Addenda de la asignatura Análisis, Diseño y Mantenimiento del Software

Manuel Arias Calleja
Dpto. de Inteligencia Artificial - ETSI Informática - UNED

Actualizada en agosto de 2007

Índice general

1. Contexto de la asignatura en la Ingeniería de Software **1**
1.1. Necesidad de una metodología 1
1.1.1. Sistemas 1
1.1.2. La crisis del software 2
1.1.3. Definición de metodología 2
1.1.4. Finalidad de una metodología 3
1.1.5. Taxonomía de las metodologías 3
1.2. Ciclo de vida del software 5
1.2.1. Ciclos de vida en cascada 7
1.2.2. Modelo de ciclo de vida en espiral 10
1.2.3. Ciclos de vida orientados a objetos 13
1.3. Notaciones de especificación y diseño (UML) 14
1.3.1. Introducción 14
1.3.2. Modelo de casos de uso 17
1.3.3. Modelo estructural estático 19
1.3.4. Modelo de comportamiento 30
1.3.5. Modelo estructural de implementación 36
Ejercicios y actividades 39

2. Fase de especificación **41**
2.1. Obtención de requisitos 41
2.1.1. Introducción 41
2.1.2. Técnicas de obtención de requisitos 43
2.2. Análisis de requisitos 57
2.3. Representación de requisitos 58
2.4. Análisis orientado a objetos 58
2.5. Bases de documentación 58
Ejercicios y actividades 64

3. Fase de diseño **65**
3.1. Conceptos y elementos del diseño 65

3.1.1. Patrones 65
3.2. Diseño estructurado 72
3.3. Diseño orientado a objetos 72
3.4. Validación y confirmación del diseño 75
3.4.1. Revisión del diseño 75
3.4.2. Verificación del diseño 75
3.4.3. Validación del diseño 75
3.5. Documentación: especificación del diseño 75
Ejercicios y actividades 78

4. Fase de implementación **79**
4.1. Técnicas de depuración 79
4.2. Documentación del código 79
4.2.1. Tipos de comentarios 79
4.2.2. Consideraciones generales 80
Ejercicios 81

5. Fases de pruebas **83**
5.1. Técnicas y métodos de prueba 83
5.2. Documentación de pruebas 83
Ejercicios 86

6. Fase de entrega y mantenimiento **89**
6.1. Finalización del proyecto 89
6.1.1. Aceptación 90
6.1.2. Informe de cierre del proyecto 90
6.1.3. Indicadores del proyecto 90
6.2. Planificación de revisiones y organización del mantenimiento 92
6.3. Recopilación y organización de documentación 92
6.3.1. Motivos de la documentación 92
6.3.2. Directrices que se deben seguir para la redacción de un documento . . . 93
6.3.3. Tipos de documentos 93
6.3.4. Manual de usuario 95
6.3.5. Manual del sistema 97
6.3.6. Manual de mantenimiento 100
Ejercicios y actividades 102

7. Metodologías de desarrollo **103**
7.1. Introducción a las metodologías de desarrollo 103
7.2. Proceso unificado de Rational 103
7.2.1. Introducción 104

7.2.2. Las cuatro “P”: Personas, Proyecto, Producto y Proceso 106
7.2.3. Proceso dirigido por casos de uso 107
7.2.4. Proceso centrado en la arquitectura 108
7.2.5. Proceso iterativo e incremental 109
7.2.6. Captura de requisitos 111
7.2.7. Diseño 117
7.2.8. Implementación 120
7.2.9. Prueba 122
7.3. Método extreme programming 124
7.3.1. Historias de usuario 125
7.3.2. Plan de publicación de versiones 125
7.3.3. Tarjetas CRC: Clases, Responsabilidades y Colaboraciones 126
7.3.4. Planificación de cada iteración 126
7.3.5. Integración 127
7.3.6. Codificación de cada unidad 128
7.3.7. Recomendaciones generales 130
7.4. Métrica 3 132
7.4.1. Introducción 132
7.4.2. Objetivos 132
7.4.3. Estructura 132
7.4.4. Planificación de Sistemas de Información 133
7.4.5. Estudio de la viabilidad del sistema 135
7.4.6. Análisis del sistema de información 136
7.4.7. Diseño del sistema de información 138
7.4.8. Construcción del sistema de información 141
7.4.9. Implantación y Aceptación del Sistema 143
7.4.10. Mantenimiento de Sistemas de Información 144
7.5. Métodos de software libre: “cathedral” vs. “bazaar” 154
7.5.1. La catedral 154
7.5.2. El bazar 155
Ejercicios y actividades 157

8. Herramientas de desarrollo y validación 159
8.1. Herramientas CASE 159
8.1.1. Funciones de las herramientas CASE 159
8.1.2. Clasificación de las herramientas CASE 160
8.2. Gestión de la configuración 162
8.2.1. Terminología y definiciones básicas 162
8.2.2. Identificación de la configuración 163
8.2.3. Control de cambios 165
8.2.4. Generación de informes de estado 165
8.2.5. Auditoría de la configuración 166

8.2.6. Plan de gestión de la configuración 166
8.2.7. Herramientas de la Gestión de la Configuración 167
8.2.8. Software libre 169
8.3. Entornos de desarrollo de interfaces 171
8.3.1. Introducción 171
8.3.2. Componentes 172
8.3.3. Creación de interfaces de usuario 173
8.3.4. Metodología 174
8.3.5. Heurísticas de usabilidad 175
8.3.6. Glade 175
8.3.7. GTK+ 176
8.3.8. Anjuta 176
Ejercicios y actividades 178

Bibliografía **181**

Índice alfabético **182**

Capítulo 1

Contexto de la asignatura en la Ingeniería de Software

1.1. Necesidad de una metodología

El proceso de construcción del software requiere, como en cualquier otra ingeniería, identificar las tareas que se han de realizar sobre el software y aplicar esas tareas de una forma ordenada y efectiva. Adicionalmente y aunque no es el objeto principal de esta asignatura, el desarrollo del software es realizado por un conjunto coordinado de personas simultáneamente y, por lo tanto, sus esfuerzos deben estar dirigidos por una misma metodología que estructure las diferentes fases del desarrollo. En temas posteriores, en primer lugar se explicarán en detalle las mencionadas fases y a continuación las metodologías usuales que las gestionan.

En esta asignatura se hará especial énfasis en los temas relacionados con el análisis, diseño y mantenimiento del software, mencionando apenas los temas específicos de la organización y gestión de proyectos que conciernen a la distribución de medios y tareas entre las personas a cargo del desarrollo, ya que estos últimos temas son objeto de otras asignaturas de la titulación.

1.1.1. Sistemas

La **Ingeniería de Sistemas** es el contexto genérico en que se sitúan las herramientas y metodologías usadas para crear sistemas. Un sistema puede estar formado por subsistemas de diferentes tipos. La ingeniería de sistemas constituye el primer paso de toda ingeniería: proporciona una visión global de un sistema en su conjunto para que, posteriormente, cada uno de sus subsistemas se analice con la rama de la ingeniería adecuada. Una de esas ramas es la ingeniería del software.

La **Ingeniería del Software** trata, según Bauer [Bau72], del establecimiento de los principios y métodos de la ingeniería, orientados a obtener software económico, que sea fiable y funcione de manera eficiente sobre máquinas reales.

El sistema es un conjunto de elementos que cooperan entre sí para proporcionar una funcionalidad. En el caso de un sistema informático, consta de dos tipos de elementos: hardware y software.

1.1.2. La crisis del software

Desde el momento en que se introdujeron computadores con capacidad para soportar aplicaciones de tamaño considerable (años 60), las técnicas de desarrollo para los hasta entonces pequeños sistemas dejaron progresivamente de ser válidas. Estas técnicas consistían básicamente en codificar y corregir (*code-and-fix*) que se resume en lo siguiente:

> No existe necesariamente una especificación del producto final, en su lugar se tienen algunas anotaciones sobre las características generales del programa. Inmediatamente se empieza la codificación y simultáneamente se va depurando. Cuando el programa cumple con las especificaciones y parece que no tiene errores se entrega.

La ventaja de este enfoque es que no se gasta tiempo en planificación, gestión de los recursos, documentación, etc. Si el proyecto es de un tamaño muy pequeño y lo realiza un equipo pequeño (una sola persona o dos) no es un mal sistema para producir un resultado pronto. Hoy en día es un método de desarrollo que se usa cuando hay plazos muy breves para entregar el producto final y no existe una exigencia explícita por parte de la dirección de usar alguna metodología de ingeniería del software. Puede dar resultado, pero la calidad del producto es imprevisible. Las consecuencias de este tipo de desarrollo son:

1. El costo es mucho mayor de lo originalmente previsto.
2. El tiempo de desarrollo excede lo proyectado.
3. La calidad del código producido es imprevisible y en promedio baja.

Aunque se han desarrollado técnicas para paliar estos problemas, hoy en día aún se considera normal que una aplicación rebase sus proyecciones iniciales de tiempo y dinero y que se descubran *bugs* (errores informáticos) en su ejecución. Esto se debe a que todavía no se aplica de un modo sistemático la ingeniería del software durante el desarrollo.

1.1.3. Definición de metodología

En la literatura sobre este tema existen muchas definiciones sobre lo que es una metodología. El común denominador de todas ellas es la siguiente lista de características:

1. Define cómo se divide un proyecto en fases y las tareas que se deben realizar en cada una de ellas.
2. Especifica para cada una de las fases cuáles son las entradas que recibe y las salidas que produce.
3. Establece alguna forma de gestionar el proyecto.

Resumiendo lo anterior definimos *metodología* como un modo sistemático de producir software.

1.1.4. Finalidad de una metodología

La diferencia entre *code-and-fix* (no usar ninguna metodología) y usar una es que se pueden alcanzar los siguientes atributos en el producto final:

1. *Adecuación*: el sistema satisface las expectativas del usuario.
2. *Mantenibilidad*: facilidad para realizar cambios una vez que el sistema está funcionando en la empresa del cliente.
3. *Usabilidad*: facilidad de aprender a manejar el sistema por parte de un usuario que no tiene por qué ser informático. La resistencia de los usuarios a aceptar un sistema nuevo será mayor cuanto peor sea la usabilidad.
4. *Fiabilidad*: capacidad de un sistema de funcionar correctamente durante un intervalo de tiempo dado. La diferencia con la corrección es que en este atributo interesa el tiempo, es decir, no se trata del número absoluto de defectos en el sistema sino de los que se manifiestan en un intervalo de tiempo. Interesan sobre todo:
    a) *MTBF* (*Mean Time Between Failures*): tiempo medio entre fallos.
    b) *Disponibilidad*: probabilidad de que el sistema esté funcionando en un instante dado.
5. *Corrección*: baja densidad de defectos.
6. *Eficiencia*: capacidad del sistema de realizar su tarea con el mínimo consumo de recursos necesario.

1.1.5. Taxonomía de las metodologías

Existen dos grupos de metodologías en función de la mentalidad con la que aborda un problema: metodología estructurada y metodología orientada a objetos.

Metodología estructurada

Constituyó la primera aproximación al problema del desarrollo de software. Está orientada a procesos, es decir, se centra en especificar y descomponer la funcionalidad del sistema. Se utilizan varias herramientas:

- *Diagramas de flujo de datos (DFD)*: representan la forma en que los datos se mueven y se transforman. Incluyen:
    - Procesos
    - Flujos de datos
    - Almacenes de datos

  Los procesos individuales se pueden a su vez ”explosionar” en otros DFD de mayor nivel de detalle.

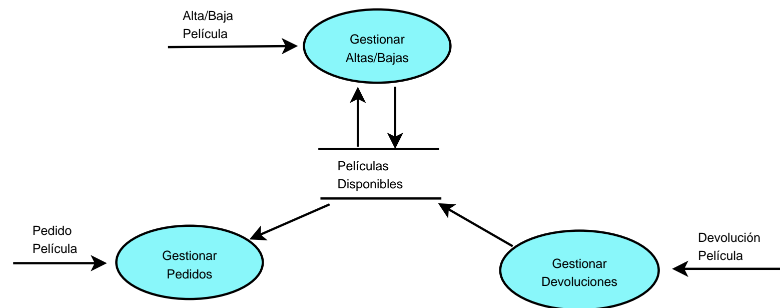


Figura 1.1: Ejemplo de DFD

- *Especificaciones de procesos*: descripciones de los procesos definidos en un DFD que no se puede descomponer más. Pueden hacerse en pseudocódigo, con tablas de decisión o en un lenguaje de programación.
- *Diccionario de datos*: nombres de todos los tipos de datos y almacenes de datos junto con sus definiciones.
- *Diagramas de transición de estados*: modelan procesos que dependen del tiempo.
- *Diagramas entidad-relación*: los elementos del modelo E/R se corresponden con almacenes de datos en el DFD. En este diagrama se muestran las relaciones entre dichos elementos.

Los lenguajes de programación también reflejan esta dicotomía que existe entre las metodologías; así, existen lenguajes para la programación estructurada. Los más famosos son: Cobol (destinado a aplicaciones financieras, en su mejor momento el 90 % del código estaba escrito en este lenguaje), Fortran (usado en aplicaciones matemáticas), C (aplicaciones de propósito general en los años 80), Pascal y Modula 2.

Metodología orientada a objetos

Constituye una aproximación posterior. Cuenta con mayor número de desarrolladores y es previsible que termine sustituyendo a la anterior. Además, es ampliamente admitido que proporciona estas ventajas:

1. Está basada en componentes, lo que significa que facilita la reutilización de código. De todos modos hay que añadir que la reutilización de código es un tema complejo y que requiere en cualquier caso un diseño muy cuidadoso. Una metodología orientada a objetos no garantiza por si misma la producción de código reutilizable, aunque lo facilita.
2. Simplifica el mantenimiento debido a que los cambios están más localizados.

La mentalidad que subyace al diseño estructurado es: ¿Cómo se puede dividir el sistema en partes más pequeñas que puedan ser resueltas por **algoritmos** sencillos y qué información se intercambian?. En el diseño orientado a objetos la idea es sin embargo: ¿Cuáles son los tipos de **datos** que hay que utilizar, qué características tienen y cómo se relacionan?.

La orientación a objetos supone un paradigma distinto del tradicional (no necesariamente mejor o peor) que implica focalizar la atención en las estructuras de datos. El concepto de objetos tuvo sus orígenes en la inteligencia artificial como un modo de representación del conocimiento. El primer lenguaje orientado a objetos fue **Simula67**, desarrollado por Kristen Nggaard y Ole-Johan Dahl en el Centro de Cálculo noruego, pero el que se considera el primer lenguaje orientado a objetos puro fue **Smaltalk**, donde todos los elementos del lenguaje son objetos. El lenguaje **C++** fue una ampliación de C para que soportara objetos; resultó muy eficiente pero también muy complejo. **Java** es otro lenguaje orientado a objetos derivado del C++ pero más sencillo y concebido con la idea de minimizar los errores relativos a punteros.

Objetos y clases Un objeto consta de una estructura de datos y de una colección de métodos (antes llamados procedimientos o funciones) que manipulan esos datos. Los datos definidos dentro de un objeto son sus atributos. Un objeto sólo puede ser manipulado a través de su interfaz, esto es, de una colección de funciones que implementa y que son visibles desde el exterior.

Los conceptos importantes en el contexto de clases y objetos se pueden consultar en [Pre01, sec. 20.1] y [Pre01, sec. 20.2]. En resumen son:

1. *Clases*: describen abstracciones de datos y operaciones necesarias para su manipulación. Dentro de una clase se definen:
   - *Atributos*: los datos contenidos en la clase.
   - *Métodos*: los algoritmos internos de la clase.
2. *Polimorfismo*: capacidad de un objeto de presentar varios comportamientos diferentes en función de cómo se utilice; por ejemplo, se pueden definir varios métodos con el mismo nombre pero diferentes argumentos.
3. *Herencia*: relación de generalización; cuando varias clases comparten características comunes, éstas se ponen en una clase antecesora.
4. *Asociación*: relación entre clases que cooperan con alguna finalidad.

Durante la etapa de análisis se **identifican** los objetos del dominio del problema. En el diseño se **definen** las características de los objetos.

1.2. Ciclo de vida del software

Al igual que otros sistemas de ingeniería, los sistemas de software requieren un tiempo y esfuerzo considerable para su desarrollo y deben permanecer en uso por un periodo mucho mayor. Durante este tiempo de desarrollo y uso, desde que se detecta la necesidad de construir un sistema de software hasta que éste es retirado, se identifican varias etapas (o fases) que en conjunto se denominan ciclo de vida del software. En cada caso, en función de cuáles sean las características del proyecto, se configurará el ciclo de vida de forma diferente. Usualmente se consideran las fases: especificación y análisis de *requisitos*, *diseño* del sistema, *implementación* del software, aplicación y *pruebas*, entrega y *mantenimiento*. Un aspecto esencial dentro de las

tareas del desarrollo del software es la *documentación* de todos los elementos y especificaciones en cada fase. Dado que esta tarea siempre estará influida por la fase del desarrollo en curso, se explicará de forma distribuida a lo largo de las diferentes fases como un apartado especial para recalcar su importancia en el conjunto del desarrollo del software.

Tal como ya hemos mencionado, las fases principales en cualquier ciclo de vida son:

1. *Análisis*: se construye un modelo de los requisitos
2. *Diseño*: partiendo del modelo de análisis se deducen las estructuras de datos, la estructura en la que descompone el sistema, y la interfaz de usuario.
3. *Codificación*: se construye el sistema. La salida de esta fase es código ejecutable.
4. *Pruebas*: se comprueba que se cumplen criterios de corrección y calidad.
5. *Mantenimiento*: se asegura que el sistema siga funcionando y adaptándose a nuevos requisitos.

Las etapas se dividen en actividades que constan de tareas. La *documentación* es una tarea importante que se realiza en todas las etapas y actividades. Cada etapa tiene como entrada uno o varios documentos procedentes de las etapas anteriores y produce otros documentos de salida según se muestra en la figura 1.2.

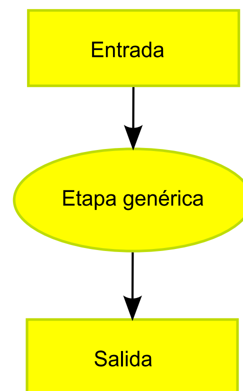


Figura 1.2: Etapa genérica

Algunos autores dividen la fase del diseño en dos partes: *diseño global* o arquitectónico y *diseño detallado*. En el primero se transforman los requisitos en una arquitectura de alto nivel, se definen las pruebas que debe satisfacer el sistema en su conjunto, se esboza la documentación y se planifica la integración. En el detallado, para cada módulo se refina el diseño y se definen los requisitos del módulo y su documentación.

Las formas de organizar y estructurar la secuencia de ejecución de las actividades y tareas en las diferentes fases de cada uno de los métodos pueden dar lugar a un tipo de ciclo de vida diferente. Los principales ciclos de vida que se van a presentar a continuación realizan todas las fases, actividades y tareas. Cada uno de ellos tiene sus ventajas e inconvenientes.

1.2.1. Ciclos de vida en cascada

El ciclo de vida inicialmente propuesto por Royce en 1970, fue adaptado para el software a partir de ciclos de vida de otras ramas de la ingeniería. Es el primero de los propuestos y el más ampliamente seguido por las organizaciones (se estima que el 90 % de los sistemas han sido desarrollados así). La estructura se muestra en la figura 1.3.

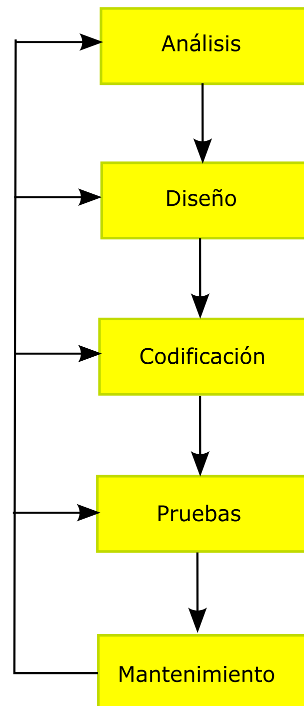


Figura 1.3: Ciclo de vida en cascada

Descripción

Este modelo admite la posibilidad de hacer iteraciones. Así por ejemplo, si durante el mantenimiento se ve la necesidad de cambiar algo en el diseño se harán los cambios necesarios en la codificación y se tendrán que realizar de nuevo las pruebas, es decir, si se tiene que volver a una de las etapas anteriores al mantenimiento se recorrerán de nuevo el resto de las etapas.

Después de cada etapa se realiza una revisión para comprobar si se puede pasar a la siguiente.

Trabaja en base a documentos, es decir, la entrada y la salida de cada fase es un tipo de documento específico. Idealmente, cada fase podría hacerla un equipo diferente gracias a la documentación generada entre las fases. Los documentos son:

- Análisis: Toma como entrada una descripción en lenguaje natural de lo que quiere el cliente. Produce el S.R.D. (Software Requirements Document).
- Diseño: Su entrada es el S.R.D. Produce el S.D.D. (Software Design Document)

- Codificación: A partir del S.D.D. produce módulos. En esta fase se hacen también pruebas de unidad.
- Pruebas: A partir de los módulos probados se realizan la integración y pruebas de todo el sistema. El resultado de las pruebas es el producto final listo para entregar.

Ventajas

- La planificación es sencilla.
- La calidad del producto resultante es alta.
- Permite trabajar con personal poco cualificado.

Inconvenientes

- Lo peor es la necesidad de tener todos los requisitos al principio. Lo normal es que el cliente no tenga perfectamente definidas las especificaciones del sistema, o puede ser que surjan necesidades imprevistas.
- Si se han cometido errores en una fase es difícil volver atrás.
- No se tiene el producto hasta el final, esto quiere decir que:
    - Si se comete un error en la fase de análisis no se descubre hasta la entrega, con el consiguiente gasto inútil de recursos.
    - El cliente no verá resultados hasta el final, con lo que puede impacientarse .
- No se tienen indicadores fiables del progreso del trabajo (síndrome del 90 %).[1]
- Es comparativamente más lento que los demás y el coste es mayor también.

Tipos de proyectos para los que es adecuado

- Aquellos para los que se dispone de todas las especificaciones desde el principio, por ejemplo, los de reingeniería.
- Se está desarrollando un tipo de producto que no es novedoso.
- Proyectos complejos que se entienden bien desde el principio.

Como el modelo en cascada ha sido el más seguido ha generado algunas variantes. Ahora veremos algunas.

Ciclo de vida en V

Propuesto por Alan Davis, tiene las mismas fases que el anterior pero tiene en consideración el nivel de abstracción de cada una. Una fase además de utilizarse como entrada para la siguiente, sirve para validar o verificar otras fases posteriores. Su estructura está representada en la figura 1.4.

[1] Consiste en creer que ya se ha completado el 90 % del trabajo, pero en realidad queda mucho más porque el 10 % del código da la mayor parte de los problemas

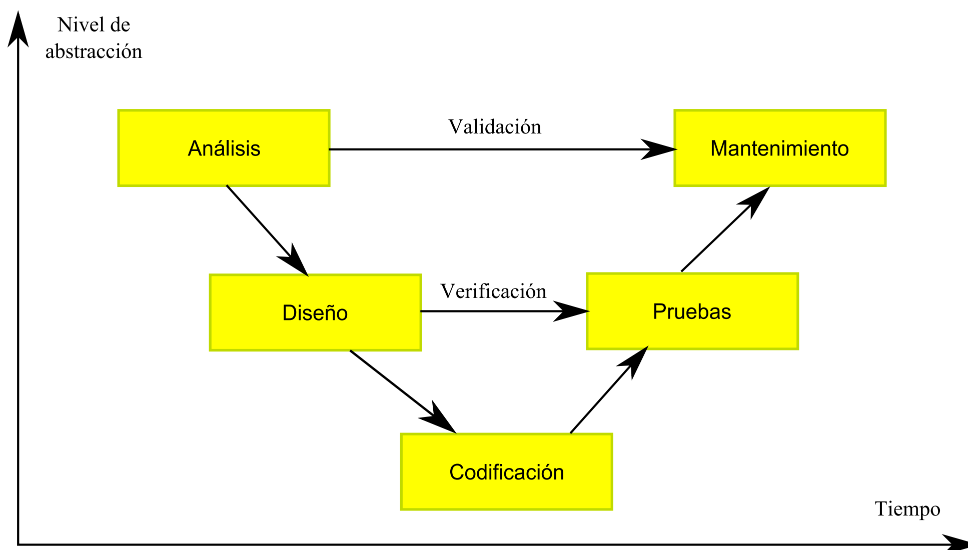


Figura 1.4: Ciclo de vida en V

Ciclo de vida tipo sashimi

Según el modelo en cascada puro una fase sólo puede empezar cuando ha terminado la anterior. En este caso, sin embargo, se permite un solapamiento entre fases. Por ejemplo, sin tener terminado del todo el diseño se comienza a implementar. El nombre “sashimi” deriva del estilo de presentación en rodajas de pescado crudo en Japón. Una ventaja de este modelo es que no necesita generar tanta documentación como el ciclo de vida en cascada puro debido a la continuidad del mismo personal entre fases. Los problemas planteados son:

- Es aún más difícil controlar el progreso del proyecto debido a que los finales de fase ya no son un punto de referencia claro.
- Al hacer cosas en paralelo, si hay problemas de comunicación pueden surgir inconsistencias.

La fase de “concepto” consiste en definir los objetivos del proyecto, beneficios, tipo de tecnología y tipo de ciclo de vida. El diseño arquitectónico es el de alto nivel, el detallado el de bajo nivel. En la figura 1.5 se ha representado la estructura del ciclo de vida sashimi.

Ciclo de vida en cascada con subproyectos

Si, una vez que se ha llegado al diseño arquitectónico, se comprueba que el sistema se divide en varios subsistemas independientes entre sí, sería razonable suponer que a partir de ese punto cada uno se puede desarrollar por separado y en consecuencia en paralelo con los demás. Cada uno tendrá seguramente fechas de terminación distintas. Una vez que han terminado todos se integran y se prueba el sistema en su conjunto. La ventaja es que se puede tener a más gente

Figura 1.5: Ciclo de vida sashimi

trabajando en paralelo de forma eficiente. El riesgo es que existan interdependencias entre los subproyectos.

Ciclo de vida en cascada incremental

En este caso se va creando el sistema añadiendo pequeñas funcionalidades. Cada uno de los pequeños incrementos es comparable a las modificaciones que ocurren dentro de la fase de mantenimiento. La ventaja de este método es que no es necesario tener todos los requisitos en un principio. El inconveniente es que los errores en la detección de requisitos se encuentran tarde.

Hay dos partes en el ciclo de vida, que lo hacen similar al ciclo de vida tipo sashimi. Por un lado están el análisis y el diseño global. Por otro lado están los pequeños incrementos que se llevan a cabo en las fases de diseño detallado, codificación y mantenimiento.

Ciclo de vida en cascada con reducción de riesgos

Como se ha comentado anteriormente, uno de los problemas del ciclo de vida en cascada es que si se entienden mal los requisitos esto sólo se descubrirá cuando se entregue el producto. Para evitar este problema se puede hacer un desarrollo iterativo durante las fases de análisis y diseño global. Esto consistiría en:

1. Preguntar al usuario.
2. Hacer el diseño global que se desprende del punto 1.
3. Hacer un prototipo de interfaz de usuario, hacer entrevistas con los usuarios enseñándoles el prototipo y volver con ello al punto 1 para identificar más requisitos o corregir malentendidos.

El resto es igual al ciclo de vida en cascada.

1.2.2. Modelo de ciclo de vida en espiral

Propuesto inicialmente por Boehm en 1988. Consiste en una serie de ciclos que se repiten. Cada uno tiene las mismas fases y cuando termina da un producto ampliado con respecto al ciclo anterior. En este sentido es parecido al modelo incremental, la diferencia importante es que tiene en cuenta el concepto de riesgo. Un riesgo puede ser muchas cosas: requisitos no comprendidos,

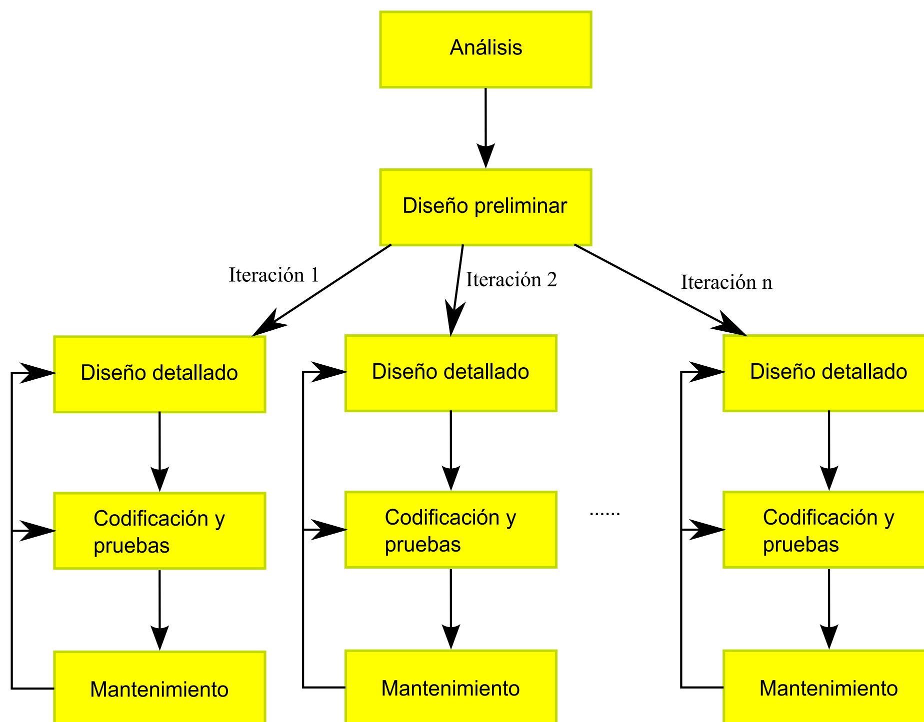


Figura 1.6: Cascada incremental

mal diseño, errores en la implementación, etc. Un representación típica de esta estructura se muestra en la figura 1.7.

En cada iteración Boehm recomienda recopilar la siguiente lista de informaciones:

- *Objetivos*: se obtienen entrevistando a los clientes, haciendo cuestionarios, etc.
- *Alternativas*: son las diferentes formas posibles de conseguir los objetivos. Se consideran desde dos puntos de vista
  - Características del producto.
  - Formas de gestionar el proyecto.
- *Restricciones*: son condiciones o limitaciones que se deben cumplir. Hay dos tipos:
  - Desde el punto de vista del producto: interfaces de tal o cual manera, rendimiento, etc.
  - Desde el punto de vista organizativo: coste, tiempo, personal, etc.
- *Riesgos*: es una lista de peligros de que el proyecto fracase.
- *Resolución de riesgos*: son las posibles soluciones al problema anterior. La técnica más usada es la construcción de prototipos.
- *Resultados*: es el producto que queda después de la resolución de riesgos.

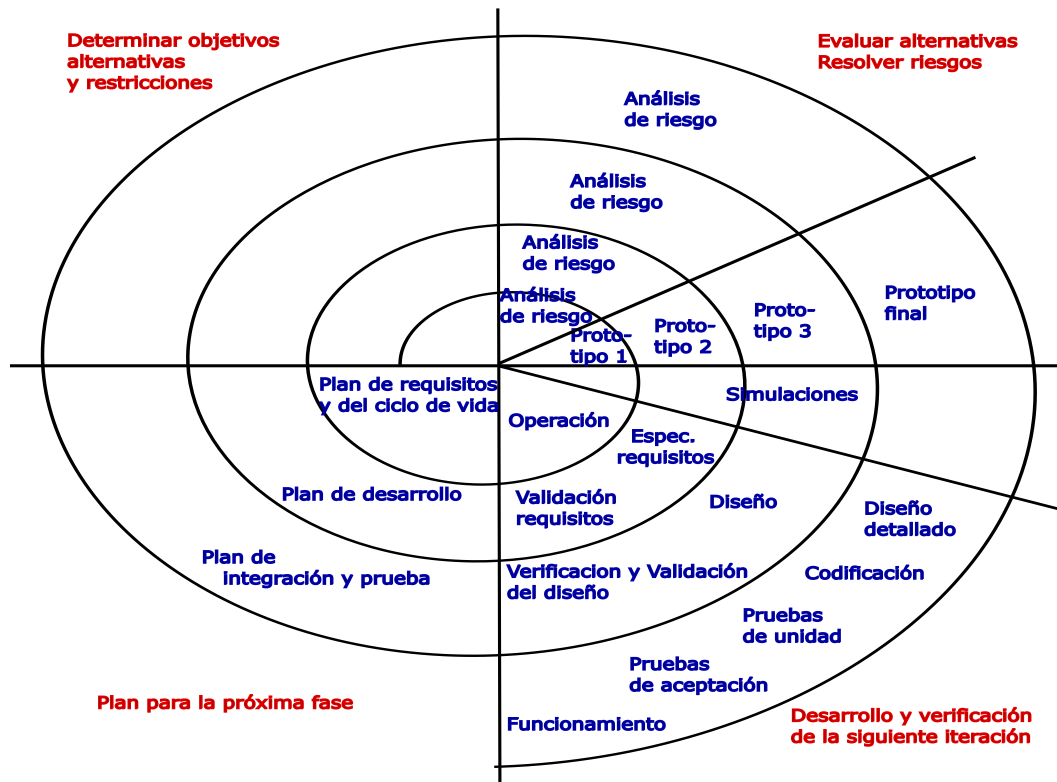


Figura 1.7: Ciclo de vida en espiral

- *Planes*: lo que se va a hacer en la siguiente fase.
- *Compromiso*: son las decisiones de gestión sobre cómo continuar.

Al terminar una iteración se comprueba que lo que se ha hecho efectivamente cumple con los requisitos establecidos; también se verifica que funciona correctamente. El propio cliente evalúa el producto. No existe una diferencia muy clara entre cuándo termina el proyecto y cuándo empieza la fase de mantenimiento. Cuando hay que hacer un cambio, éste puede consistir en un nuevo ciclo.

Ventajas

- No necesita una definición completa de los requisitos para empezar a funcionar.
- Es más fácil validar los requisitos porque se entregan productos desde el final de la primera iteración.
- El riesgo de sufrir retrasos es menor, ya que al identificar los problemas en etapas tempranas hay tiempo de subsanarlos.
- El riesgo en general es menor porque, si todo se hace mal, sólo se ha perdido el tiempo y recursos invertidos en una iteración (las anteriores iteraciones están bien).

Inconvenientes

- Es difícil evaluar los riesgos.
- Necesita de la participación continua por parte del cliente.
- Cuando se subcontrata hay que producir previamente una especificación completa de lo que se necesita, y esto lleva tiempo.

Dónde es adecuado

- Sistemas de gran tamaño.
- Proyectos donde sea importante el factor riesgo.
- Cuando no sea posible definir al principio todos los requisitos.

1.2.3. Ciclos de vida orientados a objetos

Los tipos de ciclos de vida que se han estudiado hasta se han utilizado sobre todo en proyectos desarrollados con análisis y diseño estructurados. El desarrollo de sistemas orientados a objetos tiene la particularidad de estar basados en componentes, que se relacionan entre ellos a través de interfaces, o lo que es lo mismo, son más modulares y por lo tanto el trabajo se puede dividir en un conjunto de miniproyectos. Debido a todo esto, el ciclo de vida típico en una metodología de diseño orientado a objetos es el ciclo de vida en espiral.

Además de lo anterior, hoy en día existe una tendencia a reducir los riesgos y, en este sentido, el ciclo de vida en cascada no proporciona muchas facilidades.

En este texto sólo veremos un tipo de ciclo de vida orientado a objetos, que es además el más representativo: el modelo fuente.

Modelo fuente

Fue creado por Henderson-Sellers y Edwards en 1990. Es un tipo de ciclo de vida pensado para la orientación a objetos y posiblemente el más seguido. Un proyecto se divide en las fases:

1. Planificación del negocio
2. Construcción: es la más importante y se divide a su vez en otras cinco actividades: planificación, investigación, especificación, implementación y revisión.
3. Entrega o "liberación".

La primera y la tercera fase son independientes de la metodología de desarrollo orientado a objetos. Además de las tres fases, existen dos periodos:

1. Crecimiento: es el tiempo durante el cual se construye el sistema
2. Madurez: es el periodo de mantenimiento del producto. Cada mejora se planifica igual que el periodo anterior, es decir, con las fases de Planificación del negocio, Construcción y Entrega.

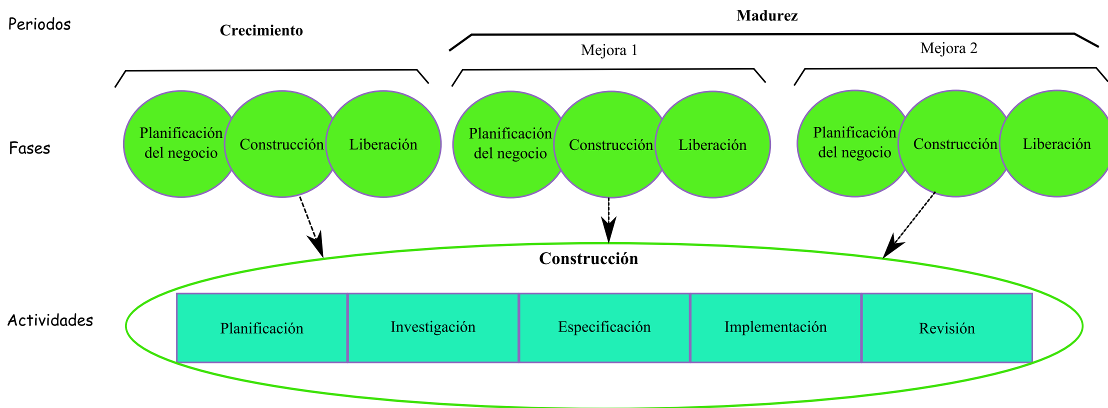


Figura 1.8: Ciclo de vida fuente

Cada clase puede tener un ciclo de vida sólo para ella debido a que cada una puede estar en una fase diferente en un momento cualquiera. La ventaja es que permite un desarrollo solapado e iterativo. En la figura 1.8 se muestra un esquema de este tipo de ciclo de vida.

1.3. Notaciones de especificación y diseño (UML)

Una parte esencial de todas las tareas del desarrollo del software, en particular de la especificación de requisitos y del diseño, es la utilización de una notación para la descripción de modelos que permita la mecanización parcial del proceso de producción. Dado que en la actualidad la fase de implementación se suele realizar con tecnologías orientadas a objetos, y que adicionalmente este tipo de enfoque también es aplicable en otras fases del desarrollo, es importante que el alumno conozca al menos los principios básicos de las notaciones orientadas a objetos, y en especial de la más extendida últimamente, la notación UML (*Unified Modelling Language*, Lenguaje Unificado de Modelado).

1.3.1. Introducción

Modelos

La construcción de todo sistema de ingeniería requiere un estudio previo que consiste en la identificación de sus componentes y de las relaciones entre éstas, la comprensión de sus propiedades, y la concepción de su comportamiento dinámico. El resultado de este estudio es un **modelo** o representación del sistema. Un modelo es una representación de la realidad donde se abstrae lo no esencial. La abstracción permite gestionar la complejidad y experimentar con diferentes soluciones. Para un mismo sistema se puede definir más de un modelo diferente en función del aspecto que interese destacar o el nivel de detalle deseado. El modelado permite pues ver un sistema desde diferentes perspectivas, haciendo así más sencillo su entendimiento y desarrollo. Finalmente, los modelos juegan un papel esencial en la comunicación con el cliente y entre los profesionales que forman parte del equipo de desarrollo.

“UML”, siglas de **Unified Modeling Language** es un lenguaje concebido, como su nombre indica, para expresar modelos. No es nada más que eso, no indica cómo se deben hacer el desarrollo y ni siquiera el análisis orientados a objetos y, en consecuencia, no es una metodología de desarrollo. Es importante tener bien claro que tan sólo se trata de una notación; ahora bien, es la notación que se ha convertido en el estándar *de facto* de la mayor parte de las metodologías de desarrollo orientado a objetos que existen hoy en día.

El UML se utiliza para la especificación, visualización, construcción y documentación de sistemas software. De esta frase, la palabra *visualización* es la más importante; UML es un lenguaje gráfico, esto es, basado en diagramas, y está pensado para ”entrar por los ojos”, tanto de los desarrolladores como de los clientes. Se buscó en él un lenguaje fácil de aprender pero rico en significado, estándar, estable, configurable e independiente de lenguajes de programación o procesos de desarrollo particulares. Al mismo tiempo se pretendió que soportara conceptos de desarrollo de alto nivel tales como colaboraciones, armazones (*frameworks*), patrones y componentes, y permitiera abordar aspectos clave del desarrollo de software actual tales como escalabilidad, concurrencia, distribución, ejecutabilidad, etc.

El UML soporta todo el ciclo de vida del desarrollo de software y en distintas áreas de aplicación (sistemas distribuidos, tiempo real, aplicaciones monoproceso, sistemas de información corporativos, banca/finanzas, telecomunicaciones, defensa/espacio, electromedicina, ciencia, ...). Diferentes herramientas CASE orientadas a objeto que dan soporte al desarrollo basado en UML (Rational Rose, Together, Objecteering, Paradigm Plus, ...) proliferan en el mercado del software.

Este capítulo no pretende ser exhaustivo, debe entenderse más bien como una introducción inicial abreviada a este lenguaje de modelado.

Para ver una mínima introducción a conceptos de orientación a objetos puede ser interesante leer primero la sección **??** de esta guía.

Historia

Grady Booch, Jim Rumbaugh e Ivar Jacobson (los apodados "tres amigos") desarrollaron los tres métodos de desarrollo orientado a objetos más seguidas de la industria. De la unificación del OMT (*Object Modeling Technique*) de Rumbaugh y el método de Booch nació la primera versión del UML en el año 1994. Posteriormente, se incorporaron el método OOSE (*Object-Oriented Software Engineering*) de Jacobson y algunos conceptos de otros lenguajes de modelado. En la actualidad el UML es un estándar abierto del grupo OMG (*Object Management Group*) y se halla en su versión 2.0.

Elementos del lenguaje UML

A continuación se presenta una clasificación general de los elementos del lenguaje UML. En las secciones siguientes se explica cómo estos elementos se combinan para construir diferentes tipos de modelos, y se definen algunos elementos adicionales que intervienen específicamente en estos modelos.

En el lenguaje UML se distinguen básicamente tres tipos de elementos:

1. **Bloques de construcción**: Pueden ser **Entidades**, **Relaciones** o **Diagramas**.
2. **Mecanismos comunes**: Entre ellos se distinguen **Especificaciones**, **Adornos** y **Mecanismos de extensión**.
3. **Reglas de combinación**

Los bloques de construcción se definen como abstracciones y tienen manifestaciones concretas denominadas **Instancias**. Las instancias más comunes son las instancias de clase (**Objetos**) y las instancias de asociación (**Enlaces**). Las instancias pueden identificarse mediante un nombre o bien ser anónimas.

En cuanto a las **Entidades**, pueden ser:

- **Estructurales**: Entre ellas se distinguen **Casos de uso**, **Clases**, **Componentes** y **Nodos**. Un tipo particular de clases son las **Interfaces**.
- **De agrupamiento**: Son los **Paquetes**. Tipos particulares de paquetes son los **Modelos** y los **Subsistemas**. Entre los modelos se distinguen **Modelos de casos de uso**, **Modelos estructurales estáticos**, **Modelos de comportamiento** y **Modelos estructurales de implementación**.
- **De comportamiento**: Son, principalmente, las **Acciones**, los **Estados**, las **Transiciones** y los **Mensajes**.

Las **Relaciones**, se clasifican principalmente en:

- **Dependencias**
- **Asociaciones**
- **Generalizaciones**

Se distingue entre **Diagramas** de:

- **Casos de uso**
- **Clases**
- **Objetos**
- **Secuencia**
- **Colaboración**
- **Estados**
- **Actividades**
- **Componentes**
- **Despliegue**

Como veremos en las secciones siguientes, los modelos de casos de uso se describen mediante diagramas de casos de uso y diagramas de secuencia; Los modelos estructurales estáticos mediante diagramas de clases y de objetos, los modelos de comportamiento mediante diagramas de secuencia, diagramas de colaboración,diagramas de estados y diagramas de actividades; y los modelos estructurales de implementación mediante diagramas de componentes y diagramas de despliegue.

En cuanto a los **Mecanismos comunes** se clasifican en:

- **Especificaciones**: Descripciones textuales detalladas de la sintaxis y semántica de un bloque de construcción.
- **Adornos**: Elementos gráficos y textuales que pueden añadirse a la notación gráfica básica para proporcionar detalles adicionales de la especificación (símbolos de visibilidad, compartimentos adicionales, notas, roles, representación de clases y características especiales - abstracta, raíz, hoja - etc).
- **Mecanismos de extensión**: Posibilidades de extensión controlada del lenguaje para adecuarse a aplicaciones, procesos o dominios de aplicación concretos. Se distingue entre **Estereotipos**, **Valores etiquetados** y **Restricciones**.

En las secciones siguientes explicamos los diferentes tipos de modelos que pueden definirse en UML, describiendo al mismo tiempo los elementos que intervienen en los correspondientes diagramas. Hemos optado por describir estos elementos a medida que se hace referencia a ellos por entender que en un contexto de modelado concreto se pueden entender mejor su utilidad y significado. Por lo general no distinguiremos entre los diferentes tipos de adornos o mecanismos de extensión que estos elementos constituyen, por no considerarlo relevante en el nivel de estudio del lenguaje exigido para esta asignatura. Los símbolos gráficos que los denotan aparecen ilustrados en las figuras de esta sección. Las antes mencionadas reglas de combinación están implícitas en la descripción de los diferentes diagramas.

1.3.2. Modelo de casos de uso

Representa la visión que tendría del sistema un usuario externo, y es desarrollado por los analistas con la colaboración de los expertos del dominio de aplicación. Un caso de uso expresa lo que comúnmente se denomina una transacción, esto es, una interacción entre el sistema y el usuario u otro sistema que es visto como un ”actor” externo: p.e., un usuario que pulsa el botón de llamada de un ascensor, o un periférico que solicita datos de un ordenador central.

Los casos de uso sirven para especificar los requisitos funcionales del sistema e identificar los escenarios de prueba. Normalmente se utilizan en las primeras etapas de desarrollo. En particular, algunas metodologías están dirigidas por casos de uso, de forma que es a partir de ellos que se derivan los modelos estructurales y de comportamiento. En cualquier caso, siempre es preciso asegurarse de que el modelo de casos de uso es coherente con estos modelos.

En UML los casos de uso se definen mediante diagramas de casos de uso y de secuencia, y descripciones textuales (especificaciones) que resultan de cumplimentar unas plantillas estándar.

Entidades

En los casos de uso intervienen dos entidades estructurales:

- **Casos de uso**: Consisten en una secuencia de **Acciones** (unidades básicas de comportamiento; habitualmente implementadas por una sentencia de ejecución), y posibles variantes de estas acciones, que puede realizar el sistema al interaccionar con los actores.
- **Actores**: Definen un conjunto de roles que desempeñan los usuarios cuando interaccionan con los casos de uso.

Interviene también en los casos de uso una entidad de agrupamiento, el **Límite del sistema**, que representa el límite entre el sistema físico y los actores que con él interaccionan.

Relaciones

En los modelos de casos de uso pueden estar implicadas relaciones de los tres tipos principales:

- **Asociación**: Indica la participación de un actor en un caso de uso (véase la figura 1.9), esto es, la comunicación de una ”instancia” de un actor con ”instancias” del caso de uso o de otro actor que interviene en el caso de uso.
- **Generalización**: Muestra una taxonomía entre casos de uso o actores generales, y casos de uso o actores más específicos.
- **Dependencia**: Puede indicar que el comportamiento de un cierto caso de uso extiende el de un caso de uso base, cualificándose con un estereotipo «extend», o bien que el comportamiento de un caso de uso incluye el de un caso de uso base, cualificándose con un estereotipo «include» (véase la figura 1.10). Cuando un caso de uso extiende a otro, añade algunos pasos a la secuencia de interacciones por él especificada. Cuando un caso de uso incluye a otro, significa que contiene la secuencia de interacciones por él especificada. Este último concepto se identifica con el de llamada a subrutinas: Cuando dos o más casos de uso tienen una parte significativa en común, es útil definir casos de uso elementales que representan las secuencias repetidas.

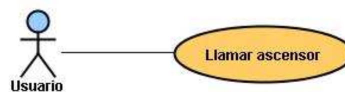


Figura 1.9: Asociación entre un caso de uso y un actor

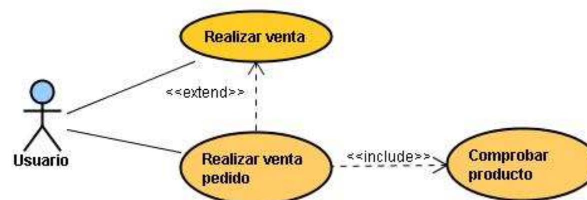


Figura 1.10: Relaciones entre casos de uso

Diagramas

En la figura 1.9 se ilustra un diagrama de casos de uso elemental. En cuanto a los diagramas de secuencia, que completan los modelos de casos de uso, se verán en detalle en la sección posterior ”Modelado de comportamiento”.

Mecanismos comunes

Como especificaciones de los casos de uso se utilizan las denominadas **Descripciones de casos de uso**, obtenidas al cumplimentar una plantilla donde se detallan los actores implicados, las precondiciones y postcondiciones, la secuencia normal de acciones emprendidas por los actores y respuestas asociadas del sistema, y los posibles escenarios alternativos y excepcionales. La información contenida en estas especificaciones tiene una traducción diagramática en los diagramas de casos de uso y diagramas de secuencia.

Modelado de casos de uso

Al modelar casos de uso es importante asegurarse de que cada caso definido describe una parte significativa del funcionamiento del sistema. Para evitar la definición de un número excesivo de casos de uso, hay que tener presente que un caso de uso no es un paso, operación o actividad individual de un proceso, sino un proceso completo que incluye varios pasos.

Otro aspecto importante a tener en cuenta es la utilidad de los casos de uso en la comunicación entre analistas, desarrolladores del software y expertos del dominio de aplicación: deben se entendibles por todos y constituir una descripción de alto nivel del sistema que no incorpore conceptos de diseño.

Para identificar casos de uso es recomendado comenzar por modelar el contexto con que habrá de relacionarse, fase que implica los siguientes pasos:

1. Identificación de los actores que interactúan con el sistema.
2. Organización de estos actores.
3. Especificación de la correspondientes vías de comunicación.

A continuación, para cada uno de estos actores, habrán de identificarse los procesos que inician o en los que participan. Los casos de uso resultantes deben constituir un modelo de requisitos del sistema.

Los siguientes pasos a seguir en la definición de casos de uso serán pues:

1. Identificación de las necesidades de los actores.
2. Designación de estas necesidades como casos de uso.
3. Identificación de los casos de uso que pueden ser especializaciones de otros y búsqueda de secuencias parciales comunes en los casos de uso ya encontrados.

Un enfoque alternativo consiste en prestar inicialmente atención a los **Eventos** relevantes que condicionan respuestas del sistema, y buscar después qué actores y casos de uso están relacionados con estos eventos.

1.3.3. Modelo estructural estático

Muestra una visión de la estructura interna del sistema en términos de las entidades que lo integran y las relaciones que existen entre ellas, capturando el vocabulario del dominio de aplicación. Este modelo es desarrollado por analistas, diseñadores y programadores y se define mediante diagramas de clases y de objetos.

Entidades estructurales

Las entidades estructurales implicadas en un modelo estructural estático son de tipo:

- **Clase** : Descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica.
- **Interfaz**: Nombre asignado a un conjunto de operaciones que caracterizan el comportamiento de un elemento.Puede verse como un tipo particular de clase.

Clases Las clases son abstracciones del dominio de aplicación o bien de la tecnología utilizada en su desarrollo. Constituyen el elemento básico de modelado en el paradigma de orientación a objetos. Una clase se instancia en objetos excepto cuando se define como **Clase abstracta** (un adorno de clase).

Las clases abstractas se utilizan en niveles altos de las jerarquías de clases para definir abstracciones muy generales de las que derivar otras clases. En una jerarquía de clases existe una clase **Raíz** o base sin **Superclases**, que da origen a la jerarquía, y clases **Hojas** sin **Subclases**. El significado de estas jerarquías está ligado al concepto de **Herencia**, que se presentará en una sección posterior.

En la figura 1.11 se ilustran las partes de una clase, que se disponen de arriba a abajo en el orden siguiente:

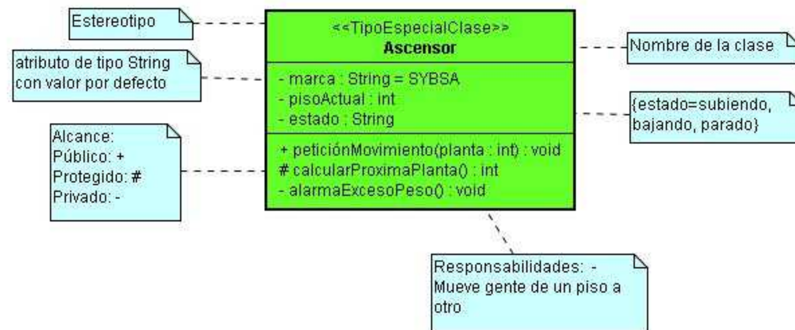


Figura 1.11: Plantilla para una clase en UML

1. **Nombre**: Distingue a la clase del resto de clases en el ámbito del modelo. Consiste en una cadena de cualquier longitud que puede contener letras, números y signos de puntuación (exceptuando " " y "::"). Aparece en *cursiva* en el caso de las clases abstractas (clases que carecen de instancias). El nombre de clase puede aparecer opcionalmente precedido del nombre del paquete en que la clase se utiliza; su sintaxis se define pues, en notación BNF: `[paquete::]nombre-simple`.
2. **Atributos**: Representan propiedades características compartidas por todos los objetos de la clase. Una clase puede tener cualquier número de atributos. Un nombre de atributo consiste en una cadena de cualquier longitud que puede contener letras y números. Sobre ellos pueden proporcionarse diferentes informaciones:

a) **Tipo**: `Atributo:Tipo`

b) **Valor inicial**: `Atributo:Tipo=Valor` ó `Atributo=Valor`

c) **Restricciones**: Condiciones semánticas que pueden expresarse utilizando cualquier lenguaje, incluido el lenguaje natural, si bien UML tiene un lenguaje semi-formal asociado para expresar restricciones: el lenguaje OCL. Se encierran siempre entre corchetes.

d) **Visibilidad** o **Alcance**: Determina en qué medida otras clases pueden hacer uso del atributo. Un atributo público (+) es visible para cualquier clase externa; un atributo protegido (#) es visible para cualquier descendiente en la jerarquía de herencia; y un atributo privado (-) es sólo visible para la propia clase. El valor por defecto de la visibilidad es siempre público.

e) **Ámbito**: Puede ser de dos tipos:

- **de instancia**: cuando cada objeto tiene su propio valor,
- **de clase**: cuando todas las instancias de una clase comparten un valor. (p.e., las variables *static* en Java). Este tipo de ámbito se indica subrayando el nombre del atributo.

Toda esta información es opcional, de hecho, es común que en las especificaciones de las clases no se detallen más que los nombres de sus atributos. Su sintaxis se resumen en: `[visibilidad]nombre [:tipo][=valorInicial][restricciones]`. **Atributos derivados** son aquellos cuyo valor puede computarse a partir de valores de otros atributos; aunque no añaden información semántica al modelo, se hacen explícitos por claridad o propósitos de diseño.

3. **Operaciones**: Son servicios básicos ofrecidos por los objetos de una clase, que con frecuencia provocan cambios en sus estados (se adornan con **query** aquellas operaciones que no tienen ningún efecto secundario, esto es, que no modifican el estado del sistema). Al igual que en el caso de los atributos, pueden especificarse su visibilidad, ámbito y propiedades. Se pueden indicar también el tipo de los argumentos que reciben y el tipo del valor que retornan. Tanto si se especifican sus argumentos como si no, han de añadirse un paréntesis abierto y otro cerrado al final de su nombre, de acuerdo a la sintaxis: `[visibilidad]nombre([parámetros])[:tipoRetorno][propiedades]`. La sintaxis completa de los parámetros es: `[dirección]nombre:tipo[= valorPorDefecto]`. La implementación concreta que una clase hace de una operación se denomina **Método**. La descripción de un método puede realizarse mediante una especificación textual descrita en cualquier lenguaje elegido (p.e., el OCL). Al igual que existen clases abstractas existen **Operaciones abstractas**, esto es, operaciones que se definen exclusivamente como una signatura. Las clases que contienen este tipo de operaciones requieren subclases que proporcionen métodos para su implementación. Las operaciones de las clases hoja de una jerarquía de clases obligadamente habrán de tener un método asociado.
4. **Responsabilidades**: Son descripciones de lo que hace la clase en su conjunto. Esta información casi nunca se incluye en los diagramas.

5. **Otras**: Una clase puede incluir partes adicionales predefinidas (p.e., **excepciones**) o definidas por el usuario.

La información que se ha incluido en la figura 1.11 es mucha más de la que se suele mostrar: una clase puede representarse con un simple rectángulo que contiene su nombre; véase la figura 1.12. También es común omitir parte de la información de un apartado. Los puntos suspensivos que aparecen el la figura 1.11 al final de las secciones de atributos y operaciones indican que la representación de la clase es **abreviada**, o lo que es lo mismo, que faltan atributos y operaciones que no se ha considerado necesario detallar para el propósito del diagrama en cuestión.

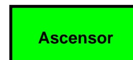


Figura 1.12: La representación de clase más sencilla posible

Las instancias de clase se denominan **objetos**. Los objetos se caracterizan por un estado definido por los valores específicos de sus atributos en cada instante de tiempo.

La notación gráfica de una clase en un diagrama de clases coincide con la de un objeto en un diagrama de objetos. La sintaxis de su nombre, que aparece subrayado, es: [nombreInstancia][:nombreClase] (véase la figura 1.13).

Figura 1.13: Símbolo en UML de un objeto

Interfaces Las interfaces son conjuntos de operaciones que especifican parte de la funcionalidad (un servicio) proporcionada por una clase o componente, con independencia de su implementación. Pueden verse como contratos entre los clientes de la interfaz y sus implementaciones; el programador responsable de la interfaz puede optar por implementar una nueva o bien adquirir o reutilizar otra implementación, con tal de que el estipulado contrato se cumpla.

Una interfaz y la clase que la implementa están relacionadas mediante una **realización** (un tipo particular de relación de dependencia), donde una clase puede realizar varias interfaces y una interfaz puede realizarse por más de una clase. Por otro lado, una clase ”cliente” puede usar varias interfaces con las que estará relacionada mediante una relación de **uso** (otro tipo de relación de dependencia).

Las interfaces establecen conexiones entre los componentes de una aplicación, y representan los principios de modularidad y ocultación necesarios en diseños orientados a objetos de calidad.

En cuanto a su representación diagramática, las interfaces se muestran como clases sin atributos mediante dos representaciones alternativas que se muestran en la figura 1.14: como clases con estereotipo *«interfaz»* y, en forma abreviada, sin mostrar las operaciones. Al tratarse de un tipo particular de clases, entre las interfaces pueden darse relaciones de **herencia**. Todas las operaciones de una interfaz son públicas.

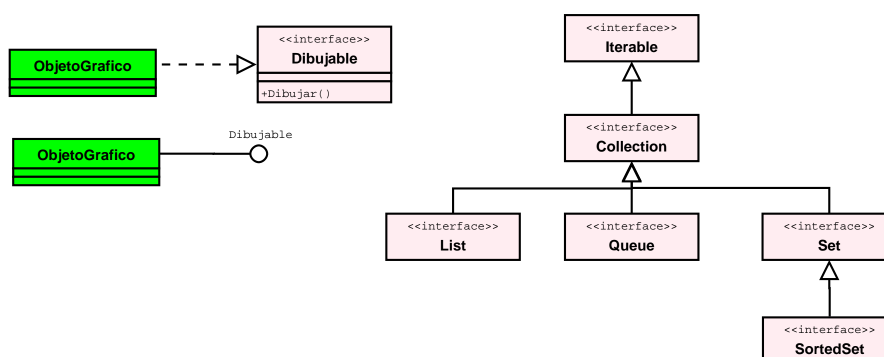


Figura 1.14: Interfaces

Relaciones

Asociación Es una relación conceptual, que no es inherente a la naturaleza de las clases sino al papel que juegan en el modelo. Especifica que las instancias de una clase están conectadas con las de otra. No se trata de una relación fuerte, es decir, el tiempo de vida de dos objetos relacionados es independiente (a menos que exista una restricción asociada a la relación que indique lo contrario).

La representación gráfica de una asociación (véase la figura 1.15) es básicamente una línea que conecta las clases, pero puede incluir opcionalmente un nombre que da idea de su interpretación en el dominio de la aplicación, así como otros adornos en sus extremos tales como:

- Un triángulo relleno que indica su dirección (cuando la asociación no es simétrica).
- Un nombre de ***rol*** que cualifica el papel que juega cada una de las partes en la asociación (una clase puede desempeñar distintos roles en distintas asociaciones).
- Una indicación de ***Navegabilidad***, propiedad del rol que consiste en la posibilidad de "ir" desde el objeto origen al objeto destino, es decir, implica la visibilidad de los atributos del objeto destino por parte del objeto origen. Por defecto la navegabilidad es bidireccional, En la figura 1.17 se ilustra una asociación no bidireccional: dada una *password*, no se puede acceder directamente al usuario a que pertenece (al menos en principio).
- Un valor de **multiplicidad** que especifica por cada clase de la asociación el número de objetos que se relacionan con un solo objeto de la clase asociada.
- Una ***Calificación***, atributo de la asociación que permite localizar una cierta instancia cuando la multiplicidad de la asociación es mayor que la unidad en el extremo considerado. (véase la figura 1.18).
- Un símbolo de ***Agregación***, que indica una asociación de tipo todo/parte y se representa con un símbolo de diamante (véase la figura 1.19). Sirve para modelar elementos que se relacionan utilizando expresiones como: *"es parte de", "tiene un", "consta de"*, etc. Una asociación de agregación fuerte, donde las partes (componentes) no pueden existir independientemente del todo que las integra (entidad compuesta), se denomina (***Composición***), y se denota mediante un diamante negro.

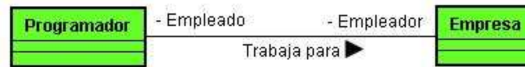


Figura 1.15: Ejemplo de una asociación

En una composición, cada componente pertenece a un todo y sólo a uno (en la figura 1.19 no se cumple esto, por ejemplo, la antigua Unión Soviética tenía una parte europea y una parte asiática). En la figura 1.20 vemos un ejemplo de este tipo de relación.

Es posible que dos clases estén relacionadas entre sí por más de una asociación o que una clase esté relacionada con muchas clases por diferentes asociaciones. Por lo general, las asociaciones son relaciones simétricas. Se dice que una relación es **reflexiva** cuando asocia entre sí a objetos de una misma clase. La notación con que se expresa la multiplicidad de una asociación es diversa:

1. **Números concretos**. Un número exacto: `A`. Por ejemplo: `5` (significa exactamente 5).
2. **Intervalos**. Entre A y B, ambos incluidos: `A..B`. Por ejemplo: `2..6` (es entre 2 y 6).
3. **Asterisco**. El símbolo `*` significa muchos. Si está en un intervalo se pone en el extremo superior. `A..*` se lee: A o más. Por ejemplo: `3..*` (es 3 o más). Si el `*` está solo significa que puede ser un número cualquiera, cero incluido.
4. Combinación de los elementos anteriores: Por ejemplo: `1..4,6,9..*` (que significa entre 1 y 4, ó 6, ó 9 ó más de 9, es decir, los valores 0, 5, 7 u 8 no estarían permitidos); `2,4,6` (los valores 2, 4 ó 6 y ninguno más).

Así, en el ejemplo representado en la figura 1.16, se hacen las siguientes suposiciones:

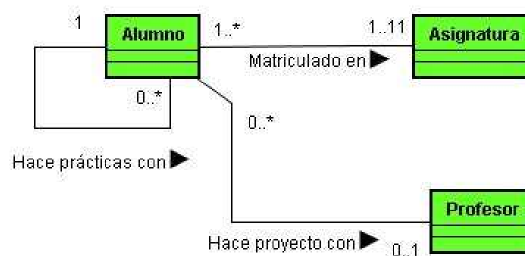


Figura 1.16: Relaciones entre alumnos, asignaturas y profesores

- Un alumno puede estar matriculado como mínimo en una asignatura (si no está en ninguna no es alumno) y como máximo en 11.
- En una asignatura se puede matricular un número ilimitado de alumnos pero tiene que haber al menos un matriculado, o de lo contrario la asignatura no existe.
- Un alumno puede estar haciendo o no el proyecto fin de carrera. En caso afirmativo tendrá un profesor asignado como coordinador.

- Existen asignaturas que tienen prácticas y algunas de ellas se realizan entre varios alumnos. Un alumno puede estar asociado con un número variable de compañeros para realizarlas.

Figura 1.17: Navegabilidad

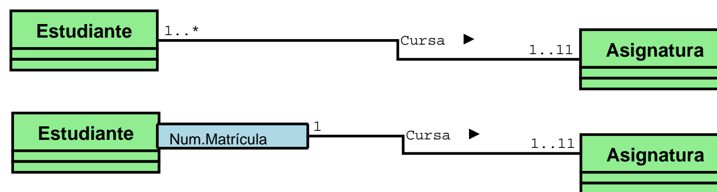


Figura 1.18: Calificación

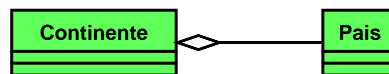


Figura 1.19: Agregación

Asociaciones **derivadas** son aquellas cuyos enlaces pueden deducirse de la existencia de otros enlaces.

Generalización Es una relación entre una abstracción general y una abstracción más concreta. Las relaciones de generalización entre clases (una superclase o clase principal y una subclase o clase secundaria) se denominan relaciones de **herencia**. Indica que la subclase "hereda" las operaciones y atributos definidos en la superclase. Este tipo de conexión se lee como *"es un tipo de"*. En la figura 1.21 se muestra un ejemplo de representación.

Una clase puede tener ninguna, una (**Herencia simple**) o varias (**Herencia múltiple**) superclases. La relación de generalización o herencia permite definir taxonomías de clases. En una jerarquía de herencia, las clases sin superclases se denominan clases **Raíz**, y las clases sin subclases se denominan clases **Hoja**. En las jerarquías de herencia se da el **Polimorfismo**, que consiste en que un objeto de una subclase puede comportarse como un objeto de cualquiera de sus superclases en cualquier contexto (el recíproco no es cierto). Así, si una operación de la subclase tiene la misma signatura que una operación de la superclase, cuando se invoque la operación de un objeto de la subclase se ejecutará el método definido en la subclase y, cuando se invoque la operación de un objeto de la superclase, se ejecutará el método de la operación de la superclase.

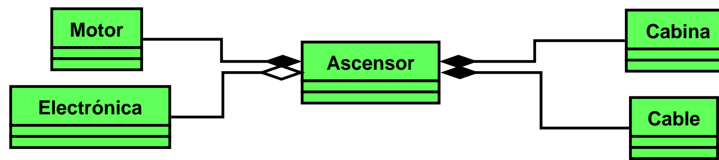


Figura 1.20: Composición con una agregación (la electrónica puede ser compartida por un grupo de ascensores)

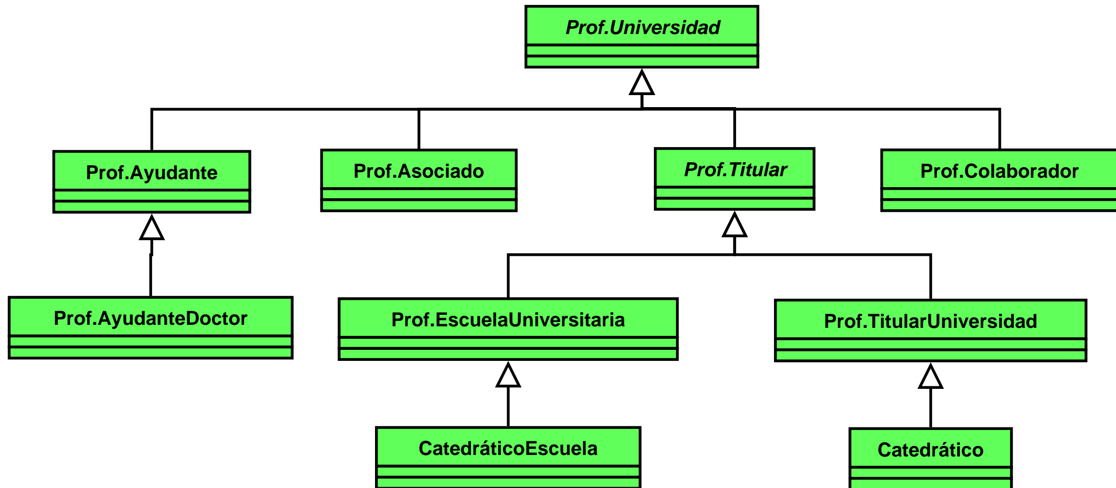


Figura 1.21: Herencia; los nombres de las clases abstractas se escriben en cursiva

Dependencia Es una relación entre dos entidades estructurales tal que un cambio en una de ellas (la independiente) puede afectar a la otra (la dependiente). Suele ser unidireccional y se interpreta muchas veces como una relación ”de uso”, entendiéndose que una clase ”usa” a otra cuando ésta define el tipo de alguno de los argumentos que aparece en las signaturas de sus operaciones (véase la figura 1.22).

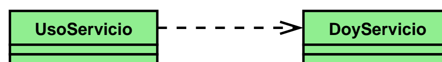


Figura 1.22: Dependencias

En UML se contemplan otros tipos de relaciones de dependencia, tales como **derivación**, **instanciación**, **refinamiento** y **realización**. Esta última es una relación semántica que se da entre clases, una de las cuales especifica un contrato que la otra se compromete a cumplir. Sirve habitualmente para separar la definición de un servicio o contrato y su implementación, y relaciona, típicamente, una interfaz con la clase o componente que la implementa.

Diagramas

Diagrama de clases Es el diagrama central de una especificación UML. Describe un modelo estático del sistema en términos de las entidades y relaciones descritas en las secciones anteriores (clases; interfaces; y relaciones de asociación, herencia y dependencia).

Veamos cómo sería el modelo de clases que corresponde con este texto: *En un conocido juego de estrategia hay tres tipos de razas: Terrans, Zergs y Protos. Cada tipo de raza define a su vez dos tipos de personajes: Trabajadores (consiguen recursos y construyen edificios) y Soldados (pegan a personajes de otro bando). Un personaje tiene como características: Si es aéreo o terrestre, salud (número entero), velocidad, escudo, etc. Un soldado además define si puede pegar a personajes aéreos o terrestres, daño que hace, etc. Un trabajador recoge recursos y los lleva a un edificio de recogida. Los recursos son de dos tipos: Gemas y Gas y se van agotando a medida que son explotados. Hay diferentes tipos de edificios propios de cada raza y cada uno proporciona capacidades diferentes a los miembros de su bando, por ejemplo, el máximo número de personajes de un bando está determinado por el número de edificios granja multiplicado por 8; el edificio armería mejora en +1 el daño de los soldados. Un bando está definido por un conjunto de personajes y edificios de una o más razas. El juego tiene lugar sobre un tablero definido en una retícula romboidal. En cada punto de la retícula se define el tipo de terreno y se pueden posicionar personajes, edificios y recursos. Un bando puede ser dirigido por un jugador o por la inteligencia artificial del programa.*

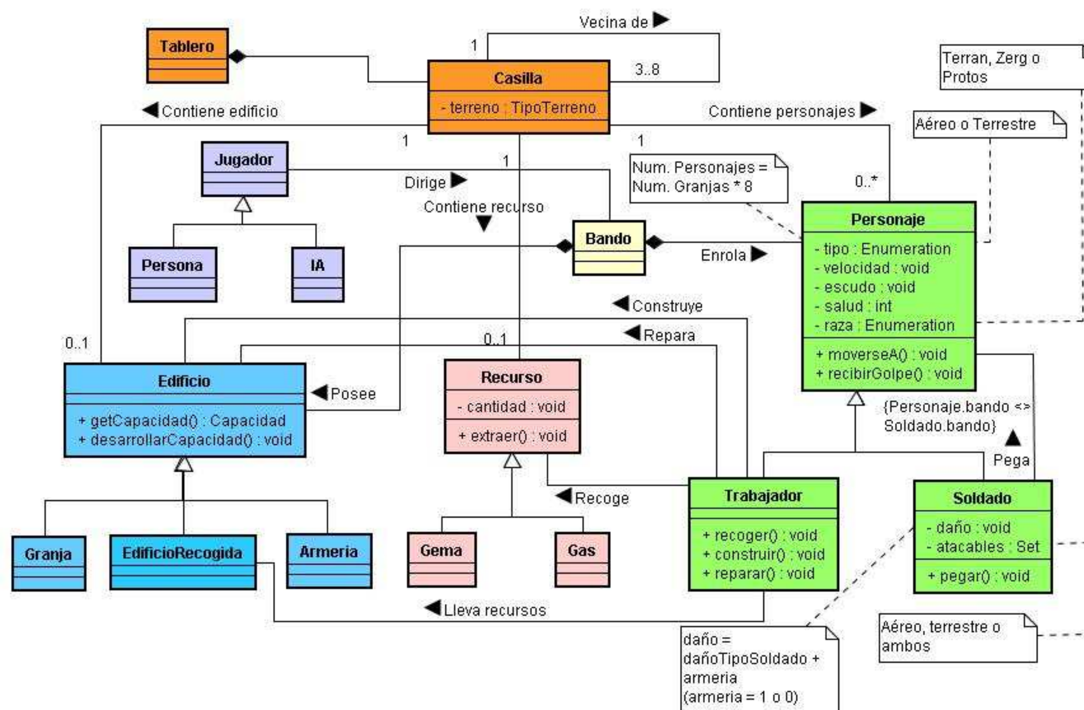


Figura 1.23: Diagrama de clases

Diagrama de objetos Este diagrama permite modelar una instantánea del sistema en un momento dado de su funcionamiento. Nótese que, no obstante, describe un modelo estructural estático, ya que no incluye información alguna sobre evolución temporal. Contiene principalmente objetos y enlaces que ligan estos objetos (esto es, instancias de clases y relaciones definidas en los diagramas de clases). Los objetos pueden aparecer agrupados en paquetes. En los diagramas de clases también es posible mostrar objetos que son instancias de las clases definidas, siempre que se considere relevante para el propósito de la representación; por esta razón, los diagramas de objetos pueden verse como diagramas de clases que no contienen clases. En el ejemplo de la figura 1.24, se ilustra la notación de un diagrama de objetos. Hacemos notar que:

1. Dar nombre a objetos y enlaces es opcional pero, en el caso de los objetos, es obligado indicar la clase de que son instancia.
2. También pueden mostrarse opcionalmente los atributos, con sus correspondientes valores, en caso de considerarse relevantes.
3. Al igual que las clases, los objetos pueden estereotiparse.

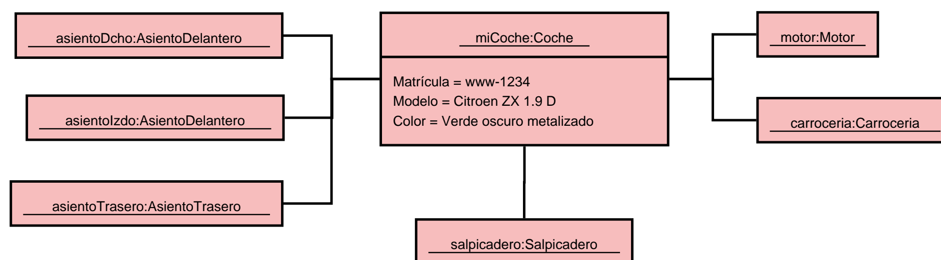


Figura 1.24: Diagrama de objetos

Mecanismos comunes

En los modelos estructurales estáticos suele utilizarse todo tipo de mecanismos comunes del UML. Algunos de ellos han sido mencionados en las secciones anteriores, donde se ha explicado cómo se añaden a las notaciones básicas de los bloques de construcción para enriquecer la representación (símbolos de visibilidad, estereotipos, etc) Definimos brevemente a continuación el sentido preciso de los dos más comúnmente utilizados:

- **Estereotipos**. Suponen una extensión del metamodelo del UML, esto es, del conjunto de conceptos (metaconceptos) predefinidos (tales como clase, atributo, caso de uso, etc.) en términos de los cuales se definen los modelos. Definir un nuevo estereotipo supone añadir un nuevo concepto por herencia, de modo que constituya un caso particular de algún concepto ya definido. Está aceptado asimismo definir nuevas notaciones gráficas para representar los nuevos conceptos (es el caso del símbolo de actor, que representa una clase estereotipada como *«actor»*.

- **Notas** Anotaciones escritas habitualmente de modo informal, a modo de texto explicativo ligado a algún elemento. Se representan en la forma indicada en la figura 1.25.

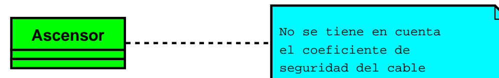


Figura 1.25: Nota relativa a una clase

Entidades de agrupamiento

En los modelos estructurales, los **Paquetes** agrupan clases vinculadas por alguna relación (su representación se ilustra en la figura 1.26). En cuanto a los **Subsistemas**, agrupan paquetes que participan en un comportamiento, desde una visión de alto nivel del sistema.

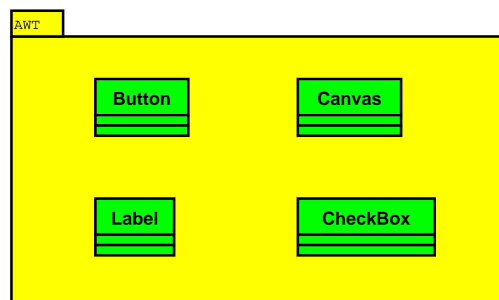


Figura 1.26: Paquete incluido en las librerías estándar de Java para la implementación de interfaces gráficos

Modelado estructural

Los modelos estructurales son útiles tanto cuando análisis y diseño se abordan con una perspectiva ascendente (”bottom-up”) como cuando se abordan con una perspectiva descendente (”top-down”). En el primer caso, el objetivo del modelado estructural será especificar la estructura de alto nivel del sistema, para lo que se utilizarán clases con significado arquitectural y entidades de agrupamiento para gestionar la complejidad (paquetes y subsistemas). En el segundo caso, se tratará de especificar la estructura de bajo nivel, que se irá refinando a medida que progrese el desarrollo (añadiendo detalles, reclasificando, definiendo nuevas relaciones, etc.).

Para comenzar un desarrollo especificando el modelo estructural estático es preciso conocer bien el dominio de aplicación. De otro modo, será más adecuado utilizar, p.e., un método dirigido por casos de uso, y el modelo estructural habrá de definirse en coherencia con los casos de uso o colaboraciones ya especificados.

Un buen modelo estructural debe constituir un esqueleto extensible y refinable a medida que se profundiza en el conocimiento del dominio de aplicación. En las primeras fases de definición

siempre es preferible limitarse a utilizar las notaciones más básicas, dejando las más avanzadas para añadir detalles en fases posteriores. En particular, es importante evitar limitar los modelos iniciales añadiendo aspectos propios de fases de implementación. También debe evitarse mezclar en un mismo diagrama entidades descritas en niveles de abstracción diferentes. Finalmente, constituye también una buena práctica organizar las clases en paquetes cuando empiezan a resultar numerosas.

1.3.4. Modelo de comportamiento

Un modelo de comportamiento describe la evolución del estado del sistema y las interacciones que tienen lugar entre sus elementos. Se expresa mediante diagramas de interacción (de dos tipos: diagramas de colaboración y diagramas de secuencias), diagramas de estado y diagramas de actividad. Entre las entidades de comportamiento, las principales entidades implicadas en los diagramas de interacción son los mensajes, mientras que en los diagramas de estado y actividad las entidades protagonistas son los estados, las transiciones y las acciones. Los cuatro tipos de diagramas de comportamiento pueden implicar asimismo diferentes entidades estructurales, y los diagramas de colaboración exhiben, adicionalmente, asociaciones.

Diagramas de interacción

Un **Mensaje** es la especificación de un conjunto de **Estímulos** comunicados entre instancias de clase, de componente o de caso de uso, junto con la especificación de las correspondientes clases, componentes o casos de uso emisores y receptores, del mensaje previo que los induce y de los subsecuentes procesos desencadenados (nuevos mensajes y acciones).

Los estímulos pueden ser **Señales**, entendidas como informaciones de datos o de control comunicadas entre instancias, o invocaciones de operaciones. Una **Interacción** es un conjunto de estímulos, intercambiados entre un grupo de instancias (de clases, componentes o casos de uso), en un contexto dado, y con un determinado propósito. Se entiende que dos instancias sólo podrán intercambiar estímulos cuando estén ligadas por algún enlace.

Un **Evento** es una ocurrencia significativa para una instancia; en el contexto de los diagramas de interacción son, principalmente, **Recepciones de estímulos** y **Satisfacción de condiciones**.

En un diagrama de interacción figuran:

- **Instancias** (de clases, de componentes o de casos de uso). Entidades con identidad única a las que se puede aplicar un conjunto de operaciones, que tienen un estado y almacenan los efectos de estas operaciones.
- **Acciones**. Existen acciones predefinidas (p.e.: CreateAction, CallAction, DestroyAction, UninterpretedAction...), y acciones definidas por el desarrollador.
- **Estímulos**. Las invocaciones de operaciones pueden ser de dos tipos
    - **Síncronas**: las instancias en comunicación tienen que estar sincronizadas, esto es, la instancia que invoca se queda bloqueada hasta recibir una respuesta.
    - **Asíncronas**: las instancias en comunicación no tienen que estar sincronizadas

Las señales siempre son asíncronas. Existen algunos tipos característicos de señales:

- **Retorno de invocación síncrona**.
- **Retorno de invocación asíncrona**.
- **Creación de instancia**.
- **Destrucción de instancia**.

- **Operaciones**. Servicios que pueden solicitarse de las instancias

Como ya hemos mencionado, existen dos tipos de diagramas de interacción: los diagramas de secuencias y los diagramas de colaboración. Además de los elementos antes listados, en los diagramas de colaboración pueden figurar explícitamente enlaces que conectan las diferentes instancias, y atributos de estos enlaces.

Diagrama de secuencias Muestra instancias y mensajes intercambiados entre ellas teniendo en cuenta la temporalidad con la que ocurren (lo que comúnmente se denomina escenario en el contexto del software de comunicaciones). Las instancias pueden ser de clases y también de componentes y de casos de uso. Los tres tipos de instancias se ven como objetos únicos, a efectos de estos modelos. Sirven para documentar el diseño, y especificar y validar los casos de uso. Gracias a estos diagramas se pueden identificar los cuellos de botella del sistema, reflexionando sobre los tiempos que consumen los métodos invocados.

En un diagrama de secuencias figuran, además de los elementos básicos de un diagrama de interacción antes presentados:

1. **Línea de vida de un objeto**: Es una representación de la actividad del objeto durante el tiempo en que se desarrolla el escenario. Se entiende que la dimensión temporal crece de arriba a abajo de la línea. Se representa con una cabecera en forma de rectángulo, que incluye el nombre del objeto y una línea vertical de puntos por debajo. Durante el tiempo en que la instancia está ejecutando un método, la línea de puntos se convierte en un rectángulo.
2. **Activación**: Punto en que un objeto pasa a tener un método en ejecución, bien por autoinvocación de la correspondiente operación, bien por invocación procedente de otro objeto.
3. **Tiempo de transición**: Es el tiempo que transcurre entre dos mensajes.
4. **Condicional**: Representa alternativas de ejecución o hilos de ejecución (procesos ligeros) paralelos. Indica que un mensaje sólo se envía si una condición se satisface. La condición se escribe entre corchetes y puede referenciar a otro objeto (ver figura 1.27).

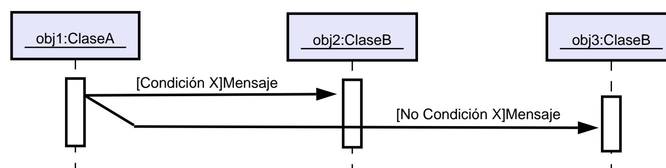


Figura 1.27: Diagrama de secuencias; condicionales

5. **Iteración**: Indica una acción que se repite mientras se satisfaga una cierta condición. Se denota con el símbolo * previo a la condición (ver figura 1.28).

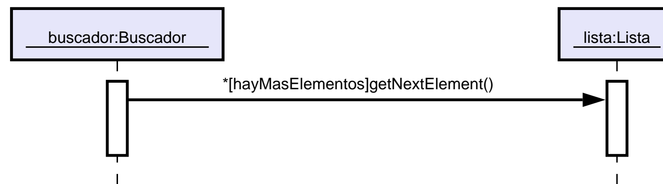


Figura 1.28: Diagrama de secuencias; iteración

6. **Invocación recursiva**: Se representa superponiendo un rectángulo al que representa al método desde el que se activó (ver figura 1.29).

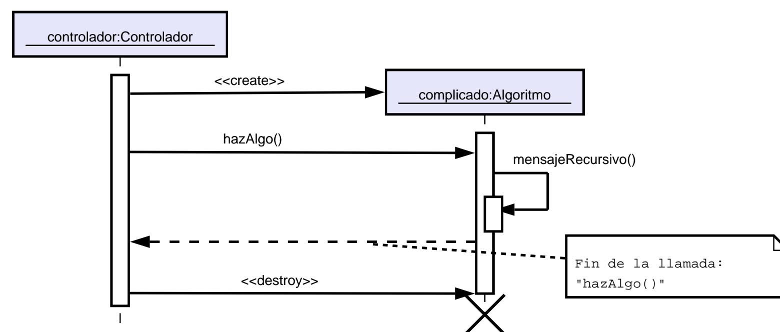


Figura 1.29: Creación de objeto, invocación recursiva y destrucción de objeto

Nótese que, en los diagramas de secuencias, los estímulos se denotan mediante flechas horizontales trazadas desde la línea de vida del objeto que los envía hasta la del objeto que los recibe. En cuanto a las señales de creación y destrucción de objetos, la creación se representa mediante un mensaje de creación que termina en la cabecera de una línea de vida (la de la nueva instancia creada), mientras que la destrucción se representa mediante un aspa que figura al final de la línea de vida del objeto destruido (véase la figura 1.29).

Diagrama de colaboración Muestran las interacciones que tiene lugar entre los objetos, organizadas en términos de los objetos y de los enlaces que los conectan (véase el ejemplo de la figura 1.30). Proporcionan una visión de la secuencia de interacciones alternativa de la proporcionada por los diagramas de secuencia. Para cada diagrama de colaboración existe un diagrama de secuencias equivalente (se puede generar de un modo automático un tipo de diagrama a partir del otro). Es un diagrama más compacto que el de secuencias pero muestra peor la evolución temporal.

Diagrama de estados

La evolución de un sistema a lo largo del tiempo puede modelarse como la transición entre una serie de estados de un autómata o máquina de estados. Los diagramas de estados del UML representan el comportamiento de las instancias de clases, componentes o casos de uso, como

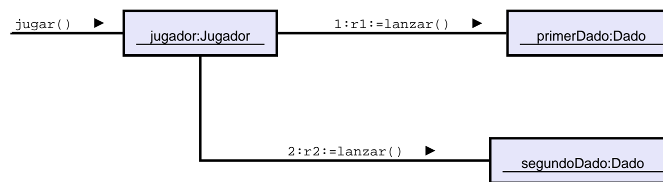


Figura 1.30: Diagrama de colaboración

máquinas de estados extendidas que constan siempre de un estado inicial y un estado final. A cada estado se asignan:

- **Nombre**.
- **Variables de estado**.
- **Acciones** que se realizan al entrar en el estado.
- **Acciones** que se realizan hasta el abandono del estado.
- **Eventos** que disparan transiciones hacia otros estados.
- **Transiciones**, asociadas a los respectivos conjuntos de eventos que las estimulan y, opcionalmente, a un conjunto de acciones, parámetros y condiciones que habilitan o no la transición.

Un estado puede, alternativamente, expandirse en una máquina de estados representable a su vez mediante un diagrama de estados. Las máquinas de estado representadas por los diagramas de estados soportan concurrencia: esta se representa mediante un estado compuesto que se expande en un diagrama con más de un estado de inicio y un único estado final.

El estado de un sistema en un instante dado resume la historia de eventos pasados. Por ejemplo, imaginemos un ascensor que en cuanto llega al sótano (área restringida a la que sólo se puede acceder con llave) sube a los pocos segundos por sí mismo para evitar que posibles intrusos puedan acceder a las viviendas. Por otra parte el ascensor puede estar subiendo, bajando o parado. Estar en el sótano se considera otro estado diferente. El comportamiento del ascensor aparece representado mediante un diagrama de estados UML en la figura 1.31.

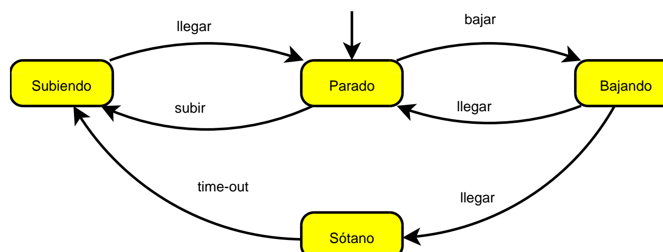


Figura 1.31: Diagrama de estados para un ascensor

Diagrama de actividades Representan asimismo un caso particular de máquinas de estado donde las transiciones entre estados no son motivadas por eventos externos, sino que tienen lugar espontáneamente cuando concluyen las actividades a ellos asociadas. Modelan el comportamiento del sistema haciendo énfasis en la participación de los objetos en ese comportamiento. Se centran en mostrar el flujo de control y datos entre objetos y su comportamiento interno (las acciones que conllevan sus operaciones y cómo se ven afectados por ellas), en lugar de mostrar su respuesta a eventos. Las máquinas de estado representadas por los diagramas de actividades soportan también concurrencia.

Para diseñarlos se sugiere seguir la siguiente estrategia:

1. Identificar una clase que participa en el comportamiento que se desea modelar, cuyas actividades de proceso deban ser especificadas.
2. Identificar las distintas actividades cuyo modelado es de interés.
3. Identificar estados, flujos y objetos.

Este tipo de diagrama es útil para modelar el flujo de trabajo de un negocio a alto nivel (véase el ejemplo de la figura 1.32).

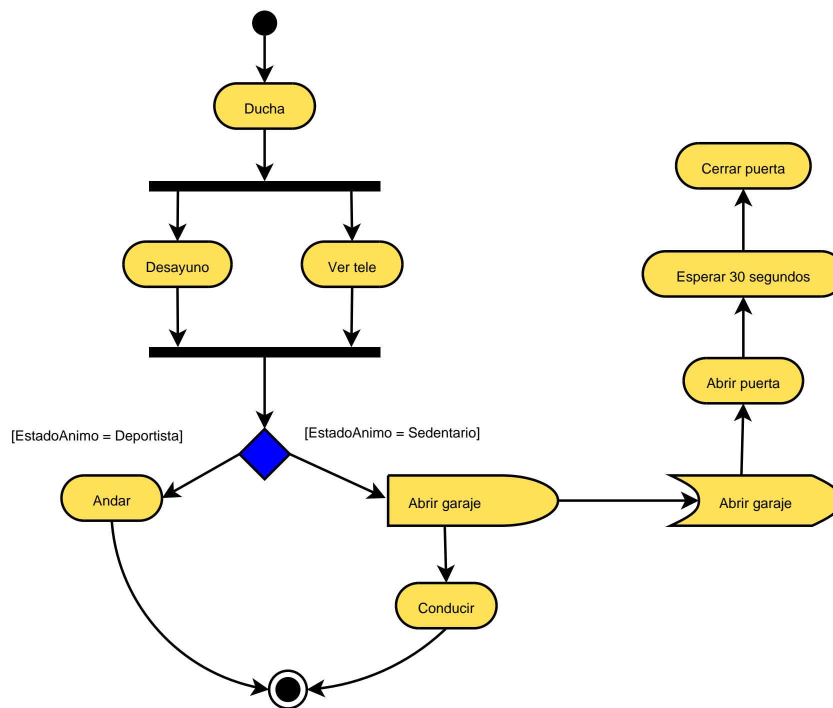


Figura 1.32: Diagrama de actividades; concurrencia, bifurcación e indicación

Los diagramas de actividades constan de los siguientes elementos:

1. **Punto inicial**. Representa el estado inicial. Se denota mediante un círculo negro.

2. **Punto final**. Representa el estado diana. Se denota mediante un círculo negro rodeado exteriormente por otro círculo.
3. **Actividades**. Estados de actividad donde se ejecutan conjuntos de acciones que pueden describir parcial o totalmente una operación de clase o acciones de usuario. Un estado de actividad puede expandirse en un nuevo diagrama de actividades que suponen una descomposición funcional de la actividad principal. Se simbolizan mediante rectángulos con bordes redondeados.
4. **Transiciones**. Suponen el paso de una actividad a otra. Se representan mediante flechas.
5. **Actividades concurrentes**. Se representan mediante sus correspondientes actividades y una barra negra de la que parten las transiciones que las inician.
6. **Bifurcaciones**: Se representan mediante un rombo o sencillamente mediante dos flechas que parten de una actividad hacia otras dos. En cualquier caso, la condición se expresa en cada uno de los ramales entre corchetes.
7. **Indicaciones**: Representan eventos que pueden enviarse o recibirse. El envío se denota mediante un rectángulo acabado en flecha, y la recepción mediante la figura complementaria.

En los diagramas de actividades el flujo de control y el flujo de objetos no aparecen separados: ambos se modelan como transiciones de estados. Es aconsejable definir diagramas bien anidados y asegurar que habrá una máquina de estados correspondiente a cada actividad ejecutable, así como evitar el ”go to” en la medida de lo posible.

Modelado del comportamiento

Los diagramas de interacción habrán de utilizarse principalmente cuando haya interés en conocer cómo se comportan las instancias implicadas en un caso de uso, en definitiva, cómo interaccionan unas con otras y cómo se reparten las responsabilidades (cómo atienden los requisitos). Sirven para identificar las interfaces de las clases. Un buen diagrama de interacción habrá de incluir indicaciones sobre el contexto de la interacción (mediante especificaciones textuales).

Los diagramas de secuencia permitirán ilustrar escenarios típicos, mostrando el orden explícito de los estímulos, por lo que resultarán muy adecuados para el modelado del tiempo real. El flujo deberá presentarse de izquierda a derecha y de arriba a abajo.

Los diagramas de colaboración muestran cómo se coordinan las entidades en la estructura del sistema y cómo el control pasa de unas instancias a otras. Al hacer explícitas las relaciones entre instancias, ayudan a realizar su agrupación en módulos. Permiten asimismo concentrarse en los efectos de la ejecución sobre las diferentes instancias.

Los diagramas de estado serán útiles para mostrar el comportamiento de una única instancia dentro de un caso de uso. Permiten capturar el comportamiento dinámico de sistemas de cierta complejidad, resultando particularmente adecuados para el modelado de sistemas reactivos orientados a eventos (p.e., interfaces de usuario, dispositivos hardware o protocolos de comunicación).

Los diagramas de actividad permitirán observar el comportamiento a través de varios casos de uso o muchos hilos de ejecución. Serán particularmente útiles en aplicaciones que requieran

modelos de flujo de control o de flujo de datos/objetos (p.e., modelos de procesos de negocios), en lugar de modelos dirigidos por eventos. En estos casos, el comportamiento no depende mucho de eventos externos, los pasos se ejecutarán de principio a fin sin ser interrumpidos, y entre pasos se producirán flujos de objetos/datos. Estos diagramas se construirán normalmente durante el análisis para estudiar qué actividades ocurren, en lugar de qué objetos son responsables de ellas (posteriormente se podrán usar **Particiones** para asignar esas responsabilidades).

1.3.5. Modelo estructural de implementación

El modelo estructural de implementación muestra los aspectos de implementación de un sistema: la estructura del código fuente y objeto, y la estructura de la implementación en ejecución. En UML se utilizan dos tipos de diagramas para expresar estos modelos: los diagramas de componentes y los diagramas de despliegue. Las principales entidades implicadas en los primeros son componentes y, en los segundos, componentes y nodos.

Un **Componente** se define como una parte modular, reemplazable y significativa del sistema que empaqueta una implementación y expone una interfaz. Un **Nodo** designa un objeto físico de ejecución que representa un recurso computacional.

Diagrama de componentes

Modelan una visión estática del sistema en el nivel de implementación. Ilustran la organización y las dependencias entre componentes software. Los componentes pueden ser código fuente (clases de implementación,una implementación de una interfaz, ficheros de scripts...) binarios, ejecutables, tablas, o cualquier otro tipo de elemento que forme parte de la implementación de sistema o de su documentación. En un diagrama de componentes no necesariamente estarán presentes todos los componentes. En la figura 1.33) se muestra cómo un componente implementa una interfaz.

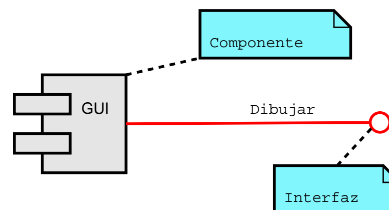


Figura 1.33: Diagrama de componentes

El estándar de UML define cinco estereotipos para caracterizar tipos de componentes: *executable*, *library*, *table*, *file* y *document*.

Ejecutables Estos componentes se modelan siguiendo los siguientes pasos (véase el ejemplo de la figura 1.34):

1. Se identifican los componentes y posibles particiones del sistema en susbsistemas, determinándose cuáles son susceptibles de ser reutilizados, se agrupan por nodos y se realiza

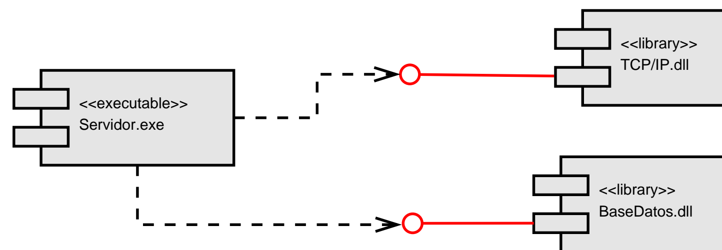


Figura 1.34: Ejecutable que usa una librería de comunicaciones y una base de datos

un diagrama por cada nodo que se desee modelar.

2. Se identifica cada componente con su correspondiente estereotipo.
3. Se relacionan los componentes entre sí.

Código fuente Modelar estos componentes es útil para expresar las dependencias que existen entre los módulos, con vistas a componer librerías o programas ejecutables (véase el ejemplo en la figura 1.35).

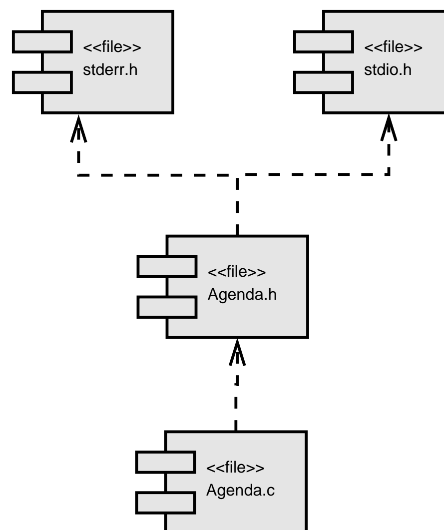


Figura 1.35: Dependencias entre el código

Diagrama de despliegue

Este diagrama refleja la organización del hardware. Su entidad central es el nodo. Los nodos pueden o no tener capacidad para ejecutar componentes. Un nodo lleva asociado un nombre y un conjunto de componentes (que pueden representarse sólo mediante sus nombres o bien mediante un diagrama de componentes). Cada nodo puede estar físicamente conectado a otros nodos. La

utilidad principal de este tipo de diagramas es la modelización de redes (véase el ejemplo de la figura 1.36).

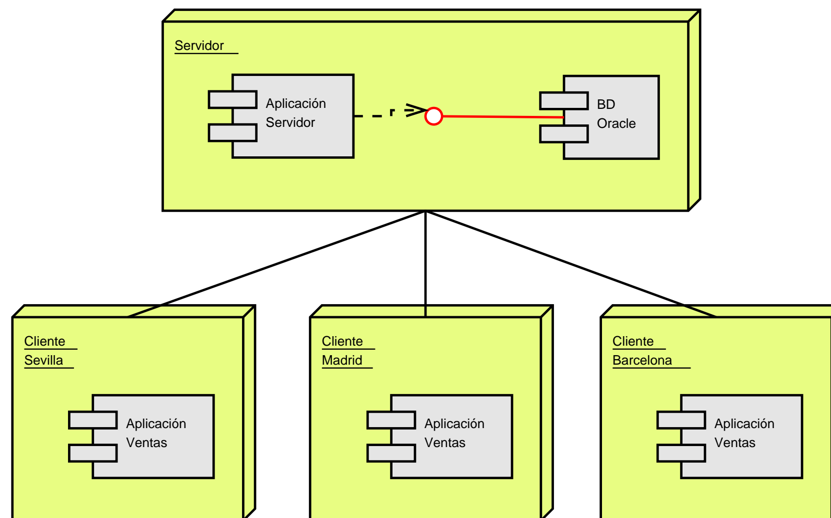


Figura 1.36: Diagrama de distribución

Tutorial posterior

Resumen de contenidos

En este capítulo introductorio se abordan la división en fases de un proyecto (ciclos de vida), las características de las metodologías que ordenan estas fases, y la notación de modelado actualmente más difundida (UML).

1. **Metodologías**: Se plantea el porqué de la ingeniería del software. Se define lo que se entiende por sistema y por metodología (una forma podría decirse "burocrática" de producir un sistema con costes y tiempos controlados). Se establece también una clasificación de las metodologías que, al igual que se verá con los ciclos de vida, pertenecen a dos tipos principales: estructuradas y orientadas a objetos.
2. **Ciclos de vida**: Se explica lo que es un ciclo de vida, las fases de las que se compone y la forma en que se ordenan estas fases. Se presenta una clasificación de los distintos ciclos de vida que se pueden dar en un proyecto. Básicamente, se consideran dos tipos principales que son: el ciclo de vida en cascada (el más antiguo y preferido en las metodologías estructuradas) y el ciclo de vida en espiral (el más nuevo, preferido en las metodologías orientadas a objetos). Estos se dividen a su vez en varios subtipos que cumplen diferente propósitos. Los ciclos de vida son:

   a) *Ciclo de vida en cascada*. Subtipos: ciclo de vida en V, Sashimi, cascada con subproyectos, cascada incremental y cascada con reducción de riesgos.

b) *Ciclo de vida en espiral*. Subtipos: Fuente y además los subtipos cascada incremental y cascada con reducción de riesgos, que también se pueden considerar como ciclos de vida en espiral.

c) *Notaciones de especificación y diseño (UML)*: Se proporciona una introducción a la notación UML. E existen libros muy completos al respecto, aquí solo se presentan nociones básicas para componer un diagrama pequeño partiendo de un texto o para entender diagramas ya construidos. El motivo de explicarlo en este tema es su utilización a lo largo de todas las fases del ciclo de vida.

Ejercicios y actividades propuestos

1. ¿Qué ciclos de vida elegiría para cada una de estas aplicaciones?
   - Una biblioteca de cálculo numérico para hacer operaciones con matrices de gran tamaño.
   - Una agenda electrónica para guardar teléfonos, direcciones, etc. Es más que posible que se quiera ampliar.
2. ¿Cómo representaría la siguiente descripción en UML?:
   Una casa está en una ciudad, las casas pueden estar en un edificio y varios edificios forman una manzana. Hay casas que son chalés y los chalés pueden estar solos o adosados. Los chalés y los edificios están en una calle, dentro de la cual tienen un número asignado; por otra parte, las casas que forman parte de edificios tienen un número que indica el piso y una letra. Una casa pertenece a una calle. Una zona es un conjunto de calles y cada una es conocida por un código postal. Un conjunto de zonas es una ciudad, que tiene un nombre y pertenece a una provincia, la cual a su vez pertenece a un país, el cual está en un continente o como mucho en dos (p. ej Rusia). En una casa viven personas y el número de personas que vive en una casa puede ser cualquiera. Una persona puede vivir en más de una casa (por ejemplo, durante el verano en una y en el invierno en otra). Una casa es poseída por una o varias personas.
3. Represente la siguiente situación en UML:
   Una persona pertenece a tres posibles grupos: Estudiante, Trabajador o Jubilado. Un trabajador puede tener empleo o no y puede cambiar entre cualquiera de estas situaciones siendo despedido si está trabajando o siendo contratado si está parado. Cuando llega a los 65 años pasa a estar jubilado; a su vez, un estudiante pasa a ser un trabajador cuando acaba sus estudios. Un jubilado no cambia de estado (y si cambia, mal asunto).
4. En el ejercicio anterior con los tres grupos de personas, ¿Cómo representaría en un diagrama de clases a un estudiante que trabaja al mismo tiempo?
5. Represente la siguiente descripción en UML:
   Un paracaidista se tira desde un avión, momento a partir del cual está *cayendo deprisa*. Pueden pasar dos cosas: el paracaídas se abre o no se abre. Si no se abre existe un segundo paracaídas de emergencia que se despliega tirando de una anilla. Nuevamente pueden pasar dos cosas. Si todo sale bien, tanto con el primero como con el segundo se pasa al

estado de *cayendo despacio*. Se puede estar cayendo deprisa un máximo de 30 segundos y cayendo despacio un tiempo que oscila entre 0 y 5 minutos en función del momento en el que se abra uno de los paracaídas. Al final el paracaidista llega al *suelo*.

6. Represente esta situación en UML:
   Un objeto A (cliente) hace una llamada a un objeto B (interfaz) que se encarga de crear un objeto C (servidor) si no existe ninguna instancia de C. En caso contrario no hace nada, es decir, sólo puede existir una instancia de C. En cualquier caso devuelve a A una referencia al objeto C, entonces A invoca una función de C, tras lo cual A imprime un mensaje y se autodestruye.

Capítulo 2

Fase de especificación

2.1. Obtención de requisitos

Los métodos tradicionales en cualquier ingeniería requieren como primer paso la obtención de los requisitos en forma de especificaciones por parte del cliente. Este problema habitualmente tiene complicaciones debidas al paso entre el lenguaje natural y los lenguajes más formales en ingeniería. Por lo tanto la obtención de los requisitos es un paso complejo y que no tiene una solución sencilla. Se suelen utilizar los métodos de pregunta-respuesta o los de cuestionarios plantilla para perfilar la versión inicial, pero se requieren sucesivas iteraciones (incluso con otras fases de desarrollo) antes de obtener unas especificaciones adecuadas.

2.1.1. Introducción

Un **requisito** se define como una capacidad que el sistema debe tener porque el cliente lo ha pedido explícita o implícitamente. Lógicamente, la determinación del conjunto de requisitos es el primer paso que se debe dar en la construcción de una aplicación. Existen dos subtareas en la obtención de los requisitos:

- **Análisis**: Consiste en comprender el problema del cliente para conseguir una lista de las características que debe tener el producto.
- **Especificación**: Consiste en traducir los requisitos a un documento con un formato concreto que pueda servir de entrada a la fase de diseño.

La obtención de requisitos es difícil por varias razones:

- La naturaleza de los requisitos es cambiante.
- Surgen nuevos requisitos en cualquier momento.
- El cliente puede no tenerlos claros.
- Pueden existir malos entendidos debidos a:
    - Falta de conocimientos por parte del equipo desarrollador sobre el problema.

- Falta de conocimientos técnicos (informáticos) por parte del cliente para expresarse con claridad.

Análisis

Como se ha dicho es la captación de un listado de requisitos que se van a conseguir comunicándonos con el cliente. Para facilitar esta comunicación se han desarrollado varias técnicas: entrevistas, prototipos, desarrollo conjunto de aplicaciones (*Joint Application Development*, JAD), planificación conjunta de requisitos (*Joint Requirements Planning*, JRP) y casos de uso del UML.

Especificación

Después del análisis se redacta un documento técnico con todos los requisitos anteriores, este documento tiene que tener estas propiedades:

- **Completitud**: Están todos los requisitos.
- **Concisión**: Es importante no hacer una novela, hay que contar lo que hay pero pensando que quien se lea el documento cobra por horas.
- **Legibilidad**: Es similar al punto anterior, pero el sentido de este es la claridad.
- **Consistencia**: No existen contradicciones entre diferentes puntos del documento.
- **Facilidades de prueba**: De algún modo se debe poder comprobar cada uno de los requisitos.
- **Facilidades de cambio**: Es bastante probable que el documento cambie a lo largo del ciclo de vida (requisitos añadidos, eliminados o cambiados).
- **Facilidades de seguimiento**: Debe ser posible comprobar si se van cumpliendo los objetivos.
- **Factibilidad**: Los objetivos definidos deben ser alcanzables con el tiempo y presupuesto disponibles.

Tipos de requisitos

Requisitos funcionales: Dicen **qué** debe hacer el sistema, en el sentido de servicios proporcionados al usuario.

Requisitos no funcionales: Hablan de **características** del sistema, como pueda ser la fiabilidad, mantenibilidad, sistema operativo, plataforma hardware, lenguaje de programación, etc.

En el siguiente texto: *Se desea desarrollar un editor de texto para programadores que emplee menos de cinco megas de memoria y tarde menos de un segundo en cargarse. Debe reconocer y colorear adecuadamente el código fuente de programas escritos en C, C++, Java y Modula 2.*

Debe ser capaz de colorear asimismo texto escrito en LATEX. El editor deberá ser multiplataforma, o lo que es lo mismo, funcionará en Linux, las diferentes versiones de Windows y Mac. El lenguaje de programación es indiferente pero el editor no debe presentar problemas de "fuga de memoria". Debe ser capaz de invocar comandos para compilar y recoger en una ventana los errores y advertencias de dicha compilación. El código fuente se documentará con Doxygen.

Siguiendo la definición los requisitos funcionales son:

- Debe reconocer y colorear adecuadamente el código fuente de programas escritos en C, C++, Java y Modula 2.
- Debe ser capaz de colorear texto escrito en LATEX[1].
- Debe ser capaz de invocar comandos para compilar y recoger en una ventana los errores y advertencias de dicha compilación.

Y los no funcionales. Se pone en paréntesis el tipo de característica:

- Emplea menos de cinco megas de memoria. (eficiencia)
- Tarda menos de un segundo en cargarse. (eficiencia)
- Debe ser multiplataforma. (sistemas operativos)
- El editor no debe presentar problemas de "fuga de memoria" (corrección)
- Se documentará con Doxygen (método de desarrollo)

2.1.2. Técnicas de obtención de requisitos

En esta sección veremos las técnicas de comunicación con el cliente.

Entrevistas genéricas

Las entrevistas en términos generales generan estrés porque se ponen de manifiesto las carencias de formación, de conocimiento o realización del propio trabajo, problemas personales o interpersonales, pero sobre todo porque suele haber algo en juego.

Aparte del tema del estrés en toda reunión suele haber una relación de poder entre los integrantes. Las relaciones de poder son el mejor contaminante de la buena comunicación, ejemplos: *"si digo que nuestra parte del proyecto tiene tal problema delante del jefe de mi jefe directo, mi jefe directo me despide"*, el individuo X impone una alternativa peor que otras porque no las comprende o por interés personal: *"el servidor web de contenidos se programa en C desde cero (porque es el lenguaje que X conoce)"* o *"las compras de hardware se hacen siempre en la empresa Z (que es de mi hermana)"*, etc.

Debido a lo anterior, sobre éste tema de las entrevistas y reuniones se han vertido ríos de tinta. El resultado ha sido una "burocratización" del proceso que es necesaria por como funciona la psicología humana. Veremos un resumen de lo esencial adaptado a las necesidades de construir una especificación software.

[1] Sistema de proceso de textos de varias décadas de antigüedad muy usado por investigadores

Figura 2.1: Relaciones de poder en la empresa. El dibujo es del autor argentino Quino (Joaquín Salvador Lavado)

Aunque también existan otras técnicas, ésta es inevitable. La tendremos al menos al inicio del proyecto. Una entrevista genérica sigue este patrón: Preparación, Desarrollo y Análisis.

Fases de la entrevista genérica

- **Preparación**: es el trabajo previo. Tiene cuatro apartados: documentarse sobre el tema que se va a tratar, seleccionar a la gente que se va a entrevistar, determinar el objetivo de la entrevista y logística de la misma

  - **Documentación**: el entrevistador se informa acerca del tema a tratar. Puede hacerlo de varias formas:

    - Estudiar la *bibliografía* sobre el tema.
    - Estudiar *documentos* sobre proyectos similares.
    - *Inmersión* dentro de la organización para la que se desarrolla el proyecto

  - **Personal**: se seleccionan las personas a las que se va a entrevistar. Existen dos tipos de personas:

    - *Directivos*: dan una imagen de alto nivel de la empresa. Puede ser útil para determinar la estructura arquitectónica de la aplicación. Ejemplos (inspirados en la figura): política que se sigue para comprar escobas, departamento responsable de su compra, almacenaje y distribución, etc.

- ◦ *Plantilla de base*: dan una imagen de un grano más fino. Son los que pueden concretar las funciones a implementar. Ejemplos (inspirados en la figura): un requisito sería el desarrollo de una escoba especializada en barrer rincones, otra con mango más largo para llegar al techo, etc.

- Determinar el/los **objetivos** de la entrevista. Previamente a la entrevista se pueden distribuir a los entrevistados cuestionarios sobre el tema a tratar y una introducción.
- **Logística**. La tercera entrada del DRAE[2] define la logística como *Conjunto de medios y métodos necesarios para llevar a cabo la organización de una empresa, o de un servicio, especialmente de distribución*. En este contexto trata de los temas prácticos acerca de como discurre la entrevista: lugar, hora, minimizar interrupciones, encontrar un hueco común en la agenda de todos, etc.

- **Desarrollo**

  Hay tres etapas [PV96]:

  1. **Apertura**: el entrevistador se presenta e informa al entrevistado de cuales van a ser los puntos tratados en la entrevista.
  2. **Desarrollo**: el entrevistado debería hablar el 80 % del tiempo. No debe durar más de dos horas para no cansar al entrevistado, que empezará a cometer errores y a dar señales de *"estoy perdiendo el tiempo y tengo otras cosas que hacer"*. Consiste en realizar una serie de preguntas. Dichas preguntas se pueden clasificar en dos tipos:

     - *Preguntas abiertas*, también conocidas como de *contexto libre*. No se pueden contestar con "Si" o "No". Por ejemplo: ¿Cuál es la lista de pasos para dar de baja un producto?. Más tarde se pasa a preguntas más concretas.
     - *Preguntas cerradas*, tienen una respuesta simple y concreta, como afirmaciones o negaciones o de tipo numérico.

     Hay tres modelos

     - **Forma de expresarse**: se deben evitar los tecnicismos que el entrevistado pueda no conocer.
     - **Psicología**: el problema fundamental de las entrevistas es que se trata con personas en vez de con máquinas, por eso la información conseguida puede tener omisiones o imprecisiones. Hay que tener en cuenta las siguientes reglas entre muchas otras de la comunicación no verbal.
       - ◦ No insinuar que el entrevistado debería saber algo que no sabe para que no se ponga a la defensiva. También hay que dejar claro que los intereses del entrevistador son únicamente la adquisición de requisitos, no hacer un examen de conocimientos, y por tanto las lagunas que pueda tener no trascenderán a sus superiores.

[2] Diccionario de la Real Academia de la Lengua

  - Lenguaje del cuerpo: dicen los psicólogos que el 90 % de la comunicación es no verbal. Se debe estar atento a los signos que puedan denotar inseguridad en algunos temas para preguntar a otras personas.
  - Usar técnicas para mantener la atención del entrevistado, como por ejemplo evitar las circunstancias que sometan a la persona a interferencias o intentando estimular su creatividad.

3. **Terminación**: se hace un resumen de la información recogida (para validar que es correcta) y, de ser necesario, se cita para la siguiente entrevista. Uno de los problemas recurrentes en la especificación de requisitos es que suele ser difícil tener este tipo de sesiones. Se debe intentar que el entrevistado se sienta cómodo y con la sensación de haber aprovechado el tiempo para que podamos recurrir a él en otra ocasión. Por último se le agradece que nos haya dedicado su tiempo.

- **Análisis**: se trata de ver como utilizar los conocimientos adquiridos. Para ello las actividades son:

  1. De tipo *administrativo*, como por ejemplo, pasar a limpio la entrevista.
  2. *Asimilación* de la información: Se contrasta con otras entrevistas, bibliografía, etc. Se llega a conclusiones.
  3. *Evaluación* de la entrevista: ¿Qué objetivos se pretendía conseguir y qué es lo que realmente se ha conseguido?

  Para *validar* una vez más la entrevista se puede mandar la documentación generada al entrevistado.

Desarrollo conjunto de aplicaciones (JAD)

En el apartado anterior se vio una somera introducción a una entrevista *clásica* donde existe *un* entrevistador y *un* entrevistado. Ahora se contempla un tipo de entrevista desarrollada por IBM para varias personas. Consiste en un conjunto de reuniones en un periodo de entre dos y cuatro días. Se basa en cuatro principios:

1. *Dinámica de grupo*: son técnicas de discusión para generar ideas y confrontar puntos de vista.
2. Uso de *ayudas audiovisuales*: diagramas, transparencias, pizarras, etc.
3. Modo de trabajo sistemático: se sigue el guión que se expone en las fases. Todo el mundo tiene asignado un rol.
4. Filosofía de documentación WYSIWYG (*What You See Is What You Get*).

Existen dos tipos de sesiones JAD: JAD/Plan, para obtención de requisitos y JAD/Design, para el diseño. Veremos sólo el primero.

Ventajas del JAD:

1. La información obtenida se puede contrastar *in situ* porque están todos los interesados. En las entrevistas individuales este es un proceso lento. Además en esta contrastación participan tanto los usuarios como los desarrolladores, esto quiere decir que los requisitos que se obtienen son los correctos.
2. Los clientes se sienten involucrados en el proceso de desarrollo porque ellos mismos participan en la exploración de los problemas que se plantean, con lo cual la resistencia al cambio será menor.

Inconvenientes:

1. Al ser varias personas es difícil encontrar un hueco en la agenda de todos para estas reuniones.
2. Es una técnica complicada.

Participantes: Participan de ocho a doce usuarios a parte de los analistas. Hay seis tipos:

1. *Jefe del JAD*: debe tener las características clásicas de una persona que dirige una reunión: Dotes de comunicación y liderazgo. Son deseables también conocimientos de psicología para por ejemplo conseguir que todo el mundo participe o evitar conflictos, pero sobre todo, para dirigir la sesión a sus objetivos.
2. *Analista*: la función realizada es la de secretario de la sesión. Es la persona encargada de poner por escrito la información. No es una tarea tan trivial como parece, tiene que ser alguien que haya comprendido el tema y lo pueda expresar bien con las herramientas que se empleen.
3. *Patrocinador*: es el cliente: ya sea el jefe de la empresa contratante o quien tiene la capacidad para decidir si se sigue con el desarrollo. Debe informar de cuales son las necesidades que cubrirá el producto.
4. *Representantes de los usuarios*: en el JAD/Plan son directivos porque proporcionan una visión global del sistema. En el JAD/Design son usuarios.
5. *Desarrolladores*: son las personas que pueden informar de las dificultades de implementación de ciertas características.
6. *Especialistas*: son la "autoridad" a consultar sobre aspectos puntuales tanto por parte de los usuarios como por parte de los desarrolladores.

Fases del JAD

1. **Adaptación**: el jefe del JAD es quien debe adaptar la sesión a las características propias del proyecto en curso. Para ello debe:
   - Documentarse sobre la organización y el proyecto.
   - Decidir cuestiones organizativas sobre las sesiones JAD: Número y duración, lugar (mejor si es fuera de la empresa para evitar interrupciones), fechas, etc.

- Seleccionar a los participantes adecuados para cada reunión.

2. **Sesión JAD**: Está dividida en una serie de pasos:

   - *Presentación*: al igual que en la entrevista individual, el jefe del JAD y el patrocinador ejecutivo se presentan. El patrocinador explica los motivos del proyecto. El jefe del JAD explica la mecánica de la reunión.
   - *Definición de requisitos de alto nivel*: esto ya es parte del trabajo productivo. El jefe del JAD hace preguntas del tipo: *¿Qué beneficios se esperan del sistema? ¿Cuáles son los recursos disponibles?* Estos requisitos se van escribiendo en algún medio que permita que todo el mundo lo pueda ver, por ejemplo transparencias.
   - *Delimitar el ámbito del sistema*: cuando se ha reunido un conjunto de requisitos lo suficientemente grande se les puede organizar y decidir el ámbito del sistema-
   - *Documentar temas abiertos*: si un tema queda sin resolver se debe documentar para otra sesión y asignar una persona responsable de su solución.
   - *Concluir la sesión*: el jefe del JAD hace un repaso de la información obtenida con los participantes. Es el momento de hacer correcciones o añadidos.

3. **Organización de la documentación**: se trata de producir un documento con la información recabada en la fase anterior. Hay tres partes que se siguen secuencialmente.

   - *Compilar la documentación*: la documentación recogida se redacta en un documento normalizado.
   - *Revisar la documentación*: se envía la documentación a los participantes para que efectúen correcciones.
   - *Validar la documentación*: el patrocinador ejecutivo da su aprobación.

Planificación conjunta de requisitos (JRP)

Es un subconjunto de las sesiones JAD. Se caracterizan por estar dirigidas a la alta dirección y en consecuencia los productos resultantes son los requisitos de alto nivel o estratégicos. La planificación de una sesión consiste en varios pasos:

1. *Iniciación*: se delimita el alcance del plan de sistemas de información, unidades organizativas afectadas y perfiles necesarios para la reunión.
2. *Búsqueda*: se identifican objetivos, situación actual e información al respecto.
3. *Lugar*: es importante que al igual que en la sesión JAD sea fuera de la organización para evitar interrupciones. La sala de reuniones debe ser además confortable y equipada con el mobiliario adecuado. Las ayudas audiovisuales pueden ser pizarras, proyectores, etc. Se debe contar además con ordenadores equipados con herramientas CASE, procesadores de texto, hojas de cálculo y herramientas de prototipado.
4. *Seleccionar a los participantes*. Los perfiles son los siguientes:

   - *Jefe JRP*: debe tener las mismas aptitudes que en el caso del JAD.

- *Patrocinador*: el que respalda económicamente el proyecto.
- *Director del proyecto*
- *Consultores*: traduce los requisitos de usuario a información estructurada inteligible por los usuarios.
- *Especialista en modelización*: Elabora los modelos en la reunión.
- *Usuarios de alto nivel*: Definen los procesos organizativos y los sistemas de información afectados por el plan de sistemas de información y las prioridades de implantación.

5. Redactar la **agenda** de asuntos a tratar. Esta agenda debe ser planificada asignando tiempos para cada cuestión.
6. **Realización**: Discurre igual que la sesión JAD.

Brainstorming

Este es otro tipo de entrevista de grupo. A diferencia del JAD que está fuertemente estructurada, el brainstorming (tormenta de ideas) se caracteriza precisamente por lo contrario, porque aunque existe un jefe del grupo, su función es meramente anecdótica. El objetivo es la generación de ideas novedosas para la resolución de un problema. Su utilización es adecuada al principio del proyecto, pues puede explorar un problema desde muchos puntos de vista. Una característica importante es que sirve para encontrar soluciones novedosas.

Ventajas del brainstorming

1. Es fácil (la única norma es que vale todo y no se puede criticar a nadie). Esta técnica es muy usada por este motivo.
2. No está tan formalizado como el JAD.

Inconvenientes

1. No proporciona resultados con mucho nivel de detalle.
2. Es difícil reunir a todo el mundo.

Fases del Brainstorming

1. **Preparación**
   - Se selecciona a las personas involucradas, es decir: Jefe de la reunión, usuarios y desarrolladores.
   - Se les convoca a una hora y en un lugar determinados.
2. **Desarrollo**: El jefe expone el problema a resolver, entonces cada persona expone sus ideas para la solución. El jefe da la palabra a una persona u otra. Cuando se han generado suficientes ideas se termina la reunión. Normas a seguir:

- No se permite la crítica a ningún participante, diga lo que diga.
- Se promueven las ideas más creativas aunque no sean factibles para estimular la creatividad del resto del grupo.
- Cuantas más ideas salgan mejor.
- Los participantes pueden añadir ideas propias a las ideas de otros.

3. **Consolidación**: Es similar a la fase de análisis en la entrevista individual o la fase de organización de la documentación de JAD. Se divide en tres partes

   - *Clasificar ideas* para agruparlas o fusionarlas.
   - *Descartar ideas* peregrinas o poco factibles.
   - *Priorizar ideas* en función de su importancia.

4. **Documentación**: Se pone por escrito la información que sale de la fase de consolidación.

Prototipos

Un prototipo es una versión reducida de la aplicación final. Se puede construir con dos filosofías[Som98]:

1. **Prototipos de desarrollo rápido**. Sirven para obtener y validar requisitos. Cuando han cumplido esta finalidad se "desechan".
2. **Prototipo inicial**: Se desarrolla el sistema de un modo incremental partiendo de una versión inicial.

El segundo caso se ha discutido ya en el capítulo dedicado a ciclos de vida. El primer caso sigue el proceso indicado en la figura 2.2.

Un prototipo no es utilizable como sistema final porque se ha desarrollado con la máxima rapidez y en consecuencia:

1. Los requisitos no funcionales tales como rendimiento, seguridad, etc no se implementan.
2. No hay documentación.
3. El sistema será caro de mantener porque el prototipo habrá sufrido muchos cambios a lo largo del desarrollo y su estructura estará en mal estado.
4. La calidad del código es mala.

Existen tres **técnicas de desarrollo rápido de prototipos**[Som98] que se pueden usar conjuntamente: Lenguajes de alto nivel, Programación de bases de datos y Componentes reutilizables.

- *Lenguajes de alto nivel*: Son lenguajes de gran nivel de abstracción que necesitan menos líneas de código para hacer lo mismo que en un lenguaje de bajo nivel como C. Ejemplos de lenguajes de alto nivel son: Lisp, Smaltalk y Prolog. Los lenguajes de alto nivel tienen la desventaja de cargar mucho al sistema, razón por la cual no están más extendidas, aunque este motivo es cada vez menos importante dada la potencia de las máquinas actuales.

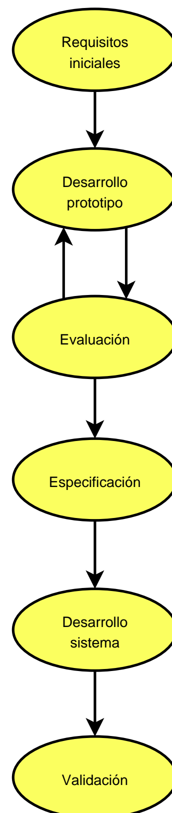


Figura 2.2: Obtención y validación de requisitos a partir de prototipos [Som98]

Cuando se desarrolla un prototipo hay que responder algunas cuestiones para escoger el lenguaje, tales como cual es el mejor lenguaje para el dominio del problema que se está abordando. Por ejemplo, Lisp y Prolog pueden ser adecuados para inteligencia artificial, Java si se necesita un software multiplataforma, etc. Se deben tener en cuenta asimismo factores tales como las facilidades suministradas por la herramienta para la construcción automatizada de interfaces de usuario, la experiencia que tiene con dichas herramientas el personal con el que contamos, si se puede conectar o forma parte de una herramienta CASE, etc.

- *Programación de bases de datos*: Una base de datos tiene un lenguaje de programación que permite manipularla y utilidades. Un lenguaje de cuarta generación es el lenguaje de programación de la base de datos y su entorno. Son interesantes las herramientas que permiten acceder a la base de datos a través de un navegador ya que así se puede acceder desde cualquier punto.
- *Componentes reutilizables*: El principio en el que descansan los componentes es que es más rápido usar software hecho que hacerlo, por ejemplo, los *beans* de Java. Si además existe algún mecanismo que permita que esos componentes se comuniquen y se integren fácilmente, mejor. Problemas que se pueden presentar:

1. No existen todos los componentes necesarios para el prototipo que se quiere desarrollar, con lo que hay que desarrollarlos.
2. Los componentes existen, pero no son exactamente lo que se necesita y hay que adaptarlos.

El desarrollo de prototipos con reutilización se puede hacer a dos **niveles**:

1. *Nivel de aplicación*: Los sistemas que componen la aplicación pueden ser compartidos por otras aplicaciones. Por ejemplo: se puede insertar un gráfico desarrollado con una aplicación en otra. Las aplicaciones actúan como si estuvieran conectadas entre si al estilo de las tuberías de Unix.
2. *Nivel de componente*: Los componentes están integrados en un armazón estándar, como puede ser un lenguaje de desarrollo de componentes, por ejemplo: PYTHON o PERL o algo más general como CORBA, DCOM o JAVABEANS.

JAVABEANS es un ejemplo de componentes reutilizables. Son un tipo de componentes escritos en Java, que generalmente tienen algún tipo de interfaz gráfica (*Enterprise-JavaBeans* sin embargo no es así, pues se ocupa más de la lógica de la aplicación). Sus características son:

1. Tienen un constructor predeterminado sin parámetros (es decir, un *bean* es una clase que cumple una serie de restricciones)
2. Son *persistentes*. La persistencia es la capacidad de un objeto de convertirse en un flujo de bytes para por ejemplo poder ser escrito en disco o recuperado del mismo. Un objeto persistente puede sobrevivir al apagado del sistema y seguir funcionando del mismo modo.
3. No son nunca clases abstractas, es decir, siempre se pueden crear instancias de ellos.
4. Para escuchar un evento se usan estas dos funciones:
   ```
   public void addEventListenerType(EventListenerType evento)
   public void removeEventListenerType(EventListenerType evento)
   ```
5. La forma de dar valor o conocer el valor de una propiedad es con métodos de la forma: `getX()` y `setX(Propiedad propiedad)`. Si la propiedad es booleana en vez de `getX()` se usa `isX()`. Si la propiedad es indexada se utiliza: `getX(int index)` ó `getX()` que devuelve un *array* y los métodos para asignar valor serían: `setX(int index, propiedad)` ó `setX(Propiedad[] propiedad)`.

Casos de uso

Son una forma de especificar los requisitos de un sistema. Un caso de uso consiste en una interacción de algo externo al sistema y el sistema. Fueron introducidos por Jacobson en 1992. Aunque es una técnica definida dentro del ambiente del análisis orientado a objetos no tiene que ver con objetos, se podría utilizar perfectamente dentro del análisis estructurado. Más adelante veremos la técnica de los casos de uso. Un caso de uso:

- Describe la interacción entre un *actor* externo al sistema y el sistema con texto en lenguaje natural.

- Representa los requisitos funcionales desde el punto de vista del usuario y por lo tanto produce un resultado observable por él.
- Es iniciado por un único actor.
- Realiza una funcionalidad concreta.

Los objetivos de los casos de uso son:

- Comprender la funcionalidad del sistema.
- Discutir con el usuario nuestra visión de esa funcionalidad.
- Identificar conceptos del sistema, clases, atributos y operaciones.
- Validar el análisis y el modelo del diseño.
- Proporcionar información para las pruebas del sistema y de aceptación.

Existen dos tipos de elementos:

1. **Actores**: El actor puede ser tanto una persona como otro sistema que juega un **rol** en la interacción con el mismo. Un mismo rol puede ser desempeñado por varias personas y una persona puede desempeñar más de un rol. Un usuario no es un actor, sino que asume un rol cuando interactúa con el sistema y por lo tanto funciona como un tipo de actor.
2. **Caso de uso**: Es la interacción que se quiere modelar. Pueden agruparse en paquetes y tener relaciones entre ellos.

Identificar actores Un caso de uso se compone de actores y de interacciones de esos actores sobre el sistema. por lo tanto, lo primero es buscar actores. Teniendo en cuenta lo que es un actor, hay que encontrar la siguiente información:

1. Identificar los **usuarios** del sistema. Para ello tener en cuenta para quien se está diseñando o con que personas va a interactuar de un modo directo (actores principales) y quienes va a tener un papel de supervisión o mantenimiento del sistema (actores secundarios).
2. Identificar los **roles** que juegan esos usuarios desde el punto de vista del sistema. Hay varios tipos, gestores de alto nivel, gente que introduce datos, etc.
3. Identificar **otros sistemas** con los cuales exista comunicación. Estos sistemas también se consideran como actores dado que son algo externo a nuestro sistema.

Identificar operaciones Una vez que se tienen los actores se trata de encontrar los casos de uso, como lo que tenemos en este momento son los actores, partimos de esta base para encontrar la información que falta. Los actores interactuarán con el sistema realizando un conjunto de tareas que tendremos que enumerar y manejarán información, tanto la que suministren al sistema como la que el sistema les suministre a ellos. Una vez que se dispone de los actores y de los casos de uso hay que encontrar las relaciones que hay entre ellos. Las relaciones que hay entre casos de uso son de dos tipos:

- Aquellas en las que uno **extiende** a otro: Son muy parecidos (comparten muchos pasos) pero tienen ligeras diferencias adaptadas a la forma particular en la que se realiza una tarea.
- Aquellas en las que uno **usa** a otro: En esta situación un caso de uso es similar a una función o subrutina que es llamada por otro.

Otra forma de hallar casos de uso es hacer una lista de eventos. Un evento es una ocurrencia a la que el sistema tiene que responder. No supone un diálogo como un caso de uso porque ocurre de un modo atómico. Los pasos a seguir en este caso son por un lado identificar todos los eventos a los que el sistema tiene que responder y luego relacionarlos con los actores adecuados.

Una vez identificados estos casos de uso podemos sacar otros nuevos de varias formas:

1. Por variaciones respecto a casos de uso existentes si:
    - Existen diferentes actores que puedan utilizar un mismo caso de uso pero con variaciones significativas.
    - Existen diferentes versiones del mismo caso de uso con variaciones significativas.
2. Buscando el caso de uso opuesto a uno dado, por ejemplo, si tengo *dando de alta cliente* podría existir el caso de uso *dando de baja cliente*
3. Preguntarnos que tiene que ocurrir para que se cumplan las precondiciones de un caso de uso, por ejemplo, para hacer un pedido es necesario que el cliente entre con su *password*, con lo que tiene que existir un caso de uso para dar de alta.

Ejemplo *En una universidad el proceso de matriculación es el siguiente: Si el alumno es nuevo se le pide cierta documentación (fotocopia del DNI o pasaporte, fotografías, certificados, etc.) si es antiguo se pide sólo la fotocopia del DNI o pasaporte. El alumno escoge las asignaturas en las que desea matricularse. En este proceso puede ocurrir algún error, como que escoja alguna en la que no pueda matricularse, que no escoja ninguna, etc. El sistema deberá dar los oportunos mensajes de error para ayudar al usuario. Otro tipo de alumno es el que pide traslado de expediente de otra universidad. En este caso debe aportar la misma documentación que el alumno nuevo y la documentación que acredite que ha aprobado las asignaturas que sean. Debe hacer asimismo una solicitud de convalidación de asignaturas. Cuando esto acaba se matricula como un alumno antiguo. Por otra parte, los profesores usan el sistema para poner las calificaciones. Una vez que el profesor se identifica adecuadamente y selecciona la asignatura y la convocatoria el sistema presenta un listado y el profesor rellena las notas. Existe un proceso análogo para modificar la nota de un alumno. El jefe de estudios es el profesor que se encarga de gestionar el proceso de composición de horarios: introduce el número de horas, los equipos docentes y otros datos y restricciones de cada asignatura, el sistema genera unos horarios usando un algoritmo de investigación operativa y los presenta al jefe de estudios. Si está satisfecho termina el proceso, en caso contrario introduce una nueva restricción y se generan nuevamente los horarios. Si las restricciones son demasiado fuertes puede ocurrir que no exista solución, en este caso el sistema imprime una advertencia y el jefe de estudios debe eliminar alguna restricción.*

Para resolver el problema vamos a adoptar la estrategia recomendada.

1. **Identificar actores**

   - Identificamos usuarios. Hay dos: Estudiantes y profesores.
   - Identificamos roles que juegan. Los estudiantes por un lado *consultan* datos (asignaturas disponibles en las que pueden matricularse) e *introducen* datos (asignaturas en las que quedan matriculados). Los profesores tienen los mismos papeles. El jefe de estudios juega un papel de *gestor* de alto nivel.
   - Identificamos otros sistemas: No hay.

2. **Identificar operaciones**:

   - Matricular a un alumno.
   - Introducir notas.
   - Modificar notas.
   - Generar horarios.

   Posibles variaciones de los casos de uso existentes: La operación de matriculación es diferente en función del tipo de alumno: Nuevo, Antiguo y el que viene de otra universidad. Podemos derivar de eso tres posibles casos de uso:

   - Matricular alumno nuevo.
   - Matricular alumno antiguo de esta universidad.
   - Matricular alumno nuevo que viene de otra universidad.

   Estos tres casos de uso se parecen entre si. Podemos pensar que entre ellos puede haber algún tipo de relación de herencia. Como la herencia trata de hallar la parte común deberíamos buscar el más sencillo. El más sencillo es el de *matricular alumno antiguo*, los otros dos son modificaciones más o menos complicadas.

   El hecho que un caso de uso haya generado otros dos relacionados entre si por herencia nos puede hacer pensar que al actor relacionado (*alumno*) le pasa lo mismo. Lo que está claro es que un mismo alumno sólo va a emplear uno de los tres casos de uso, aunque no creemos que esto por si solo sea motivo para crear tres tipos de alumno[3]. Si que está claro sin embargo que se deben hacer tres casos de uso diferentes y relacionados del modo en el que lo están porque son funciones que difieren ligeramente entre ellas.

   Se puede pensar en crear un caso de uso para seleccionar las asignaturas en las que se matricula un alumno. El criterio para generar el caso de uso o dejarlo como una mera línea en la descripción de los casos de uso va a ser la complejidad del proceso. Vamos a suponer

[3] Esto es subjetivo, la otra solución sería perfectamente válida. La ventaja de esta es no complicar el modelo más de lo necesario.

que es lo suficientemente complejo[4], con lo que tendremos el caso de uso *Seleccionar asignaturas*, usado por los casos de uso de matriculación.

Hemos visto también que hay un tipo de profesor especial llamado *Jefe de estudios*. Como tiene una función específica dentro de la organización podemos suponer que es un tipo de actor con algunas características que le distinguen del profesor normal.

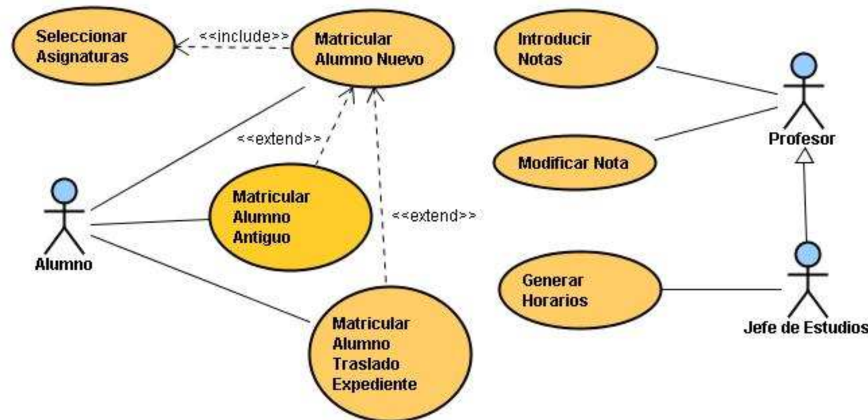


Figura 2.3: Conjunto de casos de uso

Dar detalle a los casos de uso descritos Esto que se ha descrito hasta ahora es la forma de construir un armazón de casos de uso, veamos como se pueden concretar un poco más. Para cada caso de uso hay que:

- Describir la información de entrada y de salida.
- Identificar las *precondiciones*, o lo que es lo mismo, todo lo que se tiene que cumplir antes de realizar el caso de uso.
- Identificar las *postcondiciones*; lo que se tiene que cumplir después de realizar el caso de uso.
- La descripción detallada del caso de uso contiene:
  - Una descripción textual de su objetivo.
  - Variantes posibles para realizar este caso de uso. Diagramas de interacción de detalle (de secuencia o colaboración).
  - Errores y excepciones posibles.
- Relacionar el caso de uso con la interfaz de usuario que lo representa.
- Especificar el diálogo que da solución al caso de uso.

La descripción de un caso de uso consta de los siguientes elementos:

[4] Sólo para poner el ejemplo

1. Descripción del escenario.
2. Suposiciones acerca del escenario.
3. Condiciones previas para el caso de uso.
4. Se expresa en una tabla con la secuencia de interacciones. La tabla puede tener una columna con el curso normal de acontecimientos y otra con posibles alternativas. Cada alternativa representa un error o excepción y no tienen suficiente entidad como para ser un caso de uso al que se llegue con una relación *extiende*.
5. El actor responsable.
6. Resultado una vez terminado el caso de uso.

Actores y casos de uso abstractos

Un caso de uso que sólo exista para ser utilizado por otros casos de uso se dice que es abstracto. Como ese caso de uso no tiene un actor que lo utilice, nos inventamos un actor ficticio (abstracto). La manera en la que se relaciona este nuevo actor ficticio con los de verdad es por medio de la herencia. Por lo tanto, todos los actores que utilicen un caso de uso que a su vez utiliza un caso de uso abstracto heredarán de él. Un actor puede heredar de cualquier otro, tanto si es abstracto como si no, con lo que hereda también todas las funcionalidades a las que puede acceder.

Otras clasificaciones de los casos de uso

Casos de uso esenciales y de implementación Cuando se está en la fase inicial de recopilar todos los casos de uso, no se ponen en detalle, se indica su nombre y poco más (casos de uso esenciales, o de trazo grueso). Cuando se va a implementar el caso de uso hay que definirlo con más detalle (caso de uso de implementación o trazo fino).

Casos de uso temporales Son aquellos iniciados no por un actor, sino por un evento de reloj. Se debe indicar el momento en el que esto ocurre, p. ej.: cada 500 milisegundos. La notación consiste en poner el caso de uso en un óvalo con línea de puntos o con símbolo de reloj.

Casos de uso primarios y secundarios Los casos de uso secundarios son aquellos que existen por que son necesarios para que el sistema funcione pero que no son inherentes a su funcionalidad.

2.2. Análisis de requisitos

Una vez que hemos extraído los requisitos del problema es necesario convertir esos requisitos en un modelo del problema que distinga los elementos que forman parte de los requisitos y las relaciones entre ellos, así como las funcionalidades esperadas del conjunto. En este apartado se estudiarán las diferentes estrategias o patrones de modelado del problema.

2.3. Representación de requisitos

Como resultado del análisis de los requisitos especificados inicialmente, se obtiene un modelo del problema que es necesario representar de una manera formal que permita la aplicación de técnicas sistemáticas para su resolución. Existen diversos lenguajes y métodos de especificaciones que tienen características diferentes según el método posterior del desarrollo. Por ejemplo diagramas entidad-relación para modelado, diagramas contextuales, plantillas o marcos conceptuales, diagramas funcionales, etc. El método más extendido de representación es mediante técnicas orientadas a objetos y los lenguajes asociados a ellas.

2.4. Análisis orientado a objetos

El análisis orientado a objetos representa el problema en términos de clases, sus propiedades y sus relaciones. El modelo que resulta de este análisis utiliza técnicas muy diversas, algunas como la de los casos de uso son perfectamente aplicables también al análisis estructurado. La ventaja del análisis orientado a objetos es que la representación que se logra del mundo es más próxima a la realidad. La transformación de los requisitos en términos de usuario a la especificación en términos de sistema informático es más suave.

2.5. Bases de documentación

Ya desde los primeros pasos del desarrollo del software es necesario comenzar a organizar toda la documentación. En primer lugar se han de gestionar los formatos en los que se va a escribir la documentación. Preferiblemente se deben utilizar herramientas de ayuda que semiautomatizan la recopilación de la documentación durante el desarrollo. Evidentemente, tanto las especificaciones iniciales del problema (en lenguaje natural probablemente) como las especificaciones finales de los requisitos (en un lenguaje formal), pasando por los estados intermedios de análisis y modelado y las diferentes pruebas de validación se deben guardar como parte inicial de la documentación.

El documento de especificación de requisitos SRS (*Software Requirements Specification*) es como el contrato oficial que se establece entre los desarrolladores y los clientes. Incluye tanto los requisitos funcionales como los no funcionales. Una recolección correcta y completa es crucial para el desarrollo de un sistema de información exitoso. Para alcanzar el mayor grado de calidad es esencial que el SRS sea desarrollado de un modo sistemático e inteligible, en cuyo caso reflejará las necesidades del usuario y será útil para todos. Lo que sigue es una plantilla del SRS según el estándar IEEE 830.

En la portada se indican: Nombre de la empresa, Nombre del proyecto, Número de versión y Fecha.

El índice debe constar de las siguientes secciones: Introducción, Descripción general, Requisitos específicos, Gestión del cambio del proceso, Aprobación del documento e Información de soporte. El desarrollo del índice anterior debe contener la siguiente información:

1. **Introducción**. Las siguientes subsecciones dan una visión general del documento.

a) *Propósito*: Identifica el propósito del documento y el perfil de usuario al que está dirigido.

b) *Alcance*. En esta subsección:

- Se identifica el nombre de los productos software que se van a desarrollar.
- Se explica lo que hacen los productos y lo que no hacen.
- Se describen las aplicaciones del software especificado, incluyendo beneficios y objetivos.
- Se debe ser coherente con especificaciones similares de más alto nivel si existen, es decir, en el caso de que estas sean las especificaciones de un subsistema.

c) *Definiciones, acrónimos y abreviaturas*: Es un listado de las definiciones de todos los términos, acrónimos y abreviaturas necesarios para interpretar correctamente el SRS. La información puede ser proporcionada con referencias a apéndices.

d) *Referencias*:

- Se da la lista completa de todos los documentos referenciados en cualquier lugar de la SRS.
- Se identifica cada documento por título, versión, fecha y responsable de su redacción.
- Se especifican las fuentes desde las que se pueden obtener las referencias.

e) *Visión de conjunto*: Describe la organización general de la SRS.

2. **Descripción general**: Describe los factores generales que afectan al producto y sus requisitos. No se definen requisitos específicos (sección 3), sólo su contexto.

a) *Perspectiva del producto*: Relación del producto con otros similares. Si el producto es independiente y autocontenido se debe decir aquí. Si es un componente de un sistema mayor aquí deben estar los requisitos para que el sistema mayor lo pueda utilizar e identifica las interfaces entre el sistema y el software. Se puede poner un diagrama de bloques para mostrar las interconexiones del sistema con otros e interfaces externas.

b) *Interfaces del sistema*a: Identifica la funcionalidad del software y la descripción de la interfaz. Especifica:

- Características lógicas de cada interfaz entre el producto y sus usuarios.
- Todos los aspectos relativos a la optimización de la interfaz con los usuarios.

1) *Interfaces hardware*: Se especifican las características de cada interfaz entre el producto y los componentes hardware. Incluye la configuración, dispositivos soportados, cómo son soportados y protocolos.

2) *Interfaces software*: Es una lista de otros productos software e interfaces con otras aplicaciones.

a' Para cada producto se debe incluir: Nombre, Mnemónico, Número de versión y Fabricante.

b' Para cada interfaz se debe desarrollar en: motivos para interactuar con el producto y definición de la interfaz acerca de contenido de los mensajes y formatos.

3) *Interfaces de comunicaciones*. Por ejemplo protocolos de comunicaciones.

c) *Restricciones de memoria*.

d) *Operaciones normales y específicas* tales como:

1) Modos de operación en la empresa del usuario.
2) Operaciones interactivas y no interactivas.
3) Funciones de proceso de datos.
4) Operaciones de *backup*.

e) *Requisitos de instalación*. Se definen:

1) Requisitos para datos y secuencias de inicialización específicos para un emplazamiento, misión o modo de operación.
2) Características necesarias para adaptar el software a un entorno concreto.

f) *Funciones del producto*: Es un resumen de las funciones que proporciona el software. A veces esta lista puede copiarse de las especificaciones de alto nivel. Este resumen:

- Debe ser inteligible por el cliente con una sola lectura.
- Se pueden utilizar elementos textuales o gráficos para mostrar las diferentes funciones y sus relaciones. No es un gráfico del diseño, sólo muestra relaciones lógicas entre funciones y variables.

g) *Características del usuario*: Es una lista de aptitudes y motivos para ellas.

h) *Restricciones*: Descripción general de limitaciones que afectarán a los desarrolladores tales como: Políticas, Limitaciones de hardware, Interfaces con otras aplicaciones, Operaciones en paralelo, etc.

i) *Suposiciones y dependencias*: Lista de cada uno de los factores que afectan a los requisitos. No son restricciones de diseño pero un cambio en ellos afectaría a los requisitos.

j) *Requisitos no implementados*: Lista de requisitos que se dejan para futuras versiones.

3. **Requisitos específicos**: Es una lista de los requisitos a un nivel de detalle lo suficientemente fino que permita a los desarrolladores diseñar un sistema que los satisfaga. Deben incluir las entradas, las salidas y las funciones suministradas en respuesta a entradas o para suministrar una salida.

a) *Interfaces externos*: Es una lista detallada de todas las entradas y salidas. Complementa las descripciones de interfaz en el apartado $2b$ pero sin repetir información.

b) *Requisitos funcionales*: Definen las acciones que el software tiene que tomar para producir las salidas a partir de las entradas. Ejemplos:

- Comprobar la corrección de la entrada.
- Secuencia correcta de las operaciones.
- Respuestas a situaciones anormales.
- Relaciones entre entradas y salidas.

c) *Requisitos de rendimiento*: Especifica requisitos estáticos y dinámicos. Un requisito estático de rendimiento puede ser el número de usuarios conectados simultáneamente. Un requisito dinámico sería por ejemplo el número de transacciones por minuto. Estos requisitos deben especificarse en unidades medibles.

d) *Requisitos relativos a bases de datos*: Requisitos lógicos que tiene que tener la información que se deje en la base de datos. Por ejemplo: Tipos de información, frecuencia de uso, etc.

e) *Restricciones de diseño*: Restricciones de diseño que pueden ser impuestas por otros estándares, limitaciones de hardware, etc.

    1) Conformidad con los estándares: Requisitos derivados de la existencia de estándares. Por ejemplo: Formato de los informes, nombre de los datos, procedimientos de contabilidad o auditorias.

f) *Atributos del sistema software*: Son propiedades del sistema. El hecho de tenerlas constituye un requisito. Los más importantes son: Fiabilidad, Disponibilidad, Seguridad, Mantenibilidad y Portabilidad.

g) *Organización de los requisitos*. Los requisitos deben ser presentados de un modo jerárquico para que la cantidad de información no sea abrumadora para el lector y se puedan comprender con facilidad. No existe un modo óptimo para presentarlos. Esto es una lista de posibles criterios.

- Modo del sistema: Es el modo de operación. Algunos sistemas se comportan de un modo muy distinto en función del modo en el que estén.
- Clases de usuario: En función de a quién esté dirigido interesará resaltar unas funciones u otras.
- Objetos: Se pueden agrupar las funciones en términos de las suministradas por jerarquías de objetos.
- Características: Son servicios observables externamente que puede requerir una secuencia de entradas y salidas en un orden concreto.
- Estímulo: Algunos sistemas se describen mejor en términos de estímulo - respuesta.
- Respuesta: Clasificación en función del tipo de respuesta que da el sistema.
- Jerarquía funcional: Se trata de organizar el sistema en términos de funciones con entradas y salidas similares.

   h) *Comentarios adicionales*.

4. **Gestión del cambio del proceso**: Seleccionar la gestión del proceso de cambio que deba ser usada para identificar, registrar, evaluar y actualizar el SRS para que refleje el alcance de los cambios en el proyecto y sus requisitos.
5. **Aprobación del documento**: Identificar a las personas que deben dar su visto bueno. Deben constar su nombre, firma y fecha.
6. **Información de soporte**. Hace que este documento sea más fácil de usar. Incluye índice y apéndices.

Tutorial posterior

Resumen de contenidos

En este capítulo se ven las formas de captura de requisitos y su representación. La finalidad de esta fase es producir un documento que represente los requisitos y que sirva de entrada para la fase de diseño. El análisis es la forma de conseguir los requisitos y la especificación la forma de representarlos.

Técnicas de obtención de requisitos

1. **Entrevistas**: Consiste en hablar con el cliente. Hay que tener sobre todo conocimientos de psicología para facilitar la comunicación.
2. **JAD**: Consiste en un tipo de entrevista muy estructurada aplicable a grupos de personas. Cada persona juega un papel concreto y todo lo que se hace está reglamentado.
3. **JRP**: Es un subconjunto de JAD.
4. **Brainstorming**: Es un tipo de entrevista que no tiene ningún tipo de estructura o reglamentación. Genera ideas novedosas.
5. **Prototipos**: Se trata de construir una mini-aplicación inicial para clarificar algunos puntos y luego tirarla o bien para usarla como base para añadir más cosas.
6. **Casos de uso**: Representan gráficamente los requisitos funcionales y relaciones de uso y generalización entre ellos. Es la técnica definida en UML. No está necesariamente asociada a la programación orientada a objetos, puede usarse también con metodologías más clásicas.

Análisis de requisitos

Características del análisis antes de la existencia de técnica alguna del tema: Monolítico, ambiguo, difícil de mantener y redundante. Posteriormente (años 70 y 80) surge el análisis estructurado moderno que es gráfico y soluciona algunos de los problemas anteriores. Su objetivo es modelar por separado los datos, procesos y el control usando como nexo común el diccionario de datos.

Técnicas de representación de requisitos

1. **Diagramas de flujo de datos**: Se describen los elementos de los que consta, lo que es el diagrama de contexto y heurísticas para desarrollarlo.
2. **Diagramas de flujo de control**: Es similar a los DFD.
3. **Diagramas de estados**: Son un tipo de modelización matemática usada también en diseño de circuitos, compiladores, etc.
4. **Modelo Entidad / Relación**: Usado para representar los datos y la forma en la que se relacionan entre ellos.
5. **Diccionario de datos**: Se puede definir como el pegamento que mantiene unido a todo lo anterior. Es una descripción detallada de los datos.

Análisis orientado a objetos

Es la otra forma de ver el problema. Se representa el mundo en función sobre todo de los **datos**, características e interrelaciones entre ellos en vez de en función de los algoritmos o funciones.

Propiedades de un sistema orientado a objetos: Abstracción, Encapsulamiento, Modularidad, Jerarquía, Polimorfismo, Persistencia.

Conceptos importantes: Clase, Objeto y Atributos.

Sesión CRC: Es una simulación hecha por personas del funcionamiento dinámico de un sistema orientado a objetos. Se basa en que cada persona representa una clase y su interacción con las demás. Cada clase tiene escritas sus propiedades en una tarjeta.

Identificación de clases, atributos y relaciones entre clases.

Refinamiento del modelo: Consiste en otro conjunto de heurísticas para eliminar elementos sobrantes o identificar otros nuevos en el modelo anterior.

Validación de requisitos

Es la comprobación de que todo el trabajo anterior se ha realizado bien.

Bases de documentación

Es el documento que resulta de todo lo anterior pero orientado al cliente, es decir, a las obligaciones contraídas con él, la idea no consiste en usar este documento como entrada para otra fase. Se ha tomado la plantilla de documento definida por el estándar IEEE 830.

Ejercicios y actividades propuestas

1. Compare los diagramas del análisis estructurado y los del UML. ¿Existen similitudes?

1. ¿Qué diferencia existe entre los requisitos que se obtienen en una sesión JAD y una sesión JRP?
2. Si se tiene que construir un sistema novedoso en el que los requisitos están claros pero se tienen dudas técnicas acerca de la mejor solución discuta que tipos de sesiones serían las más adecuadas.
3. Un videoclub tiene diferentes tipos de precios para sus películas en función de su categoría. Cada película tiene un código y aunque haya varias películas con el mismo título el código es distinto para cada una. Los datos que interesa guardar de cada película son: Título, Director, Actores principales, Localización, Categoría y Código. Los clientes tienen asignado un Código de cliente y se conoce también su Nombre, Dirección, Teléfono. El videoclub hace préstamos a sus clientes por dos días y se podrá sancionar a quien se retrase según el número de días.

    - Desarrolle los DFD que sea posible.
    - Desarrolle la especificación de los procesos.

Capítulo 3

Fase de diseño

3.1. Conceptos y elementos del diseño

Partiendo de los requisitos para el problema, es necesario decidir cómo se va a realizar la solución para el problema. Existen unos cuantos principios de diseño básicos y elementos que se deben considerar antes de ponerse a decidir un diseño en particular. En primer lugar el principio de modularidad está presente en casi todos los aspectos del diseño al igual que en muchas otras áreas de ingeniería. Junto con las especificaciones funcionales y de datos hay que considerar también el diseño de la interfaz y el acoplamiento de las diferentes partes.

3.1.1. Patrones

En otras ramas de la ingeniería establecidas desde hace más tiempo, existen prototipos de soluciones estándar para problemas conocidos, por ejemplo, los tornillos siguen una normativa en cuanto a diámetro de la rosca, longitud, material, forma de la cabeza, etc. Los patrones se pueden ver como un intento de hacer eso mismo pero a diferentes niveles de abstracción.

Es frecuente encontrarse con el mismo tipo de problema más de una vez. Si se ha diseñado una solución para ese problema sería deseable ahorrar trabajo de algún modo. Un patrón da una solución ya ensayada para un tipo de problema. La idea partió en un principio del arquitecto Christopher Alexander en los años 70.

Un **patrón** de diseño identifica las clases y sus instancias, sus papeles y colaboraciones y la distribución de responsabilidades.

Un **anti-patrón** es un error y las consecuencias que se deducen de él. Hay dos tipos:

- Una mala solución a un problema y la situación causada.
- Una descripción como salir de una situación mala.

Un **framework** (suele traducirse por “armazón” pero usaremos el término inglés) es un conjunto integrado de componentes que colaboran para proporcionar una arquitectura reutilizable para una familia de aplicaciones.

Las diferencias entre los patrones y los frameworks son:

- El grado de abstracción, que en el caso del patrón es mayor.
- El tamaño: los patrones son elementos arquitectónicos más pequeños (un framework puede contener varios patrones).
- La especialización: los patrones están menos especializados que los frameworks.

Tipos de patrones

Hay muchas clasificaciones posibles, una sencilla podría ser esta:

- *Patrones arquitectónicos*: es la organización estructural de un sistema. Consiste en un conjunto de subsistemas predefinidos y sus responsabilidades. Incluye reglas para organizar las relaciones entre ellos.
- *Patrones de diseño*: son esquemas para refinar los subsistemas o componentes de un sistema o las relaciones entre ellos. Describen estructuras de comunicaciones entre componentes en un contexto.
- *Patrones de codificación*: es un patrón de bajo nivel que describe como implementar aspectos particulares de los componentes o relaciones entre ellos con un lenguaje de programación.
- *Patrones de análisis*: son un conjunto de prácticas que guían la obtención de un modelo.
- *Patrones de organización*: describen la estructuración del personal en las organizaciones.

La diferencia entre unos tipos y otros está tanto en el nivel de abstracción como en el contexto en el que son aplicables.

Además de esta clasificación general se pueden encontrar otros tipos de patrones para situaciones más específicas como programación concurrente, interfaces gráficas, organización y optimización del código, etc.

Características de los patrones

Para que un patrón se pueda considerar como tal debe pasar unas pruebas llamadas *test de patrones*, mientras tanto recibe el nombre provisional de *proto-patrón*. Según Jim Coplien un patrón debe cumplir con estos requisitos:

- Soluciona un problema.
- La solución no es obvia.
- Ha sido probado.
- Describe participantes y relaciones entre ellos.
- Tiene un componente humano.

Descripción de un patrón

Tiene varias partes:

1. *Nombre*: es una palabra significativa o una frase corta
2. *Problema*: es la descripción los objetivos dentro de un contexto
3. *Contexto*: son las precondiciones que se han de dar para que el patrón sea aplicable.
4. *Fuerzas*: son las restricciones que han de cumplirse.
5. *Solución*. relaciones estáticas y reglas dinámicas que describen como realizar la función. Es una descripción que puede tener diagramas y texto que muestra la estructura del patrón, sus participantes y sus colaboraciones. Esta solución no muestra sólo la estructura estática, sino también el comportamiento dinámico. En la descripción se incluyen además las posibles trampas que puedan existir y las variantes de la solución.
6. *Resultado*: es el contexto que queda como consecuencia de la aplicación del patrón.
7. *Justificación*: es una explicación de porque se sigue cada uno de los pasos y del patrón como un todo.
8. *Patrones relacionados*.
9. *Utilización*: es la aplicación del patrón en sistemas existentes

Patrones arquitectónicos

La arquitectura divide la aplicación en las partes que la componen. Este paso es el primero y supondrá la futura división del trabajo en los diferentes grupos.

1. **Sistemas genéricos**

   - *Layers*: organiza la estructura de las aplicaciones que pueden ser divididas en grupos de subtareas en las que cada una de ellas tenga un nivel de abstracción. El ejemplo más representativo de este patrón es la arquitectura de siete niveles OSI (físico, enlace, red, transporte, sesión, presentación y aplicación.) donde se definiría una clase para cada nivel.
   - *Tuberías y filtros*: está pensado para aplicaciones que manejan corrientes de datos. Cada paso del proceso se encapsula en un filtro. Los datos se pasan a través de tuberías entre filtros adyacentes. Se definiría una clase tubería, una clase genérica filtro y una clase por cada filtro concreto por donde deban pasar los datos. Ejemplo: La estructura de un compilador (análisis léxico, sintáctico, semántico, generador de código intermedio, optimizador y generador de código final).
   - *Pizarra*: contiene una estructura de datos llamada pizarra en la que varios procesos pueden escribir una solución parcial al problema. En este caso el sistema se descompone en la pizarra y en los diferentes subsistemas. Es apropiado para sistemas expertos o no deterministas.

2. **Sistemas distribuidos**

- *Broker*: se tienen varios sistemas diferentes que interactúan entre ellos. El broker es el elemento que coordina la comunicación y gestiona los errores.

3. **Sistemas interactivos**

   - *Modelo-Vista-Controlador*: son aplicaciones que se dividen en tres componentes: El modelo que contiene el núcleo de la funcionalidad y de los datos; la vista, que proporciona información al usuario y los controladores, que gestionan las entradas del usuario. La interfaz está formada por la vista y los controladores. Cada vez que se modifique la interfaz o el modelo hay que revisar la otra parte.
   - *Presentación-Abstracción-Control*: es otro tipo de software interactivo dentro del marco de los agentes (inteligencia artificial). Cada agente es responsable de una parte de la funcionalidad y se divide en tres partes: presentación, abstracción y control. Esta separación está diseñada para separar la interacción con el usuario del núcleo de funcionalidad del agente y de las comunicaciones con otros agentes.

4. **Sistemas adaptables**

   - *Microkernel*: se tiene por una parte un núcleo de funcionalidad y por otro una funcionalidad extendida y partes específicas de cada cliente. El microkernel funciona además como un punto de conexión al que se podrán añadir extensiones a las que gestionará. Se ha utilizado en la construcción de sistemas operativos. Es adecuado para sistemas que van a ser sometidos a cambios en sus requisitos.
   - *Reflexión*: es el caso en el que el comportamiento del sistema tiene que cambiar de un modo dinámico. Se pueden modificar las estructuras de datos y los mecanismos de llamada a las funciones. La aplicación se divide en dos partes: Un metanivel que da información acerca de las propiedades del sistema y que por lo tanto posee autoconocimiento y un nivel base donde reside la lógica de la aplicación. La implementación y los cambios se hacen en el metanivel.

Patrones de diseño

El problema del diseño es que es la etapa difícil del ciclo de vida por estar menos sistematizada y su calidad depende sobre todo de la experiencia del personal. Un patrón de diseño puede aliviar esa dificultad al tener guardada en si la experiencia de los diseñadores que lo crearon, que pueden no formar parte del equipo actual.

Los patrones de diseño han sido clasificados. Una de las clasificaciones es la del libro *Patterns in java (volume 1)*, según la cual hay seis categorías (se exponen también los patrones de cada una). Durante el diseño se trata de identificar situaciones en las que sonaplicables.

No es necesario estudiar en detalle la siguiente lista, sólo es una panorámica general.

1. **Fundamentales**: son los más importantes y los más utilizados en la definición de otros patrones.

   - *Delegation*: extiende las funcionalidades suministradas por una clase escribiendo otra que utiliza instancias de la original.

- *Interface*: una clase utiliza los servicios suministrados por instancias de otras clases a través de un interfaz.
- *Inmutable*: pensado para objetos referenciados por muchos otros o accedidos por varios procesos concurrentes de modo que no cambie su estado después de ser creados.
- *Marker interface*: son para clases que utilizan interfaces que acceden a objetos sin hacer suposiciones acerca de la semántica de tales objetos.
- *Proxy*: utilizado para que un objeto A (cliente) haga llamadas a otro B (servidor) a través de un tercer objeto C que actúa como representante de B.

2. **De creación**: describen cómo crear objetos cuando hay que tomar alguna decisión.

   - *Factory method*: define una interfaz para crear un objeto, pero la elección de la clase que se va a instanciar se decide en tiempo de ejecución. Los objetos de las subclases tienen una interfaz común.
   - *Abstract factory*: este patrón permite, al igual que el anterior, crear objetos partiendo de un conjunto de clases abstractas de las cuales heredan clases concretas.
   - *Builder*: un objeto cliente crea otro objeto especificando tipo y contenido pero sin detalles acerca del objeto creado.
   - *Prototype*: un objeto crea objetos sin conocer su clase o detalles de los mismos dando objetos prototipo a un objeto de creación que se encarga de ello haciendo copias de esos objetos prototipo.
   - *Singleton*: es el caso en el que se quiere garantizar que sólo se crea una instancia de una clase.
   - *Object pool*: gestiona un objeto para que pueda ser reutilizado cuando es computacionalmente costoso crear instancias de ese objeto.

3. **De partición**: dan una guía sobre como dividir los casos de uso y actores en clases.

   - *Layered initialization*: trata el problema que se plantea cuando se necesitan varias implementaciones de una clase común en la que además la lógica común se utiliza para decidir que clases especializadas hay que crear.
   - *Filter*: permite que un conjunto de objetos con la misma interfaz y que realizan distintas operaciones sobre datos puedan de un modo dinámico conectarse en el orden en el que se realizan las operaciones.
   - *Composite*: usado para construir objetos que se componen de otros objetos.

4. **Estructurales**: describen las diferentes formas de interacción entre objetos.

   - *Adapter*: es como un puente entre dos clases con interfaces incompatibles que permite que puedan relacionarse.
   - *Iterator*: es una forma de definir interfaces para hacer un recorrido de un conjunto de datos.

- *Bridge*: permite que una jerarquía de abstracciones y sus correspondientes implementaciones puedan cambiar por separado.
- *Facade*: es una forma de crear una interfaz unificada para manipular un conjunto de objetos.
- *Flyweight*: trata el caso cómo crear una sola instancia en vez de muchas cuando contienen la misma información. Está pensado para mejorar la eficiencia.
- *Dynamic linkage*: permite que un programa solicite cargar clases que implementan un interface conocido.
- *Virtual proxy*: si un objeto es caro de instanciar, se pospone la instanciación hasta que realmente se necesite el objeto. El patrón oculta esta inexistencia dando a los clientes un objeto proxy que implementa el mismo interface. Esto se conoce como *instanciación perezosa*.
- *Decorator*: extiende la funcionalidad de un objeto de un modo transparente a sus clientes utilizando una instancia de la clase original.
- *Cache management*: se tiene una copia de los objetos que son costosos de construir para acelerar el acceso a los mismos.

5. **De comportamiento**: Organizan y gestionan los comportamientos.

- *Chain of responsability*: permite que un objeto pase un comando a un objeto que es parte de alguna estructura sin saber que objeto lo va a ejecutar. Un objeto de la estructura puede ejecutar el comando o pasárselo a otro.
- *Command*: permite que un objeto pueda realizar una operación parametrizada de modo que se puede controlar las operaciones que se ejecutan, la secuencia, encolarlas, etc.
- *Little languaje*: si la solución a varios problemas se puede expresar como las combinaciones de una serie de operaciones del tipo creación o búsquedas en estructuras de datos y asignación de valores este patrón define un lenguaje con estas operaciones para resolver un conjunto de problemas.
- *Mediator*: modela una clase cuya responsabilidad es controlar y coordinar las interacciones de un grupo de objetos. Encapsula el comportamiento colectivo en una clase mediadora.
- *Snapshot*: es una vista de un objeto sin información temporal acerca de él. El objeto puede ser capturado o restaurado.
- *Observer*: sirve para detectar dinámicamente los cambios en un objeto y comunicar esos cambios a los objetos que dependan de él para que se actualicen.
- *State*: es útil cuando un objeto tiene que cambiar su comportamiento cuando cambia su estado. Cada estado queda encapsulado como un objeto separado.
- *Null object*: cuando no existe un objeto se denota con un puntero a *null*, el problema de esto es que supone añadir estructuras de control que comprueben si el objeto es null antes de hacer algo. La alternativa es definir un *null object*, que es un objeto que

no hace nada.

- *Strategy*: es una forma de seleccionar el algoritmo que se quiere emplear en una situación. La forma es tener una clase antecesora y en cada subclase se define un algoritmo distinto. La selección del algoritmo se realiza en función del objeto y puede cambiar dinámicamente en el tiempo.
- *Template method*: si se tiene un algoritmo que tiene una secuencia fija de pasos y uno o más de los pasos son variables este patrón lo que hace es poner cada paso variable en una función abstracta, después extiende la clase con una clase derivada que implementa la variación que se necesita.
- *Visitor*: una forma de realizar una operación es implementar la lógica de esa operación en los objetos involucrados. El patrón visitor es una alternativa a esta opción que implementa esa lógica en una clase separada *visitor*. Las ventajas son que no se complican las clases de los objetos involucrados y que se pueden usar varias clases visitor distintas.

6. **De concurrencia**. En la concurrencia existen dos tipos de problemas:

   - *Acceso a recursos compartidos*. el problema que se debe resolver son los interbloqueos.
   - *Secuenciación de operaciones*. el problema es garantizar que se ejecutan una serie de pasos en orden.

   Los patrones definidos para este apartado son:

- *Single threaded execution*: evita que se hagan llamadas concurrentes que accedan a recursos compartidos al mismo tiempo.
- *Guarded suspension*: impide que se llame a un método hasta que se cumpla una precondición.
- *Balking*: hace que se retorne de un método sin hacer nada en el caso de que el objeto no esté preparado para procesar la llamada.
- *Scheduler*: define una política sobre cuando se permite que un thread use un recurso compartido.
- *Read/Write lock*: soluciona el problema de lectores y escritores sobre un área protegida. Se puede leer concurrentemente pero se necesita acceso exclusivo para escribir.
- *Producer-Consumer*: gestiona la forma en la que se comunican de forma asíncrona un productor y un consumidor.
- *Two-phase termination*: es un modo de terminar la ejecución de un hilo. Lo hace comprobando una variable en una serie de puntos.

3.2. Diseño estructurado

Para realizar el diseño de una solución a un problema concreto se pueden utilizar diferentes enfoques, según se basen en los datos en la funcionalidad o en la interacción. En este tema se describen los dos métodos más utilizados: estructurado y orientado a objetos. El diseño estructurado con estilo procedimental (o funcional) descendente, en el cual se van refinando y dividiendo los módulos funcionales de la solución, se basa en la utilización de los elementos: secuencia, condición y repetición.

3.3. Diseño orientado a objetos

El otro tipo de métodos de diseño es el diseño orientado a objetos, que se basa en dividir jerárquicamente los elementos del diseño (objetos) y encapsular conjuntamente los datos con sus funciones de acceso, de forma que la interacción se realiza por paso de mensajes entre los objetos.

Es el paradigma[1] que se empezó a seguir a partir de la década de los 80. A continuación veremos los conceptos más importantes.

El concepto central en orientación a objetos es la **clase**. Una clase[2] es una forma de llamar por el mismo nombre un conjunto de cosas que comparten características comunes. Por ejemplo podríamos pensar en una clase llamada *NumeroComplejo*. Un número complejo está formado por dos números reales (uno es la parte real y otro la imaginaria) y esos dos números son los **atributos** que describen el objeto. Un **objeto** es un individuo concreto que pertenece a la clase *NumeroComplejo*, por ejemplo (3, 4i).

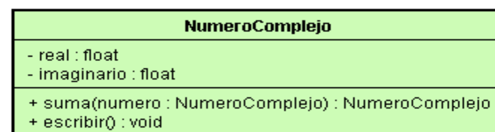


Figura 3.1: Representación en UML de una clase. Tiene Nombre de la clase, Atributos y Métodos

Otra cuestión es que posiblemente queramos tener rutinas manipular esos números: suma, escribir por pantalla, etc. Cada una de esas rutinas las llamamos **métodos**.

Definición de clase

[1] El concepto de objeto surgió a mediados de los 60 gracias a dos noruegos: *Dahl* y *Nygaard*, que desarrollaron el primer lenguaje orientado a objetos: Simula 67

[2] La palabra “clase” viene de clasificador

Primera definición (de andar por casa): Una clase es un conjunto de atributos ($\equiv$ datos) y de métodos ($\cong$ funciones $\cong$ procedimientos $\cong$ rutinas) con los que se manipulan dichos datos $\equiv$ Estructura de datos.

Las clases suelen representar cosas que existen en el mundo real. Por ejemplo: La clase *Estudiante* la podemos definir con los atributos: *Nombre*, *Apellido1*, *Apellido2*, *NIF*, etc. Podría tener los métodos: *irAula()* (Mueve al estudiante del punto A al punto B dada una ruta), *hallarRuta()* (este método averigua la ruta mencionada antes y es usado por el método anterior), etc.

Los métodos *públicos* son aquellos que pueden ser invocados por otros objetos. El conjunto de métodos públicos es lo que se conoce como la **Interfaz** del objeto. En este ejemplo, *irAula()* sería un método público, el método *hallarRuta()* sería un método *privado* que solo los objetos de la clase *Estudiante* serían capaces de invocar cuando necesitasen ir de un punto a otro y por tanto no formaría parte de su interfaz.

Segunda definición (un poco mejor): Una clase es algo que tiene responsabilidades. Pueden ser de dos tipos: De comportamiento, que se implementan con métodos y de estado, que se implementan con datos.

$$\text{Responsabilidades}\begin{cases}\text{Comportamiento (Métodos)}\\ \text{Estado (Datos)}\end{cases}$$

Una **responsabilidad** es como un contrato entre dos partes, es algo que una clase se compromete a hacer a cambio de otra cosa. En el ejemplo anterior la clase *Estudiante* tiene la responsabilidad de saber llegar al aula x. En general, cuando una clase A invoca un método de una clase B, A es responsable de que los parámetros que se pasen a B sean coherentes[3] (precondición) y B es responsable de que se devuelvan resultados correctos (postcondición).

Ejemplo: la clase A invoca un método de B. La clase B tiene el método *escribirTipoTriangulo(lado1, lado2, lado3)*, que puede ser Equilátero (3 lados iguales), Isósoceles (2 lados iguales) o Escaleno (todos los lados distintos). La clase A es responsable de que los lados sean números positivos y que la suma de las longitudes de dos cualesquiera de los lados sea mayor que la longitud del otro lado (en otro caso no sería un triángulo válido). La clase B es responsable de clasificar correctamente el triángulo y no se preocupa de que los datos de entrada sean válidos.

Encapsulamiento: una clase debe ocultar los detalles de cómo está implementada y en general toda aquella información que no sea imprescindible para manipularla. En el ejemplo del estudiante el método *hallarRuta()* no es conocido por las otras clases. El encapsulamiento reduce la complejidad.

[3] La definición de coherencia dependerá del dominio de aplicación

Ejemplo: un *Abonado* es una persona que *usa* un *Teléfono*. Cuando un abonado quiere *hablar* puede usar el teléfono para *llamar* a otras personas y *colgar* cuando acaba la conversación. Existen dos *tipos de* teléfonos: *Móvil* y *Fijo*. Un teléfono sea del tipo que sea se compone de un *Teclado*, un *Altavoz* y un *Micrófono*.

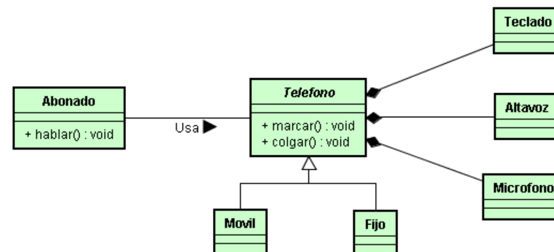


Figura 3.2: Ejemplo de clases y relaciones entre ellas

Asociación: Es una relación que existe entre dos clases. En el ejemplo anterior la clase *Abonado* está relacionada con la clase *Teléfono*. Una asociación se suele implementar en el código con un puntero dentro de un objeto de una clase a la otra. Una **composición** es un tipo de asociación entre clases del tipo *es parte de*. Un teclado, un altavoz y un micrófono son parte de un (y solo un) teléfono.

Herencia: Es una relación entre clases cuando una *es un tipo de* la otra. En el ejemplo, las clases *Móvil* y *Fijo* son tipos de teléfonos. Se dice que la clase *Teléfono* es padre de *Móvil* y *Fijo*. La herencia permite encapsular en una clase antecesora los aspectos comunes de las clases hijas.

Se dice que una clase es **abstracta** cuando no se pueden crear instancias de objetos partiendo de ella. En el ejemplo todos los teléfonos son o fijos o móviles, así que no sería posible crear un objeto de la clase Teléfono porque hay que concretar de qué tipo sería. La finalidad de Teléfono es especificar la parte compartida de la interfaz.

Una **colaboración** es una solicitud hecha por un objeto a otro. Para identificarlas, se debe examinar cada par clase-responsabilidad con el fin de determinar si es necesario que la clase interactúe con otra/s para llevar a cabo esa responsabilidad. En el ejemplo anterior un *abonado* debe usar una instancia de la clase teléfono para poder hablar con otro abonado.

Polimorfismo: Es la capacidad de un objeto de responder de formas distintas al mismo mensaje. Ejemplo: Un teléfono móvil para establecer una llamada tiene que hallar y reservar una banda de frecuencias libre para comunicarse con la torre de telefonía más cercana mientras que un teléfono fijo se limita a mandar un mensaje a la centralita. La operación (*marcar()*) es la misma, pero cada hijo de la clase Teléfono implementa esa misma interfaz del modo adecuado a sus características.

3.4. Validación y confirmación del diseño

Para cerrar un paso en la fase de diseño es necesario hacer la validación de los resultados obtenidos y comprobar si cumplen con las especificaciones de requisitos que eran la entrada a esta fase. Se trata de un conjunto de actividades para garantizar que se esta construyendo el producto correcto de la manera adecuada. Cada una de las actividades del diseño deben estar reflejadas en los planes del mismo, estos planes se actualizaran cuando sea necesario para adaptarse a los cambios que vayan surgiendo pero cada cambio deberá ser revisado y aprobado.

Debe existir un procedimiento de control de diseño que especifique cómo se planifica y desarrolla. Por otra parte, la información de entrada al diseño, que es la resultante de la fase anterior debe incluir además de los requisitos del usuario cosas tales como requisitos legales y regulaciones que se apliquen al proyecto. Las entradas al diseño deben tener en cuenta cualquier modificación del contrato. Los resultados del diseño deben ser documentados de forma que se pueda hacer una verificación y validación de los mismos.

3.4.1. Revisión del diseño

Durante el proceso de diseño se deben planificar algunas revisiones formales del trabajo que se va realizando. Los participantes son todas aquellas personas involucradas en el diseño que se esté revisando, así como representantes del cliente. Cuando finalice la revisión se debe redactar un documento con el resultado de la revisión y las partes afectadas. La forma de revisar el diseño estará documentada en el procedimiento de control del diseño.

3.4.2. Verificación del diseño

Las actividades que se realizan son:

- Realización de cálculos alternativos.
- Comparación con diseños preliminares si es aplicable.
- Pruebas y demostraciones
- Revisión del diseño.

3.4.3. Validación del diseño

Garantiza que el producto cumple con los requisitos definidos por el cliente. Se realiza después de la verificación si esta fue satisfactoria. Una validación se realiza en un entorno operativo normal. También esto se realizará conforme al procedimiento de control del diseño.

3.5. Documentación: especificación del diseño

Evidentemente uno de los puntos más importantes en la documentación es la especificación del diseño. Dado que el diseño tiene una gran componente de invención, es necesario dejar muy claramente documentadas las decisiones y elementos utilizados en el diseño. Además el diseño

será el elemento clave para la fase siguiente de implementación que debe seguir fielmente los dictados del diseño.

Para dejar constancia de los diseños se deben utilizar lenguajes lo más formales posible, como tablas, diagramas y pseudocódigo, que en algunos casos pueden permitir incluso la utilización de herramientas para automatizar parcialmente el proceso de construcción del código en la siguiente fase de implementación.

Tutorial posterior

Resumen de contenidos

El diseño es la fase que sigue a la especificación y el análisis. Consiste en añadir a la representación anterior los detalle necesarios para pasar a la implementación. Al igual que en el capítulo anterior, existen 2 tipos de diseño: orientado a objetos y estructurado. También existen diferentes formas de realizarlo en función de características del sistema (tiempo real, distribuido, etc.)

Diseño arquitectónico: independientemente del tipo de diseño (estructurado u orientado a objetos) existe una primera fase que consiste en la división del sistema en subsistemas. Los apartados a tener en cuenta son:

1. *Descomposición estructural*: Descomposición en subsistemas
2. *Intercambio de información entre subsistemas*: Hay dos opciones: con una base de datos central o por medio de mensajes.
3. *Formas de realizar el control*: Centralizado o basado en eventos.

Sistemas distribuidos: Son un tipo especial de sistema que merece un tratamiento aparte debido a que parece ser la tendencia actual. Hay varios tipos de arquitecturas:

1. *Arquitectura cliente-servidor*: puede ser de dos o de tres capas.
2. *CORBA*: es un sistema de objetos distribuidos. Utiliza IDL como lenguaje de definición de interfaces. **{ATENCIÓN: CORBA No se trata en este curso,** *ver en [Pre01, cap. 28]***}**

Diseño estructurado

Es el análisis clásico. Esta basado en el flujo de los datos a través del sistema. En una primera fase se hace un diseño preliminar y posteriormente un diseño detallado. *Objetivos*: Comprensión, mantenibilidad, facilidad de pruebas e integración con otras aplicaciones. *Principios*: Abstracción, modularización, independencia, arquitectura, ocultamiento de la información y refinamiento paso a paso. Sus elementos importantes son:

1. *Módulo*: el árbol de módulos define las relaciones entre los mismos.
2. *Tabla de interfaz*: especifica como son las comunicaciones inter-módulos.

3. *Cohesión*: un módulo es cohesivo si sus tareas se realizan con independencia del sistema pero relacionadas entre sí.
4. *Acoplamiento*: media de la complejidad de las interfaces de un módulo.
5. *Diagrama de estructuras*: es un método de descomposición funcional donde el bloque básico es el módulo. Es un árbol o diagrama jerárquico. Se definen en él: bucles, alternativas y subrutinas.
6. *Heurísticas de diseño*: consejos a aplicar.
7. *Tipos de flujos de información*: flujos de transformación y de transacción.

Diseño detallado: es una especificación con mayor nivel de detalle de las estructuras de datos y algoritmos.

Otras *notaciones* que forman parte de metodologías pensadas en torno al diseño estructurado (no se tratan en este curso) son las siguientes:

1. *Diagramas HIPO*: muestran entradas, salidas y funciones. Muestran lo que hace el sistema pero no el cómo.
2. *Diagramas de Warnier-Orr*: son una representación jerárquica de los programas, sistemas y estructuras de datos. Puede representar iteraciones, secuencias y alternativas.
3. *Diagramas de Jackson*: produce un programa a partir de su especificación. Las entradas y salidas son secuenciales al igual que el proceso.

Diseño orientado a objetos

Como se ha dicho antes, es la otra forma de hacer el diseño. Se incluye también en este resumen el diseño arquitectónico por ser parte integrante de la metodología, aunque lo dicho en el apartado anterior dedicado al tema es igualmente válido. Hay dos partes:

1. Diseño del sistema = (Diseño arquitectónico)
2. Diseño de objetos = (Diseño detallado)

Patrones: Consisten en la identificación de problemas usuales en el análisis, diseño, codificación, etc e incluir una plantilla de solución.

Frameworks: Son otro mecanismo de reutilización pero un *framework* se puede componer de varios patrones. Generan aplicaciones para un dominio.

Validación y confirmación del diseño

Son un conjunto de actividades que garantizan que el producto se está construyendo de la forma adecuada. Se compone de la revisión, la verificación y la validación del diseño.

Documentación

Se incluye una plantilla para redactar el documento correspondiente al igual que en la fase anterior.

Ejercicios y actividades propuestas

1. ¿Cuáles son las principales diferencias entre el diseño estructurado y el orientado a objetos?
2. ¿Por qué el diseño arquitectónico es el primer paso?
3. ¿Cuáles son los tipos de modelos cliente-servidor de tres capas?
4. *{Este ejercicio no se tratará en este curso}* ¿Qué es necesario para que un objeto pueda invocar los servicios de otro en CORBA?
5. *{Este ejercicio no se tratará en este curso}* ¿Cuál es la secuencia de pasos que ocurre cuando un cliente utiliza los servicios de un servidor en CORBA? ¿Podría el servidor ser cliente de su cliente?
6. ¿Qué nivel de acoplamiento hay entre dos módulos A y B donde A utiliza una función de B pero ambos manipulan el mismo conjunto de variables globales?

Capítulo 4

Fase de implementación

4.1. Técnicas de depuración

Durante el desarrollo de la fase de implementación se deben realizar dos tipos de labores relacionadas con la comprobación del código obtenido. La primera consiste en elaborar los conjuntos de prueba para los diferentes módulos que se programan. La segunda consiste en comprobar si los módulos cumplen las especificaciones de diseño con esos conjuntos de prueba. También se puede incluir dentro de este proceso la comprobación de corrección en la ejecución de los módulos y evitar posteriormente la aparición de errores frecuentes como pérdidas de memoria, comportamientos extraños ante entradas de datos imprevistas, etc.

4.2. Documentación del código

Además de la documentación incluida dentro del código fuente correspondiente, es necesario generar los manuales técnicos y de referencia adecuados, así como la parte inicial correspondiente del manual de usuario (que se finalizará en la fase de entrega). Estos manuales se deben ir escribiendo simultáneamente al código y se deben actualizar en concordancia con los cambios en el mismo.

Esta documentación es esencial para las fases de pruebas y de mantenimiento, así como para la entrega final del producto. Por tanto se deben mantener actualizadas y funcionales desde el primer momento.

Además de las recomendaciones que se han dado para lenguajes específicos vamos a ver de que modo se puede hacer la documentación interna de un programa (comentarios). Esta documentación es importante de cara al mantenimiento futuro del programa, tanto si lo hacen terceras personas como si lo hace el encargado de programar el código fuente del módulo. Además de estos comentarios en el código se extraen los esbozos para los manuales técnicos y de referencia.

4.2.1. Tipos de comentarios

Distinguiremos tres tipos de comentarios desde el punto de vista de la parte que comentan y dónde se localizan.

- **Directorio**: Son los situados al principio de cada módulo. La información que contiene es: Lista de funciones o clases implementadas en el módulo, Versión, Fecha, Programador, *Copyright* e Interfaces con otros módulos.
- **Prólogo**: Cumple la misma función que el directorio pero para cada función o método. Indica la siguiente información: Finalidad, Comentarios acerca de las variables significativas, Descripción general del funcionamiento e Información para la reutilización (efectos colaterales, variables globales que utiliza, etc.)
- **Explicativo**: Aclaraciones acerca de partes del código. Hay dos tipos: de línea y de bloque.

  Las ventajas de los comentarios de bloque es que están localizados en un punto concreto y no dispersos por varias líneas entre el código, con lo que además son fáciles de modificar. Los comentarios de una línea deben ponerse en la misma línea del código que se esta comentando y se usan para explicar sentencias complicadas.

Los tipos de comentarios según cómo están redactados son estos cinco (de peor a mejor):

- **Repetir el código**: Consiste en expresar el código de otra forma pero sin añadir información significativa. Este tipo de comentarios son innecesarios.
- **Explicación del código**: Ayudan a comprender el código. Es mejor reescribir el código de modo que sea más inteligible que utilizar este tipo de comentarios.
- **Marcadores**: Son notas temporales para la gente que trabaja en el proyecto. Se eliminan cuando el proyecto finaliza.
- **Resumen del código**: Explica en pocas palabras la función de un trozo de código.
- **Descripción de la finalidad**: Explica el por qué. Es el más fácil de entender.

4.2.2. Consideraciones generales

- Los comentarios deben ser entre el 25 % y el 33 % de las líneas de un módulo.
- No se deben comentar cosas evidentes. Ejemplo:

```
if (x<5) i+=2; // Suma 2 a "i" cuando "x" es menor que 5
```

- Se debe comentar más el *cómo* y no el *qué*.
- Los datos se deben comentar cuando el nombre no sea suficiente como para que se entienda. Si un dato se compone de otros más pequeños aplicar esta misma regla recursivamente caso de ser necesario.

Tutorial posterior

Resumen de contenidos

En teoría es posible realizar la implementación de un modo totalmente automatizado partiendo del diseño. Como esto a veces es difícil de llevar a la práctica, se dan una serie de consejos

para pasar el diseño a código fuente y como redactar este código fuente en un conjunto de lenguajes.

Traducción del diseño a la implementación

Ingeniería inversa: Consiste en averiguar las especificaciones de un sistema a partir de su funcionamiento. Se dan algunas heurísticas para averiguar los casos de uso de un sistema: definiciones de clases, creación y destrucción de objetos, invocación de métodos, herencia e implementación de asociaciones.

Estilo de programación orientado a objetos: Son un conjunto de consejos para conseguir: reusabilidad, reconocer la herencia, construir métodos extensibles, código robusto y trabajo en grupo eficaz.

Normas para programadores en C: Son un conjunto de normas que cubren todos los apartados de la codificación, tanto del tamaño de los ficheros, como de la elección de nombres (quizás lo más importante), indentación, etc.

Normas para programadores en C++: Es una recopilación similar para el lenguaje C++. Es más extenso debido a la complejidad del lenguaje. Estas normas fueron compiladas por *Ellmentel Telecomunications System Laboratories*.

Normas para programadores en Java: Es un resumen de las normas publicadas por SUN para el lenguaje Java.

Técnicas de depuración

Consiste en un conjunto de técnicas para descubrir errores de codificación. Incluye también un pequeño apartado para procesos concurrentes.

Documentación del código

Al igual que el diseño el análisis producen documentos según una plantilla estándar, el código tiene que tener una documentación interna. Se distinguen los tipos de comentarios (Directorio, prólogo y explicativo), donde se colocan, que información se incluye en cada uno de ellos y la forma de redacción de la información.

Ejercicios y actividades propuestas

1. Suponga que tiene este código:

```
char *posicion(char *cad,char *cad2) {
for (char *c1=cad,char *c2=cad2; *c1; c1++) {
for (char *c3=c1,char *c4=c2; *c3 && *c4 &&
(*c3==*c4); c3++, c4++)
if (!*c4) return c1;
```

```
}
}
```

a) ¿Qué propósito tiene?

b) ¿Es correcto?. Si no es así, ¿a qué se debe?

c) Redacte de nuevo el código de modo que se pueda entender su propósito y funcionamiento de un vistazo.

- Asigne nombres significativos a las variables y a la función.
- Escriba comentarios.
- Ponga la indentación correcta.
- Si es necesario cambie la estructura de la función.

2. Suponga que tiene un programa en C++ que calcula un fractal y lo dibuja como una imagen en la pantalla. Para estas funciones utiliza un módulo con una estructura de datos que define los números complejos y las operaciones relativas a ellos: Producto, Suma, Resta, Potencia. A cada punto de la pantalla se le aplica un algoritmo, lo cual estaría en otro módulo. Se tendría otro módulo con las funciones relativas a los gráficos: Dibujar matriz de puntos, Zoom, etc.
   Redacte la documentación de cada módulo del programa: prólogo y comentarios antes de cada función (haga las suposiciones que considere oportunas sobre que funciones existirían).

Capítulo 5

Fases de pruebas

5.1. Técnicas y métodos de prueba

Existen diferentes técnicas para realizar las pruebas de verificación del software en diferentes fases. La mayoría hacen uso de las características de modularidad del diseño e implementación para ir integrando de forma ascendente o descendente los módulos de diferentes niveles.

5.2. Documentación de pruebas

Como en toda fase, todos los elementos preparados para las pruebas y los resultados de las mismas deben ser documentados y adjuntados a cada versión correspondiente de los productos de las fases anteriores. Esta forma de proceder evitará la repetición de pruebas y facilitará la preparación de nuevas pruebas a partir de otras anteriores.

Plan de pruebas

Son un conjunto de actividades que pretenden demostrar que el producto cumple con lo estipulado en el contrato. En el plan de pruebas se especifica:

1. *Identificador del plan*: debería ser una palabra que lo identificase con su alcance. Debe incluir además la versión y fecha.
2. *Alcance*: tipo de prueba y propiedades del software a ser probado.
3. *Items a probar*: configuración a probar y condiciones que se deben satisfacer para ello.
4. *Estrategia*: técnica y herramientas que se van a utilizar. Hay que indicar el número mínimo de casos de prueba que se van a realizar y el grado de automatización tanto en la generación de casos de prueba como en la ejecución
5. *Categorización de la configuración*: condiciones bajo las cuales el plan debe ser:
    - Suspendido: Si se presentan demasiados defectos o fallos.
    - Repetido.

   - Dado por terminado: Si se cumple el número mínimo de casos de prueba.

6. *Documentación*: Según el estándar IEEE 829-1983 hay que producir los siguientes documentos:
   - Plan de prueba.
   - Especificación de los requerimientos para el diseño de los casos de prueba.
   - Casos de prueba. Para cada uno se especifican: el procedimiento de prueba y descripción del ítem a probar.
   - Bitácora de pruebas.
   - Informe de incidentes de prueba.
   - Resumen de pruebas.
7. *Procedimientos especiales*: grafo de las tareas necesarias para preparar y ejecutar las pruebas, así como habilidades necesarias.
8. *Recursos*: pueden ser muy variados
   - Características de hardware y software.
   - Software necesario para hacer las pruebas.
   - El software bajo prueba.
   - Configuración del software de apoyo.
   - Recursos humanos necesarios para el proceso.
   - Requerimientos especiales: actualización de licencias, espacio de oficina, tiempo de máquina, seguridad, etc.
9. *Calendario*: hitos del proceso y grafo de dependencias.
10. *Riesgos*: riesgos del plan, acciones para mitigarlos y planes de contingencia.
11. *Responsables*: lista de personas y tareas asignadas.

Existen cuatro tipos de pruebas que deben superarse

- *Pruebas funcionales*: se realizan con los datos de entrada normales en el uso diario del sistema. Se comprueban sobre todo valores en los límites de las variables y un poco más allá de los límites. También algunos valores especiales.
- *Pruebas de desempeño*: miden el rendimiento del sistema: tiempos de respuesta, utilización de la memoria, tráfico generado en las comunicaciones, etc. Estas pruebas nos van a indicar dónde están los cuellos de botella del sistema.
- *Pruebas de tensión*: se trata de sobrepasar los límites de tolerancia del sistema de varias maneras, por ejemplo: conectar más usuarios al sistema de los permitidos, si se tienen que gestionar x mega-bits por segundo de una línea de comunicaciones probar a aumentar a $x+1$, etc.
- *Pruebas estructurales*: es un análisis de la lógica interna de un programa.

Tutorial posterior

Resumen de contenidos

Verificación y validación

La Verificación y validación se debe llevar a cabo a lo largo de todo el ciclo de vida.

Definición 1 (Verificación) *Comprobar si el producto se está construyendo correctamente.*

Definición 2 (Validación) *Comprobar si estamos construyendo el producto que se nos ha pedido.*

Se abordan un conjunto de actividades y tareas para garantizar que se da una respuesta positiva a esas dos cuestiones. Uno de los puntos importantes es que la *V&V debe ser independiente*, es decir, no pueden hacerlo las mismas personas que están construyendo el producto. Esta independencia se debe dar tanto en la gestión como en la financiación. Las tareas de V&V se contemplan desde tres perspectivas: sistemas genéricos, sistemas expertos y software reutilizado, y en todas y cada una de las etapas del ciclo de vida.

Técnicas y métodos de prueba

Se trata en esta sección de realizar las pruebas que den una garantía acerca del código. Se introducen los conceptos de *Error***,** *Defecto* y *Fallo*. Se definen un conjunto de principios que debe seguir toda prueba:

Casos de prueba: Es un conjunto de entradas y salidas esperadas para ellas. Existen dos tipos de pruebas: de caja blanca y de caja negra. Existe un método para crear casos de prueba para casos de uso.

- Pruebas de caja blanca. Técnicas y medidas: grafo de flujo, complejidad ciclomática y criterios de cobertura del grafo.
- Pruebas de caja negra. Técnicas: particiones o clases de equivalencia y análisis de valores límite.

***Cleanroom*:** Es una metodología de desarrollo pensada para producir software muy fiable.

Revisiones sin ejecución de código: *wallthroughs*, inspecciones y auditorías,

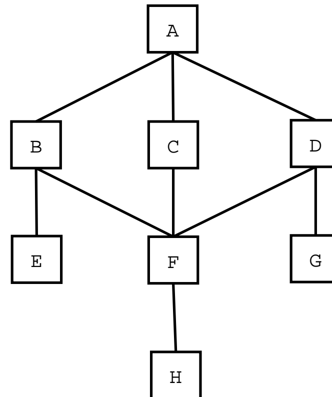
Calidad del software: El modelo de McCall tiene en cuenta una serie de factores de calidad.

Documentación de pruebas

Como en los demás capítulos este es el apartado donde se pone sobre el papel el trabajo realizado con un documento llamado **plan de pruebas** que es un documento que pretende demostrar que el producto cumple con lo estipulado en el contrato.

Ejercicios

1. En el siguiente diagrama de módulos:



   a) ¿Cuántos *controladores* y cuántos *resguardos* son necesarios con **integración ascendente**?

   b) Si se considera que el módulo **B** hay que probarlo cuanto antes, porque es problemático, ¿cuál es el orden de integración hasta llegar a **B** con **integración ascendente**. Si en algún caso hay más de un módulo para integrar escoger por orden alfabético.

   Solución

   a) Controladores: 7. Resguardos: 0. En la integración ascendente no hay resguardos.

   b) Orden: E, H, F, B, ... (El resto de los módulos ya no importa porque hemos llegado a B)

2. Sea un algoritmo que para dos intervalos de fechas consecutivas definidos por sus fechas de comienzo y fin (I_1, F_1 para el primero y I_2 y F_2 para el segundo) devuelve verdadero (V) si alguna fecha coincide en ambos intervalos, incluidas las de comienzo y fin, y falso (F) en caso contrario. Se pide: programación en pseudolenguaje, grafo de flujo, los caminos independientes, la complejidad ciclomática y los casos de prueba.

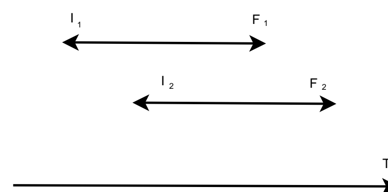


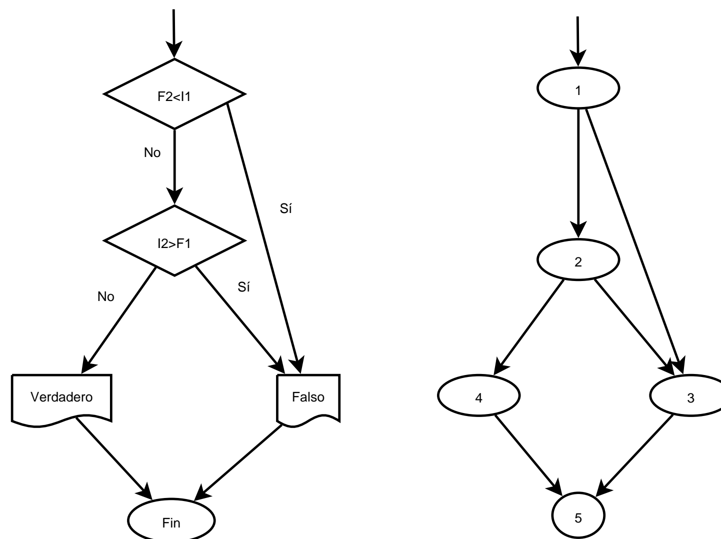
Figura 5.1: Ejemplo de intervalos de fechas

Solución Algoritmo:

a) Programación en pseudolenguaje:

```
if F2 < I1
    then Falso
    else if I2 > F1
        then Falso
        else Verdadero
```

b) Diagrama y Grafo de flujo



c) Caminos independientes:

- Camino 1: 1-3-5
- Camino 2: 1-2-3-5
- Camino 3: 1-2-4-5

d) Complejidad ciclomática:

V(G) = Numero de aristas- numero de nodos +2 = 6-5+2= 3

O bien:

V(G) = Numero de nodos predicado (con condición) +1 = 2+1=3

e) Casos de prueba:

- Camino 1: $F_2 = I_1 - 1, I_1 = F_2 - 1$

- Camino 2: $F_2 = F_1 + 2, I_1 = F_2 - 1$
- Camino 3: $F_2 = I_1, I_1 = F_2 - 1$

3. ¿Cuáles son los motivos por los que es conveniente que las actividades de V&V sean realizadas por un equipo independiente?
4. ¿Cuándo es mejor hacer la V&V, después de construir el producto, antes, después de una etapa concreta, ...?
5. Dado el programa de la figura 5.2:

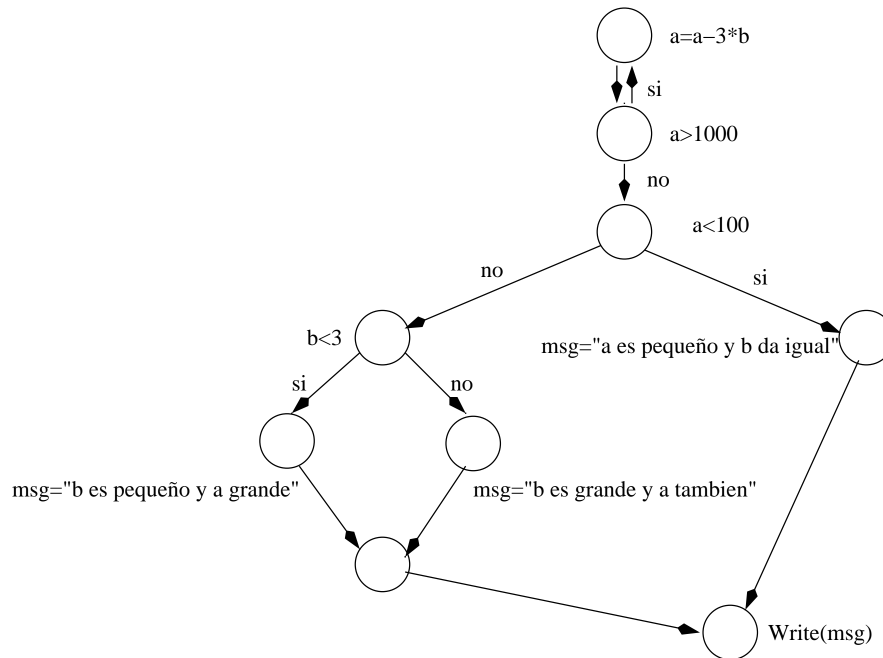


Figura 5.2: Programa

- Represéntelo en algún lenguaje de programación.
- Halle los valores de a y b para cobertura de sentencias, decisiones, condiciones y decisión/condición.
- ¿Hay algún bucle? ¿De qué tipo? ¿Cuántas veces conviene ejecutarlo?

Capítulo 6

Fase de entrega y mantenimiento

6.1. Finalización del proyecto

Dentro del ciclo de vida del software, y como en cualquier actividad de ingeniería, es necesario tener presente la finalidad del desarrollo y la obligación de entrega del producto acabado al cliente. Una vez que el sistema ha pasado las pruebas se trata de entregárselo al cliente y ponerlo en condiciones de empezar a funcionar. La estrategia de implantación se decide desde la fase de especificación y se tiene en cuenta tanto la cobertura global como la geográfica. Es recomendable hacer las siguientes consideraciones:

1. Número de copias de cada parte del software que se va a entregar.
2. Tipo de medios para cada parte del software, incluyendo formato y versión en lenguaje legible por humanos.
3. Documentación de requerimientos estipulados (manuales de usuario).
4. Derechos de propiedad y asuntos de licencia, señalamientos y acuerdos.
5. Custodia de los programas del sistema, incluyendo planes de recuperación y de contingencia por desastres.
6. Instalación.
7. Las responsabilidades y obligaciones del personal encargado de informática y de los usuarios finales deben definirse teniendo en cuenta lo siguiente:

    - Acceso a las instalaciones del nuevo usuario.
    - Disponibilidad y acceso a los sistemas y equipo del usuario final.
    - Disponibilidad de personal experto en las áreas correspondientes.
    - Las necesidades de validación se deben determinar.
    - Se debe validar el funcionamiento de todo el sistema después de cada instalación.
    - Un procedimiento formal para aprobar cada instalación al complementarla.

6.1.1. Aceptación

Se dice que el proyecto ha finalizado cuando todas las actividades técnicas han terminado o se ha agotado su plazo y se han facturado todos los conceptos al cliente (se hayan cobrado o no). Para evitar problemas en el momento en el que se pide al cliente su aceptación se debe redactar un documento que especifique claramente los requisitos (lógicamente este documento se redacta en la fase de especificación). Una vez que el cliente lo firma acepta que está conforme.

6.1.2. Informe de cierre del proyecto

Es un resumen de lo sucedido, de si se han conseguido los objetivos y en caso negativo de las razones para ello. De un modo esquemático, la documentación generada debe contener la siguiente información:

- Balance de ingresos y gastos.
  - Valores positivos: Gastos superiores ->Menos beneficio.
  - Valores negativos: Ahorro ->Más beneficio.
- Informes de situación finales.
  - Informe económico.
  - Informe de situación final. Se redacta en lenguaje no técnico. Incluye:
    - Datos del proyecto: Nombre, duración, etc.
    - Resumen de las incidencias: Modificaciones, problemas, soluciones, sugerencias futuras, etc.
- Lista de documentación generada.
- Lista de productos generados.

La documentación es también un modo de preservar el conocimiento de la empresa independientemente de la permanencia de los empleados, asimismo detecta tendencias en el tipo de requisitos demandados, tipos de errores más comunes, etc.

6.1.3. Indicadores del proyecto

Son un conjunto de medidas para evaluar de un modo objetivo los resultados.

1. *Indicadores económicos*:
   - Facturación del proyecto.
   - Margen del proyecto: Diferencia entre ingresos y costes, en valor absoluto y en porcentaje.
   - Beneficio del proyecto.

- Coste por hora trabajada.
- Precio de venta de la hora trabajada.
- Componentes del coste del proyecto. Son los porcentajes del coste total del proyecto dedicados a cada uno de estos apartados.
  - Valor relativo del esfuerzo propio.
  - Valor relativo de las subcontrataciones.
  - Valor relativo de los suministros.

2. *Indicadores financieros*:
   - Porcentaje de endeudamiento externo (ajeno): Es el montante que hay que afrontar en caso de impago. Es el porcentaje dedicado a subcontrataciones, suministros, viajes y gastos varios.
   - Porcentaje de endeudamiento interno (propio): Son los gastos que en caso de impago la empresa debería cubrir. Es el complementario del anterior. Porcentaje de endeudamiento externo + Porcentaje de endeudamiento propio = 100 %.
   - Valor actual neto del proyecto: es un método para calcular el valor presente de un determinado número de flujos de caja futuros teniendo en cuenta una tasa de interés i.
   $$VAN = \sum_{i=1}^{n} \frac{F_i}{(1+r)^i} \tag{6.1}$$
   Donde: F_i son los flujos monetarios del proyecto, n es el número de años o meses desde que se produjo el flujo monetario y r es la inflación anual o mensual acumulada desde entonces.
   - Tasa interna de retorno (tasa de rendimiento interno (r) : es el interés que hace que el VAN sea 0. Se puede utilizar para comparar diferentes inversiones. Se escoge la que tiene el TIR más alto.
   $$VAN = -I + \sum_{i=1}^{n} \frac{F_i}{(1+r)^i} = 0 \tag{6.2}$$
   Es fácil comprobar que
   $$r = \frac{-I + \sum_{i=1}^{n} F_i}{\sum_{i=1}^{n} i \times F_i} \tag{6.3}$$
3. Indicadores de ocupación laboral. Indican la cantidad de personal para cada proyecto, su tasa de ocupación y como consecuencia si sobra o falta personal.
   - Carga de trabajo: Número de horas para realizar un trabajo.
   - Número de personas necesarias para el proyecto.
   - Índice de ocupación.
4. Indicadores de gestión. Hay cuatro indicadores de la calidad de este apartado:

- Plazo de ejecución.
- Coste (global y por partidas)
- Margen.
- Contingencias.

6.2. Planificación de revisiones y organización del mantenimiento

Adicionalmente a las revisiones durante el proceso de desarrollo inicial del software, también hay que considerar las posibles modificaciones y revisiones como consecuencia del mantenimiento y actualización del mismo.

También hay que considerar la posibilidad/certeza de que el cliente solicite cambios posteriores no previstos inicialmente, es decir, hay que estar preparados para esta posibilidad.

6.3. Recopilación y organización de documentación

La documentación global del producto es una parte integral del mismo. En esta fase es necesario reunir todos los documentos generados y clasificarlos según el nivel técnico de sus descripciones. Es muy importante distinguir entre la documentación orientada a futuros desarrollos o modificaciones de mantenimiento (documentación de diseño, implementación y pruebas, manual técnico, manual de referencia), y la documentación de uso y aplicación (introducción de uso rápido, manual de configuración, manual de usuario, manual de interfaz). Cuando ha finalizado el proyecto se debe tener una documentación útil para el mantenimiento posterior y para la operación normal por parte de los usuarios. Esta sección es una enumeración bastante exhaustiva de esta documentación.

6.3.1. Motivos de la documentación

Hay dos motivos para contar con una buena documentación:

1. Facilita el mantenimiento porque:
   - Ayuda a localizar dónde hay que modificar o añadir código.
   - Los mantenedores necesitarán menos tiempo para comprender cada módulo.
   - Supone una forma de comunicación entre profesionales, sobre todo si la forma en la que se ha realizado sigue un estándar; por ejemplo, todo el mundo sabe interpretar un diagrama de clases de UML.
   - Una buena documentación facilita que el mantenimiento lo puedan realizar terceras empresas o personas.
2. Facilita la auditoría. Una auditoría es un examen del sistema que se puede hacer desde varios puntos de vista. Por ejemplo, una auditoría permite saber si se están cumpliendo plazos o las normas de la empresa en la redacción del código.

6.3.2. Directrices que se deben seguir para la redacción de un documento

Muchas organizaciones tienen un *programa de documentación*, que es un procedimiento estandarizado de redactar los manuales técnico y de usuario. Los procedimientos de documentación deben ser estandarizados para que de esta forma la comunicación sea rápida, no ambigua y reduzca costes de personal, es decir, se debe huir de las “florituras” lingüísticas propias de la literatura haciendo énfasis en la claridad y la concisión.

En general todos los tipos de documentos deben tener un estilo homogéneo, para ello se debe seguir un formato que tenga estos elementos:

1. *Preámbulo* con: Autor(es) del manual, fecha, versión y revisores.
2. *Índice* con secciones y subsecciones. A la hora de empezar a redactar un documento se hace un primer esbozo de las secciones que se deben cubrir. Posteriormente, a medida que se van escribiendo se pueden hacer cambios, añadir o quitar subsecciones, pero no en los puntos principales.
3. Se debe seguir un conjunto de normas respecto a dos aspectos:
   - Respecto al *contenido*. P. ej: Se pondrán ejemplos explicativos, se tenderá a la brevedad, etc.
   - Respecto a la *forma*. P. ej: Tipo de letra “*Times new roman 11pt*”, los párrafos se indentarán con cuatro espacios, etc.
4. Se deberían usar las mismas herramientas de proceso de textos para todos los documentos, preferiblemente de alguna que haya demostrado su continuidad en el tiempo, por ejemplo, LaTeX, porque nunca se sabe cuanto durará el mantenimiento (quizás décadas).
5. Se cuidará en especial la claridad de expresión; los manuales deben tener ejemplos.
6. Los diagramas deben tener distribuidos los elementos de un modo lógico, no se deben poner demasiados elementos en un único diagrama.
7. Al final debe existir un *glosario* de términos.
8. *Completitud*: Se documentarán todas las partes del sistema.
9. *Consistencia interna*: No debe haber contradicciones entre diferentes partes del documento. Esto suele ocurrir cuando se hacen cambios durante el mantenimiento y no se revisa correctamente toda la documentación.

6.3.3. Tipos de documentos

Existen tres tipos de documentos:

- Los dirigidos a usuarios, que se centran en la operación del sistema.
- Los dirigidos al mantenimiento.
- Los dirigidos al desarrollo.

Veremos ahora una lista de documentos. En un proyecto no es necesario que estén todos, sobre todo en proyectos pequeños donde posiblemente no se haga un análisis de riesgo o un análisis coste-beneficio.

La información que se recoge en estos documentos deberá ser actualizada durante el mantenimiento o cuando se hagan cambios en los requisitos durante el desarrollo para conservar la consistencia entre lo que dice la documentación y la realidad.

1. *Estudio de viabilidad*: Es una evaluación de un proyecto, requisitos y objetivos y sus posibles alternativas.
2. *Análisis de riesgo*: Identifica puntos vulnerables, sus posibles consecuencia, su importancia, causas y posibles soluciones. Este documento será actualizado en cada fase de desarrollo.
3. *Análisis costo-beneficio*: Es una evaluación de lo que va a costar el sistema y los beneficios esperados. Uno de los usos de este análisis es la selección de proyectos de desarrollo. Cada modificación que se proponga al sistema tendrá un impacto en este análisis, con lo que será necesario revisarlo.
4. *Informe de decisión de sistemas*: Es el lugar donde está la información para la toma de decisiones de desarrollo. Incluye: Necesidades que se necesita cubrir, Objetivos de cada uno de los hitos, Riesgos, Alternativas, Coste-beneficio y Planes de administración o justificación de decisiones.
5. *Plan de proyecto*: Define los objetivos y actividades para cada fase. Incluye estimaciones de recursos y objetivos de los hitos a lo largo de todo el ciclo de vida. Es donde están definidos los procedimientos para diseño, documentación, notificación de problemas y control del cambio. Se detallan las herramientas y técnicas seguidas.
6. *Requisitos funcionales*: Son una lista de los servicios que suministra el sistema a los usuarios, que estarán definidos en términos cualitativos y cuantitativos. También están los requisitos de seguridad y control interno.
7. *Requisitos de datos*: Enumeración de los tipos de datos, su descripción y modo de captura. Deben identificarse especialmente los datos críticos para tener localizados los riesgos que de ellos se deriven y poder idear mecanismos de recuperación y seguridad.
8. *Especificaciones de sistema/subsistema*: Es un documento para los analistas de sistemas y programadores. Son los requisitos, entorno operativo, características de diseño, y especificaciones de seguridad y control para el sistema o subsistemas.
9. *Especificaciones del programa*: Es otro documento dirigido a los desarrolladores. Son los requisitos, entorno operativo y características de diseño de cada uno de los ejecutables.
10. *Especificaciones de la base de datos*: Descripción física y lógica de la base de datos junto a las especificaciones de seguridad y control.
11. *Pruebas*: Estrategia de pruebas del sistema, con especificaciones detalladas, descripciones y procedimientos, así como datos de prueba y criterios de evaluación.
12. *Manual de usuario*: Describe las funciones del sistema. No debe usar jerga técnica.
13. *Manual de operación*: Describe el software y el entorno operativo en el que puede funcio-

nar el software.

14. *Manual de mantenimiento*: Información acerca del código fuente, sistemas y subsistemas para que el equipo de mantenimiento pueda entender el funcionamiento del programa y hacer cambios.
15. *Plan de instalación*: Una vez que el sistema haya superado todas las pruebas está listo para ser instalado. Este manual describe como se realiza el proceso en diferentes entornos.
16. *Análisis de la prueba e informe de evaluación de seguridad*: Es el resultado de las pruebas. Se muestran las fortalezas y debilidades del sistema. Debe incluir un informe de evaluación de seguridad para certificar el sistema.

6.3.4. Manual de usuario

La importancia de la documentación de usuario está en que los usuarios no aceptarán un sistema que no puedan operar por tener un manejo poco claro; un porcentaje alto de la culpa del fracaso en la aceptación de los sistemas recae en una documentación pobre.

Desde el punto de vista de los usuarios la opinión generalizada es que la documentación no se entiende porque el usuario no tiene (ni tiene por qué tener) conocimientos informáticos o del funcionamiento interno del sistema, el estilo de redacción es poco claro, el documento está mal estructurado y no es fácil encontrar la información. A veces la información que se busca ni siquiera está.

Cuando se redacte un documento se tiene que hacer un esfuerzo por "empatizar" con el usuario (ponerse en su lugar). Además de escribir una enumeración de cada opción de menú, icono, etc. se debe documentar cada uno de los procesos en los que el usuario va a estar interesado (casos de uso). Se debe prestar atención a todas las posibles variantes de dichos procesos y documentarlas, porque se ha de tener la certeza de que ocurrirán y los usuarios se quejarán. Se debe evitar también el miedo que tienen muchos usuarios a enfrentarse a un programa totalmente nuevo por medio de una introducción con algún ejemplo muy sencillo guiado paso a paso y con capturas de pantalla.

Recursos

¿Cuánto esfuerzo se debería dedicar a la documentación? En la empresa desarrolladora en total (análisis, diseño, implementación, pruebas y manuales técnico y de usuario) entre un 20 y un 30 %. Durante el mantenimiento: para estudiar la documentación 6 % y para actualizar la documentación 6 %.

Objetivos del manual de usuario

1. El usuario debe saber introducir los datos de entrada y obtener los de salida.
2. El usuario debe poder interpretar correctamente los mensajes de error.
3. Es deseable que una parte del mismo pueda servir como tutorial del sistema.
4. Debe ser también un manual de referencia.

Es importante tener en cuenta a qué personas va dirigido el manual; no se redacta igual para un directivo, que probablemente esté más interesado en una visión general del sistema que no va a usar directamente que a un empleado de base que se dedica a introducir algunos datos, o que a un miembro administrativo que usará el sistema para por ejemplo sacar estadísticas de ventas. Como existen diferentes perfiles de utilización el manual de usuario deberá reflejar esto estructurándose en módulos dirigidos a tipos diferentes de personas.

Dentro de cada uno de los módulos del sistema deberían existir procedimientos específicos para funcionalidades concretas. Un procedimiento sería una lista ordenada de acciones para conseguir algo. Las posibles bifurcaciones de cada procedimiento respecto al camino principal deberán documentarse igualmente.

Existen estándares de documentación que especifican el índice del manual (y algunos como el de GNU incluso el formato, que debe ser TEXINFO para así poder ser traducido automáticamente a otros formatos). En cualquier caso, el índice deberá contener los siguientes puntos:

1. *Instalación del producto*. Se indicarán los requisitos, tanto software como hardware.
2. *Configuración* teniendo en cuenta los diferentes perfiles de usuario.
3. Si el manual tiene varios módulos en función de los perfiles de usuario, para cada uno de ellos:
   - Se indica un procedimiento para obtener cada funcionalidad. Debe indicarse la secuencia en la que se ejecutan las acciones y las posibles bifurcaciones.
   - Cada procedimiento se entiende mucho mejor si tiene ejemplos.
   - Es deseable que exista un tutorial, ya sea en la aplicación o en el manual.
   - Si se manejan documentos de entrada y/o salida se dará si ha lugar una breve explicación sobre su nombre, formato, origen, localización, usuarios que lo manejan y procesos de *backup*.
4. Manual de referencia: Es el documento que contiene toda la información en detalle.
5. Lista de errores con su significado y posible solución; el usuario no tendrá que recurrir a esta parte del manual si esta información está disponible en la propia aplicación. Esta parte puede estar en el manual de referencia.
6. Glosario de términos.

Procedimientos y tutoriales

- *Formularios de entrada/salida*: Especifican las normas para rellenar los formularios de entrada/salida (si es que existen).
- *Pantallas y mensajes*: Contienen las instrucciones de operación para los procesos *on-line*, incluyendo el flujo de pantalla para cada uno de ellos, los mensajes del sistema y las asignaciones de las teclas de función. Se debe confeccionar un glosario, preferentemente *on-line*, con todos los mensajes de error que puede generar el sistema.

- *Informes del sistema*: Presenta el inventario, el ejemplo y la descripción detallada de los informes del sistema, incluyendo su ordenamiento, cortes de control, frecuencia de emisión, número de copias y forma de distribución. Se debe dividirlos en categorías de acuerdo al nivel de los usuarios a quienes están orientados.
- *Procedimientos de control*: Describe la estructura general de los controles del sistema y define su relación con los controles manuales. Incluye la descripción de los mecanismos para efectuar controles cualitativos y cuantitativos sobre los datos que ingresan al sistema, los que se transfieren entre los distintos programas y sobre la información de salida, asegurando su integridad y confiabilidad.
- *Procedimientos de usuario*: Son los procedimientos requeridos para el uso del sistema. Deben quedar reflejadas las responsabilidades de cada tipo de usuario, en cuanto al envío y/o carga de información (tiempos y oportunidad de proceso), como también sus responsabilidades en cuanto al control y aprobación de las salidas del sistema.
- *Procedimientos de reabastecimiento*: Son los procedimientos requeridos para la provisión de los insumos para utilizar el sistema, como: formularios preimpresos, etiquetas, recargas para impresoras, disquetes, cintas, etc.
- *Soporte a usuarios*: Son los procedimientos disponibles para obtener ayuda sobre el uso del sistema. Deben quedar reflejadas las opciones que tiene el usuario para solucionar una duda identificando el o los sectores responsables del servicio.
- *Material de capacitación*: Inventario de cada una de las opciones disponibles para capacitación, ya sea autoasistida o a través de cursos, tutoriales, etc.

6.3.5. Manual del sistema

Desde el punto de vista de las personas que hacen el mantenimiento los *problemas* que se encuentran es que la documentación:

1. ¡Nunca existió!
2. Es escasa o incompleta.
3. No está actualizada con los cambios que se han ido haciendo al sistema.
4. Presenta contradicciones con el contenido real del código.
5. No es clara.

Objetivos

- Documentar en forma detallada las características de cada componente de la arquitectura técnica del sistema.
- Servir como base para el mantenimiento futuro de los procesos y programas del sistema.

Contenidos

1. *Descripción general del sistema*

- Identificación del sistema: Nombre y siglas que lo identifican.
- Objetivos: Descripción breve del objetivo general del nuevo sistema, incluyendo referencia a las funciones del negocio (planificación, gestión, etc) *qué* cubre y *cómo* lo hace.
- Alcance: Diagrama de contexto que muestre la interacción del sistema con el resto de los sistemas de la organización.
- Características generales. Descripción breve de las siguientes especificaciones funcionales y/o técnicas:
    - Enfoque de desarrollo (a medida, paquete de software, etc).
    - Herramientas de desarrollo.
    - Tipo de procesos.
    - Tecnología de implementación (hardware, software, comunicaciones, etc).
- Políticas relacionadas con su uso. Siguiendo las políticas, normas y estándares vigentes, se deberán adjuntar:
    - Identificación del responsable de los datos.
    - Identificación de información confidencial y pública para su tratamiento.
    - Principales sectores usuarios.
    - Perfiles de usuarios y esquema general de seguridad.
    - Plan de contingencia que explique los métodos de trabajo alternativos ante fallos.
    - Plan de recuperación que explique los procesos de recuperación y reinicio ante los casos eventuales de destrucción o desastre.
- Principales funciones: descripción breve que describa las principales funciones del sistema. Se podrá acompañar de un esquema donde se identifiquen las actividades del negocio cubiertas por el sistema y sus necesidades de información.
- Observaciones: Cualquier información que no haya sido especificada antes y que se considere de interés.

2. *Arquitectura general del sistema*

    - Requisitos funcionales: expresar los requisitos funcionales que debe satisfacer el sistema
    - Eventos del sistema: extraer del modelo de eventos una lista conteniendo todas aquellas funciones ejecutadas por los distintos usuarios que originan una entrada al sistema. Indicando:

        a) Número de evento.

        b) Descripción.

        c) Origen de la información.

        d) Destino de la información.

   e) Flujo de datos.

- Procesos del sistema: Contiene el inventario y la descripción general de cada una de las opciones de la aplicación, incluyendo imágenes de las pantallas utilizadas.
- Diálogo del sistema: Muestra la comunicación entre las distintas pantallas de la aplicación identificando las condiciones que provocan la navegación entre las mismas.
- Modelo de procesadores: Muestra de manera gráfica los distintos dispositivos físicos requeridos, la asignación de tareas y la conectividad entre ellos.
- Esquema de módulos:Esquema donde se visualice el sistema como un conjunto de bloques, cada uno de los cuales representa un módulo o subsistema del mismo.

3. *Especificaciones de los módulos*. Por cada módulo deberá tenerse la siguiente información:

   - Nombre del módulo.
   - Descripción detallada del módulo y sus funciones.
   - Diagrama general del módulo.
   - Flujo de pantallas (si corresponde)
   - Lista de componentes que lo implementan.
   - Lista de bases de datos.
   - Lista de archivos y tablas que utiliza.
   - Lista de las vistas lógicas de los datos.
   - Lista de los informes que genera.

4. *Especificaciones de los procesos*

   - Inventario general de procesos del sistema: es una lista de todos los procesos, que contendrá para cada proceso los siguientes datos: nombre del proceso, Descripción breve y Módulo al que pertenece ó módulos que incluye.

     Se dejará documentado que para obtener información particular de cada proceso se deberá relacionar este manual con el manual de operaciones.

5. *Especificaciones de programación*: Para cada componente de programación del sistema (dependiendo del lenguaje apropiado: programa, subrutina, procedimiento o función), deberá incluirse la siguiente información:

   - Nombre del componente.
   - Descripción y objetivos.
   - Diagrama general del componente.

6. *Especificaciones de los datos*

   - Modelo conceptual de datos: Permite visualizar las entidades de datos requeridas por el sistema y sus relaciones. Se deberán adjuntar:

    - Diagrama general.
    - Diagrama de las distintas vistas (si fuese necesario).
    - Definición de las entidades.
    - Diccionario de datos.

- Diseño lógico de base de datos: Documenta la base de datos según la perspectiva de los desarrolladores de aplicaciones y los usuarios finales, sin preocuparse por su implementación física en un DBMS (*Database Management System*) particular. Se deberán adjuntar:

    - Diagrama de tablas y sus relaciones.
    - Diagrama de las distintas vistas (si fuese necesario).
    - Definición de atributos y dominios.

- Diseño físico de base de datos: Contiene las definiciones detalladas de todos los archivos y/o bases de datos del sistema y cada uno de sus componentes (tablas, columnas, claves, índices, integridad referencial, *triggers*, etc). Adjuntar la forma en que cada programa los accede y procesa sus datos, su localización física y la descripción de su sistema de *backup*. Para aquellos que tengan proceso de mantenimiento independiente del sistema, se deberá adjuntar la forma de acceso al menú correspondiente y las instrucciones para su uso.

6.3.6. Manual de mantenimiento

Objetivos

- Contener los procedimientos requeridos para asegurar el mantenimiento del sistema después de su instalación.
- Identificar los materiales necesarios para el proceso de mantenimiento.
- Documentar la ubicación de cada componente del sistema susceptible de ser modificado.

Contenidos

1. *Documentación del sistema*: Contiene el inventario y una breve descripción del contenido de cada uno de los manuales que conforman la documentación del sistema, así como el número de copias emitidas, su distribución y las ubicaciones físicas en las que se encuentran almacenadas. Esta información permite asegurar que todas las modificaciones que se efectúen sobre el sistema después de su instalación se reflejen en la documentación existente.
2. *Librerías del sistema*: Presenta el inventario de todas las librerías fuente y objeto del sistema, detallando para cada una de ellas los siguientes datos:

    - Nombre, versión, fecha, ubicación, tipo y uso (de test, producción, etc.)
    - Contenido (directorio de cada librería).

- Instrucciones de uso.
- Lista de autorizaciones de acceso.

3. *Modelo de pruebas del sistema*: Contiene toda la información generada para la ejecución de las pruebas del sistema (pruebas unitarias, prueba de integración, pruebas de usuario y prueba de aceptación del usuario).
4. *Material de capacitación*: Incluye el inventario y la descripción del material de capacitación disponible (manual del instructor, guías de participante, casos prácticos, etc.) incluyendo su ubicación, número de copias existentes y la audiencia a las que se encuentran orientados.
5. *Instrucciones de soporte a usuarios*: Define los procedimientos de servicio al usuario para dar respuesta a las llamadas de emergencias, los requisitos de mejoras o absorber consultas específicas acerca del sistema. Asimismo especifica los procedimientos para registrar los fallos o problemas que se presenten durante la operación diaria del sistema.

Tutorial posterior

Resumen de contenidos

Finalización del proyecto

El objetivo llegados a este punto consiste en entregar el producto al cliente y ponerlo en condiciones de funcionar. **La aceptación** ocurre cuando han terminado todas las actividades. Se adjunta una plantilla para el **informe de cierre del proyecto**. Los **indicadores del proyecto** son un conjunto de medidas para evaluar de un modo objetivo los resultados. Son de varios tipos: económicos, financieros, de ocupación laboral y de gestión.

Planificación de revisiones y organización del mantenimiento

Se definen los 4 tipos de mantenimiento (perfectivo, correctivo, adaptativo y preventivo) y la problemática asociada. Se define el concepto de "código heredado" (*legacy code*) y se da una descripción de como se debe gestionar el mantenimiento. Se define a su vez una plantilla de documento para esta tarea. El objetivo de la **planificación del mantenimiento** es asignar recursos lo antes posible para que estén disponibles en su momento, para ello se define un **plan de mantenimiento**. Las **técnicas de mantenimiento** son: ingeniería inversa, identificación de componentes funcionales, reestructuración y reingeniería.

Recopilación y organización de la documentación

Se exponen los motivos para documentar, directrices genéricas a seguir para la redacción de un documento, una clasificación de los documentos y una plantilla a seguir para los más importantes. El tipo de documentos que se contemplan aquí son los dirigidos a usuarios o mantenedores. Se hace hincapié en el **manual de usuario**, el **manual del sistema** y el **manual de mantenimiento**.

Ejercicios y actividades propuestas

1. ¿Qué es la barrera del mantenimiento? ¿Hay alguna solución?
2. ¿Qué cosas debe tener en cuenta el plan de mantenimiento?
3. Las pruebas de regresión son las mismas pruebas que se han aplicado antes pero que vuelven a pasarse cada vez que se hace un cambio. ¿Es razonable pasar las pruebas de regresión de los módulos que no se cambian?.
4. ¿Cuáles son los objetivos del manual de usuario?
5. ¿En qué manuales situaría la siguiente documentación?

    - Arquitectura general del sistema
    - Lista de mensajes de error
    - Tutorial
    - Diseño lógico de la base de datos
    - Diseño físico de la base de datos
    - Información acerca de las pruebas
    - Especificación de los procesos

Capítulo 7

Metodologías de desarrollo

7.1. Introducción a las metodologías de desarrollo

En los capítulos precedentes hemos visto las fases del ciclo de vida del desarrollo de aplicaciones. Estas fases son relativamente independientes del tipo de metodología que se siga. Una metodología consiste en concretar el tipo de ciclo de vida que se va a seguir y la forma en la que se realizan las actividades dentro de cada etapa, ahora bien, las etapas tienen que seguir siendo las mismas sea cual sea la metodología; es necesario tener una fase de especificación porque se trabaja con los requisitos que proporciona el cliente. Que una metodología utilice el ciclo de vida en cascada y que esto se haga solo al principio o que sea iterativa y haya varias mini-fases de este tipo es lo que distingue una de otra, pero esta actividad hay que realizarla de todas formas. El diseño es otra fase que es necesaria sea cual sea la metodología por los mismos motivos. La especificación es relativamente independiente de la metodología porque las técnicas de comunicación con el cliente son siempre las mismas, pero en el caso del diseño es donde las cosas empiezan a divergir. Existen dos tipos de diseño: estructurado y orientado a objetos. Una metodología se decanta entre uno de los dos. En este capítulo hemos decidido dar como botón de muestra dos metodologías de diseño orientadas a objetos actuales: Extreme Programming y el Proceso Unificado de Rational. El análisis y diseño estructurados, que son métodos bastante formalizados, han sido cubiertos en capítulos anteriores.

7.2. Proceso unificado de Rational

Es una de las metodologías más extendidas y conocidas por su amplia difusión comercial. Se puede estudiar como una metodología representativa de tipo clásico. Fue definida por los creadores del UML unificando los métodos de Jacobson, Booch y Rumbaugh. El hecho de que la empresa RATIONAL también distribuya herramientas específicas basadas en el mismo método, que facilitan el desarrollo, ha contribuido a su gran expansión.

Este proceso se maneja por casos de uso (correspondientes a los modos uso por los "actores" o agentes usuarios) para la extracción de requisitos y la identificación de las partes funcionales en las que se divide la solución. La arquitectura del proceso se modela con orientación a objetos.

7.2.1. Introducción

Toda esta sección es un resumen de los 11 primeros capítulos del libro [JBR00]. Las ilustraciones también están tomadas de ese libro. El Proceso Unificado de Rational es una metodología de desarrollo de software orientada a objetos creada por *Rational Software Corporation*. Los creadores de la metodología son los mismos que los del UML: Ivar Jacobson, Grady Booch y James Rumbaugh, que respectivamente eran autores de las metodologías: *Process Objectory*, el método Booch y la metodología OMT. Como toda metodología de desarrollo software su finalidad es convertir las especificaciones que da el cliente en un sistema software. Las características que tiene el R.U.P. son:

1. Está basado en componentes. Estos componentes a su vez están conectados entre sí a través de interfaces.
2. Su notación básica es el UML.
3. Está dirigido por casos de uso.
4. Se centra en la arquitectura.
5. El ciclo de vida es iterativo e incremental.

El proceso unificado está dirigido por casos de uso

Los casos de uso se vieron en el apartado dedicado a UML. Lo importante acerca de ellos son dos cosas:

1. Representan los requisitos funcionales del sistema desde el punto de vista del usuario.
2. Se usan como guía para el proceso de diseño, implementación y pruebas, por eso se dice que el RUP está dirigido por casos de uso.

El proceso unificado es iterativo e incremental

El proyecto se divide en una serie de partes o mini-proyectos. Cada uno de esos mini-proyectos va a ser una iteración. En cada iteración se trata un conjunto de casos de uso y los riesgos más importantes.

La vida del proceso unificado

El proceso unificado consiste en una serie de ciclos. Al final de cada ciclo se tiene una versión del producto. Las fases de cada ciclo son: inicio, elaboración, construcción y transición. Cada fase termina con un hito (ver figura 7.1), que se determina por la disponibilidad de un conjunto de artefactos (modelos o documentos desarrollados hasta cierto punto).

1. *Inicio*: se describe el producto final. Se responde a las siguientes preguntas:
    - ¿Cuáles son las principales funciones del sistema para sus usuarios más importantes?. La respuesta está en el modelo de casos de uso simplificado.

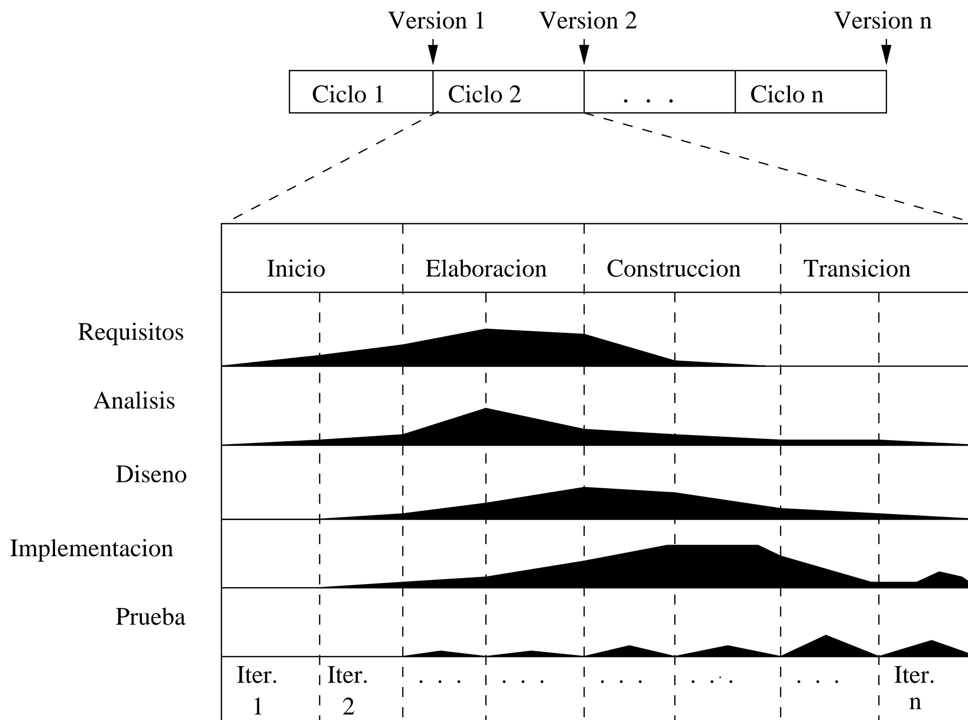


Figura 7.1: La vida del proceso unificado

- ¿Cómo podría ser la arquitectura del sistema?
- ¿Cuál es el plan del proyecto y cuánto costará desarrollar el producto?

2. *Elaboración*: Se especifican en detalle la mayoría de los casos de uso y se diseña la arquitectura del sistema. La arquitectura se especifica en forma de vistas de todos los modelos del sistema y todas ellas especifican el sistema entero.
3. *Construcción*:
   - Se construye el producto.
   - Se utiliza la mayor parte de los recursos.
   - Al finalizar se cubren todos los casos de uso.
   - La pregunta es: ¿ Cubre el producto las necesidades de los usuarios como para hacer una primera entrega?
4. *Transición*:
   - El producto existe en versión Beta.
   - Unos pocos usuarios experimentados prueban el producto.
   - Tipos de defectos:

   a) Los que tienen importancia como para justificar una versión incremental (*versión delta*)
   b) Los que se pueden corregir en la siguiente versión.

A su vez, cada fase puede tener varias iteraciones, cada una con cinco flujos de trabajo: Requisitos, Análisis, Diseño, Implementación y Prueba.

El producto

El producto terminado debe incluir más cosas que el código ejecutable: requisitos, casos de uso, especificaciones no funcionales y casos de prueba. Para llevar a cabo un ciclo se necesitan todas las representaciones del producto software:

1. Modelo de casos de uso.
2. Modelo de análisis para refinar los casos de uso y establecer la asignación inicial de funcionalidad del sistema a un conjunto de objetos que proporciona el comportamiento.
3. Modelo de diseño, que define:
   a) Estructura estática del sistema en la forma de subsistemas, clases e interfaces.
   b) Casos de uso reflejados como colaboraciones.
4. Modelo de implementación, que incluye:
   a) Componentes (que representan al código fuente).
   b) Correspondencia de las clases con los componentes.
5. Modelo de despliegue, que define:
   a) Nodos físicos (ordenadores).
   b) La correspondencia de los componentes con esos nodos.
6. Modelo de prueba, que describe los casos de prueba que definen los casos de uso.
7. Una representación de la arquitectura.

7.2.2. Las cuatro "P": Personas, Proyecto, Producto y Proceso

1. *Personas*: existen una serie de factores motivacionales que pueden mejorar o empeorar la eficiencia. Conviene tener en cuenta lo siguiente:
   - El proceso debe parecer "viable", se debe gestionar el riesgo, la gente debería estar estructurada en pequeños equipos (de seis a ocho personas), la planificación del proyecto debe ser realista, el proyecto debe ser comprensible y es mejor tener sensación de cumplimiento de objetivos.
   - Debe existir un proceso de desarrollo estandarizado que será conocido por todos.

- Una persona puede asumir uno o varios papeles como trabajador en el proceso de desarrollo en función de sus aptitudes, que deben ser consideradas cuidadosamente.

2. *Proyectos*: los equipos de proyecto tienen que tener en cuenta que un proyecto es una sucesión de cambios, que pasa por iteraciones, que cada una es en sí misma un miniproyecto y se debe tener un patrón organizativo.
3. *Producto*
   - Un sistema software no es solo los binarios (código ejecutable), es todos los artefactos que se necesitan para representarlo en forma comprensible para máquinas y personas (trabajadores y usuarios)
   - Artefactos: Son la información que crean y manejan los trabajadores, como los diagramas UML, prototipos, etc. Hay dos tipos: de ingeniería (los que nos ocupan) y de gestión.
   - Colección de modelos: Un sistema se construye utilizando distintos modelos por cada posible perspectiva del sistema. La elección de los modelos correctos es uno de los factores críticos para una correcta comprensión del sistema. Un modelo:
     - Es una abstracción semánticamente cerrada del sistema, es decir, para poder interpretarlo no se necesita información de otros modelos.
     - Siempre identifica al sistema que está modelando.
     - El sistema está formado aparte de sus modelos por las inter-relaciones que se establecen entre estos.
4. *Proceso*:
   - Un proceso es una plantilla que sirve para hacer proyectos igual que de una clase se derivan instancias.
   - Las actividades relacionadas conforman flujos de trabajo, en UML se representan como estereotipos de colaboración en el cual los trabajadores y los artefactos son los participantes.
   - El proceso unificado se adapta en función de las necesidades del proyecto según factores: organizativos, de dominio, del ciclo de vida y técnicos.

   Las herramientas: El RUP está soportado por herramientas CASE. Es mejor esto al proceso manual para evitar trabajo repetitivo. La herramienta escogida deberá dar soporte a todo el ciclo de vida y usará UML. Las herramientas son importantes porque influyen en el proceso, el cual a su vez dirige a las herramientas.

7.2.3. Proceso dirigido por casos de uso

Los casos de uso son los encargados de la captura de requisitos. Con ellos se identifican y especifican clases, subsistemas, interfaces, casos de prueba y se planifican las iteraciones del

desarrollo y de la integración. En una iteración nos guían a través del conjunto completo de flujos de trabajo. Los objetivos de la captura de requisitos son dos:

1. Encontrar los verdaderos requisitos
2. Representarlos de un modo adecuado para los usuarios, clientes y desarrolladores.

Veamos ahora lo que es un caso de uso. Un **actor** es una persona o proceso externo al sistema que interactúa con él. Un **caso de uso** es una secuencia de acciones que el sistema lleva a cabo para ofrecer un resultado de valor para un actor, es decir, un caso de uso proporciona un resultado observable para el usuario. Dentro de una interacción con el sistema puede haber muchas variantes, muchas de ellas pueden ser recogidas en un único caso de uso. Durante el análisis y el diseño el modelo de casos de uso se transforma en un modelo de diseño a través de un modelo de análisis. Los casos de uso son importantes por lo siguiente:

1. Proporcionan un medio sistemático de capturar *requisitos funcionales* centrándose en el punto de vista del usuario.
2. Dirigen el proceso de desarrollo porque el análisis, diseño y prueba se planifican en términos de casos de uso.
3. Un pequeño subconjunto de ellos sirve para idear la arquitectura (cada iteración implementa un conjunto de casos de uso proporcionando un incremento).

7.2.4. Proceso centrado en la arquitectura

La arquitectura es un conjunto de representaciones de un sistema tomadas cada una desde diferentes perspectivas. Cada una de estas representaciones se llaman **vistas**. La información que contiene es:

1. Organización del sistema software.
2. Elementos estructurales del sistema, sus interfaces y sus comportamientos.
3. Composición de los elementos estructurales y del sistema en subsistemas progresivamente más grandes.

Las vistas que se incluyen son: los casos de uso, modelo de análisis, modelo del diseño, etc. La arquitectura es necesaria para: Comprender el sistema, Organizar el desarrollo, Fomentar la reutilización y Hacer evolucionar el sistema.

Casos de uso y arquitectura

Se construye la arquitectura de forma que permita implementar los casos de uso actuales y previsibles en el futuro. Otros factores que influyen en la arquitectura son:

1. El tipo de producto software que queremos desarrollar.
2. El tipo de *middleware* (capa intermedia) queremos utilizar.
3. Los sistemas heredados que tenemos que utilizar.

4. Los estándares y políticas corporativas a los que debemos ajustarnos.
5. Los requisitos no funcionales generales.
6. Las necesidades de distribución.

Para integrar todas estas necesidades primero se hace un diseño de alto nivel para la arquitectura, a modo de **arquitectura de capas** (una capa es una parte bien definida de un sistema a partir de paquetes o subsistemas). Después formamos la arquitectura en un par de **construcciones** (versión ejecutable del sistema o de una parte del mismo). Todo esto dentro de la primera iteración. Al principio se trabaja con las partes generales de la aplicación y requisitos generales no funcionales. En esta primera pasada se trata de tener una visión general.

Segunda construcción: Se trabaja con requisitos específicos a base de escoger unos cuantos casos de uso relevantes. Se implementan subsistemas a través de una serie de iteraciones. Al final se debería conseguir una arquitectura estable. Con una arquitectura estable se implementan los casos de uso que quedan para proporcionar toda la funcionalidad.

7.2.5. Proceso iterativo e incremental

El proceso de desarrollo consta de una serie de hitos que dan el criterio a seguir por los diseñadores para dar el paso de una fase a la siguiente. En una fase se pasa por una serie de iteraciones e incrementos que nos llevan hasta esos hitos. Los criterios que se siguen en las fases son:

- *Inicio*: Viabilidad.
- *Elaboración*: Capacidad de construir el sistema con un presupuesto limitado.
- *Construcción*: Sistema capaz de una operatividad inicial en el entorno del usuario.
- *Transición*: Sistema que alcanza operatividad final.

Un proceso iterativo e incremental significa llevar a cabo un desarrollo en pequeños pasos. Para ello:

1. Se escoge una pequeña parte del sistema y se sigue con el todo el ciclo de vida clásico en cascada (planificación, especificación, diseño ...).
2. Si estamos satisfechos con el paso anterior damos otro. Cada uno proporciona retroalimentación.
3. Las iteraciones son distintas. Al principio del proyecto proporcionan una comprensión de los requisitos, del problema, de los riesgos y el dominio de la solución; las últimas nos proporcionan la visión externa (producto para el cliente).

Motivos para adoptar un ciclo de vida iterativo e incremental:

1. Para identificar **riesgos**. Esto ocurre en las dos primeras fases: Inicio y Elaboración, en vez de en la etapa de integración como con el modelo en cascada.

2. La **arquitectura** se establece en la fase de elaboración y eso permite cambiarla si se considera necesario en una etapa temprana del desarrollo, por tanto con pocos costes. En el ciclo de vida en cascada esto se descubre más tarde.
3. Gestión de **requisitos cambiantes**: Gracias a que se hace una integración continua los usuarios disponen desde las primeras iteraciones de versiones ejecutables que permiten un cambio de impresiones en este sentido.
4. **Fallos**: Al igual que en los puntos anteriores, la ventaja de este ciclo de vida es que los fallos se van descubriendo a medida que se implementan nuevas funcionalidades; esto significa que no hay una avalancha de problemas al final.
5. **Aprendizaje**: Con un par de iteraciones es suficiente para que todo el mundo comprenda los diferentes flujos de trabajo.

Gestión de riesgos

Un riesgo es cualquier cosa que pone en peligro el éxito del proyecto. Las iteraciones se organizan para reducir riesgos. Tipos de riesgos:

1. Técnicos:
    a) Relacionados con nuevas tecnologías como puede ser distribuir procesos en muchos nodos o técnicas aún incompletas como reconocimiento de lenguaje natural.
    b) Relativos a la arquitectura. Una arquitectura robusta se adapta a los cambios y muestra donde encajan los componentes reutilizables.
    c) Los que tienen que ver con la fase de requisitos.
    d) Relacionados con temas tecnológicos.
2. No técnicos: son problemas de gestión o financiación de los que es responsable la dirección de la empresa.

Tratamiento de los riesgos

Para cada riesgo hay cuatro posibles tratamientos:

- *Evitarlo*: replanificando o cambiando los requisitos.
- *Limitarlo*: que solo afecte a una parte del proyecto.
- *Atenuarlo*: mediante pruebas se aísla y se aprende sobre él.
- *Controlarlo*: si no se puede evitar se diseña un plan de contingencia.

Iteraciones

Cuando una iteración termina se analiza para ver si han aparecido nuevos requisitos. Se examinan también los riesgos que quedan. Las **pruebas de regresión** comprueban que no se han introducido errores sobre partes anteriores a las de la iteración en curso. La **planificación**

de cada iteración sólo se puede hacer en detalle para iteración más cercana a la actual y con menos detalle para las siguientes.

Secuenciación

Los casos de uso establecen una meta y la arquitectura un patrón. Con esto en mente se planifica la secuencia de iteraciones. Las primeras se centran en los requisitos, problemas y riesgos y las siguientes en la visión externa. Es posible que las iteraciones se solapen un poco en el tiempo. El orden de las iteraciones es el que permita que las decisiones importantes se tomen antes.

El conjunto de modelos que representa al sistema en un momento dado se llama línea base. En la fase de elaboración se identifican los casos de uso que tienen un impacto sobre la arquitectura y se representan como colaboraciones. De esta forma se construye la línea base. El resultado de una iteración es un incremento, y consiste en la diferencia entre dos líneas base sucesivas. Cada fase termina con un hito, el desarrollo iterativo supone un cambio de actitud: es necesario dejar de valorar tanto las líneas de código y valorar más la reducción de riesgos y la línea base.

7.2.6. Captura de requisitos

Cada tipo de proyecto es diferente y tendrá una aproximación diferente pero se puede decir que un flujo de trabajo arquetípico tendrá que cubrir los siguientes puntos:

1. *Enumerar los requisitos candidatos*: es una lista provisional de características que se van convirtiendo en requisitos o en artefactos. Sirve para la planificación del trabajo. Se indica: su nombre, una descripción, su estado, su coste estimado, prioridad y nivel de riesgo.
2. *Comprender el contexto del sistema*. Hay dos forma de entender este contexto:

   a) *Modelado del dominio*: Describir los objetos importantes del contexto como objetos del dominio (clases) y enlazarlos. Se modela en UML. Estos objetos se pueden educir gracias a reuniones con expertos del dominio. Los productos son: Un glosario de términos y los objetos. Tipos de objetos: Objetos del negocio (p. ej: pedidos, cuentas, contratos), Objetos del mundo real y Sucesos.

   El glosario y los objetos se usarán al descubrir casos de uso y describir la interfaz de usuario y para sugerir clases internas del sistema en desarrollo.

   b) *Modelado del negocio*: Describir los objetos para comprenderlos. No es exclusivo de los "negocios", es una forma de llamar a este tipo de modelado. Está soportado en UML por el modelo de casos de uso y el modelo de objetos (modelo interno). Lo que hace es describir los procesos en términos de casos de uso y actores del negocio. Se desarrolla en dos pasos:

      1) Se crea un modelo de casos de uso que identifique a los actores.

2) Se desarrolla un modelo de objetos del negocio que realiza los casos de uso anteriores compuesto por trabajadores, entidades del negocio y unidades de trabajo.

Búsqueda de casos de uso a partir de un modelo del negocio: un actor es un trabajador o cliente del negocio. Cada trabajador cumple un papel en cada caso de uso. Después se buscan los casos de uso de los actores del sistema.

3. *Capturar requisitos funcionales*: se hace con casos de uso. Aparte de esto hay que especificar cómo será la interfaz de usuario.
4. *Capturar requisitos no funcionales*. Por ejemplo:

    - Requisito de interfaz: especifica como se relaciona con un elemento externo.
    - Requisito físico: características como forma, tamaño, peso, etc.
    - Restricción de diseño: extensibilidad, mantenibilidad o reutilización.
    - Restricción de implementación: normas de codificación, lenguaje, etc.

En la fase de inicio se capturan los casos de uso para delimitar el sistema y el alcance del proyecto. En la fase de elaboración se capturan los requisitos para delimitar el esfuerzo. Al finalizar esta fase se deben haber capturado el 80 %.

Artefactos

Veamos los artefactos utilizados en la captura de requisitos.

1. *Modelo de casos de uso*: es un modelo con actores, casos de uso y relaciones entre ellos. Pueden agruparse en paquetes y se puede ver desde distintos puntos de vista.
2. *Actor*: un actor es cualquier cosa externa al sistema, desde un usuario hasta otro sistema. El conjunto de actores delimita todo lo externo. Puede jugar varios roles y para cada uno de ellos tendrá un caso de uso.
3. *Caso de uso*: cada uno es una especificación, una secuencia de acciones que el sistema lleva a cabo para realizar una funcionalidad. Puede incluir diagramas de estados, diagramas de actividad, colaboraciones y diagramas de secuencia. Cada caso de uso tiene atributos. Una **instancia de caso de uso** es la ejecución de un caso de uso, que estará desencadenada por un evento o por la instancia de un actor. El **flujo de sucesos** de un caso de uso especifica cómo interactúa el sistema con los actores. Consta de secuencias de acciones.
4. *Descripción de la arquitectura*: es una vista de los casos de uso significativos para la arquitectura. Usada para decidir que casos de uso se implementan en cada iteración.
5. *Glosario*: conjunto de términos comunes en el sistema. Sirve para evitar confusiones. Sale del modelo de negocio o del modelo del dominio.
6. *Prototipo de interfaz de usuario*: útiles para depurar los casos de uso.

Trabajadores

Vemos a un trabajador como una persona real que desempeña una función dentro del proyecto. Una misma persona puede ser varios trabajadores. Tipos de trabajadores:

1. *Analista de sistemas*: modela los casos de uso, encuentra los actores y redacta el glosario. También es la persona que se encarga de capturar los requisitos.
2. *Especificador de casos de uso*: es un asistente del anterior. Realiza cada caso de uso en detalle trabajando con el usuario.
3. *Diseñador de interfaz de usuario*: se suelen usar prototipos, uno por cada actor.
4. *Arquitecto*

Flujo de trabajo

Se representa mediante un diagrama (ver figura 7.2). El diagrama tiene calles, en la cabecera se sitúan los actores y en las calles las actividades. Cuando un trabajador ejecuta una actividad crea y modifica artefactos. Se representa el flujo lógico, pero las actividades reales no tienen por qué ser secuenciales. Veamos las actividades una a una.

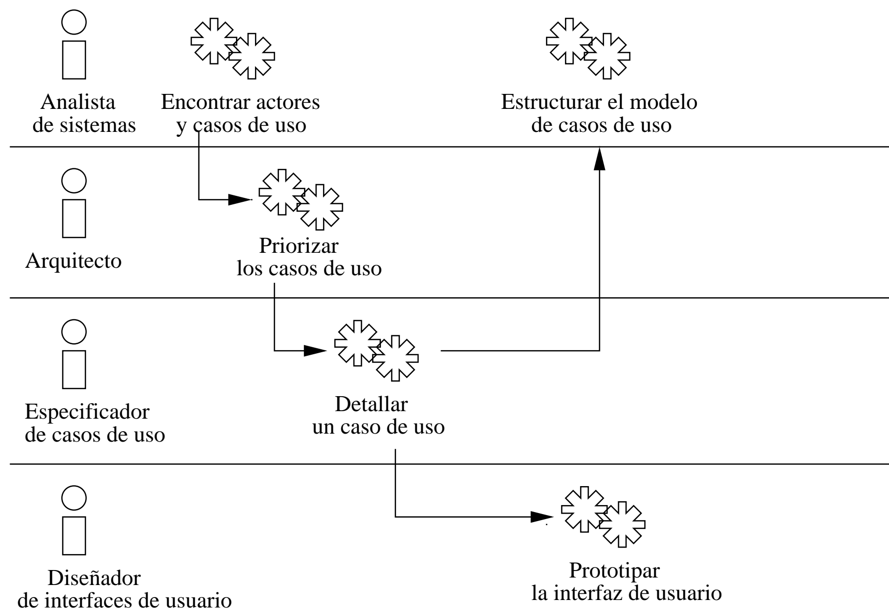


Figura 7.2: Flujo de trabajo: Captura de requisitos

1. *Encontrar actores y casos de uso*: Se realiza por un equipo de analistas y usuarios. La actividad consta de cuatro pasos:

   a) Encontrar los actores. Hay dos estrategias para ello:

- Encontrar un usuario que represente al actor.
- Encontrar roles iguales y fusionarlos en un único actor.

b) Encontrar los casos de uso: Si se parte del modelo del negocio cada rol de cada trabajador se corresponderá con un caso de uso. En otro caso se identifican hablando con los usuarios. Por otra parte, los casos de uso se caracterizan por proporcionar algún servicio de utilidad al actor y es mejor que el actor sea una persona real.

c) Describir brevemente cada caso de uso: Una vez identificados los casos de uso se les da una descripción breve o con los pasos a seguir.

d) Describir el modelo de casos de uso completo: Se trata de dar una visión general de los casos de uso. Se puede utilizar cualquier conjunto de diagramas que se consideren oportunos.

2. *Priorizar casos de uso*: Se trata de ver qué casos de uso se hacen en qué iteraciones, para ello hay que tener en cuenta consideraciones económicas y en general no técnicas. El resultado se pone en la vista del modelo de casos de uso.
3. *Detallar un caso de uso*. Para ello se deben realizar tres actividades:

   - Estructuración de la descripción de casos de uso: Un caso de uso incluye estados y transiciones entre ellos y el gráfico resultante puede ser complicado. Se puede describir primero el camino básico (el más normal) de principio a fin y luego el resto de caminos alternativos.
   - La descripción debe incluir: Estado inicial como precondición, posibles estados finales como postcondición, la primera acción a ejecutar, orden (si existe) numerada de la acciones, caminos de ejecución no permitidos, descripción de caminos alternativos, la interacción con los actores, utilización de recursos y acciones ejecutadas.
   - Formalización de la descripción: Un caso de uso se puede representar como una máquina de estados y si es demasiado complejo se puede utilizar alguno de los diagramas de UML: diagramas de estados, de actividad o de interacción.

4. *Prototipar la interfaz de usuario*: Se trata de crear una interfaz que permita la interacción del usuario para poder realizar los casos de uso. Se hace en dos pasos:

   - Crear el diseño lógico: Se identifican todos los elementos de la interfaz con los que va a interactuar el usuario. Cada elemento puede jugar varios roles porque puede estar en varios casos de uso.
   - Crear el diseño y prototipo físico: Se identifican los elementos necesarios y su configuración.

5. *Estructurar el modelo de casos de uso*: La finalidad es extraer descripciones de funcionalidad de dos tipos:

   - *Generales y compartidas*: Se buscan funcionalidades de casos de uso compartidas. La reutilización en los casos de uso supone un tipo de herencia, porque es necesario que exista una instancia tanto del caso que reutiliza como del que es reutilizado

- *Adicionales y opcionales*: La relación de extensión entre casos de uso consiste en que un caso de uso A añade una secuencia de acciones a un caso de uso B. La extensión se ejecuta en función de que se cumpla una condición.

Análisis

Durante el análisis se estructura el conocimiento que se ha conseguido de los usuarios por clases y paquetes en vez de casos de uso, de modo que no contenga inconsistencias. El análisis está pensado para dar la visión interna del sistema (en vez de la externa de los casos de uso) a los desarrolladores y por eso está descrito con su lenguaje. Es una primera aproximación al diseño y define realizaciones de casos de uso. El resultado del análisis es el modelo de análisis. Se debe hacer análisis cuando:

1. Se quiere planificar cada una de las iteraciones.
2. Para dar una visión general del sistema.
3. Si se tienen varias opciones de diseño, el análisis da una visión unificadora de todas ellas.
4. Se utiliza un sistema heredado complicado.

El modelo de análisis describe los resultados del análisis y mantiene la consistencia de este modelo durante el ciclo de vida. En las primeras iteraciones se hace énfasis en este modelo, más adelante se deja de actualizar.

Artefactos

1. *Modelo del análisis*: consiste en una jerarquía de paquetes, que son abstracciones de subsistemas o capas de diseño. Los paquetes contienen clases del análisis y realizaciones de casos de uso.
2. *Clase del análisis*: es una abstracción de una o varias clases y/o subsistemas del diseño del sistema. Se centra en los requisitos funcionales. Su comportamiento en lugar de definirse de con interfaces se define con *responsabilidades*: que son una descripción textual. Estas clases tienen atributos del dominio del problema, que en el diseño pueden pasar a ser clases. Se pueden corresponder con tres estereotipos: de interfaz, de control o de entidad.
3. *Realización de caso de uso-análisis*: explica como se lleva a cabo un caso de uso. Utiliza para ello diagramas de clases, diagramas de interacción y una descripción textual del flujo de sucesos.
4. *Paquete del análisis*: incluye clases del análisis, realizaciones de casos de uso y posiblemente otros paquetes de análisis. Los paquetes deben tener cohesión fuerte y acoplamiento débil. Cada paquete representa cosas distintas, reconocibles por los conocedores del dominio.
5. *Paquete de servicio*: es un tipo de paquete de análisis, pero indivisible. Un servicio es un paquete de funcionalidad que puede ser utilizado en varios casos de uso y los casos de uso funcionan utilizando varios servicios. Un paquete de servicio puede contener una o más clases relacionadas funcionalmente.

6. *Descripción de la arquitectura*: es la vista del modelo de análisis. Contiene la descomposición del modelo y sus dependencias, las clases de entidad, de interfaz, de control y las de análisis. También las realizaciones de casos de uso.

Trabajadores

1. *Arquitecto*: Es el responsable de la integridad del modelo de análisis y de la descripción de la arquitectura. Un modelo de análisis es correcto si realiza la funcionalidad de los casos de uso.
2. *Ingeniero de casos de uso*: Es responsable de que cada caso de uso responda a la funcionalidad requerida, tanto en el análisis como en el diseño.
3. *Ingeniero de componentes*: Comprueba que cada clase del análisis cumpla los requisitos que se esperan de ella. Mantiene la integridad de los paquetes del análisis. El ingeniero de componentes de un paquete lo es también de las clases del análisis contenidas en él.

Flujo de trabajo

Diagrama del flujo de trabajo en el análisis (ver figura 7.3) que muestra las actividades, los artefactos y los participantes. Veamos las actividades:

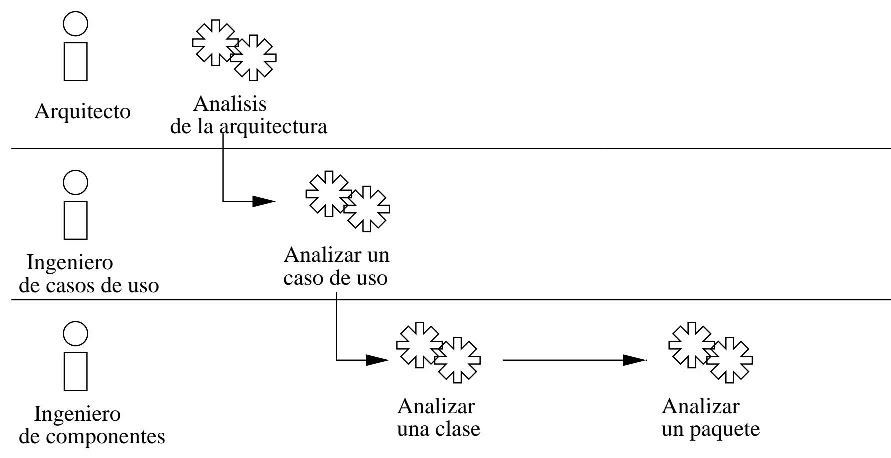


Figura 7.3: Flujo de trabajo del análisis

1. *Análisis de la arquitectura*: Para esbozar la arquitectura se realizan tres tareas:

   a) Identificar los paquetes de análisis. Una forma es asignar casos de uso a un paquete y realizar esa funcionalidad en el paquete. Si hay clases comunes entre diferentes paquetes se pueden sacar a otro paquete o fuera de cualquier paquete. Los paquetes de servicio se identifican cuando el análisis ya está avanzado. La forma de identificarlos es:

- Hay uno por cada servicio opcional.
- Por cada servicio que pueda hacerse opcional o por clases que estén relacionadas funcionalmente.

b) Identificar clases de entidad obvias. Se trata de identificar las clases necesarias para esbozar la arquitectura y no más. Las agregaciones y asociaciones entre clases del dominio pueden identificar asociaciones entre las clases de entidad.

c) Identificación de requisitos especiales comunes. Los requisitos especiales son los que aparecen durante el análisis.

2. *Analizar un caso de uso*. Se analiza un caso de uso con tres finalidades:
   - Identificación de clases de análisis.
   - Descripción de interacciones entre objetos del análisis.
   - Captura de requisitos especiales.
3. *Analizar una clase*. Los objetivos son:
   - Identificar y mantener las responsabilidades de una clase del análisis.
   - Identificar y mantener los atributos y relaciones de la clase de análisis.
   - Capturar requisitos especiales sobre la realización de la clase de análisis.
4. *Analizar un paquete*: Cada paquete debe realizar algunas clases del dominio o casos de uso, además, se trata de que cada paquete sea tan independiente de los demás como sea posible y deben documentarse las dependencias entre paquetes para el mantenimiento. Las normas que se deben seguir con los paquetes respecto a cohesión y acoplamiento son las mismas que con los módulos en la programación estructurada, es deseable que el paquete sea cohesivo, es decir, que solo tenga clases relacionadas funcionalmente.

7.2.7. Diseño

La entrada del diseño es la salida de la fase anterior y la salida del diseño es un modelo directamente implementable. Debido a esta proximidad con la implementación hay que comprender aspectos no funcionales como lenguajes de programación, sistema operativo, etc. También es necesario tener el sistema dividido en trozos manejables por equipos de trabajo. Los interfaces entre los diferentes subsistemas deberían estar claros. La implementación debería seguir la misma estructura que el diseño y de esta forma se podría hacer un camino de ida y vuelta automatizado.

Papel del diseño en el ciclo de vida

Se realiza sobre todo en las fases de elaboración y de construcción. Los artefactos son:

1. *Modelo de diseño*: es un modelo de objetos que describe cómo los casos de uso influyen en el sistema. Es también una abstracción de la implementación. Cada subsistema o clase

del diseño representa una abstracción con una correspondencia con la implementación. Los casos de uso se realizan por clases de diseño, lo cual se representa por colaboraciones en el modelo del diseño.

2. *Clase del diseño*: tiene una correspondencia directa con una clase en la implementación. Se utilizan características del lenguaje de programación para describirlas.
3. *Realización de caso de uso-diseño*: es una colaboración que describe cómo se realiza un caso de uso del diseño, el cual tiene una relación directa con un caso de uso del análisis. Se compone de: Descripción textual del flujo de eventos, diagrama de clases de diseño, diagramas de interacción y requisitos de implementación.
4. *Subsistema de diseño*: representa una parte del sistema. Debe tener alta cohesión y bajo acoplamiento. Consta de clases del diseño, realizaciones de casos de uso, interfaces y posiblemente otros subsistemas. Un subsistema de servicio se corresponde con un paquete de servicio del análisis que ofrece sus servicios en términos de interfaces y tiene en cuenta requisitos no funcionales.
5. *Interfaz*: una interfaz separa la especificación de la funcionalidad en términos de sus implementaciones. Son importantes porque describen las interacciones entre subsistemas.
6. *Vista de la arquitectura del modelo de diseño*. Consta de los siguientes elementos:

   - Descomposición del modelo de diseño en subsistemas, interfaces e interdependencias.
   - Clases importantes del diseño.
   - Realizaciones de caso-de-uso-diseño importantes.

7. *Modelo de despliegue*: Describe cómo se reparte la funcionalidad entre los nodos físicos. Tiene nodos, que son procesadores o recursos hardware y relaciones entre ellos que son los modos de comunicación (*intranet*, *bus*, etc). Describe la configuración de la red. La funcionalidad de un nodo depende de los componentes que estén en él.
8. *Vista de la arquitectura del modelo de despliegue*: Muestra los artefactos relevantes para la arquitectura, como la correspondencia entre los componentes y los nodos.

Trabajadores

1. *Arquitecto*: Es responsable de la integridad y de la arquitectura de los modelos de diseño y de despliegue. Un modelo es correcto si realiza la funcionalidad descrita en los casos de uso y nada más. La arquitectura es correcta si están presentes sus partes significativas.
2. *Ingeniero de casos de uso*: Responsable de la integridad de las realizaciones de casos de uso-diseño y su comportamiento. También es responsable de las descripciones textuales de los casos de uso.
3. *Ingeniero de componentes*: Garantiza que las clases del diseño estén correctamente definidas, así como los subsistemas y sus interrelaciones.

Flujo de trabajo

Las actividades y sus relaciones de precedencia temporal están reflejadas en el gráfico de la figura 7.4:

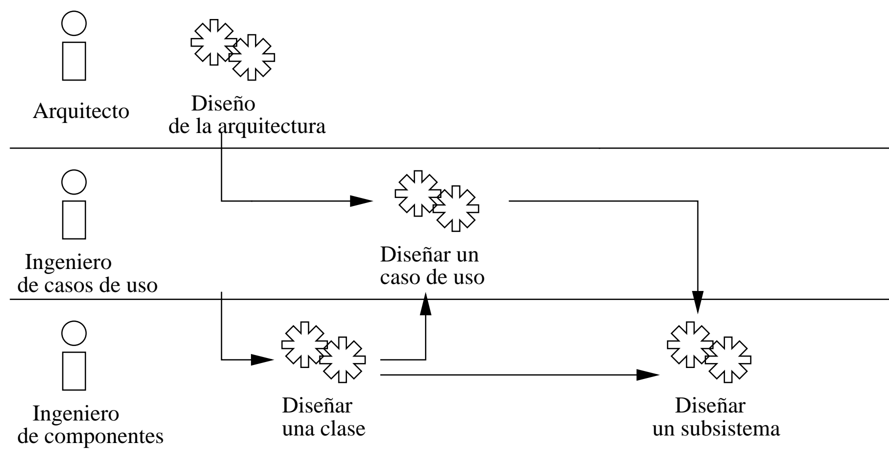


Figura 7.4: Flujo de trabajo del diseño

1. *Diseño de la arquitectura*: Se trata de esbozar los modelos de diseño y despliegue y su arquitectura identificando los elementos necesarios para estos modelos:
   - Nodos y configuraciones de red.
   - Subsistemas e interfaces.
   - Clases de diseño relevantes para la arquitectura.
   - Identificación de mecanismos genéricos de diseño.
2. *Diseño de un caso de uso*. Los objetivos son:
   - Identificar las clases del diseño y subsistemas necesarios para el flujo de sucesos.
   - Distribuir el comportamiento entre los objetos del diseño y los subsistemas.
   - Definir los requisitos sobre las operaciones de las clases del diseño, subsistemas e interfaces.
   - Capturar requisitos de implementación del caso de uso.
3. *Diseño de una clase*. Las actividades que se deben realizar son:
   - Esbozar la clase del diseño. Una clase puede ser de interfaz, de entidad o de control.
   - Identificar operaciones.
   - Identificar atributos.
   - Identificar asociaciones y agregaciones.

  - Identificar las generalizaciones.
  - Describir los métodos.
  - Describir estados.
  - Tratar requisitos especiales.

4. *Diseño de un subsistema*. Las actividades para ello son:

   - Mantener las dependencias entre subsistemas.
   - Mantener interfaces proporcionadas por el subsistema.
   - Mantener los contenidos de los subsistemas.

7.2.8. Implementación

La implementación tiene como finalidades:

- Planificar las integraciones. Se sigue un enfoque incremental.
- Distribuir el sistema entre los nodos.
- Implementar clases y subsistemas del diseño.
- Probar los componentes individuales hasta donde sea posible.

Artefactos

1. *Modelo de implementación*: Describe la implementación de las clases y la organización de los componentes. Se compone de un sistema de implementación, que a su vez consta de varios subsistemas. Cada sistema o subsistema consta de interfaces y componentes.
2. *Componente*: Es el empaquetamiento físico de los elementos de un modelo. Algunos estereotipos son: <<executable>>, <<file>>, <<library>>, <<table>>, <<document>>. Un *stub* es el esqueleto de un componente de propósito especial utilizado para desarrollar o probar otro componente.
3. *Subsistema de implementación*: Es una forma de organizar los artefactos del modelo de implementación. El mecanismo de empaquetamiento depende de la herramienta de implementación. Cada subsistema de implementación se puede trazar hasta un subsistema de diseño (dependencias, interfaces, clases de diseño y subsistemas de diseño).
4. *Interfaz*: Los componentes y subsistemas de implementación pueden utilizar las mismas interfaces y tener dependencias de uso sobre ellas. Cuando un componente proporciona una interfaz debe implementar todas sus operaciones.
5. *Vista de la arquitectura del modelo de implementación*. Consta de los siguientes elementos:

   - *Descomposición del modelo de implementación, subsistemas e interfaces.*
   - *Componentes clave.*

6. *Plan de integración de construcciones*: la forma de hacer la integración es incremental. El plan describe la funcionalidad implementada en cada construcción y las partes del modelo de implementación que están afectadas por la construcción.

Trabajadores

1. *Arquitecto*: responsable de la integridad del modelo de implementación, que es correcto si implementa la funcionalidad descrita en el modelo de diseño. También es responsable de la arquitectura del modelo de implementación y de la asignación de componentes a nodos.
2. *Ingeniero de componentes*: se encarga del código fuente de uno o varios componentes y de uno o varios subsistemas de implementación y de los elementos del modelo contenidos en él.
3. *Integrador de sistemas*: planifica la secuencia de construcciones necesarias en cada implementación y la integración de cada construcción.

Flujo de trabajo

La figura 7.5 es una representación dinámica de las actividades con sus respectivos trabajadores. Veámoslas una a una.

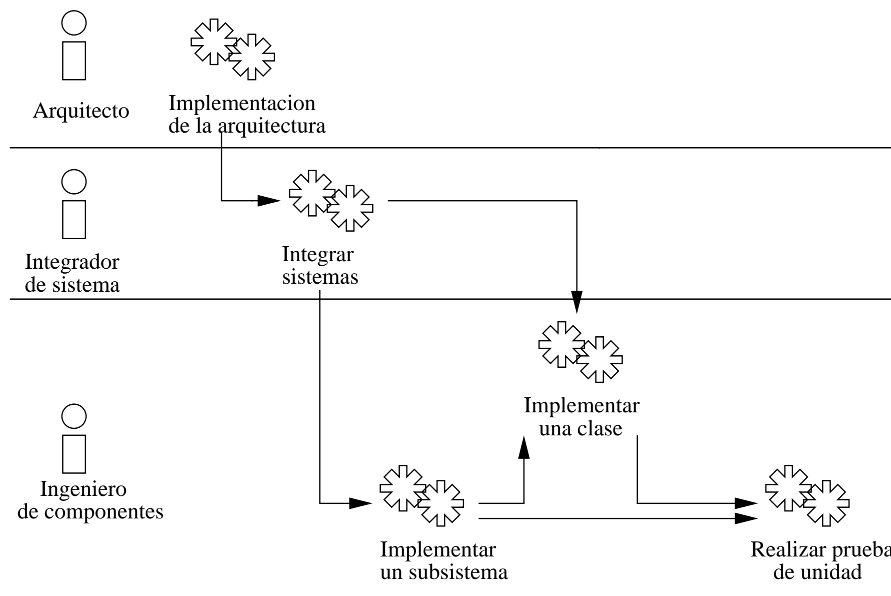


Figura 7.5: Flujo de trabajo de la implementación

1. *Implementación de la arquitectura*: esboza el modelo de implementación identificando componentes significativos y asignándolos a nodos
2. *Integrar el sistema*: se crea un plan de integración de construcciones y se integra cada construcción antes de que sea sometida a pruebas de integración.

3. *Implementar un subsistema*. se trata de asegurar que los requisitos implementados en la construcción y los que afectan al subsistema son implementados correctamente.
4. *Implementar una clase*. para ello es necesario:
   - Esbozar un componente fichero que contiene la clase.
   - Generar el código fuente a partir de la clase de diseño y sus relaciones.
   - Implementar sus operaciones.
   - Comprobar que el componente proporciona las mismas interfaces que la clase de diseño.
5. *Realizar prueba de unidad*. Hay dos tipos de pruebas: prueba de especificación (caja negra) y prueba de estructura (caja blanca)

7.2.9. Prueba

La finalidad de esta fase es planificar, diseñar e implementar las pruebas de cada iteración.

Artefactos

1. *Modelo de pruebas*: especifica cómo son las pruebas de integración y de sistema para los ejecutables. Pueden probarse también componentes como manuales de usuario o interfaces.
2. *Caso de prueba*: especifica la prueba que se hace sobre un requisito o conjunto de requisitos de un caso de uso o de una realización de un caso de uso-diseño.
3. *Procedimiento de prueba*: especifica cómo se lleva a cabo una realización de un conjunto de casos de prueba.
4. *Componente de prueba*: es la automatización de un caso de prueba.
5. *Plan de prueba*.
6. *Defecto*.
7. *Evaluación de prueba*.

Trabajadores

1. *Diseñador de pruebas*: responsable de la integridad del modelo de pruebas, de los objetivos y la planificación de las pruebas.
2. *Ingeniero de componentes*: responsable de la automatización de algunos procedimientos de prueba.
3. *Ingeniero de pruebas de integración*: las pruebas de integración verifican que los componentes integrados funcionan bien juntos.
4. *Ingeniero de pruebas de sistema*: realiza las pruebas sobre el resultado de una iteración y documenta los defectos encontrados.

Flujos de trabajo

En la figura 7.6 se representa el diagrama de actividades:

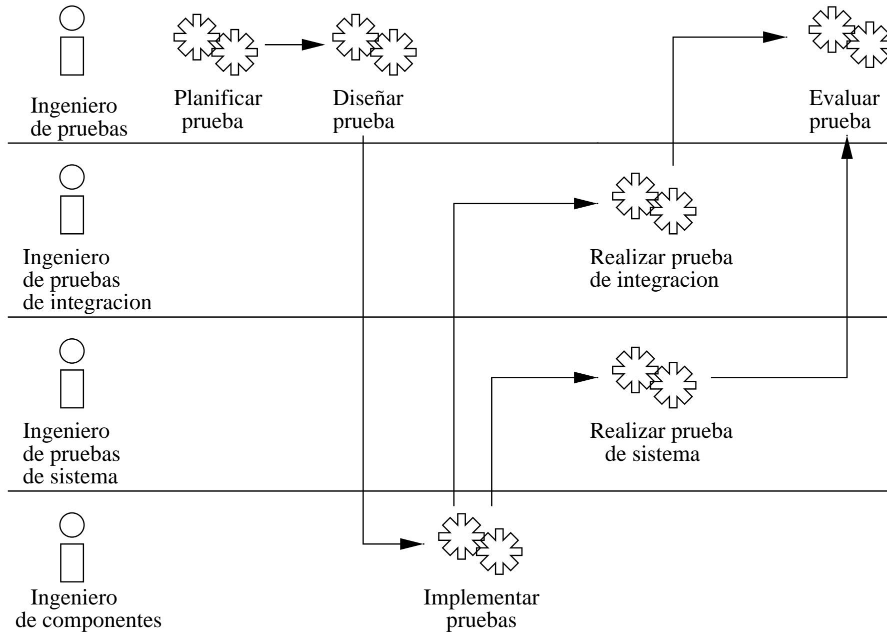


Figura 7.6: Flujo de trabajo de las pruebas

1. *Planificar prueba*: para esta planificación se describe primero una estrategia de prueba, se estiman los recursos necesarios y entonces se planifica el esfuerzo de la prueba.
2. *Diseñar prueba*. se dividen en estas dos actividades:
    - *Identificar y definir los casos de prueba para cada construcción*, que consiste en el diseño de los casos de prueba de integración, los casos de prueba del sistema y los casos de prueba de regresión.
    - *Identificación y estructuración de los casos de prueba.*
3. *Implementar prueba*: consiste en automatizar los procedimientos de prueba en la medida de lo posible creando procedimientos de prueba.
4. *Realizar pruebas de integración*: se realizan las pruebas contrastando los resultados con lo esperado e informando a los responsables y diseñadores de los resultados.
5. *Realizar prueba de sistema*: es una prueba posterior a las de integración que se realiza de un modo similar.
6. *Evaluar prueba*: Se comparan los resultados obtenidos con los objetivos del plan de prueba. Se tienen en cuenta dos factores:

- Terminación de la prueba: Número de casos de prueba y cantidad de código comprobado.
- Fiabilidad: Análisis de las tendencias de las pruebas y defectos observados.

7.3. Método extreme programming

Este método reciente de desarrollo orientado a objetos fue descrito originalmente por Kent Beck en su libro [Bec99]. En la actualidad está ganando muchos adeptos a través de Internet y tiene cada vez una mayor presencia como un método alternativo de desarrollo frente a los métodos más clásicos. Se basa principalmente en la simplicidad, la comunicación e interacción permanente con el cliente (comprobación de requisitos constante) y en el "pair-programming", que es la técnica de programación por parejas donde uno de los programadores escribe código y el otro lo prueba y después se cambian los papeles. De esta forma ya desde el principio se van probando los programas en cuanto a cumplimiento de requisitos como a funcionalidad. La simplificación de los protocolos de comunicación entre las diferentes fases (ver figura 7.7) y la inmediatez del desarrollo la convierten en una metodología muy "rápida" de desarrollo. Sus características son:

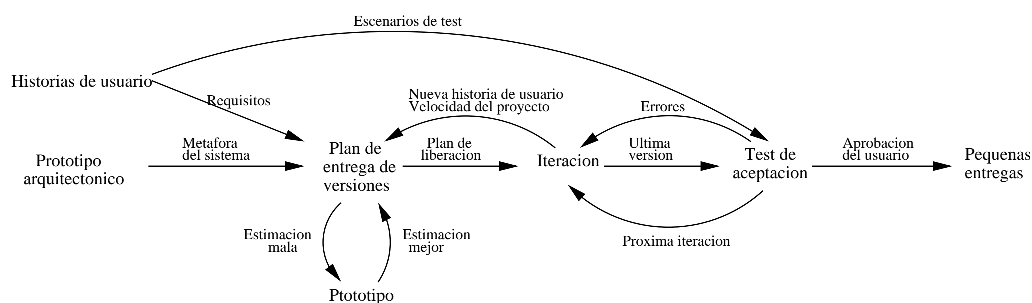


Figura 7.7: Etapas de extreme programming

- Permite introducir nuevos requisitos o cambiar los anteriores de un modo dinámico.
- Publica pronto versiones que implementan parte de los requisitos.
- Es adecuado para proyectos de tamaño pequeño o mediano.
- Es adecuado para proyectos de alto riesgo.
- Su ciclo de vida es iterativo e incremental. Cada iteración dura entre una y tres semanas.
- Un defecto de extreme programming es que necesita una continua interacción con el cliente. Esta es la limitación principal y hay que tenerla muy en cuenta.
- Otra crítica que se ha hecho a esta metodología es que no produce demasiada documentación acerca del diseño o la planificación.

7.3.1. Historias de usuario

Es la forma de hacer la especificación en Extreme Programming. Consisten en una serie de documentos cortos escritos por los usuarios. Características:

- Son parecidas a los casos de uso de UML, pero las escriben los usuarios.
- Son cortas (unas tres líneas) y sin vocabulario técnico.
- Cada una especifica un requisito del sistema. Pueden usarse para: estimar el tiempo de desarrollo y hacer un test de aceptación partiendo de una historia de usuario.
- Los desarrolladores estiman cuanto tiempo es necesario para implementar cada una. Si ese tiempo supera en alguna las tres semanas se debe desglosar en otras más pequeñas. Si es inferior a una semana, la historia de usuario ha descendido a un nivel de detalle excesivo y habrá que combinarla con otra.
- Una historia de usuario debe durar "idealmente" entre una y tres semanas. Idealmente significa no tener distracciones, saber exactamente lo que hay que implementar y sin otras asignaciones.

Diferencias entre las historias de usuario y la especificación tradicional:

- Nivel de detalle: en principio, el usuario sólo cuenta lo necesario para poder hacer una estimación del tiempo que va a tomar la implementación. Cuando llega el momento de implementar se vuelve a preguntar.
- La atención se centra en las necesidades del usuario, no se usa "tecno-jerga".

7.3.2. Plan de publicación de versiones

Lo primero que se hace al abordar un proyecto es una reunión para decidir un esquema del proyecto global, entonces se usa este esquema para decidir cómo va a ser cada una de las iteraciones. Cada iteración se planifica en detalle justo antes de empezar. Una de las cosas que se hacen en esta reunión preliminar es estimar el tiempo de desarrollo para cada una de las historias de usuario. En este plan se descubren las funcionalidades que se pueden ir implementando en las sucesivas versiones. Esto es útil para que el cliente vaya probando el producto e intercambie impresiones con el equipo. El cliente es el que va a decidir en qué orden quiere que se vayan implementando.

Es importante que las decisiones técnicas las tome el equipo de desarrollo y las decisiones de negocio el cliente. Las historias de usuario se imprimen en tarjetas, entonces el equipo de desarrollo las pone en una mesa para crear un conjunto de historias que serán implementadas para la primera versión. Es deseable empezar a publicar versiones cuanto antes. La planificación se puede hacer por tiempo o por alcance.

Tiempo: N^{o} de historias que se pueden implementar antes de una fecha dada.

Alcance: tiempo que se tardará en implementar un conjunto de historias.

Los proyectos están condicionados por cuatro variables:

1. *Alcance*: lo que hay que hacer.
2. *Recursos*: gente disponible.
3. *Tiempo*: fecha de entrega.
4. *Calidad*: cantidad de errores.

La cuarta está condicionada por las tres primeras y es inversamente proporcional al coste del proyecto.

7.3.3. Tarjetas CRC: Clases, Responsabilidades y Colaboraciones

Suponen una forma de pensar propia de la orientación a objetos que rompe con el diseño tradicional. Cada tarjeta representa un objeto. En la cabecera se pone el nombre de la clase a la que pertenece. Las responsabilidades se ponen en la parte izquierda y las clases con las que colabora a la parte derecha de cada responsabilidad. Lo que se hace en una sesión es simular el funcionamiento del sistema a mano. Por ejemplo: Una persona coge una tarjeta que es un objeto, manda un mensaje a otro objeto, entonces alguien se levanta y coge la tarjeta correspondiente a ese objeto, hace lo que sea, etc. Utilizando este sistema se pueden explorar con rapidez las diferentes alternativas de diseño.

Problema: No queda constancia escrita del diseño, aunque se puede escribir el resultado de la sesión de simulación con una copia de cada tarjeta.

El criterio que se debe seguir con el diseño es que sea tan simple como sea posible. Una forma de explorar alternativas de diseño es crear miniprototipos. Un **miniprototipo** es una programa pequeño hecho para explorar soluciones potenciales y luego tirarlo. También pueden servir para reducir riesgos o para mejorar la estimación del tiempo que tarda en ser implementada una historia de usuario.

7.3.4. Planificación de cada iteración

Cada iteración (ver figura 7.8) sólo se planifica en detalle al comienzo de la misma y se hace en una reunión que se convoca al efecto. Lo que se hace es:

- En primer lugar se deciden cuales son las historias de usuario que hay que implementar en esta iteración. Esta decisión le corresponde en su mayor parte al usuario. En total la iteración dura entre una y tres semanas.
- Luego se escriben los tests de aceptación que tendrán que ser satisfechos por cada historia de usuario.
- Las historias de usuario y los tests se traducen en tareas, que se escriben en tarjetas. Cada una debería durar entre uno y tres días. El plan detallado para cada iteración consiste en ese conjunto de tareas.

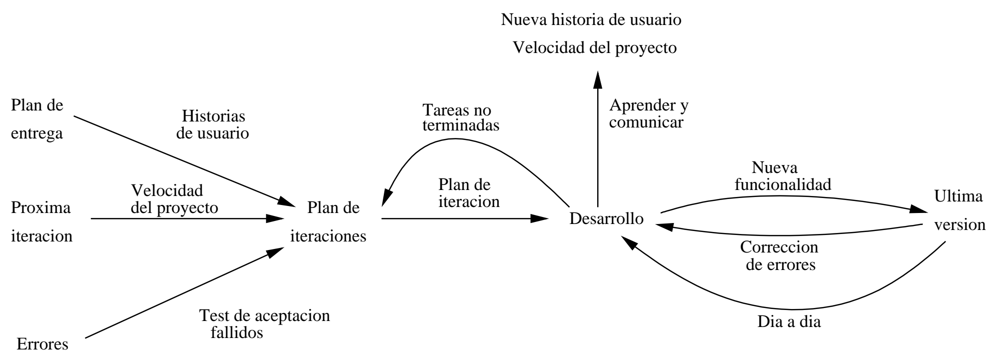


Figura 7.8: Planificación de iteración

- Al igual que ocurre con las historias de usuario, las tareas que tarden menos de un día se agrupan con otras, y las más largas que tres días se dividen en otras más pequeñas.
- Las tareas se asignan a programadores concretos y son éstos los que estiman el tiempo que puede tardar la implementación de la tarea que se compromete a hacer.
- Si la iteración va retrasada con respecto a lo proyectado se escogen algunas historias de usuario y se dejan para la siguiente iteración. Igualmente, si va adelantado se escoge historias de usuario y se introducen en la iteración actual.
- La velocidad a la que va el proyecto se estima en función del tiempo que han tardado en implementarse las historias de usuario que ya están hechas. Con intervalos de tres a cinco iteraciones habrá que reestimar el tiempo de las historias que quedan.
- Es importante que no se planifique en detalle más que lo que se va a hacer en la iteración en curso. El motivo es que eso sería perder el tiempo porque el proyecto cambiará.
- No se deben añadir funcionalidades extras antes de tiempo, es decir, no se debería implementar funcionalidades que no son necesarias en el momento. Motivo: el 90 % no serán usadas.

7.3.5. Integración

En la integración se juntan los pequeños módulos de software que tenemos y que parece que funcionan, pero resulta que esos pequeños trozos sólo han superado pruebas de unidad donde no han tenido que interactuar con otros módulos. El problema es que existen errores que sólo surgen en esa interacción, y ese es precisamente el problema de la integración. Formas de afrontarlo:

1. Cada desarrollador es propietario de algunas clases y se responsabiliza de corregir los errores que surjan en la integración.
2. Se debe tener un equipo que se dedique a eso.

Además lo típico es hacer la integración una sola vez al final. En extreme programming se hace una **integración secuencial**. En un mismo momento sólo puede existir una pareja de programadores integrando su código. Para que otra pareja pueda hacer lo mismo le tienen que pasar el

testigo. El código está todo él en el almacén (*repository*) y la pareja que está integrando es la que puede hacer test y publicar los cambios en el almacén. También es posible que una pareja pueda integrar su trabajo en la última versión en el momento que quiera con tal de que exista un mecanismo de bloqueo (p.ej. un token que se pasen de unos a otros).

La integración debe hacerse cada poco tiempo, idealmente, cada pocas horas hay que dejar código nuevo en el almacén. Todo el mundo debe hacer integraciones con la última versión disponible, de esta forma se evita trabajar con código obsoleto. Haciendo muchas mini-integraciones los problemas que surgían al final cuando se hacía la única integración se van resolviendo sobre la marcha.

7.3.6. Codificación de cada unidad

En esta sección también veremos algunas diferencias chocantes con la forma de trabajar tradicional (ver figura 7.9).

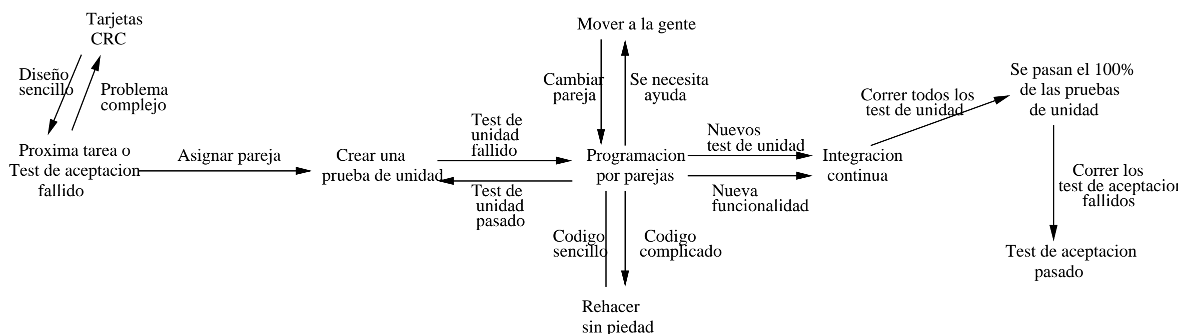


Figura 7.9: Pruebas

Codificación de la prueba de unidad

Lo primero que se hace antes de programar un módulo es preparar la prueba de unidad. Esto indica al programador qué es lo que tiene que hacer cuándo codifica. Los requisitos aparecen en forma de pruebas de unidad. Se sabe cuando se ha terminado porque se superan todos los tests de unidad. El beneficio que tiene de ello el diseño, es que en los tests de unidad se pone aquello que es importante para el cliente. Una forma de hacer esto es a base de pequeños incrementos:

1. Se crea una prueba de unidad pequeña que refleja parte de los requisitos.
2. Se hace el código que satisface ese test.
3. Se crea una segunda prueba.
4. Se añade el código correspondiente.
5. ...

Prueba de unidad

Veamos ahora cómo se hace una prueba de unidad. Hay tres pasos que se deben seguir:

1. Se crea un armazón o patrón para poder hacer partiendo de él las pruebas de unidad de un modo automatizado.
2. Se hace un test de todas las clases del sistema.
3. Se debe escribir el test antes de codificar el módulo.

La idea de hacer el test antes del código es importante, porque si se deja para el final, es posible que se encuentren problemas inesperados y se necesite más tiempo del inicialmente previsto. Además, en realidad el test no se hace *antes* sino *durante* porque se construye de forma incremental, pero siempre el test antes del código.

Cuando se publica un módulo al almacén debe ir obligatoriamente acompañado del test correspondiente, y no se puede publicar código que no haya pasado todos los tests. Como el código no tiene un propietario fijo y todo el mundo puede hacer modificaciones, esta norma es bastante razonable, cualquiera puede modificar una línea de código que haya escrito otra persona, ahora bien, la modificación tiene que pasar todos los tests asociados a la unidad.

Los tests de unidad permiten también rehacer el código porque pueden comprobar si un cambio en la estructura supone un cambio en la funcionalidad. Un pequeño problema es que los tests en sí mismos pueden tener errores. Por otra parte, una de las cosas deseables es poder hacer frecuentemente integraciones de todo el código. Si se construye un conjunto de tests globales para comprobar el producto final, será posible comprobar rápidamente si los cambios hechos en una unidad se integran bien y de esta forma no dejarlo todo para el final.

Test de aceptación

El test de aceptación también se conoce como test funcional en otros sitios. Los tests de aceptación son cajas negras que comprueban el funcionamiento del sistema. Son escritos por el cliente, y es el cliente el responsable de que sea correcto. También es el cliente el que decide la prioridad de cada test fallido. También se usan como test de regresión. Para que se pueda comprobar con rapidez y muchas veces si las historias de usuario cumplen con los tests de aceptación, estos deberían estar automatizados. Los resultados se comunican al grupo, que debe gestionar el tiempo para corregir errores.

A partir de las historias de usuario se hacen los tests de aceptación y a cada una le puede corresponder uno o varios. No se considera que una historia ha sido completada hasta que no pase todos sus tests de aceptación. Al igual que con los tests de unidad, los tests de aceptación deben hacerse *antes* de empezar a depurar. La diferencia entre un test de aceptación y un test de unidad está clara: un test de aceptación comprueba los requisitos expresados en las historias de usuario y un test de unidad comprueba que el código de una unidad es correcto.

Errores

Si se encuentra un error lo que se hace es crear un test de aceptación para ese error. De esta forma es fácil para el equipo saber cuando se ha corregido. Un error en un test de aceptación

puede verse en varios módulos diferentes en sus tests de unidad. Cuando todos los tests de unidad funcionan perfectamente se puede correr otra vez el test de aceptación para comprobar si el error ha sido realmente corregido.

7.3.7. Recomendaciones generales

Disponibilidad del cliente

Este es uno de los problemas de la metodología Extreme Programming, es necesario tener disponibles de principio a fin a los clientes hasta el punto de formar parte del equipo de desarrollo. Las funciones de los clientes son:

1. Escriben las historias de usuario.
2. Asignan prioridades a las mismas y se negocia con ellos cuáles incluir en cada iteración.
3. Una vez dentro de una iteración se les preguntan más detalles acerca de las historias de usuario.
4. Ayudan a estimar el tiempo de desarrollo.
5. Se negocia con ellos las fechas de entrega.
6. Prueban las versiones que se van publicando y pueden de esta forma proporcionar realimentación al equipo de desarrollo.
7. Ayudan a determinar cuáles son las pruebas que el sistema tiene que superar para considerarse apto (test funcional).

No podemos tener a cualquiera en el equipo, debe ser alguien con capacidad de tomar decisiones en la empresa, o al menos que pueda aconsejar, y que la conozca bien. Si hay varios clientes pueden no estar de acuerdo con una solución, la forma de solucionarlo es hacer una reunión donde se llegue a un acuerdo.

Acerca del código

1. Rehacer lo viejo: a veces mantener código antiguo no sólo no supone un ahorro de tiempo, sino un serio inconveniente. Si es necesario se deben eliminar la redundancia y las funcionalidades que no se usan ya o cambiar diseños antiguos.
2. Optimizaciones: no se debe optimizar el código hasta que está funcionando todo. Esto se deja para el final.
3. Nomenclatura: la nomenclatura es la forma en la que se escoge el nombre de las cosas. De cara a comprender bien el diseño general del sistema y poder reutilizar el código hay que escoger una forma de dar nombre a las clases y métodos de forma que todo el mundo lo pueda comprender fácilmente y que sea coherente.
4. Estándares de codificación: hay publicadas algunas normas sobre como se debe codificar en un lenguaje. Es conveniente seguir alguno de esos estándares por todos los miembros del equipo para que el código sea luego fácil de comprender y de mantener por todos.

Personal

Este es posiblemente el factor más importante. Los proyectos informáticos funcionan a base de mano de obra, todo lo que se ha dicho hasta ahora consiste en realidad en buscar formas de trabajar que sean adecuadas para que el componente humano funcione bien. Ahora veremos cuatro características propias de la metodología extreme programming a este respecto.

1. Reuniones diarias: Todos los días por la mañana se hace una reunión de 15 minutos en la que se da cita a todo el personal desarrollador. Lo que se busca es promover la comunicación entre todos los miembros del grupo. La gente cuenta los problemas que se han encontrado y cualquiera puede proponer soluciones. Este tipo de reuniones tiene algunas ventajas:

    a) Como todo el mundo asiste es más fácil encontrar soluciones.
    b) Se pueden evitar otras reuniones.

2. Propiedad compartida del código: Cualquiera puede corregir errores, añadir funcionalidades, etc de cualquier parte del proyecto (no sólo de **su** código, sino también del de los demás). Lo que se pretende es que cualquiera pueda aportar algo. Esto significa que la arquitectura del sistema en cierto modo la decide el equipo en su conjunto, lo cual resulta chocante para la mentalidad de la programación estructurada. Ventajas:

    a) El código es más fiable.
    b) Si una persona se atasca en un problema, se le ayuda y su parte no se convierte en un problema para otros.

3. Programación por parejas (*pair programming*): está basado en el principio de que dos personas trabajando juntas pueden hacer más que por separado. El resultado de esto es una mejora en la calidad del código, además no supone tardar más tiempo. La forma de ponerla en práctica es: Dos personas se sientan juntas ante el mismo ordenador. Una teclea y piensa en el problema desde el punto de vista táctico. La otra piensa desde el punto de vista estratégico.
4. Mover a la gente: no se puede permitir que todo el equipo dependa de que la única persona que conoce algo esté disponible o no. Puede ocurrir que esa persona deje la empresa o que esté sobrecargada en un momento dado. Consiste en que la gente trabaje en una parte del sistema distinta en cada iteración (o en parte de ella). Esto en combinación con la programación en parejas permite que no se pierda productividad en la parte que se deja. Las ventajas de esto son:

    a) Se evitan las “islas de conocimiento”
    b) El equipo es más flexible
    c) Si una parte del proyecto tiene problemas es posible reasignar gente a esa parte.

7.4. Métrica 3

7.4.1. Introducción

Métrica 3 es la metodología de desarrollo de aplicaciones informáticas creada por el Ministerio de Administraciones Públicas. También es utilizada en varias empresas.

Lo que se va a ver es un resumen de lo más relevante que puede ser útil a modo de introducción. Para un estudio a fondo se recomienda la documentación generada por el propio ministerio en http://www.csi.map.es/csi/metrica3/. Las figuras también han sido tomadas de esa página.

1. Su punto de partida es la versión anterior (la 2.1). La redacción comenzó en 1996.
2. Cubre los dos tipos de desarrollo, tanto el estructurado como el orientado a objetos, luego es una metodología mixta.
3. Sigue el ciclo de vida definido en la norma ISO-2.207.
4. Incluye los procesos que no forman parte de lo que se entiende como ciclo de vida a los que llama interfaces.
5. Tiene en cuenta la tecnología cliente/servidor y el desarrollo de interfaces gráficas de usuario (IGU).

7.4.2. Objetivos

Como toda metodología lo que se hace es sistematizar todas las actividades del ciclo de vida y las que no son parte del ciclo de vida pero influyen de algún modo en éste (como puede ser la planificación) para de esa forma conseguir los siguientes objetivos:

1. Dar un marco estratégico para el desarrollo de los sistemas de información dentro de las organizaciones.
2. Enfatizar el análisis de requisitos para que de esta forma los productos satisfagan las necesidades de los usuarios.
3. Mejorar la productividad del departamento de sistemas de información permitiendo una mayor adaptabilidad a los cambios y reutilización.
4. Que los procesos permitan una comunicación más fluida entre todos los miembros involucrados en la producción de software.
5. Facilitar la operación y mantenimiento de los productos obtenidos.

7.4.3. Estructura

Métrica 3 se divide por una parte en procesos principales, que son los relativos a la planificación, desarrollo y mantenimiento y por otra parte en interfaces, que son procesos asociados al desarrollo (gestión de la configuración, de proyectos y aseguramiento de la calidad). Cada proceso se divide en actividades y cada actividad tiene una descripción y una tabla de tareas propias de la actividad. Cada tarea tiene la correspondiente descripción y define los productos

que necesita de entrada, los que produce de salida, las prácticas necesarias para llevar a cabo la tarea y los participantes.

Métrica 3 es flexible en su estructura (ver figura 7.10) porque no es obligatorio seguir todos los procesos o actividades, se adapta en función de las necesidades de cada proyecto. Tampoco es necesario seguir las actividades secuencialmente, en ocasiones será factible su ejecución en paralelo. Los procesos correspondientes al desarrollo son los contemplados por el ciclo de vida ISO 12.207.

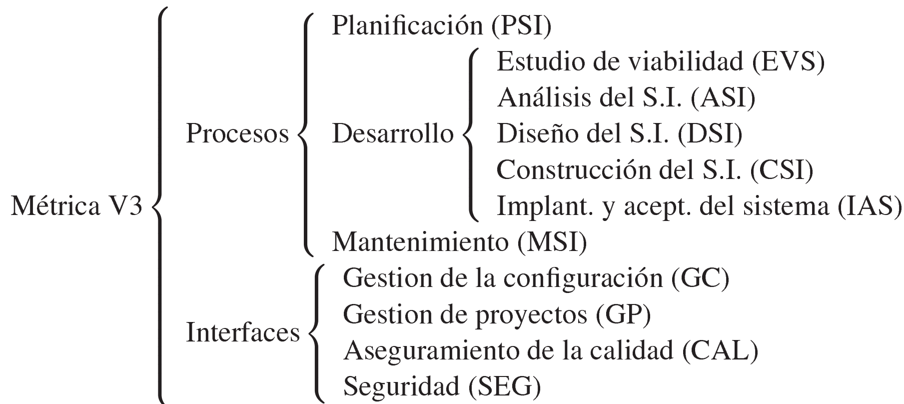


Figura 7.10: Estructura general de Métrica 3.

Procesos

7.4.4. Planificación de Sistemas de Información

El objetivo es ubicar el sistema de información en los objetivos estratégicos de la empresa. Estos objetivos sólo son bien conocidos por la alta dirección, que deberá implicarse en este proceso. Esta implicación significa proporcionar una parte de su tiempo y los recursos de la empresa que se consideren necesarios (documentación del sistema existente, personal que lo conozca, etc). Para conseguir esta implicación será necesario explicar el plan a todas las personas afectadas.

Para elaborar el plan se estudian las necesidades de información de los procesos de negocio afectados. Se deben definir los requisitos generales y obtener modelos conceptuales de información. Se evalúan las opciones tecnológicas y se propone un entorno. Debe definirse también un calendario y un plan de mantenimiento.

Se divide en las siguientes actividades:

- **PSI 1: Inicio del Plan de Sistemas de Información**. Se determina la necesidad o no del Plan de Sistemas de Información. Se obtiene una descripción general del mismo que incluye objetivos y ámbito afectado. Se identifican los factores de éxito y los participantes.
- **PSI 2: Definición y organización del PSI**. Se detalla el alcance del plan, se organiza el equipo de personas y se elabora una calendarización de tareas.

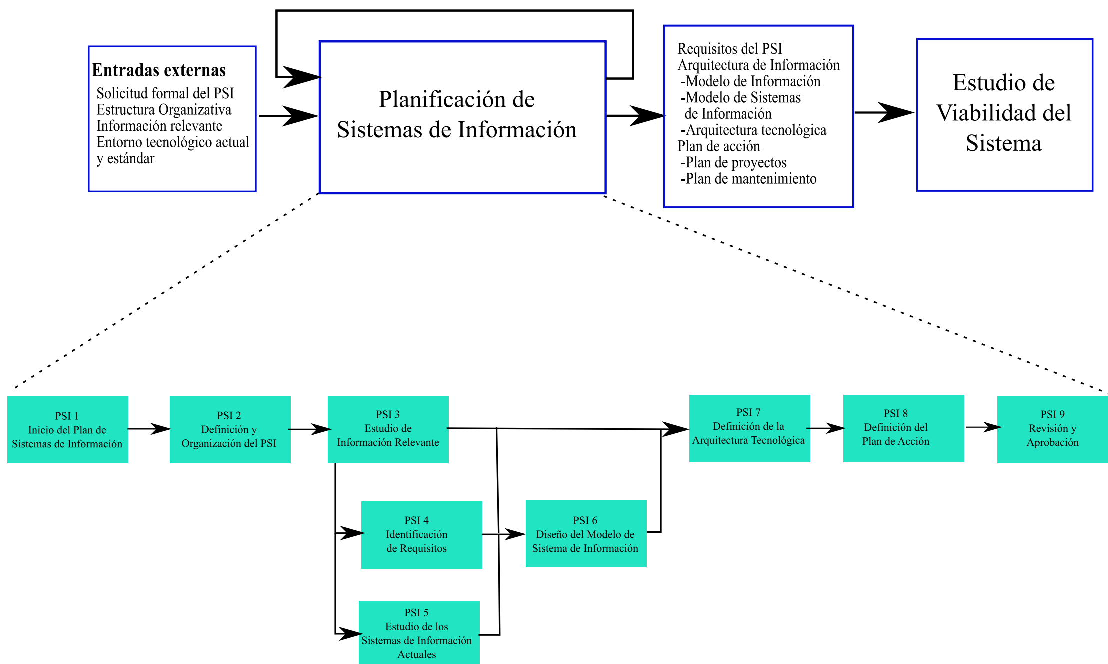


Figura 7.11: Entradas, Salidas y Actividades de la Planificación de Sistemas de Información

- **PSI 3: Estudio de la información relevante**. Consiste en analizar la documentación existente sobre los procesos y unidades organizativas involucrados.
- **PSI 4: Identificación de requisitos**. El objetivo es obtener los requisitos y un modelo de la información de la organización. Los procesos de información actuales se analizan junto con los usuarios definiendo como deberían ser (y no cómo son actualmente). Una vez obtenidos los requisitos se clasifican según su prioridad.
- **PSI 5: Estudio de los sistemas de información actuales**. Se trata de evaluar el sistema actual. Se debe tener en cuenta la opinión de los usuarios. Para cada sistema que se evalúe hay que determinar sus carencias en función de los objetivos que se han definido.
- **PSI 6: Diseño del modelo de sistemas de información**. El objetivo es dar una descripción del sistema de información que se va a construir. La primera tarea es analizar en qué medida los sistemas actuales satisfacen las necesidades e identificar posibles mejoras. Con la información anterior se define el modelo del nuevo sistema de información.
- **PSI 7: Definición de la arquitectura tecnológica**. Se selecciona una arquitectura tecnológica en función de los requisitos.
- **PSI 8: Definición del plan de acción**. Un plan de acción es el conjunto detallado de acciones que se toman para implantar el sistema de información. El plan lleva asociado un calendario y una estimación de recursos. Debe existir también un plan de mantenimiento.
- **PSI 9: Revisión y aprobación del PSI**. Consiste en la presentación de la arquitectura y el plan de acción, recoger posibles mejoras y aprobar dicho plan.

7.4.5. Estudio de la viabilidad del sistema

El objetivo es proponer una solución que satisfaga el conjunto de requisitos identificados en la PSI cumpliendo las restricciones económicas, técnicas, legales y operativas que proceda. De ser necesario se estudiará la situación actual. Puede ocurrir que existan varias alternativas, en cuyo caso se estudiará y valorará cada una para terminar seleccionando la mejor.

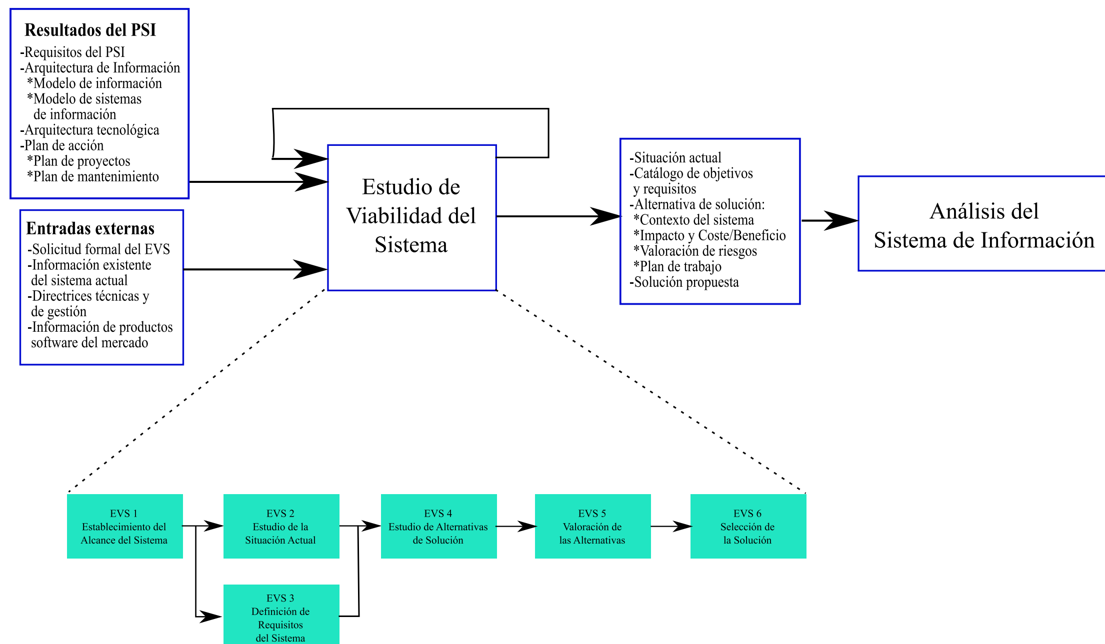


Figura 7.12: Entradas, Salidas y Actividades el Estudio de Viabilidad del Sistema

Se divide en las siguientes actividades:

- **EVS 1: Establecimiento del alcance del sistema**. Se determinan objetivos, se estudian los requisitos y se identifican las áreas afectadas. Se tienen en cuenta las restricciones aplicables a las posibles soluciones. Por último se identifica un conjunto de usuarios para conseguir su implicación en el proyecto. Se profundiza más o menos en el EVS en función de lo fuertes que sean las restricciones.
- **EVS 2: Estudio de la situación actual**. Consiste en una valoración de los sistemas informáticos utilizados en el momento. En función de la profundidad del análisis se designará un equipo específico para ello. Si se documenta dicha situación es conveniente dividir el sistema en subsistemas. El resultado es un documento que describe el sistema actual.
- **EVS 3: Definición de requisitos del sistema**. Los requisitos se obtienen en un conjunto de sesiones de trabajo con los usuarios seleccionados en la actividad anterior.
- **EVS 4: Estudio de alternativas de solución**. Se propone un conjunto de alternativas que responden a los requisitos y restricciones planteados anteriormente. Puede ser conveniente dividir el sistema en subsistemas. Se especifica si alguna de ellas está basada en soluciones existentes. De ser un desarrollo a medida se adjuntarán los modelos que describen la

solución.

- **EVS 5: Valoración de las alternativas**. Para cada posible solución se tienen en cuenta varios factores: impacto en la organización, beneficios esperados, análisis de riesgos, recursos y tiempo.
- **EVS 6: Selección de la solución**. La decisión la toma el Comité de Dirección. Se presentan las soluciones anteriores, se debaten, se proponen posibles mejoras y se toma una decisión (que puede ser concluir que el proyecto es inviable).

7.4.6. Análisis del sistema de información

El objetivo es conseguir un conjunto detallado de requisitos. El desarrollo puede ser tanto Estructurado como Orientado a Objetos; el jefe de proyecto deberá decantarse por uno de estos métodos y las actividades a llevar a cabo serán diferentes en uno y otro caso como se puede apreciar en la figura 7.13.

Se dice que el análisis es la parte importante del desarrollo. Para que esta fase tenga éxito es importante conseguir la implicación de los usuarios, para lo que se pueden utilizar técnicas interactivas como diseño de diálogos y prototipos que le permitan sentirse parte del desarrollo y aportar ideas.

Se divide en las siguientes actividades:

- **ASI 1: Definición del sistema**. Da una descripción inicial del sistema usando modelos de alto nivel; delimita el alcance y las interfaces con otros sistemas. Se selecciona un conjunto de usuarios representativos.
- **ASI 2: Establecimiento de requisitos**. Se definen, analizan y validan los requisitos completando el catálogo obtenido en la actividad anterior. Las tareas en las que se descompone esta actividad son solapables en el tiempo. Se recomienda la técnica de los casos de uso para obtener requisitos.
- **ASI 3: Identificación de subsistemas de análisis**. El objetivo es facilitar el análisis del sistema de información dividiendo el problema en otros más pequeños. Esta actividad se realiza en paralelo con la creación de otros modelos de análisis y como consecuencia será necesaria una realimentación continua.
- **ASI 4: Análisis de los casos de uso**. Para cada caso de uso identificado se describen las clases necesarias y cómo interaccionan los objetos que lo realizan.
- **ASI 5: Análisis de clases**. Se da detalle a las clases identificadas en la actividad anterior. Se identifican para cada una atributos, responsabilidades y asociaciones entre ellas. Hay un modelo de clases para cada subsistema que se irá ampliando a medida que se avance en el análisis.
- **ASI 6: Elaboración del modelo de datos**. Esta actividad sólo se realiza en el análisis estructurado. Un modelo de datos satisface las necesidades de información identificando entidades, relaciones, atributos y reglas de negocio. Se elabora siguiendo un enfoque descendente (*top-down*). La actividad se realiza en paralelo y con continuas realimentaciones con ASI 2, ASI 3, ASI 7 y ASI 8.

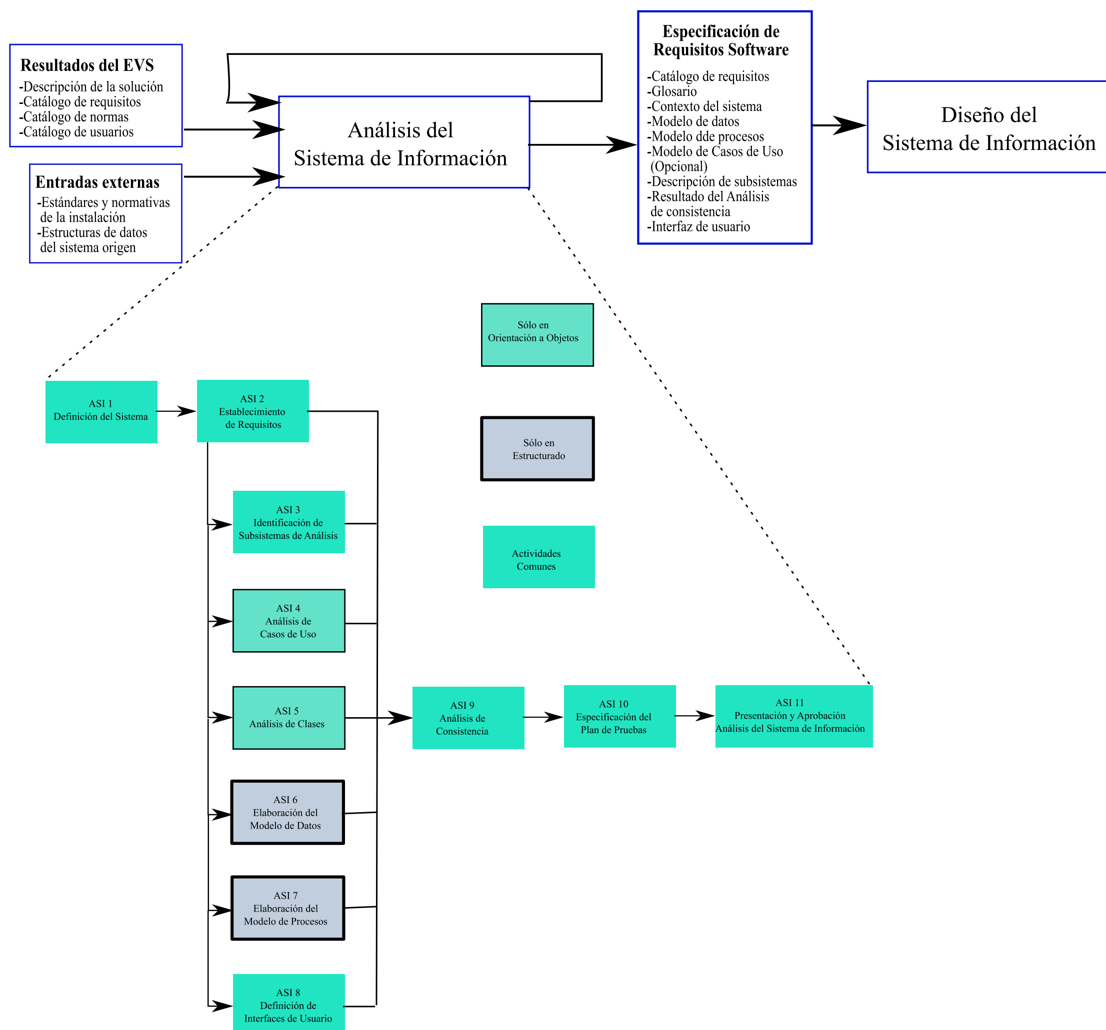


Figura 7.13: Entradas, Salidas y Actividades en el Análisis del Sistema de Información

- **ASI 7: Elaboración del modelo de procesos**. Es un análisis *top-down* de los procesos necesarios en el sistema en los desarrollos estructurados. Se hace en varios niveles de abstracción donde cada nivel da una visión progresivamente más detallada hasta llegar a procesos sencillos. Esta actividad se realiza para cada subsistema identificado en ASI 3. Se realiza en paralelo y con realimentaciones de ASI 2, ASI 6 y ASI 8.
- **ASI 8: Definición de Interfaces de Usuario**. La finalidad es modelar el comportamiento de los usuarios. Se debe tener en cuenta el entorno de operación de la interfaz, los perfiles de usuario a los que está dirigida y los tipos de proceso (en lotes o en línea). El flujo de trabajo es similar para desarrollos estructurados y orientados a objetos.
- **ASI 9: Análisis de consistencia y especificación de requisitos**. El objetivo es garantizar que los modelos generados funcionan correctamente y responden a las expectativas de los usuarios. Es necesario contar con alguna herramienta de apoyo.

- **ASI 10: Especificación del plan de pruebas**. El plan de pruebas es un documento formal que define objetivo y coordina la estrategia de trabajo. El plan se inicia en el análisis pero se va completando en los sucesivos procesos (diseño (DSI), construcción (CSI) e implantación (IAS)).
- **ASI 11: Aprobación del Análisis del sistema de información**. Se presenta el análisis del sistema de información al comité de dirección para su aprobación.

7.4.7. Diseño del sistema de información

En el diseño primero se definen la arquitectura y el entorno tecnológico. Posteriormente se especifica detalladamente cómo son los componentes del sistema de información. Esta información permite generar las especificaciones de construcción, la descripción técnica del plan de pruebas, los procedimientos de implantación y la migración y carga inicial de datos si procede.

Métrica 3 cubre desarrollos orientados a objetos y estructurados en una única estructura. Algunas actividades son comunes a ambos tipos de desarrollo y otras son sólo propias de uno de ellos (ver figura 7.14). En esta sección nos extenderemos algo más por ser más interesante desde el punto de vista técnico.

Hay dos bloques de actividades:

1. En las actividades DSI 1 a DSI 7 se definen la arquitectura y el diseño detallado.
2. En las actividades DSI 8 a DSI 12 se generan las especificaciones necesarias para la construcción del sistema de información.

Lista de actividades:

- **DSI 1: Definición de la arquitectura del sistema**. La arquitectura se define especificando las particiones físicas, la descomposición lógica en subsistemas, la ubicación de cada subsistema en cada partición y la infraestructura tecnológica.

  El particionamiento físico se especifica con los nodos y sus comunicaciones. Para cada subsistema se debe identificar claramente sus interfaces. Hay dos tipos de subsistemas: Específicos, que provienen de las especificaciones de análisis y de soporte, que se encargan de comunicar el sistema con la infraestructura que le da soporte de modo que los subsistemas específicos puedan ser independientes de dicha infraestructura.

  Una vez identificados los subsistemas se decide su ubicación en los nodos que componen el sistema. Esto es importante en sistemas cliente/servidor.

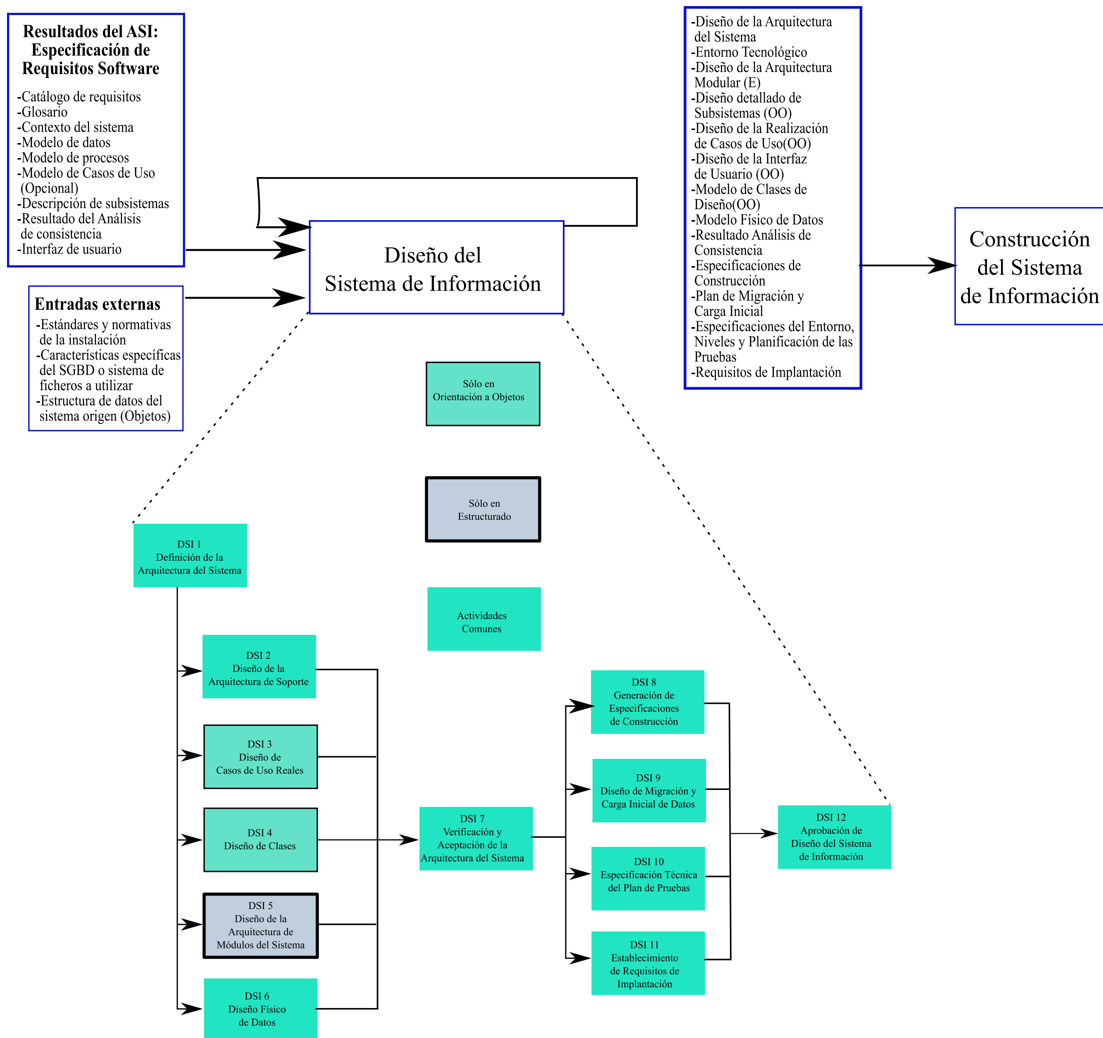


Figura 7.14: Entradas, Salidas y Actividades en el Diseño del Sistema de Información

Se definen también: Un catálogo de excepciones (situaciones anómalas), requisitos, normas y estándares originados como consecuencia de la adopción de una solución arquitectónica así como la arquitectura, los requisitos de operación, seguridad y control con los procedimientos necesarios para su cumplimiento.

- **DSI 2: Diseño de la arquitectura de soporte**. En esta actividad se diseñan los subsistemas de soporte identificados en DSI 1 y se determinan los mecanismos genéricos de diseño. El diseño de los subsistemas de soporte debe cumplir objetivos de reutilización para dar independencia a los subsistemas específicos. Se recomienda consultar la documentación de otros proyectos. Se hace en paralelo con el diseño detallado.
- **DSI 3: Diseño de casos de uso reales**. Se trata de determinar para cada uno de los casos de uso identificados las características concretas que deben tener las clases y subsistemas

que participan en los escenarios derivados del análisis. Se identificarán probablemente nuevas excepciones y características de la interfaz de usuario. Es importante validar que los subsistemas definidos en DSI 1.5 tienen la misma interfaz con otros subsistemas. En esta actividad pueden surgir nuevos requisitos de implementación.

- **DSI 4: Diseño de clases**. El objetivo es transformar el modelo de clases del análisis en un modelo de diseño que recoge información más detallada (atributos, métodos y relaciones entre clases) para lo cual se tienen en cuenta el entorno tecnológico y de desarrollo. Es posible que aparezcan clases nuevas o desaparezcan algunas. Una vez que se ha elaborado el modelo de clases de diseño se define la estructura física de los datos correspondiente a ese modelo. Si hay migración de datos se realiza su especificación a partir del modelo de clases o estructuras de datos del sistema origen.
- **DSI 5: Diseño de la Arquitectura de Módulos del Sistema**. Se identifican los módulos del sistema y sus interacciones. Se intenta que tengan alta cohesión y bajo acoplamiento. Para cada subsistema identificado en DSI 1.5 se diseña la estructura modular partiendo de los modelos obtenidos en la validación de los modelos (ASI 9.3) y el catálogo de requisitos. Se puede detectar la necesidad de características comunes que se implementarán como subsistemas de soporte. También se analizan las excepciones de los módulos, independizando las comunes. Cuando proceda se aplicarán los mecanismos genéricos de diseño de DSI 2.2.
- **DSI 6: Diseño Físico de Datos**. Partiendo del modelo lógico normalizado o el modelo de clases se diseña la estructura física teniendo presentes las características del SGBD y las restricciones impuestas por el entorno tecnológico y los requisitos; se pretende conseguir eficiencia. Se realiza en paralelo con DSI 2...DSI 5. Se realiza tanto en diseño estructurado como orientado a objetos y se tienen en cuenta los mecanismos básicos de diseño identificados en DSI 2.2.
- **DSI 7: Verificación y Aceptación de la Arquitectura del Sistema**. Se verifican las especificaciones de diseño, se analiza su consistencia y se acepta la arquitectura del sistema. El objetivo es garantizar la calidad y viabilidad de las especificaciones del diseño previamente a las especificaciones de construcción.
- **DSI 8: Generación de Especificaciones de Construcción**. El sistema se construye partiendo del diseño detallado siguiendo las especificaciones definidas en esta actividad. La construcción se realiza a partir de los componentes, que se definen como: Unidades independientes y coherentes de construcción y ejecución, que se corresponden con un empaquetamiento físico de los elementos del diseño de detalle, como pueden ser módulos, clases o especificaciones de interfaz. El sistema se divide en subsistemas y éstos a su vez en componentes.
- **DSI 9: Diseño de la Migración y Carga Inicial de Datos**. Esta actividad opcional toma como referencia el plan de migración de los sistemas de origen. A partir de dicho plan se procede a definir y diseñar en detalle los procedimientos y procesos necesarios para realizar la migración y se completa detallando pruebas y personas responsables. Se determinan las necesidades adicionales de infraestructura.
- **DSI 10: Especificación Técnica del Plan de Pruebas**. El plan de pruebas incluye pruebas

unitarias, de integración, del sistema, de implantación y de aceptación. Se toma como referencia el plan de pruebas o el plan de integración del sistema, propuesto en DSI 8.2. Las verificaciones que deben realizarse están en los catálogos de requisitos y excepciones y en el diseño detallado. Las pruebas unitarias están asociadas a los componentes, las de integración a grupos de componentes, las demás a todo el sistema, cada una con un propósito. Las pruebas del sistema son pruebas de integración del sistema completo, las de implantación verifican que el sistema funcionará en el entorno de operación y las de aceptación validan el cumplimiento de requisitos de usuario. Las pruebas unitarias, de integración y del sistema se llevan a cabo en la construcción, las otras en el entorno de implantación.

- **DSI 11: Establecimiento de Requisitos de Implantación**. Los requisitos de implantación son la documentación de usuario y los relativos al funcionamiento en el entorno de operación.
- **DSI 12: Aprobación del Diseño del Sistema de Información**. Esta actividad es un mero trámite administrativo.

7.4.8. Construcción del sistema de información

En este proceso se genera el código, se desarrollan los procedimientos de operación y seguridad y se elaboran los manuales de usuario y de explotación. También se realizan las pruebas unitarias, de integración y de sistema. Se define la formación del usuario final y si procede se construyen los procedimientos de migración de datos. La base para este proceso son las especificaciones de construcción, definidas en la actividad DSI 8.

Actividades de la construcción:

- **CSI 1: Preparación del Entorno de Generación y Construcción**. El objetivo es asegurar la disponibilidad de los medios para la construcción del sistema. Estos medios están formados por el hardware y software necesarios. El punto de partida de esta actividad son las especificaciones de construcción generadas en DSI 8.
- **CSI 2: Generación del Código de los Componentes y Procedimientos**. Se crea el código de los componentes partiendo del diseño junto con los procedimientos de operación y seguridad. Las pruebas unitarias y de integración se realizan en paralelo con lo anterior.
- **CSI 3: Ejecución de las Pruebas Unitarias**. Las pruebas unitarias se redactan una vez codificados los componentes según el plan de pruebas.
- **CSI 4: Ejecución de las Pruebas de Integración**. Las pruebas de integración comprueban que los componentes o subsistemas interactúan correctamente, es decir, el funcionamiento de sus interfaces. La estrategia de integración se establece en el plan de pruebas. Esta actividad se realiza en paralelo con CSI 2 y CSI 3, pero es preciso que los componentes sobre los que actúa hayan pasado las pruebas unitarias.
- **CSI 5: Ejecución de las Pruebas del Sistema**. Las pruebas del sistema comprueban la integración global del sistema, verificando el funcionamiento correcto de las interfaces entre subsistemas y con otros sistemas de información. En estas pruebas se comprueba la cobertura de los requisitos.

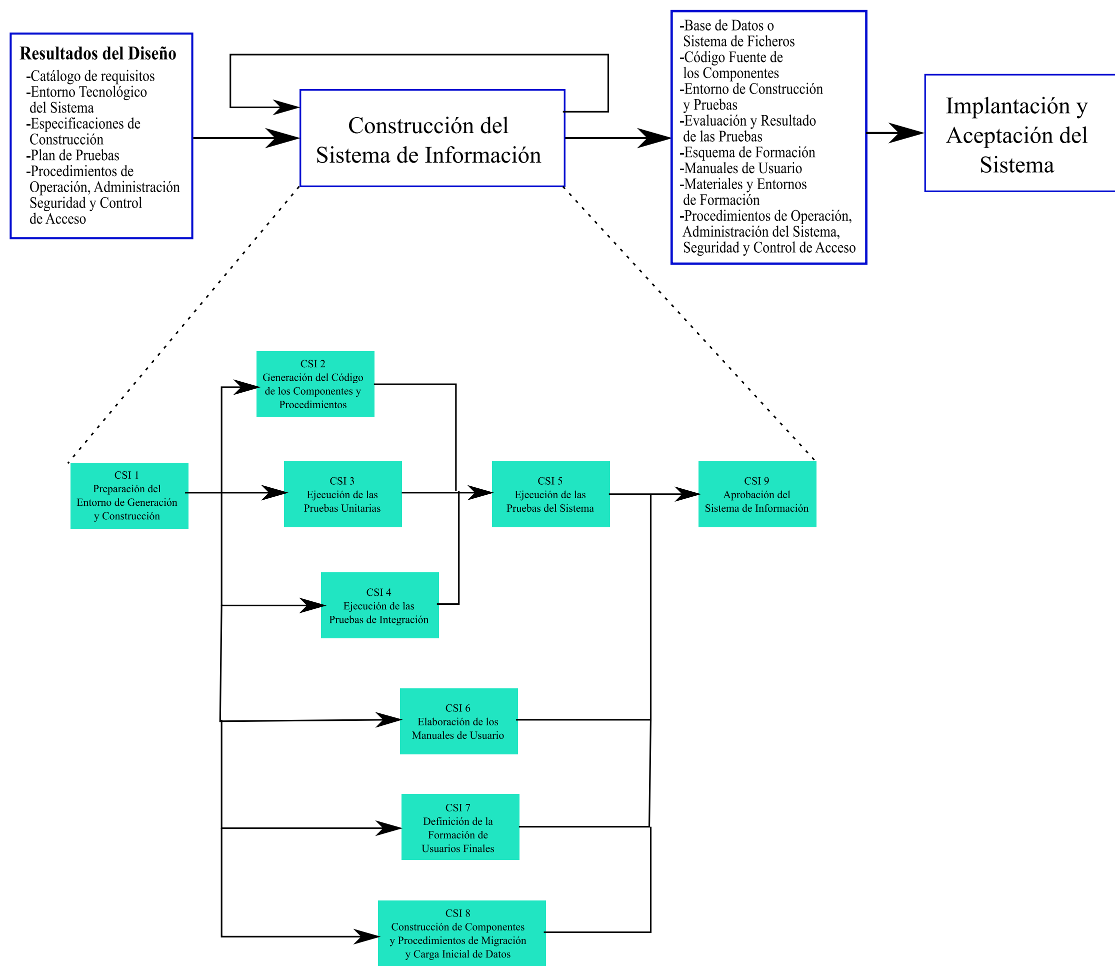


Figura 7.15: Entradas, Salidas y Actividades en la Construcción del Sistema de Información

- **CSI 6: Elaboración de los Manuales de Usuario**. Se elabora la documentación de usuario final y de usuario de explotación. Se parte de la especificación de requisitos de documentación de usuario
- **CSI 7: Definición de la Formación de Usuarios Finales**. Para determinar la formación necesaria de los usuarios se tienen en cuenta las características funcionales y técnicas del sistema y los requisitos de formación especificados en la tarea especificación de requisitos de implantación (DSI 11.2). La especificación de formación resultante consta de dos partes: Esquema de formación y Materiales y entornos de formación.
- **CSI 8: Construcción de los Componentes y Procedimientos de Migración y Carga Inicial de Datos** Esta actividad tiene la misma estructura que las anteriores: Primero se prepara la infraestructura tecnológica necesaria, luego se codifica y por último se realizan las pruebas.
- **CSI 9: Aprobación del Sistema de Información**. Esta actividad es un mero trámite ad-

ministrativo.

7.4.9. Implantación y Aceptación del Sistema

El objetivo del proceso es que el cliente acepte el sistema y pasarlo a producción. Se debe revisar el plan de implantación definido en el estudio de viabilidad del sistema (EVS). Los usuarios adecuados deberán colaborar en el proceso.

Se debe hacer una preparación previa del entorno de operación. Se realizan pruebas de dos tipos: 1) de implantación para asegurar que el sistema funciona incluso en condiciones extremas y 2) de aceptación, que las realizan los usuarios para validar que el sistema satisface sus necesidades. Posteriormente se prepara el sistema para su mantenimiento. Finalmente se determinan los servicios que requiere el sistema que se va a implantar.

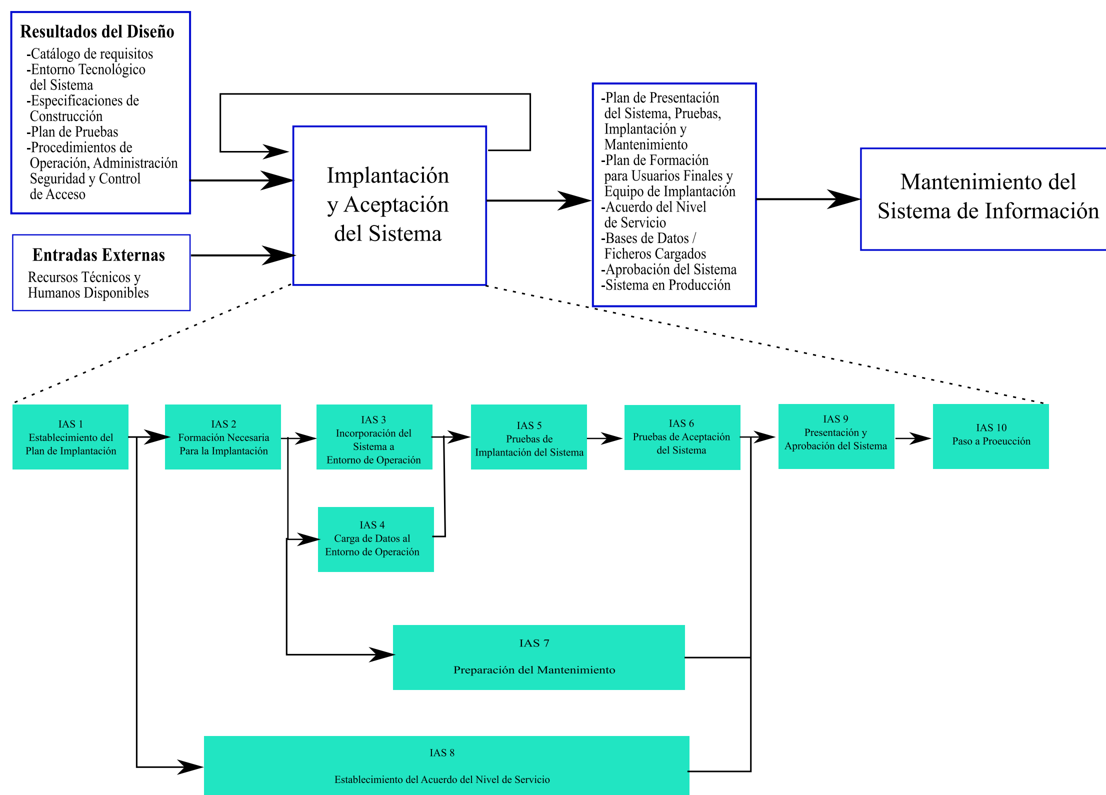


Figura 7.16: Entradas, Salidas y Actividades en la Implantación y Aceptación del Sistema

Lista de actividades:

- **IAS 1: Establecimiento del Plan de Implantación**. Después de revisar la estrategia de implantación del sistema(s) se decide si la implantación se puede llevar a cabo. En caso afirmativo se determinan los recursos humanos necesarios para la implantación.

- **IAS 2: Formación Necesaria para la Implantación**. Se establece un plan de formación para el equipo de implantación en función de los diferentes perfiles involucrados. Los usuarios tienen un plan ya definido en la actividad *definición de la formación de los usuarios finales* (CSI 7).
- **IAS 3: Incorporación del Sistema al Entorno de Operación**. Se garantiza que todos los recursos necesarios están disponibles y se realizan las pruebas de implantación y aceptación en el entorno real de operación. Se establecen los procedimientos de explotación y uso de las bases de datos.
- **IAS 4: Carga de Datos al Entorno de Operación**. Si el sistema sustituye a uno antiguo será necesaria una migración y carga inicial de datos. Esta necesidad se identifica en el estudio de viabilidad en la actividad selección de la solución (EVS 6). En la actividad diseño de la migración y carga inicial de datos (DSI 9) se habrá concretado la forma de hacerlo, y en la actividad construcción de los componentes y procedimientos de migración y carga inicial de datos (CSI 8) se habrá codificado.
- **IAS 5: Pruebas de Implantación del Sistema**. Estas pruebas verifican el funcionamiento en el entorno de operación y permiten que el usuario compruebe el funcionamiento del sistema desde su punto de vista. Las pruebas las realiza un equipo de usuarios.
- **IAS 6: Pruebas de Aceptación del Sistema**. Los usuarios finales realizan un conjunto de pruebas al sistema que validan el cumplimiento de los requisitos y redactan un informe. Los directores de los usuarios revisan los criterios de aceptación y dirigen las pruebas.
- **IAS 7: Preparación del Mantenimiento del Sistema**. Se intenta conseguir que el responsable de mantenimiento esté familiarizado con el sistema. Por este motivo esta persona formará parte del equipo de implantación. Es recomendable que exista una gestión de la configuración.
- **IAS 8: Establecimiento del Acuerdo de Nivel de Servicio**. Antes de aprobar el sistema hay que hacer tres cosas: determinar los servicios necesarios, especificar los niveles de servicio con los que se va a valorar la calidad de esa prestación y definir qué compromisos se adquieren.
- **IAS 9: Presentación y Aprobación del Sistema**. Tiene lugar una presentación al comité de dirección para aceptar formalmente el sistema.
- **IAS 10: Paso a Producción**. Se pasa la responsabilidad al equipo de mantenimiento. El entorno de producción tiene que cumplir todos los requisitos necesarios.

7.4.10. Mantenimiento de Sistemas de Información

El objetivo de este proceso es obtener mejora a partir de las peticiones de mantenimiento. El mantenimiento puede ser de 4 tipos:

1. **Correctivo**: Corrección de errores detectados.
2. **Evolutivo**: Cambios necesarios para incorporar nuevos requisitos de usuario o cambios en los mismos.
3. **Adaptativo**: Cambios para que el sistema siga funcionando en entornos distintos al origi-

nal.

4. **Perfectivo**: Mejoras en la calidad interna: reestructuración de código, definición más clara del sistema y optimización del rendimiento y eficiencia.

Los dos últimos no se contemplan en Métrica.

Cuando se registra una petición se designa a la persona que atenderá la petición. Se estudia el alcance del problema, las alternativas de solución y el impacto de las posibles soluciones en la organización.

El cambio seguirá el mismo ciclo de vida que el desarrollo de un sistema, es decir, los mismos procesos.

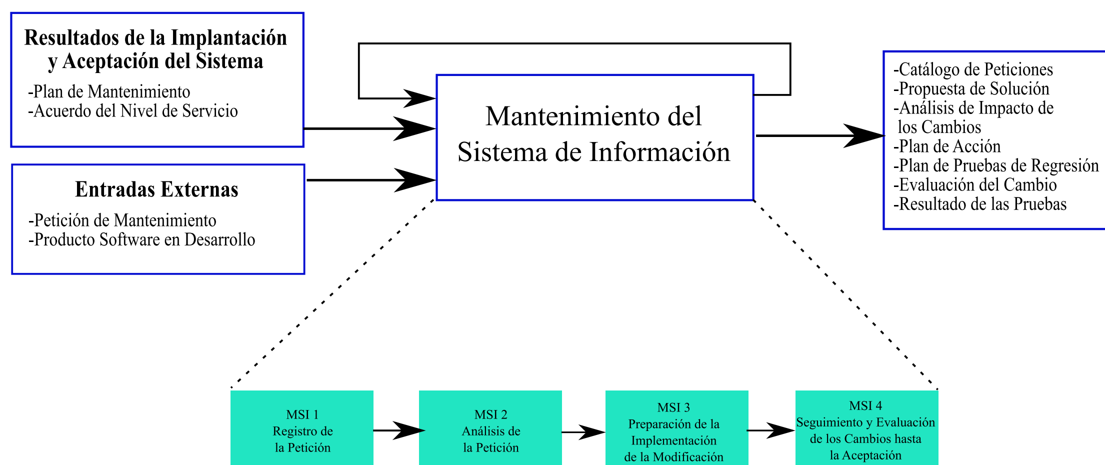


Figura 7.17: Entradas, Salidas y Actividades en el Mantenimiento de Sistemas de Información

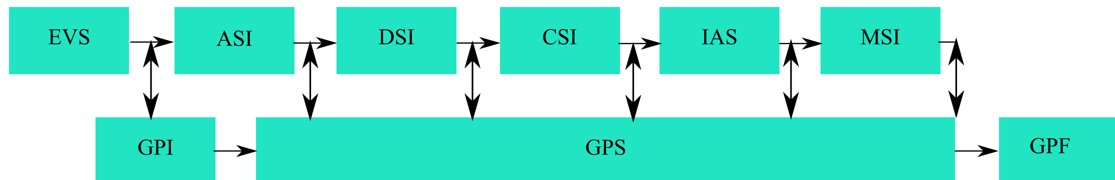
Lista de actividades:

- **MSI 1: Registro de la Petición**. Se establece un sistema de registro de las peticiones, que se presentan de un modo estandarizado. Después de registrar la petición se la clasifica y se comprueba su viabilidad.
- **MSI 2: Análisis de la Petición**. Se analiza el alcance de la petición respecto a los sistemas de información afectados. Posiblemente haya que considerar el estudio de viabilidad del sistema o el análisis del sistema. El enfoque varía en función del tipo de mantenimiento.
- **MSI 3: Preparación de la Implementación de la Solución**. Se hace un análisis de impacto para cada uno de los componentes afectados que permitirá establecer un plan de acción. Se especifican también las pruebas de regresión.
- **MSI 4: Seguimiento y Evaluación de los cambios hasta la Aceptación**. Se comprueba que se han modificado sólo los elementos afectados por el cambio y que se han realizado las pruebas. Se evalúa el cambio. Se realizan las pruebas de regresión y si el resultado es positivo se aprueba la petición.

Interfaces

Interfaz Gestión de Proyectos Software (GPS)

El objetivo de la gestión de proyectos es el control de recursos humanos y materiales. Existen tres grupos de actividades: de inicio del proyecto, de seguimiento y control y de finalización del proyecto.



Actividades de inicio del proyecto Tienen dos objetivos: Estimar el esfuerzo identificando los elementos que se deben desarrollar y planificar las actividades (recursos, planificación de tareas y calendario). Lista de tareas:

1. Estimación de esfuerzo. Tareas:
   - Identificación de elementos a desarrollar.
   - Cálculo del esfuerzo.
2. Planificación. Tareas:
   - Selección de la estrategia de desarrollo.
   - Selección de la estructura de actividades, tareas y productos.
   - Establecimiento del calendario de hitos y entregas.
   - Planificación detallada de actividades y recursos necesarios.
   - Presentación y aceptación de la planificación general del proyecto.

Actividades de seguimiento y control El objetivo es vigilar todas las actividades de desarrollo. El jefe de proyecto vigila cada tarea, centrándose en las que estén retrasadas, caso de que esto ocurra se averiguan las causas. Estas actividades de seguimiento se llevan a cabo a lo largo de todo el ciclo de vida. Actividades:

1. Asignación detallada de tareas.
   - Asignación de tarea.
2. Comunicación al equipo del proyecto.
   - Información al equipo de proyecto.
3. Seguimiento de tareas.

- Seguimiento de tareas.

Gestión de incidencias: Es un grupo de actividades de seguimiento y control. Una incidencia es un hecho inesperado que produce desviaciones respecto a lo planificado. Un caso especial son los cambios de requisitos.

4. Análisis y registro de la incidencia. Tareas:

   - Analizar impacto.
   - Propuesta de solución de la incidencia.
   - Registro de la la incidencia.

Gestión de cambios en los requisitos: Es mejor que estos cambios no ocurran, pero si así es, se plantean al comité de seguimiento. El impacto del cambio del requisito en términos de plazos y presupuestos se debe pactar con el cliente. Deberá existir un registro de cambios que refleje para cada cambio:

- Formulario de petición de cambio.
- Catálogo de necesidades.
- Análisis funcional del cambio.
- Estimación de esfuerzo.
- Variaciones en coste y plazos.

El control y seguimiento de los cambios forma parte de las actividades de seguimiento y control de todo el proyecto.

5. Petición de cambio de requisitos. Tarea:

   - Registro de la petición de cambio de requisitos.

6. Análisis de la petición de cambio de requisitos. Tareas:

   - Estudio de la petición de cambio de requisitos.
   - Impacto de la petición de cambio de requisitos.
   - Estudio de alternativas y propuesta de solución.

7. Aprobación de la solución. Tareas:

   - Aprobación de la solución.

8. Estimación del esfuerzo y planificación de la solución. Tarea:

   - Estimación de esfuerzo para el cambio.
   - Planificación de los cambios.

9. Registro del cambio de requisitos. Tareas:

   - Registro del cambio de requisitos.

10. Finalización de la tarea. Tareas:

    - Comprobación de la tarea.

11. Actualización de la planificación. Tareas:

    - Actualización de tareas.
    - Obtención de la extrapolación.
    - Elaboración del informe de seguimiento.

12. Reuniones de seguimiento. Tarea:

    - Reunión interna de seguimiento.

13. Aceptación. Tarea:

    - Verificación de aceptación interna.

    Actividades de finalización La finalización ocurre cuando el cliente expresa su conformidad.

14. Cierre del proyecto. Tareas:

    - Inclusión en histórico de proyectos.
    - Archivo de la documentación de gestión del proyecto.

Interfaz Aseguramiento de la calidad (CAL)

El objetivo es garantizar que el sistema resultante cumpla con unos requisitos mínimos de calidad. Se llega a este objetivo haciendo revisiones exhaustivas de todos los documentos producidos. El equipo que participa en el aseguramiento de la calidad es independiente al de desarrollo. Sus funciones son:

1. Identificar desviaciones respecto a los estándares aplicados, de los requisitos y de los procedimientos especificados.
2. Comprobar que se han llevado a cabo las medidas preventivas o correctoras necesarias.

El aseguramiento de la calidad está dividido en seis áreas, cada una de ellas cuenta con sus actividades y cada actividad con sus tareas. Estas áreas se corresponden con las etapas del desarrollo, es decir, el aseguramiento de la calidad se lleva a cabo durante todos los procesos del ciclo de vida. Veamos cada una de estas áreas una a una.

Estudio de viabilidad del sistema Se analizan las condiciones de desarrollo de cada una de las alternativas propuestas y las características que deben cumplir. Los riesgos específicos de los sistemas que se van a desarrollar pueden influir en la forma concreta en la que se aplique el plan de calidad, que puede estar definido en alguna norma de la propia empresa o seguir un estándar publicado. Actividades:

1. Identificación de las propiedades de calidad para el sistema.
   - Constitución del equipo de aseguramiento de la calidad.
   - Determinación de los sistemas de información objeto de aseguramiento de calidad.
   - Identificación de las propuestas de calidad.
2. Establecimiento del plan de aseguramiento de calidad.
   - Necesidad del plan de aseguramiento de calidad para las alternativas propuestas.
   - Alcance del plan de aseguramiento de calidad.
   - Impacto en el coste del sistema.
3. Adecuación del plan de aseguramiento de calidad.
   - Ajuste del plan de aseguramiento de calidad.
   - Aprobación del plan de aseguramiento de calidad.

Análisis del sistema de información Se detallan el plan de calidad y los estándares o normas a seguir. El grupo encargado de esta tarea revisa: catálogo de requisitos, modelos resultantes del análisis y plan de pruebas. Actividades:

1. Especificación inicial del plan de aseguramiento de la calidad.
   - Definición del plan de aseguramiento de calidad para el sistema de información.
2. Especificación detallada del plan de aseguramiento de calidad.
   - Contenido del plan de aseguramiento de calidad para el sistema de información.
3. Revisión del análisis de consistencia.
   - Revisión del catálogo de requisitos.
   - Revisión de la consistencia entre productos.
4. Revisión del plan de pruebas.
   - Revisión del plan de pruebas.
5. Registro de la aprobación del análisis del sistema.
   - Registro de la aprobación del análisis del sistema de información.

Diseño del sistema de información Se revisan los siguientes conjuntos de requisitos: los especificados en el análisis, los de las pruebas y los no funcionales. Actividades:

1. Revisión de la verificación de la arquitectura del sistema.

- Revisión de la consistencia entre productos del diseño.
- Registro de la aceptación de la arquitectura del sistema.

2. Revisión de la especificación técnica del plan de pruebas.
   - Revisión del diseño de las pruebas unitarias, de integración y del sistema.
   - Revisión del plan de pruebas.
3. Revisión de los requisitos de implantación.
   - Revisión de los requisitos de documentación de usuario.
   - Revisión de los requisitos de implantación.
4. Registro de la aprobación del diseño del sistema de información.
   - Registro de la aprobación del diseño del sistema de información.

Construcción del sistema de información Se revisan los estándares de codificación, de evaluación de las pruebas, manuales de usuario y formación. Se comprueba que las pruebas se han llevado a cabo según los criterios del plan de aseguramiento de la calidad.

1. Revisión del código de componentes y procedimientos.
   - Revisión de normas de construcción.
2. Revisión de las pruebas unitarias, de integración y del sistema.
   - Revisión de la realización de las pruebas unitarias.
   - Revisión de la realización de las pruebas de integración.
   - Revisión de la realización de las pruebas del sistema.
3. Revisión de los manuales de usuario.
   - Revisión de los manuales de usuario.
4. Revisión de la formalización a usuarios finales.
   - Revisión de la formalización a usuarios finales.
5. Registro de la aprobación del sistema de información.
   - Registro de la aprobación del sistema de información.

Implantación y aceptación del sistema Hay que comprobar que efectivamente existe un plan de implantación y que se han realizado las pruebas de implantación y de aceptación del estudio de viabilidad y la normalización del plan de aseguramiento de la calidad. Se debe verificar que se entrega el sistema a los responsables del mantenimiento en condiciones para que éste se pueda realizar. Lista de actividades y tareas:

1. Revisión del plan de implantación del sistema.
    - Revisión del plan de implantación del sistema.
2. Revisión de las pruebas de implantación del sistema.
    - Revisión de la realización de las pruebas de implantación del sistema.
    - Registro de la aprobación de las pruebas de implantación del sistema.
3. Revisión de las pruebas de aceptación del sistema.
    - Revisión de la realización de las pruebas de aceptación del sistema.
    - Revisión de la aprobación de las pruebas de aceptación del sistema.
4. Revisión del plan de mantenimiento del sistema.
    - Revisión del plan de mantenimiento del sistema.
5. Registro de la aprobación de la implantación del sistema.
    - Registro de la aprobación de la implantación del sistema.

Mantenimiento del sistema de información Se realizan revisiones periódicas para constatar que el mantenimiento se lleva a cabo del modo correcto. Puede ser necesario revisar:

- El contenido del plan de pruebas de regresión.
- La ejecución de las pruebas de regresión. En este caso se registra la aprobación de las pruebas.
- Las verificaciones y casos de prueba del plan de pruebas correspondientes a un cambio.
- Las incidencias.

Lista de actividades y tareas:

1. Revisión del mantenimiento del sistema de información.
    - Revisión del mantenimiento.
2. Revisión del plan de pruebas de regresión.
    - Comprobación de la existencia del plan de pruebas de la regresión.
3. Revisión de la realización de las pruebas de regresión.
    - Revisión de la realización de las pruebas de regresión.

Interfaz Gestión de la Configuración (GC)

La gestión de la configuración se ve en el capítulo de “Herramientas de desarrollo y validación” y sería redundante incluirlo aquí.

Interfaz Seguridad (SEG)

La palabra seguridad hace referencia a la gestión de riesgos. Esta interfaz pretende dotar a Métrica 3 de mecanismos de seguridad adicionales a los que tiene de por si la metodología. Hay dos tipos de actividades diferenciadas:

1. Actividades relacionadas con la seguridad intrínseca del sistema de información.
2. Actividades relacionadas con la seguridad del proceso de desarrollo.

Planificación de sistemas de información Si la organización no tiene definida una política en materia de seguridad habrá que hacerlo. Esa política influirá en las decisiones adoptadas en el proceso de planificación de sistemas de información.

1. Planificación de la seguridad requerida en el proceso planificación de sistemas de información.
2. Evaluación del riesgo para la arquitectura tecnológica.
3. Determinación de la seguridad en el plan de acción.
4. Catalogación de los productos generados durante el proceso de planificación de sistemas de información.

Estudio de viabilidad del sistema En función de la seguridad requerida se selecciona el equipo de seguridad, que se basará en la política de seguridad de la organización.

1. Estudio de la seguridad requerida en el proceso *estudio de viabilidad del sistema*.
2. Selección del equipo de seguridad.
3. Recomendaciones adicionales de seguridad para el sistema de información.
4. Evaluación de la seguridad de las alternativas de solución.
5. Evaluación detallada de la seguridad de la solución propuesta.
6. Catalogación de los productos generados durante el proceso de estudio de viabilidad del sistema.

Análisis del sistema de información Las funciones de seguridad son un servicio que garantiza la seguridad del sistema de información. Un mecanismo de seguridad es la lógica o algoritmo que lo implementa. Actividades:

1. Estudio de la seguridad requerida en el proceso de análisis del sistema de información.
2. Descripción de las funciones y mecanismos de seguridad.
3. Definición de los criterios de aceptación de la seguridad.
4. Catalogación de los productos generados durante el proceso de diseño del sistema de información.

Diseño del sistema de información Se minimizan los riesgos intrínsecos al sistema de información. Determinar el entorno tecnológico es importante porque sobre él se implementan las medidas de seguridad.

1. Estudio de la seguridad requerida en el proceso de diseño del sistema de información.
2. Especificación de requisitos de seguridad del entorno tecnológico.
3. Requisitos de seguridad del entorno de construcción.
4. Diseño de pruebas de seguridad.
5. Catalogación de los productos generados durante el proceso de construcción del sistema de información.

Construcción del sistema de información Se debe evitar que se filtren datos relativos al sistema de información. Se verifica el resultado de las pruebas de las funciones y mecanismos adicionales de seguridad. También se completa la definición de formación a usuarios finales. Actividades:

1. Estudio de la seguridad requerida durante el proceso de construcción del sistema de información.
2. Evaluación de los resultados de pruebas de seguridad.
3. Elaboración del plan de formación de seguridad.
4. Catalogación de los productos generados durante el proceso de construcción del sistema de información.

Implantación y aceptación del sistema Se especifican las actividades que tienen que ver con la seguridad del sistema construido, tanto intrínseca como las que tienen que ver con la seguridad del proceso. Asegura que se cubran los requisitos de seguridad a través de las pruebas de implantación. Lista de actividades:

1. Estudio de la seguridad requerida en el proceso de implantación y aceptación del sistema.
2. Revisión de medidas de seguridad del entorno de operación.
3. Evaluación de resultados de pruebas de seguridad de implantación del sistema.
4. Catalogación de los productos generados durante el proceso de implantación y aceptación del sistema.
5. Revisión de medidas de seguridad en el entorno de producción.

Mantenimiento de sistemas de información Actividades:

1. Estudio de la seguridad requerida en el proceso de mantenimiento de sistemas de información.
2. Especificación e identificación de las funciones y mecanismos de seguridad.
3. Catalogación de los productos generados durante el proceso de mantenimiento de sistemas de información.

7.5. Métodos de software libre: “cathedral” vs. “bazaar”

Internet ha supuesto una revolución, no sólo en el mundo de la informática, sino en todos los campos del conocimiento. Nunca antes la información había estado disponible como lo está ahora para todos. Internet es algo así como el sistema nervioso del mundo. Una de las consecuencias es que hay formas nuevas de trabajar, por ejemplo, ahora una persona puede colaborar con otras que no conoce en un proyecto. La información compartida se puede dejar en un almacén común y se definen políticas poco o nada restrictivas acerca de quién puede leer esa información o a quién se acepta como colaborador.

El inicio del movimiento del desarrollo de Software Libre a través de Internet ha supuesto también la aparición de nuevos métodos y la adaptación de otros a esta nueva forma de desarrollo. Las metodologías tradicionales presuponen una organización del reparto de tareas gestionado mediante responsables que distribuyen y supervisan el trabajo. Cuando se trata de trabajo cooperativo voluntario a través de la Red ese esquema ya no es válido y se plantean nuevas formas de coordinación distribuida del trabajo. Los términos “catedral” y “bazar” hacen referencia a una metáfora de ambos enfoques que propuso Eric S. Raymond en su famoso libro titulado precisamente así, “The Cathedral & the Bazaar” [Ray99]. Donde comparaba las metodologías tradicionales de desarrollo con la construcción y desarrollo de una catedral, en contraposición de la forma de crecimiento y expansión aparentemente caótica de un bazar.

Los métodos de desarrollo de software libre, realizados por programadores muy variados y con distintas formas de codificar, funcionan en la práctica porque todo el producto está disponible desde el primer momento públicamente para su revisión y comprobación. Se logran resultados extensos a base de pequeñas contribuciones individuales pero muy numerosas. Para poder aportar alguna contribución es necesario haber visto y comprendido el trabajo que ya está realizado y por tanto se produce una sinergia con los programadores predecesores que resulta en un desarrollo armonioso. Evidentemente en estos proyectos es necesario algún tipo de coordinación mínima que es llevada a cabo por el desarrollador más reconocido que habitualmente es el que más ha contribuido. Nos extenderemos más en el método “bazar” por ser la parte novedosa.

7.5.1. La catedral

Es el método tradicional y seguido por la mayor parte de los fabricantes de software hoy en día. Características:

1. El software de gran tamaño se construye como las catedrales, es decir, cuidadosamente planificado por equipos de expertos que se comunican lo justo entre sí.
2. Hay poco personal y bien escogido. Aumentar mucho el número de personas es caro y pasado cierto punto no acelera el desarrollo, sino que lo ralentiza.
3. No se publican versiones beta hasta poco antes de terminar.
4. El motivo de que las versiones se publiquen tarde en este modelo es que de lo contrario estarían plagadas de errores.
5. Los errores son difíciles de encontrar, requieren una fase de pruebas que se hace al final.
6. El código fuente se guarda a cal y canto.

7. Subyace a él una estructura organizativa piramidal.
8. El jefe de proyecto debe tener gran talento para el diseño.

7.5.2. El bazar

Es diferente al anterior en todos los puntos anteriores:

1. En un principio hay una idea, pero no se tiene una imagen clara de en que se convertirá al final.
2. Existe una ingente cantidad de personas en su elaboración y cuantos más mejor.
3. En el momento en el que se puede publicar una versión se hace, aunque esté muy incompleta.
4. Los errores se encuentran con facilidad porque hay muchas personas trabajando simultáneamente en ello. Su descubrimiento se pone en marcha desde el principio.
5. El código es abierto, o lo que es lo mismo, cualquiera puede leerlo y modificarlo.
6. La estructura organizativa es difusa, hay una serie de normas para participar pero no una jerarquía claramente definida. Una persona puede contribuir durante toda la vida del proyecto o de un modo fugaz.
7. El coordinador del proyecto más que tener talento tiene que tener olfato para ver cuando una idea de otro es buena e incorporarla.

El *kernel* de Linux es el paradigma número uno que sigue la filosofía del bazar. Es un software de gran tamaño y complejidad, iniciado en principio como un pequeño proyecto por Linus Torvalds tomando como base el sistema operativo Minix (un antiguo UNIX para clónicos de IBM-PC). El proyecto sigue en la actualidad mas vivo que nunca y cada día va ganando cuota de mercado a sus competidores (a algunos los ha dejado atrás ya, p. ej: SCO-Unix. Según Eric S. Raymond en su libro [Ray99]: “La Catedral y el Bazar”, las conclusiones que se sacan del método bazar son:

1. Todo trabajo de software comienza a partir de las necesidades personales del programador. No es lo mismo trabajar para un proyecto a cambio de un salario que trabajar gratis en algo en lo que se encuentra una satisfacción personal o se cubre una necesidad. Lo segundo motiva más.
2. Los buenos programadores saben qué escribir. Los mejores qué reescribir (y reutilizar). Es más fácil partir de una solución aunque sea incompleta y mala que de cero. Linus lo que hizo no fue construir un sistema operativo desde la primera línea de código hasta el final (probablemente aún estaría en ello), en lugar de eso, tomó como punto de partida Minix. Por tanto, una forma de iniciar un proyecto puede ser buscar un proyecto similar ya hecho y tomarlo como guía.
3. Contemple desecharlo, de todas formas tendrá que hacerlo. Al final todo el código de Minix fue reemplazado, pero mientras existió fue un armazón a partir del cual se pudo ir cambiando partes. El código de partida no es importante, de hecho seguro que tiene un

montón de errores y no es adecuado a nuestras necesidades, solo sirve para comprender el problema.

4. Si tienes la actitud adecuada, encontrarás problemas interesantes.
5. Cuando se pierde el interés en un programa, el último deber es dejarlo en herencia a un sucesor competente.
6. Tratar a los usuarios como colaboradores es la forma más apropiada de mejorar el código, y la más efectiva de depurarlo.
7. Publique rápido y a menudo, y escuche a sus clientes.
8. Dada una base suficiente de desarrolladores asistentes y *beta-testers*, casi cualquier problema puede ser caracterizado rápidamente, y su solución ser obvia al menos para alguien.
9. Las estructuras de datos inteligentes y el código burdo funcionan mucho mejor que en el caso inverso.
10. Si usted trata a sus analistas (*beta-testers*) como si fueran su recurso más valioso, ellos le responderán convirtiéndose en su recurso más valioso.
11. Lo más grande después de tener buenas ideas es reconocer las buenas ideas de sus usuarios. Esto último es a veces lo mejor.
12. Frecuentemente, las soluciones más innovadoras y espectaculares provienen de comprender que la concepción del problema era errónea.
13. La perfección (en diseño) se alcanza no cuando ya no hay nada que agregar, sino cuando ya no hay algo que quitar.
14. Toda herramienta es útil empleándose de la forma prevista, pero una “gran” herramienta es la que se presta a ser utilizada de la manera menos esperada.
15. Cuando se escribe software para una puerta de enlace de cualquier tipo, hay que tomar la precaución de alterar el flujo de datos lo menos posible, y nunca eliminar información a menos que los receptores obliguen a hacerlo.
16. Cuando su lenguaje esté lejos de uno Turing-completo, entonces las “ayudas” sintácticas pueden ser su mejor amigo.
17. Un sistema de seguridad es tan seguro como secreto. Cuídese de los secretos a medias.
18. Para resolver un problema interesante, comience por encontrar un problema que le resulte interesante.

Tutorial posterior

Resumen de contenidos

Una metodología es una forma de concretar todo lo que se ha visto hasta ahora, que es más que nada una enumeración ordenada de técnicas. Las metodologías tienen un ciclo de vida concreto, son o bien orientadas a objetos o estructuradas, realizan las pruebas de un modo, etc.

Proceso Unificado de Rational (RUP)

Es una metodología desarrollada por las personas que han creado el UML, por lo tanto este se usa a lo largo de todas las etapas. La documentación recomendada es el libro [JBR00].

Extreme Programming

Es otra metodología de desarrollo orientado a objetos pensada para proyectos de tamaño pequeño o mediano donde se tiene contacto muy directo con el cliente. Contiene conceptos muy interesantes: la programación por parejas, la publicación de versiones cada muy poco tiempo, la codificación de las pruebas para cada unidad antes que la propia unidad, breves reuniones diarias, la integración secuencial prácticamente a diario, la propiedad compartida del código (todo el mundo puede modificar cualquier parte del código).

Métrica 3

Es la metodología definida oficialmente en este país por el Ministerio de Administraciones Públicas, por lo tanto, es especialmente recomendada para aquellas personas que tengan pensado hacer oposiciones. Su punto de partida es la versión anterior (2.1). Es en parte orientada a objetos y en parte estructurada. Su ciclo de vida es el definido en la norma ISO-12.207. Incluye un conjunto de procesos que no forman parte del ciclo de vida a los que llama interfaces. Tiene en cuenta la tecnología cliente servidor y el desarrollo de interfaces gráficas de usuario (IGU).

Métodos de software libre: “Cathedral vs. Bazaar”

Este apartado es más que nada una comparación entre las metodologías de desarrollo seguidas por empresas que ocultan el código fuente (*cathedral*) y el fenómeno del software libre (*bazaar*) donde el código está a la vista de todos (*open source*) y puede contribuir mucha gente.

Ejercicios y actividades propuestas

Como actividad general y dado que en este capítulo se han presentado metodologías de desarrollo de software, sería conveniente que el alumno realizara pruebas con los ejemplos desarrollados en a lo largo de los temas anteriores para entender cómo se aplicarían estas técnicas sobre ellos, imaginando diferentes escenarios de desarrollo donde se aplicarían las metodologías aquí explicadas u otras similares.

1. ¿Qué información contiene una vista en el RUP?
2. ¿Cuáles son los defectos principales de “*Extreme Programming*”?
3. ¿Cuáles son los pasos a seguir para crear una prueba de unidad en “*Extreme Programming*”?
4. ¿En qué consiste la programación por parejas?
5. Características principales de *catedral*

6. ¿Qué tipo de producto es “Microsoft Windows”: Catedral o Bazar? ¿y “GNU/Linux”?. Compare “Microsoft Windows” y “GNU/Linux” desde los siguientes puntos de vista:

    - Facilidad de instalación, configuración y uso.
    - Documentación existente.
    - Soporte al usuario.
    - Número de errores.
    - Eficiencia en la gestión de recursos.

Capítulo 8

Herramientas de desarrollo y validación

8.1. Herramientas CASE

CASE (Computer Aided Software Engineering ≡ Ingeniería del Software Asistida por Ordenador) es un conjunto de programas creados para automatizar las tareas mecánicas del desarrollo de programas. Existen herramientas para cada fase del desarrollo: análisis, diseño, codificación y pruebas pasando también por la generación de interfaces gráficas de usuario y de la documentación. En términos generales se entiende por herramienta CASE como un programa orientado al diseño que es capaz de generar el código de forma automática. Las herramientas CASE pueden estar especializadas en un solo aspecto o pueden cubrir todo el desarrollo. En este último caso estaríamos hablando de herramientas CASE integradas o I-CASE.

8.1.1. Funciones de las herramientas CASE

Como se ha dicho el objetivo fundamental es automatizar tareas mecánicas. Esto se traduce en una mejora de la productividad del equipo en términos de tiempo y costes y de la calidad del producto.

Las tareas que suelen cubrir son las siguientes:

- Representación de diferentes modelos gráficos que describen el sistema, generalmente los diferentes tipos de diagramas del UML pero también con otros formalismos como los relativos a la programación estructurada.
- Desarrollo de interfaces gráficos de usuario. Este apartado se ha conseguido bastante bien.
- Depuración de programas.
- Generación de código partiendo del diseño detallado.
- Generación de documentación.
- Generación automática de casos de prueba.

Existe un límite a la capacidad de estas herramientas: disponer de la mejor herramienta del mercado no supone ventajas si no se sabe diseñar.

8.1.2. Clasificación de las herramientas CASE

Una clasificación se debe hacer en base a un criterio. Los posibles criterios son [Som01]:

- *Perspectiva funcional*: función que realizan.
- *Perspectiva de proceso*: las actividades de proceso que ayudan.
- *Perspectiva de integración*: modo en el que están organizados los módulos que ayudan a realizar las actividades del proceso.

Si atendemos a la *funcionalidad*, los tipos de herramientas son:

1. Herramientas de planificación de sistemas de información.
2. Herramientas de análisis y diseño.
    - Herramientas de modelado.
    - Herramientas de creación de prototipos y simulación.
    - Herramientas para el diseño de interfaces.
3. Herramientas de programación.
    - Herramientas de codificación convencionales.
    - Herramientas de codificación de cuarta generación
    - Herramientas de programación orientadas a objetos.
4. Herramientas de integración y prueba.
5. Herramientas de gestión de prototipos.
6. Herramientas de mantenimiento.
    - Herramientas de ingeniería inversa.
    - Herramientas de reestructuación y análisis de código.
    - Herramientas de reingeniería.
7. Herramientas de gestión de proyectos.
    - Herramientas de planificación de proyectos.
    - Herramientas de seguimiento de requisitos.
    - Herramientas de gestión y medida.
8. Herramientas de soporte
    - Herramientas de documentación.

- Herramientas para software de sistemas.
- Herramientas de control de calidad.
- Herramientas de bases de datos.

Si atendemos a las *actividades* cubiertas tenemos:

1. Herramientas de reingeniería.
2. Herramientas de pruebas.
3. Herramientas de depuración.
4. Herramientas de análisis.
5. Herramientas de procesamiento de lenguajes.
6. Herramientas de apoyo a métodos.
7. Herramientas de construcción de prototipos.
8. Herramientas de gestión de la configuración.
9. Herramientas de gestión del cambio.
10. Herramientas de documentación.
11. Herramientas de edición.
12. Herramientas de planificación.

En función de la *integración* de estas herramientas tenemos:

1. *Herramientas integradas (I - CASE)*: Cubren todo el ciclo de vida, desde la especificación hasta el mantenimiento. Son una o varias herramientas distintas que trabajan coordinadamente. Esto tiene varias ventajas sobre las soluciones parciales, a saber:
    - La salida de una fase es la entrada de la siguiente, si la herramienta que trata ambas fases es la misma o están bien integradas entre sí, el tránsito es más fácil y automatizado.
    - La gestión de configuración es más sencilla.
2. Herramientas no integradas.
    - *U - CASE*: Herramientas CASE de alto nivel.
    - *L - CASE*: Herramientas CASE de bajo nivel.

Las I - CASE pueden estar integradas en cinco aspectos (según Wasserman, 1990):

1. Integración de la plataforma: Las herramientas funcionan todas en el mismo hardware y sistema operativo.
2. Integración de datos: Las herramientas son capaces de intercambiar información entre ellas. Hay varios niveles:
    - Ficheros compartidos: Ficheros con el mismo formato para todas.

  - Estructuras de datos compartidas.
  - Repositorio compartido.

3. Integración de la presentación: Todas las herramientas usan una misma interfaz de usuario.
4. Control: Las herramientas son capaces de llamarse entre ellas.
5. Integración de procesos:

8.2. Gestión de la configuración

El título de este capítulo es engañoso, debería llamarse Gestión de Archivos del Proyecto, pero es el nombre con el que se le conoce en la literatura que existe del tema. Este capítulo es un resumen del interfaz de Gestión de la Configuración definido en Métrica 3 ampliado con el documento de "Gestión de configuración" de Angélica de Antonio, que se encuentra en `http://www.ls.fi.upm.es/udis/docencia/plani/G_Configuracion.pdf` y otras fuentes.

El propósito de esta importante actividad es asegurar la integridad de los productos generados a lo largo del ciclo de vida del proyecto, poder trazar su evolución y facilitar la visibilidad sobre ellos.

La gestión de la configuración se realiza durante todo el ciclo de vida del sistema y facilita su mantenimiento porque permite valorar el impacto de los cambios. Esta actividad tiene sentido sobre todo cuando se trabaja en equipo.

8.2.1. Terminología y definiciones básicas

- *Elemento de Configuración Software* (ECS): Cada uno de los componentes que se va a guardar en el sistema. Los siguientes componentes siempre deberían ser ECSs: la especificación del sistema, el plan del proyecto, especificación de requisitos, prototipos, el diseño preliminar, el diseño detallado, el código fuente, los ejecutables, todos los manuales (usuario, operación e instalación), el plan de pruebas, los casos de prueba ejecutados y sus resultados, informes, peticiones de mantenimiento, productos hardware y software utilizados junto a sus manuales, diseños de bases de datos, información de dichas bases de datos, etc.
- *Versión (o Revisión)*: Un ECS en un momento dado. Las versiones se van numerando.
- *Cadena de Revisión*: Son las sucesivas versiones que se van produciendo. Cada nueva versión parte de la anterior y sólo existe una versión nueva.

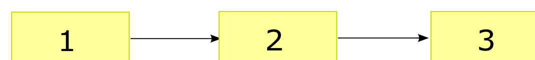


Figura 8.1: Cadena de revisión

- *Variante*: Son diferentes versiones de un mismo ECS que coexisten en el tiempo. La razón de que existan es que a veces un mismo ECS debe satisfacer distintos requisitos en un momento dado. Puede ocurrir que una variante pase por varias revisiones.
    - *Variante temporal*: Se mezclará con otra en un momento dado. Es conveniente que no pase mucho tiempo para que no difieran en mucho y no sea difícil su fusión. Se pueden desarrollar para explorar distintas posibilidades de desarrollo o para hacer pruebas y luego tirarlas.
    - *Variante permanente*: No se llegan a mezclar nunca. Se pueden desarrollar por dos motivos: Diferentes requisitos para clientes distintos o desarrollos adaptados a plataformas diferentes.
- *Grafo de Evolución (o de Revisión)*: Representación gráfica de las distintas versiones de un ECS a lo largo del tiempo.

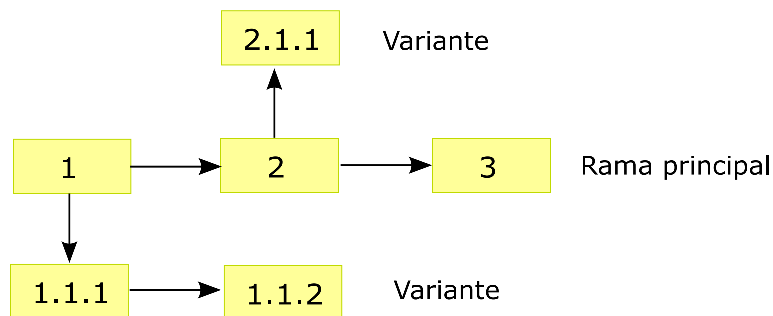


Figura 8.2: Grafo de evolución de un ECS con variantes

- *Línea base*: Es un punto de referencia en en proceso de desarrollo en el que se aprueba uno o varios ECS a través de una revisión técnica formal. El o los elementos que han sido aceptados se pueden usar como base para desarrollar otras versiones, pero sólo podrán ser parte de una línea base cuando el nuevo elemento pase otra revisión. Hay varios tipos de líneas base.
- *Configuración*: Es un conjunto de ECS y su versión. Está pensada para un entorno particular. Pueden existir varias configuraciones distintas de un mismo proyecto.
- *Release*: Es una configuración que se va a comercializar.

8.2.2. Identificación de la configuración

Se trata de ver un método general para seleccionar qué elementos vamos a considerar como ECSs y asignarles nombres. Los pasos a seguir en este método son los siguientes[1]:

1. **Establecimiento de una jerarquía preliminar**: Consiste en obtener la estructura de lo que va a ser el sistema software y sus elementos principales.

[1]Este método es lo que teóricamente debe hacerse pero para la mayoría de los proyectos puede ser un poco farragoso. Lo más práctico es quedarse con un subconjunto de todo esto en función de lo que permita la herramienta de Gestión de Configuración de la que se disponga

2. **Selección de los elementos de configuración**: Para esta selección se tiene que llegar a un compromiso entre dos necesidades contrapuestas: 1) No tener una cantidad inmanejable de información y 2) Tener todos los ECS necesarios para no perder visibilidad sobre el producto.
3. **Establecer relaciones entre ECSs**: Los ECSs son objetos relacionados entre ellos. Estas relaciones permiten ver que ECSs se pueden ver afectados cuando hay un cambio sobre uno de ellos. Los tipos de relaciones son:
   - *Equivalencia*: Por ejemplo hay varios archivos en distintos sitios que son el mismo.
   - *Composición*: Es un ECS que en realidad está formado por varios, por ejemplo, el diseño detallado está formado por diagramas de clases, diagramas de estados, etc.
   - *Dependencia*: Expresa que un cambio en un ECS supone cambios en otro. Sirven para facilitar la trazabilidad de los requisitos.
   - *Derivación*: Un ECS se puede generar de forma automática partiendo de otros. Por ejemplo el código fuente genera el código objeto.
   - *Sucesión*: Es el grafo de la evolución que ha sufrido un ECS.
   - *Variante*: Un ECS modificado que coexiste con otras versiones del mismo.
4. **Definir un esquema de identificación**: Es forma de localizar inequívocamente cada ECS. Se puede definir de varias maneras, sea cual sea debe incluir la siguiente información: Nombre, Fecha de creación, Tipo de elemento, Proyecto al que pertenece, Línea base, Fase y subfase a las que pertenece, Número y fecha de la Versión y Localización (idioma).
5. **Definición y establecimiento de líneas base**: Se trata de establecer unos hitos a los largo del desarrollo. Lo normal es que esos hitos se establezcan al final de una fase del ciclo de vida. Al principio del proyecto se deben haber establecido cuales van a ser los ECSs que contenga cada línea base. Los tipos de líneas base más importantes son:
   - *Línea base funcional*: Indica las funciones que realizará el sistema. Se establece cuando termina la fase de análisis y especificación. Incluye documentos de requisitos, de planificación, de costes y de interrelación con otros sistemas.
   - *Línea base de asignación de funciones*: Indica la documentación que acompañará a cada elemento software que se ha identificado. Se establece al final de la fase de análisis y especificación de requisitos software.
   - *Línea base de desarrollo*: Recoge el diseño, codificación y pruebas del sistema en desarrollo. El número de elementos de los que consta va aumentando.
   - *Línea base de producto*: Contiene el producto terminado y documentos asociados a las pruebas. Se establece después de que éste ha sido verificado y validado.
6. **Definición y establecimiento de bibliotecas de software**: Una biblioteca es una colección de software y documentación. La más importante y a veces la única es la biblioteca de trabajo. Contiene las especificaciones, diseño, código fuente y pruebas. No se hace control de cambios sobre ella. Existen más tipos de bibliotecas pero no se pretende dar una visión excesivamente detallada.

- *Biblioteca de trabajo*: Es el conjunto de elementos con los que está trabajando el equipo de programación. Ellos lo gestionan.
- *Biblioteca maestra*: Almacena *releases* del sistema y los ECS asignados a líneas base. Esta biblioteca está sujeta a un control de cambios formal.
- *Repositorio de software*: Guarda líneas base validadas y verificadas.

8.2.3. Control de cambios

Esta sección es importante. El objetivo es que exista un cierto orden cada vez que alguien solicite un cambio sobre algún ECS. Los cambios consisten en la corrección de defectos o en ampliaciones del sistema. La lista de acciones que se siguen para hacer un cambio son las siguientes:

- *Presentación de la solicitud de cambio*: Por la detección de fallos o por nuevos requisitos.
- *Alta, clasificación y registro*: Se asigna un identificador a la solicitud. Se debe poder trazar la evolución que va a seguir. Se archiva la solicitud en una base de datos.
- *Generación de informe de cambio*: Es redactado por el jefe de proyecto y se dirige al Responsable del Control de Cambios (RCC).
- *Aprobación o rechazo de la solicitud*: El RCC aprueba o rechaza la solicitud y le asigna una prioridad.
- *Evaluación de la solicitud*: El jefe de proyecto evalúa el esfuerzo técnico: horas de trabajo, coste e impacto en otras partes del sistema.
- *Generación de orden de cambio*: Es un documento generado por el RCC que contiene el tipo de cambio, las restricciones a tener en cuenta y los criterios de revisión y auditoría.
- *Ejecución del cambio*.
- *Certificación del cambio*: Se realiza una auditoría para certificar que el cambio se ha producido correctamente. Se liberan los ECS modificados.

8.2.4. Generación de informes de estado

El objetivo de esta actividad es mantener informados a las personas involucradas en el proyecto del estado de la configuración. Se producen dos tipos de documentos: Registros e Informes.

Registros

Son documentos o bases de datos con información relacionada con la gestión de la configuración de un determinado producto. Los registros importantes son:

- *Registro de solicitud de cambio*. Información más importante que contiene: Código de solicitud, solicitante, fecha, descripción del cambio, ECS relacionados, documentación relacionada y resolución tomada acerca del cambio.
- *Registro de cambios*: Es un registro para fines estadísticos. Contiene información de creación y modificación de ECS y cambios sobre ellos.

- *Registro de incidencias*: Existe para registrar las necesidades de cambio y cómo se reacciono ante ellas. Recoge: Descripción, resultado de la incidencia y fechas de la incidencia y de su cierre.
- *Registro de modificaciones*: Para realizar el seguimiento de las modificaciones.

Informes

Son documentos con un formato específico. Puede producirse bajo demanda o como resultado de una planificación. Informes más importantes:

- *Informe de estado de los cambios*: Es un resumen del estado de las solicitudes de cambio producidas en un periodo.
- *Inventario de ECS*: Lista de ECS de un proyecto.
- *Informe de incidencia*: Resumen del estado en el que se encuentran las incidencias producidas en un periodo.
- *Informe de modificaciones*: Resumen de las modificaciones producidas en un periodo.

8.2.5. Auditoría de la configuración

Una auditoría es un examen o prueba que se realiza a un sistema para comprobar y certificar si cumple determinados requisitos, es decir, es una actividad de control de la calidad. En la gestión de la configuración existen tres tipos de auditoría:

- *Auditoría funcional*: Comprueba que se han realizado las pruebas relativas a los ECS auditados y que satisfacen los requisitos exigidos.
- *Auditoría física*: Comprueba la integridad y corrección de los elementos físicos de documentación de una línea base.
- *Revisión formal de certificación*: Comprueba que los ECS se comportan correctamente en su entorno normal de producción.

8.2.6. Plan de gestión de la configuración

Es un documento que se debe producir al principio del proyecto. Define la política a seguir. Según el estándar IEEE tiene que tener los siguientes apartados:

1. **Introducción**
    a) Propósito del plan.
    b) Alcance.
    c) Definiciones y acrónimos.
    d) Referencias.
    e) Definición de alto nivel del proceso de GCS.
2. **Especificaciones de gestión**

a) Organización.
b) Responsabilidades.
c) Implantación del plan de gestión de configuración.
d) Políticas, directivas y procedimientos aplicables.

3. **Actividades de gestión de configuración**

   a) Identificación de la configuración.
   b) Control de la configuración.
   c) Contabilidad de estado de la configuración.
   d) Auditoría de la configuración.

4. **Control de suministradores**
5. **Recogida y retención de registros**

8.2.7. Herramientas de la Gestión de la Configuración

No sería práctico intentar llevar a la práctica el contenido de la sección anterior sin una herramienta que automatice los aspectos mecánicos. Dentro del mundo del software libre la herramienta más popular para gestión de la configuración ha sido CVS (Concurrent Versioning System). El sucesor de esta herramienta es SubVersion aunque CVS se sigue utilizando.

Uno de los problemas que debe resolver cualquier herramienta de gestión de la configuración es la modificación simultanea de un mismo archivo por dos usuarios distintos. Subversion y CVS implementan un algoritmo *copiar-modificar-mezclar* en lugar de *bloqueo-modificación-desbloqueo*. En este algoritmo, el cliente de cada usuario se conecta al repositorio del proyecto y descarga una copia local de los archivos y directorios del repositorio. Los usuarios pueden entonces trabajar en paralelo, modificando sus copias privadas. Por último, todas las copias privadas se mezclan en una nueva versión final. El sistema de control de versiones tiene un algoritmo de mezcla de archivos, pero no siempre funciona y los usuarios tendrán que resolver manualmente el problema.

Términos básicos

Repositorio: Almacén central donde están las copias maestras. El repositorio central es un árbol de directorios.

Módulo: Un directorio o árbol de directorios del repositorio maestro.

RCS (Revision Control System): Conjunto de utilidades de bajo nivel.

***Check out*:** Hacer una copia de un fichero.

Revision: Etiqueta numérica que identifica la versión de un fichero.

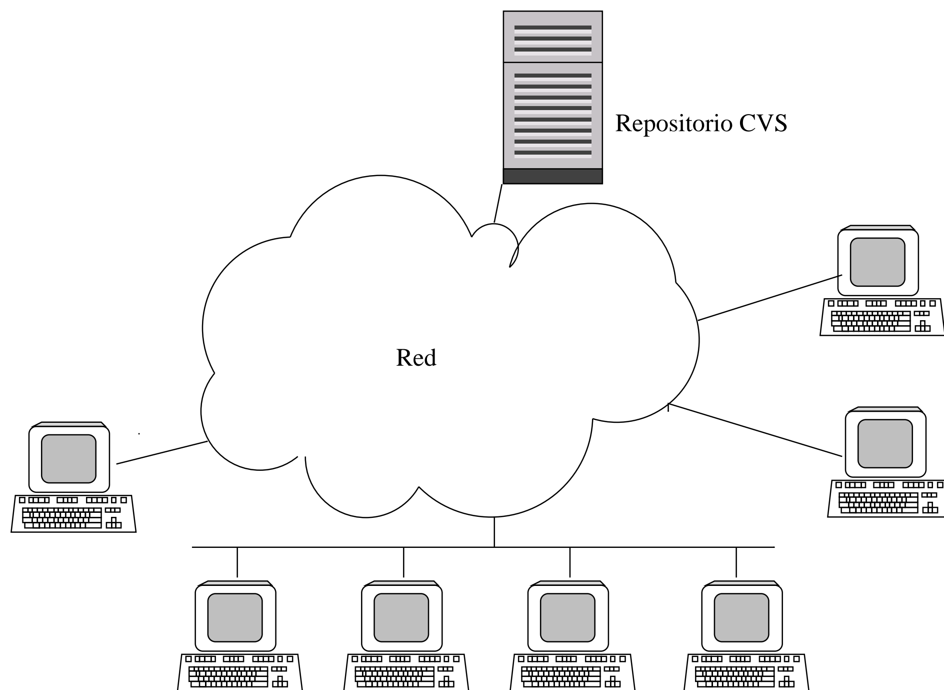


Figura 8.3: CVS

Características principales de CVS

Funciona como un almacén de archivos. Cada archivo puede tener varias versiones identificadas por un número. Además para cada versión se guarda la fecha y el autor de la modificación entre otras cosas. En resumen las características más importantes son las siguientes:

- Licencia GPL.
- Arquitectura cliente-servidor sobre Internet.
- Existen versiones para UNIX y Windows.
- Se puede sacar una instantánea (el conjunto de todos los archivos) del estado en el que estaba el proyecto en cualquier momento temporal.
- Se pueden comparar diferentes versiones de un mismo archivo.
- Se puede pedir toda la historia de cambios de un archivo.
- Está pensado para proyectos de tamaño pequeño o mediano.
- Se maneja con una interfaz basada en linea de comandos aunque los entornos de desarrollo tienen una interfaz que facilita el uso.

Limitaciones de CVS:

- La gestión de permisos es problemática.

- Platea dificultades en renombrar los archivos y borrar directorios.

SubVersion

Es una herramienta similar pero es más moderna que CVS. Tiene todas las características anteriores pero corrige algunos problemas. Diferencias con CVS:

- Algoritmos mejorados: algunos algoritmos pasan a tener complejidad constante en lugar de lineal. Por la red sólo se transmiten las diferencias entre los ficheros en vez del fichero entero como en CVS.
- En vez de tener un número de versión para cada archivo hay un único número de versión para todo el contenido del repositorio que se incrementa cada vez que alguien actualiza una copia.
- Gestiona mejor los archivos binarios porque se utiliza el mismo algoritmo que para los de texto.
- Modificaciones atómicas: cada vez que se cambia un grupo de archivos o directorios se hacen las modificaciones sobre todos ellos o sobre ninguno.

8.2.8. Software libre

El software libre es aquel que puede ser obtenido, modificado y distribuido libremente. No se debe confundir con el software gratuito (*freeware*), que es el que puede usarse sin pagar pero que no se puede modificar. El software de *dominio público* es aquel que no tiene ningún tipo de licencia y la única obligación que se tiene es consignar el nombre del autor en aquellas aplicaciones que se deriven de él.

Richard Stallman introdujo una definición de software libre, según la cual el software se considera libre si se puede hacer lo siguiente:

1. Ejecutar el programa con cualquier finalidad: comercial, educativo, privado, etc.
2. Acceder al código fuente para modificarlo.
3. Copiar el programa para ayudar a cualquiera.
4. Mejorar el programa y hacer públicas dichas mejoras.

Licencias

Una licencia por definición es un contrato que establece un derecho de uso sobre algún producto. Este derecho de uso puede tener ciertas restricciones. En las licencias libres el programador inicial retiene los derechos de autor, pero permite a los usuarios normalmente más cosas que las licencias cerradas. Otorga a los posibles receptores de la distribución por parte de terceras personas los mismos derechos de quienes distribuyen.

Existen muchas licencias libres de las cuales vamos a ver las principales:

- *GPL*: Es una de las licencias más importantes. GPL (GNU General Public License, Licencia Pública General de GNU). Es la licencia bajo la que se distribuye el proyecto GNU, que surgió en 1984 para desarrollar un sistema operativo de tipo Unix gratuito. La licencia GPL está pensada para que no se pueda hacer uso de partes de software libre en programas no libres, también obliga a distribuir el código fuente junto con los binarios.
- *LGPL*: LGPL (Lesser Gnu Public Licence, Licencia GPL reducida). La licencia anterior tiene algunas restricciones como por ejemplo que un componente que sea GPL no puede ser utilizado en ningún software comercial. La licencia LGPL permite ser utilizada junto a software no libre.
- *BSD y BSD modificada*: Es menos restrictiva que las anteriores. Sólo exige que se cite la frase: *All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed by the University of California, Berkeley and its contributors.* y debe incluirse una vez por cada componente BSD. La licencia BSD modificada no tiene la clausula de publicidad.
- *MIT* (Massachusetts Institute of Technology, Instituto Tecnológico de Massachusetts): Permite usar el software libremente, copiarlo, publicarlo, distribuirlo, sub-licenciarlo, siempre que se incluya la nota de copyright en todas las distribuciones.

Concepto de *copyleft*

Dentro de la comunidad de software libre se desea que el trabajo realizado por una persona no pueda ser utilizado por otros dentro de software propietario al que luego el autor no tiene derechos de acceso *software hoarding* (acaparamiento de software). Según Stallman *Copyleft dice que cualquiera que redistribuye el software, con o sin cambios, debe dar la libertad de copiarlo y modificarlo más. Copyleft garantiza que cada usuario tiene libertad.*

Definición de *Open Source*

En la página `http://www.opensource.org/docs/definition.html` se encuentra la definición más precisa de lo que se considera "Open Source" (denominación más genérica del software libre) desde el punto de vista de Sourceforge para permitir que un proyecto se aloje en sus servidores.

Como su nombre indica "Open Source" significa que el código fuente está disponible para el público en general. Existen una serie de condiciones que se deben cumplir:

1. *Distribución libre*: No se exige el pago de derechos de patente (*royalties*) por el código. Motivo: Eliminar la tentación de tirar por la borda ganancias a largo plazo por ganancias a corto plazo. En caso contrario, los colaboradores se podrían ir.
2. *Código fuente*: Debe estar disponible el código fuente y deberá ser gratis. No se pueden utilizar ofuscadores de código (programas que reescriben el código fuente dejando la misma funcionalidad pero haciéndolos ilegibles para humanos). Motivo: No se puede hacer evolucionar una aplicación si no se dispone de código fuente.

3. *Modificación del código*: La licencia permite la modificación del código fuente y el resultado deberá ser distribuido en los mismos términos que el original. Motivo: Es necesario hacer modificaciones y distribuirlas para que exista un desarrollo evolutivo rápido.
4. *Integridad del código fuente del autor*: Se puede restringir que se modifique el código fuente *sólo* si se permite la distribución de parches que modifiquen el programa. Las modificaciones deben llevar un nombre o un número de versión diferente del software original. Motivo: Los autores tienen derecho a que se sepa lo que ha hecho cada uno y los usuarios quien ha hecho el software que usan.
5. *No se discriminará a personas o grupos*: La licencia no debe excluir a nadie. Motivo: Cuanta más diversidad, mejor.
6. *No se discriminarán tipos de actividades*. Por ejemplo: Empresas, Investigación, etc. Motivo: Se intenta evitar trucos legales que permita que el código abierto sea usado comercialmente.
7. *Distribución de la licencia*: Los derechos asociados a la licencia se aplican a todos aquellos a los que se redistribuya la aplicación sin que se tenga que aplicar una licencia adicional. Motivo: Se trata de que no se pueda restringir el acceso al software por medios indirectos.
8. *La licencia no será específica de un producto*: A cualquier parte de una aplicación se la aplican los mismos derechos que a toda la aplicación.
9. *La licencia no establece restricciones para otro software*: El resto del software que se distribuya no será afectado por las restricciones del GPL.

Un ejemplo de repositorio de software libre es SourceForge `http://www.sourceforge.net/`. Es un sitio web de desarrollo *open source*. Los servicios gratuitos que proporciona SourceForge son: repositorio CVS, listas de correo, seguimiento de errores, foros, gestión de tareas software, *backups* y administración basada en web.

8.3. Entornos de desarrollo de interfaces

En este apartado se estudiarán algunos de los entornos de programación orientados al desarrollo de interfaces existentes actualmente. Se hará especial énfasis en los de libre distribución para que el alumno pueda experimentar con ellos. Los entornos que en la actualidad tienen más ímpetu de desarrollo y mayores capacidades son: GNOME/GTK+ y KDE/QT. En los ejemplos incluidos aquí nos centraremos en uno de ellos (GTK+), ya que ambos son muy similares y tienen herramientas equivalentes.

8.3.1. Introducción

A continuación se describen algunas herramientas *freeware* de creación de interfaces. El contenido de esta sección será familiar para aquellas personas que hayan desarrollado algún proyecto con compiladores con nombre más comercial como Visual Basic, Delphi o C++ Builder. La razón por la que se incluye este apartado es que las interfaces constituyen el 50 % de

las líneas de código de muchos proyectos y al ser un tipo de actividad susceptible de reutilizar código y al ser bastante mecánica existen herramientas pensadas para acelerar su desarrollo. La idea en la que se basan estos entornos es muy parecida:

1. Se tiene una librería extensa de componentes gráficos (etiquetas, editores, botones, etc) que se muestran en una ventana. En cualquier caso, si se necesita un componente que no está es casi seguro que se puede encontrar en algún lugar de internet.
2. Se tiene otra ventana que representa la interfaz gráfica que se está construyendo donde se depositan los componentes del punto anterior. El modo de hacerlo puede ser arrastrar y soltar.
3. Los componentes tienen variables que se manipulan en tiempo de diseño en la ventana correspondiente.
4. El entorno genera el código que produce la interfaz con un aspecto como el que se está diseñando, encargándose de manipular los componentes de forma adecuada pero sin que el programador tenga que escribir el código.
5. El entorno puede ir solo o integrado en un entorno de desarrollo con editor, compilador, etc.

8.3.2. Componentes

Un componente es un objeto, que como todo objeto consta de código y datos. De ellos nos importan tres cosas:

1. *Propiedades*: Son los valores de las variables públicas. Son un conjunto de variables que se pueden manipular en tiempo de diseño y de ejecución y que controlan el aspecto externo de dicho componente.
2. *Eventos*: Son sucesos como puede ser la pulsación de una tecla o el movimiento del ratón. Existe un **gestor de eventos** que captura estos sucesos y manda una señal al objeto adecuado, por ejemplo, si se pulsa el ratón sobre un botón el gestor de eventos invoca el método *OnClick* de ese botón.
3. *Métodos*: Son las funciones del objeto. Se pueden escribir en el diseño, es decir, el desarrollador escribe lo que quiere que ocurra cuando se invoca el método, por ejemplo, que cuando se pulsa un botón (se invoca *OnClick*) el título del botón cambie.

Los componentes están estructurados en jerarquías, de modo tal que cuando hay que crear un nuevo componente lo mejor es derivarlo de uno antiguo reutilizando tanta funcionalidad como sea posible. Tipos de componentes. La forma de escribir componentes, la forma en la que se comunican con el resto del software e incluso el sitio donde pueden ejecutarse definen una taxonomía muy interesante al respecto.

1. *ActiveX*: Es un tipo de componente que se distribuye como binario, así que puede estar escrito en cualquier lenguaje de programación. El objeto tiene una interfaz estándar para que otros objetos se comuniquen con él. Los programas que usan controles ActiveX se

llaman contenedores. El contenedor de un control es una aplicación capacitada para el manejo de ActiveX que actúa como soporte de interfaz de usuario para dicho control. Se puede por ejemplo, presentar un botón que, una vez pulsado, envíe un mensaje al control. O también responder a diversos sucesos o mensajes especiales que se manden desde el control al contenedor. Un ejemplo de contenedor es un navegador, que puede mostrar controles ActiveX en una página web incluso aunque el control provenga de un ordenador remoto.

2. *VCL* (Visual Component Library): Son componentes gráficos para la interfaz de usuario. Constan de propiedades, métodos y eventos. El código que maneja cada evento es escrito en un método. Las propiedades se pueden editar en modo de diseño, de modo que no es necesario escribir líneas de código para inicializar cada componente. Las propiedades y métodos pueden tener los siguientes tipos de acceso:

    - *Privado*: El código que no pertenece al componente no puede acceder a esta parte.
    - *Protegido*: Sólo pueden acceder a esta parte los componentes que hereden de este.
    - *Publico*: Se puede acceder libremente a estos métodos y propiedades pero no para depuración.
    - *Publicado*: Se puede acceder tanto para manipulación como para depuración.

8.3.3. Creación de interfaces de usuario

Esto podría definirse como el apartado “artístico”. No debe ser despreciado por ello, es quizás lo más importante. Una interfaz es la parte externa del programa, lo que el usuario ve. Si está mal diseñada sacará una pobre impresión de ella aunque funcione perfectamente. El usuario ha de comprender la mecánica y la operativa que le ofrece la interfaz (sintaxis, órdenes, códigos, abreviaciones e iconos ...), todo esto supone un trabajo de memorización. Una buena interfaz minimiza el esfuerzo mental que se demanda al usuario.

Con el fin de que esa dificultad se minimice es importante establecer un sistema de ayudas adecuado. Estas ayudas se centrarán en la operativa y la aclaración de funciones de los elementos visuales o acústicos. Por otra parte, el diseño de la interfaz supone que el usuario hace un modelo mental. Es importante la coherencia entre las distintas partes para que ese modelo se mantenga. Por tanto, el objetivo de una interfaz es que tenga las siguientes propiedades:

1. Facilidad de aprendizaje y uso.
2. El objeto de interés ha de ser de fácil identificación.
3. Diseño ergonómico, por ejemplo: barra de acciones o iconos preferentemente a la derecha.
4. Las interacciones se basarán preferiblemente en acciones físicas sobre elementos de código visual o auditivo (iconos, imágenes, mensajes ...) antes que en selecciones tipo menú con sintaxis y órdenes.
5. Las operaciones serán rápidas, incrementales y reversibles con efectos inmediatos (de ser posible).
6. Los mensajes de error será adecuado a los conocimientos que tiene el usuario, deberán ser útiles y descriptivos.

7. Como el elemento fundamental es la pantalla del ordenador, se cuidará especialmente la organización, combinando información y elementos de interacción.
8. El tratamiento del color debe contar con una serie de normas tales como luminosidad, saturación, tono, etc. Es mejor tener una profundidad de color lo mayor posible (8 bits dan 256 colores, 16 bits 65536 colores, etc). Los iconos deben ser descriptivos.
9. Tipografía: Se procurará combinar letras mayúsculas y minúsculas, procurando no mezclar en pantalla más de dos tipos y tres medidas diferentes de letra.
10. Multimedia: Los gráficos móviles o las grabaciones pueden aportar dinamismo a la interfaz, pero no es bueno abusar de ellos.

8.3.4. Metodología

Se puede entender una interfaz como una aplicación en sí misma, y por tanto, es razonable que sigamos los mismos pasos que en la construcción de cualquier sistema, esto es, definir una serie de etapas para su construcción. Esas etapas son las de siempre: Especificación, diseño, codificación y en lugar de pruebas lo llamaremos test de usabilidad, porque es eso lo que se pretende comprobar.

El test de usabilidad valida el diseño de la interfaz de usuario. La forma de realizarlo es sentar a un usuario delante de la máquina con una lista de tareas y pedirle que exprese verbalmente lo que va pensando mientras tanto. De esta forma se puede ver cuales son los problemas que se encuentra la gente

La forma de definir una interfaz del sistema para un sistema basado en casos de uso tiene una serie de pasos y subpasos:

1. Dibujar las pantallas de interacción para los distintos actores-usuarios. Para ello:
   - Copiar el modelo mental del usuario.
   - Revisar los elementos del modelo del mundo interesantes para el actor-usuario.
   - Visualización típica de los elementos del modelo del mundo.
   - Información relevante para el actor.
   - Metáforas de interacción válidas.
2. Especificar el diálogo que da solución a cada caso de uso que se soluciona con la interacción con esta interfaz. Puede especificarse este diálogo de varias maneras, dependiendo de la complejidad de la interfaz definida (en esta etapa se sugiere escoger el mínimo nivel de detalle posible, para dar más libertad de diseño en las etapas posteriores):
   - Por medio de una descripción textual de su funcionamiento
   - Por medio de diagramas de interacción que muestren la secuencia de operaciones entre los objetos de interfaz y los actores involucrados.
   - Por medio de diagramas de estados, donde se muestre claramente los estados de la interfaz.
   - Por medio de un prototipo funcional, en términos de la interacción con el usuario.

3. Definir restricciones para la comunicación con actores y sistemas.
   - Describir en el detalle del actor o de la relación con el caso de uso particular.

8.3.5. Heurísticas de usabilidad

Son el decálogo a seguir para conseguir un buen resultado en este parámetro. Estas normas fueron compiladas por R. Molich y J. Nielsen en 1990.

1. *Visibilidad del estado del sistema*: El sistema debe mantener informado al usuario de lo que está pasando para obtener de él una realimentación adecuada.
2. *Correspondencia entre el sistema y el mundo real*: El sistema se debe expresar en los términos del usuario en lugar de en jerga técnica. La información debe aparecer en el orden en el que ocurre en el mundo real.
3. *Control del usuario y libertad*: Los usuarios de vez en cuando cometen errores y deben tener alguna forma sencilla de salir del estado provocado. Se debe soportar deshacer y rehacer (*undo* y *redo*).
4. *Consistencia y estándares*: No se deben emplear diferentes palabras para expresar lo mismo en diferentes partes del sistema. Seguir alguna convención.
5. *Prevención de errores*: Es mejor tener un diseño cuidadoso para prevenir los errores que mensajes de error.
6. *Instrucciones accesibles*: La información acerca de cómo usar el sistema debe estar visible para el usuario cuando lo necesite.
7. *Flexibilidad y eficiencia de uso*: Debe existir alguna forma de que las acciones más comunes se puedan realizar rápidamente, por ejemplo con combinaciones de teclas.
8. *Diseño minimalista*: No deben existir más elementos gráficos de los necesarios pues el exceso de información enturbia la comprensión.
9. *Ayudar a los usuarios con los errores*: Los mensajes de error deben estar expresados en lenguaje sencillo (sin códigos de error), indicando el problema y vías de solución.
10. *Ayuda y documentación*: Debe ser fácil buscar información en la documentación, debe estar centrada en las tareas de usuario (por ejemplo con una lista de pasos a seguir) y no debe ser demasiado larga.

8.3.6. Glade

Glade es un programa gratuito de desarrollo de interfaces. Usa las librerías de GTK+ y de GNOME. Genera el código que crea la interfaz en varios lenguajes: C, C++ Ada, Perl y Eiffel (con lo que puede considerarse como una herramienta CASE). También es capaz de crear interfaces dinámicas usando *libglade*. La herramienta se compone de tres ventanas:

1. Ventana principal: Barra de menú, la barra de herramientas y una lista de ventanas de alto nivel.
2. El editor de propiedades.

3. La paleta: En ella están desplegados los elementos que se pueden poner en la interfaz de usuario. Están divididos en tres categorías: GTK+ Básico, GTK+ Avanzado y Gnome.

La forma de utilizar la herramienta es: Primero se escoge el elemento *ventana* y entonces se pueden poner otros elementos sobre ella. Para organizar los elementos Glade utiliza cajas. Para poner más de un elemento en la ventana hay que poner primero cajas. Las hay de seis tipos: caja horizontal, caja vertical, tabla, posiciones ajustables, caja de botones horizontal, caja de botones vertical y las cajas a su vez se pueden anidar. Cuando se crean cajas horizontales y verticales hay que especificar el número de filas y columnas.

Cuando se selecciona un componente la ventana de propiedades muestra sus propiedades. Existe un árbol de componentes que se puede ver seleccionando: “View” y “Show widget tree”. También hay un editor de menús, tanto los normales como los emergentes, que pueden ser anidados. El editor también crea manejadores de señales para cada opción, que son las funciones invocadas cuando una opción de menú es seleccionada.

Un *evento* es un tipo de suceso, como puede ser presionar seleccionar una opción de menú o pulsar un botón. Cuando ocurre un evento el sistema llama a una función que lo trata y es en esa función donde se escribe el código con las instrucciones que se deben ejecutar para responder al evento. Por ejemplo, si se pulsa una opción de menú `File`→`Save` en un editor de texto, habrá que guardar el texto en un fichero. Glade genera automáticamente los nombres de las funciones que responden a los eventos y al menos cada elemento tiene uno.

8.3.7. GTK+

GTK+ (*GIMP ToolKit*) es una librería en lenguaje C para crear interfaces gráficas de usuario. Tiene licencia GPL, lo cual significa que es *open source* y de libre distribución. Fue diseñado originalmente como una herramienta de desarrollo del GIMP (*General Image Manipulation Program*) y está construido sobre GDK (*GIMP Drawing Kit*) que es un añadido de las librerías Xlib.

GTK+ es un interfaz de programación de aplicaciones orientadas a objetos (API). Aunque está escrito en lenguaje C, fue pensado con la idea de las clases y los punteros a funciones.

Al igual que el anterior permite usar gran variedad de componentes (botones, cajas, etc), pero tiene una flexibilidad mayor en el sentido de que los componentes están formados a su vez partiendo de otros componentes, que lógicamente pueden ser reemplazados. Por ejemplo, un botón puede tener como hijo un componente etiqueta para representar el texto, pero se puede sustituir por un *bitmap*. Por último, GTK+ define un conjunto de tipos dinámico propio con reglas nuevas acerca de como gestionarlos.

8.3.8. Anjuta

Es un entorno de desarrollo escrito por un estudiante de informática de último curso llamado Kh. Nabakumar Singh (Anjuta es el nombre de su novia). Esta pensado para proyectos en C/C++ para GTK+/GNOME, es decir, para GNU/Linux. Anjuta es libre y su código fuente está disponible en la red. Proporciona una IGU (interfaz gráfica de usuario) dividida en tres partes:

- Editor: Muestra el texto coloreado. Tiene las funciones estándar de Crear, Salvar, Buscar, etc.

- Proyecto.
- Mensajes.

Las funcionalidades ofrecidas son accedidas a través de un sistema de menús que se pueden desmontar de su lugar original. Las funciones más comunes pueden encontrarse en las cuatro barras de herramientas. Desde el entorno se puede compilar y ejecutar el programa. Puede asimismo configurarse para especificar rutas de ficheros *include*, rutas de librerías, librerías para enlazar, macros, tipos de alertas del compilador, optimizaciones del código, etc. El entorno consta asimismo de algunas utilidades para gestionar proyectos. Existe también la posibilidad de depurar el programa, o lo que es lo mismo, de ejecutarlo paso a paso, poner puntos de ruptura, evaluar expresiones, ver los registros de la CPU, etc.

Tutorial posterior

Resumen de contenidos

En capítulo se han colocado todos aquellos temas que no tienen entidad suficiente como para ser un capítulo por sí mismos, es decir, el capítulo es una recopilación de conceptos interesantes y prácticos acerca de la ingeniería del software.

Herramientas CASE

Una herramienta CASE es una aplicación diseñada para facilitar tareas mecánicas que se realizan en las diferentes etapas del ciclo de vida. Hay varios tipos y las mejores son las que cubren todas las etapas del ciclo de vida y pensadas además para una metodología concreta. Se incluye una clasificación de herramientas CASE.

Gestión de la configuración

Es el conjunto de prácticas y técnicas que ayudan a gestionar todos los documentos (código, manuales, etc.) que se generan a lo largo del ciclo de vida del proyecto. Existen herramientas que lo soportan, como CVS del que se hace una pequeña descripción. Los temas que importan en la gestión de la configuración son: cuáles son los documentos que hay que guardar, cómo se almacenan, cómo se relacionan entre sí y cómo se gestionan los cambios. **El plan de gestión de la configuración** es un documento que define la política a seguir. Se da una plantilla de dicho documento. **CVS** es una herramienta de libre distribución para la gestión de la configuración que se puede encontrar en cualquier sistema de tipo UNIX ó GNU/Linux. Se describe brevemente por estar muy extendida. **Sourceforge** `http://sourceforge.net/` es un ejemplo de almacén centralizado en Internet para el desarrollo de software de libre distribución. Se hace un listado de sus características más relevantes y se define lo que se entiende por software libre.

Entornos de desarrollo de interfaces

El desarrollo de interfaces es importante por el porcentaje tan elevado de líneas de código que suponen en un proyecto (en torno al 50 %). Para este fin se han desarrollado herramientas CASE concretas que facilitan el trabajo mecánico. Los **componentes** son módulos de software reutilizable, sobre todo en interfaces gráficas. Tipos de componentes: ActiveX y VCL. En la **creación de interfaces de usuario** se define un conjunto de objetivos que tiene que cumplir toda interfaz, todos ellos están orientados a usuarios, una metodología para el desarrollo de interfaces y heurísticas de usabilidad. Se ve una somera descripción de algunas herramientas de libre distribución para desarrollo de interfaces: Glade, Gtk y Anjuta.

Ejercicios y actividades propuestas

1. ¿Qué es mejor en principio, una herramienta I-CASE ó una U-CASE?
2. ¿Cuáles son los objetivos de la gestión de la configuración?
3. En la gestión de la configuración, ¿cómo se llama lo que se entrega al usuario?
4. ¿Qué es y cómo se define una línea base?
5. ¿Qué criterios se emplean para decir que una interfaz de usuario es usable?

Modificaciones en la guía

Los cambios más importantes de la guía son todo lo referente al UML, contenido en el capítulo 1.

Se ha añadido un gráfico de ejemplo de DFD en el capítulo 1, página 3.

Se ha ampliado el resumen de los conceptos importantes en orientación a objetos en el capítulo 1, página 4.

Se ha eliminado el resumen de UML del tutorial posterior correspondiente al capítulo 1.

Se han eliminado las siguientes referencias a páginas web que ya no están disponibles:

- Capítulo 1
  - `liinwww.ira.uka.de/bibliography/SE`
  - `usuarios.lycos..es/oopere/uml.htm`
- Capítulo 2: La referencia: `www.cibertienda.org` parece estar temporalmente no disponible en la red. No se ha eliminado porque hay numerosas páginas que hacen referencia a esta aplicación. Se puede encontrar información relativa en `www.onirica.com/es/productos/cibertienda`
- Capítulo 4
  - `www.oprenresources.com/es/magazine/tutoriales/corba`
- Capítulo 5
  - `www.bagley.org/doug/shootout/`
  - `internet.ls-la.net/mirrors/99bottles/`

Se han añadido las siguientes referencias a páginas web:

- Capítulo 1
  - `www.eclipse.org`

  - eclipse-plugins.2y.net/eclipse/index.jsp

- Capítulo 4

  - java.sun.com/developer/onlineTraining/corba/corba.html

Bibliografía

[Bau72] F. L. Bauer. Software Engineering. *Information Processing*, (71), 1972.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[FW94] Neville J. Ford and Mark Woodroffe. *Introducing Software Engineering*. Prentice-Hall, 1994.

[JBR00] Ivar Jacobson, Grady Booch, and James Rumbaugh. *El proceso unificado de desarrollo de software*. Addison Wesley, 2000.

[Pre01] Roger S. Pressman. *Ingeniería de Software: un Enfoque Práctico*. McGraw-Hill, 2001.

[PV96] Mario G. Piattini Velthuis. *Análisis y diseño detallado de aplicaciones informáticas de gestión*. RAMA, 1996.

[Ray99] Eric S. Raymond. *The Cathedral & the Bazaar*. O'Reilly & Associates, Inc., October 1999. http://tuxedo.org/˜esr/writings/cathedral-bazaar/.

[Sch01] Stephen R. Schach. *Object oriented and classical software engineering*. Mc GrawHill, 2001.

[Som98] Ian Sommerville. *Ingeniería de Software*. Addison-Wesley Iberoamericana, 1998.

[Som01] Ian Sommerville. *Software Engineering*. Addison-Wesley, 2001.

Índice alfabético

Abstract factory, 69
Aceptación, 90
ActiveX, 172
Actividades, 132
Actor, 112
Adapter, 69
Adecuación, 3
Almacén, 154
Análisis, 41, 152
    Clase, 115
    Modelo, 115
    Paquete, 115
        Paquete de servicio, 115
    RUP, 115
Análisis costo-beneficio, 94
Análisis de riesgo, 94
Análisis de sistema de información, 149
Anjuta, 176
Anti patrón, 65
Arquitecto, 121
Arquitectura
    Análisis, 116
    Descripción, 112, 116
    Diseño, 119
    Implementación, 121
Arquitectura de capas, 109
Artefactos, 115, 120
Aseguramiento de la calidad, 148
Ayuda, 175

Balking, 71
Bazaar, 154
Brainstorming, 49
Bridge, 70
Broker, 68
BSD, 170
Builder, 69

C++, 5
Código
    Propiedad compartida, 131
Cache management, 70
Capa intermedia, 108
Caso de prueba, 122
Caso de uso, 112
    Análisis, 117
    Diseño, 119
    Instancia, 112
    Realización, 118
Casos de uso, 52, 53, 104
    Actor, 52, 53
        Rol, 53
    Actores abstractos, 57
    Casos de uso abstractos, 57
    Casos de uso de implementación, 57
    Casos de uso esenciales, 57
    Casos de uso primarios, 57
    Casos de uso secundarios, 57
    Casos de uso temporales, 57
    Detallar, 114
    Estructura, 114
    Extends, 18
    Include, 18
    Priorizar, 114
    Realización, 115
Cathedral, 154
Chain of responsability, 70
Check out, 167
Ciclo de vida, 5
    Cascada con reducción de riesgos, 10

Cascada con subproyectos, 9
Cascada incremental, 10
Ciclo de vida en cascada, 7
Ciclo de vida en V, 8
Diseño, 117
Espiral, 10
Fuente, 13
Sashimi, 9
Cierre, 90
Clase, 20
Abreviada, 22
Abstracta, 20
Análisis, 117
Asociación, 23
Agregación, 23
Calificación, 23
Composición, 23
Dependencia, 26
Herencia y generalización, 25
Interfaces, 22
Multiplicidad, 23
Navegabilidad, 23
Reflexiva, 24
Rol, 23
Atributos, 20
Diseño, 119
Estereotipo, 28
Implementación, 122
Método, 21
Nombre, 20
Notas, 29
Operaciones, 21
Paquete, 29
Responsabilidades, 21
Subsistema, 29
Clase abstracta, 20
Clase del diseño, 118
Cliente, 130
Cliente/Servidor, 132
code-and-fix, 2
Codificación
Estándares, 130
Prueba de unidad, 128
Unidad, 128
Command, 70
Componente, 120
Componente de prueba, 122
Componentes, 172
Composite, 69
Construcción, 153
Construcción del sistema de información, 150
Corrección, 3

Decorator, 70
Defecto, 122
Delegation, 68
DFD, 3
Diagrama de actividades, 34
Actividades, 35
Actividades concurrentes, 35
Bifurcaciones, 35
Indicaciones, 35
Punto final, 35
Punto inicial, 34
Transiciones, 35
Diagrama de clases, 27
Diagrama de colaboración, 32
Diagrama de componentes, 36
Diagrama de despliegue, 37
Nodo, 36
Diagrama de estados, 32
Diagrama de objetos, 28
Diagrama de secuencias, 31
Activación, 31
Condicional, 31
Creación, 32
Destrucción, 32
Invocación recursiva, 32
Iteración, 32
Línea de vida, 31
Tiempos de transición, 31
Diccionario de datos, 4
Directorio, 80
Diseñador de pruebas, 122
Diseñar prueba, 123
Diseño, 153
Revisión, 75

RUP, 117
Subsistema, 118
Validación, 75
Verificación, 75
Diseño del sistema de información, 149
Disponibilidad, 3
Documentación, 79, 175
Dynamic linkage, 70

Eficiencia, 3
Entrevistas
Fases, 44
Entrevistas genéricas, 43
Errores, 129, 175
Especificación, 41
Especificación de procesos, 3
Especificaciones de la base de datos, 94
Especificaciones de sistema, 94
Especificaciones de subsistema, 94
Especificaciones del programa, 94
Estudio de viabilidad, 94, 148, 152
Evaluación de prueba, 122
Evaluar prueba, 123
Eventos, 172
Explicativo, 80
Extreme programming, 124

Facade, 70
Factory method, 69
Fases ciclo de vida, 5
Análisis, 6
Codificación, 6
Diseño, 6
Mantenimiento, 6
Pruebas, 6
Fiabilidad, 3
Filter, 69
Finalización, 89, 148
Flujo de sucesos, 112
Flujo de trabajo, 113, 116, 119, 121
Flujos de trabajo, 123
Flyweight, 70
Framework, 65

Gestión de cambios, 147
Gestión de configuración, 161
Gestión de la configuración, 151, 162
Gestión de proyectos, 146
Gestor de eventos, 172
Glade, 175
Glosario, 112
GNU, 170
GPL, 170
GTK+, 176
Guarded suspension, 71

Heurísticas de usabilidad, 175
Historias de usuario, 125

I-CASE, 161
IGU, 132
Implantación y aceptación, 150
Implantación y aceptación del sistema, 153
Implementación, 120
Implementar prueba, 123
Indicadores, 90
Ingeniería de sistemas, 1
Ingeniería del software, 1
Ingeniero de componentes, 121, 122
Ingeniero de pruebas de integración, 122
Ingeniero de pruebas de sistema, 122
Inicio de proyecto, 146
Inmutable, 69
Instrucciones de soporte, 101
Integración, 121, 127
Secuencial, 127
Integrador de sistemas, 121
Interface, 69
Interfaces, 132, 145
Interfaz, 118, 120
Interfaz de usuario
Diseñador, 113
Prototipo, 112, 114
Islas de conocimiento, 131
Iteración
Planificación, 111, 126
Iteraciones, 110
Iterator, 69

JAD, 46
Java, 5
JRP, 48

L-CASE, 161
La crisis del software, 2
Layered initialization, 69
Layers, 67
LGPL, 170
Librerías del sistema, 100
Licencia, 169
Linux, 155
Little languaje, 70

Módulo, 167
Mantenibilidad, 3
Mantenimiento, 153
Mantenimiento del sistema de información, 151
Manual de mantenimiento, 95
Manual de operación, 94
Manual de pruebas, 101
Manual de usuario, 94
Marker interface, 69
Material de capacitación, 101
Mediator, 70
Metodología, 2, 174
    Estructurada, 3
    Orientada a objetos, 4
Microkernel, 68
Minimalista, 175
Miniprototipo, 126
Modelado del dominio, 111
Modelado del negocio, 111
Modelo, 14
Modelo de casos de uso, 112
Modelo de despliegue, 118
    Arquitectura, 118
Modelo de diseño, 117
    Arquitectura, 118
Modelo de implementación, 120
Modelo de implementacion
    Vista de la arquitectura, 120
Modelo-Vista-Controlador, 68
MTBF, 3

Nomenclatura, 130
Null object, 70

Object pool, 69
Observer, 70
OMG, 15
OMT, 15
OOSE, 15
Operaciones abstractas, 21
Optimizaciones, 130

Público, 173
Pair programming, 131
Paquete
    Análisis, 117
Patrón, 65
Patrones, 65
    Patrones arquitectónicos, 66, 67
    Patrones de análisis, 66
    Patrones de codificación, 66
    Patrones de diseño, 66
    Patrones de organización, 66
    Sistemas adaptables, 68
    Sistemas distribuidos, 67
    Sistemas genéricos, 67
    Sistemas interactivos, 68
Patrones de diseño, 68
    De comportamiento, 70
    De concurrencia, 71
    De creación, 69
    De partición, 69
    Estructurales, 69
    Fundamentales, 68
Personal, 131
Pizarra, 67
Plan de instalación, 95
Plan de integración de construcciones, 121
Plan de proyecto, 94
Plan de prueba, 122
Plan de pruebas, 83
Plan de publicación, 125
Planificación, 152
Planificar prueba, 123
Prácticas, 133

Prólogo, 80
Presentación-Abstracción-Control, 68
Privado, 173
Procedimiento de prueba, 122
Procedimientos, 96
Procesos, 132
Producer-Consumer, 71
Protegido, 173
Prototipos, 50
  Componentes reutilizables, 51
    Nivel de aplicación, 52
    Nivel de componente, 52
  Desarrollo rápido de prototipos, 50
  Lenguajes de alto nivel, 50
  Programación de bases de datos, 51
Prototype, 69
Proxy, 69
Prueba, 122
Prueba de integración, 123
Prueba de sistema, 123
Prueba de unidad, 122, 129
Pruebas, 94
  Modelo, 122
  Pruebas de desempeño, 84
  Pruebas de tensión, 84
  Pruebas estructurales, 84
  Pruebas funcionales, 84
Pruebas de regresión, 110
Publicado, 173

RCS, 167
Read/Write lock, 71
Reflexión, 68
Rehacer, 130
Repositorio, 167
Requisito, 41
Requisitos
  Requisitos funcionales, 112
  Requisitos no funcionales, 112
Requisitos cambiantes, 110
Requisitos de datos, 94
Requisitos funcionales, 42, 94, 108
Requisitos no funcionales, 42
Reuniones, 131
Revisión, 167
Riesgos, 109
  Gestión, 110
  Tratamiento, 110
RUP, 104
  Construcción, 105
  Elaboración, 105
  Inicio, 104
  Transición, 105

Scheduler, 71
SCO-Unix, 155
Secuenciación, 111
Seguimiento y control, 146
  Gestión de incidencias, 147
Simula67, 5
Single threaded execution, 71
Singleton, 69
Sistema, 1
Smaltalk, 5
Snapshot, 70
Software libre, 169
SRS, 58
State, 70
Strategy, 71
Subsistema
  Diseño, 120
  Implementación, 122
Subsistema de implementación, 120

Tabla de tareas, 132
Tarjetas CRC, 126
Template method, 71
Test de aceptación, 129
Test de unidad, 129
Trabajadores, 113, 116, 118, 121, 122
  Analista de sistemas, 113
  Arquitecto, 113, 116, 118
  Diseñador de interfaz de usuario, 113
  Especificador de casos de uso, 113
  Ingeniero de casos de uso, 116, 118
  Ingeniero de componentes, 116
Tuberías y filtros, 67
Tutoriales, 96

Two-phase termination, 71

U-CASE, 161
UML, 15, 104
Unix, 155
Usabilidad, 3

VCL, 173
versión delta, 106
Virtual proxy, 70
Visibilidad, 175
Visitor, 71
Vistas, 108