

Ingeniería del Software

Escuela Técnica Superior de Ingeniería
Universidad de Santiago de Compostela

Curso 2019–2020

Los autores de este documento informan:

Este documento se encuentra bajo la licencia EUPL V1.1.

Como el CMMI, este documento se encuentra en una mejora continua de su calidad. En caso de encontrar fallos ortográficos, discrepancias con el temario de la asignatura, o querer ampliar sus contenidos, por favor, considere el colaborar con el repositorio oficial disponible en: https://github.com/Zambumon/ENSO_DENSO.

Contribuidores (en orden cronológico):

@Zambumon

@manudroid19

@alvrogd

Índice

I El Software	5
1. Características	5
2. Atributos deseables en el software	6
2.1. Aplicaciones del software	6
3. Software heredado	7
4. Principales problemas asociados a la producción del software	7
II El proceso	8
1. Conceptos	8
2. Procesos para la construcción del software	8
2.1. Norma IEEE 1074 – 2006	9
2.2. Norma ISO 12207–1	10
2.2.1. Procesos principales	10
2.2.2. Procesos de soporte	11
2.2.3. Procesos de la organización	12
2.3. Norma ISO/IEC TR 15504–2 (2003)	12
2.4. Descripción de procesos en la Norma ISO 12207 (2008)	13
3. Evaluación del proceso software	13
3.1. Capability Maturity Model Integration (CMMI)	13
III Ciclos de vida	16
1. Ciclo de vida en cascada	16
1.1. Fases	17
1.2. Aportaciones	18
1.3. Problemas	18
2. Construcción de prototipos	18
2.1. Sistemas que resultan ser buenos candidatos	19
2.2. Beneficios de los prototipos	19
2.3. Formas de prototipos	19
2.4. Construcción de un prototipo	20
2.5. Problemas	20
3. Ciclo de vida incremental	20
3.1. Sistemas que resultan ser buenos candidatos	21
3.2. Ventajas	21
3.3. Desventajas	21
4. Técnicas de cuarta generación	21
4.1. Críticas más habituales	22

5. Modelo en espiral	22
5.1. Descripción del modelo	22
5.2. Ventajas	23
5.3. Desventajas	23
5.4. Análisis de riesgos	23
5.4.1. Definiciones	23
5.4.2. Anticipación a los riesgos	24
6. Metodologías ágiles	25
6.1. Variabilidad del entorno	26
6.2. Fragilidad por las debilidades humanas	26
6.3. Diferentes modelos de proceso	26
6.4. Modelado ágil	27
6.5. Programación extrema	27
IV Análisis de requisitos	30
1. Introducción	30
1.1. Requisito	30
1.1.1. Características comunes a todos los requisitos	30
1.1.2. Clasificación de los requisitos	30
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 de recogida de información	32
2.2. TFEA	33
3. Análisis de requisitos del sistema	33
3.1. Identificación de las necesidades del cliente	33
3.2. Análisis de alternativas	34
3.3. Evaluación de la viabilidad del sistema	35
3.4. Representación de la arquitectura del sistema	36
4. Especificación del sistema	38
5. Análisis de requisitos del software	38
5.1. Objetivos y actividades	38
5.1.1. Principios del análisis de requisitos del software	39
6. Especificación del software	40
6.1. Principios de especificación	40
6.1.1. Características del documento de especificación	40
7. Verificación y Validación de requisitos	42
7.1. Estrategias	42
7.2. Técnicas	42

8. Administración de requisitos	43
8.1. Clasificación cualitativa	43
8.2. Clasificación cuantitativa	43
8.3. Planificación de la administración de requisitos	44
8.4. Administración del cambio de requisitos	44
V Análisis estructurado	45
1. Introducción	45
1.1. Problemas del análisis clásico	45
1.2. Soluciones al análisis clásico	46
2. Técnicas de especificación y modelado	46
2.1. Diagramas de flujo de datos (DFD)	47
2.1.1. Elementos	48
2.1.2. Niveles	49
2.2. Especificaciones de proceso (PSPEC)	52
2.3. Estrategia de creación de DFDs y PSPECs	52
2.4. Diagramas de flujo de control (DFC)	53
2.4.1. Elementos	54
2.4.2. Cómo separar datos y control	56
2.4.3. Cuándo usar especificaciones de control	56
2.5. Especificaciones de control (CSPEC)	56
2.6. Estrategia de creación de DFCs y CSPECs	58
2.7. Diagramas Entidad–Relación (DER)	59
2.8. Comprobaciones	59
2.9. Consistencia entre modelos	59
3. Metodología del análisis estructurado	60
3.1. Fases	60
3.1.1. Creación del modelo de procesos	60
3.1.2. Creación del modelo de control	61
3.1.3. Creación del modelo de datos	61
3.1.4. Consistencia entre modelos	61
4. Modelos del sistema	62
4.1. El modelo esencial	62
4.2. Modelo de implementación	62
VI Pruebas del software	64
1. Introducción	64
2. Definiciones	64
3. Filosofía de las pruebas del software	65
3.1. Principios	66
4. El proceso de prueba	68

5. Técnicas de diseño de casos de prueba	68
6. Pruebas estructurales (de caja blanca)	69
6.1. Criterios de cobertura lógica	70
6.2. Prueba de bucles	71
6.3. Utilización de la complejidad ciclomática de <i>McCabe</i>	72
7. Pruebas funcionales (de caja negra)	74
7.1. Enfoque sistemático	74
7.1.1. Partición o clases de equivalencia	74
7.1.2. Análisis de valores límite (ALV)	75
7.1.3. Conjetura de errores	76
7.2. Enfoque aleatorio	77
7.3. Métodos de prueba basados en grafos	77
8. Enfoque práctico recomendado para el diseño de casos	79
9. Documentación del diseño de las pruebas (IEEE 829)	79
10. Ejecución de las pruebas	81
10.1. El proceso de ejecución (IEEE 1008)	81
10.2. Documentación de la ejecución de las pruebas	83
10.3. Depuración	83
10.3.1. Consejos para la depuración	84
10.4. Análisis de errores a análisis causal	85
11. Estrategia de aplicación de las pruebas	85
12. Pruebas en desarrollos orientados a objetos (<i>OO</i>)	86
12.1. Punto de vista del diseño de casos de pruebas	86
12.2. Cuestiones a tener en cuenta	87
13. Ejemplo práctico de cálculo de la complejidad ciclomática de un método	88

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, en lugar de modificar uno ya existente, pero que acabaría presentando los mismos problemas.

1. Características

Principalmente, el software se diferencia principalmente por presentar una **naturaleza lógica** (es inmaterial); por ello, se dice que el **software se desarrolla, no se fabrica** en sentido estricto. A pesar de presentar muchas similitudes con los productos de otras ingenierías (*fases de análisis, diseño, etc.*), el software se diferencia 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 por introducir más problemas antes de solucionar los ya existentes, degradando así 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.

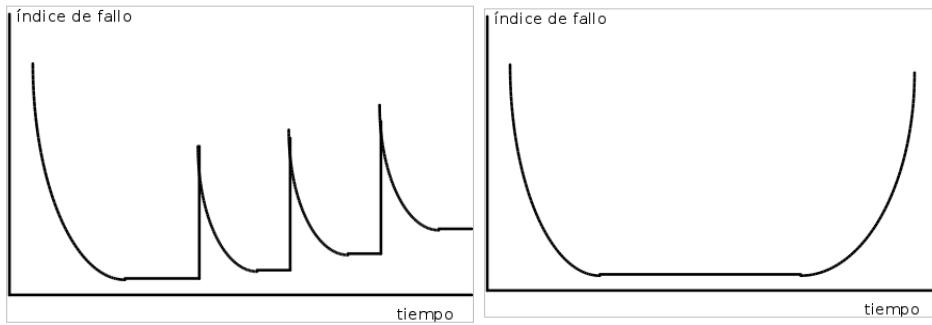


Figura 2: Curva de fallos del Software (izquierda) y del Hardware (derecha). Obsérvese como, en el caso del software, el índice de fallos se va acumulando. Idealmente, el software solo presentaría fallos al principio de su vida, por errores no detectados durante el desarrollo.

- Aplicación de patrones de diseño, que supone el poder reutilizar diseños, el lugar de solo código.

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, realmente, 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; es decir, están hechos a medida.
- **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* (**NO** son máquinas de estados).

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 (por ejemplo, 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. *Por ejemplo: Ejecución del caso de prueba.*

Actividad Conjunto de tareas que producen un producto importante del trabajo. *Por 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. *Por ejemplo: Validación.*

Modelo de los procesos Descripción de los procesos involucrados en el desarrollo del software sin especificar cómo 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. *Por ejemplo: IEEE 1008.*

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

Herramienta Cualquier software que nos ayude en cualquier etapa del desarrollo. *Por 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.

El 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 de 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.

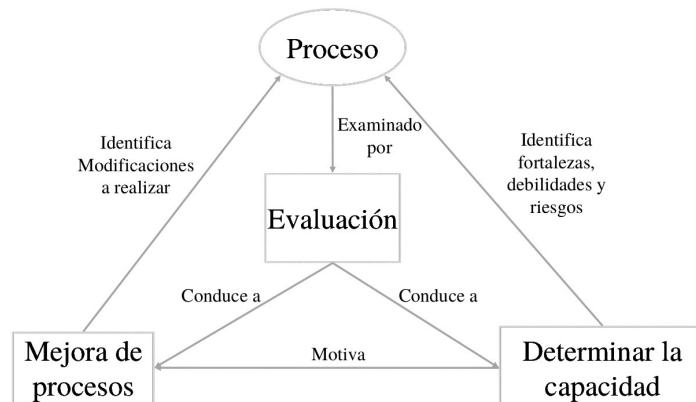


Figura 3: Esquema de la evaluación del proceso software.

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 Permite establecer el nivel de madurez global de los proyectos y de la organización. Define las mismas áreas y metas específicas. 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 (*un nivel de madurez es para un conjunto de áreas, mientras que un nivel de capacidad es para cada área*):

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, tal 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 son 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

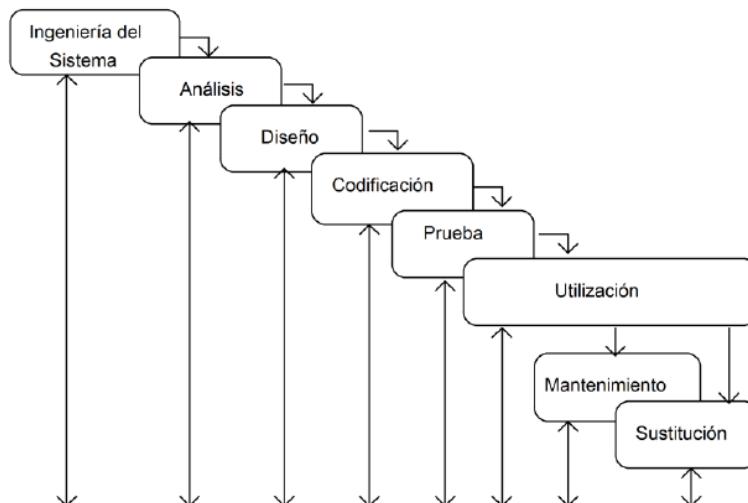


Figura 4: Etapas del ciclo de vida en cascada.

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 tampoco 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, por lo que sabes lo que tienes que hacer en cada momento.
- Permite generar software eficientemente, y de acuerdo a las especificaciones, evitando así sobrepasar fechas de entrega o costes esperados.
- Los interesados pueden comprobar el estado del proyecto al final de cada fase.

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 que 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 que 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 todo 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 palia 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.

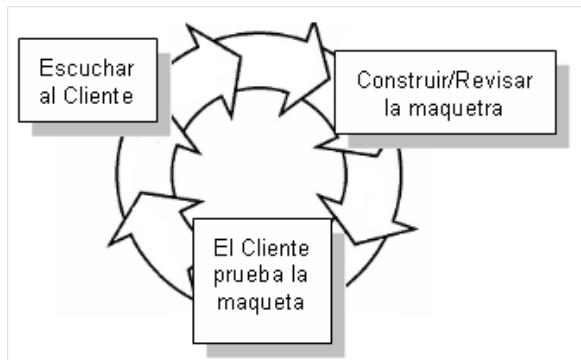


Figura 5: Etapas del paradigma de la construcción de prototipos.

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, o incluso simulando la funcionalidad del *back-end* de la aplicación.

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 y 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. Por lo tanto, 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 final** (es decir, del sistema final) 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

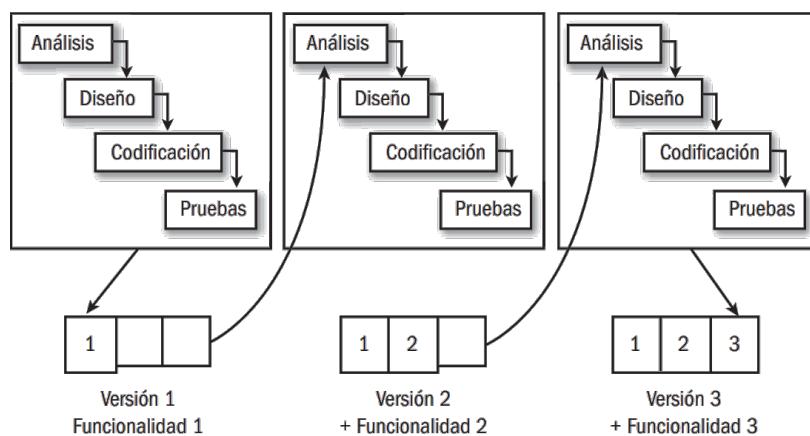


Figura 6: Etapas del ciclo de vida incremental.

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**.

En comparación al modelo por prototipos, el incremental se centra en obtener un producto operativo en cada iteración, en lugar de simples prototipos; 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.

Nota: *En relación a la figura mostrada, el analizar solo al principio los requisitos del sistema sería suficiente generalmente, e incluso ni siquiera sería necesario volver al análisis del software en sucesivas iteraciones si está bien claro qué hacer desde el principio, y el equipo de desarrollo se encuentra seguro de ello.*

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

- Si la parte de requisitos no se encuentra en la fase iterativa, es difícil ver si estos son válidos, y se siguen detectando los errores relativamente tarde, al haberse construido ya productos operativos en cada iteración (mismos problemas que con el ciclo en cascada). Y, aunque se realice una fase de análisis en cada iteración, al no haber un contacto con el cliente hasta dar por finalizado el incremento, los requisitos mal interpretados siguen siendo detectados relativamente tarde.

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.
- Generalmente, solo son aplicables al software de gestión.

5. Modelo en espiral

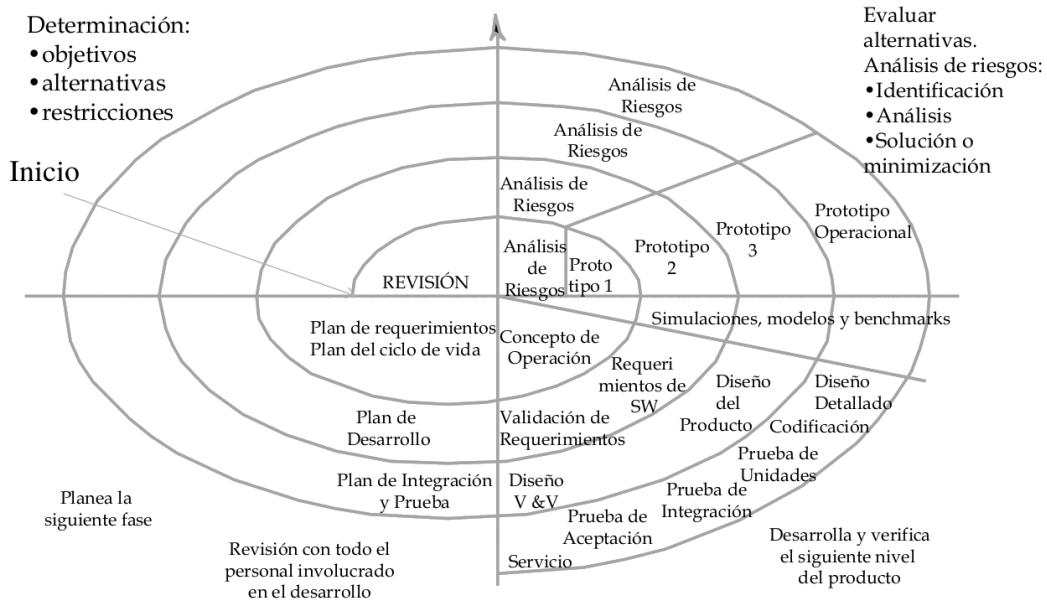


Figura 7: Etapas del modelo en espiral.

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 que 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 generan 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 contratación 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 evaluación 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, y por lo que 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, o 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.

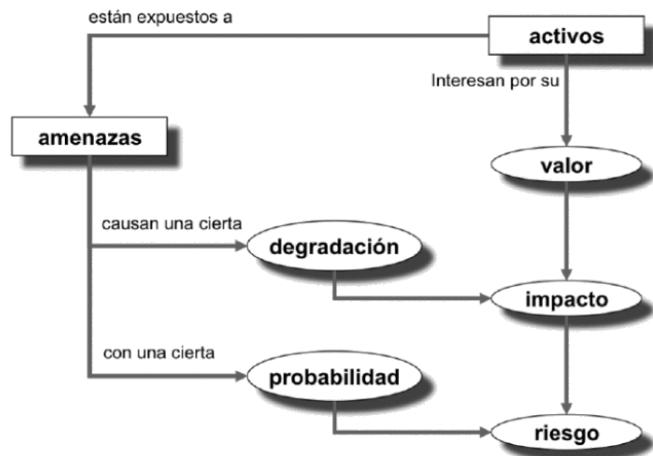


Figura 8: Elementos del análisis de riesgos potenciales.

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 a los recursos.
 - Del producto: Afectan a la calidad del producto.
2. **Análisis de riesgos:** Se evalúa, 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 e insignificante, por ejemplo). A pesar de emplear una escala a intervalos, cada una de estas valoraciones debe sustentarse sobre medidas objetivas, por lo que se emplearán medidas objetivas para estimar subjetivamente las consecuencias; 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; dicho de otro modo, si sucede algún problema, la responsabilidad es exclusiva al experto subcontratado (*por ejemplo, deberán llevar a cabo indemnizaciones...*).

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.

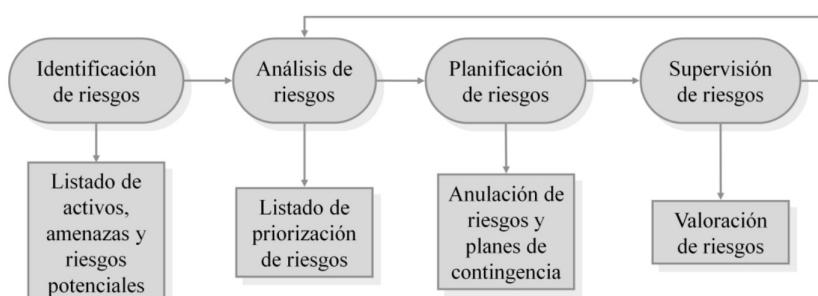


Figura 9: Administración del riesgos.

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, y 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; de todos modos, esto 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 soltar 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 el modelado y la 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; lo demás tan solo supone esfuerzo invertido de forma innecesaria.
- 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

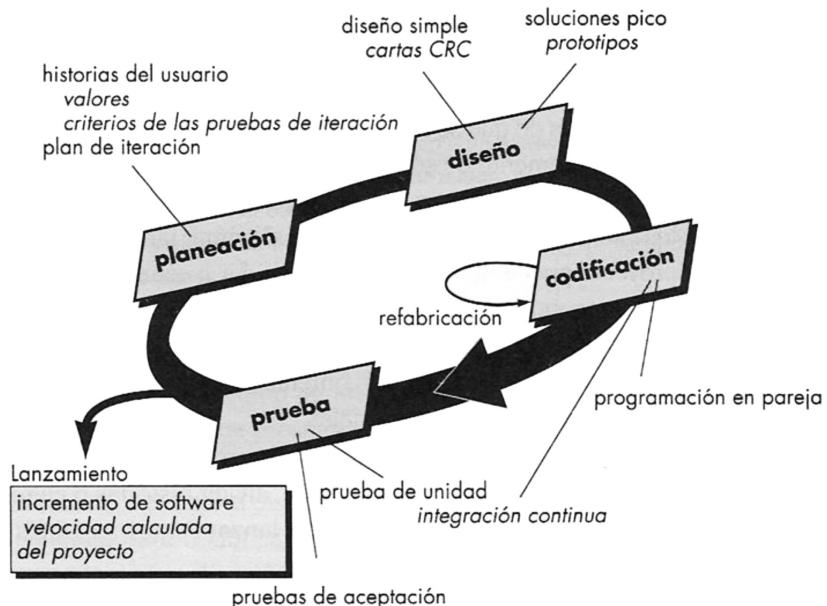


Figura 10: Etapas de la programación extrema.

Utiliza, preferiblemente, un enfoque orientado a objetos, y abarca un conjunto de reglas y prácticas que ocurren en 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 caso 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 del desarrollo del proyecto, no solo en la tarea anterior.*

- **Plan de construcción:** Los clientes y el equipo se reúnen para decidir qué historias hay en el siguiente incremento del software, priorizando unas sobre otras 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:

- Despues 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

En todos los modelos de ciclos de vida existe la fase de **análisis de requisitos**, en la cual se estudian las características y funciones del sistema, la definición de los requisitos del software y del sistema del que forma parte, la planificación inicial del proyecto, y tareas de análisis de riesgos. Este análisis se centra en la información, funcional y de comportamiento del problema, el cual se divide en partes que se modelan para comprenderlo mejor, con el objetivo de desarrollar una **especificación de requisitos**, en donde se describe el problema analizado y la solución propuesta.

Como se puede prever, esta especificación se convierte en base de todas las actividades que siguen y, por ende, tiene una gran influencia en la calidad de la solución implementada finalmente. El principal problema es que, al inicio de un proyecto, es difícil tener una idea clara y segura de los requisitos del sistema y del software, para comprender qué debe realizar este último; por ello, algunos de los ciclos de vida estudiados proponen enfoques cíclicos de refinamiento de los requisitos.

1.1. Requisito

Se define como una condición o capacidad que necesita el usuario para resolver un problema o conseguir un objetivo determinado. Por extensión, se corresponde también con las condiciones o capacidades que debe cumplir o poseer un sistema para satisfacer un contrato, una norma o una especificación.

1.1.1. Características comunes a todos los requisitos

- Son una abstracción de las características del sistema (modelan el mismo).
- Representan el sistema de forma jerárquica, particionando el problema en varios niveles de detalle.
- Definen interfaces del sistema, tanto externas como internas.
- Sirven de base para las etapas posteriores del ciclo de vida.
- Existen requisitos de usuario, de sistema, esenciales, y de implementación.
- No prestan demasiada atención a las restricciones o criterios de validación (exceptuando los métodos de especificación formal).

1.1.2. Clasificación de los requisitos

- **Requisitos funcionales:** Servicios que proveerá el sistema, concretando de qué manera reaccionará a entradas particulares, y cómo se comportará en situaciones particulares; también declaran lo que el sistema no debe hacer. *Ejemplo: El usuario tendrá la posibilidad de buscar por tamaño.*

Si un requisito funcional no se cumple, el sistema se verá degradado al no contener toda su funcionalidad.

- **Requisitos no funcionales:** Restricciones sobre los servicios o funciones ofrecidos por el sistema; es decir, no se refieren directamente a las funciones específicas del sistema, sino a propiedades de estas. Pueden ser temporales, de fiabilidad, de robustez, de capacidad, de ajuste a estándares, etc.; además, deben expresarse de forma cuantitativa. *Ejemplo: La duración de la sesión ha de ser superior a 30 minutos.*

Con frecuencia hacen referencia al sistema como un todo, por lo que los fallos en ellos lo inutilizan.

- **Requisitos del dominio:** Provienen del entorno en el que se realiza la aplicación, reflejando por lo tanto las características de este, y pueden ser funcionales o no funcionales. Dada esta naturaleza, presentan la dificultad de que sean expresados en un lenguaje específico de la aplicación en donde el analista no comprenda todos los detalles. *Ejemplo: usar el formato DICOM en medicina es a la vez un requisito no funcional y de entorno.*

Es importante respetarlos dado que es imposible que el sistema trabaje satisfactoriamente si se incumplen.

Nota: *Al expresar requisitos funcionales, es necesario ser precavido si se emplea el lenguaje natural, dado que puede dar lugar fácilmente a ambigüedades e inconsistencias.*

1.2. Análisis de requisitos

De forma concisa, y visto todo lo anterior, se podría decir que es el 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.

Cabe destacar que, dadas todas las dificultades que puede presentar, y presentará con alta probabilidad, es una fase que tiende a extenderse indefinidamente. El tiempo que se le dedique dependerá al final del propio proyecto, pero puede emplearse la regla de 40–20–40 como una guía; *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** (comprometido): Es el encargado de la comunicación con el usuario/cliente (no saben qué información es necesaria para el desarrollo) y los desarrolladores de software (no conocen el dominio de explotación), **sirviendo de puente** entre ambos; por ello, es además habitual que esté presente en todas las revisiones que se hagan a lo largo del desarrollo del proyecto.

Para esta labor, debe contar con las capacidades de comunicarse con múltiples personas, cada una con una visión distinta del problema, extraer información de ellas y reorganizarla para sintetizar soluciones (traducir ideas vagas de necesidades de software en funciones y restricciones concretas) y hacer de mediador. Para ello, es además probable que requiera conocimientos sobre el campo de actividad del cliente.

2. **Stakeholders** (involucrados, que son meros observadores, o comprometidos): Personal involucrado en el proyecto, incluidos los usuarios/cliente, y que tiene influencia directa o indirecta sobre los requisitos del sistema.

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

Esencialmente, es porque consiste en la **traducción de vagas necesidades de software**, en lenguaje natural que puede ser ambiguo, a un **conjunto concreto de funciones y restricciones**. Concretamente, *Sommerville* establece las siguientes razones:

- Los stakeholders a menudo no conocen realmente lo que desean obtener del sistema excepto en términos generales.
- Los stakeholders expresan los requisitos con sus propios términos y con conocimiento implícito, lo cual debe ser interpretado por los ingenieros.
- 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.
- Siempre se desarrolla el proyecto en un entorno variable, y será inevitable que algunos requisitos puedan cambiar con el paso del tiempo, así como que surjan algunos nuevos.

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 de 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 (JAD)**: Se crean equipos de usuarios y analistas que determinan conjuntamente las características que debe tener el software; el involucrar al usuario implica una mayor probabilidad de éxito. *Por ejemplo, TFEA*.
3. **Prototipado.**
4. **Observación: In situ.**
5. **Estudio de documentación.**
6. **Cuestionarios.**
7. **Brainstorming**: Se sugiere toda clase de ideas sin juzgarse su validez, para luego analizar cada propuesta de modo detallado.
8. **Escenarios/casos de uso.**

Nota: En la práctica, es habitual utilizar combinaciones de diversas técnicas.

2.2. TFEA

Son un conjunto **Técnicas para Facilitar la Especificación de una Aplicación**, que se acerca al concepto de brainstorming (propuesta de ideas y debate para alcanzar un consenso).

La correspondiente metodología de trabajo sería:

1. Reuniones preliminares con el cliente en las que **aclarar el ámbito del problema y los objetivos de una solución**. Debería obtenerse un documento con una descripción del problema, junto con dichos objetivos, e incluso una propuesta de solución si se puede concretar en el momento.
2. Convocatoria de reunión entre el analista, el cliente y el equipo de desarrollo. Habiendo conocido de antemano el anterior documento, cada uno crea además **listas individuales** de:
 - a. Objetos del sistema y del entorno.
 - b. Operaciones que realizan estos objetos o que los relacionan.
 - c. Restricciones.
 - d. Rendimiento.
3. Al comienzo de la reunión **se discute la necesidad y justificación del proyecto**. Si todo el mundo está de acuerdo en desarrollar el proyecto, se prosigue con la creación de una lista conjunta a partir de las individuales, sin eliminar nada.
4. **Discusión de la lista conjunta**, para añadir, eliminar o modificar elementos.
5. Redacción de **miniespecificaciones** de cada elemento (breves definiciones).
6. Redacción de un **borrador de la especificación del proyecto**, junto con las miniespecificaciones, por parte del analista.

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 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 del sistema consta de varias fases.

3.1. Identificación de las necesidades del cliente

El analista de sistemas debe **definir los elementos de un sistema informático dentro del contexto del sistema en que va a ser usado**. Debe distinguir, desde uno de los dos siguientes puntos de vista, entre:

- Requisitos imprescindibles (lo que el cliente necesita) e imprescindibles (lo que le será útil pero no necesita).

- Requisitos normales (vitales), esperados (importantes), y estimulantes (quedaría bien).

Así, el analista parte de los objetivos y restricciones definidos en la obtención de requisitos, y representa la función del sistema, la información que maneja, sus interfaces, y el rendimiento y restricciones del mismo.

El proyecto habitúa comenzar con un concepto más bien vago y ambiguo de cuál es la función deseada, por lo que el analista debe **delimitar el funcionamiento del sistema** durante todo este procedimiento.

3.2. Análisis de alternativas

Una vez delimitado el sistema, el analista debe proceder a la asignación de funciones; cada función del sistema debe ser asignada a alguno de sus componentes (sean software o no). Cada opción de asignación debe ser valorada, en cuanto a ventajas y desventajas, para poder decidirse por una de ellas.

Además de proponer soluciones propias, el ingeniero de sistemas debería valorar también la adopción de soluciones estándar disponibles en el mercado, frente al riesgo de desarrollar una solución propia.

Nota: *Como se ha podido ver, la labor del analista de sistemas consiste en asignar, a cada elemento del sistema, un ámbito de funcionamiento y de rendimiento. El ingeniero del software se encargará de refinar el componente software para producir un elemento funcional que sea integrable con el resto del sistema.*

Para distinguir entre cada una de las alternativas, pueden emplearse técnicas como el **análisis paramétrico** (fijar unos parámetros de evaluación a tener en cuenta), 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.

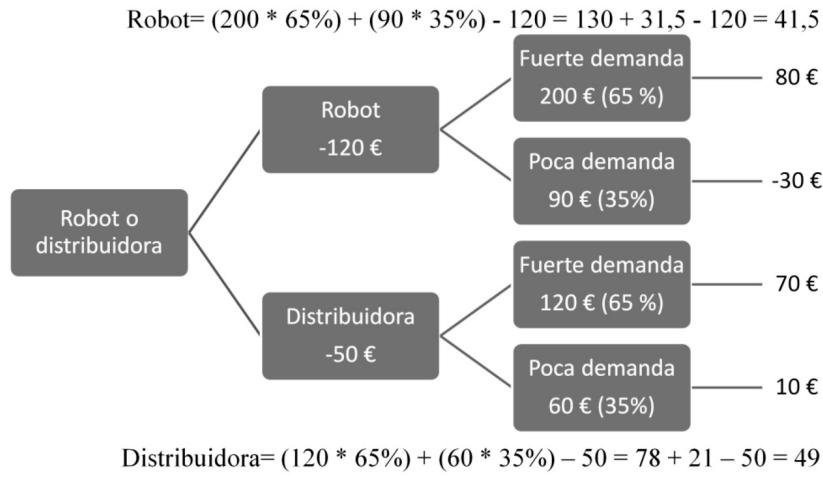


Figura 11: EMV de un sistema de clasificación de paquetes.

3.3. Evaluación de la viabilidad del sistema

Dado que los recursos siempre son limitados, **debe determinarse la viabilidad del proyecto lo antes posible**, para no invertir recursos y tiempo en el desarrollo de proyectos que resulten inviables finalmente. Para ello, el sistema debería poder implantarse con las restricciones de coste y tiempo con las tecnologías actuales, y con un alcance que le resulta útil al usuario.

Nota: *El estudio de viabilidad está muy relacionado con el análisis de riesgos.*

- **Viabilidad económica:** Comparar los costes de desarrollo con los beneficios que producirá. Mientras que los costes de desarrollo pueden ser cuantificados fácilmente, los beneficios futuros son difíciles de determinar; algunos pueden ser tangibles (mayor rapidez en algunas tareas), pero otros muchos son intangibles (satisfacción de uso). Los costes y beneficios pueden ser directos o indirectos dependiendo si se pueden relacionar directamente con la implantación del software.

Al final, la única forma de demostrar los beneficios será comparando los modos de trabajo con el nuevo sistema, frente a los actuales (disminución del tiempo de determinadas tareas, menor necesidad de mano de obra, aumento de productividad...) al ser aspectos evaluables cuantitativamente.

- **Viabilidad técnica:** Determinar si es posible desarrollar o no el proyecto teniendo en cuenta sus restricciones, y los recursos humanos y tecnológicos a nuestra disposición. Además, deben estudiarse las características técnicas del nuevo sistema (capacidad, rendimiento, fiabilidad, seguridad, etc.) para complementar el análisis de coste–beneficio.
- **Viabilidad legal.**
- **Viabilidad operativa:** Si es posible integrar el producto de forma efectiva en la organización de destino.
- **Viabilidad de plazos.**



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

3.4. Representación de la arquitectura del sistema

El sistema debe **ser modelado** como un conjunto de componentes y relaciones entre ellos para **servir de base al trabajo posterior**. Para ello, se emplea habitualmente una representación en diagrama de bloques que represente los subsistemas implicados en él y la interconexión entre ellos.

Pressman propone el uso de los **diagramas de arquitectura**, que se dividen en cinco regiones. De esta forma, permiten además identificar el entorno de una parte del sistema, de forma que se distingan claramente sus interfaces externas.

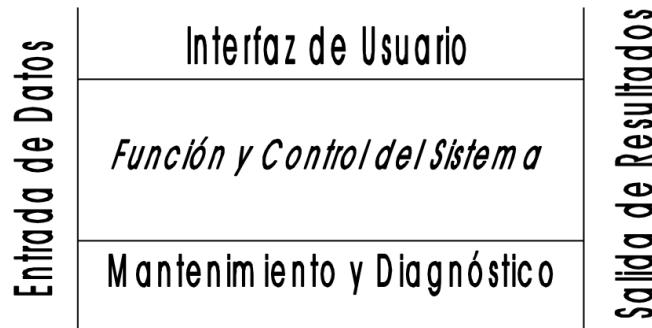


Figura 13: Diagrama de arquitectura de un sistema.

Si la complejidad del sistema lo requiere, puede formarse una jerarquía de niveles en la que el nivel superior se represente el sistema mediante un diagrama de contexto (es el diagrama de arquitectura), y se irá detallando en sucesivos diagramas de flujo (especifican el flujo de información en los subsistemas).

- **Diagrama de contexto:** Representa el sistema en relación con su entorno, definiendo sus límites, y mostrando todos los productores y consumidores de información. Para ello, el sistema se encuentra en el centro del diagrama mientras que, a su alrededor, se sitúan las entidades o agentes externos, en sus correspondientes regiones, con los que se el sistema se relaciona mediante flujos de información.

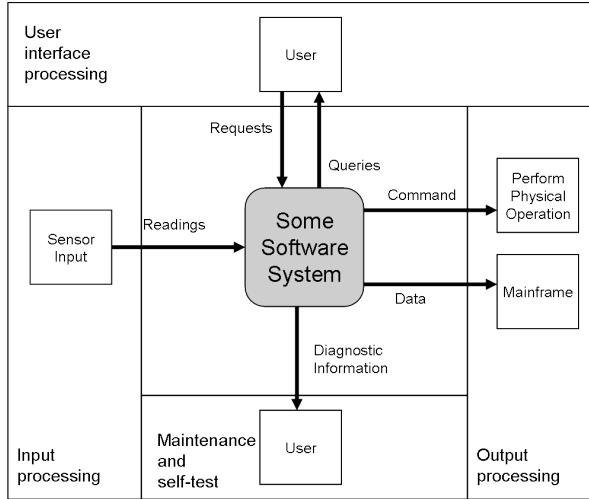


Figura 14: Ejemplo de diagrama de contexto.

- **Diagrama de flujo de arquitectura:** Determinan el flujo de entre los diferentes subsistemas a través de diagramas de flujo convencionales. Cada subsistema puede dar lugar a un diagrama de flujo, que pueden ser descompuestos a su vez en nuevos subsistemas, y ocupa una región del diagrama dependiendo de cuál sea su función (adquisición de datos, salida de datos, interfaz, etc.).

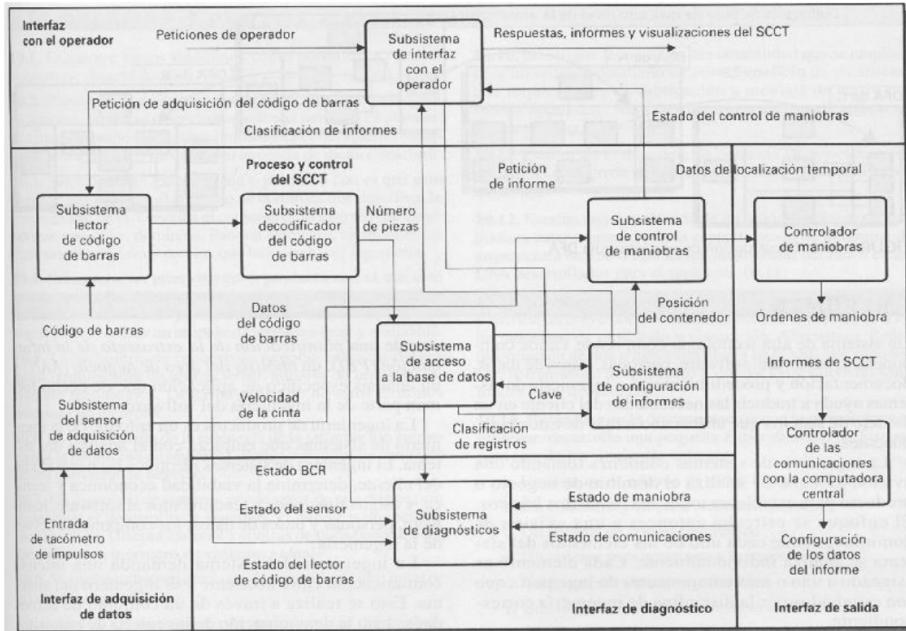


Figura 15: Ejemplo de diagrama de contexto.

Nota: Debe mantenerse la consistencia entre los diagramas de distinto nivel. Al expandir un determinado subsistema en un diagrama de flujo, los flujos de información que conectan dicho subsistema con otros o con agentes externos deben figurar también. Estos flujos tendrán un extremo libre, que será el que los conectaba con dichos elementos que quedan fuera del ámbito del nuevo diagrama de flujo.

4. Especificación del sistema

Es un documento que sirve de base para analizar más adelante cada uno de los componentes que intervienen en el sistema, sean hardware, software o personas. Para ello, se describe la **función, rendimiento y restricciones que debe cumplir el sistema, limitando e indicando también el papel e interfaz de cada uno de sus componentes.**

Este documento debería permitir al cliente comprobar si en análisis del ingeniero del sistema satisface sus necesidades, por lo que debe ser revisado de forma conjunta para comprobar si:

- Se ha delimitado correctamente el ámbito del proyecto.
- Se han definido correctamente la funcionalidad, las interfaces y el rendimiento.
- Las necesidades del usuario y el análisis de riesgos justifican el desarrollo.
- El cliente y el analista tienen la misma percepción de los objetivos del proyecto.

Además de la evaluación con el usuario, debe realizarse una evaluación técnica para determinar si:

- Las estimaciones de riesgos, coste y de agenda son correctas.
- Todos los detalles técnicos (funciones, interfaces...) están bien definidos.
- La especificación sirve como base para las fases siguientes.

5. Análisis de requisitos del software

Como resultado de la fase de análisis de requisitos del sistema, el componente de software tiene ahora asignados una función y rendimiento. Es ahora tarea del ingeniero de software el desarrollar, o adquirir si existen alternativas en el mercado, los componentes software para ello.

5.1. Objetivos y actividades

El análisis de requisitos del software tiene como objetivo **desarrollar una representación del software que pueda ser revisada y aprobada por el cliente**, y que sirva de enlace entre la asignación de funciones al software realizada, y el posterior diseño del software.

Desde el punto de vista del analista de sistemas, esta fase **define con mayor precisión** las funciones y rendimiento del software, las interfaces con otros componentes y las restricciones que debe cumplir. Desde el punto de vista del diseñador, este análisis proporciona una representación de la información, la función y el comportamiento del sistema que él se encargará de traducir en un diseño de datos y programas. Además, el análisis de requisitos permite a todos, incluido el cliente, valorar la calidad del software una vez haya sido construido, o incluso antes de ello.

Nota: *El analista debe centrarse en el qué, no en el cómo.*

Pressman propone que el análisis de requisitos del software puede dividirse en cinco áreas:

- Inicialmente, el analista estudia la especificación del sistema para comprender el papel del software dentro de este, para ir definiendo y refinando los flujos y la estructura de la información, la función del programa y su papel, así como las interfaces y restricciones.
- Con este conocimiento, se proponen y evalúan diferentes soluciones al problema, hasta que se acuerda una solución con el cliente, y se puede especificar así el software de forma adecuada para que se puedan efectuar las siguientes fases de desarrollo.
- La **síntesis de modelos** que se realiza, además de permitir entender mejor los flujos de información y la función de cada elemento del software, dará pie a modelos que sirvan de base para el diseño de software, así como que puedan ser empleados para la creación de prototipos.
- La **especificación del software debe indicar qué es lo que debe hacer**, pero no cómo debe hacerlo. Pueden figurar características como el rendimiento, protocolos, estándares, soporte de hardware...
- Será necesario **revisar la especificación**, dado que inicialmente será un documento ambiguo, incompleto, incorrecto e inconsistente.

5.1.1. Principios del análisis de requisitos del software

Por una parte, los dos primeros puntos se refieren a representar el sistema y la información que maneja mediante diagramas o representaciones textuales; por otra parte, los dos últimos se refieren a un método de trabajo *top-down*.

1. **Identificar y representar el ámbito de información del sistema:** La tarea que realiza el software consiste siempre en procesar información, pero esta puede estar representada tanto por **datos** (información que procesa el sistema) como por **sucesos o eventos** (cuándo debe procesarse esa información). Así, el ámbito de información admite dos puntos de vista: flujo de datos y flujo de control; de todos modos, en ambos casos se requiere concretar el contenido de la información, su estructura y el flujo.
2. **Modelar la información, la función y el comportamiento del sistema:** Se emplean **modelos de datos** (información que transforma el software), **modelos de procesos** (funciones que transforman la información) y **modelos de control** (comportamiento del sistema). Los modelos se utilizan para comprender mejor lo que se quiere construir, dado que:
 - 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:** Para abordar problemas que son demasiado grandes o complejos.
4. **Avanzar desde lo más general a lo más detallado:** Descomponiendo un problema en subproblemas, y definiendo modelos sobre cada uno, conformando una jerarquía de modelos.

6. Especificación del software

La especificación del software es el documento que culmina el análisis de requisitos, conteniendo una **descripción detallada** del ámbito de información, funciones y comportamiento asignados al **software**, así como **información sobre requisitos** de rendimiento, restricciones diseño, y sobre las pruebas a realizar una vez se haya construido el software (de aceptación).

6.1. Principios de especificación

Pressman propone los siguientes principios a seguir en la especificación del software:

1. **Debe modelar el dominio del problema:** Debe describir el sistema tal y como es percibido por los expertos del dominio de aplicación.
2. **Es necesario separar funcionalidad e implementación:** El *qué*, y no el *cómo*.
3. **El lenguaje de especificación debe estar orientado al proceso:** El sistema debe considerar la posibilidad de modificar su comportamiento por interactuar en un entorno dinámico.
4. **Debe abarcar todo el sistema del que el software es parte:** El comportamiento de una parte, como lo es el software, solo puede ser definido en el contexto del sistema completo.
5. **Debe abarcar también el entorno del sistema:** Siguiendo el principio anterior. Permite además describir la interfaz del sistema, del mismo modo que sucede con las interfaces de los componentes del mismo.
6. **La especificación debe ser operativa:** Debe servir para determinar si una implementación concreta la satisface.
7. **Debe ser ampliable y tolerante a la incompletitud:** Al ser una especificación un modelo de un sistema real, nunca será completa (*desde luego, será extremadamente difícil conseguirlo*), pero puede desarrollarse de forma incremental, a distintos niveles de detalle.
8. **Debe estar localizada y débilmente acoplada:** La especificación sufrirá continuas modificaciones, por lo que su estructura debe facilitarlas todo lo posible.

6.1.1. Características del documento de especificación

En cambio, *Piattini* propone una serie de características que debe cumplir la especificación para ser un documento útil:

1. **No ambigua:** Cada característica del producto debe ser descrita con un término único y, en los casos en que un término se use en distintos contextos por distintos significados, debe incluirse un **glosario**. El uso del lenguaje natural implica un gran riesgo por su ambigüedad inherente.
2. **Completa:** Una especificación de requisitos software (**ERS**) está completa si:
 - Incluye todos los requisitos significativos del software.
 - Define la respuesta software a todas las posibles entradas en todas las posibles 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). Sin embargo, hay 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:** Si, y solo si, ningún conjunto de requisitos descritos en ella son contradictorios, total o parcialmente. Se pueden presentar tres tipos distintos de conflictos:
- Se define el mismo objeto real pero con distintos términos para designarlo.
 - Las características especificadas para objetos reales están en conflicto; *por ejemplo, luces azules vs. luces verdes*.
 - Hay un conflicto lógico o temporal entre dos acciones; *por ejemplo, sumar dos valores vs. multiplicarlos*.
5. **Fácil de modificar:** Estar apropiadamente organizada, y no contener redundancia.
- Nota:** *La redundancia en sí no es un error, pero puede conducir fácilmente a ello. Es mejor emplear referencias cruzadas en su lugar.*
6. **Facilidad para identificar el origen y las consecuencias de cada requisito:** De este modo, cuando se debe llevar a cabo una modificación, pueden determinarse claramente los requisitos que pueden verse afectados por ellas. Se consigue estableciendo la trazabilidad.
7. **Facilidad de utilización durante la fase de explotación y mantenimiento:**

- Si el personal que encarga del mantenimiento no ha estado relacionado con el desarrollo del producto, y debe llevar a cabo modificaciones “profundas”, es necesario actualizar la documentación de diseño y de los requisitos. Por ello, además de que la ERS debería ser fácilmente modificable, debería contar con un registro de las características especiales de cada componente, como criticidad u origen, para facilitar dichos cambios; *es decir, será mejor poder disponer de toda la información posible para evitar cometer errores*.
- Podría darse el caso de que parte de la información necesaria para el mantenimiento se dé por supuesta en la organización del desarrollo y, si la organización del mantenimiento no cuenta con ella, supondrá un gran problema.

Nota: *Al igual que sucede con la especificación del sistema, la especificación de requisitos del software debe ser revisada conjuntamente por el cliente y el ingeniero del software. En primera instancia, debe determinarse si el sistema cumple, según el papel aquí especificado para el software, con la función, restricciones y rendimiento requeridos por el cliente, y ver si es necesario cambiar algunas de las estimaciones de coste, riesgo o agenda. En segundo lugar, debe realizarse una revisión mucho más detallada para detectar imprecisiones o ambigüedades en la especificación.*

7. Verificación y Validación de requisitos

La **validación de los requisitos** es la que permite **mostrar** (verificar) que éstos son los que **definen el sistema que el cliente desea**.

Presenta una relativa similitud con el análisis, ya que implica encontrar problemas con los requisitos, pero discieren en que la validación comprende un bosquejo completo del documento de requisitos, mientras que el análisis implica trabajar con requisitos incompletos.

7.1. Estrategias

Durante el proceso de validación, deben llevarse a cabo diferentes tipos de comprobaciones sobre los requisitos:

1. **Comprobación de validez.** Dado que cada sistema tendrá sus propias necesidades, ¿la especificación responde a todas las necesidades del proyecto? *Por ejemplo, es posible que se identifique una necesidad de contar con determinadas funciones adicionales a medida que los analistas comprenden mejor el sistema que el cliente desea.*
2. **Comprobación de consistencia:** Los requisitos en el documento no deben contradecirse.
3. **Comprobación de totalidad:** Se deben incluir requisitos que definan todas las funciones y restricciones propuestas para el sistema.
4. **Comprobación de realismo:** Asegurar que los requisitos puedan implementarse actualmente, y teniendo en cuenta además restricciones de presupuesto y calendarización.
5. **Verificabilidad:** Debe ser posible diseñar un conjunto de verificaciones (pruebas) para demostrar que el sistema a entregar cumple los requisitos.

7.2. Técnicas

Las técnicas dictan cómo llevar a cabo las estrategias:

1. **Revisiones de requisitos:** Varios implicados, pertenecientes tanto al cliente como al contratista, llevan a cabo un proceso de análisis sistemático del documento de requisitos para encontrar anomalías y omisiones.
2. **Construcción de prototipos:** Se muestra un modelo ejecutable del sistema a los usuarios finales para que experimenten y comprueben si cumple sus necesidades.
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 en una notación estructurada o formal.

Nota: *No deben menospreciarse las dificultades en la validación de requisitos, dado que es difícil demostrar que un conjunto de requisitos cumple las necesidades del usuario, por requerir visualizar cómo el sistema final encajaría en su entorno de explotación; esto será complejo para los profesionales de la computación, pero aún más para los usuarios. Por ende, la validación de requisitos probablemente descubrirá todos los problemas en las interpretaciones realizadas sobre las necesidades del cliente, y será inevitable corregir todos estos problemas.*

8. Administración de requisitos

La **ERS** **habitúa ser un proceso iterativo**, como consecuencia del progreso del producto de software; a fin de cuentas, es **casi imposible especificar algunos detalles en el momento que se inicia el proyecto**. También es muy probable que se realicen cambios adicionales como resultado de haber encontrado deficiencias.

Dos consideraciones a tener en cuenta en este proceso son:

- Especificar los requisitos de la forma más completa posible, a pesar de que se prevean de forma inevitable revisiones en él.
- Debe iniciarse un proceso de gestión formal del cambio, para identificar, controlar, seguir e informar de posibles cambios tan pronto como sean identificados. De este modo, se podrá contar con un rastro preciso de las modificaciones de la ERS, así como discernir entre los fragmentos actuales de la ERS y sus anteriores versiones.

Como se puede ver, la administración de requisitos es el proceso de comprender y controlar los cambios en los requisitos del sistema. Consta de **dos etapas**: la planificación debería comenzar al mismo tiempo que la obtención de requisitos inicial, y la administración activa debe comenzar tan pronto como esté lista la primera versión del documento de requisitos.

8.1. Clasificación cualitativa

Desde una perspectiva evolutiva, los requisitos son de una de las siguientes dos clases:

- **Requisitos duraderos**: Relativamente estables dado que derivan de la actividad de la organización y, por lo tanto, están relacionados directamente con el dominio del sistema. *Por ejemplo, en un hospital siempre habrá pacientes, doctores...*
- **Requisitos volátiles**: Probablemente cambiarán durante el desarrollo del sistema o después de que se haya puesto en operación. *Por ejemplo, por cambios en las políticas gubernamentales de salud.*

Sommerville clasifica estos últimos en:

- **Cambiantes**: Debido al ambiente en que opera la organización.
- **Emergentes**: Surgen al incrementarse la compresión del cliente en el desarrollo del sistema; dicho de otro modo, *se entiende más sobre lo que el cliente desea*.
- **Consecutivos**: Son resultado de la introducción del sistema en su entorno de explotación (puede suponer cambios en los procesos de la organización).
- **De compatibilidad**: Dependen de sistemas particulares o procesos de negocios dentro de la organización.

8.2. Clasificación cuantitativa

Desde otro punto de vista, simplemente pueden clasificarse asignándoles el **valor de estabilidad** que se considere apropiado: baja, media y alta.

8.3. Planificación de la administración de requisitos

La administración de requisitos es muy cara, por lo que es necesario definir en todo proyecto el nivel de detalle necesario para este proceso. Para ello, deben concretarse los siguientes aspectos:

- **La identificación de requisitos:** Cada requisito debe ser identificable de forma única.
- **Un proceso de administración del cambio:** Conjunto de actividades que evalúan el impacto y costo de los cambios.
- **Políticas de rastreo:** Definen las relaciones entre requisitos dependientes, entre requisitos y los módulos del diseño en los cuales serán implementados, y entre los requisitos y sus fuentes (quién y por qué se propusieron). Esto permite que, cuando un cambio sea propuesto, se pueda rastrear el impacto en los requisitos y en el diseño, de modo que se pueda valorar si proceder a realizar el cambio o no.
- **Ayuda de herramientas CASE** (*Computer Aided Software Engineering*).

Nota: Es habitual registrar esta información de rastreo mediante matrices de rastreo. Además, la administración de requisitos idealmente recurrirá a la ayuda de herramientas CASE para almacenarlos, y administrar sus cambios y trazabilidad.

8.4. Administración del cambio de requisitos

Esta administración se aplica a todos los cambios propuestos en los requisitos. La ventaja de emplear un proceso formal es que todos los cambios propuestos son tratados **de forma consistente** y que los cambios en el documento de requisitos se realizan **de forma controlada**.

Nota: La gestión del cambio es un proceso aparte para todos los artefactos del proyecto, pero los primeros sobre los que se aplica son los requisitos.

Existen tres etapas principales:

- **Análisis del problema y especificación del cambio:** Tras **identificar un problema**, o a veces recibir una **propuesta de cambio**, se analiza la información correspondiente para verificar que es válida y, entonces, se hace una propuesta de cambio de requisitos más específica.
- **Análisis del cambio y costeo:** Se valora el efecto de un cambio propuesto empleando la información de rastreo y el conocimiento general de los requisitos del sistema. Finaliza con la toma de decisión de si procede o no el cambio.
- **Implementación del cambio:** Se modifica el documento de requisitos y, si procede, el diseño e implementación del sistema.

En caso de que se requiera de forma urgente un cambio en los requisitos del sistema, existirá la tentación de hacer el cambio del sistema y de modificar el documento tras ello. Esto conduce, inevitablemente, a que la especificación de requisitos y el sistema se desfasen, de modo que no sean consistentes el uno con el otro por, por ejemplo, posponer la actualización del documento.

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.

Todos los métodos de análisis de requisitos se basan en la construcción de modelos del sistema que se pretende desarrollar, creando modelos que reflejen el sistema, y aplicando técnicas de descomposición y razonamiento *top-down*. El desarrollo de modelos presenta claras ventajas como:

- **Ayudar a entender y corregir:**
 - Permiten centrarse en determinadas características del sistema.
 - Permiten realizar cambios y correcciones en los requisitos a bajo coste y sin correr ningún riesgo. Si no se realizasen modelos, estos cambios solo se efectuarían después de construir el producto software.
- **Ayudar a representar y transmitir:**
 - Permiten verificar que el ingeniero del software ha entendido correctamente las necesidades del usuario, y que las ha documentado de forma que los diseñadores y programadores puedan construir el software.
- **Ayudar en el proceso de reflexión (*feedback*):**
 - Permiten comprobar la correcta realización de cada fase determinando si verifican los modelos.

1.1. Problemas del análisis clásico

No todas las técnicas de análisis logran estos objetivos. Por ejemplo, una descripción del sistema de 500 páginas ocultará características del sistema, tendrá un desarrollo enormemente costoso, y será difícil de modificar, entre otros motivos.

Esto se ve claramente reflejado en el análisis de requisitos clásico, usado hasta finales de los 70. Este consistía en redactar especificaciones funcionales, en forma de documentos de texto que eran:

1. **Monolíticos:** Había que leerlos de principio a fin.
2. **Redundantes:** Lo cual suele inducir a inconsistencia incluso de querer hacer cambios.
3. **Ambiguas:** 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”.

Es por ello que la mayor parte del software de la época carece de documentación fiable.

1.2. Soluciones al análisis clásico

Como consecuencia, fueron surgiendo nuevos métodos de análisis con los que obtener especificaciones:

1. **Gráficas:** Diagramas acompañados de información textual detallada. Sirven únicamente material de referencia, y no de cuerpo principal de la especificación.
2. **Particionadas.**
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** (como si fuesen tres dimensiones del espacio):

1. **Punto de vista de los datos (dimensión de la información):** Se centra en la información que utiliza el sistema, representando el modelo de los datos usados, así como las relaciones entre ellos.
 - Diagramas Entidad–Relación (**DER**).
 - Diagramas Estructura de Datos (**DED**).
2. **Punto de vista del proceso (dimensión de la función):** Se centra en qué hace el sistema, describiendo el conjunto de operaciones que reciben flujos de datos de entrada, y los transforman en flujos de datos de salida.
 - Diagramas de Flujo de Datos (**DFD**).
 - **Especificaciones de Procesos:** Término genérico que engloba la definición de cómo un sistema transforma unas entradas en salidas mediante pseudocódigo, lenguaje natural, diagramas de flujo...
3. **Punto de vista del comportamiento (dimensión del tiempo):** Se centra en cuándo sucede algo en el sistema, modelando el sistema como una sucesión de estados o modos de funcionamiento, así como se indica cuáles son las condiciones o eventos que hacen que el sistema pase de un modo a otro.
 - Diagramas de Flujo de Control (**DFC**).
 - Especificaciones de Control.
 - Diagramas de Estados.
 - Redes de Petri.

Nota: *Todos los modelos describen el mismo sistema. Por ende, debe asegurarse la consistencia entre ellos.*

Cada sistema en particular tendrá una representación de mayor o menor importancia para cada una de estas dimensiones. Por ejemplo, un sistema de información basado en una gran base de datos, tendrá una importante componente en la dimensión de la información; un sistema de gestión en tiempo real, tendrá mayor peso en la dimensión del tiempo; y, un sistema

de simulación, hará mayor énfasis en la dimensión de la función.

Al final cada modelo se centra en un número limitado de aspectos del sistema, dejando de lado otros, y será necesario combinar todos los que se consideren necesarios para tener una visión detallada de todas las características del sistema.

Por otra parte, la relación entre los diagramas de cada dimensión se establece con las técnicas que representan los planos formados por cada dos dimensiones. En la tabla siguiente se representa una posible clasificación de las técnicas de modelado según su dimensión; las que tengan la misma fila y columna, se centrarán en una dimensión, mientras que las otras harán referencia al plano formado por las dimensiones indicadas por su fila y columna.

	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	D. de historia y 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: Diferentes técnicas de modelado

Dichas técnicas buscan modelar a alto nivel el sistema en cada una de sus dimensiones. Las técnicas siguientes dan el máximo nivel de detalle posible de aquel aspecto que representan, por lo que se presentan como técnicas de especificación.

	Información	Función	Tiempo
Información	Especificación de entidad		
Función		diccionario de datos Especificación de procesos Especificación de entidades externas	
Tiempo		Definición de función	Especificación de eventos

Tabla 4: Diferentes técnicas de especificación

2.1. Diagramas de flujo de datos (DFD)

Surgen como respuesta a la necesidad de describir:

- **Qué funciones** son las que realiza el sistema.
- **Qué interacción** se produce entre estas funciones: A dónde va la información de salida de una determinada función.
- **Qué transformaciones de datos** realiza el sistema, especificando **qué datos de entrada se transforman en qué datos de salida**: Qué se convierte en qué, y por dónde va.

Así, el **DFD** representa el flujo de información y las transformaciones que se aplican a los datos al moverse desde la entrada a la salida. Dicho de otro modo, se representa el sistema desde un **punto de vista funcional**; esto es, las **entidades básicas son funciones o procesos** que transforman unos datos de entrada en salidas, y el sistema resulta ser el flujo de información a través de estas funciones.

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

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

2.1.1. Elementos

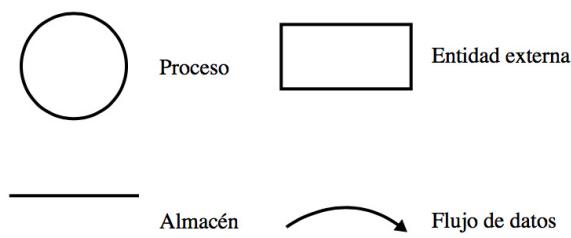


Figura 16: Notación de Yourdoun utilizada para representar los elementos del DFD.

1. **Procesos:** Representan elementos software que transforman información; por ende, son los componentes del software que realizan cada una de las funciones del sistema. Deben verificar las siguientes reglas:
 - **De conservación de datos:** El proceso debe ser capaz de generar las salidas a partir de los flujos de entrada más una información local.
 - **Pérdida de información:** Todas las entradas del proceso tienen que emplearse para calcular alguna de sus salidas.
2. **Entidades externas:** Representan elementos del sistema informático o de otros sistemas adyacentes (en todo caso, fuera del los límites del sistema software) que producen información que será transformada por el software, o consumen la resultante. Los flujos de datos que comunican el sistema con las entidades externas representan las interfaces del sistema. **Sólo aparecen en el diagrama de contexto**, y los flujos entre unidades externas no son objeto de estudio.
3. **Almacenes de datos:** Representan información almacenada que puede ser utilizada por el software, dado que permiten guardar temporalmente información que será procesada, o bien por el mismo proceso que la creó, o bien por otro distinto; en la mayoría de los casos, utilizaremos almacenes de datos cuando los procesos intercambien información pero no estén sincronizados. Se pueden representar múltiples veces para una mayor legibilidad, y no pueden existir comunicaciones entre almacenes.
4. **Flujos de datos:** Representan datos o colecciones de datos que fluyen a través del sistema, conectando los procesos con otros procesos, con entidades externas o con almacenes de datos. Posiblemente, en los diagramas de nivel mayor existirán flujos de datos bidireccionales (par de diálogo), o incluso varios flujos de datos agrupados en uno

solo (flujos múltiples) que serán refinados en sucesivos diagramas. Contienen información de las tres dimensiones, aunque normalmente sólo representan las dimensiones de Función e Información. Según su **dimensión temporal**, los flujos pueden ser:

- Discretos** (*flecha con una cabeza*): Representan movimiento de datos en un instante determinado de tiempo.
- Continuos** (*flecha con dos cabezas*): Implican una transmisión continua de información; *por ejemplo, comprobar si un registro ha cambiado*.

Nota: La comunicación entre almacenes y entidades externas solo se representará excepcionalmente cuando el almacén haga de interfaz con la entidad.

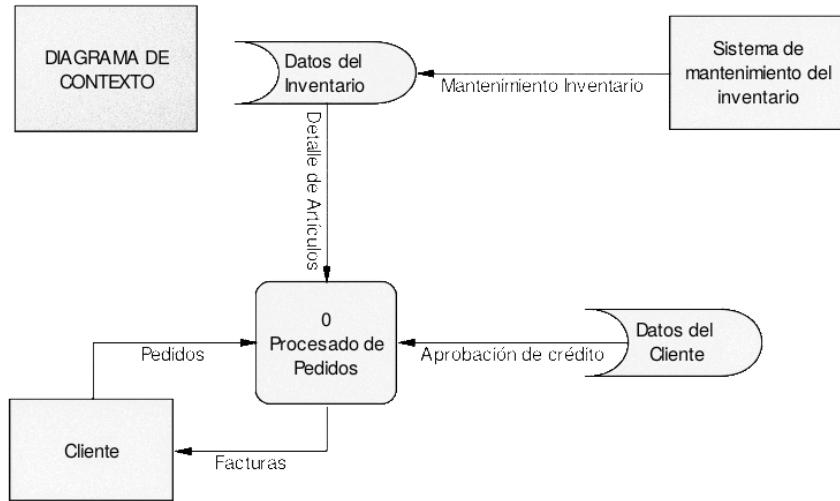


Figura 17: Ejemplo de un Diagrama de Flujo de Datos en el que un almacén actúa de interfaz con una entidad.

Cualquiera de los elementos de un DFD tiene que estar etiquetado con un nombre significativo y único:

- Los procesos y entidades externas se etiquetan con la función que realizan.
- Los flujos de datos se etiquetan con un nombre identificativo de la información, e incluso su estado. *Por ejemplo, número de teléfono, número de teléfono correcto, número de teléfono incorrecto....*
- Los almacenes de datos se etiquetan con un nombre significativo de la información contenida, generalmente en plural.

2.1.2. Niveles

Los **DFD** permiten la representación del sistema en múltiples niveles de abstracción, de modo que se pueden representar gráficos con un número reducido de procesos (máximo 7 ± 2), y con un máximo de 7-8 niveles. Cabe distinguir:

1. Nivel 0: Diagrama de Contexto.

Representa el sistema como un único proceso, el **proceso 0**, que realiza la función principal del sistema, como si de una caja negra se tratase. Se representan también las entidades que interactúan con el mismo, dejando claro los **límites del sistema**, así

como sus **interfaces**.

A partir de este diagrama de contexto, se pueden ir construyendo nuevos diagramas que vayan definiendo con mayor nivel de detalle el sistema ya que, en general proceso que aparezca en un DFD puede ser descrito más detalladamente con un nuevo DFD; a este procedimiento se le llama explosión de un proceso. Así el proceso aparece descompuesto en una serie de subprocessos o subsistemas. Los flujos de datos que entraban y salían del proceso deben hacerlo también del DFD que lo desarrolla, y este contendrá por lo general nuevos flujos que comunican los procesos que figuren en él, y posiblemente almacenes de datos.

Nota: *Las entidades externas solo aparecen en el DFD de contexto.*

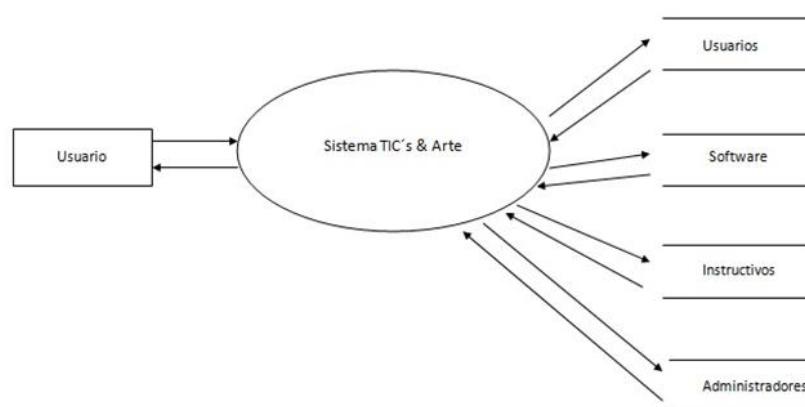


Figura 18: DFD de nivel 0 de un sistema.

2. Nivel 1: Diagrama 0 o de Sistema.

Denominado así por *descomponer el proceso 0* en sus funciones principales; es interesante que éstas sean *independientes* y que procesen *los mismos flujos* que el de nivel 0.

3. Niveles 2...N-1

Se continúa descomponiendo los procesos en subprocessos, recogiendo las interfaces (flujos) de nivel superior y asignándoselas a subprocessos.

4. Nivel N

Contiene los procesos primitivos, que no se pueden descomponer. Para describirlos, deben emplearse técnicas de especificación de procesos.

5. Procesos primitivos

Nota: *Como se puede observar, un DFD puede dar lugar a tantos DFDs como procesos contiene.*

La descomposición de un sistema en diferentes niveles (jerarquía de DFDs) debe respetar la regla del balanceo. Sus restricciones pueden separarse en dos partes.

En primer lugar, los procesos en el DFD tienen un **identificador numérico** de modo que:

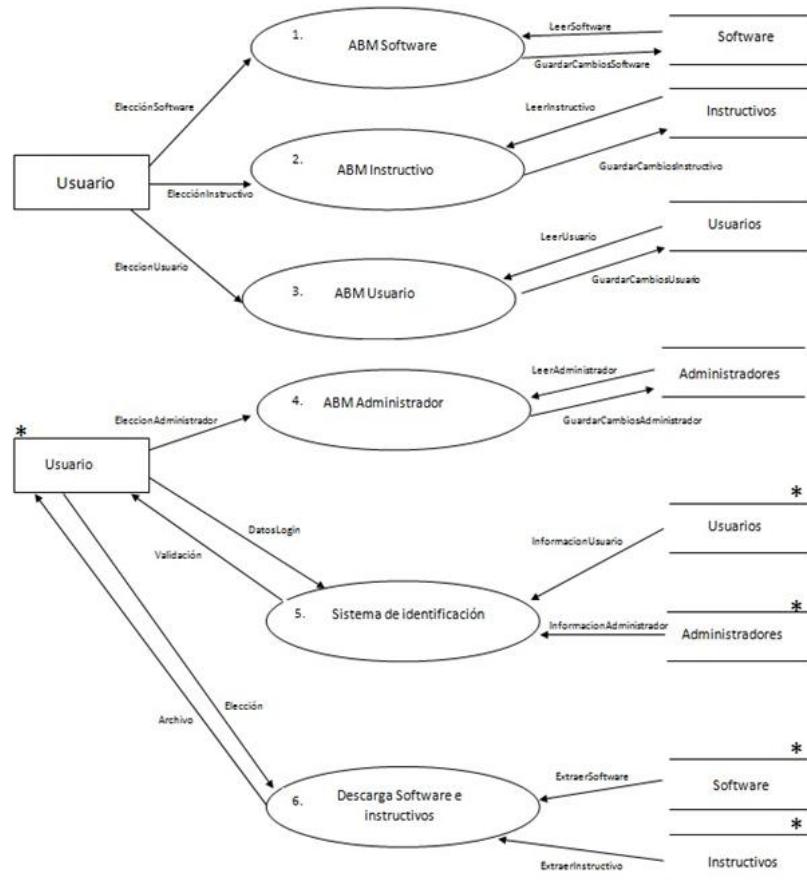


Figura 19: DFD de nivel 1 de un sistema.

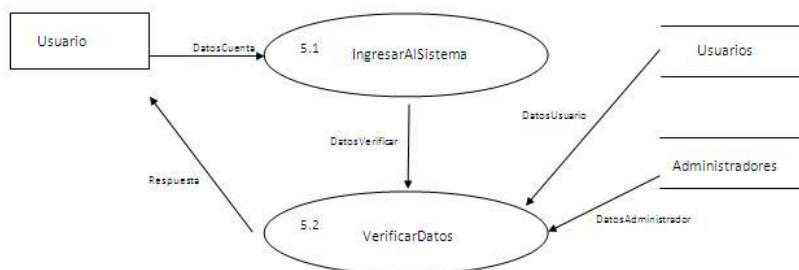


Figura 20: DFD de nivel 2 del proceso 5 del sistema.

- Nivel 0: Solo contiene un proceso, el “0”
- Nivel 1: Se le asigna a cada proceso un número secuencialmente, {1, 2, 3...}
- Niveles 2...N: La numeración de los procesos en el DFD hijo se deriva del número del DFD padre; por ejemplo, para el proceso 3, serían 3.1, 3.2...

Por otra parte:

- El título del DFD hijo debe ser el nombre del proceso que desarrolla.

- Debe mantenerse la consistencia de los flujos de datos entre los DFDs padre e hijo. En el diagrama hijo, estos flujos tendrán un extremo libre, puesto que conectan el proceso padre con algún elemento que ya no va a estar representado; una excepción son los flujos que conectan los procesos con los almacenes de datos.

De todos modos, esta regla del balanceo puede contener excepciones. Un ejemplo de ello lo son los flujos de datos bidireccionales, y los compuestos, dado que cada uno de estos deberá descomponerse en algún momento en varios flujos.

2.2. Especificaciones de proceso (PSPEC)

Los DFDs no indican nada acerca de los detalles de cómo se realizan las transformaciones en el sistema. Es aquí en donde entra en juego el **Process Specification**, que es un documento que describe textualmente los detalles de un proceso, indicando cómo es el procedimiento de transformación de una información de entrada en otra de salida; dicho de otro modo, debe definir de forma más o menos formal cómo se obtienen los flujos de salida a partir de los de entrada más una información local.

Para generar estos documentos, los procesos de los DFDs de menor nivel (más altos en la jerarquía) se irán describiendo mediante nuevos DFDs, hasta alcanzar un nivel en el que un proceso pueda ser descrito textualmente de forma sencilla y no ambigua; estos procesos suelen llamarse **procesos primitivos**. De este modo, las PSPEC servirán para complementar los DFDs, tal y como puede verse.

Las descripciones de estos procesos deben:

- Ser **breves** (menos de una página).
- Incluir **precondiciones y postcondiciones**.

Y, para su elaboración, se utilizan diferentes técnicas de especificación:

1. Lenguaje natural; *por ejemplo, mediante una definición como “invertir una matriz”.*
2. Diagramas de flujo.
3. Lenguaje estructurado, y diagramas de acción (pseudocódigo, especificar un algoritmo).
4. Árboles de decisión.
5. Tablas de decisión.

2.3. Estrategia de creación de DFDs y PSPECs

1. Diagrama de contexto:

- a) Localizar todas las entidades.
- b) Definir sus flujos con precisión (interfaces).

2. Diagrama de sistema:

- a) Seleccionar las funciones principales.
- b) Definir los flujos entre dichas funciones; normalmente, a través de almacenes.

CONDICIONES	Sí	Sí	No	No	No
Condición 1	Sí	-	-	-	-
Condición 2	-	Sí	-	-	-
Condición 3	-	-	Sí	-	-
Condición 4	-	-	-	Sí	-
Condición 5	-	-	-	-	Sí
Condición 6	-	-	-	-	-

ACCIONES	X	X	X	X	X
Acción 1	X				
Acción 2		X			
Acción 3				X	

• TABLA DE DECISIÓN

• ÁRBOL DE DECISIÓN

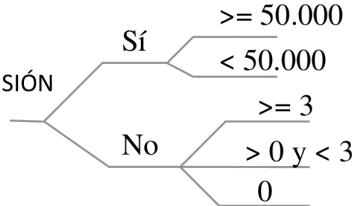


Figura 21: Ejemplos de dos PSPEC mediante (1) una tabla de decisión y (2) un árbol de decisión.

- c) Recoger los flujos del diagrama de contexto; normalmente, cada uno entrará en un proceso distinto, y no conviene dividir los aunque sean flujos múltiples.

3. Resto de diagramas:

- a) No descomponer al máximo.
- b) Subfunciones principales de cada proceso, e interfaces entre sus procesos resultantes.
- c) Se recogen los interfaces (flujos) de nivel superior y se asignan a alguno de los procesos.
- d) Se pueden desglosar flujos múltiples.

4. Procesos primitivos: Detenerse cuando:

- a) La función puede expresarse en una página.
- b) Los procesos tienen pocos flujos de E/S.
- c) Descomponer implica perder el significado de la función (diagramas demasiado sencillos que complican la compresión global).

2.4. Diagramas de flujo de control (DFC)

Dado que en los DFDs no se representa explícitamente el control o flujo de sucesos del sistema (cuándo se realizan los procesos o en qué orden), hay determinados sistemas de software en donde se podrá intuir, pero habrá otros en que no será posible. A la hora de **interpretar el comportamiento de un sistema**, podemos encontrarnos con **tres situaciones posibles**:

- **Los procesos que figuran en el DFD están activos siempre.** Por ende, no es necesario especificar el control del sistema.
- **Los procesos se activan cuando llegan datos a través de los flujos de entrada,** los transforman y emiten los resultados a través de los flujos de salida, **permaneciendo inactivos hasta la llegada de nuevos datos.** Este comportamiento está implícito en la notación usada.

- **Cada proceso pasa por períodos de actividad e inactividad**, gobernados por mecanismos más complejos. Este tipo de situaciones no pueden representarse debidamente en un DFD, por lo que debe crearse un modelo adicional en donde se representen las señales de control no implícitas en el DFD.

Nota: *El modelo de control del sistema establece el comportamiento de éste a alto nivel, indicando qué procesos están activos en cada momento o en qué secuencia se realiza la transformación de los datos. Existe un control de bajo nivel referido a los saltos condicionales y a los bucles, que estaría reflejado en las PSPECs de los procesos.*

Para modelar el control del sistema, se emplearán los **Diagramas de Flujo de Control (DFC)**, los cuales consisten en eliminar de los DFDs todo lo relativo a la información de control, y construir una jerarquía de DFCs paralela a la de DFDs. Según esto, se comenzará por el diagrama de contexto, representando en cada par DFD/DFC los mismos procesos y entidades externas, puesto que representan modelos de la misma parte del sistema con el mismo nivel de detalle, aunque con puntos de vista distintos.

2.4.1. Elementos

Las reglas sobre denominación de numeración, relaciones padre-hijo, y balanceo que se aplican a los DFCs son las mismas que se establecen para los DFDs.

Por otra parte, los elementos que aparecen en un DFC son prácticamente los mismos:

- **Procesos, entidades externas, y almacenes de datos:** Serán los mismos y tendrán el mismo significado que en el DFD.
- **Flujos de control:** Se representan mediante trazos discontinuos y modelan el flujo de información de control en el sistema. Habrá procesos o entidades externas que generen información de control, y otras que la consuman. Aunque, en general, los flujos entrantes sirven para activar un proceso, también se utilizan para transformarlos en flujos de salida según la PSPEC correspondiente.
- **Condiciones de datos:** Flujos de control generados por un proceso del sistema. Figuran en el DFC y no en el DFD, pero es la PSPEC la que define cómo se calcula su valor a partir de los flujos de entrada del proceso.
- **Almacenes de control:** Se representan igual que los almacenes de datos, pero con trazos discontinuos. Permiten almacenar información de control para ser utilizada posteriormente.
- **Ventanas de especificaciones de control:** Se representan mediante barras, y reciben flujos de control (condiciones de datos junto con flujos de control del exterior), así como los emiten, actuando de este modo como las transformaciones de flujos de control en el sistema; dicho de otro modo, dedican el comportamiento del sistema. Su comportamiento se define en las especificaciones de control (CSPEC). No están etiquetadas, puesto que todas las que aparecen en un determinado DFC hace referencia a una única CSPEC que define cómo se realizan estas transformaciones.
- **Activadores de procesos** Flujos de control especiales que activan o desactivan procesos tomando dos posibles valores, **ON** y **OFF**. Tienen el mismo nombre que los procesos

que controlan, por lo que no suelen representarse en el DFC excepto por fines de claridad. Siguen la jerarquía de los modelos, es decir: un proceso está activo si y sólo si todos sus antecesores están activos.

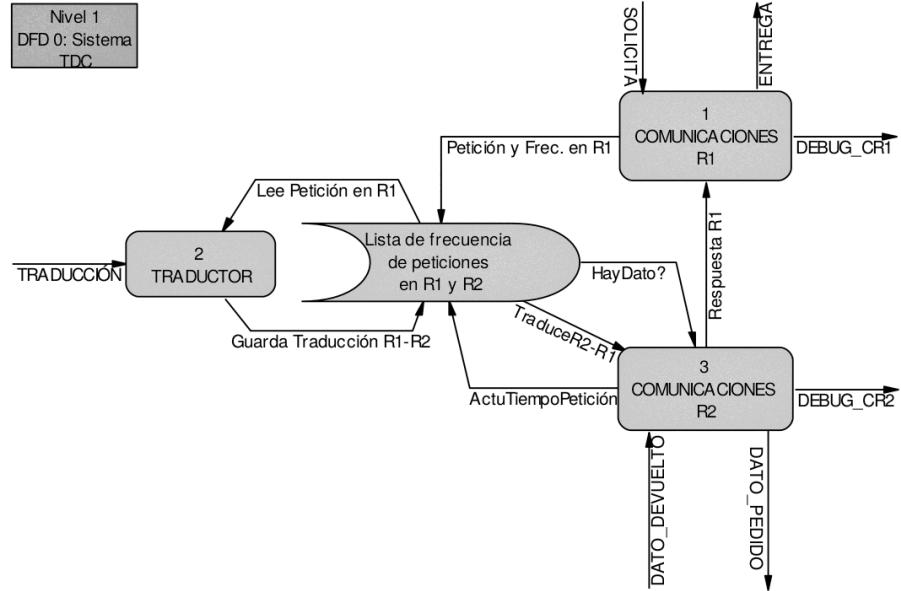


Figura 22: Ejemplo de un sistema de traducción. Nótense los flujos de control *DEBUG_CR1* y *DEBUG_CR2*, que denotan que procedían de un flujo compuesto situado en un diagrama antecesor.

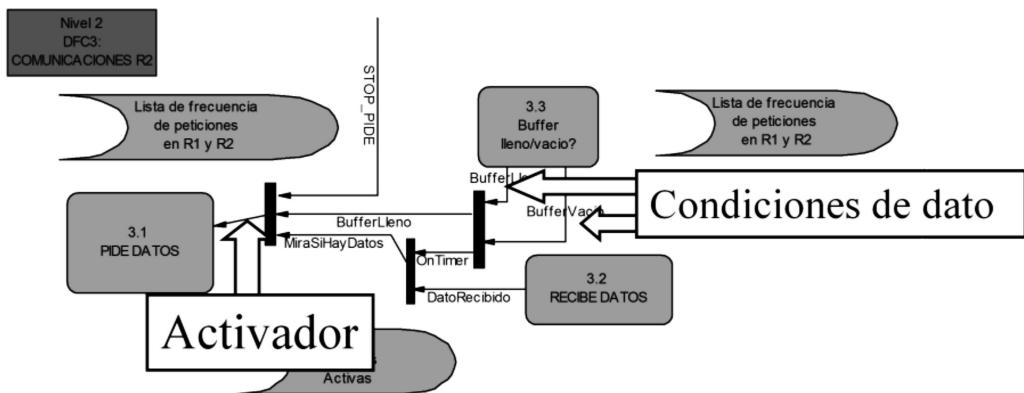


Figura 23: Ejemplo de un sistema de traducción. Préstese atención a las condiciones de dato: *buffer lleno/buffer vacío*, las ventanas de control (barras) y el activador.

Nota: Recordar que, mediante los flujos de control, se define cómo se activan o desactivan los procesos.

Nota: Dado que los modelos del sistema tienen estructura jerárquica, se considera que un proceso está activo solo si todos sus antecesores están activos. Un proceso que no tiene activador se considera, por lo tanto, activo siempre que sus antecesores lo estén; en caso de

tener activador, éste debe tenerse también en cuenta.

Nota: Los procesos en un DFC indican cómo fluyen los flujos de control a través de ellos. No representan los estados del sistema (se encargan los DEs), y tampoco representan procesamiento ni transformación de los flujos de control (se encargan las CSPECs).

Nota: Los DFCs reflejan la información de control que existe en el sistema, y qué procesos y entidades las producen y consumen. Deben ser combiandos con las CSPECs para reflejar el comportamiento del sistema.

Otro detalle importante es que un flujo de control que entra en un proceso no indica necesariamente que ese proceso se active; la activación y desactivación de procesos se indica en las CSPECs. Por lo tanto, un flujo de control entrante puede indicar que:

- O bien va a ser utilizado como un dato más para que el proceso lleve a cabo su transformación.
- O bien va a ser utilizado para controlar alguno de los procesos en los que se descompone el proceso.

2.4.2. Cómo separar datos y control

No existen unas normas estrictas para separar, de entre todos los flujos de información, los datos del control. Por norma general, **se modelarán como señales de control únicamente aquellas que intervengan en la activación y desactivación de algunos de los procesos del sistema de forma no trivial**, mientras que el resto de la información se modelará como datos.

En determinadas situaciones, es posible que un elemento de información determinado se emplee como dato en un proceso y como control en otro. Para ello, se modelará como dato (trazo continuo) en el DFD, y como control (trazo discontinuo) en el DFC, pero se asignará el mismo nombre a ambos flujos para mostrar que son el mismo.

2.4.3. Cuándo usar especificaciones de control

Lo ideal es utilizar DFDs siempre que sea posible, puesto que son más sencillos de realizar y de entender; por ejemplo, el uso de DFCs obligará a mirar en paralelo a los DFDs.

En todo caso, debe recordarse siempre que es necesario centrarse en el **modelo abstracto o lógico** del sistema, mientras que las especificaciones ya tratarán los aspectos de más bajo nivel.

2.5. Especificaciones de control (CSPEC)

Son similares a las PSPECs, ya que en ambos casos se definen los detalles procedimentales de cómo se realiza el procesamiento de los flujos de entrada y salida. Sin embargo, hay una diferencia importante entre PSPECs y CSPECs:

- Las PSPECs se utilizan para describir las primitivas de proceso.
- Las CSPECs sirven para modelar el comportamiento de un DFC, describiendo cómo se procesan los flujos de control. Por lo tanto, habrá, como máximo, una CSPEC por cada

DFC de la jerarquía, ya que el primero determina la activación del segundo, utilizando las **ventanas de control de los DFC como interfaces**.

Así, se puede concretar que una Control SPECification es un documento que **especifica el comportamiento un DFC** (y, por lo tanto, del sistema). Concretamente, se indica cómo, a partir de las señales de control que entran en una ventana, se determina la activación o desactivación de los procesos comprometidos.

Estas CSPECs pueden caracterizarse mediante:

- **Lenguaje estructurado** (pseudocódigo).
- **Sistemas combinacionales** (valor de salida es exclusivo a los valores de entrada):
 - Tablas de decisión: Indican cómo calcular las señales de salida en función de las de entrada.

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

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

- Tablas de activación de procesos: Similares a las anteriores, pero indicando para cada proceso del DFC si está activo o no ante cada combinación.
- **Sistemas secuenciales:**
 - Diagramas de Estados: Autómatas finitos.

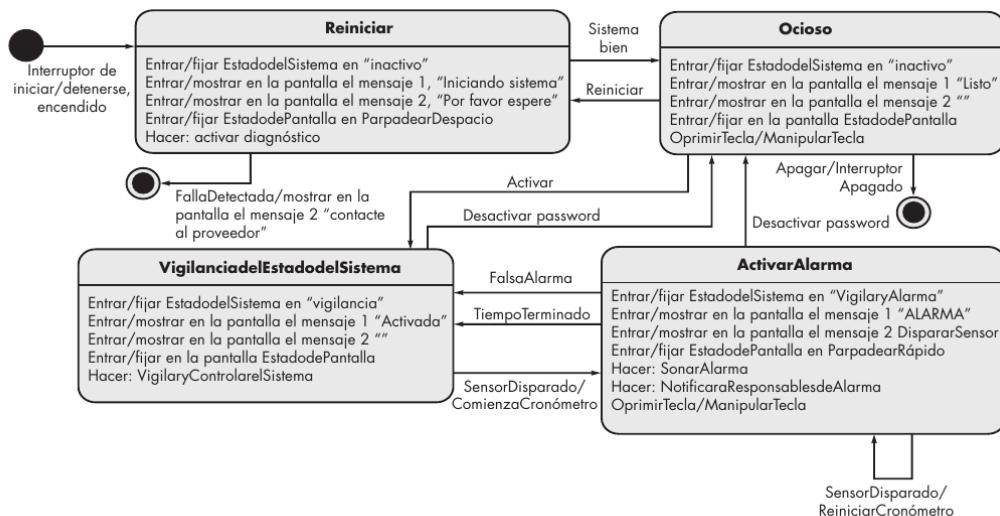


Figura 24: Ejemplo de un diagrama de estados.

- Redes de Petri: Técnica muy apropiada en el caso de tratar con sistemas de comportamiento asíncrono concurrente.

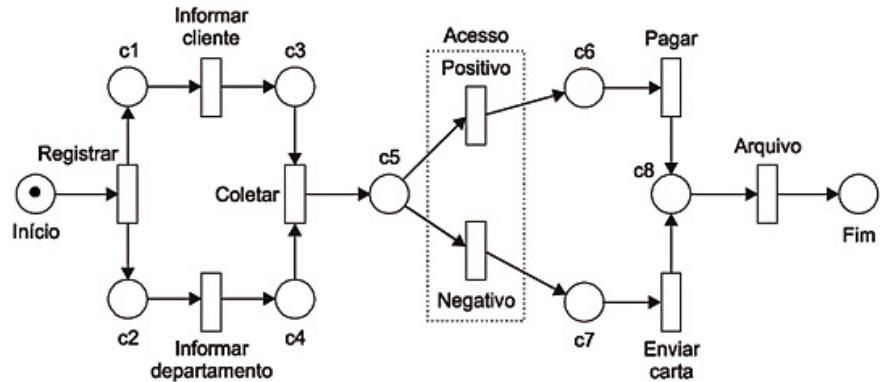


Figura 25: Proceso de generación de una reclamación modelado en redes de Petri.

2.6. Estrategia de creación de DFCs y CSPECs

1. Construir una jerarquía de DFCs paralela a la de DFDs.
2. Cada par DFC/DFD representa los mismos procesos y las mismas entidades externas.
3. Solo se introducen las señales de control que no estén implícitas en el DFD.
4. Cada DFC se desarrolla en un CSPEC.

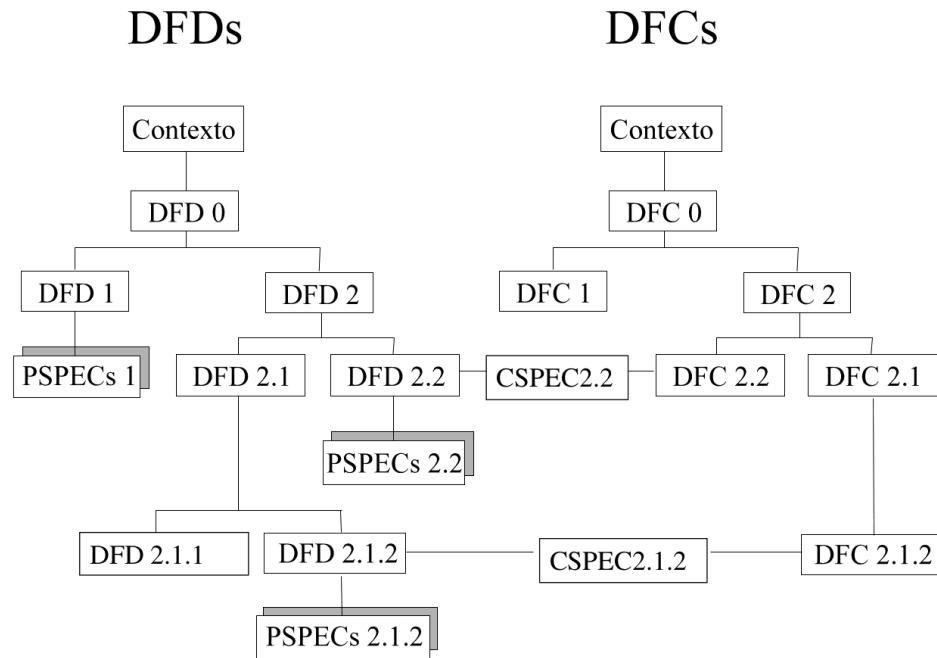


Figura 26: Ejemplificación de la jerarquía de DFCs paralela a la de DFDs.

En resumen, por cada DFD se crea un DFC gemelo si alguno de los procesos del DFD genera o consume información de control. Además, si alguno de los procesos del DFD es activado o desactivado de forma no implícita, éste DFC llevará una ventana de control y se hará una CSPEC para indicar cuándo se realiza la activación y desactivación.

2.7. Diagramas Entidad–Relación (DER)

Los **Diagramas Entidad–Relación** resuelven las insuficiencias que se pueden producir en la representación de datos complejos mediante los almacenes en los diagramas de flujo de datos (DFD). Para ello, no solo se muestra la información contenida, sino también las relaciones que existen entre los propios datos.

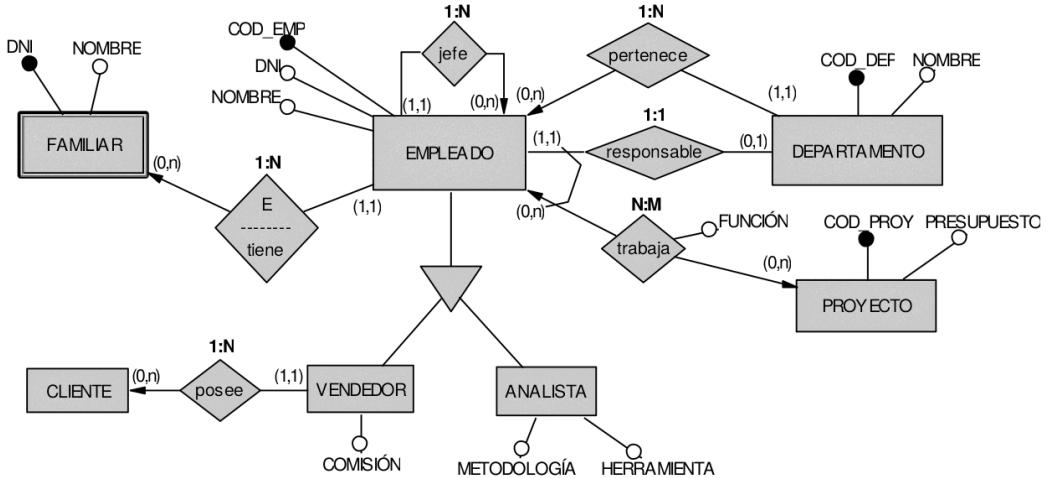


Figura 27: Ejemplificación de un Diagrama Entidad–Relación.

2.8. Comprobaciones

Una vez esté realizada la especificación estructurada, debe comprobarse si cumple las siguientes propiedades:

- Compleción:** Los modelos son completos.
- Integridad:** No existen contradicciones ni incoherencias entre modelos.
- Exactitud:** Los modelos cumplen los requisitos del usuario.
- Calidad:** Estilo, legibilidad y facilidad de mantenimiento.

Nota: Es recomendable realizar las comprobaciones elaborando una checklist en la que se comprueben, con profundidad, aspectos sobre todas estas propiedades.

2.9. Consistencia entre modelos

El conjunto de modelos tiene como misión dar una visión global del sistema, desde diversos puntos de vista; por lo tanto, representan siempre el mismo sistema, así que deben ser consistentes.

Las técnicas matriciales se utilizan **principalmente para ayudar a verificar la consistencia entre los componentes de distintos modelos** de un sistema, ya sean centrados en la dimensión de las funciones, en la de la información, o en la temporal.

- Matriz Entidad/Función:** 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 DER, mientras que los de las columnas pueden ser funciones de alto nivel representadas en un DFD.

En cada celda se incluyen las **acciones** que puede realizar la función (**Insertar**, **Leer**, **Modificar** y **Borrar**).

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

Tabla 6: Ejemplo de matriz Entidad/Función.

2. **Matriz Entidad/Entidad:** Muestra las relaciones habituales del DER, indicando en cada celda la relación entre las entidades implicadas. Es de utilidad, sobre todo, cuando se presenta un número alto de entidades.

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

Tabla 7: Ejemplo de Matriz Entidad/Entidad.

3. **Matriz Evento/Entidad** (evento ≡ señal de control): Muestra las acciones que provocan los eventos sobre las entidades (datos almacenados): (**Insertar**, **Leer**, **Modificar** y **Borrar**).

		Entidades		
		Cliente	Presupuesto	...
Eventos	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

Una vez vistas todas las anteriores notaciones, se verá ahora cómo coordinarlos durante el trabajo.

3.1. Fases

3.1.1. Creación del modelo de procesos

El punto de partida será la creación del modelo de procesos del sistema, modelando el ámbito de información y el funcional mediante DFDs y PSPECs.

Inicialmente, debe revisarse toda la documentación inicial; es importante pensar antes de ponerse a dibujar diagramas. Por ejemplo, puede realizarse un **análisis gramatical** de toda esta información, para identificar muchos de los componentes del sistema:

- Las entidades externas se corresponderán con los nombres.
- Los flujos y almacenes de datos también se corresponderán con los nombres.
- Los procesos se corresponderán con los verbos.

Además, la especificación (documentación) estudiada también establecerá las relaciones entre dichos nombres y verbos de por sí. A pesar de que lo obtenido hasta el momento no sea completamente correcto, o esté incompleto, sí marca un buen punto de partida.

Tras ello, se comienza a realizar el **DFD de contexto**, y se va refinando en mayores niveles de detalle siguiendo las **reglas de construcción**. Es importante respetar las reglas de **acoplamiento mínimo** entre procesos distintos, y de **máxima cohesión** dentro de un proceso, así como mantener la **consistencia** entre los diferentes componentes de la jerarquía de DFDs. También debe recordarse el **evitar detalles de implementación**, así como **ir incluyendo en el diccionario de datos (DD) los elementos de los DFDs**.

El proceso de descomposición finalizará una vez de **encuentren los procesos primitivos** (procesos sencillos con máxima cohesión \equiv realizar una única función), y que se describirán mediante **PSPECs**. Estos últimos no contienen definiciones de datos, por lo que no pueden actuar como almacenes de datos, sino emplear tan solo variables locales.

3.1.2. Creación del modelo de control

No todos los sistemas requieren **modelos de control**, al estar esta información presente implícitamente en los DFDs. En caso de realizarse, hay que tener **especial cuidado en no caer en detalles de implementación**.

Esta actividad debe comenzar por **establecer una jerarquía de DFCs simétrica a la de DFDs**, de modo que cada par DFD/DFC contiene los mismos procesos, almacenes de datos, y entidades externas, y hasta en la misma posición, para facilitar la identificación. A continuación, se **eliminan del DFD todos los flujos que transporten información de control**, y se representarán en los DFCs.

A cada par DFD/DFC (en donde el DFD es antecesor inmediato de procesos primitivos) le corresponde una **CSPEC**; esta puede ser combinacional, secuencial o compuesta, y se escogerá siempre el modelo más sencillo posible. Es importante recordar que debe **mantenerse la consistencia de flujos entre las ventanas de control y las CSPECs**.

3.1.3. Creación del modelo de datos

Si los datos que maneja el sistema tienen una estructura compleja, no bastará con definir en el DD el contenido de cada uno. Se desarrollará en su lugar un modelo de datos empleando **DERs**, donde se muestren las **relaciones** que existan entre los datos que maneje el sistema.

3.1.4. Consistencia entre modelos

Finalmente, una vez realizados los distintos modelos del sistema, deberían llevarse a cabo todas las **técnicas de consistencia entre modelos** que se consideren necesarias.

4. Modelos del sistema

4.1. El modelo esencial

El modelo esencial del sistema (algunas veces llamado modelo lógico) representa lo que **el sistema debe hacer para satisfacer los requisitos del usuario**. Es, por lo tanto, un modelo abstracto que supone que disponemos de una tecnología perfecta sin coste alguno.

Este modelo es el **resultado de la fase de análisis de requisitos del sistema**, por lo que debe estar completamente libre de detalles de implementación. Algunos errores típicos al realizarlo son:

- **Secuenciar los procesos del DFD:** Deben ser lo más concurrentes posibles, en lugar de pensar que el sistema realiza las tareas una detrás de otra. El único secuenciamiento que debe aparecer en los DFDs es por la dependencia de datos; cualquier otra secuencia es puramente arbitraria, y dependerá más de necesidades de implementación.
- **Utilizar ficheros temporales o de backup:** Estos se usan para conectar procesos que no pueden ejecutarse simultáneamente por limitaciones de la capacidad del hardware, y no porque deban ejecutarse de forma secuencial. Pero, en este modelo esencial, estamos suponiendo que contamos con una tecnología perfecta con la que evitar este tipo de limitaciones, por lo que no se requieren estos ficheros.
- **Utilizar información redundante o derivada:** En el modelo esencial no se usará nunca información redundante o derivada, ya que tiene más que ver con la eficiencia de implementación que con los requisitos del modelo. *Por ejemplo, un error sería almacenar en una base de datos un dato que podría declararse como derivado, solo por evitar calcularlo cada vez que se requiera.*

A partir del modelo esencial, los diseñadores podrán decidir cómo implementar el sistema, usando la tecnología disponible.

4.2. Modelo de implementación

Sin embargo, lo normal es que el cliente proporcione más información que los requisitos del sistema, y que decida también sobre detalles de implementación: qué funciones serán manuales y cuáles automáticas, el formato de los datos de salida, etc. Toda esta información no se refiere a los requisitos esenciales del sistema, sino a los requisitos de implementación, que formarán parte del modelo de implementación del sistema.

El desarrollo de este modelo es una tarea que está a **medio camino entre el análisis y el diseño**. No puede ser desarrollado solo por el analista y el cliente, dado que se necesita consejo de diseñadores e implementadores ya que conocen la tecnología disponible. Por otra parte, no puede ser realizado únicamente por diseñadores e implementadores, porque el usuario debe definir una gran cantidad de requisitos de información, y es el analista quien los describe, haciendo de vínculo entre el cliente y el equipo de desarrollo.

En definitiva, este modelo será una **versión revisada y anotada del modelo esencial, especificando todos esos detalles adicionales**. Sobre ellos, se puede destacar:

- La elección de dispositivos de entrada y salida.
- La elección de los dispositivos de almacenamiento.

- El formato de las entradas y las salidas.
- La secuencia de operaciones de entrada y salida, incluyendo la definición de cómo será el diálogo con el usuario.
- El volumen de datos esperado.
- El tiempo de respuesta requerido.
- La realización de copias de seguridad, y las posibilidades de exportar bajo demanda datos contenidos en el sistema.
- La seguridad.

Tema VI

Pruebas del software

1. Introducción

Una de las características típicas de los ciclos de vida en el desarrollo de software es la realización de controles periódicos, con el objetivo de evaluar la calidad de los productos generados, y poder **detectar así fallos en ellos cuanto antes**. Aún así, todo sistema/aplicación debe ser **probado**, independientemente de estas revisiones, mediante su **ejecución controlada antes de ser entregado al cliente**; estas se denominan habitualmente como pruebas.

Las pruebas permiten verificar y validar el software; cabe recordar el significado de estos dos términos:

- **Verificación** (pruebas de caja blanca): Proceso de evaluación de un sistema o de uno de sus componentes para determinar si los productos de la fase sobre el que se realiza satisfacen las condiciones impuestas al principio de esta fase. Dicho de otro modo, se puede decir que se pregunta si se está construyendo correctamente el producto.
- **Validación** (pruebas de caja negra): Proceso de evaluación del sistema o de uno de sus componentes para determinar, bien sea durante o al final del desarrollo, si satisface los requisitos especificados. Dicho de otro modo, se puede decir que se pregunta si se está construyendo el producto correcto.

Como se puede ver, el contar con la posibilidad de enfocar las pruebas hacia cualquiera de estos dos puntos de vista permite enfrentar cualquier fase del ciclo de vida a:

- La metodología de la propia fase: Verificación.
- Los requisitos: Validación.

2. Definiciones

El *IEEE* propone, entre otras, las siguientes definiciones en relación a las pruebas.

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 determinado de antemano. *Proceso de ejecutar un programa con el fin de encontrar errores*.

Una prueba consta de uno o más casos de prueba.

Caso de prueba Conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular. *Por ejemplo: ejercitarse un camino concreto de un programa, o verificar el cumplimiento de un determinado requisito*.

Por extensión, un caso de prueba válido debe comprobar que se deja hacer lo que se debería, y que no se permite hacer lo que no. Además, un buen caso de prueba es aquel que tiene una alta probabilidad de detectar un error.

Nota: Por ejemplo, una prueba del módulo de menús de un software para comedores constaría de varios casos de prueba; estos podrían ser que un jefe de una cafetería pueda {insertar, borrar, modificar} un menú.

Defecto Incorrección en el software que **genera un fallo**. Por ejemplo: un proceso, una definición de datos o un paso de procesamiento incorrectos en un programa.

Fallo Incapacidad de un sistema o de alguno de sus componentes para realizar las funciones requeridas dentro de los requisitos de rendimiento especificados. Es el hecho de que el software no pueda funcionar.

Nota: Un fallo causado por un defecto es culpa de los desarrolladores, mientras que un fallo sin defecto es culpa del entorno, al poder presentar características como ser altamente cambiante. Por este motivo, es importante concretar un entorno para las pruebas, así como replicar las condiciones en las que el software será explotado.

Error Presenta varias acepciones:

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



Figura 28: Recuerda: error, defecto y fallo no son lo mismo. Al final, se provocan efectos negativos.

3. Filosofía de las pruebas del software

Las características especiales del software, como que carezca de leyes que rijan su comportamiento, o presentar una gran complejidad habitualmente, hacen aún más difícil la tarea de probarlo, de modo que la **prueba exhaustiva del software es impracticable**: no se pueden probar todas las posibilidades incluso en programas pequeños y sencillos. Además, esta situación se ve empeorada por la presencia de prejuicios y por la habitual falta de tiempo

en el desarrollo del proyecto. Por ello, es interesante estudiar cuál es la mejor actitud a seguir en esta actividad; en general, será interesante sistematizar todo lo posible de las pruebas, para ahorrar en tiempo y recursos.

Es necesario, en primer lugar, un fuerte cambio de mentalidad, dejando atrás la visión constructiva del desarrollo del software, para dar paso a la visión destructiva de intentar tirar abajo lo construido. Al final el objetivo de las pruebas, desde el punto de vista del proyecto, es la detección de defectos en el software, por lo que **el descubrir un defecto debe considerarse como el éxito de una prueba**.

En segundo lugar, también es necesario tener presente que todo el mundo comete errores, por lo que no hay que sentirse culpable por haber encontrado una metedura de pata; desde luego, es mejor que el desarrollador se encuentre con un fallo, en lugar de que tenga que ser reportado por el cliente. Es más, los defectos no son siempre el resultado de la negligencia, sino que influyen múltiples factores en su aparición, como la mala comunicación entre los miembros del equipo, que da lugar a malentendidos.

Además del objetivo mencionado anteriormente, también cabe destacar que el objetivo de las pruebas, pero ahora desde el punto de vista de ser un proceso, es el **diseñar técnicas que permitan un desarrollo sistemático de pruebas** para garantizar ese primer objetivo; es en esta visión en la que se centrará el documento.

Finalmente, la realización de pruebas en el software también cuenta con determinadas ventajas secundarias:

- **Demuestran hasta qué punto se verifican los requisitos:** No siempre va a ser posible satisfacer todos; por ejemplo, es posible que queden en “segundo plano” los requisitos estimulantes (quedarían bien), o incluso podrían verse afectados requisitos más relevantes por limitaciones de tiempo y de recursos, agravados aún más en el caso de que se produzcan cambios en los requisitos durante el desarrollo del proyecto.
- **Se generan datos de prueba que informan sobre la fiabilidad del software.**

Nota: *La realización de pruebas no asegura la ausencia de fallos; pero, el descubrimiento de un defecto, significa un éxito para la mejora de la calidad.*

Nota: *A veces, no será posible subsanar todos los errores encontrados por limitaciones de tiempo y de recursos. En estos casos, es necesario discernir entre qué errores se corregirán y cuáles no; por ejemplo, no se corrregirán aquellos que fallen en menos del 10 % de los tests.*

3.1. Principios

Los seis primeros fueron enunciados por *Davis*, y los restantes por *Myers*; a estos últimos, es además necesario incluir de nuevo el principio de Pareto, y que el grupo de pruebas deba ser independiente del de programación.

1. **A todas las pruebas se les debería poder hacer un seguimiento hasta los requisitos del cliente.** Se logra mediante la trazabilidad:

- Trazabilidad hacia arriba: Si se detecta un fallo, es posible identificar a qué requisitos afecta para evaluar si compensa o no solventarlo.

- Trazabilidad hacia abajo: Si se produce un cambio en un requisito, qué pruebas es necesario cambiar para poder validar este nuevo requisito.
2. **Las pruebas deberían planificarse mucho antes de que empiecen, y ser repetibles** (probar → corregir, probar → corregir...): La planificación de pruebas puede comenzar tan pronto como esté completo el modelo de requisitos. La definición detallada de los casos de prueba puede empezar tan pronto como el modelo de diseño se haya consolidado. Como se puede ver, es posible planificar y diseñar algunas pruebas incluso antes de generar ningún código.
 3. **El 80 % de los errores surgen al hacer el seguimiento del 20 % de los módulos del Software (Principio de Pareto)**: Conviene aislar los módulos sospechosos.
 4. **Las pruebas tendrían que hacerse de lo pequeño hacia lo grande**: En caso contrario, será difícil concretar el origen de los problemas.
 5. **No son posibles pruebas exhaustivas, ni probar todo al mismo tiempo**. 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**: El haber sido el responsable de la construcción del software probablemente lleve a realizar pruebas menos rigurosas, así como es habitual que condiciones que se olvidaron al crear el programa vuelvan a ser obviadas al escribir casos de prueba (por ejemplo, que se reciba un fichero vacío).
 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**; es decir, los no documentados o diseñados sin cuidado: Como suele ser necesario probar una y otra vez el software, el no documentar o no guardar los casos significa repetir constantemente el diseño de casos de prueba. *Por ejemplo, se incluirían aquí las pruebas tecleadas sobre la marcha.*
 12. **No deben hacerse casos de prueba suponiendo que no hay defectos en los programas**: Dicho de otro modo, no deben dedicarse pocos recursos a las pruebas.
 13. **Las pruebas son una tarea tanto o más creativa que el desarrollo de software**: Dado que no existen técnicas rutinarias para concebirlas.

Una vez visto todo esto, se puede concluir que la filosofía más adecuada consiste en planificar y diseñar las pruebas de forma sistemática para poder detectar el máximo número y variedad de defectos con el mínimo consumo de tiempo y esfuerzo. Es también necesario recordar que un buen caso de prueba es aquel que tiene una gran probabilidad de encontrar un defecto no descubierto aún, y que el éxito de una prueba consiste en detectar un defecto no encontrado antes.

4. El proceso de prueba

En la siguiente figura, se puede ver una representación del proceso completo relacionado con las pruebas basada, en parte en el estándar, y en parte en *Pressman*. En este documento, no se hará más énfasis en él.

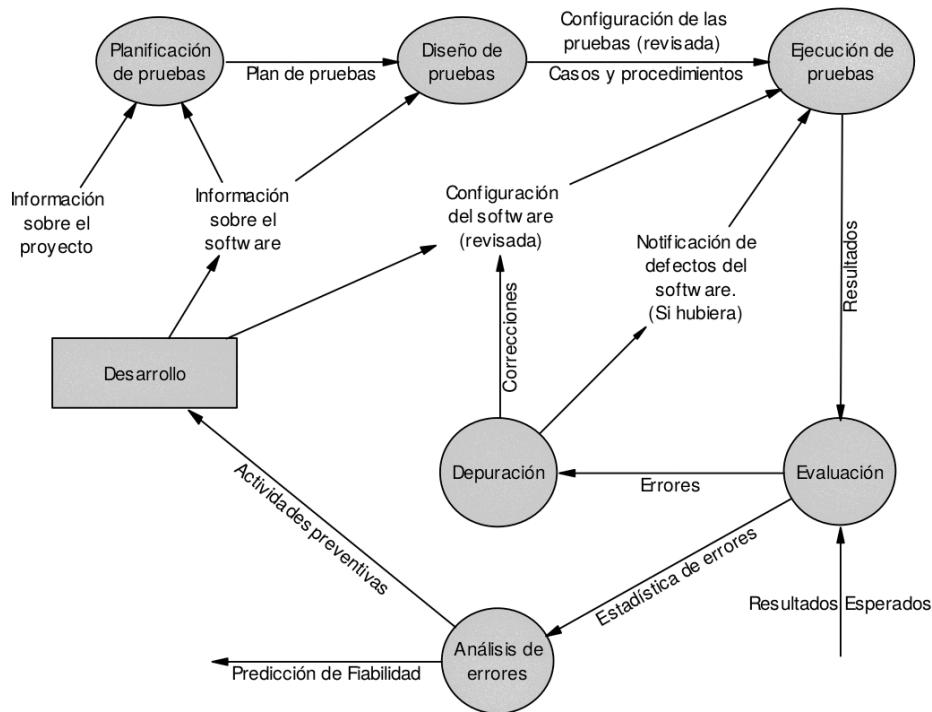


Figura 29: Proceso relacionado con las pruebas según el estándar y según *Pressman*. Las flechas entrantes en cada actividad representan información de entrada requerida para su realización.

5. Técnicas de diseño de casos de prueba

Como se comentó anteriormente, el diseño de casos de prueba está totalmente condicionado por la imposibilidad de probar exhaustivamente el software. Por ello, las técnicas de diseño de casos de prueba tienen como objetivo **conseguir una confianza aceptable en la que se detectarán los efectos existentes**, al no ser obtenible la seguridad total, **sin consumir una cantidad excesiva de recursos**. Alcanzar un equilibrio entre dicha confianza y el uso de recursos no es sencillo cuanto menos.

La idea fundamental para el diseño de casos de prueba es **elegir algunas de ellas que, por sus características, se consideren representativas del resto**; de este modo, se asume que, si no se detectan defectos en el software al ejecutar dichos casos, se puede contar con un cierto nivel de confianza en que el programa no tiene defectos. Es necesario recurrir a ciertos criterios de elección para construir los mejores casos de prueba posibles, como preguntarse cómo puede fallar el software, escoger casos no redundantes, que no sean ni demasiado sencillos ni demasiado complejos...

Existen dos enfoques principales:

1. **Estructural o de caja blanca** (verificación): Consiste en centrarse en la estructura

interna (implementación) del programa, realizando pruebas que aseguren que todos los módulos o componentes internos funcionan bien y encajan correctamente. *La prueba exhaustiva consistiría en probar todos los posibles caminos de ejecución.*

2. **Funcional o de caja negra** (validación): Consiste en estudiar la especificación de las funciones (interfaz), dejando de importar cómo está implementada la aplicación. De esta especificación, se derivan los casos, probándose que cada función es completamente operativa puesto que se conoce qué es lo que debe realizar. *La prueba exhaustiva consistiría en probar todas las posibles entradas y salidas del programa.*

Cabe destacar que estos enfoques no son excluyentes, sino que **son complementarios**. Por ejemplo, podrían diseñarse inicialmente los casos de prueba de caja negra y, a la hora de diseñar los de caja blanca, cabría la posibilidad de que algunos casos de estos últimos ya estuviesen cubiertos; al final, es necesario que se recorran determinados caminos en la ejecución de las pruebas de caja negra.

6. Pruebas estructurales (de caja blanca)

Es interesante realizar pruebas de caja blanca porque:

- Los errores tipográficos son aleatorios, por lo que pueden aparecer en cualquier parte del programa, se ejecute regularmente o no.
- Se suele creer que un determinado flujo es poco probable cuando, de hecho, puede ejecutarse regularmente.
- En relación al anterior punto, en general, la probabilidad e importancia de un trozo de código suelen ser calculadas de forma muy subjetiva. *Por ejemplo, si el programador sabe que un camino probablemente se ejecute poco, es probable que le preste menos atención que a otros, incrementándose por lo tanto la probabilidad de que se encuentren errores en él.*

El diseño de casos de prueba 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 (estrechamente relacionados con los diagramas de flujo de control), con el objetivo de convertir el código en un grafo y facilitar así la visualización de sus posibles caminos.

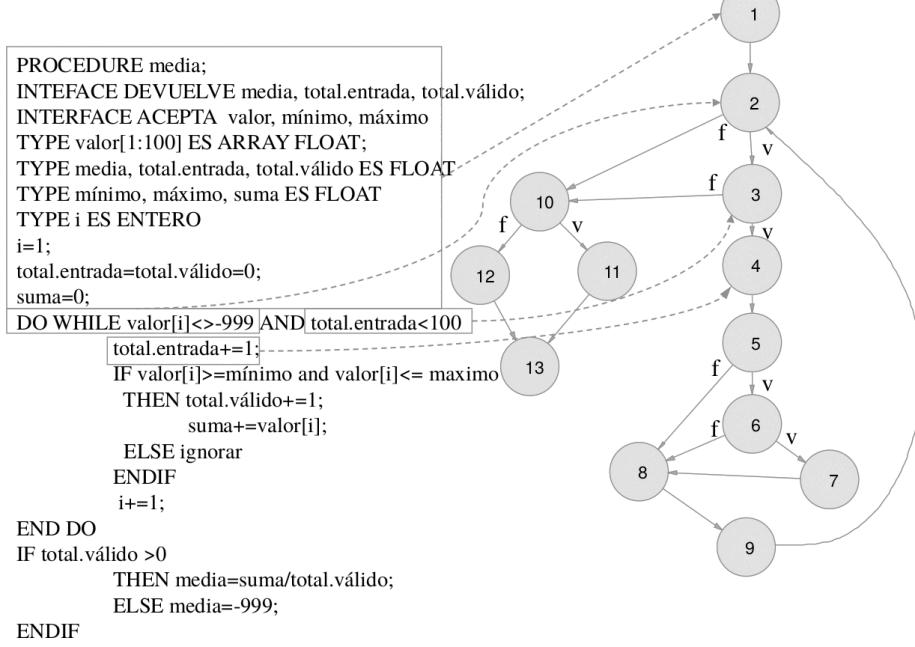


Figura 30: Ejemplo de un grafo de flujo. Nótese que (1) no todos los caminos tienen por qué poder recorrerse, dado que no existirá ningún conjunto de variables que aseguren ir por él, y que (2) ejecutar un bucle un número distinto de veces, supone caminos distintos.

Nota: *Un camino lógico es equivalente a un caso de prueba.*

6.1. Criterios de cobertura lógica

A continuación, se dispone una posible clasificación de los criterios de cobertura lógica según *Myers*. Se encuentran en orden de exigencia (confianza en haber detectado fallos) y, por ende, también en coste económico:

1. **Cobertura de sentencias:** Cada sentencia o instrucción del programa se ejecuta al menos una vez. *Por ejemplo: un bucle while solo se ejecutaría una vez.*
2. **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.
3. **Cobertura de condiciones:** Cada condición de cada decisión adopta, al menos una vez, un valor verdadero y otro falso. No garantiza la cobertura de decisiones.
4. **Criterio de decisión/condición:** Se exige el criterio de cobertura de condiciones, obligando a que se cumpla también el criterio de decisiones. El objetivo es complementarlos, pero buscando igualmente el mínimo número posible de casos de prueba.
5. **Criterio de condición múltiple:** Dado que la evaluación de las condiciones de una decisión no se realiza de forma simultánea, se descompone cada decisión múltiple en una secuencia de decisiones unicondicionales¹, y se exige que todas las combinaciones posibles de resultados (verdadero/falso) de cada condición en cada decisión se ejecuten al menos una vez.

¹Una decisión es un conjunto de condiciones. Por ejemplo, `a != null && a.length > 3` es una decisión que se descompone en las condiciones `a != null` y `a.length > 3`.

Dicho de otro modo, se descompone una decisión múltiple en decisiones unicondicionales, y se asegura el cumplimiento del anterior criterio para cada una de estas.

6. **Cobertura de caminos:** Cada uno de los posibles caminos del grafo de caminos (es decir, del programa) se ejecuta al menos una vez; un camino se define como la secuencia de sentencias encadenadas desde la sentencia inicial del programa hasta su sentencia final.

Nota: *Los bucles suponen un gran problema al incrementar enormemente los posibles caminos del grafo de caminos. Por ello, para reducir el número de caminos a probar, se habla del concepto de camino de prueba (test path), que es un camino del programa que atraviesa, como máximo, una vez el interior de cada bucle que encuentra; sin embargo, otros especialistas recomiendan probar los bucles múltiples veces, para comprobar cómo se comporta a partir de los valores procedentes de las operaciones realizadas en su interior.*

6.2. Prueba de bucles

Es necesario tener presente que los bucles son una importante pieza de la inmensa mayoría de los algoritmos implementados en software.

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 cuatro clases de bucles:

1. **Bucle simple:** 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 $m < n$, y tal que n es el mayor número de pasos permitidos por el bucle.
 - Hacer $n - 1$ y $n + 1$ pasos
2. **Bucle anidado:** Si se extendiese el enfoque de prueba de los bucles simples a los bucles anidados, en número posible de pruebas aumentaría geométricamente, llegando a ser impracticable. Por ello, se opta por la siguiente metodología:
 - Comenzar por el bucle más interior, estableciendo los demás bucles con sus valores mínimos.
 - Llevar a cabo la prueba de bucles simples al más interior, mientras se mantienen los valores en los bucles externos.
 - Progresar hacia fuera, llevando a cabo pruebas para el siguiente bucle, pero manteniendo el resto de bucles externos en sus valores mínimos, y los demás bucles anidados en sus valores típicos (es decir, se fija un valor para estos).
 - Continuar hasta haber probado todos los bucles.
3. **Bucle concatenado:** Se pueden probar mediante el enfoque para bucles simples, siempre que cada bucle sea independiente del resto. Si son dependientes (por ejemplo, hay dos bucles concatenados y el segundo usa el controlador del primero como valor inicial), se usa la técnica para bucles anidados.

4. **Bucles no estructurados:** Deben ser rediseñados para que se ajusten a las construcciones de la programación estructurada.

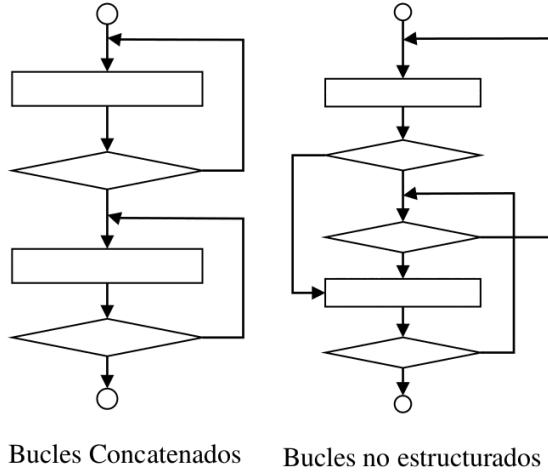


Figura 31: Ejemplificación de dos bucles: el izquierdo se ajusta a la programación estructurada, mientras que el derecho no lo hace.

6.3. Utilización de la complejidad ciclomática de *McCabe*

La métrica de *McCabe* es un **índicador del número de caminos independientes que existen en un grafo**; dicho de otro modo, trata de determinar, dado un grafo, cuántos caminos son precisos para probarlo. Se basa en la teoría de grafos. Por extensión, esta métrica también proporciona una medida cuantitativa de la complejidad lógica del módulo de software que se está probando.

El propio *McCabe* definió como un buen criterio de prueba la consecución de la ejecución de un conjunto de caminos independientes igual al indicado por la métrica; es más, la métrica de *McCabe* coincide con el número máximo de caminos independientes que puede haber un grafo. Cabe destacar que **un camino es independiente de otros si incorpora un arco que los demás no incluyen**.

Además, el conjunto de caminos que permite establecer asegura que todo el código de ejecuta, al menos una vez, por lo que cubriría la cobertura de sentencias. Este criterio también se ha propuesto como equivalente a la cobertura de decisiones, pero han surgido contraejemplos que lo invalidan.

La complejidad de *McCabe* $V(G)$ puede ser calculada de tres maneras a partir de un grafo de flujo G :

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.
3. $V(G) = c + 1$, siendo c el número de nodos de condición. Una condición de n arcos de salida se contabiliza como $n - 1$ (si $n > 2$, equivale al número de bifurcaciones binarias necesarias para simular dicha bifurcación n -aria).

$$V(G) = A - N + 2 = 11 - 9 + 2 = 4$$

$$V(G) = R = 4$$

$$V(G) = P + 1 = 3 + 1 = 4$$

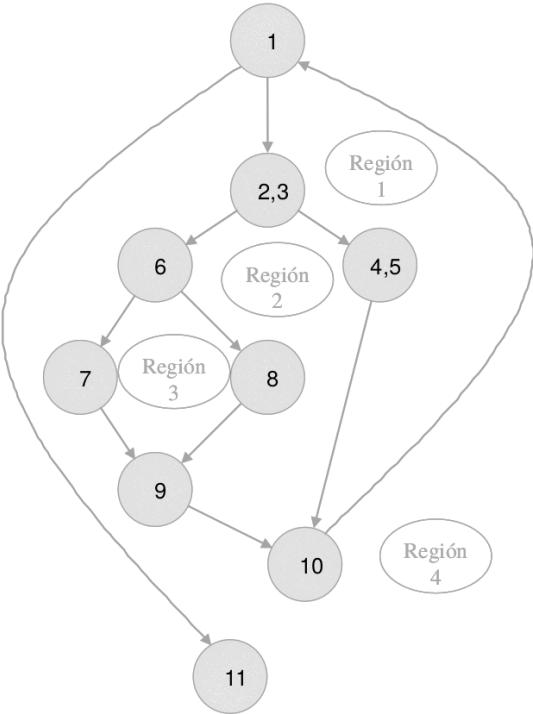


Figura 32: Ejemplificación del cálculo de la complejidad ciclomática de *McCabe*.

Nota: Hay que tener presente, para el número de regiones, que la fórmula solo es aplicable a grafos fuertemente conexos (siempre existe un camino para cualesquiera dos nodos que se escojan). Esto no se verifica en los programas con un nodo de inicio y otro de final, por lo que, para calcular las regiones, deben unirse estos nodos, o bien debe contabilizarse la región externa.

Una vez calculado del valor de $V(G)$, permitiendo afirmar por lo tanto el número máximo de caminos independientes del grafo G , para ayudar a la elección de los caminos de prueba, McCabe propone el **método del camino básico** para la selección de los caminos independientes. Consiste en:

1. Se escoge un camino de prueba típico (básico).
2. Se crean variaciones sobre el camino, de forma que cada una de ellas se distinga en, al menos, una arista de las demás.
3. Una vez seleccionados los caminos, se analiza el código para determinar las entradas que los fuerzan.
4. Finalmente, se revisa la especificación para predecir las salidas correctas (en teoría) ante las entradas determinadas.

Es conveniente tener presente que algunos caminos no se podrán ejecutar solos, sino que requerirán la ejecución de algún otro. Además es posible que, para un camino, no exista un conjunto de entradas que lo fuercen, por lo que deberá ser sustituido por otro para seguir satisfaciendo el criterio de *McCabe*.

Nota: $V(G)$ 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 en el módulo aumenta por

presentar una elevada complejidad (salvo que sea debido a sentencias switch case o similares); en estos casos, debería replantearse el diseño del módulo.

7. Pruebas funcionales (de caja negra)

Estas pruebas se centran en el estudio de la especificación del software; es decir, conocidas las entradas, es posible determinar las salidas esperadas, y el flujo de datos interno no importa. Se tratará de encontrar errores en las siguientes categorías mediante las pruebas de caja negra:

- Funciones incorrectas o ausentes.
- Errores de interfaz.
- Errores en estructuras de datos o accesos a bases de datos.
- Errores de rendimiento.
- Errores de inicialización y terminación.

De nuevo, la prueba exhaustiva es generalmente impracticable, por lo que deben buscarse criterios que permitan elegir un subconjunto de casos cuya ejecución aporte una cierta confianza en detectar los posibles defectos del software.

Existen dos enfoques a la hora de seleccionar casos de prueba de caja negra.

7.1. Enfoque sistemático

El diseño de pruebas se apoyará en las siguientes dos definiciones de *Myers* para buscar buenos casos de prueba (presenta una alta probabilidad de detectar un nuevo error):

- El caso ejecuta el máximo número de posibilidades de entrada diferentes para reducir así el total de casos.
- Cada entrada cubre un conjunto extenso de otras: Además de dar información sobre la ausencia o presencia de defectos con las entradas probadas, estos resultados pueden ser generalizados con cierta seguridad para otro conjunto de entradas similares que no hayan sido probadas.

7.1.1. Partición o clases de equivalencia

Esta técnica de diseño de casos se basa en **dividir el dominio de valores de entradas** en un número finito de clases, de modo que **la prueba de un valor representativo de una clase permite suponer**, razonablemente, que **el resultado obtenido será el mismo** que el obtenido probando cualquier otro valor de la clase.

Para ello, hay que realizar las siguientes tareas:

1. **Identificar las posibles clases de equivalencia a partir de la especificación del programa.** Consta de los siguientes pasos:
 - a. **Identificar las condiciones de las entradas del programa;** es decir, las restricciones de formato y posibles valores de las entradas.

- b. **Identificar, a partir de ellas, las clases de equivalencia**, que pueden ser (1) de datos válidos, o (2) de datos no válidos o erróneos. Esta identificación debe realizarse, como ya se comentó, basándose en el principio de igualdad de tratamiento de los valores de cada clase.

Existen algunas reglas que ayudan a identificar clases:

- R1. Si se especifica un rango de valores, se creará una clase válida y dos clases no válidas. *Por ejemplo, $5 < n < 7$.*
- R2. Si se especifica una lista de valores de tamaño variable, se creará una clase válida y dos no válidas. *Por ejemplo, puede haber de 1 a 4 titulares para una cuenta bancaria.*
- R3. Si se especifica una situación del tipo *debe ser* o booleana, se creará una clase válida y una no válida. *Dos ejemplos serían que (1) el primer carácter debe ser una letra, o que (2) la edad debe introducirse en formato numérico.*
- R4. Si se especifica un conjunto de valores admitidos, que son tratados de forma distinta, se creará una clase válida por cada valor, y una clase no válida. *Por ejemplo, pueden registrarse tres tipos de inmuebles, siendo chalets, pisos, o locales comerciales.*
- R5. En cualquier caso, si se sospecha que ciertos elementos de una clase no se tratan igual que el resto de la misma, deben dividirse en clases menores.

2. Crear los casos de prueba correspondientes.

Consta de las siguientes fases:

- a. Asignar un número único a cada clase de equivalencia.
- b. Hasta que todas las clases de equivalencia hayan sido cubiertas por casos de prueba, se tratará de escribir un caso que cubra tantas clases válidas no incorporadas como sea posible.
- c. Hasta que todas las clases de equivalencia no válidas hayan sido cubiertas, se escribe un caso para cada una de las clases no válidas sin cubrir.

El motivo de cubrir, con casos individuales, las clases no válidas, es que ciertos controles de entrada pueden enmascarar o invalidar otros controles similares. Por ejemplo, en un programa donde hay que *introducir cantidad (1-99) y letra inicial (A-Z)*, ante el caso *105 &* (dos errores), se puede indicar solo el mensaje *105 fuera de rango*, y dejar sin examinar el resto de la entrada a pesar de ser también inválida. Esto sería incluso más peligroso si el programa respondiese con un mensaje ambiguo, como *dato incorrecto, introduzca valor*, dado que podría llevar a pensar al equipo de pruebas a que ambos errores fueron correctamente identificados.

7.1.2. Análisis de valores límite (ALV)

Mediante la experiencia, se ha podido constatar que los los casos de prueba que exploran las condiciones límite de un programa producen un mejor resultado para la detección de defectos. Las condiciones límite se pueden definir como las situaciones que se hayan directamente arriba, abajo, y en los márgenes de las clases de equivalencia.

Realmente, esta técnica de diseño de casos complementa a la de particiones. Las diferencias entre ellas son:

- En lugar de elegir “cualquier” elemento como representativo de una clase de equivalencia, se requiere que se escojan uno o más elementos tal que los márgenes se sometan a prueba.
- En lugar de concentrarse únicamente en el dominio de entradas, los casos de prueba se generan considerando también el espacio de salida.

De nuevo, el proceso de selección es heurístico, como en la técnica de particiones, pero existen ciertas reglas orientativas:

- R1. Si una condición de entrada especifica un intervalo cerrado de valores (*por ejemplo* $[-1,0, 1,0]$), se deben generar casos para los extremos del rango ($-1,0$ y $1,0$), y casos no válidos para situaciones justo más allá de los extremos ($-1,001$ y $1,001$, *en el caso de que se admitan tres decimales*).
- R2. Si la condición de entrada especifica un número de valores (*oir ejemplo, el fichero de entrada tendrá de 1 a 255 registros*), se diseñarán casos para (1) el valor máximo, (2) el valor mínimo, (3) uno más que el máximo, y (4) uno menos que el mínimo (*en este caso, 0, 1, 255, 256*).
- R3. Usar la regla R1 para la condición de salida. *Por ejemplo, si el descuento máximo aplicable en una compra será el 50 %, y el mínimo será el 6 %, se escribirán casos para intentar obtener descuentos de 5.99 %, 6 %, 50 %, y 50.01 %.*
- R4. Usar la regla R2 para cada condición de salida. *Por ejemplo, si el programa puede mostrar de 1 a 4 listados, se escribirán casos para intentar generar 0, 1, 4, y 5 listados.*

En esta regla, como en la R3, debe recordarse que:

- Los valores límite de entrada no generan necesariamente los valores límite de salida.
- No siempre se pueden generar resultados fuera del rango de salida, pero debe ser considerado igualmente.
- Si la entrada o salida de un programa es un conjunto ordenado, los casos deben concentrarse en el primer y último elemento. *Por ejemplo, una tabla, o un archivo secuencial.*

7.1.3. Conjetura de errores

La idea básica de esta técnica consiste en **enumerar una lista de equivocaciones que pueden cometer los desarrolladores**, para después generar casos de prueba por cada uno de los elementos en ella. No existen directivas eficaces, ya que la lista será creada en base a la intuición y la experiencia; sin embargo, algunos valores a tener en cuenta son:

- El valor cero, tanto en la entrada como en la salida.
- En situaciones en las que se introduce un número variable de valores, conviene centrarse en los casos de (1) que no haya ningún valor, (2) que solo haya uno, y (3) que todos los valores sean iguales.
- Es recomendable imaginar que el programador pudiera haber interpretado algo mal en la especificación.
- También interesa imaginar lo que el usuario pueda introducir como entrada a un programa, previniendo toda clase de acciones, incluse como si fuese poco hábil, o que presentase malas intenciones.

7.2. Enfoque aleatorio

Se basa en la generación de pruebas aleatorias, en las cuales se simula la entrada habitual del programa creando los datos de entrada en la secuencia y frecuencia con las que podrían aparecer en la práctica. Generalmente, el objetivo de estos llamados **tests de esfuerzo** es el **evaluar el rendimiento del sistema**, puesto que, como las entradas son generadas aleatoriamente, no se pueden predecir las salidas. Esto es debido a que, si por el contrario contásemos con un software que predijese las salidas con total fiabilidad, este podría reemplazar directamente al software que está siendo probado.

Para crear estos tests, es necesario contar con una herramienta generadora de pruebas, a la que se especifica la descripción de las entradas, las secuencias de entrada posibles, y las probabilidades de ocurrir en la práctica (por ejemplo, indicando una distribución estadística si la siguen). Si el proceso de generación se ha realizado correctamente, eventualmente se crearán todas las posibles entradas del programa, en todas las posibles combinaciones y permutaciones.

7.3. Métodos de prueba basados en grafos

El primer paso en la prueba de caja negra es **entender los objetos que se modelan en el software**, y las relaciones entre ellos. Una vez realizado todo esto, se continúa definiendo una serie de pruebas que verifiquen que todos los objetos tienen, entre ellos, las relaciones esperadas. Dicho de otra manera, la prueba del software empieza creando un grafo de objetos importantes y sus relaciones, y después **diseñando una serie de pruebas que cubran el grafo de manera que se ejerciten todos los objetos y sus relaciones** para descubrir los errores.

Para llevar a cabo estos pasos, debe comenzarse creando el grafo, que representará:

- Objetos mediante sus nodos.
- Relaciones entre los objetos mediante los arcos.
- Propiedades de un objeto mediante el peso de su correspondiente nodo.
- Características de las relaciones mediante el peso de su correspondiente arco.



Figura 33: Representación simbólica de un grafo para pruebas de caja negra.

Una vez creado el grafo, se puede proceder a la creación de los casos de prueba. Entre los beneficios que estos grafos a este procedimiento, se encuentra el facilitar:

- La identificación de los bucles a probar.
- El cumplimiento de la cobertura de nodos, de modo que ninguno haya sido omitido.
- El cumplimiento de la cobertura de enlace, probándose todas y cada una de las relaciones. Cada relación debe ser probada basándose en sus propiedades:
 - Se estudia la transitividad de las relaciones secuenciales para determinar cómo se propaga el impacto de las relaciones a través de los objetos definidos en el grafo.
 - La simetría de una relación también es importante para el diseño de casos de prueba; si un enlace es bidireccional, debería probarse esta característica.
 - Todos los nodos del grafo deberían ser reflexivos, es decir, tener una relación que los devuelva a ellos mismos.

A continuación, se describe un número de métodos de prueba de comportamiento que pueden hacer uso de los grafos:

- **Modelado del flujo de transacción:** Los nodos representan los pasos de alguna transacción, y los enlaces son las conexiones lógicas entre ellos; sería posible partir de un DFD. *Por ejemplo, los pasos requeridos para hacer una reserva en un hotel; son pasos del proceso, se use software o no.*
- **Modelado de estados finitos:** Los nodos son estados del software observables por el usuario, y los enlaces representan las transiciones que ocurren para moverse de un estado a otro. *Por ejemplo, el usuario puede interactuar con pantallas, sobre las cuales realizar acciones que lleven a otras pantallas.*
- **Modelado del flujo de datos:** Los nodos son objetos de datos, y los enlaces son las transformaciones que sufren para pasar de ser un objeto a otro.

Es importante tener presente que **todos son modelos de caja negra**, por lo que no modelan cómo está programado el software; es decir, no modelan ni los estados del software, ni sus transacciones programadas, ni sus flujos de datos. En realidad, **modelan su comportamiento visible**.

8. Enfoque práctico recomendado para el diseño de casos

El enfoque recomendado para el uso de las técnicas de diseño de casos pretende mostrar el uso más apropiado de cada técnica para la obtención de un conjunto de casos útiles sin perjuicio de las estrategias de niveles de prueba.

1. Si la especificación contiene combinaciones de condiciones de entrada, se comienza formando sus grafos causa–efecto para comprenderlas con más facilidad.
2. Se identifican las clases válidas y no válidas de equivalencia para la entrada y la salida.
3. En todos los casos, se usa 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.
4. Se utiliza la conjectura de errores para añadir nuevos casos, principalmente referidos a valores especiales.
5. Se ejecutan los casos generados hasta el momento (de caja negra), y se analiza la cobertura obtenida.
6. Se examina la lógica del programa para añadir los casos precisos (de caja blanca) para cubrir el criterio de cobertura elegido, siempre y cuando no haya sido satisfecho en el anterior punto.

9. Documentación del diseño de las pruebas (IEEE 829)

Conviene tener siempre presente que la documentación de las pruebas es necesaria para una buena organización de las mismas, así como para asegurar su reutilización. Según el estándar estándar IEEE 829, se generará la siguiente documentación:

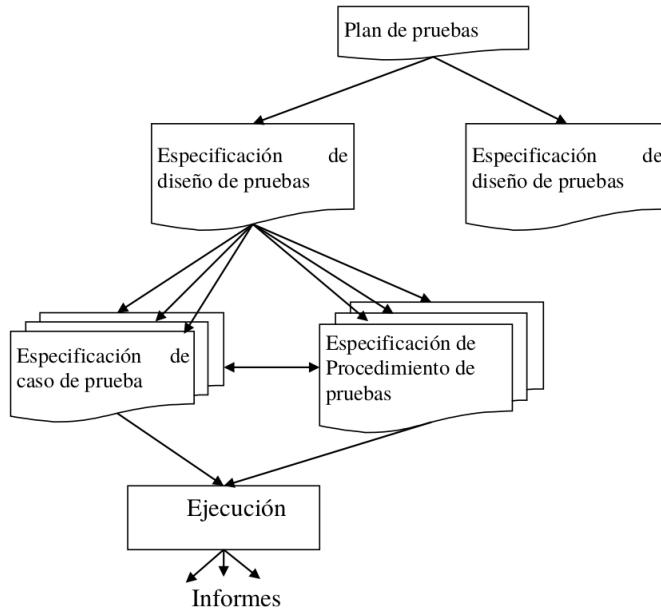


Figura 34: Representación de los documentos del estándar IEEE 829 para la documentación del diseño de pruebas.

1. **Plan de pruebas:** El documento tiene el objetivo de mostrar la planificación general del esfuerzo para cada fase de la estrategia de prueba del producto. Parte de los elementos fijados en el estándar son:
 - a) Una introducción y resumen de los elementos y características a probar.
 - b) Los elementos de software a probar (programas o módulos).
 - c) Qué características se van a probar y cuáles no.
 - d) Actividades, técnicas, herramientas...
 - e) Criterios de paso/fallo para cada elemento.
 - f) Documentos a entregar.
 - g) Actividades de preparación y ejecución de las pruebas.
 - h) Necesidades del entorno.
 - i) Responsables de la realización de las pruebas, y sus necesidades.
 - j) Esquema de tiempos y riesgos asumidos.
2. **Especificación del diseño de pruebas:** Partiendo del plan de pruebas, se realizan ampliaciones de los detalles mostrados, especificando un diseño de prueba por cada módulo del programa. En el estándar se fija la siguiente estructura:
 - a) Las características a probar de los elementos software.
 - b) Detalles sobre el plan de pruebas creado, incluyendo las técnicas de prueba específicas y los métodos de análisis de resultados.
 - c) Se identifica cada prueba: Identificador, casos que se van a utilizar, procedimientos que se van a seguir.
 - d) Criterios de paso/fallo de la prueba.

3. **Especificación del caso de prueba:** A partir del anterior diseño, se definen con detalle cada uno de los casos de prueba mencionados, especificando datos de pruebas exactos, resultados esperados, y demás detalles, para obtener un conjunto de pruebas de validación y de verificación. En el estándar se fija la siguiente estructura:
 - a) Elementos software a probar, y cuáles de sus características serán ejercitadas por el caso.
 - b) Especificaciones de cada entrada requerida para ejecutar el caso. Deben detallarse además las relaciones entre estas; *por ejemplo, la sincronización de las mismas*.
 - c) Especificaciones de todas las salidas y las características requeridas (*por ejemplo, el tiempo de respuesta*).
 - d) Necesidades del entorno. *Por ejemplo, en cuanto a hardware, software, o personal.*
 - e) Requisitos o restricciones especiales de procedimiento.
 - f) Dependencias entre casos.
4. **Especificación del procedimiento de prueba ()**: Tras generar los casos de prueba detallados, se especifica cómo proceder en detalle a su ejecución. En el estándar se fija la siguiente estructura:
 - a) Objetivo del procedimiento y lista de casos que se ejecutan con él.
 - b) Requisitos especiales para la ejecución.
 - c) Pasos en el procedimiento. Además de la manera de registrar los resultados y los incidentes de la ejecución, se debe especificar:
 - La secuencia necesaria de acciones para preparar la ejecución.
 - Acciones necesarias para empezar la ejecución.
 - Acciones necesarias durante la ejecución.
 - Cómo se realizarán las medidas. *Por ejemplo, del tiempo de respuesta.*
 - Cómo tratar incidencias y restaurar el entorno de ejecución.

10. Ejecución de las pruebas

10.1. El proceso de ejecución (IEEE 1008)

El proceso abarca las siguientes fases:

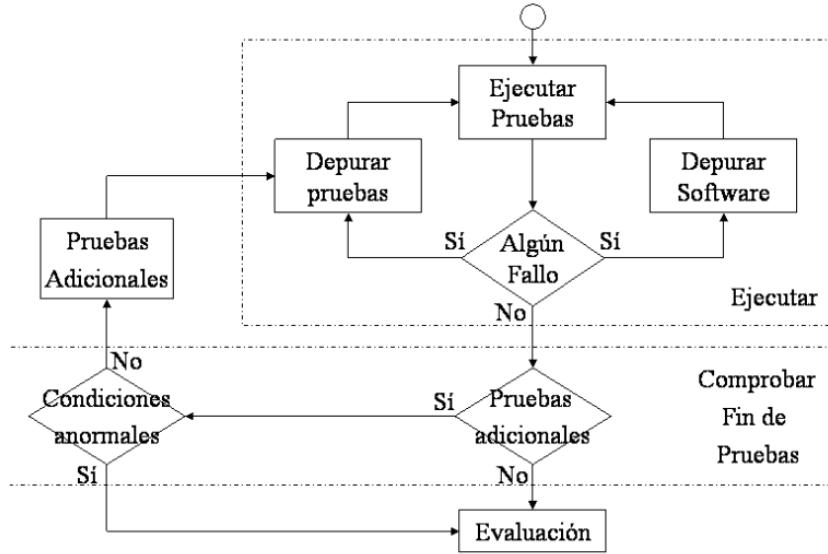


Figura 35: Representación de proceso de pruebas según el estándar IEEE 1008.

1. **Proceso de ejecución:** Se ejecutan las pruebas, cuyos casos y procedimientos han sido ya diseñados previamente. Consta de las siguientes fases:
 - a) Se ejecutan las pruebas.
 - b) Se comprueba si ha habido algún fallo al ejecutar, que impida terminar la ejecución de algún caso. *Por ejemplo, se cae el sistema, se bloquea el teclado...*
 - c) Si lo ha habido, se puede deber a un defecto software, que llevará a la depuración o corrección del código, o a un defecto en el propio diseño de las pruebas. En ambos casos, las nuevas pruebas o las corregidas deberán ejecutarse.
 - d) De no existir fallo, se pasará a la comprobación de la terminación de las pruebas.
2. **Comprobación de la conclusión del proceso de prueba:** Se lleva a cabo según criterios de compleción de prueba que suelen ser especificados en el plan de pruebas. Consta de las siguientes fases:
 - a) Tras la ejecución, se comprobará si se cumplen los criterios de compleción descritos en el correspondiente plan de pruebas. *Por ejemplo, se terminan las pruebas cuando se han probado todos los procedimientos de operación o se ha cumplido la cobertura lógica marcada.*
 - b) En caso de terminar las pruebas, se pasa a la evaluación de los productos probados sobre la base de los resultados obtenidos. *Se alcanza una terminación normal.*
 - c) En caso de no terminar las pruebas, se debe comprobar la presencia de condiciones anormales en la prueba. Si hubiesen existido condiciones anormales (*por ejemplo, no se han podido ejecutar todos los casos porque el sistema se cae regularmente*), se pasa de nuevo a la evaluación; *se alcanza una terminación anormal.* En caso contrario, se pasa a generar y ejecutar pruebas adicionales para satisfacer cualquiera de las dos terminaciones.
3. **Si las pruebas han terminado, se evalúan los resultados; en caso contrario, hay que generar pruebas adicionales** para que se satisfagan los criterios de compleción de pruebas.

10.2. Documentación de la ejecución de las pruebas

La documentación de la ejecución de las pruebas también es fundamental para dejar constancia de los resultados de las pruebas. Parte de los documentos generados se relacionan directamente con el estándar IEEE 829, pudiéndose distinguir dos grupos principales:

- **Documentación de entrada:** Constituida principalmente por las especificaciones de los casos de prueba que se van a usar, y las especificaciones de los procedimientos de pruebas.
- **Documentación de salida o informes sobre la ejecución:** Cada ejecución generará dos tipos de documentos.
 - **Histórico de pruebas (*test log*):** Documenta todos los hechos relevantes ocurridos durante la ejecución de las pruebas. La estructura fijada en el estándar es:
 - Descripción de la prueba, detallando los elementos probados y el entorno de prueba.
 - Anotaciones de datos sobre cada hecho ocurrido, incluidos el inicio y fin de la prueba: fecha, hora, e identificador del informe de incidente asociado, en caso de haberlo.
 - Otras informaciones.
 - **Informes de los incidentes ocurridos (*test incident report*):** siempre en caso de que sucedan durante la ejecución: Cada uno documenta un incidente que requerirá una posterior investigación. La estructura fijada en el estándar es:
 - Resumen del incidente.
 - Descripción de datos objetivos: fecha, hora, entradas, resultados esperados, etc.
 - Impacto que tendrá sobre las pruebas.

Además, toda la documentación de salida correspondiente a un mismo diseño de prueba se recoge en un informe resumen de pruebas (*test summary report*), resumiéndose así los resultados de las actividades de prueba, y aportándose una evaluación del software basada en ellos. Entre otros elementos a incluir en el informe, destacan:

- Resumen de la evaluación de los elementos probados.
- Variaciones del software respecto a su especificación de diseño, así como las variaciones en las pruebas.
- Valoración de la extensión de la prueba. *Cobertura lógica, funciona, de requisitos, etc.*
- Resumen de cada una de las actividades de prueba, incluyendo detalles sobre los recursos empleados.

Nota: *Cada uno de los documentos detallados anteriormente, tanto para la documentación del diseño de las pruebas, como para la ejecución de estas, debe contar con un identificador.*

10.3. 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, que conlleva mandar un módulo a depuración. De este modo, el proceso de depuración puede dar lugar a dos situaciones:

- Encontrar la causa del error, analizarla y corregirla.
- No encontrar la causa y, por lo tanto, tener que generar nuevos casos de prueba que puedan proporcionar información adicional para su localización.

Por lo tanto, se puede distinguir las dos fases siguientes:

- **Localización del defecto:** Suele conllevar la mayor parte del esfuerzo.
- **Corrección del defecto.**

Además, tras corregir el efecto, deberán efectuarse nuevas pruebas que comprueben si efectivamente ha sido eliminado.

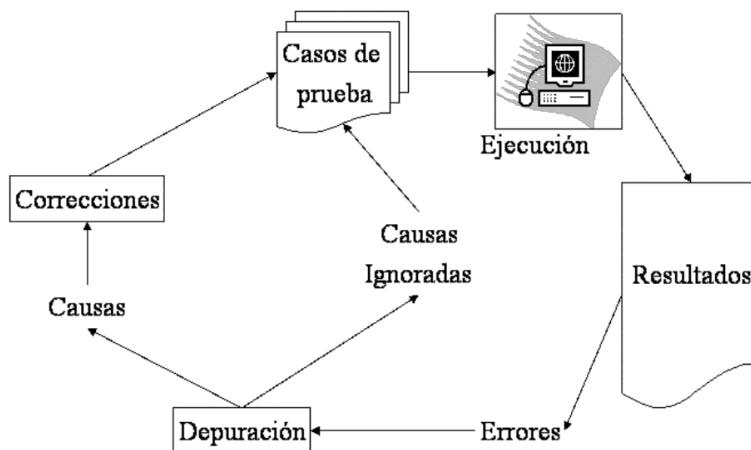


Figura 36: Representación de proceso de depuración.

10.3.1. Consejos para la depuración

- **Para la localización del error:**

- Es el proceso mental de solución de un problema. Por ello, debería analizarse la información disponible, en lugar de explorar aleatoriamente o experimentar cambiando el programa.
- Las herramientas de depuración deberían emplearse como recurso secundario.
- Al llegar a un punto muerto, puede merecer la pena pasar a otra cosa para refrescar la mente, e incluso puede ayudar el describir el problema a otra persona.
- Se deben atacar los errores individualmente, o solo se dificultará la depuración.
- Se debe fijar la atención también en los datos manejados, y no solo en la lógica del proceso.

- **Para la corrección del error:**

- Donde hay un defecto, suele haber más (*ejemplo del principio de Pareto*).
- Debe corregirse el defecto, en lugar de intentar enmascarar sus síntomas.
- La probabilidad de corregir un defecto perfectamente no es el 100 %, por lo que las correcciones deben ser revisadas antes de implantarlas.
- Debe tenerse cuidado para no crear nuevos defectos por corregir sin cautela.
- La corrección debe situarnos temporalmente en la fase de diseño, para alterar el correspondiente al código defectuoso.

10.4. Análisis de errores a análisis causal

El objetivo del análisis causal es **proporcionar información sobre la naturaleza de los defectos, para que así el personal pueda prevenirlas en el futuro**, al conocer los errores que comete; esta información no debe ser empleada nunca para evaluar al personal. Se recoge la siguiente información:

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

11. Estrategia de aplicación de las pruebas

Una vez conocidas las técnicas de diseño y de ejecución, se debe analizar cómo se plantea la utilización de las pruebas en el ciclo de vida. La estrategia de aplicación y la planificación de las pruebas pretenden integrar el diseño de los casos de prueba y permite la coordinación del personal y del cliente, gracias a la definición de los papeles que deben desempeñar cada uno de ellos, así como de la forma de llevarlos a cabo.

Las etapas de las que constaría la estrategia serían:

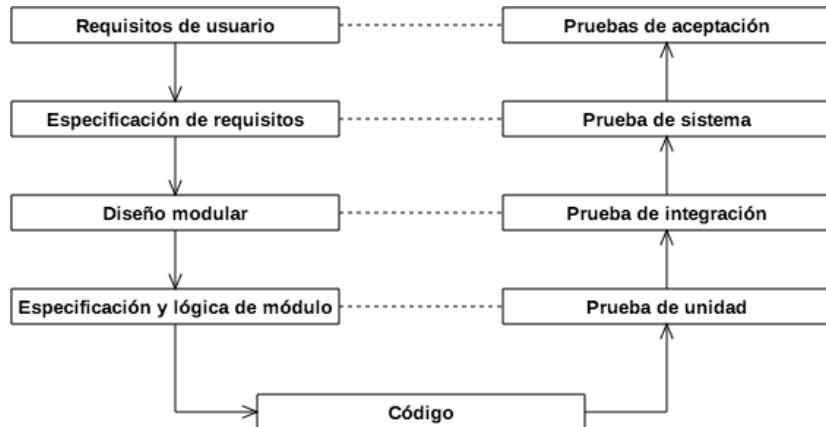


Figura 37: Las diferentes estrategias de prueba y su relación con las otras fases del construcción del software

1. **Se comienza en la prueba de cada módulo**, realizada normalmente por el propio personal de desarrollo.
2. Con el esquema del diseño del software, **los módulos probados se integran para comprobar sus interfaces**.

3. **El software totalmente ensamblado se prueba como un conjunto** para comprobar si cumple o no tanto los requisitos funcionales como los requisitos de rendimientos, seguridad, etc. Este nivel coincide con la prueba del sistema cuando solo se trata de software.
4. El caso de que el sistema final coste de más que software, **el software ya validado se integra con el resto del sistema** (*por ejemplo, elementos mecánicos*), **para probar su funcionamiento conjunto**. *Por extensión, todavía es posible simular el hardware implicado en la anterior etapa. El resultado de esta fase es la certeza de haber hecho lo deseado, además de haberlo hecho bien.*
5. Finalmente, **el producto final se pasa a la prueba de aceptación** par que el **usuario compruebe, en su propio entorno de explotación** si lo acepta o no. *El resultado de esta fase es que el usuario tenga la certeza de recibir lo que él quería.*

Como se puede ver, cada nivel de prueba se centra en probar el software en referencia al trabajo realizado en una diferente etapa de desarrollo.

Nota: *Las pruebas de aceptación, de sistema y de integración están relacionadas directamente con las pruebas de validación. Por otra parte, las pruebas de integración (se encuentra en ambos bandos) y de unidad se encuentran directamente relacionadas con las pruebas de verificación. De todos modos, una prueba de módulo puede centrarse también en la especificación de las funciones que este debe realizar (de caja negra).*

Nota: *A pesar de que un ciclo en cascada tenga una fase de pruebas, estas suelen distribuirse a lo largo del todo el ciclo de vida.*

12. Pruebas en desarrollos orientados a objetos (OO)

Las técnicas y estrategias presentadas hasta el momento están inspiradas en la aplicación a desarrollos estructurados. Por lo tanto, al trabajar con tecnología *OO*, algunas de ellas pierden su interés o deben adaptarse.

12.1. Punto de vista del diseño de casos de pruebas

- **Técnicas de caja negra:** Siguen siendo completamente válidas. *Por ejemplo, el AVL, el tratamiento de combinaciones de entrada, la conjetura de errores...* Los casos de prueba deberían ser diseñados basándose en:
 - Los datos y eventos incluidos en los escenarios de los casos de uso.
 - Los caminos que se pueden trazar en los diagramas de actividad, o en los de estados, que describan el caso, sin olvidar todos los flujos alternativos, ni los tratamientos de errores y excepciones.
- **Técnicas de caja blanca:** Disminuyen sus posibilidades de aplicación, quedando confinadas a las instrucciones de cada método de cada clase. También cabría la posibilidad de aplicar la cobertura de caminos (incluido el criterio de *McCabe*) no solo al código de los métodos, sino también a los diagramas de comportamiento con estructura de red; *por ejemplo, diagramas de estados, de actividad, etc.*

En el caso del diseño de pruebas, cabe señalar además la posibilidad de distinguir criterios según el nivel de pruebas en el que se sitúe el diseño. Así, para las pruebas de unidad (es decir, pruebas de clases), la prueba de clases de equivalencia y de AVL son apropiadas aplicándolas a las entradas y salidas representadas como parámetros de métodos fundamentalmente. Sin embargo, **los cambios más radicales aparecen en las pruebas de integración**, dado que el enfoque de integración basado en la existencia de una estructura jerárquica no tiene sentido frente a las estructuras de colaboración en red definidas para el software *OO*.

Por una parte, existen propuestas que permiten encontrar aquellas clases que dependen en su funcionamiento de sí mismas o solo de unas pocas clases, para tomarlas como destino de la primera oleada de pruebas de integración, de modo que más adelante se irán sometiendo a prueba las clases que dependen solo de las ya probadas, y así consecutivamente hasta integrar toda la aplicación. Sin embargo, el enfoque más habitual consiste en ir probando hilos de clases que colaboran para una función o servicio del sistema, seguramente definidos en diagramas de colaboración; los diagramas de estado pueden aportar una mayor información para el rigor de la prueba.

12.2. Cuestiones a tener en cuenta

Finalmente, se señalarán dos cuestiones que tienen una gran influencia en las pruebas de software *OO*. 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, empeorando la preocupación del anterior punto.
- También es conveniente recordar que la propia programación o diseño *OO* hace menos probables ciertos tipos de error y más probables otros, además de provocar la aparición de nuevos tipos de defectos, frente a los desarrollos estructurados.

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

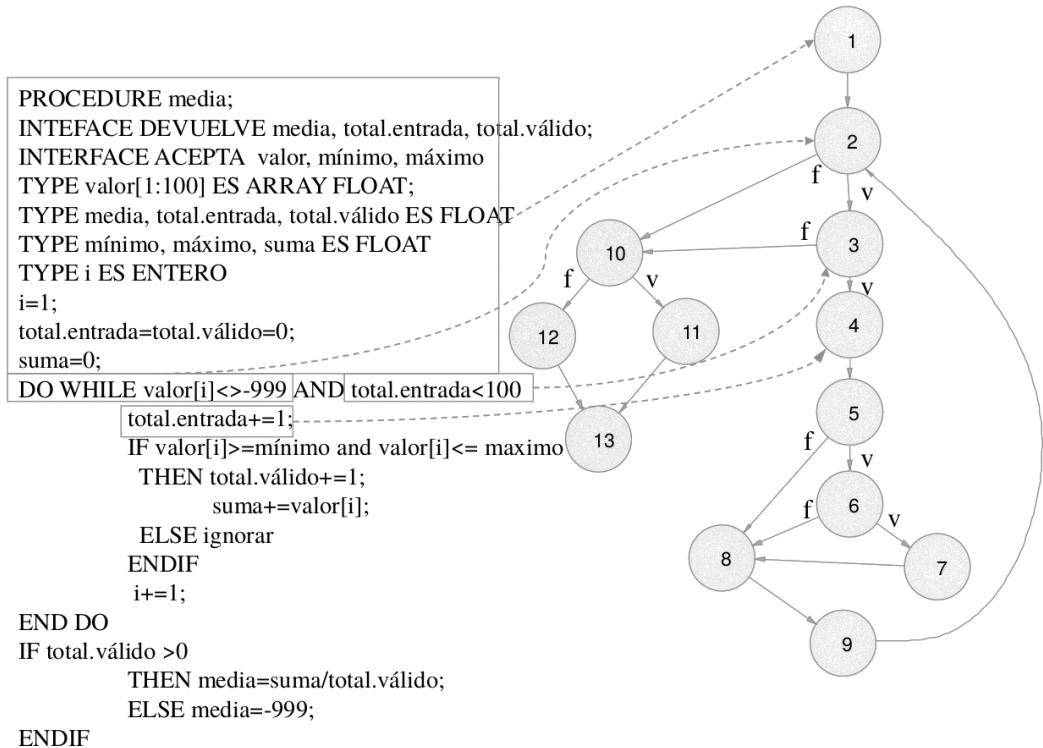


Figura 38: Ejemplo de un método y su correspondiente grafo de flujo.

Complejidad de *McCave*

1. $V(G) = a - n + 2 = 17 - 13 + 2 = 6$.
2. $V(G) = r = 6$.
3. $V(G) = c + 1 = 5 + 1 = 1$.