Documentación y pruebas

Antes del paradigma de objetos

Pablo Suárez
Carlos Fontela
2003

ÍNDICE

Índice **2**

Documentación y pruebas en el desarrollo tradicional del software **3**
Documentación y desarrollo de software 3
Las pruebas en el desarrollo de software 4
Calidad, errores y pruebas 4
Categorías de pruebas 5
Las pruebas y el desarrollo de software 5
Tipos de pruebas 6
Revisiones de código 7
Pruebas unitarias 7
Pruebas de integración 10
Pruebas de sistema 13
Pruebas de aceptación 13
Herramientas a usar por los propios programadores 14
Reacción ante los resultados de las pruebas 14
Depuración 14
Reacción ante los errores en las pruebas de sistema y de aceptación 15

DOCUMENTACIÓN Y PRUEBAS EN EL DESARROLLO TRADICIONAL DEL SOFTWARE

DOCUMENTACIÓN Y DESARROLLO DE SOFTWARE

En general se habla mucho de la documentación, pero no se la hace, no se le asigna presupuesto, no se la mantiene y casi nunca está al día en los proyectos de desarrollo de software. Lo importante es la disponibilidad de la documentación que se necesita en el momento en que se la necesita.

Muchas veces se hace porque hay que hacerla y se escribe, con pocas ganas, largos textos, a la vez que se está convencido de estar haciendo un trabajo inútil. A veces se peca por exceso y otras por defecto. Ocurre mucho en la Web y con productos RAD. En ocasiones se olvida que el mantenimiento también debe llegar a la documentación.

La documentación se suele clasificar en función de las personas o grupos a los cuales está dirigida:

- Documentación para los desarrolladores
- Documentación para los usuarios
- Documentación para los administradores o soporte técnico

La documentación para desarrolladores es aquélla que se utiliza para el propio desarrollo del producto y, sobre todo, para su mantenimiento futuro. Se documenta para comunicar estructura y comportamiento del sistema o de sus partes, para visualizar y controlar la arquitectura del sistema, para comprender mejor el mismo y para controlar el riesgo, entre otras cosas. Obviamente, cuanto más complejo es el sistema, más importante es la documentación.

En este sentido, todas las fases de un desarrollo deben documentarse: requerimientos, análisis, diseño, programación, pruebas, etc.. Una herramienta muy útil en este sentido es una notación estándar de modelado, de modo que mediante ciertos diagramas se puedan comunicar ideas entre grupos de trabajo.

Hay decenas de notaciones, tanto estructuradas como orientadas a objetos. Un caso particular es el de UML, que analizamos en otra obra. De todas maneras, los diagramas son muy útiles, pero siempre y cuando se mantengan actualizados, por lo que más vale calidad que cantidad.

La documentación para desarrolladores a menudo es llamada **modelo**, pues es una simplificación de la realidad para comprender mejor el sistema como un todo.

Otro aspecto a tener en cuenta cuando se documenta o modela, es el del nivel de detalle. Así como cuando construimos planos de un edificio podemos hacer planos generales, de arquitectura, de instalaciones y demás, también al documentar el software debemos cuidar el nivel de detalle y hacer diagramas diferentes en función del usuario de la documentación, concentrándonos en un aspecto a la vez.

De toda la documentación para los desarrolladores, nos interesa especialmente en esta obra aquella que se utiliza para documentar la programación, y en particular hemos analizado la que se usa para documentar desarrollos orientados a objetos en el capítulo respectivo.

La documentación para usuarios es todo aquello que necesita el usuario para la instalación, aprendizaje y uso del producto. Puede consistir en guías de instalación[1], guías del

[1] Incluye los requisitos del sistema, el procedimiento de instalación en pasos bien definidos y las posibilidades de personalización.

usuario[2], manuales de referencia[3] y guías de mensajes[4].

En el caso de los usuarios que son programadores, verbigracia los clientes de nuestras clases, esta documentación se debe acompañar con ejemplos de uso recomendados o de muestra y una reseña de efectos no evidentes de las bibliotecas.

Más allá de todo esto, debemos tener en cuenta que la estadística demuestra que los usuarios no leen los manuales a menos que nos les quede otra opción. Las razones pueden ser varias, pero un análisis de la realidad muestra que se recurre a los manuales solamente cuando se produce un error o se desconoce cómo lograr algo muy puntual, y recién cuando la ayuda en línea no satisface las necesidades del usuario. Por lo tanto, si bien es cierto que debemos realizar manuales, la existencia de un buen manual nunca nos libera de hacer un producto amigable para el usuario, que incluso contenga ayuda en línea. Es incluso deseable proveer un software tutorial que guíe al usuario en el uso del sistema, con apoyo multimedial, y que puede llegar a ser un curso on-line.

Buena parte de la documentación para los usuarios puede empezar a generarse desde que comienza el estudio de requisitos del sistema. Esto está bastante en consonancia con las ideas de *extreme programming* y con metodologías basadas en casos de uso.

La documentación para administradores o soporte técnico, a veces llamada manual de operaciones, contiene toda la información sobre el sistema terminado que no hace al uso por un usuario final. Es necesario que tenga una descripción de los errores posibles del sistema, así como los procedimientos de recuperación. Como esto no es algo estático, pues la aparición de nuevos errores, problemas de compatibilidad y demás nunca se puede descartar, en general el manual de operaciones es un documento que va engrosándose con el tiempo.

LAS PRUEBAS EN EL DESARROLLO DE SOFTWARE

Calidad, errores y pruebas

La calidad no es algo que se pueda agregar al software después de desarrollado si no se hizo todo el desarrollo con la calidad en mente. Muchas veces parece que el software de calidad es aquél que brinda lo que se necesita con adecuada velocidad de procesamiento. En realidad, es mucho más que eso. Tiene que ver con la corrección, pero también con usabilidad, costo, consistencia, confiabilidad, compatibilidad, utilidad, eficiencia y apego a los estándares.

Todos estos aspectos de la calidad pueden ser objeto de tests o pruebas que determinen el grado de calidad. Incluso la documentación para el usuario debe ser probada.

Como en todo proyecto de cualquier índole, siempre se debe tratar que las fallas sean mínimas y poco costosas, durante todo el desarrollo. Y además, es sabido que cuanto más tarde se encuentra una falla, más caro resulta eliminarla. Es claro que si un error es descubierto en la mitad del desarrollo de un sistema, el costo de su corrección será mucho menor al que se debería enfrentar en caso de descubrirlo con el sistema instalado y en funcionamiento.

Desde el punto de vista de la programación, nos interesa la ausencia de errores (corrección), la confiabilidad y la eficiencia. Dejando de lado las dos últimas, nos concentraremos en este capítulo en las pruebas que determinen que un programa está libre de errores.

Un **error** es un comportamiento distinto del que espera un usuario razonable. Puede haber

[2] En forma de curso o tutorial, mediante pasos que ayuden en el aprendizaje. Pueden hacerse dos guías: una para usuarios básicos y otra para avanzados.

[3] Enumeración y explicación de toda la funcionalidad del sistema, con entradas fáciles, para el usuario avanzado que está buscando una característica en particular.

[4] Sobre todo, mensajes de error e informativos.

errores aunque se hayan seguido todos los pasos indicados en el análisis y en el diseño, y hasta en los requisitos aprobados por el usuario. Por lo tanto, no necesariamente un apego a los requisitos y un perfecto seguimiento de las etapas nos lleva a un producto sin errores, porque aún en la mente de un usuario pudo haber estado mal concebida la idea desde el comienzo. De allí la importancia del desarrollo incremental, que permite ir viendo versiones incompletas del sistema.

Por lo tanto, una primera fuente de errores ocurre antes de los requerimientos o en el propio proceso de análisis. Pero también hay errores que se introducen durante el proceso de desarrollo posterior. Así, puede haber errores de diseño y errores de implementación. Finalmente, puede haber incluso errores en la propia etapa de pruebas y depuración.

Categorías de pruebas

Según la naturaleza de lo que se esté controlando, las pruebas se pueden dividir en dos categorías:

- Pruebas centradas en la verificación
- Pruebas centradas en la validación

Las primeras sirven para determinar la consistencia entre los requerimientos y el programa terminado. Soporta metodologías formales de testeo, de mucho componente matemático. De todas maneras, hay que ser cuidadoso, porque no suele ser fácil encontrar qué es lo que hay que demostrar. La verificación consiste en determinar si estamos construyendo el sistema correctamente, a partir de los requisitos.

En general a los informáticos no les gustan las pruebas formales, en parte porque no las entienden y en parte porque requieren un proceso matemático relativamente complejo.

La validación consiste en saber si estamos construyendo el sistema correcto. Las tareas de validación son más informales. Las pruebas suelen mostrar la presencia de errores, pero nunca demuestran su ausencia.

Las pruebas y el desarrollo de software

La etapa de pruebas es una de las fases del ciclo de vida de los proyectos. Se la podría ubicar después del análisis, el diseño y la programación, pero dependiendo del proyecto en cuestión y del modelo de proceso elegido, su realización podría ser en forma paralela a las fases citadas o inclusive repetirse varias veces durante la duración del proyecto.

La importancia de esta fase será mayor o menor según las características del sistema desarrollado, llegando a ser vital en sistemas de tiempo real u otros en los que los errores sean irrecuperables.

Las pruebas no tienen el objeto de prevenir errores sino de detectarlos[5]. Se efectúan sobre el trabajo realizado y se deben encarar con la intención de descubrir la mayor cantidad de errores posible.

Para realizar las pruebas se requiere gente que disfrute encontrando errores, por eso no es bueno que sea el mismo equipo de desarrollo el que lleve a cabo este trabajo. Además, es un principio fundamental de las auditorías. Por otro lado, es bastante común que a quien le guste programar no le guste probar, y viceversa.

A veces se dan por terminadas las pruebas antes de tiempo. En las pruebas de caja blanca (ver más adelante) no es mala idea probar un 85% de las ramas y dar por terminado luego de esto. Otra posibilidad es la siembra de errores y seguir las pruebas hasta que se encuentren un 85% de

[5] La prevención de los mismos es igualmente importante y se tiene que intentar con todos los medios de que se disponga.

los errores sembrados, lo que presumiblemente implica que se encontró un 85% de los no sembrados[6]. Otros métodos se basan en la estadística y las comparaciones, ya sea por similitud con otro sistema en cantidad de errores o por el tiempo de prueba usado en otro sistema.

En un proyecto ideal, podríamos generar casos de prueba para cubrir todas las posibles entradas y todas las posibles situaciones por las que podría atravesar el sistema. Examinaríamos así exhaustivamente el sistema para asegurar que su comportamiento sea perfecto. Pero hay un problema con esto: el número de casos de prueba para un sistema complejo es tan grande que no alcanzaría una vida para terminar con las pruebas. Como consecuencia, nadie realiza una prueba exhaustiva de nada salvo en sistemas triviales.

En un sistema real, los casos de prueba se deben hacer sobre las partes del sistema en los cuales una buena prueba brinde un mayor retorno de la inversión o en las cuales un error represente un riesgo mayor.

Las pruebas cuestan mucho dinero. Pero para ello existe una máxima: “pague por la prueba ahora o pague el doble por el mantenimiento después”.

Todo esto lleva a que se deban planificar bien las pruebas, con suficiente anticipación, y determinar desde el comienzo los resultados que se deben obtener.

La idea de *extreme programming* es más radical: propone primero escribir los programas de prueba y después la aplicación, obligando a correr las pruebas siempre antes de una integración. Se basa en la idea bastante acertada de que los programas de prueba son la mejor descripción de los requerimientos. Esto lo analizamos en un ítem especial.

Tipos de pruebas

Analizaremos 5 tipos de pruebas:

- Revisiones de código
- Pruebas unitarias
- Pruebas de integración
- Pruebas de sistema
- Pruebas de aceptación

No son tipos de pruebas intercambiables, ya que testean cosas distintas. En el ítem siguiente analizamos cada tipo.

Otra posible clasificación de las pruebas es:

- De caja blanca o de código
- De caja negra o de especificación

En las primeras se evalúa el contenido de los módulos, mientras en las segundas se trata al módulo como una caja cerrada y se lo prueba con valores de entrada, evaluando los valores de salida. Vistas de este modo, las pruebas de caja negra sirven para verificar especificaciones.

Las pruebas unitarias suelen ser de caja blanca o de caja negra, mientras que las de integración, sistema y aceptación son de caja negra. Las tareas de depuración luego de encontrar errores son más bien técnicas de caja blanca, así como las revisiones de código. En todos los casos, uno de los mayores desafíos es encontrar los datos de prueba: hay que encontrar un subconjunto de todas las entradas que tengan alta probabilidad de detectar el mayor número de errores.

[6] Aunque a veces quedan los errores más complejos porque se sembraron errores muy triviales.

Revisiones de código

Las revisiones de código son las únicas que se podrían omitir de todos los tipos de pruebas, pero tal vez sea buena idea por lo menos hacer alguna de ellas:

- Pruebas de escritorio
- Recorridos de código
- Inspecciones de código

La prueba de escritorio rinde muy poco, tal vez menos de lo que cuesta, pero es una costumbre difícil de desterrar. Es bueno concentrarse en buscar anomalías típicas, como variables u objetos no inicializados o que no se usan, ciclos infinitos y demás.

Los recorridos rinden mucho más. Son exposiciones del código escrito frente a pares. El programador, exponiendo su código, encuentra muchos errores. Además da ideas avanzadas a programadores nuevos que se lleva a recorrer.

Las llamadas inspecciones de código consisten en reuniones en conjunto entre los responsables de la programación y los responsables de la revisión. Tienen como objetivo revisar el código escrito por los programadores para chequear que cumpla con las normas que se hayan fijado y para verificar la eficiencia del mismo. Se realizan siguiendo el código de un pequeño porcentaje de módulos seleccionados al azar o según su grado de complejidad. Las inspecciones se pueden usar en sistemas grandes, pero con cuidado para no dar idea de estar evaluando al programador. Suelen servir porque los revisores están más acostumbrados a ver determinados tipos de errores comunes a todos los programadores. Además, después de una inspección a un programador, de la que surge un tipo de error, pueden volver a inspeccionar a otro para ver si no cayó en el mismo error.

El concepto de *extreme programming* propone programar de a dos, de modo que uno escribe y el otro observa el trabajo. Si el que está programando no puede avanzar en algún momento, sigue el que miraba. Y si ambos se traban pueden pedir ayuda a otro par. Esta no sólo es una forma más social de programación, sino que aprovecha las mismas ventajas de los recorridos e inspecciones de código, y puede prescindir de ellos.

Pruebas unitarias

Las pruebas unitarias se realizan para controlar el funcionamiento de pequeñas porciones de código como ser subprogramas (en la programación estructurada) o métodos (en POO). Generalmente son realizadas por los mismos programadores puesto que al conocer con mayor detalle el código, se les simplifica la tarea de elaborar conjuntos de datos de prueba para testearlo.

Si bien una *prueba exhaustiva* sería impensada teniendo en cuenta recursos, plazos, etc., es posible y necesario elegir cuidadosamente los casos de prueba para recorrer tantos caminos lógicos como sea posible. Inclusive procediendo de esta manera, deberemos estar preparados para manejar un gran volumen de datos de prueba.

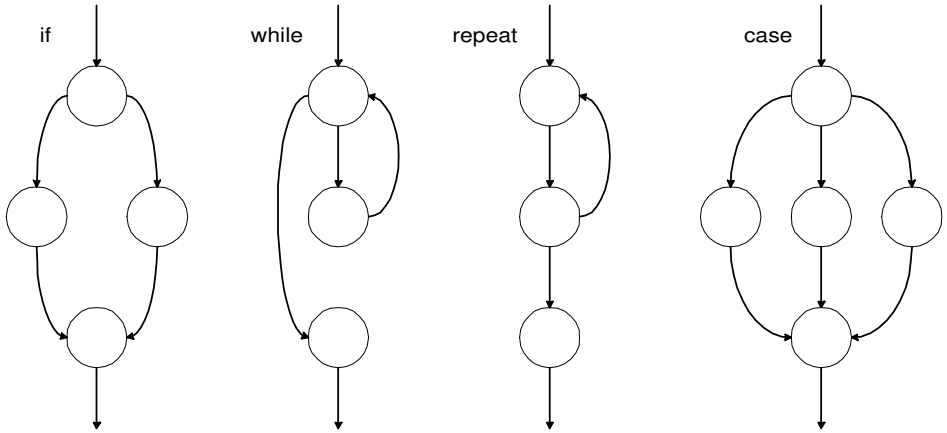
Los métodos de cobertura de caja blanca tratan de recorrer todos los caminos posibles por lo menos una vez, lo que no garantiza que no haya errores pero pretende encontrar la mayor parte.

El tipo de prueba a la cual se someterá a cada uno de los módulos dependerá de su complejidad. Recordemos que nuestro objetivo aquí es encontrar la mayor cantidad de errores posible. Si se pretende realizar una prueba estructurada, se puede confeccionar un grafo de flujo con la lógica del código a probar. De esta manera se podrán determinar todos los caminos por los que el hilo de ejecución pueda llegar a pasar, y por consecuente elaborar los juegos de valores de pruebas para aplicar al módulo, con mayor facilidad y seguridad.

Un grafo de flujo se compone de:

- Nodos (círculos), que representan una o más acciones del módulo.
- Aristas (flechas), que representan el flujo de control entre los distintos nodos.

Los nodos predicados son aquellos que contienen una condición, por lo que de ellos emergen dos o más aristas.

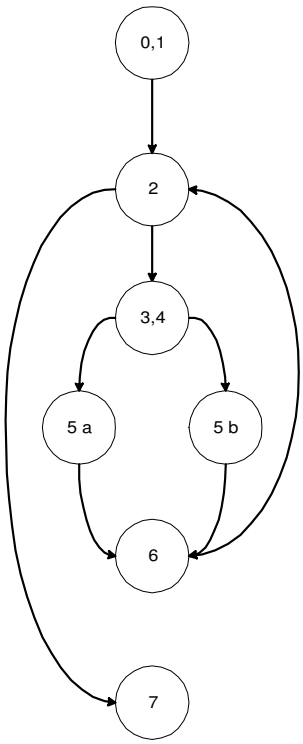


El paso de un diseño detallado o un pseudocódigo que representa una porción de programa a un grafo de flujo, requiere de las siguientes etapas:

- Señalar cada condición, tanto en sentencias *if* y *case* como en bucles *while* y *repeat*.
- Agrupar todas las sentencias en secuencias siguiendo los esquemas de representación de construcciones.
- Numerar cada uno de los nodos resultantes de manera que consten de un identificador único. Las ramas de cada bifurcación pueden identificarse por el mismo número seguido de distintas letras.
- Dibujar en forma ordenada los nodos y sus aristas.

En el siguiente ejemplo, se muestra la manera de traducir un pequeño tramo de programa escrito en pseudocódigo a forma de grafo de flujo:

```
0:  Imprimir ("Ingrese 2 números")
1:  Leer (Num1 y Num2)
2:  Mientras (Num1 < Num2) hacer
3:     Num1 = Num1 + 1
4:     Si (Num1 es par) entonces
5a:        Imprimir (Num1" es par")
5b:  Si no   Imprimir (Num1" es impar")
6:     Fin Si
7:  Fin Mientras
```



Luego de elaborado el grafo de flujo, se deberán identificar todos los *caminos independientes*. Un camino independiente es un camino que introduce un nuevo grupo de acciones o nueva condición (es decir, incluye una arista no recorrida previamente). En el ejemplo de la figura, los caminos independientes son:

0,1 – 2 – 7
0,1 – 2 – 3,4 – 5a - 6 – 2 – 7
0,1 – 2 – 3,4 – 5b - 6 – 2 – 7

Los caminos anteriores constituyen un *conjunto básico* que permite recorrer todas las acciones y el valor verdadero y falso de cada una de las condiciones (cobertura de condición). El conjunto básico de caminos para un diseño no es único. Para el caso anterior también podríamos obtener un conjunto básico compuesto por los siguientes 2 caminos:

0,1 – 2 – 7
0,1 – 2 – 3,4 – 5a - 6 – 2 – 3,4 – 5b – 6 – 2 – 7

Una vez determinado el conjunto básico, hay que preparar los casos que fuercen la ejecución de los caminos independientes.

Para recorrer el camino 0,1 – 2 – 7, la condición del nodo 2 (*Num1 < Num2*) debe ser falsa, por lo que le podemos dar valores iguales a ambos números o hacer que *Num1* sea mayor a *Num2*.

Para recorrer el camino 0,1 – 2 – 3,4 – 5a - 6 – 2 – 7, la condición del nodo 2 debe ser verdadera la primera vez y falsa la segunda. Además, *Num1* debe ser un número impar. Los juegos de valores posibles para este caso serían todos los *Num1* impares que cumplan con la condición *Num2 – Num1 = 1*.

Para recorrer el camino 0,1 – 2 – 3,4 – 5b - 6 – 2 – 7, la condición del nodo 2 debe ser verdadera la primera vez y falsa la segunda, pero en este caso *Num1* debe ser un número par. Como consecuencia, los juegos de valores posibles serán todos los *Num1* pares que cumplan con la condición *Num2 – Num1 = 1*.

Con respecto a nuestro segundo conjunto básico, necesitaríamos un *Num1* mayor o igual a *Num2* para recorrer el primer camino y un *Num1* impar que cumpla con la condición *Num2 – Num1 = 2* para recorrer el segundo camino.

Habiendo identificado un conjunto básico y una vez obtenidos y documentados los juegos de valores de prueba junto con sus resultados esperados, se procede a probar el código. Todas las salidas deberán ser documentadas y en caso de verificarse la existencia de errores, se deberá dejar constancia de ello, proceder a la corrección de los mismos y volver a realizar las pruebas.

Una posibilidad para determinar los conjuntos de datos de prueba en problemas medianamente complejos es usar particiones de equivalencias, que garantizan las recorridas de todas las ramas del código. Pero tampoco es fácil encontrar clases de equivalencia si los límites no son lineales, y puede ser un problema sin solución analítica.

Una vez halladas las particiones, deberán elegirse los valores. Por ejemplo, para valores que tienen que ser entre 1 y 10, se puede probar con uno entre 1 y 10, otro menor que 1 y otro mayor que 10, y con los propios límites de las clases de equivalencia. Los valores fuera del rango conviene que estén cerca de los límites. Todo esto tiene que ver con errores al programar condiciones de menor cuando debería decir menor o igual, etc., que se potencian si son varias. Cuando los conjuntos sean discretos (por ejemplo, casado, soltero, viudo, divorciado) probar con todos los valores y alguno inválido. Cuando hablamos de datos de prueba inválidos, esto sólo se puede aplicar en lenguajes que no permitan restringir los conjuntos en tiempo de compilación, pues si se puede, como en Pascal, habría que hacerlo con subrangos y demás. De todas maneras, aun cuando se usen subrangos, conviene forzar datos de entrada y tratar de generar errores de ejecución por combinaciones extremas. También es bueno buscar casos que cubran más de una

clase (por ejemplo, un ciclo que se recorre y luego se sale, todo con un solo dato inicial).

Hay especialistas en construir casos de prueba, que son buenos también detectando inconsistencias y faltas de completitud.

Pruebas de integración

En el caso de las pruebas de integración y de sistema, dado que ya se han realizado las pruebas unitarias, se tomará a cada uno de los módulos unitarios como una caja negra.

Las pruebas de integración tienen como base las pruebas unitarias y consisten en una progresión ordenada de testeos para los cuales los distintos módulos van siendo ensamblados y probados hasta haber integrado el sistema completo. Si bien se realizan sobre módulos ya probados en forma individual, no es necesario que se terminen todas las pruebas unitarias para comenzar con las de integración. Dependiendo de la forma en que se organicen, se pueden realizar en paralelo a las unitarias.

El orden de integración de los módulos influye en :

- La forma de preparar los casos de prueba
- Las herramientas a utilizar (módulos ficticios, muñones, impulsores o “stubs”)
- El orden para codificar y probar los módulos
- El costo de preparar los casos
- El costo de la depuración

Tanto es así que se le debe prestar especial atención al proceso de elección del orden de integración que se emplee.

Existen principalmente dos tipos de integración: La integración incremental y la no incremental. La integración incremental consiste en combinar el conjunto de módulos ya probados (al principio será un conjunto vacío) con los siguientes módulos a probar. Luego se va incrementando progresivamente el número de módulos unidos hasta que se forma el sistema completo. En la integración no incremental o Big Bang se combinan todos los módulos de una vez.

Para ambos tipos de integración se deberán preparar los datos de prueba junto con los resultados esperados. Esta tarea debe ser realizada por personas ajenas a la programación de los módulos. No es necesario que la lleven a cabo una vez codificados los módulos puesto que basta con conocer qué módulos compondrán el sistema y cuál será la interfaz entre ellos.

Si en algún momento de la prueba se detectara uno o más errores, se dejará constancia del hecho y se reenviarán los módulos afectados al responsable de la programación para que identifique la causa del problema y lo solucione. Luego se volverán a efectuar las pruebas programadas y así sucesivamente hasta que el sistema entero esté integrado y sin errores.

Por el hecho de poder ser llevada a cabo por distintos caminos, la integración incremental brinda una mayor flexibilidad en el uso de recursos. Se puede integrar la estructura de módulos desde un extremo u otro y continuar hacia el extremo opuesto según distintas prioridades. La forma de llevar a cabo esta tarea dependerá de la naturaleza del sistema en cuestión, pero sea cual fuere el camino elegido, será de suma importancia una correcta planificación.

En la **integración incremental ascendente** se comienza integrando primero los módulos de más bajo nivel. El proceso deberá seguir los siguientes pasos:

- Elegir los módulos de bajo nivel que se van a probar.
- Escribir un módulo impulsor para la entrada de datos de prueba a los módulos y para la visualización de los resultados.

- Probar la integración de los módulos.
- Eliminar los módulos impulsores y juntar los módulos ya probados con los módulos de niveles superiores, para continuar con las pruebas.

Estas tareas se pueden realizar en paralelo si es que se dispone de tres equipos de trabajo, o en serie de lo contrario. Para la prueba de cada uno de los módulos mencionados, es necesaria la preparación de un módulo impulsor. El objeto de los módulos impulsores es transmitir o impulsar los casos de prueba a los módulos testeados y recibir los resultados que estos produzcan en los casos en que sea necesario. Es decir que deben simular las operaciones de llamada de los módulos jerárquicos superiores correspondientes. Estos módulos tienen que estar bien diseñados, para que no arrojen ni más ni menos errores que los que realmente pueden producirse. Al fin y al cabo, deben simular todas las situaciones que se van a producir en el sistema real.

La **integración incremental descendente** parte del módulo de control principal (de mayor nivel) para luego ir incorporando los módulos subordinados progresivamente. No hay un procedimiento considerado óptimo para seleccionar el siguiente módulo a incorporar. La única regla es que el módulo incorporado tenga todos los módulos que lo invocan previamente probados.

En general no hay una secuencia óptima de integración. Debe estudiarse el problema concreto y de acuerdo a este, buscar el orden de integración más adecuado para la organización de las pruebas. No obstante, pueden considerarse las siguientes pautas:

- Si hay secciones críticas como ser un módulo complicado, se debe proyectar la secuencia de integración para incorporarlas lo antes posible.
- El orden de integración debe incorporar cuanto antes los módulos de entrada y salida.

Existen principalmente dos métodos para la incorporación de módulos:

- Primero en profundidad: Primero se mueve verticalmente en la estructura de módulos.
- Primero en anchura: Primero se mueve horizontalmente en la estructura de módulos.

Etapas de la integración descendente:

- El módulo de mayor nivel hace de impulsor y se escriben módulos ficticios simulando a los subordinados, que serán llamados por el módulo de control superior.
- Probar cada vez que se incorpora un módulo nuevo al conjunto ya engarzado.
- Al terminar cada prueba, sustituir un módulo ficticio subordinado por el real que reemplazaba, según el orden elegido.
- Escribir los módulos ficticios subordinados que se necesiten para la prueba del nuevo módulo incorporado.

Los módulos ficticios subordinados se crean para permitir la prueba de los demás módulos. Pueden llevar a cabo una variedad de funciones, como por ejemplo:

- Mostrar un mensaje que demuestre que ese módulo fue alcanzado ("Módulo XX alcanzado").
- Establecer una conversación con una terminal. De esta forma se puede permitir que la misma persona que realiza la prueba actúe de módulo subordinado.
- Devolver un valor constante, tabulado o elegido al azar.
- Ser una versión simplificada del módulo representado.
- Mostrar los datos recibidos.
- Ser un loop sin nada que hacer más que dejar pasar el tiempo.

Ventajas de la integración descendente:

- Las fallas que pudieran existir en los módulos superiores se detectan en una etapa temprana.
- Permite ver la estructura del sistema desde un principio, facilitando la elaboración de demostraciones de su funcionamiento.
- Concuerda con la necesidad de definir primero las interfaces de los distintos subsistemas para después seguir con las funciones específicas de cada uno por separado.

Desventajas de la integración descendente:

- Requiere mucho trabajo de desarrollo adicional ya que se deben escribir un gran número de módulos ficticios subordinados que no siempre son fáciles de realizar. Suelen ser más complicados de lo que aparentan.
- Antes de incorporar los módulos de entrada y salida resulta difícil introducir los casos de prueba y obtener los resultados.
- Los juegos de datos de prueba pueden resultar difíciles o imposibles de generar puesto que generalmente son los módulos de nivel inferior los que proporcionan los detalles.
- Induce a diferir la terminación de la prueba de ciertos módulos.

Ventajas de la integración incremental ascendente:

- Las entradas para las pruebas son más fáciles de crear ya que los módulos inferiores suelen tener funciones más específicas.
- Es más fácil la observación de los resultados de las pruebas puesto que es en los módulos inferiores donde se elaboran.
- Resuelve primero los errores de los módulos inferiores que son los que acostumbran tener el procesamiento más complejo, para luego nutrir de datos al resto del sistema.

Desventajas de la integración incremental ascendente:

- Se requieren módulos impulsores, que deben escribirse especialmente y que no son necesariamente sencillos de codificar.
- El sistema como entidad no existe hasta que se agrega el último módulo.

El método de **integración incremental sándwich** combina facetas de los métodos ascendente y descendente. Consiste en integrar una parte del sistema en forma ascendente y la restante en forma descendente, provocando la unión de ambas partes en algún punto intermedio. La principal ventaja es que nos da mayor libertad para elegir el orden de integración de los módulos según las características específicas del sistema en cuestión. De esta manera, podremos incluir y probar antes los módulos que consideremos críticos:

- Módulos dirigidos a múltiples propósitos.
- Módulos con mayor control (en general, los módulos de mayor nivel controlan a muchos otros módulos).
- Módulos con alto grado de complejidad.
- Módulos con requisitos de rendimiento muy definidos.

La **integración no incremental** puede ser beneficiosa para la prueba de sistemas de pequeñísima envergadura cuya cantidad de módulos sea muy limitada y la interfaz entre los mismos clara y sencilla. Consiste en integrar todos los módulos del sistema a la vez e ingresar los valores de prueba para testear todas las interfaces.

La única ventaja es que no se necesita ningún tipo de trabajo adicional: ni planificar el orden de integración, ni crear módulos impulsores, ni crear módulos ficticios subordinados. Por otro lado, la lista de desventajas incluye:

- No se tiene noción de la comunicación de los módulos hasta el final.
- En ningún momento se dispone de un producto —siquiera parcial— para mostrar o presentar.
- El hecho de realizar todas las pruebas de una vez hace que las sesiones de prueba sean largas y tediosas.
- La cantidad de errores que arroje puede ser atemorizante.
- La tarea de encontrar la causa de los errores resulta mucho más compleja que con los métodos incrementales.
- Se corre el riesgo de que a poco tiempo de que se cumpla el plazo de entrega, haya que volver sobre el diseño y la codificación del sistema.

Pruebas de sistema

Las pruebas de sistema se realizan una vez integrados todos los componentes. Su objetivo es ver la respuesta del sistema en su conjunto, frente a distintas situaciones. Se simulan varias alternativas que podrían darse con el sistema implantado y en base a ellas se prueba la eficacia y eficiencia de la respuesta que se obtiene. Se pueden distinguir varios tipos de prueba distintos, por ejemplo:

- Pruebas negativas: se trata de que el sistema falle para ver sus debilidades.
- Pruebas de recuperación: se simulan fallas de software y/o hardware para verificar la eficacia del proceso de recuperación.
- Pruebas de rendimiento: tiene como objeto evaluar el rendimiento del sistema integrado en condiciones de uso habitual.
- Pruebas de resistencia o de estrés: comprueban el comportamiento del sistema ante situaciones donde se demanden cantidades extremas de recursos (número de transacciones simultáneas anormal, excesivo uso de las memorias, etc.).
- Pruebas de seguridad: se utilizan para testear el esquema de seguridad intentando vulnerar los métodos utilizados para el control de accesos no autorizados al sistema.
- Pruebas de instalación: verifican que el sistema puede ser instalado satisfactoriamente en el equipo del cliente, incluyendo todas las plataformas y configuraciones de hardware necesarias.
- Pruebas de compatibilidad: se prueba al sistema en las diferentes configuraciones de hardware o de red y de plataformas de software que debe soportar.

Pruebas de aceptación

Las pruebas de aceptación, al igual que las de sistema, se realizan sobre el producto terminado e integrado; pero a diferencia de aquellas, están concebidas para que sea un usuario final quien detecte los posibles errores.

Se clasifican en dos tipos: pruebas Alfa y pruebas Beta.

Las **pruebas Alfa** se realizan por un cliente en un entorno controlado por el equipo de desarrollo. Para que tengan validez, se debe primero crear un ambiente con las mismas condiciones que se encontrarán en las instalaciones del cliente. Una vez logrado esto, se procede a realizar las pruebas y a documentar los resultados.

Cuando el software sea la adaptación de una versión previa, deberán probarse también los procesos de transformación de datos y actualización de archivos de todo tipo.

Por otro lado, las **pruebas Beta** se realizan en las instalaciones propias de los clientes. Para que tengan lugar, en primer término se deben distribuir copias del sistema para que cada cliente lo instale en sus oficinas, dependencias y/o sucursales, según sea el caso. Si se tratase de un número reducido de clientes, el tema de la distribución de las copias no representa grandes dificultades, pero en el caso de productos de venta masiva, la elección de los *beta testers* debe realizarse con sumo cuidado. En el caso de las pruebas Beta, cada usuario realizará sus propias pruebas y documentará los errores que encuentre, así como las sugerencias que crea conveniente realizar, para que el equipo de desarrollo tenga en cuenta al momento de analizar las posibles modificaciones.

Cuando el sistema tenga un cliente individual, las pruebas de aceptación se hacen de común acuerdo con éste, y los usuarios se determinan en forma programada, así como también se definen los aspectos a probar y la forma de informar resultados. Cuando, en cambio, se está desarrollando un producto masivo, los usuarios para pruebas se determinan de formas menos estrictas, y hay que ser muy cuidadoso en la evaluación del *feedback* que proveen. Por lo tanto, en este segundo caso hay que dedicar un esfuerzo considerable a la planificación de las pruebas de aceptación.

Herramientas a usar por los propios programadores

A menudo los programadores pueden insertar líneas de código que faciliten las pruebas y la depuración.

Un ejemplo de esto son los invariantes. Por ejemplo, si sabemos que a lo largo de un programa la variable *A* debe tener un valor mayor que la *B*, podemos introducir el invariante *A>B*, de modo tal que la implementación nos avise cuando algún error provoque la alteración de ese invariante.

Algunos lenguajes de programación permiten la definición de invariantes, pero en la mayoría deben implementarse especialmente.

Otra herramienta práctica son las afirmaciones. Por ejemplo, si en determinado lugar del código debe cumplirse que *A>0*, puede declararse esto como un aserto o afirmación, de modo que la computadora lo verifique y avise al programador en caso de que no se cumpla.

Sirven adicionalmente como precondiciones y poscondiciones de módulos.

REACCIÓN ANTE LOS RESULTADOS DE LAS PRUEBAS

Las pruebas nos llevan a descubrir errores, que en la mayoría de los casos son de tipo funcional, es decir, del tipo: “el sistema debería hacer tal cosa y hace tal otra”.

En este apartado analizaremos nada más que los pasos a seguir cuando el error sólo es atribuible a la codificación.

Depuración

La depuración es la corrección de errores que sólo afectan a la programación, porque no provienen de errores previos en el análisis o en el diseño. A veces la depuración se hace luego de la entrega del sistema al cliente y es parte del mantenimiento.

En realidad, en las revisiones de código y las pruebas unitarias, encontrar un error es considerablemente más sencillo, ya que se hace con el código a mano. Aun cuando se hubiera optado por una prueba unitaria de caja negra, es sencillo recorrer el módulo que revela un

comportamiento erróneo por dentro para determinar el lugar exacto del error. Existen incluso herramientas de depuración (en inglés, *debugging*) de los propios ambientes de desarrollo que facilitan esta tarea, que incluso proveen recorrido paso a paso y examen de valores de datos. Y el lenguaje C traía una macro *assert* portable, que sencillamente abortaba un programa si una condición no se cumplía.

De todas maneras, es importante analizar correctamente si el error está donde parece estar o proviene de una falla oculta más atrás en el código. Para encontrar estos casos más complejos son útiles las herramientas de recorrida hacia atrás, que permiten partir del lugar donde se genera el error y recorrer paso a paso el programa en sentido inverso.

Las pruebas de integración, de sistema y de aceptación también pueden llevar a que sea necesaria una depuración, aunque aquí es más difícil encontrar el lugar exacto del error. Por eso a menudo se utilizan *bitácoras* (*logs*, en inglés), que nos permiten evaluar las condiciones que se fueron dando antes de un error mediante el análisis de un historial de uso del sistema que queda registrado en medios de almacenamiento permanente.

La depuración se hace en cuatro pasos:

- Reproducir el error.
- Diagnosticar la causa.
- Corregirla.
- Verificar la corrección.

Si el error no se repite al intentar reproducirlo es muy difícil hacer el diagnóstico. Como en casi todas las ciencias, se buscan causas y efectos, condiciones necesarias y suficientes para que se produzca el error. Luego hay que buscar el sector del código donde se produce el error, lo que nos lleva a las consideraciones hechas recientemente. La corrección del error entraña mucho riesgo, porque a menudo se introducen nuevos errores (hay quienes hablan de tasas de 0,2 a 0,5 nuevos errores por corrección). Y nunca hay que olvidarse de realizar una nueva verificación después de la corrección.

Reacción ante los errores en las pruebas de sistema y de aceptación

Hemos dicho ya que los errores que aparezcan en estos tipos de pruebas van a llevar a la larga a una depuración, en la medida en que sean errores de codificación.

Para llegar a ello, no obstante, se requiere determinar el módulo donde se produjo el error. Esta tarea, en apariencia dificultosa, puede facilitarse considerablemente si trabajamos con un entorno de desarrollo que nos permita recorrer el código en modo de depuración sin necesidad de entrar en todos los módulos.

Como ya dijimos, podemos encontrarnos, y es más habitual en esta etapa, con errores de análisis o diseño, pero no estudiaremos esto aquí.