo de Control (DFC)** especifican todo el flujo de sucesos, señales y condiciones de datos del sistema.

Los DFC se crearon como diagramas complementarios de los DFD para suplir las carencias de los mismos:

- **NO** representan el **procesamiento ni la transformación** de los flujos de control (eso se hace con las CSPECs).
- **NO** representan los **estados del sistema**.

Para solucionar esto el DFC añade las siguientes entidades:

- **Flujos de control:** información que determina si un proceso se activa o se detiene.
- **Almacenes de control:** Sirven para almacenar la información de control (estado del sistema)
- **Condiciones de datos:** Flujos de control generados por un proceso.
- **Ventanas de control:** (representadas por una barra) Indican el procesamiento de señales de control. Su comportamiento se define en las especificaciones de control (CSPEC).
- **Activadores de procesos** Señales de control especiales que activan o desactivan procesos tomando dos posibles valores: **ON** y **OFF**. Siguen la jerarquía de los modelos, es decir: un proceso está activo si y sólo si todos sus antecesores están activos.
Por defecto: si no tiene activador, un proceso se considera activo si sus antecesores lo están.

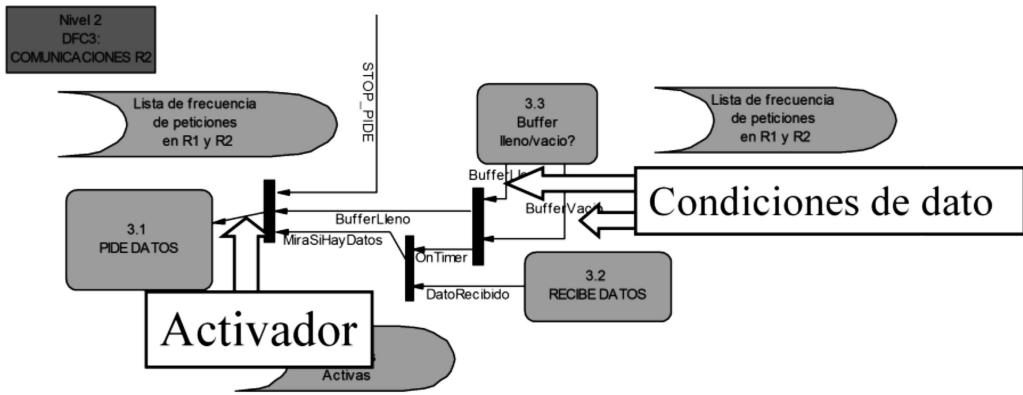


Figura 18: Ejemplo de un sistema de traducción. Préstese atención a las condiciones de dato: *buffer lleno/buffer vacío*, las ventanas de control (barras) y el activador.

2.3.1. Construcción de un DFC

1. Se elabora una jerarquía de DFCs paralela a la de los DFDs.
2. Cada para DFC/DFD representa los mismos procesos y las mismas entidades externas.
3. Sólo introducimos las señales de control que no estén implícitas en el DFD.
4. A continuación, cada DFC se desarrollará en una especificación de control (CSPEC).

2.4. Especificaciones de control (CSPEC)

El Control **SPECification** es un documento que **especifica de manera secuencial o combinacional el comportamiento del programa**. Indicando bajo qué condiciones se emiten las señales de control que activan los procesos.

Existirá por lo tanto una **relación 1:1 entre los CSPEC y los DFC** de la jerarquía ya que el primero determina la activación del segundo, utilizando las **ventanas de control de los DFC como interfaces**.

Para su elaboración, se utilizan diferentes técnicas de especificación:

1. Lenguaje estructurado (Pseudocódigo).

```

SI BufferVacio
    MIENTRAS NO BufferLleno
        ESPERA Retardo
        OnTimer
    FIN MIENTRAS NO
FIN SI

```

Figura 19: Ejemplo pseudocódigo para la activación de la función OnTimer.

2. En sistemas combinacionales:

	MiraSiHayDatos
3.1	1
3.2	0
3.3	0

Tabla 4: Ejemplo de tabla de activación de un proceso.

- Tablas de activación de procesos
- Tablas de decisión

CONDICIONES					
Condición 1	Sí	Sí	No	No	No
Condición 2	Sí	-	-	-	-
Condición 3	-	Sí	-	-	-
Condición 4	-	-	Sí	-	-
ACCIONES					
Acción 1	X		X	X	X
Acción 2		X		X	X

Tabla 5: Ejemplo de tabla de decisiones de un proceso.

3. En sistemas secuenciales:

- Diagramas de estados

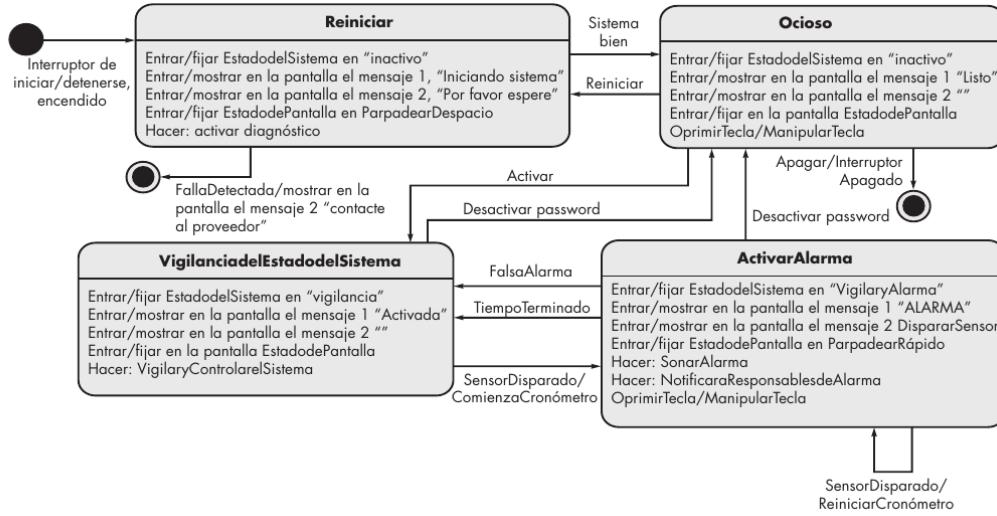


Figura 20: Ejemplo de un diagrama de estados.

- Redes de Petri

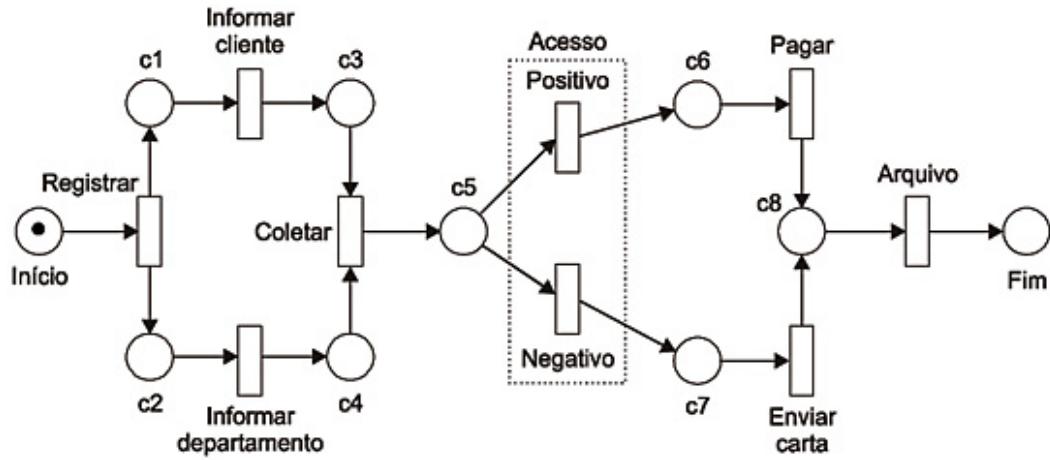


Figura 21: Proceso de generación de una reclamación modelado en redes de Petri.

2.5. Diagramas entidad–relación (DER)

Los Diagramas Entidad–Relación resuelven las insuficiencias que se pueden producir en la representación de los datos mediante los almacenes en los diagramas de flujo de datos (DFD), mostrando no sólo la información contenida si no las relaciones que existen entre los datos almacenados.

2.6. Comprobaciones a realizar sobre una especificación estructurada

Debemos revisar si nuestra especificación cumple las siguientes propiedades.

1. **Compleción:** Si los modelos son completos.
2. **Integridad:** Si no existen contradicciones ni incoherencias entre modelos.
3. **Exactitud:** Si los modelos cumplen los requisitos del usuario.
4. **Calidad:** Estilo, legibilidad y facilidad de mantenimiento.

Recomendación: checklist que compruebe a fondo que se cumplen estas propiedades.

2.7. Técnicas matriciales

Las técnicas matriciales se utilizan **principalmente para ayudar a verificar la consistencia entre los componentes de distintos modelos** de un sistema.

1. **Matriz entidad/funcióñ:** Visualiza las relaciones existentes entre las funciones que lleva a cabo un sistema y la información necesaria para soportarlas.
Los elementos de las filas son entidades o relaciones presentes en el diagrama entidad–relación, mientras que los de las columnas pueden ser funciones de alto nivel representadas en un diagrama de flujo de datos.
En cada celda se incluyen las **acciones** que puede realizar la función (Insertar, Leer, Modificar y Borrar).
2. **Matriz entidad/entity:** Muestra las relaciones normales del diagrama entidad–relación.
3. **Matriz evento/entity:** Muestra las acciones que provocan los eventos sobre las entidades: (Insertar, Leer, Modificar y Borrar).

Entidades	Funciones		
	Gestionar Presupuesto	Gestionar Cliente	...
Cliente	L	I, M, B	
Presupuesto	I, M, B		
...			

Tabla 6: Ejemplo de matriz entidad/función.

Entidades	Entidades		
	Cliente	Presupuesto	...
Cliente		Tiene	
Presupuesto			
...			

Tabla 7: Ejemplo de matriz entidad/entidad.

Eventos	Entidades		
	Cliente	Presupuesto	...
Datos del cliente	I, M, B		
Datos del presupuesto	I	I, M, B	
...			

Tabla 8: Ejemplo de matriz evento/entidad.

3. Metodología del análisis estructurado

3.1. Fases

1. Creación del modelo de procesos: Creamos los niveles de DFDs y las PSPEC.
2. Creación del modelo de control: para cada DFD creamos un DFC y una CSPEC.
3. Creación del modelo de datos: si los datos son complejos utilizamos un DER.
4. Consistencia entre modelos: podemos hacer una lista de comprobación además de tablas para comprobar la coherencia.

3.2. Modelos complementarios

Fuera de estas fases tenemos dos modelos más:

1. **Modelo esencial o lógico:** es un modelo de lo que el sistema debe hacer para satisfacer los requisitos del usuario. En él no figura nada acerca de cómo se va a implementar el sistema, como los ficheros de backup, o la secuenciación de proceso.
2. **Modelo de implementación:** En él se pueden mostrar detalles técnicos:
 - Elección de dispositivos y formatos E/S.
 - Volúmenes de datos, backups, dispositivos de almacenamiento.
 - Tiempos de respuesta.
 - Seguridad.

Tema VI

Pruebas del software

1. Conceptos

Verificación Proceso de evaluación de un sistema o de uno de sus componentes para determinar si los productos de una fase dada satisfacen las condiciones impuestas al principio de dicha fase (IEEE). *¿Son todas las fases **coherentes** entre sí y correctas? ¿Estamos **construyendo correctamente el producto**?*

Validación Proceso de evaluación del sistema o de uno de sus componentes durante o al final del desarrollo para determinar si satisface los requisitos especificados (IEEE). *¿Cumple el sistema con los requisitos **del cliente**? ¿Estamos construyendo **el producto correcto**?*

Pruebas Actividad en la que un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, y cuyos resultados son observados, registrados y evaluados comprobando algún aspecto.

Caso de prueba Conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular.

Ejemplo: ejercitarse en un camino concreto de un programa o verificar el cumplimiento de un determinado requisito.

Fallo Incapacidad de un sistema o de alguno de sus componentes para realizar las funciones requeridas dentro de los requisitos de rendimiento especificados.

Defecto In corrección en el software que **genera un fallo**

Ejemplo: un proceso, una definición de datos o un paso de procesamiento incorrectos en un programa.

Error

- Diferencia entre un valor calculado, observado o medido y el valor verdadero, especificado o teóricamente correcto.
- Resultado incorrecto.
- Defecto.
- Acción humana que conduce a un resultado incorrecto.

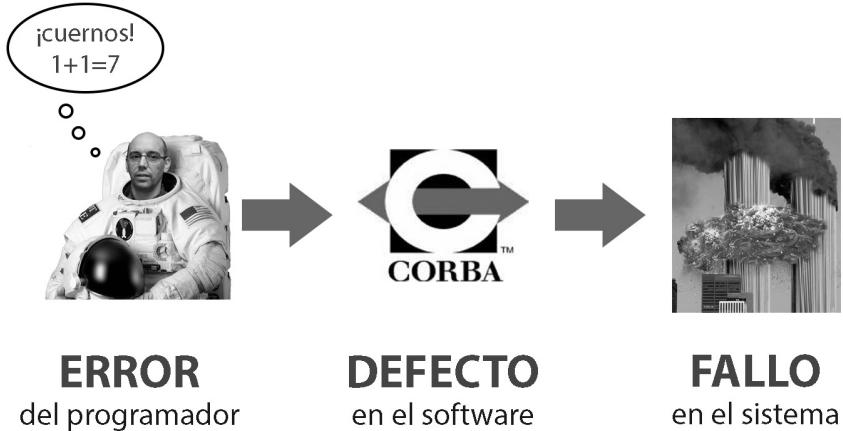


Figura 22: Recuerda: error, defecto y fallo no son lo mismo.

2. Filosofía de las pruebas del software

La prueba exhaustiva del software es impracticable: **no** se pueden probar todas las posibilidades incluso en programas pequeños y sencillos. **El objetivo de las pruebas es la detección de defectos en el software en la menor cantidad de tiempo y con el menor consumo de recursos posible.**

Descubrir un defecto constituye el éxito de la prueba, al permitir la rectificación en el software, lo que conlleva una **mejora de la calidad**. Los defectos no siempre son el resultado de una negligencia, ya que en su aparición influyen diversos factores.

2.1. Recomendaciones

1. **A todas las pruebas se les debería poder hacer un seguimiento hasta los requisitos del cliente.**
2. **Las pruebas deberían planificarse mucho antes de que empiecen:** La planificación de pruebas puede comenzar tan pronto esté completo el modelo de requisitos. La definición detallada de los casos de prueba puede empezar tan pronto como el modelo de diseño se ha consolidado.
3. **El 80 % de los errores surgen al hacer el seguimiento del 20 % de los módulos del Software (Principio de Pareto).**
4. **Las pruebas tendrían que hacerse de lo pequeño hacia lo grande.**
5. **NO son posibles pruebas exhaustivas.** Sin embargo, es posible cubrir adecuadamente la lógica del programa y asegurarse de que se han aplicado todas las condiciones en el diseño a nivel de componente.
6. **Las pruebas deberían ser realizadas por un equipo independiente del de desarrollo:** Lo ideal sería que probase el software el peor enemigo de quien lo construyó.
7. **Cada caso de prueba debe definir el resultado de salida esperado y compararlo con el realmente obtenido.**

8. **Se debe inspeccionar a conciencia el resultado de cada prueba** para así poder descubrir posibles síntomas de defectos.
9. **Al generar casos de prueba se deben incluir tanto datos de entrada válidos y esperados como no válidos e inesperados.**
10. **Las pruebas deben centrarse en probar si el software:**
 - No hace lo que debe hacer.
 - Hace lo que no debe hacer.
11. **Se deben evitar los casos desecharables:** No documentados o diseñados sin cuidado.
12. **No deben hacerse casos de prueba suponiendo que no hay defectos en los programas.**
13. **Las pruebas son una tarea tanto o más creativa que el desarrollo de software.**

3. Jerarquía de la documentación de diseño de pruebas

De acuerdo con el **estándar IEEE 829** Se generará la siguiente documentación:

1. **Plan de pruebas:** Establece las líneas generales del plan, especificando qué elementos y funcionalidades se van a probar, personal responsable, metodologías seguidas...
2. **Especificación del diseño de pruebas:** Detalla los casos de prueba y los resultados esperados así como los criterios de paso de prueba.
3. **Especificación de los casos de prueba:** Definen los datos de prueba (entradas y salidas) usados en la ejecución de los casos de prueba.
4. **Especificación de procedimiento de las pruebas:** Detalla cómo se ejecutará cada uno de los casos de prueba y los pasos necesarios a seguir.
5. Documentos de ejecución (**para Taboada el IEEE 1008**)¹:
 - **Histórico de pruebas:** provee cronológicamente detalles relevantes sobre la ejecución de las pruebas informando de cuáles fueron ejecutadas, quién las ejecutó, en qué orden y si pasaron o fallaron la prueba.
 - **Informe de incidentes:** informa de cualquier problema, incidente o defecto que ocurra durante la ejecución de las pruebas que requiera investigación.
6. **Informe resumen de pruebas:** sumario de los resultados y evaluaciones y recomendaciones basadas en éstos. Realizado tras la finalización de la ejecución de las pruebas.

¹Rabenso divide la documentación de pruebas en dos estándares: el IEEE 829 y el 1008. De acuerdo, con el IEEE 829–2008, esto no sería necesario pues éste estándar es más completo e incluye todas las fases.

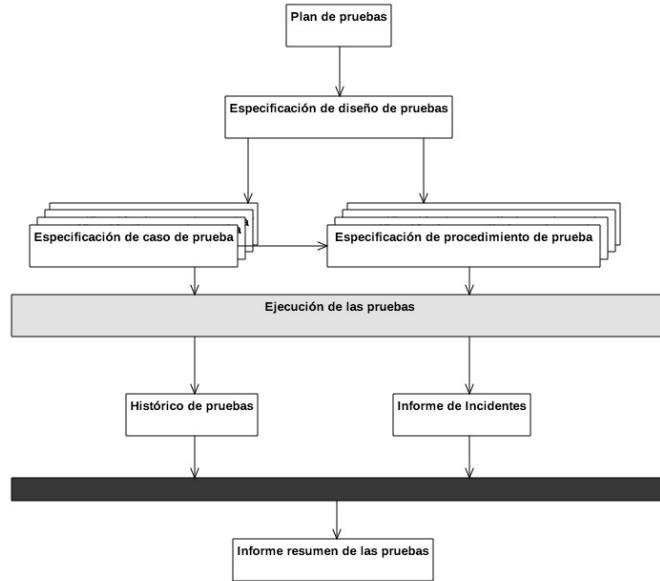


Figura 23: El estándar IEEE 829–2008 para la realización de pruebas del software diferencia tres fases: especificación del plan de pruebas, ejecución, y elaboración de informes.

4. Técnicas de diseño de casos de prueba

Dos enfoques principales:

1. **Caja negra:** sólo necesitamos conocer la interfaz del elemento probado, probamos si las salidas se corresponden con la entrada. Enfoque funcional
2. **Caja blanca:** conocemos las interioridades del sistema, probaremos que los componentes internos funcionan bien y encajan correctamente. Enfoque estructural

4.1. Pruebas de caja negra (funcionales)

Dado que no es posible probar todo, esta sección recoge varias estrategias de creación de pruebas de caja negra que aumentan las posibilidades de descubrir errores:

1. **Clases de equivalencia:** Identificamos los rangos en las restricciones de entrada o los tipos de entrada que producen un tipo de salida. Debemos **probar un valor dentro de cada rango o tipo**.
2. **Análisis de Valores Límite (AVL):** Consiste en probar los errores que están en los límites de las clases de equivalencia.
3. **Conjetura de errores:** Se prueban errores que los programadores cometen con frecuencia, bien partiendo de una lista de errores frecuentes o de la intuición del creador de la prueba.
4. **Pruebas aleatorias:** Generamos permutaciones aleatorias de los valores de entrada y comprobamos si la salida es correcta.

4.2. Pruebas de caja blanca (estructurales)

El diseño de casos tiene que basarse en la **elección de caminos importantes que ofrezcan una seguridad aceptable de descubrir un defecto**, y para ello se utilizan los criterios de cobertura lógica. No requieren el uso de representaciones gráficas, pero se suelen utilizar grafos de flujo.

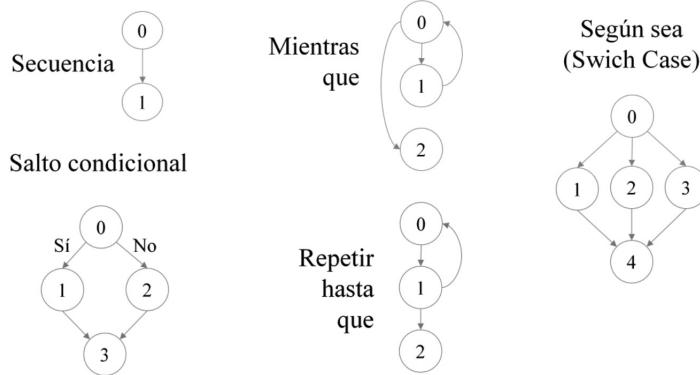


Figura 24: Grafos básicos.

4.2.1. Criterios de cobertura lógica

- Cobertura de sentencias:** Cada sentencia o instrucción del programa se ejecuta al menos una vez.
- Cobertura de decisiones:** Cada decisión tiene, al menos una vez, un resultado verdadero y uno falso. En general, la cobertura de decisiones asegura la cobertura de sentencia.
- Cobertura de condiciones:** Cada condición de cada decisión adopta, al menos una vez, un resultado verdadero y otro falso. No garantiza la cobertura de decisiones.
- Criterio de decisión/condición:** Exigir el criterio de cobertura de condiciones obligando a que se cumpla también el criterio de decisiones.
- Criterio de condición múltiple:** Descompone cada decisión múltiple en una secuencia de decisiones unicondicionales² y luego se exige que cada combinación posible de resultados de cada condición se ejecute al menos una vez.
- Cobertura de caminos:** Cada uno de los posibles caminos del grafo de caminos se ejecuta al menos una vez. Según las veces que recorramos el bucle tenemos diferentes tipos:
 - Sin entrar en su interior.
 - Ejecutándolo una vez.
 - Ejecutándolo dos veces.
 - Ejecutándolo n veces.

²una decisión es un conjunto de condiciones, por ejemplo en `a != null && a.length > 3` es una decisión que se descompone en las condiciones `a != null` y `a.length > 3`

4.2.2. Prueba de bucles

La prueba de bucles es la **técnica de prueba de caja blanca que se centra exclusivamente en la validez de las construcciones de bucles**. Se pueden definir 4 clases de bucles:

1. **Bucles simples:** Se les debe aplicar el siguiente conjunto de pruebas:

- Pasarlo por alto.
- Pasar una vez por el bucle.
- Pasar 2 veces por el bucle.
- Hacer m pasos con $m < n$ tal que n es el mayor número de pasos permitidos por el bucle.
- Hacer $n - 1$ y $n + 1$ pasos

2. **Bucles anidados:**

- Comenzar por el bucle más interior con los otros en sus valores mínimos.
- Llevar a cabo la prueba de bucle simple al más interior.
- Progresar hacia fuera, manteniendo el resto de bucles externos en sus valores mínimos y los demás bucles anidados en sus valores típicos.
- Continuar hasta probar todos los bucles.

3. **Bucles concatenados:** Se pueden probar mediante el enfoque para bucles simples siempre que cada bucle sea independiente del resto. Si son dependientes se usa la técnica para bucles anidados.

4. **Bucles no estructurados:** Se deben rediseñar para que se ajusten a las construcciones de la programación estructurada.

4.2.3. Utilización de la complejidad ciclomática de McCabe

La métrica de McCabe es un indicador del número de caminos independientes que existen en un grafo.

El propio McCabe definió como un buen criterio de prueba la consecución de la ejecución de un conjunto de caminos independientes, lo que implica probar un número de caminos igual al de la métrica. La métrica de McCabe asegura la cobertura de sentencia y sería equivalente a la cobertura de decisiones.

Un camino es independiente de otros si incorpora un arco que los demás no incluyen. La métrica de McCabe coincide con el número máximo de caminos independientes que puede haber en un grafo. Formas de calcular la complejidad ciclomática de McCabe:

1. $V(G) = a - n + 2$, siendo a el número de arcos y n , el de nodos.
2. $V(G) = r$, siendo r el número de regiones cerradas del grafo. Si el programa tiene un nodo de inicio y otro de final, la región externa se suma como región cerrada.
3. $V(G) = c + 1$, siendo c el número de nodos de condición.

Para ayudar a la elección de los caminos de prueba, McCabe propone el **método del camino básico, consistente en realizar variaciones sobre la elección de un primer camino de prueba típico**. A partir de estos caminos, se analiza el código para conocer los datos de

entrada necesarios, y se consulta la especificación para conocer la salida teóricamente correcta.

Puede suceder que las condiciones necesarias para que la ejecución pase por un determinado camino no se puedan satisfacer de ninguna manera (**camino imposible**), en cuyo caso deberemos sustituir ese camino por otro posible que permita satisfacer igualmente el criterio de prueba de McCabe.

$V(G)$ marca un límite mínimo de número de casos de prueba para un programa. Si $V(G)$ es mayor que 10, la probabilidad de encontrar defectos aumenta, salvo que sea debido a sentencias *switch case* o similares. En estos casos se debe replantear el diseño modular obtenido.

5. Enfoque práctico recomendado para el diseño de casos

1. Si la especificación contiene combinaciones de condiciones de entrada, comenzar formando sus grafos causa–efecto.
2. Usar el análisis de valores límite (AVL) para añadir casos de prueba: Elegir límites para dar valores a las causas en los casos generados, asumiendo que cada caso es una clase de equivalencia.
3. Identificar las clases válidas y no válidas de equivalencia para la entrada y la salida y añadir los casos no incluidos anteriormente.
4. Utilizar conjetura de errores para añadir nuevos casos referidos a valores especiales.
5. Ejecutar los casos generados hasta el momento y analizar la cobertura obtenida.
6. Examinar la lógica del programa para añadir los casos precisos para cubrir el criterio de cobertura elegido si no ha sido satisfecho en el punto anterior.

Esto es aplicable tanto a pruebas de caja blanca como de caja negra ya que:

- Los errores lógicos y las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa.
- Se suele creer que un determinado camino tiene pocas probabilidades de ejecutarse cuando se ejecuta regularmente.
- Los errores tipográficos son aleatorios.
- La probabilidad e importancia de un trozo de código se suele calcular de modo subjetivo.
- Una prueba exhaustiva de caja blanca no asegura la detección de los defectos de su diseño.

5.1. Depuración

La depuración es el **proceso de localizar, analizar y corregir los defectos que se sospecha que contiene el software**. Suele ser la consecuencia de una prueba con éxito. Tras corregir el defecto, se efectuarán nuevas pruebas que comprueben si se ha eliminado dicho problema.

5.2. Análisis de errores a análisis causal

El análisis causal proporciona información sobre la naturaleza de los defectos para que así el personal pueda prevenirlos en el futuro. Se recoge la siguiente información:

- Cuándo se cometió.
- Quién lo cometió.
- Qué se hizo mal.
- Cómo se podría haber prevenido.
- Por qué no se detectó antes.
- Cómo se podría haber detectado antes.
- Cómo se encontró el error.

No debe usarse para evaluar al personal.

6. Estrategia de aplicación de las pruebas

1. **Prueba de unidad:** Se centra en ejercitarse la lógica del módulo y los distintos aspectos de la especificación de las funciones que debe realizar el módulo.
2. **Prueba de integración:** Debe tener en cuenta los mecanismos de agrupación de módulos fijados en la estructura del programa, así como las interfaces entre componentes.
3. **Prueba del sistema:** Debe comprobar el cumplimiento de los objetivos indicados para el sistema.
4. **Prueba de aceptación:** El usuario verifica en su entorno si acepta el producto.

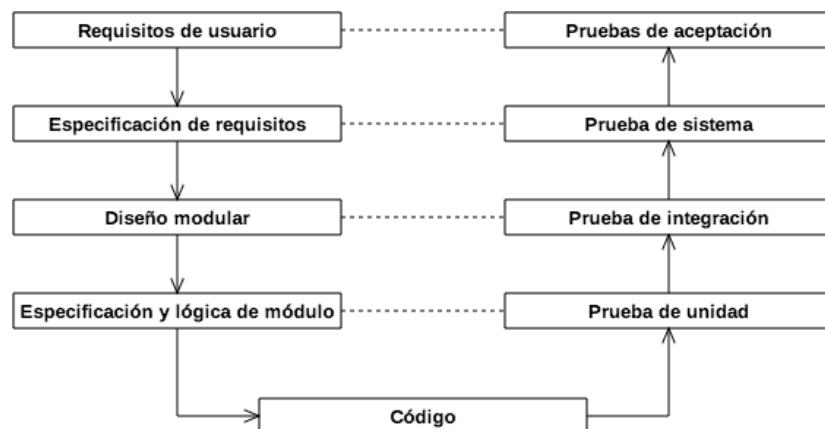


Figura 25: Las diferentes estrategias de prueba y su relación con las otras fases del construcción del software

7. Pruebas en desarrollos orientados a objetos

7.1. Desde el punto de vista de diseño de casos de pruebas

- **Las técnicas de caja negra:** Totalmente válidas.
 - Análisis de valores límite, tratamiento de combinaciones de entrada y conjectura de errores.
 - Debemos diseñar las pruebas basándonos en datos y eventos de los escenarios de los casos de uso y en flujos alternativos, tratamientos de error y excepciones.
- **Las técnicas de caja blanca:** Disminuyen sus posibilidades de aplicación. Quedan confinadas a las instrucciones de cada método de cada clase.

7.2. Desde el punto de vista del nivel

Los cambios más radicales aparecen en las pruebas de integración, cuando nos fijamos en la interacción entre objetos de distintas clases. El enfoque habitual consiste en ir probando hilos de clases que colaboran para una función o servicio del sistema.

7.3. Cuestiones a tener en cuenta

Ciertas cuestiones a tener en cuenta a la hora de probar software orientado a objetos son:

- **Herencia:** probar un método de la clase padre no garantiza su funcionamiento en clases hijas.
- **Polimorfismo:** un único método puede tener implementaciones diferentes en función de la clase en la que se use.
- Es conveniente recordar que la propia programación o diseño orientado a objetos hace menos probables ciertos tipos de error, más probables otros, y provoca la aparición de nuevos tipos de defectos.

8. Ejemplo práctico de cálculo de la complejidad ciclomática de un método

Dado el método `public String porcentajeVentasMes();` tal que:

```
public String porcentajeVentasMes() {  
    [...]  
    //recopilamos precios por dia  
    for(int i = 0 ; i < ventas.size() ; i++ ){  
        Venta v=ventas.get(i);  
        Calendar cal = Calendar.getInstance();  
        targetCalendar.setTime(cFecha.toDate(v.getFecha()));  
        precios[cal.get(Calendar.DAY_OF_MONTH)]+=v.getPrecioUnidad();  
        total+=v.getPrecioUnidad();  
    }  
  
    //calcula los porcentajes  
    for(int i = 0 ; i < columnas; i++ ){  
        if(precios[i]>0){  
            porcentajeVentas+="Día "+(i+1)+": "+  
                BigDecimal.valueOf(precios[i]/total*100)  
                    .setScale(2, RoundingMode.HALF_UP)  
                    .doubleValue()+"%\n";  
        }  
    }  
    return porcentajeVentas;  
}  
}
```

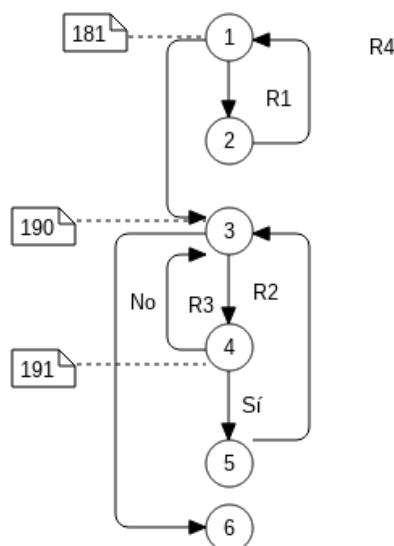


Figura 26: Grafo de flujo del método.

Caminos básicos obtenidos

1. 1-3-6
2. 1-2-1-3-6
3. 1-3-4-3-6
4. 1-2-3-4-5-3-6

Caminos independientes: 3

1. $1-(3-4)^{x31}-3-6$
 - No puede existir en el sistema ninguna venta realizada en el mes actual.
 - La variable **columnas** sólo podrá poseer los valores 28, 29, 30 ó 31, dependiendo del mes en el que es ejecutada la función. Es por esto que no se puede llevar a cabo ningún camino que contenga 1-3-6.
2. $1-2-(3-4)^{x31}-3-6$
 - Para seguir este camino independiente, deberá existir una venta realizada en el mes actual, o más si se desea repetir más veces el segmento 1-2.
 - Ningún artículo podrá tener un precio superior a 0 para poder seguir este camino estrictamente.
3. $1-(3-4)^{x31}-3-6$
 - Equivalente al camino 1.
4. $1-2-(3-(4||(4-5)))^{x31}-3-6$
 - Para seguir este camino independiente, deberá existir una venta realizada en el mes actual con un precio por unidad superior a 0 o más si se desea repetir más veces el segmento 3-4-5.

Tema VII

Preguntas de examen

1. Resolución de preguntas cortas

Las preguntas cortas consistirán en refutar sentencias falsas en base a vuestros conocimientos de la asignatura. En ningún caso se tratará de contar todo lo que sabéis sobre el punto en cuestión de que trate la frase sino sólo de rebatir la falacia con argumentos objetivos y precisos. Las respuestas de más de 10 líneas no serán evaluadas, ya que entiendo que siempre debería poder responderse como máximo en 5 o 6.

1. Las normas de procesos para la construcción del software sólo describen procesos que tienen por objetivo el desarrollo de código ejecutable.

Todas las normas vistas describen procesos incluidos en todo el ciclo de vida del software. Las normas ISO 12207 e ISO/IEC 15504 incluyen entre sus procesos principales los de adquisición y suministro (gestionan la compra-venta); también incluyen los procesos de soporte (describen tareas de apoyo, no de construcción) y los de la organización, que ni siquiera tienen que ver con el desarrollo de un proyecto software concreto. De hecho la codificación se tratar sólo como una actividad del proceso de desarrollo en estas normas.

También se puede hacer el mismo razonamiento con la norma IEEE 1074 que nos proporciona 4 secciones lógicas de las que sólo una tiene que ver con el desarrollo. Las otras tres tienen que ver con la selección del ciclo de vida, la gestión del proyecto y los procesos integrales, vinculados a asegurar la terminación y calidad de los procesos.

2. Sólo el SEI americano ha mostrado interés en la generación de normas para la Ingeniería del Software debido a su carácter académico.

Como se ha discutido indirectamente en la pregunta anterior, las normas del IEEE y la ISO/IEC acreditan la preocupación de otras instituciones, independientes del SEI, por la formalización de estas normas. Además estos estándares, como se demuestra en la explicación del CMMI, están orientados a su aplicación en las empresas a las que deben proporcionar un marco válido que evite el desarrollo ad hoc de aplicaciones.

3. Pasar del modelo en cascada al modelo incremental no solucionó ningún problema de ingeniería pero permitió ganar más dinero a los programadores.

El salto al modelo incremental evita la concepción monolítica del software que da el modelo en cascada. Propone un modelo iterativo que permite la evolución en sus requisitos, la mejora de la comunicación con el cliente y posibilita reducir el coste de los errores.

4. Ante un problema ya resuelto el planteamiento siempre debe ser reprogramar otra solución pues seguro que será más eficiente.

Si el problema está resuelto implica que el software ya existe. La conclusión de los análisis de coste realizados en prácticas y la discusión sobre el reuso hecha en clase, es que será más barato (y por tanto eficiente) comprarlo que reconstruirlo. Sólo debemos plantearnos construir cuando la funcionalidad que necesitamos no está recogida por el software disponible o se precisa una modificación significativa para adaptarlo. Evidentemente no se considera que la funcionalidad esté en la aplicación si ésta no la realiza con la calidad esperada.

5. Los únicos riesgos posibles de un proyecto son económicos y técnicos

Si bien estos riesgos importantes, existen otro tipo de riesgos que pueden resultar cruciales a la hora de desarrollar un proyecto. Es el caso de los riesgos geopolíticos, desastres naturales, realizar un mal análisis de requisitos, realizar una mala planificación, o el espionaje industrial.

6. La única cualidad importante del analista es ser persuasivo para convencer al cliente de que el software que le desarrollamos era el que necesitaba.

En tanto que la persuasión es cualidad importante y forma parte de una de las muchas dotes comunicativas que debe tener un analista, no es suficiente.

Un analista debe ser capaz de comunicar y de extraer información de los clientes.

Además, valiéndose de sus conocimientos técnicos y experiencia en el campo de la ingeniería del software, ser capaz de traducir ideas vagas de necesidades de software en un conjunto concreto de funciones y restricciones.

7. La especificación es un documento escrito en lenguaje natural que será tanto mejor cuanto más largo y enrevesado sea.

Una especificación es un documento descriptivo, que debe reflejar de forma clara y sencilla el aspecto del software que esté tratando.

Por lo tanto, se deberá evitar el uso del lenguaje natural (puesto que puede resultar vago y ambiguo) y, en su defecto, hacer uso de modelos, diagramas y herramientas que reflejen objetivamente el software.

Por otro lado, los documentos generados deben estar estructurados en particiones, permitiéndonos detectar cualquier redundancia y facilitando su modificación y lectura.

8. Un buen modelo es aquel que, sin ayuda de otros, representa todos los aspectos de un sistema.

Debido a la complejidad de los sistemas informáticos, no es posible elaborar un modelo comprensible para un ingeniero que represente todos los aspectos de un sistema de forma clara y concisa.

Es por ello que debemos optar por una estrategia fragmentada, representando el sistema mediante múltiples modelos que colaboren entre sí y nos permitan reflejar con precisión cada uno de los aspectos del sistema.

9. Cada DFD y DFC tienen los mismos procesos y almacenes pero jamás se relacionan ya que uno representa la función y el otro el comportamiento.

Los DFD y DFC están íntimamente relacionados pues el DFC es la representación del flujo de control de los procesos definidos en el DFD que representa. Los procesos y almacenes sí son los mismos en los dos diagramas. El DFC representa el comportamiento de los procesos definidos en el DFD.

10. Los casos de prueba son las entradas necesarias al sistema para que se ejecute un camino de prueba que garantice la cobertura de sentencia.

1. *La motivación de hacer una prueba es encontrar un fallo, a partir de ahí hay múltiples metodologías para realizar una prueba, existen pruebas de caja negra y blanca y dentro de las de caja blanca estaría un tipo cuyo objetivo es la cobertura de sentencia.*

2. De acuerdo con el IEEE 829, en la especificación de un caso de prueba se definen tanto las entradas como las salidas (datos de prueba) usados en la ejecución de los casos.

11. ¿Cuáles son los puntos de vista desde los que podemos modelar el software?
Sitúa en una tabla que tenga los puntos de vista tanto en las filas como en las columnas, y las metodologías o técnicas que conozcas para modelar

	Información	Función	Tiempo
Información	D. entidad-relación D. de estructura de datos Matriz entidad/entidad Diagramas de clases		
Función	D. de flujo de datos Matriz función/entidad Diagrama de clases D. de colaboración	D. de flujo de datos D. de casos de uso D. de estructura de datos Tarjetas CRC D. de componentes D. de despliegue D. de actividad	
Tiempo	H ^a de vida de la entidad D. de estados D. de secuencia	Redes de Petri D. de estados D. de secuencia D. de actividad	D. de flujo de control D. de estados

Tabla 9: Métodos de modelado según la dimensión del sistema que modelan

12. ¿Qué es un stakeholder? Pon dos ejemplos en un contexto determinado.

Personal involucrado en el proyecto, incluyendo a los usuarios finales, los ingenieros trabajando en el proyecto, el administrador de negocio y los expertos del dominio del sistema. Podemos distinguir entre involucrados, afectados e implicados.

En el desarrollo de una aplicación para ENSO, el profesor Taboada se ve involucrado, mientras que el alumno se ve comprometido.

13. Las líneas base no están relacionada con los elementos de configuración del software.

La línea base representa la configuración vigente y aprobada y sólo puede ser modificada a través de un procedimiento formal de cambios.

Está conformada por un conjunto de elementos de configuración, acabados y formalmente aprobados, designados y fijados en un momento específico del ciclo de vida.

14. La documentación de las pruebas no es un elemento de configuración.

Cualquier producto de trabajo creado como parte del proceso de ingeniería del software, tanto final como intermedio y tanto entregable al cliente como interno del proyecto, cuyo cambio puede resultar crítico para el buen desarrollo del proyecto y susceptible de ser gestionado es un elemento de configuración.

Esto incluye la documentación de las pruebas.

15. Define lo que es un plan de gestión de la configuración.

Documento que servirá de referencia para llevar a cabo el proceso de gestión de configuración, recogiendo la identificación de elementos de configuración, el control de la configuración, el registro del estado de la configuración, las auditorías de configuración y la gestión del despliegue.

16. El diagrama de flujo de datos (DFD) representa el comportamiento del sistema desde un punto de vista funcional.

Los DFD describen qué funciones son las que realiza el sistema, qué interacciones se producen entre estas funciones así como las transformaciones de datos.

Sin embargo, no describen cómo se comporta el sistema ni indican en ningún momento cuándo realiza una función ni la secuencia que se sigue.

17. Los procesos de soporte solo sirven para entorpecer el desarrollo software.

Los procesos de soporte sirven de apoyo a los procesos principales y tienen como objetivo evitar la aparición de problemas crónicos asociados al desarrollo del software.

Mediante procesos como la documentación, el control del cambio, la auditoría o el aseguramiento de la calidad, se puede mejorar la calidad de un producto, facilitar su mantenimiento y reutilización y por ende, mejorar la satisfacción del cliente.

18. Los requisitos de dominio se refieren al entorno y no tienen que ver con los requisitos funcionales.

Al contrario, realmente no se trata de un conjunto disjunto, pues los requisitos de dominio pueden ser tanto funcionales como no funcionales. Ejemplos:

- *Usar el formato DICOM en medicina es a la vez un requisito no funcional y de entorno.*
- *Las radiografías generadas por el sistema deben poder ser visualizadas en varias pantallas a la vez, mostrando al sujeto desde varias perspectivas.*

19. El único momento en el que se hace uso un DFC es durante el análisis del sistema.

El DFC es un diagrama permite la representación del sistema en múltiples niveles de abstracción y por lo tanto se utiliza cuando resulte útil para el desarrollo de software.

Adicionalmente, no en todos los ciclos de vida se realizará el diseño en la fase de análisis del sistema ya que, por ejemplo en cascada es más probable que su realización estuviese ligada a la fase de diseño.

Finalmente, el DFC se puede elaborar además en la fase de análisis de software.

20. La verificación y la validación no tienen nada que ver.

Si bien su enfoque es distinto, tanto la verificación como la validación son procesos de evaluación de un sistema o de uno de sus componentes.

La verificación determina si los productos de una fase dada satisfacen las condiciones impuestas al principio de dicha fase.

La validación determina si satisface los requisitos especificados.

21. La realización de un proceso es completamente rígida, cuando un proceso está institucionalizado nunca se deja de hacer a pesar del estrés del proyecto.

Una característica de los procesos es que es un conjunto de actividades adaptables a las necesidades del que lo utiliza.

Por lo tanto, puede llegar a suceder que en una situación de estrés y dadas las necesidades del proyecto que no se pueda completar uno de los procesos planificados y se prioricen otros.

22. La gestión de riesgos solo se realiza en una determinada fase del proceso de análisis de riesgos.

La gestión de riesgos es un proceso de supervisión cuyo objetivo es la detección precoz de riesgos. Es por ello lógico que sea un proceso en continua aplicación durante el desarrollo del proyecto como un proceso de soporte.

23. La programación extrema está englobada dentro del modelado ágil.

No se debe confundir el modelado ágil con las metodologías ágiles.

La programación extrema forma parte de las metodologías o desarrollos ágiles, que renuncian a utilizar modelos perfectos y centran sus esfuerzos en presentar incrementos software ejecutable.

Dentro de las metodologías ágiles podemos mencionar el modelado ágil (que es una colección de buenas prácticas) o la programación extrema.

24. Es necesaria una reunión TFEA con el fin de realizar un análisis de requisitos adecuado.

El TFEA es un conjunto de técnicas que facilitan el análisis de requisitos, y pueden ser utilizadas como una metodología para la generación de un buen análisis de requisitos.

Sin embargo, no es necesario seguir estas técnicas y menos aun realizar una reunión ya que se pueden obtener los requisitos realizando estudios de documentación, cuestionarios o elaborando prototipos.

25. El DFD de nivel 0 recibe el nombre de Diagrama 0.

- *El diagrama de flujo de datos de nivel 0 recibe el nombre de Diagrama de Contexto y representa al sistema como un único proceso, el proceso 0.*
- *El diagrama de nivel 1 se denomina Diagrama 0 debido a que descompone el proceso 0 en sus funciones principales.*

26. Los errores en el software sólo están causados por el programador.

Entendiendo error en las acepciones aplicables a este contexto: resultado incorrecto y defecto. Pueden causarse además de por un fallo de programación por un fallo de análisis, por ejemplo, si no se describió correctamente la salida requerida a la entrada, o el requisito que debe satisfacer la funcionalidad programada. También puede producirse por un fallo en el entorno de ejecución o en las herramientas o librerías de terceros utilizadas (CORBA).

27. La calidad del software solo viene determinada por el costo y el tiempo que se le dedica al proyecto.

El alcance de un proyecto es la suma de todos los productos y sus requisitos o características es otro factor que debemos considerar. El coste, el tiempo y el alcance son limitantes de la

calidad, lo cual no es lo mismo que determinantes.

Para un mismo tiempo, coste y alcance la aplicación de procesos de ingeniería probados, bien definidos y siguiendo los estándares repercutirá muy positivamente en la calidad. También podríamos hablar de la capacidad del personal involucrado, así como la experiencia y la aplicación de buenas prácticas a nivel de programación.

28. El nivel de madurez máximo que una empresa puede alcanzar en un proceso es el de optimización, para ello debe alcanzar los niveles previos y cumplir las metas necesarias dentro de los procesos relacionados.

Según el CMMI, se puede medir la madurez con 6 niveles, que son los siguientes: Nivel 0 N/A, Nivel 1 Inicial, Nivel 2 Gestión, Nivel 3 Definido, Nivel 4 cuantitativamente gestionado y Nivel 5: en optimización, sin embargo, no es necesario cumplir las metas de los procesos relacionados

29. La mejor forma de optimizar un proceso en CMMI es realizar una institucionalización del mismo.

El CMMI es un modelo que centra su evaluación en las áreas del proceso y puede ser aplicado de dos maneras:

- *Siguiendo un modelo continuo, que define el nivel de capacidad de cada uno de los procesos de la empresa de manera independiente.*
- *Siguiendo un modelo discreto, que define el nivel de madurez global (al que se refiere el enunciado).*

Aunque siguiendo puntos de enfoque distintos, con ambos modelos se puede alcanzar la optimización de un proceso.

30. Los procesos de soporte se realizan siempre de manera iterativa.

Los procesos de soporte sirven de apoyo al resto de procesos y no necesariamente todos ellos serán aplicados de forma iterativa sino puntual.

Esto puede suceder, por ejemplo, con el proceso de validación, que en algunos ciclos de vida puede llegar a aplicarse únicamente en las fases finales del proyecto.

31. La curva de fallos del software con respecto al tiempo aumenta a causa de los programadores y analistas que no saben arreglar software.

El software sufre cambios durante su vida, debidos al mantenimiento del mismo. Al introducir cambios en el software puede darse el caso de introducir también errores (ya sean de análisis, diseño o codificación), pero estos errores pueden corregirse. Lo que verdaderamente provoca el deterioro del software es el hecho de que con cada cambio introducido el producto se aleja cada vez más de la especificación inicial del mismo.

32. Dentro de procesos principales de la IEEE está el proceso de documentación que incluye la realización de documentación operativa.

El proceso de desarrollo de la documentación se encuentra dentro de los procesos integrales, ya que es necesario para la realización de los procesos principales, pues en cada uno de los mismos se genera documentación que ha de ser gestionada y mantenida a lo largo del proyecto.

33. Cuando tenemos claros los requisitos del sistema lo mejor es siempre usar cascada.

El ciclo de vida en cascada es un ciclo de vida cuya aplicación sólo se aconseja en el caso de tener claros los requisitos del sistema, ya que en caso de no tenerlos podría ser necesario repetir el proyecto desde la fase de análisis. Esto no implica de todas formas que sea el mejor ciclo de vida para aplicar a un proyecto, ya que se deben tener en cuenta muchos más factores. Por ejemplo aunque tengamos claros los requisitos del sistema siempre es posible que se trate de un sistema cuyos riesgos deben ser tomados en cuenta, por lo que un ciclo de vida en espiral resultaría más adecuado. También puede darse el caso de que el cliente quiera implicarse en el proceso y valore muy positivamente la entrega frecuente de incrementos.

34. La tabla de activación de procesos establece cuando un proceso se activa dada unas determinadas señales de entrada/salida.

La tabla de activación de procesos representa tanto sucesos como procesos y salidas de los mismos. Lo que se termina por reflejar en la tabla es qué procesos se verán ejecutados, bajo qué circunstancias y qué salidas tendrán los mismos.

35. En la construcción de prototipos, es poco común que se modifiquen los requisitos.

La construcción de prototipos destaca sobre otros ciclos de vida cuando se trata de proyectos de alto nivel de incertidumbre, donde los requisitos no están nada claros y son especialmente volátiles.

El feedback periódico obtenido del cliente a través de los prototipos, es usado para ir determinando y refinando los requisitos del sistema de forma fiable.

36. Los errores lógicos y las suposiciones incorrectas son directamente proporcionales a la probabilidad de que se ejecute un camino del programa

Si existe un error, será poco probable que el camino que lo incluya sea ejecutado. Por lo tanto, los errores lógicos y las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa.

37. Las técnicas matriciales se utilizan principalmente para ayudar a verificar la compleción de un sistema

El principal uso de las técnicas matriciales es la verificación de la consistencia entre los componentes de distintos modelos de un sistema pudiendo hacerse desde un enfoque entidad/función, entidad/entityad o evento/entityad.

38. Un PSPEC especifica bajo qué condiciones se activan procesos

El propósito de un PSEC (especificación del proceso) es describir textualmente los detalles de un proceso, indicando cómo es el proceso de transformación de una entrada en una salida. Bajo qué condiciones se emiten las señales de control que activan los procesos depende del CSPEC (especificación del control) y pueden ser especificados de manera secuencial o combinacional.

39. En el modelo lógico de un sistema debemos reflejar la configuración de los backups o la seguridad

El modelo lógico, también denominado esencial, recoge qué debe hacer el sistema para satisfacer los requisitos del usuario y, por lo tanto, en él no figura nada acerca de su implementación. Por otro lado, el modelo de implementación si muestra estos detalles técnicos.

40. Tras la realización de las pruebas, no hemos detectado defectos por lo que nuestro software ha sido construido correctamente

En las pruebas de software, la detección de un defecto constituye el éxito de la prueba, permitiéndonos la rectificación del software, lo que conlleva una mejora de la calidad.

Por lo tanto, no detectar defectos en el software es probablemente una señal de que no se han realizado pruebas suficientes o no se ha realizado un buen diseño de las mismas.

2. Preguntas largas

- 2.1. Enumera y comenta las principales características del software asociadas a su naturaleza “inmaterial”

Tema 1 Apartado 1

- 2.2. Enumera y comenta los principales problemas crónicos del software

Tema 1 Apartado 4

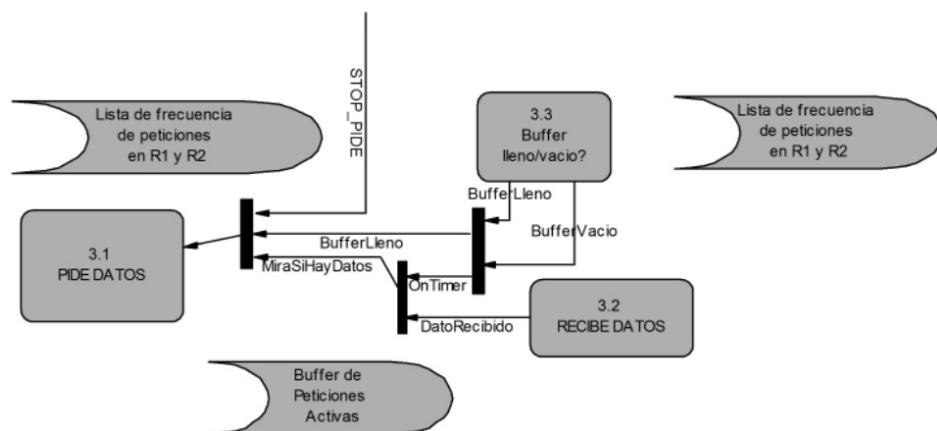
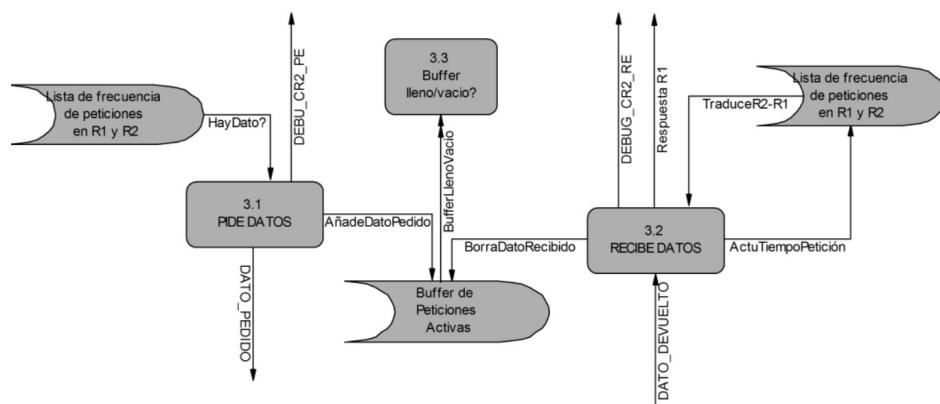
- 2.3. Norma 12207-1: Procesos de soporte. ¿Cuál es su diferencia fundamental con la 1074? ¿Qué aporta la 15504-2 que no contempla la 12207 en los procesos de la organización?

Tema 2 Apartado 2.1, 2.2, 2.3

- 2.4. Describe en qué consiste el ciclo de vida en espiral, cuáles son sus ventajas e inconvenientes. Describe la metodología para el análisis de riesgos según Sommerville

Tema 2 Apartado 5

- 2.5. Para cada uno de los siguientes diagramas, señala los elementos que contienen y describe su función en el diagrama



Tema 5 Apartado 2.3

- 2.6.** Comenta el proceso de elaboración de un plan de prueba seguido por el estándar IEEE 829 y los documentos asociados al proceso ¿Cuál es el estándar de ejecución de las pruebas? ¿Qué documentación implica?

Tema 6 Apartado 3

- 2.6.1.** Describe en qué consisten las pruebas funcionales, para qué sirven, en base a qué se construyen. Describe en detalle 3 técnicas de construcción de este tipo de pruebas.

Tema 6 Apartado 4.1

3. Caso práctico 2014

Desarrollo del software asociado a un programador de control del sistema de climatización por aireación de una casa, que tiene, por lo menos, una pantalla táctil de 7 pulgadas para la interacción con el usuario.

El sistema de aireación puede mezclar aire frío tomado del exterior de la vivienda, con aire caliente, calentado por una caldera de Diesel.

3.1. Detalla formalmente las siguientes funcionalidades, según el modelo empleado en las prácticas

- RF1: Establecer una temperatura de confort en modo manual
- RF2: Establecer la velocidad de ventilación del aire, de 0 a 5 posibilidades
- RF3: Establecer el modo de aireación (distribución posible del aire: la planta baja, al primer hangar, al segundo hangar, la toma de casa).

Comenzamos definimos el actor usuario, que será el único asociado a los requisitos funcionales aquí mencionados.

A partir de ahí creamos un caso de uso para cada requisito funcional.

3.1.1. Requisitos

- **ID:** RF-0001
 - **Título:** Establecer una temperatura de confort en modo manual.
 - **Descripción:** El sistema deberá permitir al usuario modificar la temperatura de una sala de manera manual a través de la pantalla táctil. Se permitirá un rango de temperaturas limitado. Se proporcionará frío o calor en función de la temperatura elegida por el usuario y la temperatura ambiente.
 - **Importancia:** Vital.
 - **Estabilidad:** Alta.
 - **Fuente:** Julián Flores González.
 - **Criterio de validación:** El sistema es capaz de regularse en función de la temperatura elegida por el usuario.
- **ID:** RF-0002
 - **Título:** Establecer velocidad de ventilación.
 - **Descripción:** El sistema deberá permitir al usuario modificar la velocidad de rotación de los ventiladores mediante el panel táctil. Se permiten las siguientes posibilidades:
 - **0:** 0 % de rotación.
 - **1:** 20 % de rotación.
 - **2:** 40 % de rotación.
 - **3:** 60 % de rotación.
 - **4:** 80 % de rotación.

- **5:** 100 % de rotación.

Donde el 100 % será la máxima capacidad de rotación y 0 significará que el ventilador está apagado.

- **Importancia:** Vital.
- **Estabilidad:** Alta.
- **Fuente:** Julián Flores González.
- **Criterio de validación:** el sistema es capaz de regular la velocidad de los ventiladores en función de lo que elija el usuario.
- **ID:** RF-0003
- **Título:** Establecer modo de aireación.
- **Descripción:** El sistema deberá permitir al usuario determinar que plantas de la casa se podrán ventilar con el sistema de climatización. Existen cuatro posibilidades únicamente:
 - Planta baja: el sistema de climatización solo opera en la planta designada como 0.
 - Planta 1: el sistema de climatización solo opera en la planta designada como 1.
 - Planta 2: el sistema de climatización solo opera en la planta designada como 2.
 - Planta 3: el sistema de climatización solo opera en la planta designada como 3.
- **Importancia:** Vital.
- **Estabilidad:** Alta.
- **Fuente:** Julián Flores González.
- **Criterio de validación:** el sistema es capaz de climatizar las salas en función de lo elegido por el usuario.

1. Establecer temperatura de confort

- **Precondiciones:** El software debe tener un sistema climatización asociado.
- **PostCondiciones:** Se ha establecido la temperatura de confort indicada por el usuario para el sistema asociado.
- **Actores:** Usuario.
- **Descripción:**
 - a. El usuario selecciona el sistema de aire asociado sobre el que quiere realizar la acción.
 - b. El sistema le muestra al usuario las acciones que puede realizar con el sistema de aire y el valor actual de sus parámetros, entre las acciones está la opción de modificar manualmente la temperatura de confort. El sistema no deja que este utilice rangos incorrectos.
 - c. El usuario cambia la temperatura.
 - d. El sistema en tiempo real modifica la temperatura de confort actual asociada a ese dispositivo.

2. Establecer la velocidad de ventilación

- **Precondiciones:** El software debe tener un sistema climatización asociado.
- **Postcondiciones:** Se ha establecido el modo de aireación indicado por el usuario.
- **Actores:** Usuario.
- **Descripción:**
 - a. El usuario selecciona el sistema de aire asociado sobre el que quiere realizar la acción.
 - b. El sistema le muestra al usuario los parámetros actuales y las acciones que puede realizar con el sistema de aire, entre ellas está la opción de modificar la velocidad de ventilación del aire entre los niveles 0 y 5.
 - c. El usuario cambia el nivel de ventilación.
 - d. El sistema en tiempo real modifica el nivel de ventilación actual asociada a ese dispositivo.

3. Establecer el modo de aireación

- **Precondiciones:** El software debe tener un sistema climatización asociado.
- **Postcondiciones:** Se ha establecido el modo de aireación indicada por el usuario para el sistema asociado.
- **Actores:** Usuario.
- **Descripción:**
 - a. El usuario selecciona el sistema de aire asociado sobre el que quiere realizar la acción.
 - b. El sistema le muestra al usuario los parámetros actuales del sistema y la acciones que puede realizar sobre esos parámetros, entre estas acciones está la de establecer el modo de aireación entre los modos (planta baja, primer hangar, segundo hangar, toma de casa).
 - c. El usuario cambia el modo de aireación.
 - d. El sistema en tiempo real modifica el modo de aireación asociada a ese dispositivo.

- 3.2. Diseña el Modelo de Contexto y el DFD de primer nivel de forma que englobe las funcionalidades del apartado anterior.

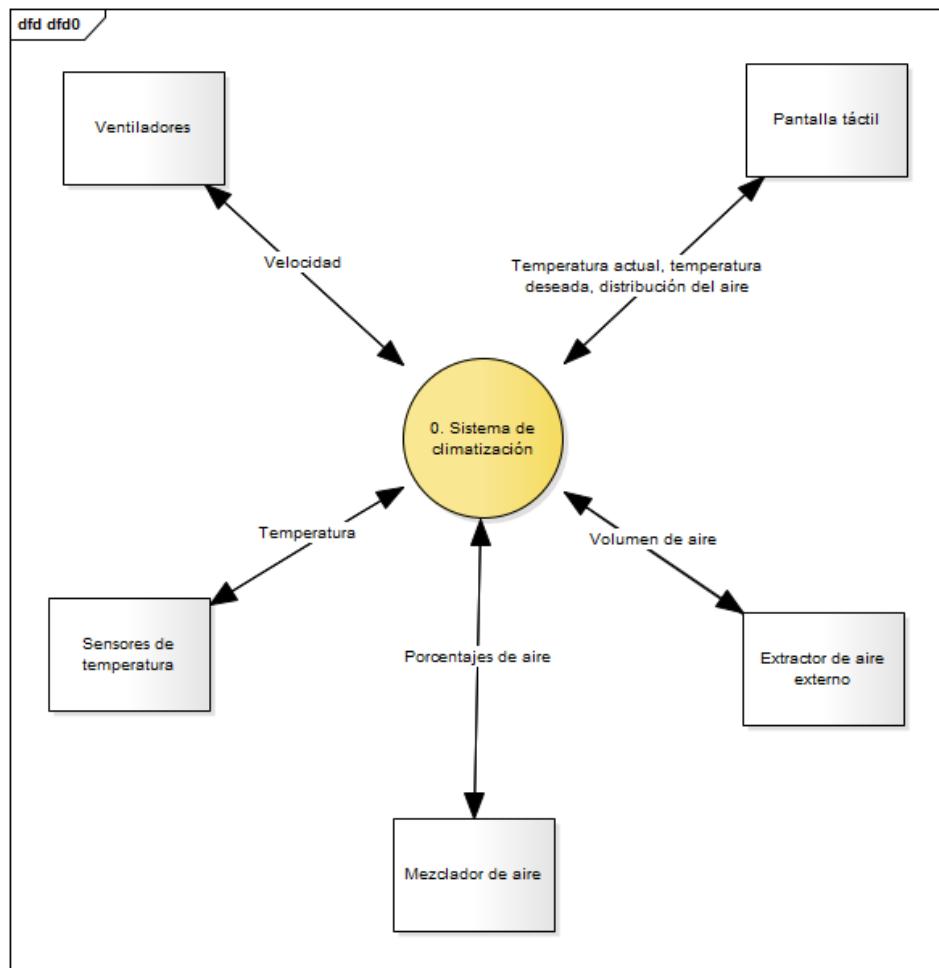


Figura 27: Diagrama de contexto

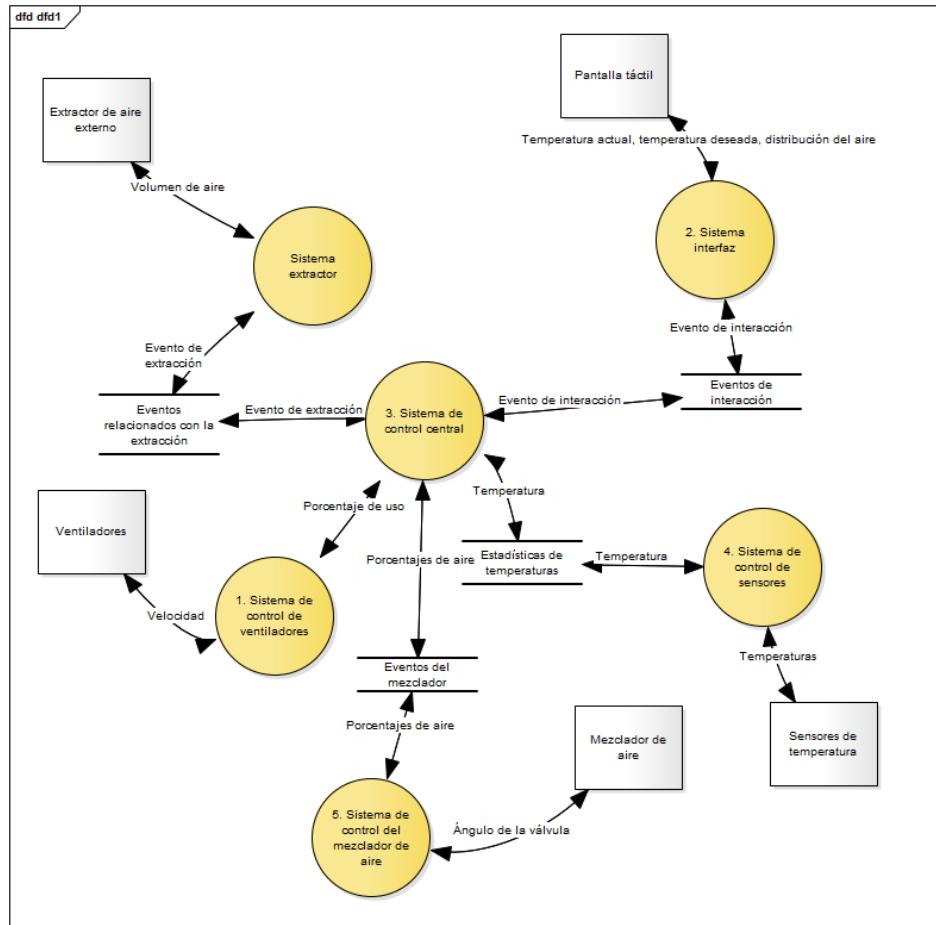


Figura 28: DFD de primer nivel

3.3. Para la funcionalidad RF1, haz un DFD de nivel 2, con su DFC asociado

3.4. En el siguiente contexto ¿cuál sería el modelo de ciclo de vida más adecuado? justifica tu respuesta.

“El software va a ser desarrollado por un empresa de amplia experiencia en el desarrollo de software empotrado, pero nunca para el sector de vivienda familiar. Los requisitos están perfectamente definidos, pero la competencia está muy activa, lo que puede resultar en una modificación de requisitos para adoptarlo a las nuevas funcionalidades. Las pantallas táctiles están empezando a ser comercializadas por lo que no se sabe muy bien como es el paradigma de iteración con el usuario”

Una posible solución: El ciclo de vida recomendado es el “ciclo de vida en espiral”. Los motivos son los siguientes:

- Los requisitos están bien definidos y los únicos cambios posibles parecen ser la adaptación de nuevas funcionalidades. Como se suceden en varias iteraciones es trivial incorporar esto.
- No se sabe como es el paradigma de interacción con el usuario por lo que interesa crear prototipos para ello. Este ciclo de vida lo permite (pues en cada iteración hay que tener un producto ejecutable). Además, hay que tener en cuenta que la inexperiencia

en el sector familiar refuerza este argumento pues varias iteraciones permiten refinar y mejorar el producto.

3.5. Establece el proceso de admisión de riesgos, según la metodología de Sommerville, que contempla solo 2 riesgos, para el desarrollo de software en el siguiente contexto

“Vamos a adquirir pantallas táctiles a una empresa china que opera por internet. Tenemos un simulador para el desarrollo de software, que emula el entorno de la vivienda pero no es muy estable, y además no tenemos acceso continuado al prototipo de vivienda en la que se va a verificar y validar el software.

1. Retraso en la realización de pruebas

- **Descripción:** Problemas de estabilidad en el software de pruebas o incapacidad para probar el software en un entorno real pueden producir retrasos tanto en el desarrollo normal como en la realización de las pruebas del software.
- **Probabilidad:** Alto.
- **Impacto:** Serio.
- **Tipo de riesgo:** Del Proyecto.
- **Tratamiento:** Minimización.
- **Acción:** Toma de contacto por parte de nuestro equipo con el simulador utilizado antes de las pruebas en si del programa, para conocer que errores da este y no confundirlos con errores del software desarrollado.
- **Secuencia de seguimiento:** Semanal.
- **Indicadores de riesgo:**
 - La realización de pruebas se retrasa durante la fase de utilización de la simulación en las pruebas.
 - imposibilidad de encontrar un lugar donde hacer la verificación y validación definitivas.
 - Surgen reportes de fallos del software de simulación durante las pruebas que deberían haberse detectado durante la toma de contacto de nuestro equipo con el software de simulación.

2. Corrupción de todos los archivos del proyecto

- **Descripción:** el simulador, debido a su baja estabilidad, corrompe todos los archivos del proyecto.
- **Probabilidad antigua:** Medio.
- **Probabilidad nueva:** Medio.
- **Impacto antiguo:** Catastrófico.
- **Impacto nuevo:** Tolerable.
- **Tipo de riesgo:** Del Producto.
- **Tratamiento:** Minimización.
- **Acción:** Se establecerá un proceso de gestión de la configuración que almacenará en un control de versiones copias diarias de los archivos a utilizar por el simulador.
- **Secuencia de seguimiento:** Diaria.
- **Indicadores de riesgo:**
 - Incapacidad de acceder a los archivos del proyecto.

3.6. Pensando en un proceso de pruebas, establece las clases de equivalencia para las siguientes entradas, asumiendo que en la pantalla táctil se puedan introducir los datos por pantalla con la simulación de un teclado qwerty

1. Temperatura de confort.
2. Velocidad del ventilador.
3. Válvula que regula la mezcla aire frío/caliente, asumiendo que la regulación de la misma es por porcentaje, desde 0 (solo aire frío) hasta 100 (solo aire caliente). Cualquier otro valor en el intervalo [0–100] dará una mezcla %aire_caliente+ %aire_frío, de forma que %aire_frío=“valor” y %aire_frío=100-“valor”. Este valor no es establecido por teclado, es una variable interna de nuestro software.

Información	Tipo dato	Regla	Clase válida		Clase no válida	
			Id	Dominio	Id	Dominio
Temperatura_confort	Entero positivo	1	1	[10, 45]	2	[0, 10)
					3	(45, +inf)
		3	4	Está compuesto de dígitos	5	Contiene letras

Tabla 10: Clases de equivalencia para “Temperatura confort”

Información	Tipo dato	Regla	Clase válida		Clase no válida	
			Id	Dominio	Id	Dominio
Velocidad_ventilador	Entero positivo	2	1	[0, 5]	2	(-inf, -1]
			8	2	3	[6, +inf)
			4	3		
			5	4		
			6	5		
			7	0		

Tabla 11: Clases de equivalencia para “Velocidad del ventilador”

Información	Tipo dato	Regla	Clase válida		Clase no válida	
			Id	Dominio	Id	Dominio
Valor_mezcla	Punto flotante	1	1	[0, 100]	2	(-inf, 0)
					3	(100, +inf)

Tabla 12: Clases de equivalencia para el “valor de la mezcla”

4. Caso práctico 2014

Desarrollo del software asociado a un cajero automático de un banco genérico, llamémosle **RABENSO**.

4.1. Detalla formalmente las siguientes funcionalidades, según el modelo empleado y la plantilla que empleasteis en las prácticas

- RF1: Cambiar PIN.
 - RF2: Consultar movimientos anteriores.
 - RF3: Retirar dinero.
- **ID:** RF-0001
 - **Título:** Cambiar pin.
 - **Descripción:** el sistema deberá permitir cambiar el número de identificación personal del usuario utilizado para autorizar transacciones bancarias. Para ello utilizará una interfaz gráfica con un teclado numérico virtual en pantalla y confirmación de pin anterior. Si el número anterior no es correcto no se realizará ningún cambio de PIN.
 - **Importancia:** Vital.
 - **Estabilidad:** Alta.
 - **Fuente:** Julián Flores González.
 - **Criterio de validación:** el sistema cambia el número de identificación según las directrices anteriormente especificadas.
- **ID:** RF-0002
 - **Título:** Consultar movimientos.
 - **Descripción:** el sistema deberá mostrar todos los movimientos del usuario en una lista desplegable por pantalla. Deberá permitir filtrar los movimientos por fecha (es decir, especificar un rango de fechas para visualizar).
 - **Importancia:** Vital.
 - **Estabilidad:** Alta.
 - **Fuente:** Julián Flores González.
 - **Criterio de validación:** El sistema muestra el conjunto de movimientos correctamente.
- **ID:** RF-0003
 - **Título:** Retirar dinero.
 - **Descripción:** el sistema deberá retirar dinero de la cuenta bancaria del usuario. Para ello, se sustrae la cantidad especificada por el usuario en el teclado numérico virtual habilitado para tal propósito y se dispone un conjunto de billetes con dicha cantidad (minimizando el número de billetes emitidos).
 - **Importancia:** Vital.

- **Estabilidad:** Alta.
- **Fuente:** Julián Flores González.
- **Criterio de validación:** el sistema realiza las operaciones correctamente (sustracción y emisión de billetes).

- 4.2. Diseña el modelo de contexto, el DFD de primer nivel de forma que solo englobe las funcionalidades del apartado anterior**
- 4.3. Para el RF3, haz el DFD de nivel 2, con su correspondiente DFC asociado**
- 4.4. En el siguiente contexto, ¿cuál sería el modelo de ciclo de vida más apropiado? Motiva tu respuesta:**

“El software va a ser desarrollado por una empresa que amplia experiencia en el desarrollo software bancario. Los requisitos están perfectamente definidos, pero el cliente quiere dotar al cajero de funcionalidades adicionales a las clásicas que le darían una ventaja competitiva sobre la competencia (que es muy activa), lo que exige resultados operativos a corto plazo.” El ciclo de vida elegido es “Programación Extrema” por los siguientes motivos:

- La dinámica de historias de usuario permite al cliente priorizar las funcionalidades que quiera por lo que es útil si se requieren resultados a corto plazo.
- La dinámica de historias de usuario proporciona entregables frecuentes lo que posibilita tener resultados operativos a corto plazo.
- Al ser una metodología ágil, tiene una menor carga de modelos y documentación en comparación con las metodologías clásicas por lo que las entregas serán más rápidas.

- 4.5. Establece un proceso de administración de riesgos, siguiendo la metodología de Sommerville, que contemple solo 3 riesgos, para el desarrollo del software en el siguiente contexto:**

“El cajero tiene que admitir tarjetas de los siguientes tipos: RB, Rabocard, ENSO, Taboabank. Tiene que tener un funcionamiento 24/7. Sabemos que en el mercado hay competidores que están trabajando en un producto similar”

4.5.1. Identificación y análisis de riesgos

- **ID:** R-001
- **Nombre:** Aparece un nuevo tipo de tarjeta de crédito.
- **Descripción:**
- **ID:** R-002
- **Nombre:** el cajero deja de dar servicio debido a un error de software.
- **Descripción:** Un error de software provoca que el cajero quede inutilizado.
- **ID:** R-003
- **Nombre:** Un competidor crea un producto similar que supera en funcionalidades nuestro producto.

4.5.2. Planificación de riesgos

4.6. Pensando en el proceso de pruebas, establece las clases de equivalencia para las siguientes entradas:

- Tipo de tarjeta
- PIN
- ¿Desea imprimir el recibo?

“Excelente”
- Sturm Magazine



**“Desafiante y
diferente”**



“Una experiencia única” - IGN



La ingeniería del software, una historia llena de trabajo, superación y conflicto que no deja indiferente a nadie.

¡Adéntrate en los entresijos de los estándares más arcanos y olvidados!

¡Siente en tus propias carnes la indefensión, el miedo y la procrastinación!

PENETRA EN LA MENTE DEL RABO.



CORBA COMICS