Capítulo

1

Introducción

Centro Asociado Palma de Mallorca

Tutor: Antonio Rivero Cuesta

1 Introducción

1.1 Introducción.

El software es una parte principal del entorno humano. Actualmente sería complicado imaginarse la vida sin toda la tecnología que utilizamos a diario.

El software está presente no sólo en los sistemas informáticos que realizan tareas de tratamiento de información sino en un sinfín de sistemas de la más diversa complejidad.

La ingeniería es según la Real Academia de la Lengua el conjunto de conocimientos y técnicas que permiten aplicar el saber científico a la utilización de la materia y de las fuentes de energía.

En esta asignatura se introduce el conjunto de técnicas y procedimientos que se han ido desarrollando a lo largo de las últimas décadas, para poder elaborar de una forma ordenada y eficiente tantas y tantas líneas de código que componen el software.

Todas estas técnicas y procedimientos componen la ingeniería de software.

1.2 Objetivos

En este capítulo se da una visión inicial de lo que es el software y cómo se produce Igualmente se profundiza en el concepto de ingeniería del software Se pretende que el lector adquiera una idea clara de los siguientes conceptos:

- Definición de software y requisitos de calidad exigibles.
- No todo el software es igual, según su aplicación y modo de desarrollarlo puede ser muy diverso Se da una visión de los distintos tipos de software
- Definición del concepto de Ingeniería del software y comprensión de su origen.
- Existen algunas ideas preestablecidas acerca del software que se analizan.

1.3 ¿Qué es el software?

El software incluye:

- Los programas que gobiernan el funcionamiento del sistema.
- Documentos.
- Bases de datos.
- Los procedimientos de operación o de mantenimiento periódicos.

El software puede ser:

- Un producto que se venda.
- Sólo una parte.

La elaboración del software ocupa a millones de personas en todo el mundo y se puede considerar una actividad económica en sí misma.

1.3.1 Calidad del software

La calidad de un producto puede valorarse desde puntos de vista diversos.

Existe un esquema general de mediciones de la calidad de software propuesto por MacCall y otros [McCall78], basado en valoraciones a tres niveles diferentes

- Factores.
- Criterios.
- Métricas.

Los factores de calidad constituyen el nivel superior, y son la valoración propiamente de la calidad. Esta valoración no se hace directamente, sino en función de ciertos criterios o aspectos de nivel intermedio que influyen en los factores de calidad.

Las métricas están en el nivel inferior, son mediciones puntuales de determinados atributos o características del producto, y son la base para evaluar los criterios intermedios.

Entre los factores de calidad propuestos encontramos los siguientes:

- **CORRECCIÓN**. Es el grado en que un producto software cumple con sus especificaciones. Podría estimarse como el porcentaje de requisitos que se cumplen adecuadamente.
- **FIABILIDAD**. Es el grado de ausencia de fallos durante la operación del producto software. Puede estimarse como el número de fallos producidos o el tiempo durante el que permanece inutilizable durante un intervalo de operación dado.
- **EFICIENCIA**. Es la relación entre la cantidad de resultados suministrados y los recursos requeridos durante la operación. Se mediría como la inversa de los recursos consumidos para realizar una operación dada.
- **SEGURIDAD**. Es la dificultad para el acceso a los datos o a los datos o a las operaciones por parte de personal no autorizado.
- **FACILIDAD DE USO**. Es la inversa del esfuerzo requerido para aprender a usar un producto software y utilizarlo adecuadamente.
- **MANTENIBILIDAD**. Es la facilidad para corregir el producto en caso necesario. Se aplica propiamente el mantenimiento correctivo.
- **FLEXIBILIDAD**. Es la facilidad para modificar el producto software. Se aplica propiamente al mantenimiento adaptativo y al perfectivo.
- **FACILIDAD DE PRUEBA**. Es la inversa del esfuerzo requerido para ensayar un producto software y comprobar su corrección o fiabilidad.
- **TRANSPORTABILIDAD**. Es la facilidad para adaptar el producto software a una plataforma (hardware + sistema operativo) diferente de aquella para la que se desarrolló inicialmente.

- **REUSABILIDAD**. Es la facilidad para emplear partes del producto en otros desarrollos posteriores. Se facilita mediante una adecuada organización de los módulos y funciones durante el diseño.

- **INTEROPERATIVIDAD**. Es la facilidad o capacidad del producto software para trabajar en combinación con otros productos.

Estos factores de calidad se centran en características del producto software.

El proceso de desarrollo influye muy directamente en la calidad del producto obtenido.

Comprobar la calidad de un software es una tarea compleja. Las pruebas o ensayos consisten en hacer un producto software o una parte de él en condiciones determinadas, y comprobar si los resultados son correctos. El objetivo de las pruebas es descubrir los errores que pueda contener el software ensayado.

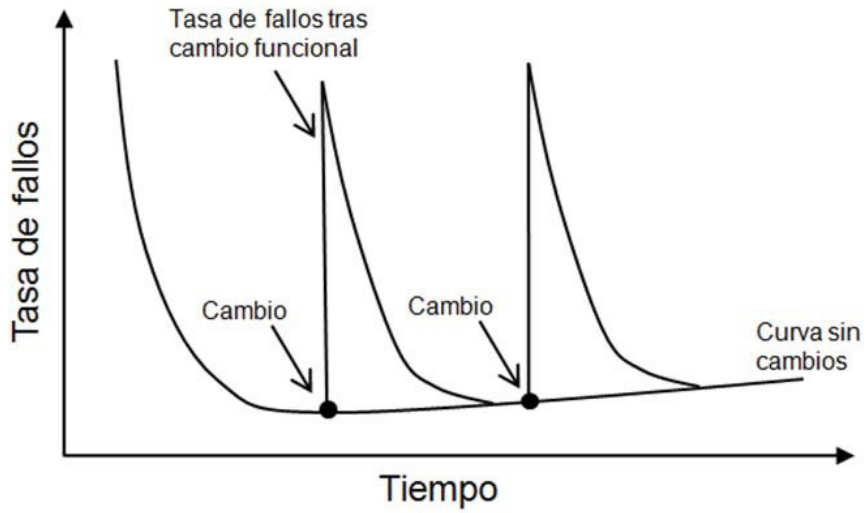
Las pruebas no permiten garantizar la calidad de un producto. Puede decirse que una prueba tiene éxito si se descubre algún error, con lo que se sabe que el producto no cumple con algún criterio de calidad. Por el contrario, si la prueba no descubre ningún error, no se garantiza con ello la calidad del producto, ya que pueden existir otros errores que habrían de descubrirse mediante pruebas diferentes.

En la figura 1.1 podemos observar de forma simplificada la evolución de la tasa de fallos de un software en el tiempo.

Inicialmente esta tasa es muy alta. Según se van corrigiendo los errores se reduce rápidamente.

Sin embargo, a lo largo de la vida de un software es frecuente realizar mejoras funcionales, dando lugar a distintas versiones.

Cada vez que introducimos cambios en las nuevas versiones, el número de errores del software se dispara, haciendo de nuevo necesario la corrección de los mismos.



1.3.2 Tipos de software

Clasificar el software no es una tarea fácil debido a la gran variedad de aplicaciones y métodos de desarrollo que existe. Una de las clasificaciones más completas agrupa el software en siete grandes categorías:

- Software de Sistemas

Lo forman todos aquellos programas necesarios para dar soporte a otros programas, como los sistemas operativos, los compiladores o los programas de gestión de redes.

Su principal característica es su alto grado de interacción con el hardware, ya que en muchos casos deben gestionar de forma eficiente el acceso al hardware por parte de otros programas o usuarios.

- Software de Aplicación

Son aplicaciones desarrolladas para resolver problemas específicos de los negocios.

En esta categoría incluiríamos el software de gestión de los bancos o de las grandes empresas en general.

- Software de Ingeniería y Ciencias

El objetivo es la programación de elaborados algoritmos matemáticos para modelar y simular complejos sistemas o procesos, tales como reacciones nucleares, modelos meteorológicos, la red eléctrica de un país o el diseño de un avión.

- Software Incrustado

Reside en el interior de un producto o sistema, y su objetivo es controlarlo, definir su comportamiento.

Suele ser muy específico y de pequeñas dimensiones, con la necesidad de operar en tiempo real.

Desde el regulador de temperatura de una vivienda hasta el sistema de frenos de un vehículo, están gobernados por este tipo de software.

- Software de Línea de Producto

Su objetivo es dar una determinada funcionalidad al consumidor.

En esta categoría encontramos procesadores de texto, hojas de cálculo o las aplicaciones de contabilidad para pequeñas empresas.

- Aplicaciones Web, WebApps

En los últimos años se ha extendido su utilización con la generalización de los aparatos móviles con acceso a redes.

Inicialmente simplemente se componían de archivos de hipertexto para la presentación de información, sin embargo hoy día tienen capacidad de cómputo y están integradas con aplicaciones y bases de datos corporativas.

A través de ellas se puede operar una cuenta bancaria, realizar todo tipo de compras, utilizar juegos muy elaborados ó conocer el tiempo en cualquier parte del mundo. La comodidad, rapidez y vistosidad son determinantes a la hora de que tengan éxito.

- Software de Inteligencia Artificial

Incluye aplicaciones de robótica, visión artificial, redes neuronales o sobre la teoría de juegos.

Utilizan algoritmos no numéricos para la resolución de los problemas, como por ejemplo árboles lógicos de búsqueda.

1.4 ¿Cómo se fabrica el software?

En los principios de la informática la labor de desarrollo de software se planteaba como una actividad artesanal basada en la labor de personas habilidosas y más o menos creativas que actuaban en forma individual y relativamente poco disciplinada.

Al aumentar la capacidad de los computadores gracias a los avances del hardware aumentó también la complejidad de las aplicaciones a programar y se apreció la necesidad de una mejor organización de la producción de software basada en el trabajo en equipo con la consiguiente división y organización del trabajo y el empleo de herramientas apropiadas que automaticen las labores triviales y repetitivas.

La identificación formal del problema origina una frenética actividad para la creación de metodologías concretas de desarrollo y en general en la concepción de la ingeniería del software como disciplina.

A finales de los años 60 se acuña el termino Ingeniería del Software en un congreso de la OTAN de manera formal.

Con esta denominación se designa el empleo en el desarrollo del software de técnicas y procedimientos típicos de la ingeniería en general.

El software tiene una particularidad especial frente a cualquier producto físico que podamos imaginar: una vez diseñado este se puede replicar con tremenda facilidad sin necesidad de un proceso de fabricación propiamente dicho.

A pesar de ello la ingeniería del software se ha desarrollado a partir de la ingeniería industrial.

La ingeniería del software amplía la visión del desarrollo del software como una actividad esencialmente de programación, contemplando además otras actividades de análisis y diseño previos, y de integración y verificación posteriores.

La distribución de todas estas actividades a lo largo del tiempo constituye los se ha dado en llamar ciclo de vida del desarrollo de software.

Ingeniería de software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software, y el estudio de estos enfoques, es decir, ***la aplicación de la ingeniería al software***.

A lo largo de los años 70 aparecen las herramientas CASE (Computer Aided Software Engineering) de diseño asistido por ordenador, que se aplican ampliamente durante los 80.

Las herramientas CASE tradicionales apoyaban a las actividades anteriores a la programación o codificación, para la que se seguías empleando herramientas tradicionales, como los compiladores, que funcionaban de forma totalmente separada de la herramienta CASE.

En los 90 se amplía el campo de aplicación de las herramientas de ingeniería de software, reconociendo la necesidad de automatizar aún más la labor de desarrollo mediante la formalización del proceso completo de producción de software y el empleo de herramientas que soporten todo el ciclo de vida de desarrollo.

Estas herramientas se designan con las siglas IPSE (Integrated Project Support Environment) o, más recientemente, ICASE (Integrated CASE).

Con la llegada del siglo XXI, el desarrollo e implantación de Internet, muchos de los planteamientos de las herramientas CASE han evolucionado.

Por un lado la posibilidad de distribuir el desarrollo de los proyectos en diferentes localizaciones y por otro la obligada necesidad de la utilización de esta plataforma como marco para el funcionamiento de la mayoría de las aplicaciones.

Otro de los grandes retos que ha aparecido y triunfado recientemente son los "smartphones" o teléfonos inteligentes.

Por el momento sólo se utilizan como plataformas de funcionamiento, pero no hay duda de que en breve formarán parte de las infraestructuras empleadas para el desarrollo del software.

A lo largo de estos períodos de tiempo fue surgiendo una importante comunidad científica en torno a la ingeniería de software.

Dos organizaciones han liderado dicha comunidad, ACM e IEEE Computer Society, que han promovido activamente la puesta en práctica de esta disciplina.

IEEE ha desarrollado la guía SWEBOK con el objeto de crear una acreditación para la profesión de ingeniero del software en Estados Unidos.

Dicha guía da forma a los conocimientos necesarios para dominar esta disciplina y los diferenciales frente a otras relacionadas con el software, como las ciencias de la computación o la gestión de proyectos de software.

1.5 Mitos falsos sobre el software

- El hardware es mucho más importante que el software.
- El software es fácil de desarrollar.
- El software consiste exclusivamente en programas ejecutables.
- El desarrollo del software es solo una labor de programación.
- Es natural que el software contenga errores.

1.6 Conclusiones

El software lo componen el conjunto de programas que gobiernan el comportamiento de cualquier sistema basado en computador.

En muchos casos el software tiene entidad en sí misma y puede ser considerado un producto propiamente dicho.

La aplicación del conocimiento y del método científico a la elaboración del software dan lugar a la disciplina que se conoce como ***ingeniería del software***

El ciclo de vida del software igual que el de cualquier otro producto que se pueda elaborar es la evolución del mismo desde el momento que se concibe hasta que se deja de utilizar

Capítulo

2

El Ciclo de Vida del Software

Centro Asociado Palma de Mallorca

Tutor: Antonio Rivero Cuesta

2 El Ciclo de Vida del Software

2.1 Introducción.

Definiremos qué es el ciclo de vida de un producto, cómo es el ciclo de vida del software, qué fases tiene y qué posibles esquemas organizativos pueden tener esas fases, dando lugar a diferentes ciclos de vida.

2.2 Objetivos

Conocer qué es el ciclo de vida de un producto Software.

Conocer las diferentes fases que integran este ciclo de vida, la documentación resultante de cada fase.

Conocer los diferentes enfoques en la organización del ciclo de desarrollo del software.

Conocer la fase de mantenimiento del software.

2.3 El ciclo de vida de un producto

A lo largo de su vida útil el software pasará por una secuencia de fases de su ciclo.

El ciclo de vida en la ingeniería pasa por una serie de etapas.

2.4 El ciclo de vida del software.

El proceso de desarrollo del software, incluyendo el mantenimiento necesario durante su explotación, se denomina **"Ciclo de Vida del Software**".

2.5 Las fases de un ciclo de vida del software

2.5.1 Análisis

Que debe hacer el sistema a desarrollar, se genera el SRD (Documento de especificación de requisitos). Especificación precisa y completa sin detalles internos.

2.5.2 Diseño

Descomponer y organizar sistema para hacer desarrollo en equipo, se genera el SDD (Documento de diseño del software). Describe estructura global, especifica que debe hacer cada parte y como se combinan.

2.5.3 Codificación

Se escribe y prueba código fuente para cada elemento, se genera el código fuente en lenguaje de programación elegido, con comentarios para que esté claro.

2.5.4 Integración

Combinar todos los componentes y hacer pruebas exhaustivas, se genera el sistema software, ejecutable junto con la documentación de las pruebas.

2.5.5 Explotación

No forma parte del ciclo de desarrollo de un producto software, pero influye en el resto de las fases.

Comprende el periodo de funcionamiento de la aplicación.

2.5.6 Mantenimiento

Durante la explotación habrá cambios por defectos no detectados o por mejoras, se generan documentos de cambios que tendrá información del problema, descripción de la solución y modificaciones realizadas.

2.6 Documentos que se generan en el ciclo de vida

Los trabajos realizados en cada una de las fases del ciclo de vida se describen formalmente en cada uno de los siguientes documentos.

2.6.1 Documento de especificación de requisitos.

El SRD (Software Requirements Document).

Especificación precisa y completa de lo que debe hacer el sistema, prescindiendo de los detalles internos de cómo lo debe hacer.

2.6.2 Documento de Diseño de Software (SDD)

El SDD (Software Design Document).

Descripción de la estructura global del sistema y la especificación de qué debe hacer cada una de sus partes así como la combinación de unas con otras.

2.6.3 Código fuente

Es el producto de la fase de codificación.

2.6.4 El sistema software

Es el producto ejecutable de la fase de integración.

2.6.5 Documentos de cambio

En caso que haya modificaciones debe quedar constancia escrita de las acciones realizadas.

2.7 Tipos de ciclo de vida del software

2.7.1 Ciclo de vida en cascada

Cada resultado de una fase es el elemento de entrada de la fase siguiente. Antes de comenzar una fase se establece un proceso de revisión para detectar errores que serian muy costosos de resolver si se pasase a la fase siguiente con ese lastre.

Las diferentes variantes del modelo se diferencian en el reconocimiento o no como fases separadas de ciertas actividades. Las fases de un ciclo de vida en cascada son: Análisis, Diseño, Codificación, Integración y Mantenimiento.

Este modelo trata de aislar cada fase de la siguiente, de manera que cada fase pueda ser desarrollada por grupos de personas diferentes, facilitando así la especialización.

Se hace énfasis en la necesidad de terminar correctamente cada fase antes de comenzar la siguiente. Esto se debe a que los errores producidos en una fase son muy costosos de corregir si ya se ha realizado el trabajo de las fases siguientes.

Para detectar los errores lo antes posible, y evitar que se propaguen a fases posteriores, se establece un proceso de revisión al completar cada fase, antes de pasar a la siguiente. Esta revisión se realiza fundamentalmente sobre la documentación producida en esa fase, y se hace de una manera formal, siguiendo una lista de comprobaciones establecida de antemano.

Si en alguna fase se detectan errores producidos en fases anteriores, será necesario rehacer parte del trabajo volviendo a un punto anterior en el ciclo de vida. Esto se indica con línea discontinua en las Figuras 2.1 y 2.2 de las Páginas 35 y 36.

2.7.2 El modelo en V

Se basa en una secuencia de fases análoga a la del modelo en cascada. Se representa con un diagrama bidimensional nivel (eje vertical) - desarrollo (eje horizontal).

En las actividades situadas en un nivel determinado se trabaja sobre una unidad del nivel de detalle superior, que se organiza en varias unidades del nivel de detalle inferior. Así durante la codificación se trabaja con un modulo que se construye con sentencias, en el diseño e integración se

trabaja con un sistema software que se organiza (durante el diseño) o se construye (durante la integración) con varios módulos.

Si nos fijamos en la Figura 2.3 de la Pagina 37 vemos como las fases iniciales, en la rama izquierda descendente, el sistema software se va descomponiendo en elementos cada vez más sencillos, hasta llegar a las sentencias del lenguaje de programación. A partir de aquí el sistema se va construyendo poco a poco a base de integrar los elementos que lo componen, siguiendo la rama derecha ascendente, hasta disponer del sistema completo listo para ser usado.

En esta misma figura podemos observar que el resultado de una fase no solo sirve como entrada para la fase inmediatamente siguiente, sino que también debe utilizarse en fases posteriores para comprobar que el desarrollo es correcto.

Podemos diferenciar los conceptos de Verificación y Validación:

Verificación: Comprobación de que una parte del sistema cumple con sus especificaciones, se produce a nivel de modulo.

Validación: Comprobación de que un elemento satisface las necesidades del usuario identificadas durante el análisis, se produce a nivel de sistema completo.

Ver Figura 2.4 Pagina 38.

2.8 Uso de prototipos.

En los modelos clásicos cada fase del desarrollo tiene una duración limitada en el tiempo, de forma que una vez terminada una fase pueden dedicarse a otra cosa los recursos humanos o materiales que se han empleado.

Esto significa que no se contempla de manera organizada las vueltas atrás debidas a errores detectados en fases anteriores.

El empleo de prototipos puede solucionar, al menos en parte, este problema.

Se definen como elementos auxiliares que permiten probar experimentalmente ciertas soluciones parciales a las necesidades del usuario o a los requisitos del sistema.

Para limitar el coste del desarrollo del prototipo, debemos de limitar sus funciones, su capacidad (permitiendo que procese solo unos pocos datos), su eficiencia, usar un hardware más potente, usar un software más potente.

2.8.1 Prototipos rápidos.

También se denominan Prototipos de Usar y Tirar y Maquetas (cuando su funcionamiento o capacidad es muy limitada).

Solo afectan a las fases de análisis y diseño. Experimentan algunas alternativas y garantizan en lo posible que las decisiones tomadas son correctas. Una vez finalizadas estas fases, el sistema final se codifica totalmente partiendo de cero. No se aprovecha el código del prototipo. Lo importante en estos prototipos es desarrollarlos en poco tiempo, para evitar alargar excesivamente la duración de las fases de análisis y diseño.Ver Figura 2.5 Pagina 40.

2.8.2 Prototipos evolutivos

En este caso el prototipo inicial se construye tras unas fases parciales de análisis y diseño. La experimentación con el prototipo permitirá avanzar en esas fases parciales, y a continuación ampliar el prototipo inicial para convertirlo en el sistema final mediante adiciones sucesivas. Al mismo tiempo los documentos de especificación, diseño, etc…, van siendo también desarrollados progresivamente.

Ver Figura 2.6 Pagina 41.

Este modelo puede considerarse como un proceso iterativo en bucle sobre el modelo en cascada, de manera que en cada iteración se hace solo una parte del desarrollo, avanzando un poco en cada fase.

Cada iteración utiliza todo lo que se ha generado en la iteración anterior y produce un nuevo prototipo, que es una nueva versión parcial del sistema, hasta llegar finalmente al sistema completo, con lo que se sale del bucle de iteraciones y termina el proceso.

2.8.3 Herramientas para la realización de prototipos

Si el prototipo es evolutivo se usaran las mismas herramientas que en el sistema definitivo, si es rápido se podrán usar diferentes. Las herramientas más idóneas para construir prototipos son:

Las de 4ª generación: Que se apoyan en bases de datos de uso general, formularios de entradas de datos, y formatos de informes.

Los lenguajes de 4ª generación: Tiene un estilo declarativo no operacional, describen el resultado que se desea obtener, en vez de describir las operaciones para conseguirlo, tales como Prolog y SmallTalk.

Los lenguajes de especificación: Tienen como objetivo formalizar la especificación de requisitos de un sistema software, y disponiendo de un compilador tendremos un prototipo ejecutable

El uso extensivo de la reutilización del software: Es otra técnica para la construcción de prototipos, que se basa en la utilización de módulos o subsistemas escritos de antemano.

2.9 El modelo en espiral.

La característica principal del modelo en espiral es la introducción de la actividad de análisis de riesgo, como elemento fundamental para guiar la evolución del proceso de desarrollo.

En la representación de este modelo podemos observar:

- **Unos Ejes Cartesianos:** cada cuadrante contiene una clase de actividad (Planificación, Análisis de Riesgo, Ingeniería y Evaluación).
- **Una Dimensión Radial:** nos indica el esfuerzo total realizado hasta cada momento. Siempre es creciente.
- **Una Dimensión Angular:** nos indica un avance relativo en el desarrollo de las actividades de cada cuadrante.
- **Ciclo de la Espiral:** en cada ciclo de la espiral se realiza una parte del desarrollo total, siguiendo la secuencia de las 4 clases de actividades.

Planificación: Sirve para establecer el contexto del desarrollo y decidir que parte del mismo se abordara en ese ciclo de la espiral.

Análisis de riesgo: Consiste en evaluar diferentes alternativas para la realización de la parte de desarrollo elegida, seleccionando la más ventajosa.

Ingeniería: Son las indicadas en los modelos clásicos: análisis, diseño, codificación, integración y mantenimiento, su resultado será ir obteniendo en cada ciclo una versión más completa del sistema.

Evaluación: Analiza los resultados de la fase de ingeniería con la colaboración del cliente y este resultado se tiene como información de entrada para la planificación del ciclo siguiente.

Tendremos diferentes variantes del modelo en espiral, según que parte del desarrollo se decida hacer en cada ciclo.

Ver Figura 2.7 Pagina 42.

2.10 Programación extrema

Los clientes suelen ser exigentes en cuanto a plazos de entrega del producto.

El objetivo de la programación extrema es ser capaz de responder de una forma rápida y de calidad a las exigencias del cliente.

El equipo de desarrollo se basa en cuatro valores principales:

- **Sencillez**: se debe programar sólo aquello que nos han pedido.
- **Comunicación**: se potencia el trabajo en equipo, dentro del grupo y con el cliente.
- **Retroalimentación**: los ciclos de proceso son muy cortos, en muy poco tiempo se evalúa el diseño y se cambia si no cumple los requisitos del cliente.
- **Valentía**: permite a los programadores afrontar la recodificación continua del programa.

2.11 Mantenimiento del software.

También llamada etapa de explotación y mantenimiento, consiste en repetir o rehacer parte de las actividades de las fases anteriores para introducir cambios en una aplicación que ha sido entregada al cliente.

2.11.1 Evolución de las aplicaciones.

Tenemos tres tipos de mantenimiento que son:

- **Correctivo**: Corrige errores que no se detectaron durante el desarrollo inicial.
- **Adaptativo**: Se da en aplicaciones que tienen un tiempo de vigencia elevado y es necesario ir adaptando la interfaz que corre sobre ellas.
- **Perfectivo**: Se da en aplicaciones en las que existe competencia de mercado, para optimizar las sucesivas versiones. También cuando las necesidades iniciales del usuario han sido modificadas.

2.11.2 Gestión de cambios.

Se pueden tener dos enfoques a la hora de hacer cambios en el software.

Como un nuevo desarrollo: Si afecta a la mayoría de los componentes, aplicando un nuevo ciclo de vida desde el principio, (aunque aprovechando lo ya desarrollado).

Como modificación de algunos elementos si afecta a una parte localizada del producto.

El cambio de código de algunos módulos puede requerir modificar los documentos de diseño o incluso en el caso del mantenimiento perfectivo, modificar el SRD.

La realización de cambios se controla mediante el informe de problema y el informe de cambio.

***Informe de Problema*:** describe una dificultad en la utilización del producto que requiere alguna modificación para subsanarla.

***Informe de Cambio*:** describe la solución dada a un problema y el cambio realizado en el producto software.

El Informe de Problema puede ser originado por los usuarios. Este informe se pasa a un grupo de Ingeniería para la comprobación y la caracterización del problema y luego a un grupo de Gestión para decidir la solución a adoptar.

Este grupo de Gestión inicia el Informe de Cambio, que se pasa de nuevo al grupo de Ingeniería para su desarrollo y ejecución.

2.11.3 Reingeniería.

Las técnicas de ingeniería inversa o reingeniería sirven para ayudar al mantenimiento de aplicaciones antiguas que no fueron desarrolladas siguiendo las técnicas de ingeniería del software, con su respectiva documentación.

Se toma el código fuente y a partir de él, se construye de forma automática alguna documentación, en particular, de diseño, con la estructura modular y las dependencias entre módulos y funciones (a veces será necesario reconstruir a mano la documentación y la modificación del código fuente).

2.12 Garantía de calidad de software

Las actividades del ciclo de vida que inciden sobre la calidad del producto software son las revisiones y pruebas.

2.12.1 Plan de garantía de calidad:

Las técnicas de Ingeniería de Software tratan de formalizar el proceso de desarrollo evitando los inconvenientes de una producción artesanal informal. Esta organización del proceso de desarrollo de software debe materializarse en un documento formal denominado Plan de Garantía de Calidad.

SQAP: Es el Plan de Garantía de Calidad del Software, debe contemplar los siguientes aspectos:

- Organización de los equipos de personas, dirección y seguimiento del desarrollo.
- Modelo de ciclo de vida a seguir (fases y actividades).
- Documentación requerida (con contenido y guión).
- Revisiones y auditorias que se harán durante el desarrollo.
- Organización de las pruebas que se harán sobre el producto a diferentes niveles.
- Organización de la etapa de mantenimiento, especificando como ha de gestionarse la realización de cambios del producto ya en explotación.

2.12.2 Revisiones

Consiste en inspeccionar el resultado de una actividad para determinar si es aceptable o tiene defectos que se deben corregir, se suelen aplicar a la documentación generada en cada fase.

Las revisiones se deben formalizar siguiendo las siguientes pautas:

- Se realizaran por un grupo de personas de 3 a 5.
- No debe ser realizada por los autores del producto.
- No se debe revisar ni el productor ni el proceso de producción, sino el producto.
- Se ha de establecer previamente una lista formal de comprobaciones a realizar, si la decisión ha de decidir la aceptación o no del producto.
- Debe levantarse Acta con los puntos de discusión y las decisiones adoptadas.

Este documento es el producto de la revisión.

2.12.3 Pruebas

Consisten en hacer funcionar un producto software bajo unas condiciones determinadas y comprobar si los resultados son los correctos.

El objetivo es descubrir los errores contenidos en el software.

Si no se descubre ningún error no se garantiza la calidad del producto. Una prueba tiene éxito si se descubre algún error, con lo que se sabe que el producto no cumple con algún criterio de calidad,

por el contrario, si la prueba no descubre ningún error , no se garantiza con ello la calidad del producto, ya que pueden existir otros errores que habrían de descubrirse mediante pruebas diferentes.

2.12.4 Gestión de configuración

La configuración de software es la manera en que diversos elementos se combinan para constituir un producto software bien organizado tanto desde el punto de vista de la explotación por el usuario como de su desarrollo o mantenimiento.

Para cubrir la doble visión del software, desde el punto de vista del usuario y del desarrollador, habremos de considerar como elementos componentes de la configuración todos los que intervienen en el desarrollo y no solo los que forman parte del producto entregado al cliente.

El problema de la Gestión de Configuración es que estos elementos evolucionan a lo largo del desarrollo y la explotación del producto software, dando lugar a diferentes configuraciones particulares, compuestas de diferentes elementos.

Los elementos individuales evolucionan a base de construir sucesivas versiones de los mismos.

Para mantener bajo control la configuración o configuraciones software hay que apoyarse en técnicas particulares de "Control de Versiones" y "Control de Cambios".

El control de versiones: Consiste en almacenar de forma organizada las sucesivas versiones de cada elemento de la configuración

El control de cambios: Consiste en garantizar que las diferentes configuraciones del software se componen de elementos compatibles entre sí. Este control se realiza normalmente usando el concepto de *Linea Base*.

Línea base: Es una configuración particular del sistema, cada línea se construye a partir de otra mediante la inclusión de ciertos cambios, que pueden ser la adición o supresión de elementos, o la sustitución de algunos por versiones nuevas de los mismos.

La aceptación de los cambios y la consiguiente creación de la nueva Línea Base ha de controlarse mediante pruebas o revisiones apropiadas, para garantizar la corrección de la nueva configuración.

Línea base congelada: Se le da este nombre a la antigua línea base que se modificó y no se borró. Se borrará cuando se esté seguro de no necesitarla más.

2.12.5 Normas y estándares

Son normas y recomendaciones sobre los diferentes aspectos del desarrollo del software.

Algunos estándares son:

- IEEE.
- DoD.
- ESA.
- ISO.
- CMMI
- MÉTRICA-2.
- MÉTRICA-3.

Capítulo

3

Especificación de Requisitos

Centro Asociado Palma de Mallorca

Tutor: Antonio Rivero Cuesta

3 Especificación de Requisitos

3.1 Introducción.

Veremos la labor de análisis y definición de los requisitos que ha de cumplir un proyecto de software.

Esta labor da lugar a la especificación del software.

3.2 Objetivos

Conocer el concepto de especificación de un sistema software, obtención, análisis y validación de requisitos.

Comprender la importancia que tiene la obtención de requisitos.

Conocer y distinguir los diferentes tipos de requisitos que se presentan en la elaboración de un sistema software.

Conocer las técnicas para la captura de requisitos y ser capaz de elaborar un SRD.

Reconocer diferentes notaciones para elaborar los diagramas que se emplean en la elaboración del SRD.

3.3 Modelado de sistemas.

El Modelado de los sistemas tiene como objetivo entender mejor el funcionamiento requerido y facilitar la comprensión de los problemas que se plantean en el nuevo sistema a desarrollar. Se establecen Modelos Conceptuales que reflejen por un lado la organización de la información y por otro las transformaciones que se deben realizar con dicha información.

3.3.1 Concepto de modelo

Un modelo conceptual es todo aquello que nos permite conseguir una abstracción lógico-matemática del mundo real, y que facilita la comprensión del problema a resolver.

Se trata de ofrecer una visión de alto nivel, sin explicar detalles concretos del mismo.

Debe explicar qué debe hacer el sistema y no como. Objetivos del modelo:

- Facilitar la comprensión del problema a resolver.
- Establecer un marco para la discusión.
- Fijar las bases para realizar el diseño.
- Facilitar la verificación del cumplimiento de los objetivos del sistema.

3.3.2 Técnicas de modelado

Técnicas útiles para realizar el modelado de un sistema software.

3.3.2.1 Descomposición, modelo jerarquizado

Cuando un problema es complejo, se descompone en otros problemas más sencillos. De esta manera se establece un Modelo Jerarquizado en el que el problema queda subdividido en un cierto número de subproblemas.

Por ejemplo: si queremos modelar un sistema para la gestión de una empresa, este sistema se puede descomponer en los subsistemas siguientes: sistema de nominas, sistema de contabilidad, etc. Este tipo de descomposición se denomina **Descomposición Horizontal** y trata de descomponer **funcionalmente** el problema.

Ahora, cada uno de los subsistemas, se pueden descomponer a su vez en otros más simples. Por ejemplo, el sistema de nominas se puede dividir en: realización de nominas, pagos a la seguridad social, etc.

Cuando se descompone un modelo tratando de detallar su **estructura,** lo denominamos **Descomposición Vertical.** Por ejemplo, la realización de nominas se descompone según la secuencia: entrada de datos del trabajador, calculo de ingresos brutos, etc.

3.3.2.2 Aproximaciones sucesivas

Es corriente que el sistema software que se quiere modelar sustituya a un proceso de trabajo ya existente, que se realiza de forma totalmente manual. En este caso se puede crear un modelo de partida basado en la forma de trabajo anterior que tendrá que ser depurado mediante aproximaciones sucesivas hasta alcanzar un modelo final.

3.3.2.3 Empleo de diversas notaciones

Lo óptimo es emplear varias notaciones juntas cuando sea necesario. También se utilizan las herramientas CASE, que son herramientas de ayuda al análisis y al diseño, que combinan texto, tablas, gráficos, diagramas etc.

3.3.2.4 Considerar varios puntos de vista

Se debe elegir el más conveniente. En ocasiones será más adecuado adoptar el punto de vista del futuro usuario del sistema, en otras será más adecuado adoptar el punto de vista del mantenedor del sistema, en algunos modelos será suficiente con perfilarlo desde el punto de vista funcional, etc.

Para la elección será conveniente esbozar distintas alternativas y elegir aquellas que resulten la más idónea.

3.3.2.5 Realizar un análisis del dominio

Entendemos por dominio el campo de aplicación en el que se encuadra el sistema a desarrollar. Por ejemplo, si tenemos que desarrollar un sistema para el seguimiento de la evolución de los pacientes de un hospital, este sistema quedara encuadrado dentro de las aplicaciones para la gestión de hospitales.

En este campo como en cualquier otro, existe desde siempre una cierta manera de realizar las cosas y una terminología ya cuñada que debe ser respetada y tenida en cuenta. A esto es lo que llamamos **Realizar un Análisis del Dominio de Aplicación**.

Para realizar este análisis es conveniente estudiar aspectos como:

- Normativa que afecte al sistema.
- Otros sistemas semejantes.
- Estudios recientes en el campo de la aplicaron, etc.

Estudiar el dominio de la aplicación nos reportará las siguientes ventajas:

Facilitar la comunicación entre analista y usuario del sistema: la creación de un nuevo modelo requiere una colaboración entre el usuario y el analista. Por ejemplo, en una aplicación que gestione un hospital, para guardar la información de cada paciente, desde siempre se emplea el término de "Historia Clínica" y resultaría confuso cambiar esta denominación por la de "Ficha del Paciente" que podría sugerir el analista.

Creación de elementos realmente significativos del sistema: por ejemplo, si tenemos que realizar una aplicación de contabilidad para una empresa determinada, es bastante normal que se adapte fielmente a las exigencias del contable de dicha empresa, lo que puede dar lugar a soluciones no validas para otras empresas. Sin embargo, si se tiene en cuenta el Plan Contable Nacional, la aplicación será válida en cualquier empresa.

Reutilización posterior del software desarrollado: por ejemplo, supongamos que queremos realizar un sistema para la gestión de una base de datos en tiempo real que necesita un tiempo de acceso que no se puede lograr con ninguna de las aplicaciones disponibles en el mercado. Para que el esfuerzo que hay que realizar no sirva solo para la aplicación concreta, lo que se debe hacer es especificar un conjunto de subrutinas de consulta y modificación que se adapten al estándar SQL. Con estas subrutinas

cualquier programa que utilice una base de datos mediante SQL podrá utilizar nuestra base de datos con un mayor tiempo de respuesta.

3.4 Análisis de requisitos software.

La etapa de análisis se encuadra dentro de la primera fase del ciclo de vida. Distinguiremos al cliente que será el encargado o encargados de elaborar junto con el analista las especificaciones del proyecto de software y de verificar el cumplimiento de las mismas.

Por ejemplo, cuando se trata de automatizar la gestión de una empresa, existe un responsable de la contabilidad, otro de los pedidos, etc. Estos son los clientes. Habrá casos donde no existirá nadie capaz de asumir el papel de cliente, bien debido a que nadie conoce exactamente lo que se requiere del sistema o bien simplemente por lo novedoso de la aplicación. En estos casos el analista debe buscar su cliente mediante una documentación exhaustiva sobre el tema.

Como se puede observar, no se asocia al cliente con la persona o entidad que financia el proyecto, aunque en ocasiones puede coincidir.

3.4.1 Objetivos del análisis

El objetivo global es obtener las especificaciones que debe cumplir el sistema a desarrollar, obteniendo un modelo valido que recoja las necesidades del cliente y las restricciones que el analista considere. En el proceso de análisis se descartan las exigencias del cliente imposibles de alcanzar o que resulten inalcanzables con los recursos puestos a disposición del proyecto.

Para que una especificación sea correcta debe tener las siguientes propiedades:

Completo y sin omisiones: tenemos que tener en cuenta que a priori no es fácil conocer todos los detalles del sistema que se pretende especificar. Por otro lado, por ejemplo , si omitimos por considerarlo sobreentendido los Sistemas Operativos o la configuración mínima que se precisara para la ejecución del sistema a desarrollar , las consecuencias pueden dar lugar a que se anule o reduzca la utilidad del sistema desarrollado finalmente.

Conciso y sin trivialidades: una documentación voluminosa suele ser una prueba de no haber sido revisada y que probablemente tenga trivialidades y repeticiones innecesarias. Por ejemplo, esto suele ocurrir cuando se elabora un nuevo modelo partiendo de otro anterior semejante. Inicialmente se mantiene todo o que no entra en contradicción con el nuevo modelo. Esto da lugar a que se mantengan párrafos del anterior que no aportan nada al nuevo modelo y que puedan dar lugar a inexactitudes.

Además, un modelo muy voluminoso no es posible estudiarlo en detalle y resulta difícil distinguir los aspectos fundamentales de aquellos que no lo son.

Sin ambigüedades: si pasamos rápidamente el análisis de requisitos para entrar de lleno en el diseño y la implementación del sistema podemos dejar en el análisis ciertos aspectos completamente ambiguos. Esto produce consecuencias negativas como, dificultades en el diseño, retrasos y errores en la implementación, etc.

Sin detalles de diseño e implementación: el objetivo del análisis es definir el QUE debe de hacer el sistema y no el COMO lo debe de hacer. Por tanto, no se debe entrar en ningún detalle del diseño o implementación del sistema en la etapa de análisis.

Fácilmente entendible por el cliente: para que el cliente este de acuerdo con el modelo fruto del análisis debe de entender el modelo para que sea capaz de discutir todos sus aspectos. Por tanto, el lenguaje que se utilice debe ser asequible para facilitar la colaboración entre el analista y el cliente.

Separando requisitos funcionales y no funcionales:

- Funcionales: serán los destinados a establecer el funcionamiento del sistema y serán el fruto de las discusiones entre analista y cliente.

- No funcionales: serán los destinados a encuadrar el sistema dentro de un entorno de trabajo, como son: capacidades mínima máxima, interfases con otros sistemas, recursos que se necesitan, seguridad, fiabilidad, mantenimiento, calidad etc.

Dividiendo y jerarquizando el modelo: para simplificar un modelo complejo se procede a dividirlo y jerarquizarlo. Esta división dará lugar a submodelos.

En la definición del modelo global del sistema, se deberá ir de lo general a lo particular.

Fijando los criterios de validación del sistema: Se realizara con carácter preliminar el Manual de Usuario del Sistema. Aunque en el Manual no se recogen todos los aspectos a validar, siempre suele ser un buen punto de partida.

3.4.2 Tarea del análisis

Para la realización correcta de un análisis de requisitos conviene dar una serie de pasos o tareas. Las tareas por su orden cronológico serán:

Estudio del sistema en su contexto: los sistemas realizados mediante software normalmente son subsistemas de otros sistemas más complejos. Por tanto, la primera tarea del análisis del sistema software será conocer el medio en el que se va a desenvolver.

Otro aspecto importante es el análisis del dominio de la aplicación que incide en la creación de sistemas con una visión más globalizadora y que facilita la reutilización posterior de alguna de sus partes.

Identificación de necesidades: el cliente tiende a solicitar del sistema todas aquellas funciones de las que en algún momento sintió la necesidad, sin pararse a pensar el grado de utilidad que tendrá en el futuro. El analista debe concretar las necesidades que se pueden cubrir con los medios disponibles y dentro del presupuesto y plazos de entrega asignados a la realización del sistema.

Se deben descartar aquellas funciones muy costosas de desarrollar y que no aportan gran cosa al sistema. Por otro lado, para las necesidades que tengan un cierto interés y que requieran más recursos de los disponibles, el analista tendrá que aportar alguna solución aunque sea incompleta, que encaje dentro de los presupuestos del sistema.

El analista debe convencer al cliente de que la solución adoptada es la mejor con los medios disponibles y nunca tratar de imponer su criterio a toda costa.

Análisis de alternativas y Estudio de viabilidad: existen infinitas soluciones para un mismo problema. El analista debe buscar la alternativa que cubra las necesidades reales detectadas y además tenga en cuenta su viabilidad tanto técnica como económica.

Si no es posible determinar un modelo que cubra todas las necesidades se deben buscar soluciones alternativas.

Establecimiento del modelo del sistema: según se van obteniendo conclusiones de las tareas anteriores, estas se deben ir plasmando en el modelo del sistema. El resultado final será un modelo del sistema global jerarquizado en el que aparecerán subsistemas que su vez tendrán que ser desarrollados hasta concretar todos los detalles del sistema.

Para la elaboración del modelo simplificar y facilitar la comunicación entre el analista, cliente y diseñador. Para ello se utilizaran todos los medios disponibles (procesadores de texto, herramientas CASE, etc.)

Elaboración del documento de especificación de requisitos: El resultado final del análisis será el documento de especificación de requisitos que servirá como punto de partida para el diseñador. Asimismo este Documento también es el encargado de fijar las condiciones de validación del sistema una vez concluido su desarrollo e implementación.

Hay que resaltar la necesidad de elaborar bien este documento para no arrastrar errores a etapas posteriores.

Revisión continuada del análisis: A lo largo del desarrollo y según aparecen problemas en el diseño y codificación, se tendrán que modificar alguno de los requisitos del sistema. Esto implica que se debe proceder a una revisión continuada del análisis y de su documento de especificación de requisitos según se producen los cambios. Si se prescinde de esta última tarea se corre el peligro de

realizar un sistema del que no se tenga ninguna especificación concreta y en consecuencia tampoco ningún medio de validar si es o no correcto el sistema finalmente desarrollado.

3.5 Notaciones para la especificación.

La especificación será fundamentalmente una descripción del modelo a desarrollar, las notaciones más frecuentes son:

3.5.1 Lenguaje natural:

Es suficiente para especificar sistemas de una complejidad pequeña, pero insuficiente para sistemas más complejos.

Sus principales inconvenientes son las imprecisiones, repeticiones e incorrecciones.

Se empleará siempre que sea necesario aclarar cualquier aspecto concreto del sistema, que no sean capaces de reflejar el resto de notaciones.

Cuando se utilice el lenguaje natural se organizaran y estructuraran los requisitos recogidos en la especificación como una cláusula de un contrato entre el analista y el cliente. Por ejemplo, se pueden agrupar los requisitos según su carácter (Funcionales, Calidad, Seguridad,…) y dentro de cada grupo se pueden numerar correlativamente las cláusulas de distintos niveles y subniveles, por ejemplo:

1. Funcionales:
   R.1.1 Modos de funcionamiento.
   R.1.2 Formatos de entrada.
   R.1.3 Formatos de salida.

El ***lenguaje natural estructurado*** es una notación más formal que el lenguaje natural, en el que se establecen ciertas reglas para la construcción de las frases que especifican acciones tipo secuencia, iteración y selección. Lo que se intenta con este lenguaje es que dentro de una misma especificación, todas las frases se construyan de igual manera.

Por ejemplo, en distintos apartados de la especificación podríamos escribir:

Cuando se teclee la clave 3 veces mal, se debe invalidar la tarjeta…

Cuando el saldo sea menor que cero pesetas se aplicara un interés…

Sin embargo sería mejor que todas las frases se construyan de igual manera, como:

SI se teclea la clave 3 veces mal ENTONCES invalidar tarjeta…

SI el saldo es menor de cero pesetas ENTONCES el interés será…

3.5.2 Diagrama de flujo de datos

Un sistema software se puede modelar mediante el flujo de datos que entran, su transformación y el flujo de datos de salida. Los símbolos que se utilizan son: (Ver Figura 3.1. Pag.72).

***Flecha*:** indica el sentido del flujo de datos. Con cada flecha se tienen que detallar sus datos mediante un nombre o una descripción.

***Círculo o burbuja*:** es un proceso o transformación de datos. Dentro del círculo se describe el proceso que realiza.

***Línea* doble:** indica un almacén de datos. Estos datos pueden ser guardados, consultados o consumidos por uno o varios procesos. Entre las dos líneas se detalla el nombre de los datos que guarda el almacén.

***Rectángulo*:** es una entidad externa al sistema software que produce o consume sus datos, por ejemplo, el usuario, otro programa, un dispositivo hardware, etc.

A esta notación la denominaremos **DFD** (Diagrama de Flujo de Datos). Al **DFD del sistema Global** también se le denomina **DFD de Contexto**.

El DFD de Contexto nunca es suficiente para describir el modelo del sistema que se trata de especificar. Para evitar esto los DFD se usan de forma jerarquizada por niveles.

DFD de contexto: Es el del sistema global, se llama de nivel 0.

DFD 0 o nivel 1: Es el resultado de explotar el DFD de contexto.

DFD 1 hasta DFD n o nivel 2: Es el resultado de la explotación de los procesos anteriores, dando lugar a otros subprocesos.

Para facilitar la identificación de los sucesivos DFD se enumeran de forma correlativa los procesos de la forma siguiente.

Nivel 0 Diagrama de Contexto.

Nivel 1 DFD 0.

Nivel 2 DFD 1 hasta DFD n,
de los procesos 1 hasta n del nivel 1.

Nivel 3 DFD 1.1 hasta DFD 1.i,
de los procesos 1.1 hasta 1.i del nivel 2.

DFD 2.1 hasta DFD 2.j,
de los procesos 2.1 hasta 2.j del nivel 2.

……

DFD n.1 hasta DFD n.k ,
de los procesos n.1 hasta n.k del nivel 2.

Nivel 4 DFD 1.1.1 hasta DFD 1.1.x

……

DFD 1.1.1 hasta DFD 1.i.z

……

…etc…

En todos ellos los flujos de entrada y salida antes de la explosión del proceso debe coincidir con los flujos de entrada y salida del DFD resultado de la explosión.

La ventaja de esta notación es su simplicidad lo que permite que sea fácilmente entendida por todos: cliente, usuario, analista, etc.

Además de esto, también es necesario describir las estructuras de cada uno de los flujos de datos y las estructuras de los datos guardados en cada uno de los almacenes de datos.

En líneas generales, se trata de describir mediante una tabla, todos los elementos básicos de información que constituyen los diversos datos que se manejan en los distintos DFD del modelo del sistema.

Los DFD presentan algunos aspectos importantes:

Sirven para establecer un modelo conceptual del sistema que facilita la estructuración de su especificación.

El modelo desarrollado es fundamentalmente estático ya que refleja los procesos necesarios para su desarrollo y la interrelación entre ellos.

Mediante un DFD nunca se puede establecer la dinámica o secuencia en que se ejecutaran los procesos.

A los DFD se le puede asociar un carácter dinámico ya que se utiliza un modelo abstracto de cómputo del tipo flujo de datos. Así, en cada ejecución se utiliza y consume un elemento de cada uno de los flujos de datos de entrada y se generan los elementos de datos para cada uno de los flujos de salida.

En el caso de los almacenes de datos, los elementos se utilizan pero no se consumen y pueden ser utilizados posteriormente.

3.5.3 Diagramas de transición de estados:

El número de estados posibles que se pueden dar en un sistema software crece de una forma exponencial con su complejidad. Todos estos estados y las sucesivas transiciones entre ellos configuran la dinámica del sistema que se produce mientras se está ejecutando.

Para especificar un sistema únicamente es necesario resaltar aquellos estados que tengan cierta trascendencia desde un punto de vista funcional.

Podemos definir un Diagrama de Transición de Estados como la notación específica para describir el comportamiento dinámico del sistema a partir de los estados elegidos como más importantes. Esta notación se complementa con los diagramas de flujo de datos y ofrece un punto de vista del sistema imprescindible en la mayoría de los casos.

Los símbolos que se utilizan son:

Estado: se suele representar mediante un rectángulo (o círculo). Se indica mediante el nombre que encierra en su interior el estado concreto que representa.

Eventos: provocan el cambio de estado. Se indican mediante una flecha dirigida desde el estado viejo al nuevo. Se emplean flechas en forma de arco cuando se utilizan círculos y formadas por tramos cuando se utiliza un rectángulo.

Estados Inicial y Final: si quieren resaltar los Estados Inicial y Final del sistema, se representan con dos círculos concéntricos.

Cuando la complejidad del sistema lo requiera se utilizaran otros diagramas de estado, además del diagrama de estados global del sistema, para especificar la dinámica de los procesos más significativos.

3.5.4 Descripciones funcionales. Pseudocódigo

Los requisitos se deberán expresar como mínimo en un lenguaje natural estructurado y a ser posible en pseudocódigo que es una notación basada en un lenguaje de programación estructurado, del que se excluyen todos los aspectos de declaración de constantes, tipos, variables y subprogramas. El Pseudocódigo maneja datos en general, no tiene una sintaxis estricta y se pueden incluir descripciones en lenguaje natural siempre que se considere necesario, como parte del pseudocódigo.

Debemos tener especial cuidado al utilizar Pseudocódigo porque si se detalla demasiado una descripción funcional se puede estar realizando el diseño del sistema e incluso la codificación. De hecho, existen notaciones similares al pseudocódigo que están pensadas para realizar el diseño como son los lenguajes de descripción de programas (PDL).

Los objetivos que persiguen Pseudocódigo y PDL son muy diferentes.

Cuando se especifica se trata de indicar cuál es la naturaleza del problema a resolver sin detallar como se debe resolver. Por ello, en una especificación no se debería establecer ninguna forma concreta de organización de la información ni tampoco ninguna propuesta concreta de los procedimientos de resolución de los problemas.

3.5.5 Descripción de datos:

Se trata de detallar la estructura interna de los datos que maneja el sistema. Solo se deben describir aquellos datos que resulten relevantes para entender QUE debe hacer el sistema.

La notación adoptada es lo que se conoce como diccionario de datos en el que se describirán:

Nombre o nombres: Los que toma el dato en la especificación.

Utilidad: Se indican los procesos, descripciones funcionales y almacenes donde se use el dato.

Estructura: Se indican los elementos de los que está constituido el dato, con arreglo a la siguiente notación:

A + B: Secuencia o concatenación de los elementos A y B.

[A | B]: Selección entre los distintos elementos A o bien B.

{A} ^{N}: Repetición de N veces del elemento A.

(A): Opcionalmente se podrá incluir el elemento A.

/descripción/: Descripción en lenguaje natural como comentarios.

=: Separador entre el nombre de un elemento y su descripción. Ver Páginas 83 y 84.

3.5.6 Diagrama de modelo de datos

El modelo de E-R (Entidad Relación) permite definir todos los datos que manejará el sistema junto con las relaciones que se desea que existan entre ellos. Como cualquier elemento de la especificación también estará sujeto a revisiones durante el diseño y codificación. La notación que se utiliza es la siguiente:

- **Entidades o datos:** se encierran dentro de un rectángulo.
- **Relaciones:** se indican mediante un rombo que encierra el tipo de relación.
- **Flechas:** las Entidades y la Relación que se establece entre ellas se indican uniéndolas todas mediante líneas. Opcionalmente, se puede indicar el sentido de la relación con una flecha.

Cardinalidad de la Relación: es entre que valores mínimos y máximos se mueve la relación entre entidades. Tiene la siguiente nomenclatura:

- **Circulo:** indica 0.
- **Raya perpendicular a la línea de relación:** indica 1.
- **Tres Rayas en Bifurcación:** indica muchos o N.

La cardinalidad se dibuja siempre unida a la segunda entidad de la relación con un símbolo para el valor mínimo y otro para el máximo.

3.6 Documento de especificación de requisitos.

El SRD (Software Requirements Document) o SRS (Software Requirements Specification) es un documento fundamental que recogerá todo lo realizado durante la etapa de análisis y servirá como punto de partida para cualquier sistema software.

La mejor manera de redactarlo es en forma de contrato con sus cláusulas organizadas según el carácter de los requisitos para facilitar la revisión y la verificación. Sus partes son:

3.6.1.1 Introducción

Esta sección debe dar una visión general de todo el documento SRD.

- **Objetivo:** expone:
  - El objetivo del proyecto (brevemente).
  - A quien va dirigido.
  - Los participantes.
  - El calendario previsto.
- **Ámbito:**
  - Se identifica y nombra el producto resultante del Proyecto.
  - Se explica qué hace el producto obtenido.

- Se explica que NO será capaz de hacer el producto (si se considera necesario).
- Se detallan las posibles aplicaciones y beneficios del Proyecto.

- **Definiciones, siglas y abreviaturas:** se incluirá un glosario con las definiciones de términos, siglas y abreviaturas particulares utilizados en el documento y que convenga reseñar para facilitar su lectura o evitar ambigüedades. Esta información también podrá figurar como apéndice al final del documento.

- **Referencias:** si el documento contiene referencias a otros documentos, se detallará la descripción bibliográfica de los documentos referenciados y la manera de acceder a los mismos. Esta información también podrá figurar como apéndice al final del documento.

- **Panorámica del documento:** se describe la organización y el contenido del resto del documento.

3.6.1.2 Descripción general

Se da una visión general del sistema, ampliando el contenido de la sección de introducción.

Relación con otros proyectos: se describen las analogías y diferencias de este proyecto con otros similares o complementarios, o con otros sistemas ya existentes o en desarrollo. Si no hay proyectos o sistemas relacionados, se indicará "No aplicable".

Relación con otros proyectos anteriores y posteriores: se indica si este proyecto es continuación de otro o si se continuará el desarrollo en proyectos posteriores. Si no hay proyectos o sistemas relacionados, se indicará "No aplicable".

Objetivo y funciones: se describe el sistema con los objetivos y funciones principales.

Consideraciones de entorno: se describen las características especiales que debe tener el entorno en que se utilice el sistema a desarrollar. Si no hay características especiales, se indicará "No existen".

Relaciones con otros sistemas: se describen las conexiones del sistema con otros sistemas, si debe funcionar integrado con ellos o utilizando entradas o salidas indirectas de información. Si el sistema no necesita intercambiar información con ningún otro, se indicará "No existen".

Restricciones generales: se describen las restricciones generales a tener en cuenta en el diseño y desarrollo del sistema, tales como el empleo de determinadas metodologías de desarrollo, lenguajes de programación, normas particulares, restricciones hardware, de sistema operativo, etc.

Descripción del modelo: se describe el Modelo Conceptual propuesto para desarrollar el sistema en su conjunto y para cada una de sus partes más relevantes. Este modelo se puede realizar utilizando todas las notaciones y herramientas disponibles.

3.6.1.3 Requisitos específicos

Es la sección fundamental del documento. Debe contener completa y detalladamente los requisitos que debe cumplir el sistema a desarrollar.

Es importante no incluir aspectos de diseño o desarrollo. Cada requisito debe ir acompañado del grado de cumplimiento necesario, es decir, si es obligatorio, recomendable u opcional.

Si en algún apartado no hay requisitos, se indicará "No existe".

Requisitos funcionales: describen las funciones o el QUÉ debe hacer el sistema y están muy ligados al modelo conceptual propuesto. Se concretarán las operaciones de tratamiento de información, generación de informe, cálculos estadísticos, operaciones, etc.

Requisitos de capacidad: son los referentes a los volúmenes de información a procesar, tiempo de respuesta, tamaño de ficheros o discos, etc. Se deben expresar mediante valores numéricos e incluso cuando sea necesario, se darán valores para el peor, el mejor y el caso más habitual.

Requisitos de interfase: son los referentes a cualquier conexión a otros sistemas (hardware o software) con los que se debe interactuar o comunicar. Se incluirán por tanto, bases de datos,

protocolos, formatos de ficheros, sistemas operativos, datos, etc. a intercambiar con otros sistemas o aplicaciones.

Requisitos de operación: son los referentes al uso del sistema en general e incluyen los requisitos de la interfase de usuario (menús de pantalla, manejo de ratón, teclado, etc), el arranque y parada, copias de seguridad, requisitos de instalación y configuración.

Requisitos de recursos: son los referentes a elementos hardware, software, instalaciones, etc. necesarios para el funcionamiento del sistema. Se deben estimar los recursos con cierto coeficiente de seguridad en previsión de necesidades de última hora no previstas inicialmente.

Requisitos de verificación: son los que debe cumplir el sistema para que sea posible verificar y certificar que funciona correctamente (funciones de autotest, emulación, simulación, etc.).

Requisitos de pruebas de aceptación: son los que debe cumplir las pruebas de aceptación a que se someterá el sistema.

Requisitos de documentación: son los referentes a la documentación que debe formar parte del producto a entregar.

Requisitos de seguridad: son los referentes a la protección del sistema contra la manipulación indebida (confidencialidad, integridad, virus, etc).

Requisitos de transportabilidad: son los referentes a la posible utilización del sistema en diversos entornos o sistemas operativos de una forma sencilla y barata.

Requisitos de calidad: son los referentes a aspectos de calidad que no se incluyan en otros apartados.

Requisitos de fiabilidad: son los referentes al límite aceptable de fallos o caídas durante la operación del sistema.

Requisitos de mantenibilidad: son los que debe cumplir el sistema para que se pueda realizar adecuadamente su mantenimiento durante la fase de explotación.

Requisitos de salvaguarda: son los que debe cumplir el sistema para evitar que los errores en el funcionamiento o la operación del sistema tengan consecuencias graves en los equipos o las personas.

3.6.1.4 Apéndices

Se incluirán todos aquellos elementos que completen el contenido del documento, y que no estén recogidos en otros documentos accesibles a los que pueda hacerse referencia.

Ver ejemplos páginas 95 y 100.

Capítulo

4

Fundamentos del Diseño de Software

Centro Asociado Palma de Mallorca

Tutor: Antonio Rivero Cuesta

4 Fundamentos del Diseño de Software

4.1 Introducción.

En este tema se inicia el estudio de las etapas de desarrollo. Después de haber especificado QUE se quiere resolver durante la especificación, las etapas de desarrollo se dedican a determinar COMO se debe resolver el proyecto.

La primera de estas etapas es la de diseño, se continua con la de codificación y se finaliza con las etapas de pruebas del software realizado. En este tema se abordan los fundamentos de la etapa de diseño.

Diseño de software: consiste en definir y formalizar la estructura del sistema con el suficiente detalle como para permitir su realización física.

El punto de partida principal para abordar el diseño es el documento de especificación de requisitos (SRD).

En la realización del diseño podemos destacar algunas características:

Es un proceso creativo que no es nada trivial.

Casi siempre se lleva a cabo de una forma iterativa mediante prueba y error.

Es muy importante la experiencia previa. El método más eficaz es participar en algún diseño y aprender de otros diseñadores sus técnicas de trabajo. También es aconsejable estudiar diseños ya realizados y analizar las razones por las que se adopta una u otra solución.

Se tratara de reutilizar el mayor número de módulos o elementos ya desarrollados. Si no es posible por tratarse de un diseño completamente original , al menos se podrá aprovechar el enfoque dado a algún otro proyecto anterior, lo que se conoce como aprovechar el KNOW-HOW (Saber hacer).

4.2 Objetivos

Qué entendemos por diseño y analizar las actividades que se deben realizar para llevarlo a cabo.

Conocer conceptos fundamentales a tener en cuenta para realizar cualquier diseño.

Conocer distintas notaciones.

Conocer los documentos de diseño arquitectónico y del diseño detallado.

4.3 ¿Qué es el diseño?

Objetivo fundamental del diseño: Es conseguir que el software sea fácil de mantener, y si es posible reutilizarlo en futuras aplicaciones. Para conseguir ambos objetivos, la etapa de diseño es la más importante de la fase de desarrollo de software.

Durante el diseño se tiene que pasar gradualmente de las ideas informales recogidas en el SRD a definiciones detalladas y precisas para la realización del software mediante refinamientos sucesivos.

Actividades habituales en el diseño de un Sistema:

Diseño arquitectónico: Se abordan aspectos estructurales y de organización del sistema y su posible subdivisión en subsistemas o módulos. Además se tienen que establecer las relaciones entre los subsistemas creados y definir las interfaces entre ellos.

Diseño detallado: Se aborda la estructura de cada uno de los módulos en los que quedó subdividido el sistema global. Si por ejemplo utilizáramos Modula-2 como notación de diseño, en esta actividad elaboraríamos los módulos de definición para cada uno de los módulos del programa global.

Diseño procedimental: Se abordan la organización de las operaciones o servicios que ofrecerá cada uno de los módulos. Siguiendo con el ejemplo de Modula-2, en esta actividad se diseñan (en pseudocódigo) los algoritmos más importantes que se emplean en los módulos de implementación.

Diseño de datos: Se aborda la organización de la base de datos del sistema. Se parte del diccionario de datos y de los diagramas E - R de la especificación del sistema. Con esta actividad se trata de concretar el formato exacto para cada dato y la organización que debe existir entre ellos. El diseño de datos es muy importante para conseguir que el sistema sea utilizable y fácilmente mantenible.

Diseño de la interfaz de usuario: Aborda la organización de la interfaz del usuario, para conseguir un diseño más ergonómico. Es decir, se intenta conseguir un dialogo mas ergonómico entre el usuario y el computador.

El resultado de todas estas actividades de diseño es el **Documento de Diseño de Software (SDD)** que contendrá una especificación lo más formal posible de la estructura global del sistema y de cada uno de los elementos del mismo.

Si la complejidad del sistema así lo aconseja se podrán utilizar varios documentos para describir de forma jerarquizada la estructura global del sistema, la estructura de los elementos de un primer nivel de detalle y en los sucesivos niveles de detalle hasta alcanzar el nivel de codificación con los listados de los programas.

Según las normas ESA tenemos 2 documentos de diseño:

- **ADD:** Documento de Diseño Arquitectónico.
- **DDD:** Documento de Diseño Detallado, al que se le puede añadir como apéndices los listados de los programas una vez completado el desarrollo.

4.4 Conceptos de base

A continuación se listan los conceptos más interesantes a tener en cuenta en cualquier diseño:

4.4.1 Abstracción

Cuando se diseña un nuevo sistema software es importante identificar los elementos significativos de los que consta y abstraer la utilidad específica de cada uno, incluso más allá del sistema software para el que se está diseñando. Gracias a esto el elemento diseñado podrá ser sustituido en el futuro por otro con mejores prestaciones y también podrá ser reutilizado en otros proyectos similares. Por tanto, se cumplen los dos objetivos principales del diseño que son, conseguir elementos fácilmente mantenibles y reutilizables.

Durante el proceso de diseño se debe aplicar el concepto de abstracción en todos los niveles de diseño. Por ejemplo, para el sistema de control de acceso del tema anterior tenemos:

En un primer nivel aparecen abstracciones tales como Tarjeta, Mensajes, Órdenes, etc. Inicialmente todos ellos son elementos abstractos y su diseño se puede realizar sin tener en cuenta al Sistema de Control de Acceso concreto en el que se utilizaran. Con esto conseguimos:

- Facilitar los cambios futuros.
- Posibilidad de reutilizarlo en diversos sistemas.

En un segundo nivel aparecen nuevas abstracciones como Clave, Control de Puerta, Comprobar Clave, etc. a los cuales se aplicaran los mismos criterios.

Este proceso de abstracción se debe continuar con cada elemento software que tengamos que diseñar.

En el diseño de los elementos software se pueden utilizar fundamentalmente tres formas de abstracción:

Abstracciones funcionales: sirven para crear expresiones o acciones parametrizadas mediante el empleo de funciones o procedimientos. Para diseñar una abstracción funcional es necesario fijar:

Los parámetros o argumentos que se le deben pasar.

El resultado que devolverá en el caso de una expresión parametrizada.

Lo que se pretende que resuelva.

Como lo debe resolver, es decir, el algoritmo que debe emplear.

Por ejemplo, se puede diseñar una abstracción funcional para "Comprobar Clave" cuyo parámetro es la clave a comprobar y que devuelva como resultado SI/NO la clave es correcta.

Tipos abstractos: sirven para crear los nuevos tipos de datos que se necesitan para abordar el diseño del sistema. Por ejemplo, un tipo Tarjeta para guardar toda la información relacionada con la tarjeta utilizada: clase de tarjeta, identificador, clave de acceso, datos personales, etc.

Además de esto, junto al nuevo tipo de dato se deben diseñar todos los métodos u operaciones que se pueden realizar con él. Por ejemplo, Leer Tarjeta, Grabar Tarjeta, Comprobar Tarjeta, etc.

Máquinas abstractas: permiten establecer un nivel de abstracción superior a los dos anteriores y en él se define de una manera formal el comportamiento de una maquina. Un ejemplo de este tipo de abstracción seria un intérprete de SQL para la gestión de una base de datos. El lenguaje SQL establece formalmente el comportamiento perfectamente definido para la maquina abstracta encargada de manejar la base de datos: construcción, búsqueda e inserción de elementos en la base de datos. La tarea de diseño consiste en este caso en definir formalmente la máquina abstracta que se necesita.

4.4.2 Modularidad

Uno de los primeros pasos que se debe dar al abordar un diseño es dividir el sistema en sus correspondientes módulos o partes claramente diferenciadas. Esta división permite encargar a personas diferentes el desarrollo de cada módulo y que todas ellas puedan trabajar simultáneamente. Podemos citar como ventajas de utilizar un diseño modular las siguientes:

Claridad: siempre es más fácil entender y manejar cada una de las partes de un sistema que tratar de entenderlo como un todo compacto.

Reducción de Costos: resulta más barato desarrollar, depurar, documentar, probar y mantener un sistema modular que otro que no lo es, excepto si el número de módulos crece innecesariamente.

Reutilización: si los módulos se diseñan teniendo en cuenta otras posibles aplicaciones resultara inmediata su reutilización.

La modularidad es un concepto de diseño que no debe estar ligado a la etapa de codificación y mucho menos al empleo de un determinado lenguaje de programación.

4.5 Refinamiento

En un diseño siempre se parte inicialmente de una idea no muy concreta que se va refinando en sucesivas aproximaciones hasta perfilar el más mínimo detalle.

El objetivo global de un nuevo sistema software expresado en su especificación se debe refinar en sucesivos pasos hasta que todo quede expresado en el lenguaje de programación del computador. Es decir, a través del refinamiento pasaremos de la especificación del SRD al lenguaje de programación del computador.

El proceso de refinamiento se puede dar por concluido cuando todas las acciones y expresiones quedan refinadas en función de otras acciones o expresiones o bien en función de las instrucciones básicas del lenguaje empleado.

4.5.1 Estructuras de datos

Para el diseño deberemos tener en cuenta las estructuras fundamentales que son: Registros, Conjuntos, Formaciones, Listas, Pilas, Colas, Árboles, Grafos, Tablas, Ficheros.

Es labor del diseñador la búsqueda, a partir de estas estructuras básicas, de la combinación más adecuada para lograr aquella estructura o estructuras que den respuesta a las necesidades del sistema planteadas en su especificación.

4.5.2 Ocultación

Para poder utilizar correctamente un programa no es necesario conocer cuál es su estructura interna. Por tanto, al programador "usuario" de un módulo desarrollado por otro programador del equipo puede

quedarle completamente oculta la organización de los datos internos que maneja y el detalle de los algoritmos que emplea.

Cuando se diseña la estructura de cada uno de los módulos de un sistema, se debe hacer de tal manera que dentro de él queden ocultos todos los detalles que resultan irrelevantes para su utilización.

Con carácter general, se trata de ocultar al usuario todo lo que pueda ser susceptible de cambio en el futuro y además es irrelevante para el uso. Sin embargo, se muestra en la interfaz solo aquello que resultará invariable con cualquier cambio. Las ventajas de aplicar este concepto son las siguientes:

Depuración: resulta más sencillo detectar que módulo concreto no funciona correctamente. También es más fácil establecer estrategias y programas de prueba que verifiquen y depuren cada módulo por separado en base a lo que hacen sin tener en cuenta como lo hacen.

Mantenimiento: cualquier modificación u operación de mantenimiento que se necesite en un módulo concreto no afectara al resto de los módulos del sistema.

4.5.3 Genericidad

En este caso, un posible enfoque de diseño es, agrupar aquellos elementos del sistema que utilizan estructuras semejantes o que necesitan de un tratamiento similar. Posteriormente, cada uno de los elementos agrupados se puede diseñar como un caso particular del elemento genérico.

Por ejemplo, supongamos que tenemos que diseñar un sistema operativo multitarea que necesita atender las órdenes que llegan de distintos terminales. Las órdenes habituales son:

- Imprimir por cualquiera de las impresoras disponibles.
- Acceder desde los terminales a ficheros y bases de datos compartidas.

Cada una de estas actividades necesita de un módulo gestor para decidir en qué orden se atienden las peticiones. La forma más sencilla de gestión es poner en cola las órdenes, peticiones de impresión o peticiones de acceso a ficheros o bases de datos compartidas. Como vemos, surge la necesidad de una estructura genérica en forma de cola para la que se necesitan también unas operaciones genéricas tales como poner en cola, sacar el primero, etc. Ahora, en cada caso el tipo de información que se guarda en la cola es diferente: tipo de orden a ejecutar, texto a imprimir, etc. A partir de la cola genérica y con un diseño posterior más detallado se tendrá que decidir la estructura concreta de las distintas colas necesarias y si para alguna de ellas es conveniente utilizar prioridades, lo que daría lugar a operaciones especificas.

A pesar de todo, tenemos que tener en cuenta que en algunos lenguajes de programación el concepto de genericidad se puede ver desvirtuado. Por ejemplo, Modula-2 impone restricciones muy fuertes a la hora de manejar datos de distinto tipo y las operaciones entre ellos. En este caso sería necesario definir un tipo distinto de cola para cada tipo de elemento a almacenar, y aunque las operaciones sean esencialmente idénticas para los distintos datos almacenados, también es necesario implementar como operaciones distintas las destinadas a manejar cada tipo de cola. Si tuviéramos, por ejemplo los tipos:

- Cola_de_enteros.
- Cola_de_reales.

Sería necesario implementar las operaciones:

- Poner_entero (en la cola de enteros).
- Poner _real (en la cola de reales).

Como vemos, esta solución que es la única posible en estos lenguajes, desvirtúa de forma evidente la genericidad propuesta en el diseño. En estos casos convendría utilizar lenguajes que tengan la posibilidad de declarar elementos genéricos como por ejemplo ADA.

4.5.4 Herencia

Otro enfoque posible del diseño cuando hay elementos con características comunes es establecer una clasificación o jerarquía entre esos elementos del sistema partiendo de un elemento "padre" que posee una estructura y operaciones básicas. Los elementos "hijos" heredan del "padre" su estructura y operaciones para ampliarlos, mejorarlos o simplemente adaptarlos a sus necesidades. A su vez los elementos "hijo" pueden tener otros "hijos" que hereden de ellos de una forma semejante. De manera consecutiva se puede continuar con los siguientes descendientes hasta donde sea necesario. Esta es la idea fundamental en la que se basa el concepto de herencia.

El mecanismo de herencia tiene como objetivo fundamental facilitar la reutilización de software ya desarrollado.

Por ejemplo, supongamos que tratamos de realizar un software para dibujo asistido por computador. Las figuras que se manejan se podrían clasificar del siguiente modo

FIGURAS:

ABIERTAS:

TRAZO RECTO:

LINEA RECTA.

TRAZO CURVO:

SEGMENTO DE CIRCUNFERENCIA.

CERRADAS:

ELIPSES:

CIRCULOS.

POLIGONOS:

TRIANGULOS:

EQUILATEROS.

RECTANGULOS:

CUADRADOS.

En este ejemplo, el elemento "padre" será FIGURAS. Las operaciones básicas con cualquier tipo de figura podrán ser:

- Desplazar.
- Rotar.
- Pintar.

Tendríamos que tener en cuenta dos puntos:

Aunque estas operaciones las heredan todos los tipos de figura, deberán ser adaptadas en cada caso. Así, Rotar los CIRCULOS significa dejarlos tal cual están.

Al concretar los elementos "hijos" aparecen nuevas operaciones que no tenían sentido en el elemento "padre". Así, la operación de calcular el Perímetro solo tiene sentido para figuras CERRADAS., etc.

El concepto de herencia está muy ligado a las metodologías de análisis y diseño de software orientado a objetos. Además, es posible realizar diseños en los que se tengan en cuenta herencias múltiples de varios "padres".

4.5.5 Polimorfismo

El concepto de polimorfismo engloba distintas posibilidades utilizadas habitualmente para conseguir que un mismo elemento software adquiera varias formas simultáneamente:

El ***concepto de genericidad*** es una manera de lograr que un elemento genérico pueda adquirir distintas formas cuando se particulariza su utilización.

El ***concepto de Polimorfismo*** está muy unido al concepto de herencia. Hay tres tipos:

De Anulación: las estructuras y operaciones heredadas se pueden adaptar a las necesidades concretas del elemento "hijo". En el ejemplo de antes no sería igual Rotar Elipses (Padre) que Círculos (hijo). Por tanto, la operación de rotar tiene distintas formas según el tipo de figura a la que se destina y es en el momento de la ejecución del programa cuando se utiliza una u otra forma de rotación. Este tipo de Polimorfismo se conoce como de Anulación dado que la rotación específica para los círculos anula la más general para las elipses.

Diferido: Cada uno de los hijos concreta la forma específica de la operación del padre. Por ejemplo, la operación Rotar FIGURAS resultara muy compleja y probablemente inútil.

Sobrecarga: Ahora son los operadores, funciones y procedimientos los que toman múltiples formas. Por ejemplo, los operadores +,-,*,/ son similares para operaciones entre enteros , reales , conjuntos o matrices. Sin embargo en cada caso el tipo de operación que se invoca es distinto:

La suma entre enteros es mucho más sencilla y rápida que la suma entre reales.

Con el mismo operador + se indica la operación suma entre valores numéricos o la operación de unión entre dos conjuntos o la suma de matrices.

Este mismo concepto (Sobrecarga) se debe aplicar cuando se diseñan las operaciones a realizar entre elementos del sistema que se trata de desarrollar.

Por ejemplo, se puede utilizar el operador + para unir ristras de caracteres o realizar resúmenes de ventas.

También se puede utilizar este tipo de polimorfismo con funciones y procedimientos. Por ejemplo, podemos tener la función Pintar para FIGURAS y la función Pintar para representar en una grafica los valores de una tabla.

Todas estas posibilidades de Polimorfismo redundan en una mayor facilidad para realizar software reutilizable y mantenible.

4.5.6 Concurrencia

Se aprovecha la capacidad de proceso del computador, ejecutando tareas de forma concurrente. Si se diseña un sistema con restricciones de tiempo hay que tener en cuenta lo siguiente:

Tareas concurrentes: ver que tareas se deben ejecutar en paralelo para respetar las restricciones impuestas. Se deberá prestar especial atención a aquellas tareas con tiempos de respuesta más críticas y aquellas otras que se ejecutan con mayor frecuencia.

Sincronización de tareas: determinar los puntos de sincronización entre las distintas tareas con semáforos o monitores.

Comunicación entre las tareas: distinguir si la cooperación se basa en datos compartidos o en paso de mensajes entre las tareas. Si se utilizan datos compartidos se tendrá que evitar que los datos puedan ser modificados en el momento de la consulta. En este caso es necesario utilizar mecanismos como semáforos, monitores, regiones críticas, etc.

Para garantizar la exclusión mutua entre las distintas tareas que modifican y consultan los datos compartidos.

Interbloqueo**:** estudiar los posibles interbloqueos entre tareas. Un interbloqueo se produce cuando una o varias tareas permanecen esperando por tiempo indefinido una situación que no se puede producir nunca.

El concepto de concurrencia introduce una complejidad adicional al sistema y por tanto solo se debe utilizar cuando no exista una solución de tipo secuencial sencilla que cumpla con los requisitos especificados en el documento SRD.

4.6 Notaciones para el diseño.

El objetivo fundamental de cualquier notación es resultar precisa, clara y sencilla de interpretar, evitando ambigüedades.

Según el aspecto del sistema que se trata de describir tenemos varios tipos:

4.6.1 Notaciones estructurales

Estas notaciones sirven para cubrir un primer nivel del diseño arquitectónico. Se trata de desglosar y estructurar el sistema en sus partes fundamentales. Podemos diferenciar dos notaciones habituales para desglosar un sistema en sus partes:

Diagrama de bloques: en la figura 4.1 de la pagina 133 se muestra el diagrama de bloques de un sistema que está dividido en cuatro bloques y en el que además se indican las conexiones que existen entre ellos. En algunos casos estas conexiones también pueden reflejar una cierta clasificación o jerarquía de los bloques.

Cajas adosadas: en la figura 4.2 de la pagina 134 se emplean "cajas adosadas" para delimitar los bloques. La conexión entre los bloques se pone de manifiesto cuando entre dos cajas existe una frontera común.

4.6.1.1 Diagramas de estructura

Estos diagramas fueron propuestos para describir la estructura de los sistemas software como una jerarquía de subprogramas o módulos en general. En la Figura 4.3 de la página 135 se muestra un Diagrama de Estructura.

En este diagrama podemos observar las siguientes figuras:

RECTÁNGULOS: representa un módulo o subprograma cuyo nombre se indica en su interior.

LÍNEA: une a dos rectángulos e indica que el módulo superior llama o utiliza al módulo inferior. A veces la línea acaba en una flecha junto al módulo inferior al que apunta.

ROMBO: se sitúa sobre una línea e indica que esa llamada es opcional. Se puede prescindir de este símbolo si en la posterior descripción del módulo superior, mediante pseudocódigo u otra notación, se indica que el módulo inferior se utiliza sólo opcionalmente.

ARCO: se sitúa sobre una Línea e indica que esa llamada se efectúa de manera repetitiva. Se puede prescindir de ese símbolo si en la posterior descripción del módulo superior, mediante pseudocódigo u otra notación, se indica que el módulo inferior se utiliza de forma repetitiva.

CIRCULO CON FLECHA: se sitúa en paralelo a una Línea y representa el envío de los datos, cuyo nombre acompaña al símbolo, desde un módulo a otro. El sentido del envío lo marca la flecha que acompaña al círculo. Para indicar que los datos son una información de control se utiliza un círculo relleno. Una información de control sirve para indicar SI/NO, o bien un estado: Correcto / Repetir / Error/ Espera / Desconectado /…

El Diagrama de Estructura no establece ninguna secuencia concreta de utilización de los módulos y tan solo refleja una organización estática de los mismos.

4.6.1.2 Diagramas HIPO

Notación propuesta por IBM para facilitar y simplificar el diseño y desarrollo de sistemas software fundamentalmente de gestión (facturación, contabilidad, etc.).

Podemos destacar que:

La mayoría de estos sistemas se pueden diseñar como una estructura jerarquizada de subprogramas o módulos.

El formato de todos los módulos se puede adaptar a un mismo patrón formado por:

- Los datos de entrada (Input).
- El tipo de proceso que se realiza con los datos (Process).
- Los resultados de salida que proporciona (Output).

Dentro de los Diagramas HIPO se diferencian dos diagramas que son:

- El Diagrama HIPO de Contenidos: se utiliza para establecer la jerarquía de los módulos del sistema. Cada módulo tiene un nombre (A, B, C,…) y una referencia al correspondiente Diagrama HIPO de Detalle (0. 0, 1.0 , …). Ver figura 4.4 página 137.
- El Diagrama HIPO de Detalle: constan de 3 zonas :
    - Entrada (I).
    - Proceso (P).
    - Salida (O).

En las zonas de Entrada y Salida se indican respectivamente los datos que entran y salen del módulo.

En la zona central se detalla el pseudocódigo del proceso con referencia a otros diagramas de Detalle de nivel inferior en la jerarquía.

La lista de los diagramas referenciado se listan a continuación de la partícula PARA, por la parte superior.

A continuación de la partícula DE se indica el diagrama de detalle de nivel superior N (i,j).

4.6.1.3 Diagramas de Jackson

Con este Diagrama se diseñada sistemas software a partir de las estructuras de sus datos de entrada y salida.

El proceso se lleva cabo en tres pasos:

Especificación de las estructuras de datos de entrada y salida.

Obtención de una estructura del programa capaz de transformar las estructuras de datos de entrada en las de salida. Este paso implica una conversión de las estructuras de datos en las correspondientes estructuras de programa que las manejan, teniendo en cuenta las siguientes equivalencias:

TUPLA – SECUENCIA: colección de elementos de tipo diferentes combinados en un orden fijo.

UNION – SELECCIÓN: selección de un elemento entre varios posibles, de tipos diferentes.

FORMACION – ITERACION: colección de elementos del mismo tipo.

Expansión de la estructura del programa para lograr el diseño detallado del sistema. Para realizar este paso se suele utilizar Pseudocódigo.

La metodología Jackson está englobada dentro de las de Diseño Dirigido por los Datos, que ha sido utilizado fundamentalmente para diseñar sistemas relativamente pequeños de procesamiento de datos.

4.6.2 Notaciones estáticas

Sirven para describir características estáticas del sistema tales como la organización de la información, sin tener en cuenta su evolución durante el funcionamiento del sistema.

En el documento SRD de especificación se realiza una propuesta de organización a grandes rasgos y de acuerdo a las necesidades externas del sistema. En la fase de diseño se completa la organización de la información con los aspectos internos: datos auxiliares, mayor eficiencia en el manejo de datos, datos redundantes etc.

Resumiendo: como resultado del diseño se tendrá una organización de la información con un nivel de detalle mucho mayor. Las notaciones empleadas son las mismas que las empleadas en la especificación.

Diccionario de datos: con esta notación se detalla la estructura interna de los datos que maneja el sistema. Para el diseño se parte del diccionario de datos incluido en el SRD y mediante refinamientos se alcanza el nivel de detalle para empezar la codificación.

Diagramas Entidad - Relación: esta notación permite definir el modelo de datos, las relaciones entre los datos y en general la organización de la información. Para la fase de diseño se parte del diagrama propuesto en el SRD y se amplía con las nuevas entidades y sus relaciones que aparecen en la fase de diseño.

4.6.3 Notaciones dinámicas

Permiten describir el comportamiento del sistema durante su funcionamiento. La especificación de un sistema es una descripción del comportamiento requerido desde un punto de vista externo. Al diseñar la dinámica del sistema se detallara su comportamiento externo y se añadirá la descripción de un comportamiento interno capaz de garantizar que se cumplen todos los requisitos especificados en el documento SRD. Las notaciones que se emplean para el diseño son las mismas utilizadas en la especificación.

Diagrama de flujo de datos: desde el punto de vista del Diseño serán más exhaustivos que los de la especificación (SRD).

Diagrama de transición de estados: en el diseño del sistema pueden aparecer nuevos diagramas de estado que reflejen las transiciones entre estados internos. Es conveniente no ampliar o modificar los recogidos en el SRD.

Lenguaje de Descripción de Programas (PDL): Sirve tanto para la especificación funcional del sistema como para elaborar el diseño del mismo. Sin embargo, si se quiere descender al nivel de detalle que se requiere en la fase de diseño, la notación PDL se amplía con ciertas estructuras de algún lenguaje de alto nivel.

4.6.4 Notaciones híbridas

Estas notaciones tratan de cubrir simultáneamente aspectos estructurales, estáticos y dinámicos. Entre ellas podemos destacar las siguientes:

- La metodología de Jackson.
- Diseño orientado a los datos de Warnier.
- Diseño basado en abstracciones.
- Diseño orientado a objetos.

4.6.4.1 Diagramas de abstracciones

En una abstracción se distinguen tres partes:

- ***Nombre***: es el identificador de la abstracción
- ***Contenido***: es el elemento estático de la abstracción y en él se define la organización de los datos que constituyen la abstracción. En Modula son las definiciones de tipos, constantes y variables declaradas en el módulo de definición.
- ***Operaciones***: es el elemento dinámico de la abstracción y en él se agrupan todas las operaciones definidas para manejar el CONTENIDO de la abstracción. En Modula este elemento estaría formado por las definiciones de funciones y/o procedimientos declarados en el módulo de definición.

Podemos distinguir tres tipos de abstracciones:

Abstracción Funcional: un subprograma constituye una operación abstracta que denominaremos Abstracción Funcional. Esta forma de abstracción no tiene la parte de Contenido y solo está constituida por una única Operación.

Los tipos Abstractos de Datos: al Analizar y Diseñar un sistema se identifican datos de diferentes tipos con los que hay que operar. Se puede agrupar en una misma entidad la estructura del tipo de datos con las correspondientes operaciones necesarias para su manejo. A esta forma de abstracción la denominamos Tipo Abstracto de Datos y tiene una parte CONTENIDO y también sus correspondientes operaciones.

Los datos encapsulados: cuando solo se necesita una variable de un determinado tipo abstracto, su declaración se puede encapsular dentro de la misma abstracción. De esta forma, todas las operaciones de la abstracción se referirán siempre a esa variable sin necesidad de indicarlo de manera explícita. A esta forma de abstracción la denominamos Dato Encapsulado y tiene CONTENIDO y OPERACIONES, pero no permite declarar otras variables de su mismo tipo.

Ver Figura 4.9. Página 145.

4.6.4.2 Diagramas de objetos

La metodología de diseño basado en abstracciones y la metodología orientada a los objetos se han desarrollado de forma paralela pero en ámbitos muy distintos. Aunque las similitudes entre ambos conceptos son muy grandes, su desarrollo en distintos ámbitos ha propiciado la utilización de una terminología distinta para indicar lo mismo.

La estructura de un objeto es exactamente igual que la estructura de una abstracción. En cuanto a la terminología empleada se pueden establecer las siguientes equivalencias:

ABSTRACCIONES	OBJETOS
Tipo Abstracto de Datos	Clase de Objeto
Abstracción Funcional	NO HAY EQUIVALENCIA
Dato Encapsulado	NO HAY EQUIVALENCIA
Dato Abstracto (Variable o Constante)	Objeto (Ejemplar de Clase)
Contenido	Atributos
Operaciones	Métodos
Llamada a una Operación	Mensaje al Objeto
Equivalencia de Terminologías	

Además solo entre objetos se contempla una relación de herencia.

Si consideramos un objeto formado solo por sus atributos tendremos una estructura de datos.

Todas las entidades en un modelo de datos son objetos (o más exactamente, clases de objetos).

Podemos establecer dos tipos de relaciones especiales entre objetos:

Clasificación, especialización o herencia (no valida en abstracciones): en este caso los objetos hijos adaptan las operaciones heredadas a sus necesidades concretas. En la Figura 4.11 de la Página 148 se muestra un ejemplo de herencia entre objetos.

La notación empleada es la sugerida por la metodología OMT (Object Modelling Technique). Con el **Triangulo** se indica que los objetos inferiores (Hijo_Uno, Hijo_Dos e Hijo_Tres) heredan los atributos y las operaciones del objeto superior (Padre).

Con la relación de herencia no es necesario indicar la cardinalidad entre las clases de objetos, que está implícita en el diagrama, ya que en él aparece expresamente cada relación de herencia entre clases.

Composición (valida en abstracciones): en este caso se permite describir un objeto mediante los elementos que lo forman. En la Figura 4.12 de la Página 149 se muestra la relación de composición

con la notación OMT. El **Rombo** indica que el objeto Estructura está compuesto de Elemento_1, Elemento_2 y Elemento_3.

En la relación de Composición solo hay que indicar la cardinalidad en un sentido.

Cada objeto hijo solo puede formar parte de un objeto padre. Solo falta indicar qué número mínimo y máximo de objetos de cada clase hija se pueden utilizar para componer un objeto de la clase padre. Como se muestra en la Figura 4.12 la cardinalidad se indica junto a cada uno de los objetos componentes. Así, para formar Estructura son necesarios cero o varios Elemento_1, uno y solo un Elemento_2 y al menos un Elemento_3.

4.7 Documentos de diseño.

El resultado de la etapa de diseño se recoge en un documento que se utilizara como elemento de partida para las sucesivas etapas del proyecto y que denominaremos **Documento de Diseño de Software (SDD).**

Cuando la complejidad del sistema haga que el documento SDD resulte muy voluminoso y difícil de manejar, es habitual utilizar dos a más documentos para describir de forma jerarquizada la estructura global del sistema.

Las normas de la **ESA** establecen el empleo de un **Documento de Diseño Arquitectónico (ADD)** para describir el sistema en su conjunto y otro **Documento de Diseño Detallado (DDD)** para describir por separado cada uno de los componentes del sistema.

4.7.1 Documento de Diseño Arquitectónico (ADD)

Consta de las siguientes partes (Ver Cuadro 4.2 Pág. 151):

4.7.1.1 Introducción:

Se da una visión general de todo el documento.

Consta de los mismos apartados que el SRD (Objetivo, Ámbito, Definiciones, Siglas y Abreviaturas, y Referencias), pero referidos al sistema tal como se ha diseñado.

Siempre que se considere interesante se debe hacer referencia al SRD.

4.7.1.2 Panorámica Del Sistema:

Se da una visión general de los requisitos funcionales (y de otro tipo) del sistema que ha de ser diseñado, haciendo referencia al documento SRD.

4.7.1.3 Contexto del Sistema:

Definición de interfaz externa:

Se indica si el sistema posee conexiones con otros sistemas y si debe funcionar de una forma integrada con ellos.

En cada apartado se define la interfase que se debe utilizar con cada uno de os otros sistemas.

Si el sistema no necesita intercambiar información con ningún otro se indicará "No existe interfaz" o “No aplicable".

4.7.1.4 Diseño del Sistema:

Se considera el sistema en su conjunto y se hace una primera estructuración en componentes.

Metodología de diseño de alto nivel:

> Se describe brevemente la metodología a seguir en el proceso de diseño de la arquitectura del sistema.

Descomposición del sistema:

> Se describe el primer nivel de descomposición del sistema en sus componentes principales.
>
> Se enumeran los componentes y las relaciones estructurales entre ellos.

4.7.1.5 Diseño de los Componentes

Las siguientes subsecciones se repiten para cada uno de los componentes mencionados en el apartado **Descomposición del sistema.**

Identificador del Componente:

Nombre del Componente.

Dos Componentes no pueden tener nunca el mismo nombre.

El nombre se elegirá tratando que refleje su naturaleza.

Tipo

Se describe la clase de componente. En algunos casos basta con indicar el tipo de componente: subprograma, modulo, procedimiento, proceso, datos, etc.

Es posible definir nuevos tipos basados en otros más elementales.

Dentro del mismo documento se debe establecer una lista coherente de los tipos usados.

Objetivo

Se debe describir la necesidad de que exista el componente haciendo referencia a un requisito concreto que se trata de cubrir por ejemplo.

Si el requisito no forma parte del documento SRD se tendrá que detallar en este momento.

Función

Se describe qué hace el componente.

Esto se puede detallar mediante la transformación entrada/salida que realiza o si el componente es un dato se describirá qué información guarda.

Subordinados

Se enumeran todos los componentes usados por éste.

Dependencias

Se enumeran los componentes que usan a éste.

Junto a cada dependencia se podrá indicar su naturaleza: invocación de operación, datos compartidos, inicialización , creación , etc.

Interfaces

Se describe cómo otros componentes interactúan con éste.

Se tienen que establecer las distintas formas de interacción y las reglas para cada una de ellas: paso de parámetros, zona común de memoria, mensajes, etc.

También se indicaran las restricciones tales como rangos, errores, etc.

Recursos:

Se describen los elementos usados por este componente que son externos a este diseño: impresoras, particiones de disco, organización de la memoria, etc.

Referencias

Se presentan todas las referencias utilizadas.

Proceso

Se describen los algoritmos o reglas que utiliza el componente para realizar su función, como un refinamiento de la sección Función.

Datos

Se describen los datos internos del componente incluyendo el método de representación, valores iniciales, formato, valores validos, etc.

Esta descripción se puede realizar mediante un diccionario de datos.

Además se indicará el significado de cada elemento.

4.7.1.6 Viabilidad y Recursos Estimados.

Se analiza la viabilidad del sistema y se concretan los recursos que se necesitan para llevarlo a cabo.

4.7.1.7 Matriz Requisitos/Componentes.

Se muestra una matriz poniendo en las filas todos los requisitos y en las columnas todos los componentes.

Para cada requisito se marcará el componente o componentes encargados de que se cumpla.

4.7.2 Documento de Diseño Detallado

Este documento irá creciendo con el desarrollo del proyecto. En proyectos grandes puede ser conveniente organizarlo en varios volúmenes.

Como se puede ver en la página 155, cuadro 4.3, el formato de este documento es bastante similar al ADD. La diferencia entre ambos es el nivel de detalle al que se desciende.

En la Parte 1, Sección del documento se recogen todas las normas, convenios y procedimientos de trabajo que se deben aplicar durante el desarrollo del sistema.

De esta Sección depende que el trabajo realizado por un equipo amplio de personas tenga una estructura coherente y homogénea.

La realización de esta sección debe ser la primera actividad del diseño detallado antes de iniciar el diseño propiamente dicho.

Si se utilizan las mismas normas en diversos proyectos, se puede sustituir esta sección por referencias a los documentos que contienen la información correspondiente.

Capítulo

5

Técnicas Generales de Diseño Software

Centro Asociado Palma de Mallorca

Tutor: Antonio Rivero Cuesta

5 Técnicas Generales de Diseño Software

5.1 Introducción.

Diseñar software requiere cierta experiencia, el conocimiento de las técnicas de diseño es esencial para poder abordar la tarea con éxito.

5.2 Objetivos

Descomposición modular y decisión sobre algoritmos y estructuras de datos fundamentales.

5.3 Descomposición modular.

Para lograr una descomposición modular es necesario concretar los siguientes aspectos:

- Identificar los módulos.
- Describir cada módulo.
- Describir las relaciones entre módulos.

Podemos decir que con carácter general, un módulo es un fragmento de un sistema software que se puede elaborar con relativa independencia de los demás. La idea básica de esto, es que la elaboración de los diferentes módulos se pueda encargar a personas distintas y que todas ellas trabajen en paralelo.

Dependiendo de lo que se entienda por módulo tenemos la siguiente clasificación:

Código fuente: Es el considerado como tal con mayor frecuencia, contiene el texto fuente escrito.

Tabla de datos: Se utiliza para tabular ciertos datos de inicialización experimentales o de difícil obtención.

Configuración: Agrupamos en un mismo módulo toda aquella información que permite configurar el entorno concreto de trabajo.

Otros: Un módulo sirve para agrupar ciertos elementos del sistema relacionados entre sí y que se pueden tratar de forma separada del resto.

Solo en casos excepcionales se sacrificará el objetivo de tener un sistema mantenible para lograr una mayor velocidad de proceso o un menor tamaño de código.

Una descomposición modular debe poseer ciertas cualidades mínimas para que se pueda considerar suficientemente válida.

5.3.1 Independencia funcional

Para que un módulo posea independencia funcional debe realizar una función concreta o un conjunto de funciones afines, sin apenas ninguna relación con el resto de los módulos del sistema.

El mayor grado de independencia se consigue cuando no existe ninguna relación entre los distintos módulos. Evidentemente al descomponer un sistema en sus módulos es necesario que existan ciertas relaciones entre ellos. En todo caso se trata de reducir las relaciones entre módulos al mínimo.

A mayor independencia mayor mantenibilidad y reutilización del módulo.

Para medir la independencia funcional entre varios módulos tenemos dos criterios, Acoplamiento y Cohesión.

Acoplamiento: el grado de acoplamiento entre módulos es una medida de la interrelación que existe entre ellos. Tenemos varios tipos de acoplamiento que pueden ser fuerte., moderado, débil, y en ese orden son: (Ver esquema página 161).

Acoplamiento por Contenido (fuerte): se produce cuando desde un módulo se pueden cambiar los datos locales e incluso el código de otro módulo. En este caso no existe una separación real entre módulos y hay solapes entre ellos.

Este tipo de acoplamiento solo se puede lograr utilizando un lenguaje ensamblador o de muy bajo nivel y debe ser evitado siempre.

Ver figura 5.1.a Pag.161.

Acoplamiento Común (fuerte): en este caso se emplea una zona común de datos a la que tienen acceso varios o todos los módulos del sistema.

Cada módulo puede estructurar y manejar la zona común con total libertad sin tener en cuenta al resto de módulos.

El empleo de este acoplamiento exige que todos los módulos estén de acuerdo en la estructura de la zona común y cualquier cambio adoptado por uno de ellos afecta al resto que deberían ser modificados según la nueva estructura. Esta tipo de acoplamiento también debe ser evitado siempre.

Ver figura 5.1.b Pag.161.

Acoplamiento Externo (fuerte): aquí la zona común está constituida por algún dispositivo externo al que están ligados todos los módulos.

En este caso la estructura de la zona común la impone el formato de los datos que maneja el dispositivo y cualquier modificación exige el cambio de todos los módulos.

Acoplamiento de control (moderado): una señal o dato de control que se pasa desde un módulo A a otro B es lo que determina la línea de ejecución que se debe seguir dentro de B.

Ver figura 5.2. Pag.162.

Acoplamiento por etiqueta (débil): se produce cuando en el intercambio se suministra una referencia que facilita el acceso no solo a los datos estrictamente necesarios sino también a la estructura completa de la que forman parte, por ejemplo, vector, pila, etc.

Acoplamiento de datos (el más débil): Los módulos sólo se intercambian los datos que únicamente necesitan. **Este es el mejor tipo posible de acoplamiento.**

Ver figura 5.3.a Pag.163.

Estos dos tipos de acoplamiento débil son los más deseables en una descomposición modular. Con ellos, cualquier modificación en un módulo afecta muy poco o nada al resto.

Evidentemente el acoplamiento más débil es el que no existe. Este caso es el que se produce entre los módulos E y B de la figura 5.3 de la página 163, entre los que no existe ningún tipo de acoplamiento directo.

En resumen, el objetivo es conseguir el acoplamiento más débil posible.

Cohesión: hace referencia a que el contenido de cada módulo ha de tener la mayor coherencia posible.

Tenemos varios tipos que pueden ser baja, media, alta, y en ese orden son: (Ver esquema página 164).

Cohesión coincidental (la más baja): se produce cuando los elementos del módulo no guardan absolutamente ninguna relación entre ellos.

La existencia de este tipo de módulos indica que no ha sido realizada ninguna labor de diseño y que simplemente se ha efectuado un troceado del sistema.

Es la peor posible y debe evitarse siempre.

Cohesión lógica (baja): se produce cuando se agrupan en un módulo elementos que realizan funciones similares desde un punto de vista del usuario. Por ejemplo, un módulo de funciones matemáticas.

Cohesión temporal (baja): se agrupan en el mismo módulo elementos que se ejecutarán en el mismo momento. Esta es la situación que se produce en la fase de inicialización o finalización del sistema en que necesariamente se deben "arrancar" o "parar" dispositivos completamente heterogéneos, teclado, pantalla, ratón, etc.

Cohesión de comunicación (media): se produce cuando todos los elementos del módulo operan con el mismo conjunto de datos de entrada o producen el mismo conjunto de datos de salida.

Cohesión secuencial (media): se produce cuando todos los elementos del módulo trabajan de forma secuencial. Es decir, la salida de un elemento del módulo es la entrada del siguiente de una manera sucesiva.

***Cohesión funcional*:** cada elemento está encargado de la realización de una función concreta y especifica.

Cuando se utilicen técnicas de Diseño Funcional Descendente este nivel de cohesión será el máximo que se puede lograr dentro de un módulo.

Cohesión abstraccional: se logra cuando se diseña un módulo como tipo abstracto de datos con la técnica basada en abstracciones o como una clase de objetos con la técnica orientada a objetos. En ambos casos se asocia una organización de datos con las correspondientes operaciones encargadas de su manejo.

Con la técnica de diseño basado en abstracciones, un módulo con cohesión funcional sería una abstracción funcional.

La cohesión Baja debe evitarse siempre.

Con una cohesión Media se puede reducir el número de módulos. Sin embargo, esto no se debe realizar a costa de aumentar el grado de acoplamiento entre módulos. Por ejemplo, no se deben unir dos módulos con acoplamiento Débil para obtener uno único con acoplamiento Moderado, en el que se requiere pasar una señal para indicar con cuál de los dos antiguos submódulos se quiere trabajar.

Una cohesión Alta debe ser el objetivo que se debe perseguir en cualquier descomposición modular. Con ello se facilitara el mantenimiento y la reutilización de los módulos así diseñados.

Para conocer y mejorar la cohesión de un módulo se suele realizar una descripción de su comportamiento a partir de la cual, se puede establecer el grado de cohesión. Se tienen en cuenta los siguientes criterios:

Si la descripción es una frase compuesta que contiene comas o más de un verbo es muy probable que se esté incluyendo más de una función y la cohesión será Media de tipo Secuencial o de Comunicación.

Si la descripción contiene palabras relacionadas con el tiempo tales como "primero", "después", "entonces", la cohesión será de tipo temporal o secuencial.

Si la descripción no se refiere a algo especifico a continuación del verbo, es muy probable que tengamos una cohesión lógica. Por ejemplo "escribir todos los mensajes de error" o "calcular todas las funciones trigonométricas".

Si se utilizan palabras como “inicializar”, “preparar”, "configurar", etc., la cohesión será probablemente de tipo temporal.

En resumen, la descomposición modular con una mayor independencia funcional se logra con un acoplamiento DÉBIL entre sus módulos y una cohesión ALTA dentro de cada uno de ellos.

5.3.2 Comprensibilidad

La dinámica del proceso de diseño e implementación de un sistema hace que los cambios sean frecuentes. A menudo estos cambios deben ser realizados por personas que no participaron ni en el diseño ni en la implementación. Para facilitar estos cambios es necesario que cada módulo sea comprensible de forma aislada.

El primer factor que facilita la comprensión de un módulo es su independencia funcional. Sin embargo, es necesario además cuidar los siguientes factores:

***Identificación*:** una elección adecuada del nombre del módulo y de los nombres de cada uno de sus elementos facilita mucho su comprensión. Los nombres deben reflejar de manera sencilla el objetivo de la entidad que representan.

***Documentación*:** deben ser aclarados todos los aspectos de diseño o implementación con la documentación apropiada.

***Simplicidad*:** los algoritmos sencillos son los mejores. El diseñador debe simplificar al máximo las soluciones propuestas.

5.3.3 Adaptabilidad

Al diseñar un sistema se pretende resolver un problema concreto.

Por tanto, la descomposición modular estará muy unida al objetivo concreto del diseño. Esto dificulta la adaptabilidad del diseño a otras necesidades y la posible reutilización de algunos de sus módulos.

Las adaptaciones suelen ser múltiples y variadas a lo largo de la vida del sistema. Así, es necesario cuidar otros factores adicionales para facilitar la adaptabilidad y de ellos destacaremos los siguientes:

***Previsión*:** los módulos que se prevén que puedan cambiarse se deben agrupar en módulos con un acoplamiento lo más débil posible con el resto de los módulos. Así, las adaptaciones se podrán realizar con correcciones que solo afectaran a los módulos previstos.

***Accesibilidad*:** para poder adaptar un sistema debe ser sencillo acceder a todos los documentos de especificación, diseño e implementación. Esto requiere una organización minuciosa que se hará normalmente con herramientas CASE.

***Consistencia*:** cuando se modifican los programas fuente se deben modificar todos los documentos implicados. Esto se puede hacer automáticamente con herramientas para el control de versiones y configuración. En los entornos orientados a objetos no existen estas herramientas por lo que se debe imponer una disciplina férrea dentro de la biblioteca.

5.4 Técnicas de diseño funcional descendente.

La descomposición del sistema se hace desde un punto de vista funcional. Se van descomponiendo la función del sistema en funciones más sencillas, las cuales se encomiendan a módulos separados.

5.4.1 Desarrollo por refinamiento progresivo

Es la programación estructurada, en la que se emplean estructuras de control claras y sencillas como secuencia, selección e iteración. Se plantea el programa como una operación global única para ir descomponiéndola en otras operaciones más sencillas.

La aplicación de esta técnica a la fase de diseño consiste en realizar solo los primeros niveles de refinamiento, asignando a módulos separados las operaciones parciales que se van identificando.

5.4.2 Programación estructurada de Jackson (JSP)

Es similar a la programación estructurada, la diferencia está en las recomendaciones para ir construyendo la estructura del programa, que debe hacerse similar en lo posible a las estructuras de los datos de entrada y salida.

Los pasos de esta técnica son:

- **Analizar** el entorno del problema y describir las estructuras de datos a procesar.
- **Construir** la estructura del programa basada en las estructuras de datos.
- **Definir** las tareas a realizar en términos de las operaciones elementales disponibles y situarlas en los módulos apropiados de la estructura del programa.

Como técnica de diseño los pasos significativos son los dos primeros, mientras que el tercero corresponde a la fase de codificación.

Ver ejemplo páginas 171, 172 y 173.

Diseño estructurado

La tarea de diseño consiste en pasar de los DFD a los diagramas de estructura. La dificultad reside en que hay que establecer una jerarquía entre los diferentes módulos que no se ve en los DFD.

Ver ejemplo páginas 174.

A veces se usa un módulo de coordinación para construir esta jerarquía.

Para establecer una jerarquía de control razonable entre las diferentes operaciones descritas en los diagramas de flujo de datos, la técnica de diseño estructurado recomienda realizar los análisis denominados de Flujo de Transformación y de Flujo de Transacción.

Análisis de flujo de transformación: se identifica el flujo global de información desde los elementos de entrada al sistema, hasta los de salida.

Los procesos se deslindan en tres regiones, flujo de entrada, de transformación y de salida. Para obtener la estructura modular del programa se asignan módulos para las operaciones del diagrama y se añaden módulos de coordinación que realizan el control de acuerdo con la distribución del flujo de transformación.

Ver figura 5.8 página 175.

Análisis del flujo de transacción: se puede aplicar cuando el flujo de datos se puede descomponer en varias líneas separadas, cada una de las cuales corresponde a una función global o transacción distinta, de manera que solo una de estas líneas se activa para cada entrada de datos de tipo diferente. El análisis consiste en identificar el llamada Centro de Transacción del que salen las líneas de flujo y las regiones correspondientes a cada una de esas líneas o transacciones.

Ver figura 5.9 página 176. Ver ejemplo página 178.

5.5 Técnicas de diseño basado en abstracciones.

La idea general es que los módulos se correspondan o bien con funciones o bien con tipos abstractos de datos.

5.5.1 Descomposición modular basada en abstracciones

Consiste en ampliar el lenguaje existente con nuevas operaciones y tipos de datos definidos por el usuario. Se dedican módulos separados para cada tipo abstracto de datos y cada función importante. Se puede aplicar:

De forma descendente: es como el refinamiento progresivo. En cada paso la operación a refinar se define separadamente como abstracción funcional o como tipo abstracto de datos.

De forma ascendente: se van ampliando primitivas existentes en el lenguaje de programación y librerías con nuevas operaciones y tipos de mayor nivel, más adecuados para el campo de aplicación que se está diseñando.

Ver ejemplo página 179, 180.

5.5.2 Método de Abott

Con este método se sugiere una forma metódica de conseguir aquellos elementos del modelo del sistema que son buenos candidatos para ser considerados como abstracciones, a partir de las descripciones del sistema hechas en lenguaje natural.

Se procede así:

Identificar en el texto de la descripción los tipos de datos como sustantivos, los atributos como sustantivos y a veces como adjetivos, las operaciones como verbos o como nombres de acciones.

Hacer dos listas: una con nombres y otra con verbos.

Reorganizar las listas extrayendo los posibles datos y asociándoles sus atributos y operaciones; además eliminar los sinónimos y añadir los elementos implícitos en la descripción.

Para **obtener el diseño** asignamos un módulo a cada abstracción de datos o grupo de abstracciones relacionadas entre sí.

El módulo puede corresponder a un dato encapsulado si solo se maneja un dato de ese tipo en todo el programa.

Este método se puede usar tanto en diseño basado en abstracciones como en diseño orientado a objetos.

Ver ejemplo páginas 180 - 185.

5.6 Técnicas de diseño orientadas a objetos.

El diseño orientado a objetos es similar al diseño basado en abstracciones, solo que añadiendo la herencia y el polimorfismo.

Cada módulo contendrá la descripción de una clase de objetos o de varias relacionadas entre si. Además del diagrama modular, nos apoyamos en diagramas ampliados del modelo de datos, como los diagramas de estructura.

5.6.1 Diseño orientado a objetos

Se basa en los siguientes pasos:

Estudiar y comprender el problema**:** esto debe haberse realizado en la fase de Análisis de requisitos. Sin embargo, puede ser necesario repetirlo en parte porque la especificación no sea suficientemente precisa, o porque el diseño va a ser realizado por personas diferentes de las que confeccionan la especificación.

Desarrollar una posible solución**:** es posible que se haya realizado en la ***fase de análisis***, aunque es más probable que se tenga que hacer o completar en la ***fase de diseño***. Se consideran varias alternativas y se elegirá la que se considere más apropiada.

Identificar las Clases y Objetos**:** puede hacerse siguiendo la técnica de Abbott. Se identifican las clases de objetos y sus atributos. Si un objeto contiene otros objetos, no se suelen tratar como atributos, sino que se establece una relación de composición entre el objeto compuesto y los objetos componentes. Tras este primer paso puede ya confeccionarse un diagrama inicial del modelo de objetos.

Identificar las operaciones sobre los objetos**:** también puede hacerse siguiendo la técnica de Abbott. Además de identificar las operaciones hay que decidir a qué objeto o clase se asocia. Esto puede ser un problema de Diseño no trivial. Por ejemplo: la operación de escribir una fecha en pantalla con caracteres puede asociarse al objeto Fecha, o bien al objeto Pantalla. Cada decisión tiene sus ventajas e inconvenientes.

En algunos casos puede fraccionarse la operación en varias más sencillas. Por ejemplo, se puede definir una operación de conversión de fecha a texto, sobre el objeto Fecha, y otra operación de escritura del texto sobre el objeto Pantalla. Las operaciones pueden reflejarse sobre el diagrama de modelo de objetos.

Aplicar Herencia**:** una vez identificados los objetos y sus operaciones asociadas, hay que detectar analogías entre ellos, si las hay, y establecer las relaciones de herencia apropiadas. Estas relaciones de herencia se incluirán en el diagrama de modelo de objetos que se va desarrollando.

Describir las Operaciones**:** esta es una manera de verificar que el diseño es consistente. Cada operación se describe en pseudocódigo, haciendo referencia a operaciones o clases de datos definidos en este mismo diseño, o bien predefinidos en el lenguaje de programación a usar. En caso necesario habrá que añadir nuevas operaciones a las ya identificadas, o incluso nuevas clases de objetos, y se actualizara el modelo de objetos.

***Establecer la estructura modular*:** hay que asignar clases, objetos y operaciones a módulos separados. En principio se intentara que cada módulo corresponda a una clase de objetos (o a un objeto en particular). Si el módulo es demasiado complicado, ciertas operaciones pueden establecerse como módulos separados . También es posible agrupar en un solo módulo varios objetos o clases muy relacionados entre sí, para mejorar las características de acoplamiento y cohesión.

Como resultado de esta etapa se obtendrá el diagrama de estructura del sistema.

Una vez llegado a este punto hay que analizar si el diseño modular resultante es apropiado para pasar a la fase de codificación. Si algún módulo es todavía demasiado complejo, o no está definido con suficiente precisión, se repetirán los pasos anteriores de diseño para ese módulo, con objeto de refinarlo.

Ver ejemplo pagina 188 - 195.

5.7 Técnicas de diseño de datos.

La mayoría de las aplicaciones informáticas requieren almacenar información de forma permanente. La forma típica de hacerlo es apoyando la aplicación en una base de datos.

En la organización de la base de datos se pueden ver ***tres niveles***:

Nivel Externo: corresponde a la visión del usuario. La organización de los datos se realiza siguiendo esquemas en el campo de la aplicación. Por ejemplo, en forma de ficha de cliente.

Nivel Conceptual: establece una organización lógica de los datos, a través de un diagrama de modelo de datos, bien E-R, bien diagrama de Modelo de objetos.

Nivel Interno: organiza los datos según los esquemas del gestor de base de datos;

Si es una base de datos relacional serian tablas.

El paso del nivel Externo al Conceptual se puede realizar durante la etapa de análisis de requisitos.

El paso del nivel Conceptual al nivel interno se puede realizar durante la fase de Diseño.

Ver esquema página 196.

5.8 Diseño de bases de datos relacionales.

Es posible dar reglas prácticas para obtener los esquemas de las tablas de una base de datos relacional que reflejen la visión lógica de los datos, y que sean eficientes.

En el modelo relacional la eficiencia se ve desde dos puntos de vista, que son ***Formas Normales*** para evitar las redundancias, **Índices** para mejorar la velocidad de acceso a los datos.

5.8.1 Formas normales

Las Formas Normales de Codd definen criterios para establecer esquemas de tablas que sean claros y no redundantes. Estos criterios se numeran correlativamente, de menor a mayor nivel de restricción, dando lugar a las formas normales 1ª, 2ª, 3ª, etc. Una tabla que cumpla con una cierta forma normal cumple también con las anteriores.

***1ª Forma Normal*:** se dice que una tabla se encuentra en 1ª Forma Normal si la información asociada a cada una de las columnas es un valor único y no una colección de valores de número variable.

***2ª Forma Normal*:** se dice que una tabla se encuentra en 2ª Forma Normal si esta en 1ª Forma Normal y además hay una Clave Primaria que distingue cada fila; además cada casilla que no sea de la clave primaria depende de toda la clave primaria.

***3ª Forma Normal*:** se dice que una tabla se encuentra en 3ª Forma Normal si esta en 2ª forma normal y además el valor de cada columna que no es Clave Primaria depende directamente de la Clave Primaria, esto es, no hay dependencias entre columnas que no son Clave Primaria.

Ver ejemplo páginas 196, 197, 198.

5.8.2 Diseño de las entidades

Cada entidad del modelo E-R se traduce en una tabla por cada clase de entidad. Cada elemento de esa clase es una fila y Cada atributo de esa entidad es una columna

Si una entidad está relacionada con otras, y se quiere tener una referencia rápida entre las entidades relacionadas, se puede incluir una columna con un número de referencia que identifique cada fila de la tabla.

El número o código de referencia si se usa sirve como Clave Primaria.

5.8.3 Tratamiento de las relaciones de asociación

En el modelo de objetos se tienen dos tipos especiales de relación, las de composición o agregación y las de herencia o especialización. Las demás son las de asociación. En el modelo E-R todas las relaciones se consideran relaciones de asociación.

La manera de almacenar en tablas la información de las relaciones de asociación depende de la cardinalidad de la relación.

La técnica general válida para todas las cardinalidades es traducir la relación a una tabla conteniendo referencias a las tablas de las entidades relacionadas, así como los atributos de la relación si los hay.

La referencia a las entidades relacionadas se hará mediante la clave primaria de cada una.

Ver figura 5.20 (a), página 200.

Si la cardinalidad es 1-N: se incluyen los datos de la relaciones en la misma tabla de una de las entidades relacionadas.

Ver figura 520 (b), página 200.

Si la cardinalidad es 1-1: se pueden fundir las tablas de las dos entidades en una sola. Ver figura 5.20 (c), página 200.

5.8.4 Tratamiento de las relaciones de composición

La cardinalidad del lado del objeto compuesto es casi siempre 1. Se aplican los mismos criterios anteriores.

5.8.5 Tratamiento de la herencia

Cuando una clase tiene varias subclases hay tres formas de almacenar en tablas la información de las entidades.

1º: se usa una tabla para la superclase, con los atributos comunes heredados por las subclases, mas una tabla por cada subclase con sus atributos específicos.

2ª: se repiten los atributos comunes en las tablas de cada subclase, por lo que desaparece la tabla de la superclase.

3º: se amplía la tabla de la superclase con todos los atributos de cada una de las subclases, prescindiendo de las tablas de cada subclase.

Ver figura 5.21 página 202.

5.8.6 Diseño de índices

Permiten acceder rápidamente a un dato concreto, a costa de aumentar el espacio de almacenamiento y el tiempo de almacenamiento de nuevos datos y la modificación del valor de un atributo indexado.

Si hay que acceder a datos a través de sus relaciones con otros, es conveniente mantener índices sobre las Claves Primarias y columnas de referencia de las entidades relacionadas.

Ver ejemplo 201 - 204.

5.9 Diseño de bases de datos de objetos.

Hay una mayor variedad de estructuras disponibles pero distintas en de cada caso. Podemos ver dos enfoques en el diseño:

1º: cuando la base de datos de objetos permite usar una gran variedad de estructuras. En este caso, el sistema de gestión de base de datos aporta como complemento la persistencia de los datos.

2º: cuando no existe esa variedad de estructuras y la base de datos de objetos es análoga a una base de datos relacional. En este caso, el sistema de gestión de base de datos aporta la existencia implícita de identificadores de objetos, que hacen innecesarias las columnas explícitas de códigos o números de referencia.

Ver ejemplos páginas 210 a 232.

Capítulo

6

UML

Centro Asociado Palma de Mallorca

Tutor: Antonio Rivero Cuesta

6 UML - Lenguaje Unificado de Modelado

6.1 Introducción.

El UML es un lenguaje universal de modelado de sistemas que se emplea para especificar y modelizar los sistemas software.

6.2 Objetivos

Familiarizarse con el lenguaje UML, su utilización para la especificación y modelado.

6.3 ¿Qué es UML?

Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema.

Ofrece un estándar para describir un plano del sistema, incluyendo aspectos conceptuales tales como procesos de negocio, funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y compuestos reciclados.

Presenta información sobre la estructura estática y el comportamiento dinámico de un sistema.

Existen herramientas que pueden ofrecer generadores de código a partir de los diagramas UML.

6.4 Orígenes de UML

La notación UML deriva y unifica las tres metodologías de análisis y diseño más extendidas:

- Metodología de Grady Booch.
- Técnica de modelado orientado a objetos.
- Metodología de casos de uso.

6.5 Objetivos del UML

Tendremos en cuenta los siguientes objetivos:

- Conseguir la capacidad de modelar sistemas, desde el concepto hasta los artefactos ejecutables, utilizando técnicas orientadas a objetos.
- Cubrir las cuestiones relacionadas con el tamaño inherente a los sistemas complejos y críticos.
- Crear un lenguaje de modelado utilizable tanto por las personas como por las máquinas.

Mediante el fomento del uso del UML, OMG pretende alcanzar los siguientes objetivos:

- Proporcionar a los usuarios un lenguaje de modelado visual expresivo y utilizable para el desarrollo e intercambio de modelos significativo.
- Proporcionar mecanismos de extensión y especialización.
- Ser independiente del proceso de desarrollo y de los lenguajes de programación.
- Proporcionar una base formal para entender el lenguaje de modelado.
- Fomentar el crecimiento del mercado de las herramientas OO.
- Soportar conceptos de desarrollo de alto nivel como pueden ser colaboraciones, frameworks, patterns, y componentes.
- Integrar las mejores prácticas utilizadas hasta el momento.

El UML debemos entenderlo como un estándar para modelado y no como un estándar de proceso software.

Es un leguaje que podemos utilizar en otros campos:

- Sistemas de información de la empresa.
- Bancos y servicios financieros.
- Telecomunicaciones.
- Transporte.
- Defensa/industria aeroespacial.
- Comercio.
- Electrónica médica.
- Ámbito científico.
- Servicios distribuidos en la Web.

6.6 Estructura de UML

Los conceptos y modelos de UML pueden agruparse en las siguientes áreas conceptuales:

1. Estructura estática:

   Cualquier modelo preciso debe primero definir su universo:

   - Los conceptos clave de la aplicación.
   - Sus propiedades internas.
   - Las relaciones entre cada una de ellas.

   Los conceptos de la aplicación son modelados como clases. La información almacenada es modelada como atributos. La estructura estática se expresa con diagramas de clases y puede usarse para generar la mayoría de las declaraciones de estructuras de datos en un programa.

2. Comportamiento dinámico:

   La visión de la interacción de los objetos se representa con los enlaces entre objetos junto con el flujo de mensajes y los enlaces entre ellos.

   Este punto de vista unifica:

   - La estructura de los datos.
   - El control de flujo.
   - El flujo de datos.

3. Construcciones de implementación:

   Los modelos UML tienen significado para el análisis lógico y para la implementación física.

   Un componente es una parte física reemplazable de un sistema y es capaz de responder a las peticiones descritas por un conjunto de interfaces.

   Un nodo es un recurso computacional que define una localización durante la ejecución de un sistema.

   Puede contener componentes y objetos.

4. Mecanismos de extensión:

   UML tiene una limitada capacidad de extensión.

5. Organización del modelo:

La información del modelo debe ser dividida en piezas coherentes, para que los equipos puedan trabajar en las diferentes partes de forma concurrente.

Debemos organizar el contenido en paquetes de tamaño modesto.

Los paquetes son unidades organizativas, jerárquicas y de propósito general de los modelos UML.

Pueden usarse para el almacenamiento, control de acceso, gestión de la configuración y construcción de bibliotecas que contengan fragmentos de código reutilizable.

6. Elementos de anotación:

Son las partes explicativas de los modelos UML.

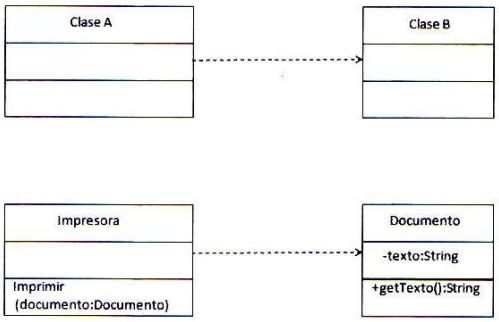
Son comentarios que se pueden aplicar para describir, clasificar y hacer observaciones sobre cualquier elemento de un modelo.

El tipo principal de anotación es la nota que simplemente es un símbolo para mostrar restricciones y comentarios junto a un elemento o un conjunto de elementos.

7. Relaciones:

Existen seis tipos de relaciones entre los elementos de un modelo UML.

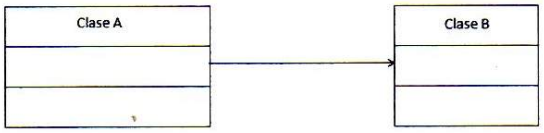
- **Dependencia**. Es una relación semántica entre dos elementos en la cual un cambio a un elemento, el elemento independiente, puede afectar a la semántica del otro elemento, el elemento dependiente. Se representa como una línea discontinua, que puede ser dirigida y a veces tiene uta etiqueta. LA clase A usa la clase B.

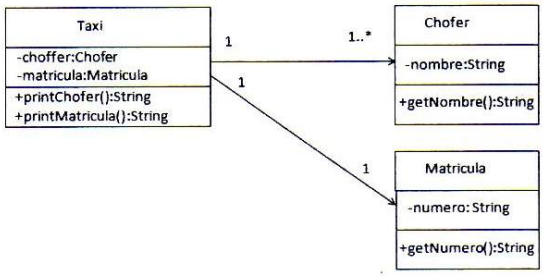


- **Asociación**. Es una relación estructural que describe un conjunto de enlaces, los cuales son conexiones entre objetos. Se representa como una línea continua, que puede ser dirigida y a veces tiene uta etiqueta. A menudo se incluyen otros adornos para indicar la multiplicidad y los roles de los objetos involucrados. La multiplicidad de una asociación determina cuantos objetos de cada tipo intervienen en la relación. Cada asociación tiene dos multiplicidades, una para cada extremo de la relación. Para especificar la multiplicidad de una asociación hay que especificar la multiplicidad mínima y la multiplicidad máxima.

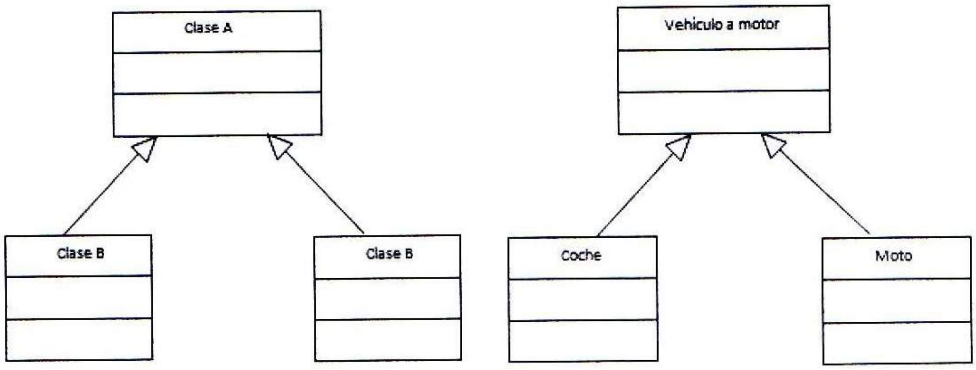
Multiplicidad	**Significado**
1	Uno y sólo uno
0..1	Cero o uno
N..M	Desde N hasta M
*	Cero o varios
0..*	Cero o varios
1..*	Uno o varios, al menos uno

La clase A está asociada a la clase B. La clase A necesita a la clase B.

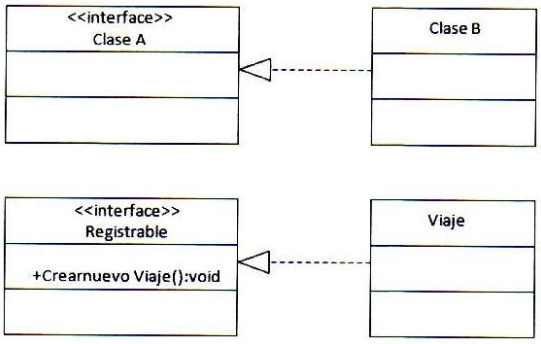




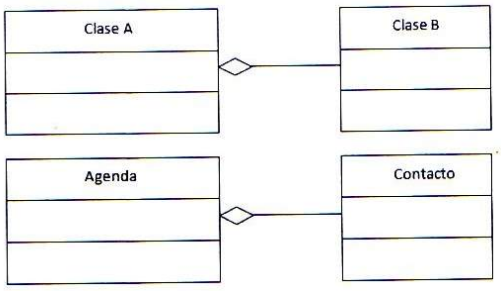
- **Generalización**. Es una relación de especialización / generalización en la cual los objetos del elemento especializado, ***el hijo***, pueden sustituir a los objetos del elemento general, ***el padre***. El hijo comparte la estructura y el comportamiento del padre. Se representa con una línea acabada en punta de flecha vacía. La clase A es una generalización de la clase B o la C. También podemos decir que la clase B y C son especializaciones de la clase A.



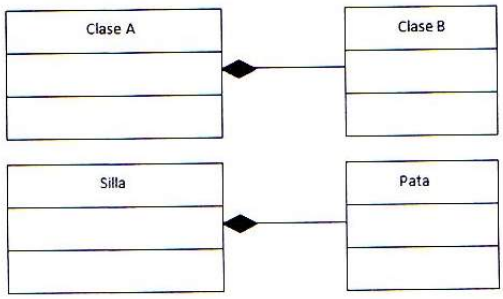
- **Realización**. Es una relación semántica entre clasificadores, donde un clasificador especifica un contrato que otro clasificador garantiza que cumplirá. Se pueden encontrar relaciones de realización en dos sitios: entre interfaces y las clases y componentes que las realizan, y entre los casos de uso y las colaboraciones que los realizan. Se representa como una mezcla entre la generalización y la dependencia, una línea discontinua con una punta de flecha vacía. En este ejemplo la clase B es una realización de la A. Viaje realiza la interface Registrable.



- **Agregación**. Es similar a la relación de agregación, solo varía en la multiplicidad ya que en lugar de ser una relación "uno a uno" es de "uno a muchos". Se representa con una flecha acabada en rombo de color blanco. La clase A agrupa varios elementos del tipo Clase B.



- **Composición**. Es similar a la agregación, pero la relación es más fuerte. Aporta documentación conceptual ya que es una "relación de vida", el tiempo de vida de un objeto está condicionado por el tiempo de vida del objeto que lo incluye. La Clase A agrupa varios elementos del tipo Clase B. el tiempo de vida de los objetos de tipo Clase B está condicionado por el tiempo de vida del objeto de tipo Clase A.



8. Diagramas.

   Se utilizan para representar diferentes perspectivas de un sistema de forma que un diagrama es una proyección del mismo UML.

   UML proporciona un amplio conjunto de diagramas que normalmente se usan en pequeños subconjuntos para poder representar las cinco vistas principales de la arquitectura de un sistema.

6.7 Diagramas UML

6.7.1 Diagramas de casos de uso

Ilustran la funcionalidad proporcionada por una unidad del sistema.

Describen las relaciones y las dependencias entre un grupo de casos de uso y los actores participantes en el proceso.

No están pensados para el diseño y no pueden describir los elementos internos de un sistema.

Sirven para facilitar la comunicación con los futuros usuarios del sistema y con el cliente.

Resultan especialmente útiles para determinar las características necesarias que tendrá el sistema.

Describen qué es lo que debe hacer el sistema, pero no cómo.

Actor

Es una entidad externa, fuera del sistema, que interacciona con el sistema participando y normalmente iniciando en un caso de uso.

Los actores pueden ser gente real, otros sistemas o eventos.

No representan a personas físicas o a sistemas, sino un rol.

Cuando una persona interactúa con el sistema de diferentes maneras, estará representado por varios actores.

Caso de uso

Describe desde el punto de vista de los actores, un grupo de actividades de un sistema que produce un resultado concreto y tangible.

Son descriptores de las interacciones típicas entre los usuarios de un sistema y ese mismo sistema.

Representan el interfaz externo del sistema y especifican qué requisitos de funcionamiento debe tener éste, únicamente el qué, nunca el cómo.

Hay que tener en cuenta las siguientes reglas:

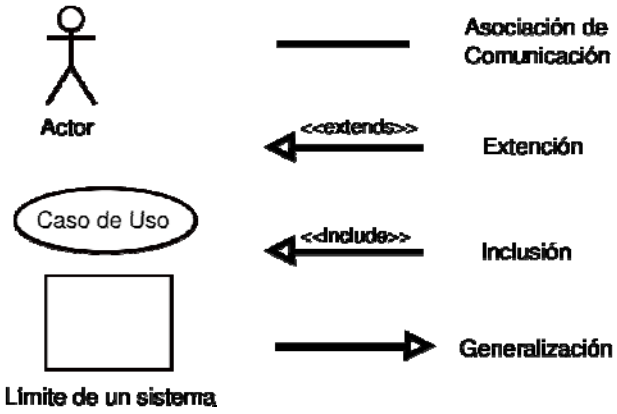
- Cada caso de uso está relacionado como mínimo con un actor.
- Cada caso de uso es un iniciador, un actor.
- Cada caso de uso lleva a un resultado relevante.

Figura 6.8 Actor y caso de uso

Los casos de uso pueden tener relaciones con otros casos de uso.

Los tres tipos de relaciones más comunes entre casos de uso son:

- "**include**" que especifica una situación en la que un caso de uso tiene lugar dentro de otro caso de uso.
- "**extends**" que especifica que en ciertas situaciones, o en algún punto un caso de uso será extendido por otro.
- Generalización que especifica que un caso de uso hereda las características del "super" caso de uso, y pude volver a especificar algunas o todas ellas de una forma muy similar a las herencias entre clases.



En la figura 6.9 se representa un diagrama de casos de uso de un cajero automático con dos actores: un cliente y un empleado de banco.

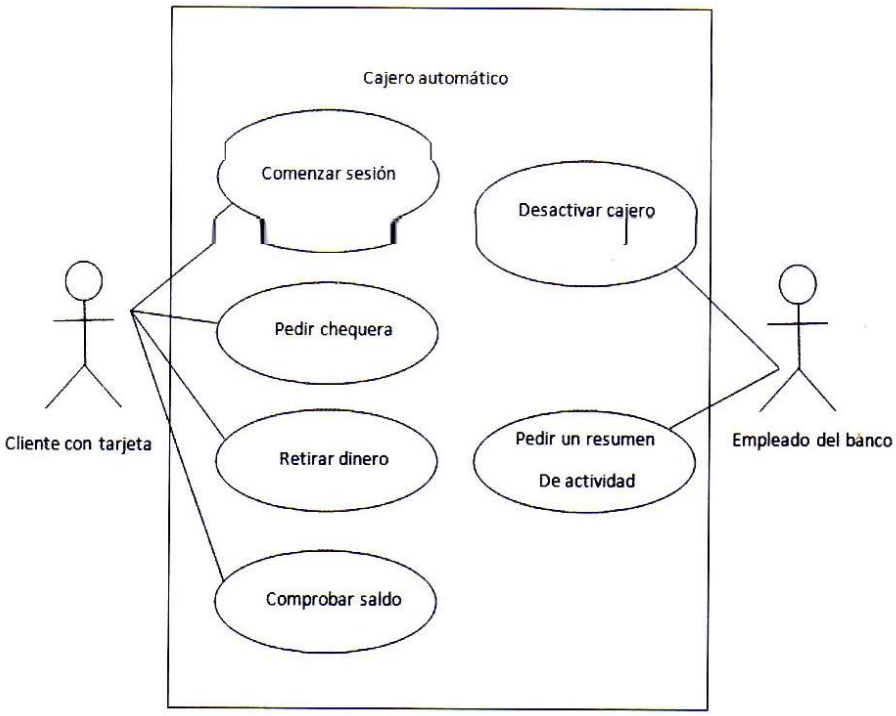
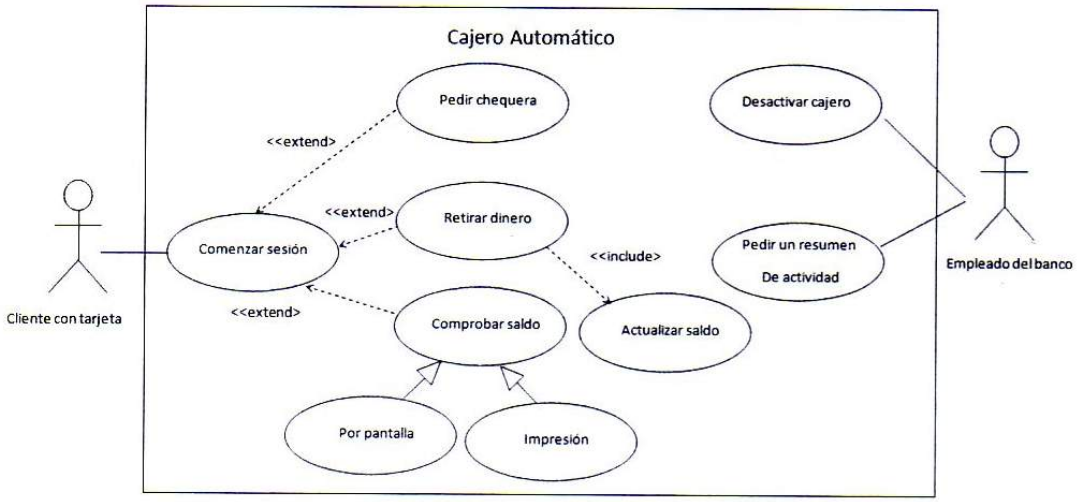


Figura 6.9 Diagrama de casos de usos

En la figura 6.10 se muestra el diagrama de casos de uso del cajero automático presentado en la figura 6.9 extendiendo los casos de uso con las diferentes relaciones.



Descripción de casos de uso

Son reseñas textuales del caso de uso.

Normalmente tienen el formato de una nota o un documento relacionado de alguna manera con el caso de uso, y explica los procesos o actividades que tienen lugar en el caso de uso.

6.7.2 Diagrama de clases

Los diagramas de clases muestran las diferentes clases que componen un sistema y cómo se relacionan unas con las otras.

Los diagramas de clases son diagramas estáticos porque muestran las clases, junto con sus métodos y atributos, así como las relaciones estáticas entre ellas: qué clases conocen a qué otras clases o qué clases son parte de otras clases.

No muestran los métodos mediante los que se invocan entre ellas.

Clase

Define los atributos y los métodos de una serie de objetos.

Todos los objetos de esta clase, instancias de clase, tienen el mismo comportamiento y el mismo conjunto de atributos.

Se representan con rectángulos, con el nombre de la clase y también se pueden mostrar los atributos y operaciones de la clase en otros dos compartimentos del rectángulo dentro de la clase.

La clase está formada por atributos y operaciones o métodos.

- Atributos

Se muestran al menos con su nombre y también pueden mostrar su tipo, valor inicial y otras propiedades.

Aparecen calificados en el diagrama dependiendo de su acceso como:

+ Indica atributos públicos.

Indica atributos protegidos.

− Indica atributos privados.

- Operaciones

Se muestran al menos con su nombre y pueden mostrar sus parámetros y valores de retorno.

Aparecen calificadas en el diagrama dependiendo de su acceso como:

+ Indica operaciones públicas.

Indica operaciones protegidas.

− Indica operaciones privadas.

- Plantillas

Las clases pueden tener plantillas o ***templates***, un valor usado para una clase no especificada o un tipo.

El tipo de plantilla se especifica cuando se inicia una clase, cuando se crea un objeto.

Asociaciones de Clases

Las clases se pueden relacionar con otras de las siguientes formas posibles:

- Generalización.
- Asociación.
- Realización.
- Agregación.
- Composición.

Los diagramas de clases pueden contener más componentes aparte de clases, pueden ser:

- **Interfaces**. Son clases abstractas, son instancias que no pueden ser creadas directamente a partir de ellas. Pueden contener operaciones, pero no atributos. Las clases pueden heredarse de las interfaces pudiendo así realizarse instancias a partir de estos diagramas.
- **Tipo de Datos**. Son primitivas incluidas en algunos lenguajes de programación. No pueden tener relación con clases, pero las clases sí pueden relacionarse con ellos.
- **Enumeraciones**. Son simples listas de valores. No pueden relacionarse con las clases, pero las clases sí pueden hacerlo con ellos.
- **Paquetes**. En lenguajes de programación, representan un espacio de nombres en un diagrama se emplean para representar partes del sistema que contienen más de una clase, incluso cientos de ellas.

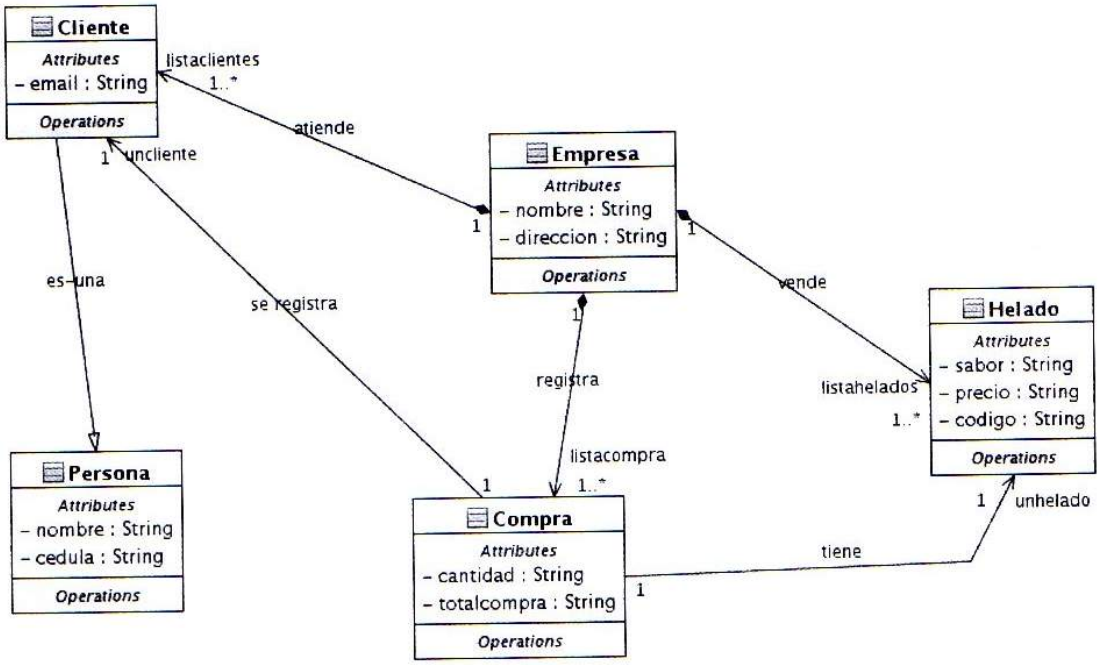


Figura 6.11 Diagrama de clases

6.7.3 Diagramas de secuencia

Muestran el flujo de mensajes entre objetos para un determinado caso de uso.

Ponen especial énfasis en el orden y el momento en que se envían los mensajes a los objetos.

Se explica la secuencia de las llamadas que se producen entre los objetos que intervienen.

Pueden tener mayor o menor detalle y representar diferentes llamadas a diferentes objetos.

Tienen dos dimensiones:

- La vertical muestra la secuencia de llamadas ordenadas en el tiempo en el que ocurren.
- La horizontal muestra las diferentes instancias de objetos a las que son enviadas las llamadas.

En la figura 6.12 se puede ver una secuencia de invocaciones para el dibujo del botón de una ventana de una aplicación

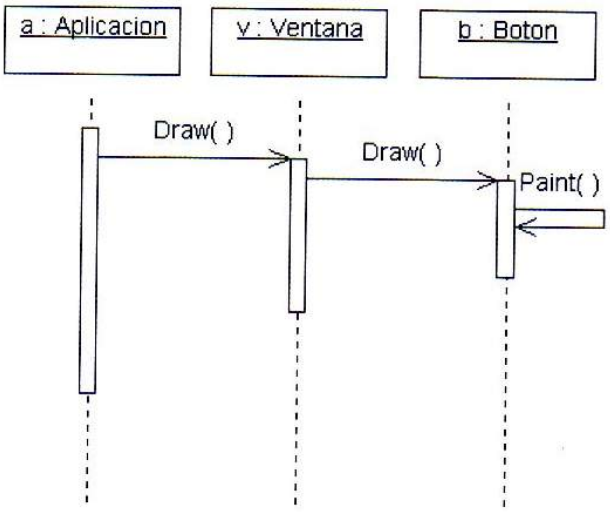


Figura 6.12 Diagrama de secuencia simple

Los objetos están representados por rectángulos con el nombre del objeto dentro y por líneas intermitentes verticales con el nombre del objeto en la parte más alta.

El eje de tiempo también es vertical, incrementándose hacia abajo, de forma que los mensajes son enviados de un objeto a otro en forma de flechas con los nombres de la operación y los parámetros.

Es conveniente dibujar una flecha de retorno de la llamada.

Los mensajes pueden ser:

- Síncronos, el tipo normal de llamada del mensaje donde se pasa el control a objeto llamado hasta que el método finalice.
- Asíncronos, donde se devuelve el control directamente al objeto que realiza la llamada.

Los mensajes síncronos tienen una caja vertical en un lateral del objeto invocante que muestra el flujo del control del programa.

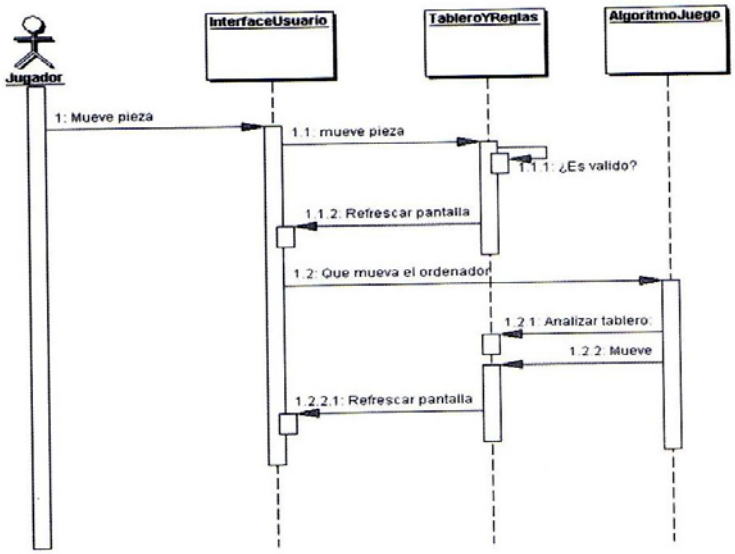


Figura 6.13 Diagrama de secuencia

6.7.4 Diagramas de colaboración

Muestran las interacciones que ocurren entre los objetos que participan en una situación determinada.

Se parece a los diagramas de secuencia, pero destacando la forma en que las operaciones se producen en el tiempo.

Los diagramas de colaboración fijan el interés en las relaciones entre objetos y su topología.

Los mensajes enviados de un objeto a otro se representan mediante flechas, mostrando el nombre del mensaje, los parámetros y la secuencia del mensaje.

Están indicados para mostrar una situación o flujo de programa específicos y son unos de los mejores tipos de diagramas para demostrar o explicar rápidamente un proceso dentro de la lógica del programa.

En las figuras 6.14 y 6.15 podemos ver los diagramas de secuencia y el respectivo de colaboración.

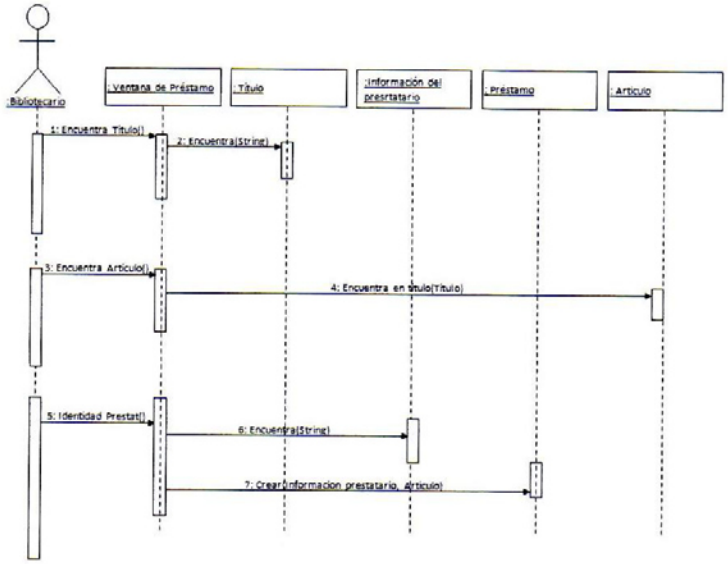


Figura 6.14 Diagrama de secuencia

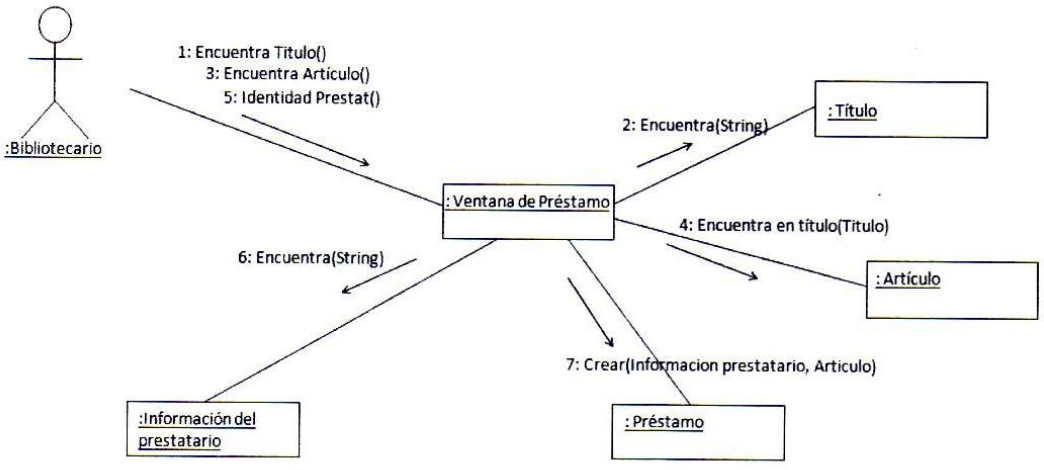


Figura 6.15 Diagrama de colaboración

6.7.5 Diagrama de estado

Muestran los diferentes estados de un objeto durante su vida, y los estímulos que provocan los cambios de estado en un objeto.

Ven a los objetos como máquinas de estado o autómatas finitos que pueden estar en un conjunto de estados finitos y que pueden cambiar su estado a través de un estímulo perteneciente a un conjunto finito.

Todos los objetos tienen un estado pero no todos los objetos son susceptibles de generar un diagrama de estados.

Solo aquellos que presenten "***estados interesantes***", es decir tres o más estados, deberán ser modelados en su diagrama de estados

La notación de un diagrama de estados tiene cinco elementos básicos:

- **Punto de inicio**: dibujado con un círculo relleno.
- **Transición entre estados**: dibujado con una línea terminada con punta de flecha abierta.
- **Estado**: dibujado con un rectángulo con los vértices redondeados.
- **Punto de decisión**: dibujado con un círculo no relleno.
- **Puntos de terminación**: dibujados con un círculo con otro relleno en su interior.

Para dibujar un diagrama de estado comenzaremos por el punto de inicio y la línea de transición que lleve hasta el primer estado del objeto.

Después se dibujan cada uno de los estados distribuidos por el diagrama y se conectan con las líneas de transición de estados.

Hay dos tipos especiales de estado: inicio y fin.

No hay ningún evento que pueda devolver a un objeto a su estado de inicio y de la misma forma no hay ningún evento que pueda sacar a un objeto de su estado de fin.

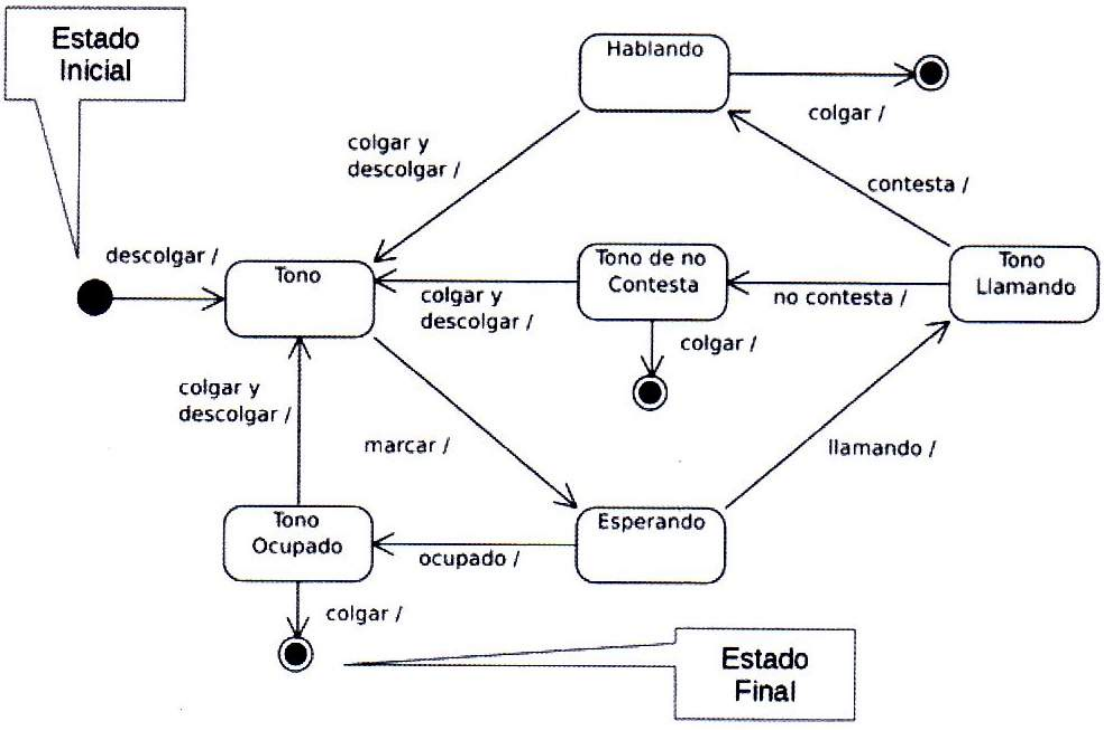
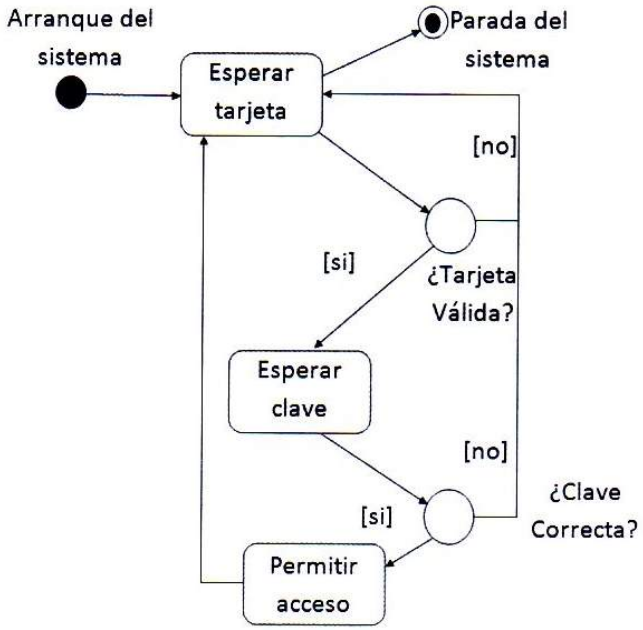


Figura 6.16 Diagrama de transición de estados

En la figura 6.17 podemos ver el diagrama de transición de estados del acceso con tarjeta con dos puntos de bifurcación



6.7.6 Diagrama de actividad

Describen la secuencia de las actividades en un sistema.

Son una forma especial de los diagramas de estado que únicamente contienen actividades.

Son similares a los diagramas de flujo procesales con la diferencia de que todas las actividades están claramente unidas a objetos.

Siempre están asociados a una clase.

Soportan actividades tanto secuenciales como paralelas.

La ejecución paralela se representa por medio de iconos de fork/espera, y en el caso de las actividades paralelas, no importa en qué orden sean invocadas, pueden ser ejecutadas simultáneamente una detrás de otra.

Actividad

Una actividad es un único paso de un proceso.

Es un estado del sistema que tiene actividad interna y al menos una transición saliente.

También pueden tener más de una transición saliente, si tienen diferentes condiciones.

Pueden formar jerarquías, es decir, formada de varias ***actividades de detalle*** en cuyo caso las transiciones entrantes y salientes deberían coincidir con las del diagrama de detalle.

Los diagramas de actividad son indicados para modelar procesos de alto nivel.

Los elementos para definir un diagrama de actividades son similares a los usados en los diagrama de secuencia.

- **Círculo** relleno de inicio.
- **Rectángulos** con los bordes redondeados para determinar las actividades.
- **Flechas** conectoras para unir las actividades.
- **Puntos** de decisión con círculos huecos.
- **Líneas** divisoras o calles para establecer las responsabilidades entre los distintos objetos del sistema.

En la figura 6.18 se ve un diagrama de actividades para la gestión de pedidos en una empresa.

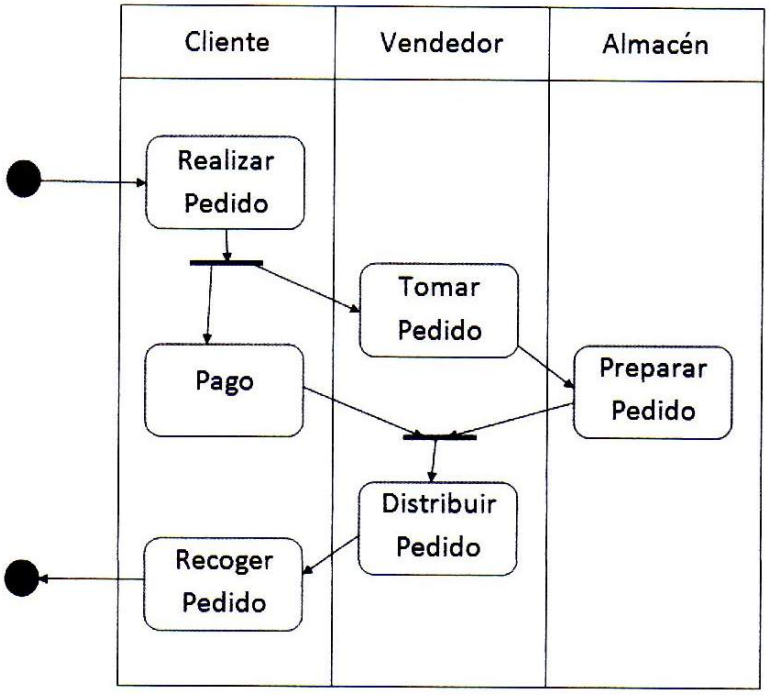


Figura 6.18 Diagrama de secuencia de actividad

6.7.7 Diagramas de componentes

Muestran los componentes del software y los artilugios de los que está compuesto, como:

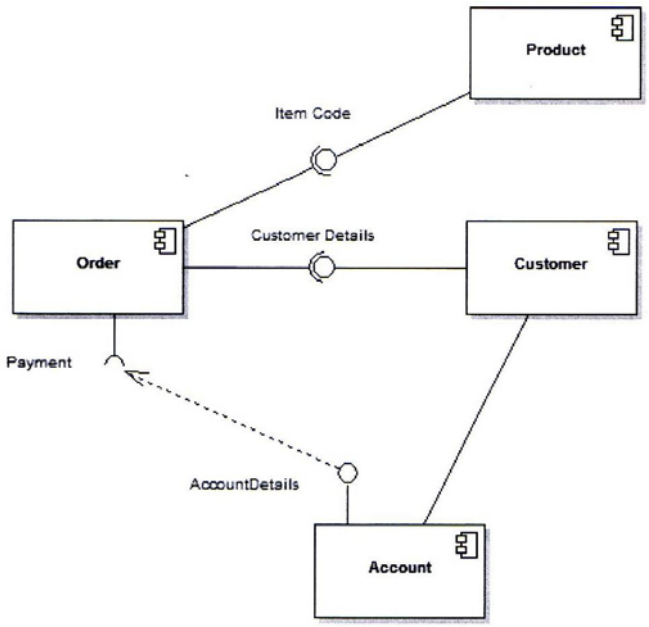
- Los archivos de código fuente.
- Las librerías.
- Las tablas en una base de datos.

Los componentes pueden tener interfaces que permiten asociaciones entre componentes.

Ilustran las piezas de software, controladores embebidos, etc. Que conformarán un sistema.

Un diagrama de componentes tiene un nivel más alto de abstracción que un diagrama de clase.

Normalmente un componente se implementa por una o más clases en tiempo de ejecución.



6.7.8 Diagrama de despliegue

Muestra cómo el sistema se asentará físicamente en el entorno hardware que lo acompaña.

El objetivo es mostrar dónde los componentes del sistema se ejecutarán y cómo se comunicarán entre ellos.

Será un diagrama muy usado por las personas que produzcan el sistema.

La notación es la misma que el diagrama de componentes.

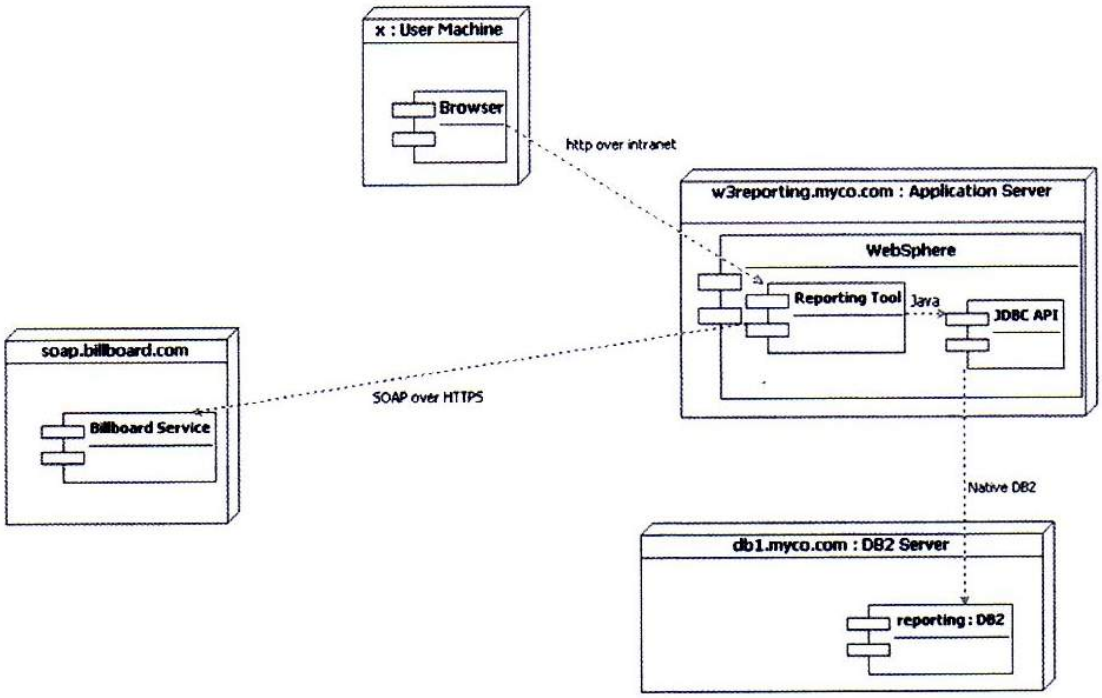


Figura 6.20 Diagrama de despliegue

Capítulo

7

Codificación del Software

Centro Asociado Palma de Mallorca

Tutor: Antonio Rivero Cuesta

7 Codificación del Software.

7.1 Introducción.

Las etapas de Análisis y Diseño tienen como misión fundamental la de organizar y traducir los requisitos del Cliente en módulos de programa que puedan ser codificados de forma independiente. En la fase de Codificación se elabora el producto fundamental de todo el desarrollo: los programas fuente.

Un elemento esencial dentro de la codificación es el lenguaje de programación que se emplea. Antes de iniciar la fase de codificación es fundamental establecer por escrito cual será la metodología de programación que se empleará por todos los miembros del equipo de trabajo.

En la fase de codificación pueden intervenir un gran número de programadores y durante un tiempo muy largo. Como parte de la metodología de programación y para garantizar la adecuada homogeneidad en la codificación se deben establecer las normas y estilo de codificación.

Un estilo homogéneo hace posible un trabajo coordinado entre los programadores ya que el entendimiento entre todos ellos resulta mucho más fácil. El empleo de estas normas facilita de forma considerable el mantenimiento posterior y la reusabilidad del software codificado.

Cuando los resultados de las pruebas no sean satisfactorios habrá que realizar cambios en la codificación. Ambos aspectos, codificación y pruebas, están muy relacionados.

La codificación se debe estructurar para facilitar su depuración y las modificaciones derivadas de las pruebas.

7.2 Objetivos

Estudiar la tarea de codificación.

Analizar los lenguajes de programación más utilizados y agruparlos por sus características comunes.

Estudiar los recursos de programación que nos ofrecen los lenguajes y las técnicas de implementación.

Repasar algunos criterios habitualmente utilizados para elegir un determinado lenguaje de programación.

7.3 Lenguajes de programación.

Los lenguajes de programación son el medio fundamental que tenemos para realizar la codificación.

Con los lenguajes actuales resulta más sencillo obtener un software de calidad, mantenible y reutilizable.

El conocimiento de las prestaciones de los lenguajes permite aprovechar mejor sus posibilidades y también salvar sus posibles deficiencias. No existe un único lenguaje ideal para todo y a veces es necesario trabajar con varios lenguajes para las distintas partes de un mismo desarrollo.

Desarrollo histórico. El estudio de la evolución histórica de los lenguajes es una buena manera de adquirir una visión panorámica de los mismos.

7.3.1 Lenguajes de primera generación

Son los ensambladores con un nivel de abstracción muy bajo. La programación en ensamblador resulta compleja, da lugar a errores difíciles de detectar, exige que el programador conozca bastante bien la arquitectura del computador y se necesita adaptar la solución a las particularidades de cada computador concreto.

Solo está justificada la utilización del ensamblador en aquellos casos en los que no se puede programar con ningún lenguaje de alto nivel.

7.3.2 Lenguajes de segunda generación

Su característica más notable es que no dependen de la estructura de ningún computador en concreto. Algunos de los lenguajes más representativos de esta generación son: Fortran, Cobol, Algol y Basic.

7.3.3 Lenguajes de tercera generación

Podemos verlos bajo dos enfoques:

1º: Programación imperativa: son lenguajes fuertemente tapados, con redundancias entre la declaración y el uso de cada tipo de dato, facilitando así la verificación en la compilación de las posibles inconsistencias.

- **Pascal:** tiene una tipificación de datos rígida, una compilación y separada deficiente.
- **Modula-2:** se separan la especificación del modulo de su realización concreta. Con esto tenemos una compilación segura y permite la Ocultación de los detalles de la implementación.
- **C:** es flexible. No tiene restricción en los datos.
- **Ada:** permite la definición de elementos genéricos, la programación concurrente de tareas y su sincronización y cooperación.

2º: Orientado a objetos, funcional o lógico: están orientados hacia el campo para el cual han sido diseñados.

Smalltalk: precursor de los lenguajes orientados a objetos.

C++: se incorpora a C, la Ocultación, las clases, herencia y polimorfismo.

Eiffel: es un lenguaje nuevo completamente orientado a objetos.

Permite la definición de clases genéricas, herencia múltiple y polimorfismo.

Lisp: precursor de los lenguajes funcionales Los datos se organizan en listas que se manejan con funciones, normalmente recursivas.

Prolog: representante de los lenguajes lógicos. Se utiliza en la construcción de sistemas expertos.

7.3.4 Lenguajes de cuarta generación

Tienen un mayor nivel de abstracción. Normalmente no son de propósito general; No son aconsejables para desarrollar aplicaciones complejas, por lo ineficiente del código que generan. Según su aplicación concreta son:

- ***Bases de datos:*** el mismo usuario genera sus informes, listados, resúmenes cuando los necesita.
- ***Generadores de programas***. Se pueden construir elementos abstractos fundamentales en un cierto campo de aplicación sin descender a los detalles concretos que se necesitan en los lenguajes de tercera generación. La mayoría generan aplicaciones de gestión en Cobol, y últimamente se han desarrollado herramientas CASE para diseño orientado a objetos, que se pueden utilizar para aplicaciones de sistemas y que generan programas en C, C++, o Ada.
- ***Calculo*:** hojas de cálculo, simulación y diseño para control. etc.
- ***Otros***: herramientas para la especificación y verificación formal de programas, lenguajes de simulación, lenguajes de prototipos.

7.4 Criterios de selección del lenguaje.

Para seleccionar el lenguaje deberían prevalecer los criterios técnicos, pero existen otros factores de tipo operativo que deben ser tenidos muy en cuenta. Estos factores son:

Imposición del cliente: es el cliente el que fija el lenguaje que se debe utilizar. Con esto se consigue disminuir los costes de desarrollo y mantenimiento que se producen cuando se utilizan cientos de lenguajes diferentes.

En otras ocasiones el cliente no es tan drástico y simplemente establece una relación reducida de lenguajes que se pueden usar en sus desarrollos.

Tipo de aplicación: cada día aparecen nuevos lenguajes asociados a un campo de aplicación concreto. Solo en determinadas aplicaciones estaría justificado el empleo de lenguajes ensambladores. En este caso, la parte realizada en ensamblador debería quedar reducida exclusivamente a lo imprescindible.

Disponibilidad y entorno: hay que ver que compiladores existen para el computador elegido. Un factor muy importante a tener en cuenta en la selección, es el entorno que acompaña al compilador. El desarrollo será más sencillo cuanto más potentes sean las herramientas disponibles. Por otro lado también se deben considerar las facilidades de manejo de estas herramientas.

Experiencia previa: siempre que sea posible es más rentable aprovechar la experiencia previa del equipo de trabajo. Cuando las condiciones de trabajo no se modifican el rendimiento aumenta y se disminuyen las posibilidades de error. Hay que tener en cuenta que la formación de todo el personal es una inversión muy importante.

Reusabilidad: es interesante conocer las librerías disponibles y tener herramientas que organicen las mismas para facilitar la búsqueda y el almacenamiento de los módulos reutilizables.

Transportabilidad: está ligada a que exista un estándar del lenguaje que se pueda adoptar en todos los compiladores.

Uso de varios lenguajes: aunque no es aconsejable mezclar varios en un mismo proyecto, hay ocasiones en las que las distintas partes del mismo resulta mucho más sencillas de codificar si se utilizan diferentes lenguajes.

7.5 Aspectos metodológicos.

Se repasan aspectos metodológicos que pueden mejorar la codificación.

7.5.1 Normas y estilo de codificación

Se deben fijar normas concretando un estilo de codificación, para que todo el equipo lo adopte y respete, en el que se deberán concretar lo siguiente:

Formato y contenido de las cabeceras de cada modulo. Formato y contenido para cada tipo de comentario. Utilización del encolumnado.

Elección de nombres.

Además hay que fijar las siguientes restricciones: Tamaño máximo de subrutinas.

Tamaño máximo de argumentos en subrutinas. Evitar anidamiento excesivo.

7.5.2 Manejo de errores

Durante la ejecución de un programa se pueden producir fallos.

Conceptos a tener en cuenta:

Defecto: errata de software. Puede permanecer oculto durante un tiempo indeterminado, si los elementos defectuosos no intervienen en la ejecución del programa.

Fallo: un elemento del programa no funciona correctamente, produciendo un resultado (parcial) erróneo.

Error: estado inadmisible de un programa al que se llega como consecuencia de un fallo. Típicamente consiste en la salida o almacenamiento de resultados incorrectos.

Actitudes respecto al tratamiento de errores son:

No considerar los errores: se exige que los datos introducidos sean correctos y que el programa no tenga ningún defecto. Cuando no se cumpla alguno de estos requisitos, no será posible garantizar el resultado correcto del programa.

Prevención de errores: es la programación a la defensiva en que el programa desconfía sistemáticamente de los datos o argumentos introducidos. Se evitan resultados incorrectos a base de no

producir resultados en caso de fallo. Una ventaja es que se evita la propagación de errores, facilitando el diagnostico de los defectos.

Recuperación de errores: cuando no se pueden detectar todos los posibles fallos es inevitable que en el programa se produzcan errores. En este caso se puede hacer un tratamiento del error con el objetivo de restaurar el programa en un estado correcto y evitar que el error se propague. Para esto se hacen dos cosas:

Detección del error: hay que concretar que situaciones se consideran erróneas y realizar las comprobaciones adecuadas en ciertos puntos del programa.

Recuperación del error: se adoptan decisiones sobre como corregir el estado del programa para llevarlo a una situación consistente.

Tenemos:

Recuperación hacia adelante: se identifica la naturaleza o el tipo de error y se toman las acciones que corrijan el estado del programa y pueda continuar su ejecución. Este esquema se puede programar mediante el mecanismo de excepciones. Por ejemplo, error de fuera de rango, etc.

Recuperación hacia atrás: trata de corregir el estado del programa restaurándolo a un estado anterior correcto a la aparición del error. En la nueva operación se parte de ese último estado correcto para obtener otro nuevo estado. Si ese nuevo estado es también correcto, la operación se da por terminada satisfactoriamente. En caso de error, se restaura el estado anterior y se trata de realizar la misma operación por un camino o algoritmo diferente.

Esta es la forma de operar de los sistemas basados en transacciones. Una transacción es una operación que puede terminar con éxito, modificando el estado del sistema, o con fallo, en cuyo caso la transacción se aborta y se restaura el estado inmediatamente anterior, de manera que la operación no produce ningún efecto. Los esquemas de transacciones mantienen la consistencia en las bases de datos.

Los sistemas que realizan una previsión o recuperación de errores se llaman tolerantes a fallos.

7.5.3 Aspectos de eficiencia

Con la potencia actual no se debe sacrificar la claridad en la codificación por una mayor eficiencia. La eficiencia se puede analizar desde varios puntos de vista:

Eficiencia en memoria: resulta suficiente recurrir a un compilador con posibilidades de compresión de memoria, y adopción de algoritmos eficientes.

Eficiencia en tiempo: adquiere su mayor importancia en sistemas de tiempo real. Esta eficiencia la podemos conseguir mediante la adopción de algoritmos eficientes y haciendo a veces un perjuicio a la eficiencia en memoria y utilizando técnicas de codificación como: Tabular cálculos complejos, Empleo de macros, Desplegado de bucles, Sacar fuera de los bucles lo que no haya que repetir, Evitar operaciones en coma flotante. Etc.

7.5.4 Transportabilidad del software

La transportabilidad permite usar el mismo software en distintos computadores actuales y futuros. Como factores esenciales de la transportabilidad se pueden destacar los siguientes:

Utilización de Estándares: un producto software desarrollado exclusivamente sobre elementos estándar es teóricamente transportable sin ningún cambio. La falta de estándares es uno de los problemas que dificulta la transportabilidad.

Aislar las Peculiaridades: la mejor manera de aislar las peculiaridades es destinar un modulo especifico para cada una de ellas. El transporte se resolverá recodificando y adaptando solamente estos módulos específicos al nuevo computador. Las peculiaridades fundamentales de los computadores suelen estar vinculadas a la arquitectura del computador y el sistema operativo.

Capítulo

8

Pruebas de Software

Centro Asociado Palma de Mallorca

Tutor: Antonio Rivero Cuesta

8 Pruebas de Software.

8.1 Introducción.

A lo largo de la fase de codificación se introducen de manera inadvertida múltiples errores de todo tipo e incorrecciones respecto a las especificaciones del proyecto. Todo esto debe ser detectado y corregido antes de entregar al cliente el programa acabado.

Para garantizar la calidad del producto es necesario someter al mismo a diversas pruebas destinadas a detectar errores.

Para un software crítico, el costo de las pruebas puede ser la partida más importante del costo de todo el desarrollo.

Para evitar el caos de una prueba global única, se deben hacer pruebas por modulo, pruebas de integración y pruebas del sistema total.

8.2 Objetivos

Conocer los distintos tipos de pruebas que se deben o pueden realizar a un software para garantizar de forma razonable su correcto funcionamiento.

Se revisarán además las técnicas más utilizadas en función del tipo de lenguaje de programación.

8.3 Tipos de pruebas

Tienen un doble objetivo: la verificación y la validación.

La verificación persigue comprobar que se ha construido el producto correctamente.

La validación comprueba que el producto se ha construido de acuerdo a los requisitos del cliente.

Los procesos de verificación y validación llevan a realizar pruebas a diferentes niveles, en los que se revisa la calidad de cada etapa del proceso de elaboración del software.

Ver figura 8.1, página 284.

Tenemos pruebas de:

- Unidades.
- Integración.
- Validación.
- Sistema.

8.4 Pruebas de unidades

El principal objetivo de las pruebas es conseguir que el programa funcione incorrectamente y que se descubran sus defectos. Esto exige elaborar un juego de pruebas que someta al programa a múltiples situaciones.

Con las pruebas solo se explora una parte de todas las posibilidades del programa (Ver Figura página 286). Se pretende que con el mínimo esfuerzo posible se detecten el mayor número posible de defectos.

Para garantizar unos resultados fiables, todo el proceso de prueba se debe realizar de la manera más automática posible. Para ello, se debe crear un entorno de prueba que asegure unas condiciones predefinidas y estables para las sucesivas pasadas.

Las pruebas de unidades se realizan en un entorno de ejecución controlado, que puede ser diferente del entorno de ejecución del programa final en explotación. El entorno deberá proporcionar al menos un informe con los resultados de las pruebas y un registro de todos los errores detectados con su discrepancia respecto al valor esperado.

Se diferencian dos técnicas de prueba de unidades:

8.4.1 Pruebas de caja negra

Se ignora la estructura interna del programa y se basa exclusivamente en la comprobación de la especificación entrada- salida del software. Ver Figura Página 274.

Se trata de verificar que todos los requisitos impuestos al programa se cumplen.

Es la única estrategia que puede adoptar el cliente. El objetivo es descubrir los errores de los módulos sospechosos y para ello hay que elegir un interrogatorio amplio y coherente.

Los métodos que ayudan en la elaboración de casos de prueba son:

Partición en clases de equivalencia: se trata de dividir el espacio de ejecución del programa en varios subespacios. Cada subespacio o clase equivalente agrupa a todos aquellos datos de entrada al modulo que resultan equivalentes desde el punto de vista de la prueba de la caja negra. Ver Figura Página 288.

Por ejemplo, la equivalencia podría corresponder a que el algoritmo de cálculo, tal como se describe externamente, siga los mismos pasos en su ejecución.

Supongamos que estamos probando una función que realiza la raíz cuadrada. En este caso sería suficiente probar cualquier número positivo, el cero y cualquier número negativo. De esta forma tendríamos tres clases equivalentes. Ahora si probáramos un cuadrado perfecto positivo, un valor positivo sin raíz exacta, el cero, un cuadrado perfecto negativo y un valor negativo arbitrario, tendríamos cinco clases de equivalencia.

Hay que tener en cuenta que un caso de prueba valido para una clase puede ser también un caso de prueba invalido para otra y asimismo puede ocurrir que un caso de prueba bien elegido sea válido para varias clases.

Los pasos a seguir con este método son los siguientes:

- Definir las clases equivalentes.
- Definir una prueba que cubra tantos casos validos como sea posible de cualquier clase.
- Marcar las clases cubiertas y repetir el paso anterior hasta cubrir todos los casos validos de todas las clases.
- Definir una prueba específica para cada caso inválido. Leer Página 276.

Análisis de valores límite: hace un especial hincapié en las zonas del espacio de ejecución que están próximas al borde. Los errores en los límites pueden ser graves al solicitar recursos que no se habían preparado. Se deben proponer casos validos y casos inválidos.

Por ejemplo, si tenemos: 0<= Edad <120 años, los casos de prueba serían:

Límite Inferior : -1 0 1

Límite Superior: 119 120 121. Ver Figura Página 277.

Leer Página 278.

Comparación de versiones: se hacen diferentes versiones del mismo modulo hechas por diferentes programadores y se someten al mismo juego de pruebas. Se coparan y evalúan los resultados.

Cuando produzcan los mismos resultados podemos elegir cualquier versión. Esto no es infalible pues un error en la especificación lo arrastrarían todas las versiones.

Empleo de la intuición: la elaboración de pruebas requiere ingenio.

Las personas ajenas al desarrollo del modulo suelen aportar un punto de vista más distante y fresco.

8.4.2 Pruebas de caja transparente

Ahora se conoce y se tiene en cuenta la estructura interna del modulo. Se trata de conseguir que el programa transite por todos los posibles caminos de ejecución y ponga en juego todos los elementos del código.

Si solo efectuamos pruebas de caja negra quedaran inexplorado multitud de caminos.

Las pruebas de caja negra y caja transparente deben ser complementarias y nunca excluyentes.

Los métodos más utilizados son:

Cubrimiento lógico: se determina un conjunto de caminos básicos que recorran las líneas del flujo del diagrama alguna vez.

Cada rombo del diagrama debe representar un predicado lógico simple, es decir, no puede ser una expresión lógica con algún operador OR, AND, etc.

El número de caminos básicos necesarios vendrá determinado por el número de predicados lógicos simples que tenga el diagrama de flujo con arreglo a la siguiente fórmula:

Nº máximo de caminos = Nº de predicados + 1.

Tenemos diferentes niveles de encubrimiento:

- Nivel1: se elaboran casos de prueba para que se ejecuten al menos una vez todos los caminos básicos, cada uno de ellos por separado.
- Nivel2: se elaboran casos de prueba para que se ejecuten todas las combinaciones de caminos básicos por parejas. Ver ejemplo Página 295.
- Nivel3: Se elaboran casos de prueba un número significativo de las combinaciones posibles de caminos. Cubrir todas las combinaciones posibles resulta inabordable.

Como mínimo las pruebas de cada modulo deben garantizar el nivel 1.

Nunca se podrá detectar la falta de un fragmento de código.

Prueba de bucles: se deben elaborar pruebas para:

Bucle con número no acotado de repeticiones: ejecutar el bucle 0, 1, 2, algunas, muchas veces.

Bucle con número máximo (M) de repeticiones: ejecutar el bucle 0, 1, 2, algunas, M-1, M, M+1 veces.

Bucles anidados: ejecutar todos los bucles externos en su número mínimo de veces para probar el bucle mas interno. Para el siguiente nivel de anidamiento, ejecutar los bucles externos en su número mínimo de veces, y los bucles internos un número típico de veces. Repetir así hasta completar todos los niveles.

Bucles concatenados: si son independientes se probaran por separado con los criterios anteriores. Si están relacionados, por ejemplo, el índice final de uno es el inicial del siguiente, se empleara un enfoque similar al indicado para los bucles anidados.

Empleo de la intuición: merece la pena dedicar un cierto tiempo a elaborar pruebas que solo por intuición podemos estimar que platearan situaciones especiales.

8.4.3 Estimación de errores no detectados

Resulta imposible demostrar que un modulo no tiene defectos. Para estimar los defectos que quedan sin detectar procedemos así:

Anotar el número de errores que se producen inicialmente con el juego de casos de prueba. E_I.

Corregir el módulo hasta que no tenga ningún error con el mismo juego de casos de prueba.

Introducir aleatoriamente un número determinado de errores en los puntos más diversos del modulo. E_A.

Someter al módulo con los nuevos errores al juego de casos de prueba y contar los errores que se detectan. E_D.

Suponiendo la misma proporción, el porcentaje de errores sin detectar será al mismo para los errores iniciales que para los deliberados. Por tanto, el número estimado de errores sin detectar será:

$E_E = (E_A - E_D) * (E_I / E_D)$.

8.5 Pruebas de unidades en programación orientada a objetos.

En programación orientada a objetos debemos probar la clase como unidad, en la que las operaciones van modificando los datos encapsulados en ella y por tanto el comportamiento de la clase.

Las secuencias de pruebas de una clase se diseñan para probar las operaciones relevantes en ella y es examinando los valores de los atributos de la clase donde comprobaremos si existen errores.

Existen varios métodos para probar una clase:

- Pruebas basadas en fallo.
- Pruebas aleatorias.
- Pruebas de partición.
- En base a la capacidad de las operaciones de cambiar el estado de la clase.
- En base a los atributos que utiliza cada operación.

8.6 Estrategias de integración.

Los módulos de un producto software se han de integrar para conformar el sistema completo. En esta fase de integración también aparecen nuevos errores y se debe proceder siguiendo estrategias para facilitar la depuración de los errores que vayan surgiendo.

Entre las estrategias básicas de integración tenemos:

8.6.1 Integración Big Bang

Se realiza la integración en un único paso. En este caso la cantidad de errores que aparecen de golpe hace imposible la identificación de los defectos, provocando un caos. Solo para sistemas muy pequeños se puede justificar su utilización.

8.6.2 Integración descendente

Tenemos un modulo principal que se prueba con módulos de andamiaje o sustitutos, que mas tarde se van sustituyendo uno por uno por los verdaderos, realizando las pruebas de integración. Los sustitutos deben ser los más simples posible.

La codificación de los sustitutos es un trabajo adicional que conviene simplificar al máximo.

La ventaja es que se ven desde el principio las posibilidades de la aplicación. Esto permite mostrar muy pronto al cliente un prototipo sencillo y discutir sobre él posibles mejoras o modificaciones.

Sus inconvenientes son que limita tanto el trabajo en paralelo como el ensayo de situaciones especiales.

8.6.3 Integración ascendente

Se codifica por separado y en paralelo todos los módulos de nivel más bajo. Para probarlos se escriben módulos gestores o conductores que los hacen funcionar independientemente o en combinaciones sencillas.

Los Gestores se van sustituyendo uno a uno por los módulos de mayor nivel según se van codificando, al tiempo que se van desarrollando nuevos gestores si hace falta. El orden de sustitución puede ser cualquiera salvo para los últimos pasos en que se necesita que todos los módulos inferiores estén disponibles.

Tiene como ventajas que se facilita el trabajo en paralelo y facilita el ensayo de situaciones especiales. Su inconveniente es que es difícil ensayar el funcionamiento global hasta el final de su integración.

La mejor solución es usar integración ascendente con los módulos de nivel más bajo y descendente con los de nivel más alto, lo que se llama integración sandwich.

8.7 Pruebas de validación.

Cuando finaliza la integración y sus pruebas asociadas y el software está completamente ensamblado, nos queda validar.

La validación se consigue a través de una serie de pruebas que comprueban la conformidad con los requerimientos, además de rendimiento, ergonomía y documentación.

8.7.1 Pruebas alfa

Para comprobar que un producto software es realmente útil para sus usuarios es conveniente que estos últimos intervengan también en las pruebas finales del sistema. Es probable que los problemas más graves aparezcan ya al comienzo y por ello es aconsejable que alguien del equipo de desarrollo acompañe al usuario durante la primera toma de contacto.

8.7.2 Pruebas beta

Uno o varios usuarios trabajan con el sistema en su entorno normal, sin apoyo de nadie, anotando cualquier problema que se presente. Es muy importante que sea el usuario el encargado de transmitir al equipo de desarrollo cual ha sido el procedimiento de operación que llevó al error. Esta información resulta vital para abordar la corrección.

8.8 Pruebas de sistema.

Son pruebas al sistema completo, tenemos las siguientes clases:

Según el objetivo perseguido, tendremos diferentes clases de pruebas:

- ***Pruebas de recuperación***: sirven para comprobar la capacidad del sistema para recuperarse ante fallos.
- **Pruebas de seguridad**: sirven para comprobar los mecanismos de protección contra un acceso no autorizado.
- ***Pruebas de resistencia***: sirven para comprobar el comportamiento del sistema ante situaciones excepcionales.
- ***Pruebas de sensibilidad***: comprobar el tratamiento que da el sistema a ciertas singularidades relacionadas casi siempre con los algoritmos utilizados.
- ***Pruebas de rendimiento***: comprobar las prestaciones del sistema que son críticas en tiempo.
- ***Pruebas de despliegue o configuración***: sirven para comprobar el funcionamiento del software que debe ejecutarse en varias plataformas y sistemas operativos distintos. Permite comprobar la correcta instalación del programa y la documentación para ello.