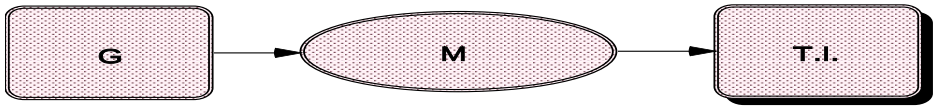
El desarrollo del software.

No disponemos de herramientas, ni siquiera de metodologías, que nos permitan transformar el software ordinario en otro que sea fiable y fácilmente mantenible. Los sistemas software medianamente grandes suelen estar "plagados" de errores, y realizar cambios en ellos es, cuando menos, una tarea arriesgada.

Frente a este duro panorama, nos encontramos con la necesidad de acometer el desarrollo de programas cada vez mayores. Para poder realizar estos desarrollos con la mejor calidad posible se hace necesaria la utilización de ciertas estrategias que, si bien no garantizan un buen resultado, si suelen mejorar bastante las características del producto desarrollado.



1. Introducción.

Como puede leerse en [Grady, 1990], hoy por hoy no disponemos de herramientas, ni siquiera de metodologías, que nos permitan transformar el software ordinario en otro que sea fiable y fácilmente mantenible. En el campo del hardware, por el contrario, esta anhelada situación está mucho más cerca de la realidad. Así, disponemos de chips que son a la vez extremadamente complejos y muy fiables. Sin embargo, los sistemas software medianamente grandes suelen estar "plagados" de errores, y realizar cambios en ellos es, cuando menos, una tarea arriesgada.

Esta diferencia puede ser debida al hecho de que el desarrollo de hardware siempre ha estado constreñido por limitaciones físicas (por ejemplo, densidad de integración). Así, la evolución se ha hecho "paso a paso", añadiendo complejidad poco a poco en cada uno de estos pasos, a medida que se lograba introducir más componentes en una superficie dada. Pero el software no tiene este tipo de limitaciones, con lo que desde el principio tenemos una gran cantidad de complejidad, que hemos de manejar de alguna forma.

Por eso, el gran desafío con que se encuentra la gestión de proyectos software consiste precisamente en limitar los productos que se desarrollan en esos proyectos a unos niveles de complejidad aceptables y manejables. Dicho de otra forma, se pretende reducir los grados de libertad en la producción de software para, al operar dentro de unos ciertos márgenes, mantener la complejidad resultante lo más baja posible.

Esto ha llevado a la concepción y uso de varios modelos del ciclo de vida. Con ellos se intenta descomponer los problemas de la gestión del proyecto de forma lógica, a la vez que generar productos tras cada etapa del modelo. Estos productos pueden ser usados para comprobar si estamos moviéndonos en la dirección deseada, o si por el contrario nos apartamos de los objetivos de complejidad previstos. Al fin y al cabo, utilizamos la acreditada técnica del "divide y vencerás".

Para enmarcar el estudio de los problemas relacionados con el desarrollo de software, señalemos que estamos tratando con uno de los llamados sistemas antropotécnicos, dentro del modelo de tres niveles de complejidad de Sáez Vacas (véase el capítulo sobre Marcos Conceptuales). El lector estará de acuerdo con esta afirmación si piensa que el proceso de desarrollo de programas un poco grandes implica la gestión y coordinación de los esfuerzos de numerosos grupos de personas, ayudadas de herramientas tecnológicas cada vez más avanzadas.

2. El ciclo de vida.

En principio, el ciclo de vida de un proyecto software incluye todas las acciones que se realizan sobre él desde que se especifican las características que debe tener, hasta que se mantiene en operación. A veces (aunque no será éste nuestro caso) se incluyen en el ciclo de vida las modificaciones que pueden realizarse al sistema para adaptarse a nuevas especificaciones.

Podría pensarse que el ciclo de vida de un programa no tiene por qué seguir un desarrollo "lineal", entendiendo como tal una sucesión de etapas. En principio, las distintas actividades que se realizan son bastante independientes, y pueden llevarse (hasta cierto punto) en paralelo. Por ejemplo, para empezar a codificar hay que tener mínimamente claras las especificaciones que hay que cumplir. Pero (aunque no es una buena decisión, como veremos más adelante), podría pensarse en comenzar la producción de código mientras se completan las especificaciones, para poder irlo probando, por ejemplo. Más adelante se harían las modificaciones necesarias.

Pero si el desarrollo de productos software ya es algo complejo en sí mismo (véase el capítulo sobre Medidas o Métricas de la Complejidad del Software), aún lo complicaremos más si intentamos "hacerlo todo a la vez", sin seguir una cuidadosa y detallada planificación. Y esto es precisamente lo que pretenden los modelos del ciclo de vida del software: simplificar en lo posible la gestión del proceso de desarrollo. La meta está en añadir la mínima complejidad que sea posible a la que de por sí ya implica el software.

Desde el punto de vista del esquema HxIxO->IO, podríamos decir que los modelos del ciclo de vida son un instrumento conceptual (I) que permite a la persona encargada (H) de gestionar un desarrollo de software (el O será por tanto el propio proceso de desarrollo) tratar con un problema más sencillo (el IO resultante).

Para ello, estos modelos dividen el proceso de desarrollo en unas cuantas etapas bien diferenciadas, y definen los posibles caminos por los que se deben relacionar. Además intentan que estos caminos lleven a un "progreso lineal", en el sentido de que antes de comenzar una etapa se haya concluido exitosamente (con las especificaciones cumplidas) la anterior. Sin embargo, esto no es siempre posible, y hay que recurrir a iteraciones (por ejemplo, entre el diseño y la codificación), que nos lleven mediante aproximaciones sucesivas a cumplir con los objetivos de la mejor forma posible.

Desde el punto de vista jerárquico (véase el capítulo sobre las jerarquías) esta división en etapas puede verse como una jerarquía multicapa de toma de decisiones. Así, cada una de las etapas (capa de decisiones) termina cuando, tras haber hecho todas las elecciones necesarias, se han cumplido los objetivos marcados, sentando las bases para la siguiente etapa. Al dividirse el problema en estas capas, en cada momento del desarrollo nos enfrentamos con una complejidad menor (únicamente la debida a cada capa, ya que las anteriores habrán sido satisfactoriamente resueltas).

3. El modelo de desarrollo en cascada.

Uno de estos modelos del ciclo de vida, quizás el más ampliamente utilizado, es el del desarrollo en cascada. En él, cada etapa deja el camino preparado para la siguiente, de forma que esta última no

debe comenzar hasta que no ha acabado aquélla. De esta forma, se reduce mucho la complejidad de la gestión, ya que basta con no dar por terminada una etapa hasta que haya cumplido totalmente con sus objetivos. De esta forma, la siguiente puede apoyarse con total confianza en ella. A la hora, por ejemplo, de fijar plazos, se podrían establecer planes de una forma totalmente secuencial, quedando perfectamente delimitadas las responsabilidades de los equipos que desarrollen cada etapa.

En la realidad la aplicación de este modelo no suele ser tan radical. Aunque se intenta conseguir la mayor secuencialidad posible, es difícil evitar las "vueltas atrás". Si después de la terminación de alguna etapa los resultados no son los esperados, en la práctica es muy posible que el problema esté en la mala realización de una etapa anterior. Y esto es así porque no sabemos cómo decidir con total certidumbre que una etapa ha sido perfectamente desarrollada hasta que se observan las consecuencias, quizás varias etapas y bastante tiempo después de que fue "cerrada". En estos casos, habrá que volver a ella, refinando el producto de una forma iterativa hasta que se considere que tiene la calidad deseada.

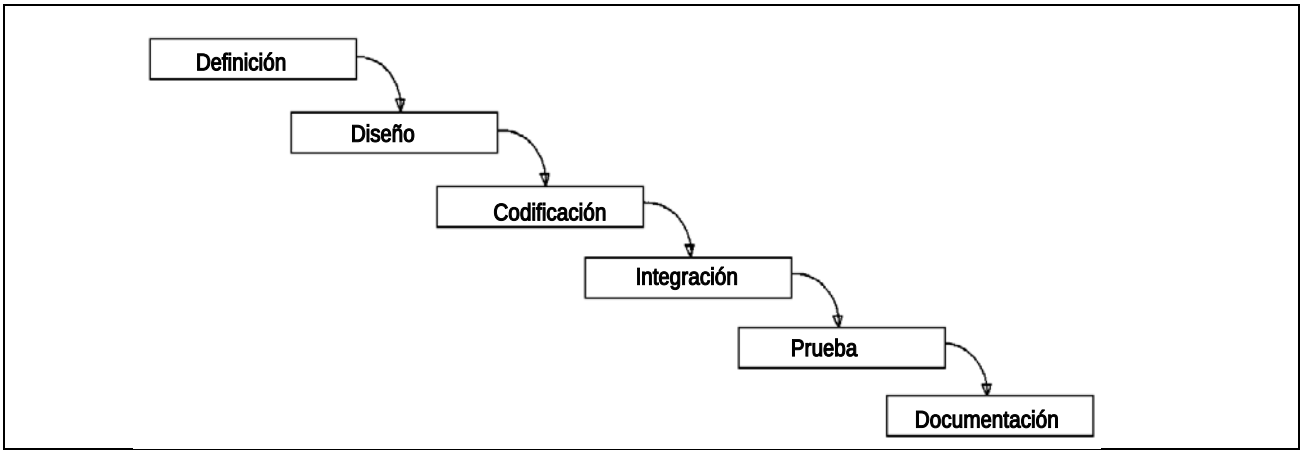


Fig. 1. Modelo en cascada del desarrollo de software.

En el modelo "puro", las fases en que se suele dividir el ciclo de vida en este modelo son [Grady, 1990]:

a. Definición (análisis de los requerimientos software).
b. Diseño (podría dividirse en preliminar y detallado).
c. Codificación.
d. Integración.
e. Prueba.
f. Documentación.

Estas fases de desarrollarían una tras otra, excepto quizás las dos últimas. La prueba de módulos podría realizarse después de la codificación y la del sistema completo tras la integración. La documentación, por su parte, puede irse creando a lo largo de todo el proceso.

Sin embargo, los caminos reales que se siguen en el desarrollo de software suelen parecerse mucho más a los que se pueden ver en la figura 2 (basada en [Fox, 1982]. En ella, las flechas que apuntan en sentido descendente representarían el modelo puro, mientras que las ascendentes corresponden a los demás caminos que se suelen seguir en la realidad.

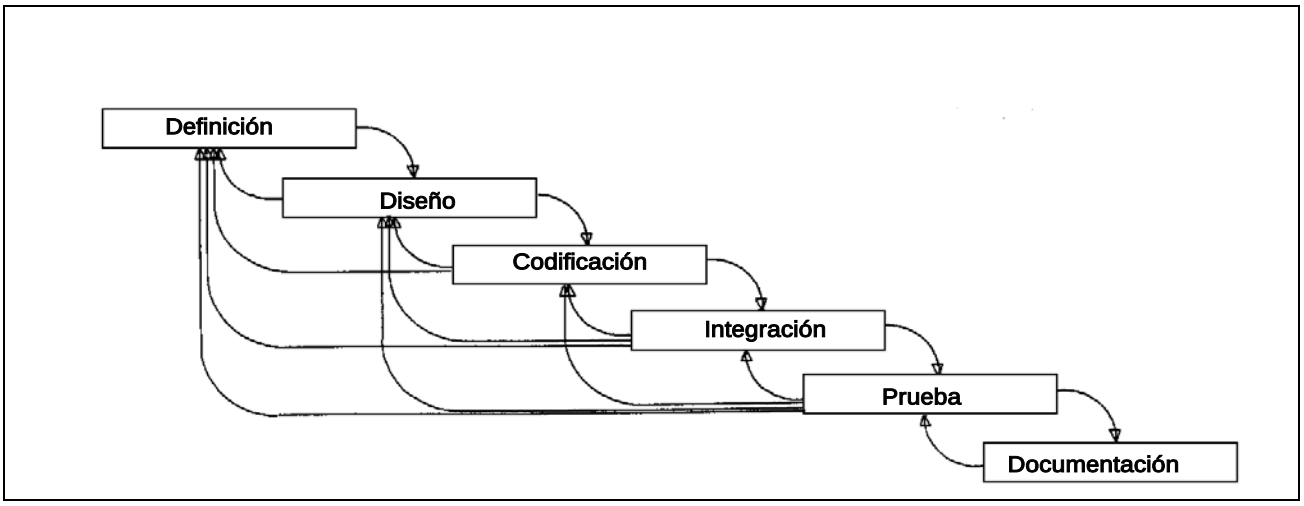


Fig. 2. Caminos reales en el desarrollo de software [Fox, 1982].

Pasemos a describir ahora cada una de las etapas del modelo en cascada, que ya hemos nombrado.

3.1. Definición.

La definición de requisitos o especificación de características que ha de cumplir el software que vamos a desarrollar es la primera etapa del modelo en cascada. Y probablemente sea la más importante. Al fin y al cabo, lo que sea o no sea el producto final depende de decisiones tomadas en esta etapa. Se trata fundamentalmente de estudiar las necesidades y preferencias del usuario. Es también muy importante dejar clara constancia de las decisiones tomadas en esta etapa, para ser tenidos en cuenta posteriormente. Por ello, la documentación producida en esta fase debe ser concreta y estar siempre disponible durante el resto del proceso.

Pero, normalmente, nuestro software no será mas que una parte de un sistema mayor. Y puede ser que "herede" problemas de indefinición de este sistema. Por ejemplo, como el hardware es muy difícil de modificar, a menudo los programas sufren cambios de última hora para "tapar" sus defectos. Esto hace que, en un proyecto real, sean bastante normales los caminos de vuelta desde etapas posteriores a ésta de definición (ver figura 2).

Por si hubiera pocos problemas debemos tener en cuenta que el entorno en el que nos movemos suele ser muy variable, de forma que las características que piden los usuarios suelen cambiar muy rápidamente con el tiempo. Además, las innovaciones tecnológicas hacen posible modificar ciertas partes del sistema para ganar, por ejemplo en eficiencia. Por todo esto, la facilidad de modificación del software resultante es siempre un requisito fundamental, que debe compaginarse con los demás.

La descomposición en niveles de abstracción es una metodología que nos puede ayudar a abordar los problemas que aparecen en esta etapa. Las características que debe tener esta descomposición son las habituales en el análisis de sistemas complejos (ver capítulo sobre las jerarquías): pocos elementos en cada nivel de abstracción, contextos limitados y bien definidos, etc.

De un tiempo a esta parte se están comenzando a utilizar técnicas formales de definición. Esto permite generar especificaciones coherentes y sin ambigüedades. Además se está investigando activamente en la generación automática de software a partir de definiciones escritas en lenguajes formales (aunque aún no se han logrado grandes resultados prácticos en este campo).

3.2. Diseño.

Una vez planteada la especificación del programa, hay que analizar desde un punto de vista técnico las posibles soluciones. Entre ellas, se elegirá la que se considere más adecuada. A partir de ese momento, se decidirá la estructura general del programa (subdivisión en partes y relaciones entre ellas). Para cada una de las partes se seguirá recursivamente un proceso similar, hasta que tengamos totalmente definido el programa y estemos listos para pasar a la fase de codificación.

En el análisis de cada una de las partes nos encontraremos normalmente con que hay varias soluciones posibles (por ejemplo varios algoritmos para realizar la misma tarea). La elección de una de ellas suele realizarse de una forma más o menos intuitiva: no hay metodologías efectivas que nos ayuden en esta decisión.

Como puede deducirse de lo dicho hasta aquí, la descomposición en niveles de abstracción también será útil en esta fase. Cada etapa del proceso recursivo descrito puede constituir un nivel de abstracción. Si además, utilizamos las posibilidades de ocultación de información que nos permite esta metodología, podremos descomponer nuestro programa en pequeños módulos fáciles de modificar.

En el nivel más bajo del diseño hay que decidir la estructura de control y el flujo de datos del módulo. El uso de la programación estructurada facilita enormemente la comprensión de los algoritmos, al limitar los flujos de control posibles.

El producto final de la etapa de diseño puede ser un organigrama, unas líneas de pseudocódigo, etc. Algunos lenguajes de programación (como Ada) permiten hasta cierto punto realizar el diseño en el propio lenguaje, y compilarlo posteriormente. Así pueden detectarse incoherencias y ambigüedades de una forma automática. Además se favorece en gran medida la integración con la etapa de codificación.

Algunos trabajos recientes ([Rombach, 1990], [Henry y Selig, 1990]) proponen utilizar métricas en la fase de diseño para predecir la calidad del producto software antes de llegar a la codificación. Así se ahorrarían esfuerzos, al encontrar pronto zonas de gran complejidad y de poca calidad. De esta forma estas zonas podrían rediseñarse, consiguiéndose así que den menos problemas en posteriores etapas del desarrollo.

3.3. Codificación.

En un proyecto grande ésta es la etapa más sencilla (en contra de lo que suele suponer cualquier persona que comienza a aprender un lenguaje de programación). Si el diseño es adecuado y suficientemente detallado la codificación de cada módulo es algo casi automático.

Una de las principales decisiones a tomar en esta fase es la del lenguaje a emplear, aunque a veces en el diseño ya está de alguna forma implícito. Desde hace tiempo la tendencia es a utilizar lenguajes de más alto nivel, sobre todo a medida de que se dispone de compiladores más eficientes. Esto ayuda a los programadores a pensar más cerca de su propio nivel que del de la máquina, y la productividad suele mejorarse. Como contrapartida este tipo de lenguajes son más difíciles de aprender. Y además hay que tener en cuenta que los programadores suelen ser conservadores y reacios a aprender nuevos lenguajes: prefieren usar los que ya conocen. La existencia, en una organización, de una gran cantidad de programas desarrollados en un determinado lenguaje, hace además muy dura la decisión de cambiar a uno nuevo.

Evaluar la calidad de la codificación es una tarea nada fácil. Para un mismo diseño son posibles muchas implementaciones diferentes. Y no hay criterios claros que no permitan decidir cuál es la mejor. En este punto, las métricas del software pueden ser utilizadas en nuestra ayuda (ver capítulo sobre las métricas).

Cuando intervienen varias personas, pueden aparecer problemas a la hora de realizar modificaciones, debido a que cada uno tiene su propio estilo. Por eso se hace necesario definir estándares de estilo para facilitar la legibilidad y claridad del software producido.

3.4. Integración.

Una vez que tenemos los módulos codificados, hay que ensamblarlos. Desgraciadamente el proceso no consiste simplemente en unir piezas. Suelen aparecer problemas con las interfaces entre los módulos, con la comunicación de datos compartidos, con el encadenamiento de flujos de ejecución, etc.

Si el programa es además bastante grande, la gestión de versiones se convierte en un problema no despreciable. Afortunadamente, ésta es una de las etapas donde disponemos de más herramientas CASE, que nos pueden ayudar.

3.5. Prueba.

En esta fase hay que comprobar que las especificaciones se cumplen perfectamente y en todos los casos. En la realidad es prácticamente imposible probar un programa totalmente: por ello siempre suele quedar algún error escondido. Este problema se agrava cuando sobre él se realizan repetidos

cambios y correcciones. Si no los gestionamos de un forma adecuada podemos acabar con un conjunto de parches que más que soluciones aportan problemas.

Actualmente se están comenzando a utilizar técnicas de verificación y validación como alternativa a la simple prueba de programas. Según Wallace y Fujii [Wallace y Fujii, 1989], la verificación y validación es una disciplina de ingeniería de sistemas, que intenta evaluar el software desde un punto de vista sistémico. Utiliza una aproximación estructurada para analizar y probar el software en relación con todos los aspectos del sistema en el cual se incluye, y en especial con el hardware, los usuarios y las interfaces con otras piezas de software.

Idealmente, la verificación y validación se realiza paralelamente al desarrollo de software, durante todo su ciclo de vida (por lo que no entra en el modelo en cascada, estrictamente hablando), y pretende alcanzar los siguientes objetivos:

a. Descubrir pronto errores de alto riesgo, dando al equipo de diseño la oportunidad de elaborar una solución adecuada, evitando que se vea obligado a poner un "parche" si el error se detecta demasiado tarde.
b. Evaluar el ajuste de los productos desarrollados a las especificaciones del sistema.
c. Proporcionar al equipo de gestión información actualizada sobre la calidad y el progreso del esfuerzo de desarrollo.

Éste de la verificación y validación es un campo donde se están realizando activas investigaciones, mientras comienzan a obtenerse los primeros frutos.

3.6. Documentación.

La documentación es algo totalmente necesario para poder mantener un programa. Incluso la persona que lo ha codificado se perderá con gran facilidad en un programa a los pocos meses de haberlo terminado. No sólo hay que documentar el código (las conocidas líneas de comentario del programa), sino todas las etapas del ciclo de vida. Especialmente es importante que todas las decisiones que se han tomado queden claramente expuestas, así como las razones que han llevado a ellas.

Además, hay que generar la documentación de "caja negra", esto es, la que se refiere no a aspectos internos del programa, sino a su manejo y características "externas". Esto incluye normalmente un manual de usuario, para las personas que normalmente van a utilizarlo (en el caso de que sea un programa directamente utilizado por personas) y un manual de referencia técnica, donde se dan detalles de su instalación y explotación, de cara al personal técnico encargado de estas tareas.

En el modelo en cascada hemos colocado la etapa de documentación al final, porque es cuando se realizará la documentación definitiva, y especialmente los manuales "de caja negra" de los que hemos hablado. Pero es conveniente ir preparándola a lo largo de todo el desarrollo, según van realizándose las actividades a documentar.

Para gestionar esta etapa (llevar el control de las versiones de la documentación, incluso generarla automáticamente en algunos casos) también se dispone de herramientas informáticas de ayuda.

4. Los "productos intermedios".

Tras cada una de las etapas del ciclo de vida se genera, como resultado final, algún tipo de producto. Son lo que llamaremos "productos intermedios". Estos productos constituyen la base del trabajo de la siguiente etapa. Por ejemplo, a partir del pseudocódigo obtenido en la fase de diseño, los codificadores escribirán el programa. Y este programa (resultado de la etapa de codificación) será la base para la integración. Una lista más exhaustiva de los productos intermedios que se obtienen en cada etapa del modelo en cascada puede verse en la figura 3.

Pero estos productos pueden usarse para algo más que meramente como apoyo de la fase siguiente. Según Grady [Grady, 1990], una correcta utilización de los productos intermedios ayuda a producir software de calidad, ya que:

a. Cada producto intermedio suele seguir alguna forma de representación estándar que garantiza un cierto grado de terminología común.
b. Existen herramientas que pueden aplicarse a estos productos, para hacer comprobaciones sobre ellos, aportando así realimentación inmediata a los ingenieros de desarrollo (generalmente mediante la forma de avisos y mensajes de error).
c. La terminología común simplifica las inspecciones por parte de otros equipos de trabajo. Así se facilita la detección de errores que las herramientas automáticas no son capaces de detectar.
d. También pueden utilizarse herramientas que calculen ciertas métricas sobre diversos aspectos de la complejidad de los productos intermedios. Así se pueden detectar zonas con mayor probabilidad de que presenten errores, o que tengan un difícil mantenimiento.

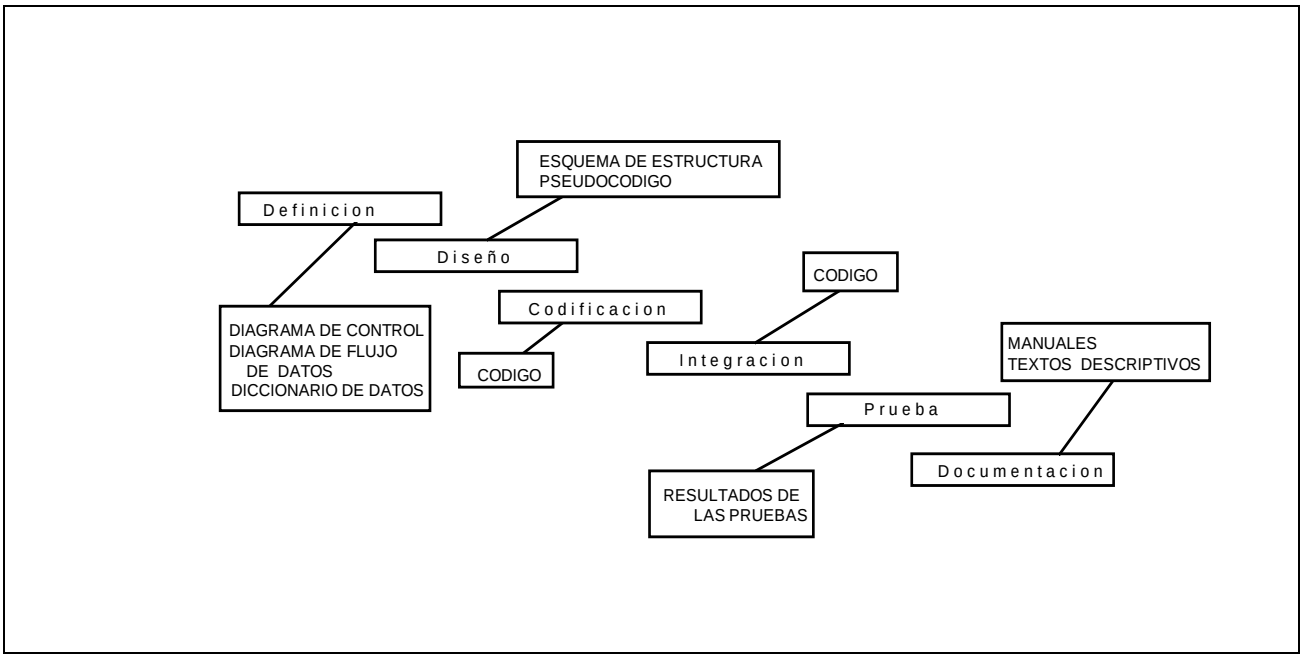


Fig.3. Productos que se generan en cada una de las etapas de producción de software (según el modelo de la cascada), basado en [Grady, 1990].

La idea central de este aprovechamiento de los productos intermedios del desarrollo software reside en la información que éstos aportan, como resumen final que son de su etapa correspondiente. Así, su análisis permite una realimentación rápida y concreta, y una forma de ir midiendo paso a paso la calidad del software que se está produciendo.

5. Resumen.

En este capítulo hemos tratado de exponer los problemas que aparecen en el desarrollo de sistemas software reales, y de los enfoques que se utilizan para abordarlos.

Comenzamos introduciendo el tipo de sistemas con el que tratamos (sistemas antropotécnicos), y las diferentes características que los hacen complejos y difíciles de abordar. Luego hemos hablado de una de las soluciones más utilizadas: la reducción de complejidad mediante la limitación de los grados de libertad del sistema. Y de cómo se concreta esta solución: el modelo en cascada del ciclo de vida de un programa.

Y por último, nos hemos detenido en las etapas que constituyen este modelo en cascada, poniendo de relieve los aspectos que nos han parecido más destacables en cada una de ellas. Es uno de los modelos posibles. Hay otros muchos, como el modelo en espiral, pero en cualquiera de ellos se hace patente que la complejidad surge de la necesidad de coordinar económicamente un elevado número de personas, técnicas y equipo. Es la complejidad de gestión del software.

Bibliografía.

Dividida en dos partes. En primer lugar, Notas Bibliográficas, donde se describen los trabajos consultados más relevantes sobre el tema. Después, Referencias Bibliográficas, donde pueden encontrarse todas las citas utilizadas en el capítulo.

Notas bibliográficas.

Un trabajo muy utilizado ha sido el de Grady [Grady, 1990], que entre otras aportaciones originales, incluye el análisis de los productos intermedios del ciclo de vida como ayuda para la producción de programas de calidad.

También es necesario citar el número de mayo de 1989 de la revista IEEE Software, dedicado a la verificación y validación, de donde están tomadas las ideas sobre este particular que pueden encontrarse en el capítulo.

El libro de Fox [Fox, 1982] puede ser de gran utilidad para ampliar conocimientos sobre las etapas del ciclo de vida de un sistema software, y su significado.

Referencias bibliográficas.

Fox, J.M. (1982): **"Software and its development"**, Ed. Prentice-Hall.

Grady, R.B. (1990): "Work-product analysis: the philosopher's stone of software?", **IEEE Software**, March, pag.26-34.

Henry,S. y Selig, C. (1990): "Predicting source code complexity at the design stage", **IEEE Software**, March, pag.36-45.

Rombach, H.D. (1990): "Design measurement: some lessons learned", **IEEE Software**, March , pag.17-25.

Wallace,D.R. y Fujii,R.U. (1989): "Software verification and validation: an overview", **IEEE Software**, May. pag.10-17.