Este documento ha sido generado para facilitar la impresión de los contenidos.
Los enlaces a otras páginas no serán funcionales.

__María__ ha encargado a __Carmen__ la elaboración del plan de pruebas del software para la aplicación que van a desarrollar para la empresa __SinUnEuro__, como parte del análisis y diseño de la misma. __Julia__ no entiende demasiado bien que ese plan de pruebas se realice antes de que la aplicación esté programada, ya que piensa que sería mejor esperar a tenerla lista para probar si funciona o no. Además, una vez que esté lista, los programadores habrán comprobado que funciona cuando la consideran terminada, y no entiende demasiado bien qué tipo de errores se podrían buscar. __María__ le explica que justamente lo que se intenta es que las pruebas verifiquen que la programación ha sido fiel al diseño, y que por tanto la aplicación hace lo que tiene que hacer, y no que las pruebas sirvan como justificación de que lo que se ha hecho funciona. Es más, lo ideal es que las personas que pasen esas __pruebas__ no sean las mismas que desarrollaron la aplicación. Está claro que ese plan de pruebas pretende garantizar que la aplicación va a suponer una mejora, y sobre todo, que no va a tener consecuencias negativas para el negocio por culpa de algún malfuncionamiento o circunstancia no prevista. Cualquier error, defecto o fallo en el sotware puede tener consecuencias desastrosas en el negocio, y eso hay que evitarlo. __Víctor__ que anda un poco despistado, se queda pensativo al oir esas palabras: Para él, fallo, error y defecto del software eran lo mismo. __Carmen__ le explica que hay matices, y le aclara las diferencias.

Pruebas del software

La Sociedad moderna actual depende en gran medida de sistemas informáticos. Estos equipos ejecutan una serie de aplicaciones que han sido diseñadas e implementadas por programadores. Estas personas deben tener muy presente las repercusiones de su trabajo. **¿Somos conscientes como programadores de las consecuencias de un error?**

Entre 1985 y 1987, al menos cinco personas, murieron al

recibir una terapia médica de radiación con la máquina Therac25. Tras diversos estudios se descubrió que la causa del fallo era un **error de software** en la interfaz gráfica de control de la máquina que permitía suministrar dosis de radiaciones mortales. Los programadores que diseñaron y programaron la interfaz gráfica probablemente no eran conscientes de la repercusión de un error en su trabajo. Afortunadamente, no todos los fallos de software terminan tan trágicamente, pero pueden provocar grandes pérdidas económicas.

El 19 de noviembre de **2003**, se produjo un apagón que dejó sin energía eléctrica a todo el noreste de Estados Unidos y Canadá. El motivo fue una serie de fallos en cadena que fueron extendiendo el apagón. Pero el problema podría haberse quedado en un corte de suministro en unos pocos pueblos dependientes de la central eléctrica que falló en primer lugar, si el sistema informático de seguridad no hubiese tenido un error de programación que impidió que saltasen las alarmas a tiempo. El resultado fue la parada de más de 100 centrales eléctricas con grandes pérdidas económicas.

¿Recuerdas el problema informático del año 2000? Cuando los programadores decidieron almacenar el año en una variable de sólo 2 cifras no eran conscientes de la repercusión de esa decisión: ¡en el año 2000 volveríamos al año 1900! Posiblemente en 1970 quedaba muy lejos el año 2000 y no pensaron en el problema, pero durante las décadas posteriores se continuaron implementando programas donde el año se almacenaba con 2 cifras (se suponía que todos los años comenzaban con 19xx), ¿por mantener compatibilidad con los sistemas anteriores?, ¿por falta de previsión? La consecuencia fue que cuando se acercaba el año 2000 hubo que solucionar el problema actualizando los programas con un gasto económico muy grande que podría haberse evitado fácilmente y con un coste mucho menor si se hubiesen almacenado las fechas en una variable con 4 cifras.

¿Hemos aprendido del problema del año 2000? Parece que no, la vieja frase "El hombre es el único animal que tropieza dos veces en la misma piedra", sigue siendo actual porque tenemos otra bomba de relojería esperándonos para el **año 2038**. **El problema del año 2038** afecta a los programas que representen el tiempo contando el número de segundos transcurridos desde el 1 de enero de 1970 a las 00:00:00. Esta representación es estándar en los sistemas tipo Unix y también en los programas escritos con el lenguaje de programación C. En la mayoría de estos sistemas el tipo de dato usado para guardar el contador de segundos es un entero de 32 bits con signo, es decir, que el mayor segundo que puede representar será el correspondiente a las 03:14:07 del 19 de enero de 2038, cuando el contador llegue a 2147483647.

Un segundo después, el contador se desbordará, y causará el fallo de los programas que interpretarán el tiempo como que están en 1970 en vez de 2038. La

solución a este problema consiste en usar números enteros de 64 bits para almacenar la fecha quedando "retrasado" el problema al domingo, 4 de diciembre del año 292 277 026 596 a las 15:30:08 ¿Estaremos vivos para comprobarlo?. La migración a estos sistemas está todavía en proceso y se espera que se complete antes del 2038. Sin embargo, cientos de millones de sistemas de 32 bits son utilizados todavía en el 2006, y adaptarlos a 64 bits será un proceso largo y costoso.

¿Qué podemos hacer para evitar estas situaciones en la medida de lo posible? Como estás viendo a lo largo de todo este módulo la **Ingeniería del Software** intenta responder a esta pregunta desarrollando unas metodologías que sistematicen y proporcionen calidad al desarrollo de aplicaciones informáticas. Sin embargo, a pesar de utilizar estas metodologías es posible que se cometan fallos, por lo que es imprescindible que dentro de la propia metodología se propongan estrategias y medidas para la detección de esos errores. En esta unidad vamos a estudiar las **pruebas del software** como mecanismos de detección y corrección de errores.

> ***Para saber más***
>
> ***Si quieres saber más sobre errores de programación famosos puedes visitar la enciclopedia de conocimiento libre Wikipedia en esta dirección:***
>
> ***Wikipedia: Error de software***
>
> ***http://es.wikipedia.org/wiki/Error_de_software*** **[versión en cache]**

Pruebas del software

Aclaremos conceptos

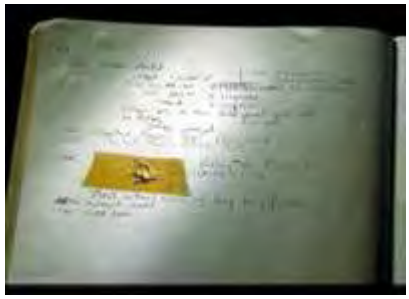
En el apartado anterior hemos visto problemas de distinta naturaleza. En Informática esos problemas han sido clasificados para poder sistematizarlos y proponer soluciones. Así surgen una serie de conceptos que vamos a describir en este apartado.

Algunos autores diferencian entre fallo, error y defecto, aunque las diferencias son sólo pequeños matices:

- **Defecto**: ocurre cuando en un programa hay un proceso, una definición de datos o un procesamiento incorrecto. Por ejemplo, si en un programa almacenamos datos con decimales en una variable entera podremos provocar que el programa deje de funcionar. El defecto está en el producto diseñado.
- **Error**: el error va a ser la manifestación del defecto cuando se ejecuta el sistema, pero un error puede producirse porque había defectos en el producto o puede ocurrir por otras situaciones, como cuando un uso inadecuado del programa provoca que el sistema no funcione bien. Por **ejemplo**, un error

puede ocurrir cuando un operador introduce datos incorrectos. El problema no era tanto del programador (no había defectos) sino del operador que no usó el programa correctamente según las instrucciones dadas. El error se produce mientras ejecutamos y usamos el software.

- **Fallo**: cuando en un programa hay defectos o errores y el sistema no funciona como debiera, decimos que se ha producido un fallo, por ejemplo cuando el ordenador queda bloqueado con una pantalla azul. El fallo indica que hay una desviación entre los requisitos o necesidades del usuario y el producto elaborado.



En inglés a los defectos y errores de programación se les llama "**computer bug**". Literalmente *bug* significa chinche y el motivo de usar dicha palabra no está claro aunque hay distintas interpretaciones. Algunos piensan que se debe a la programadora Grace Murray Hopper, una pionera en la historia de la computación desarrolladora del lenguaje de programación Cobol y que fue almirante del ejército de los EEUU. Trabajando con un Mark II en la universidad de Harvard en 1947, los ingenieros encontraron una chinche enganchado a uno de los circuitos del ordenador que impedía el funcionamiento del mismo. Dicho insecto pasó a la historia de la informática por ser pegado al libro de registro de actividad del ordenador (tal y como aparece en la imagen), con el comentario «*First actual case of bug being found*» *("Primer caso real de chinche encontrado").* Sin embargo, el término **bug** fue utilizado antes para referirse a fallos mecánicos, así por ejemplo, Thomas Edison lo empleaba en sus trabajos de 1870 para describir defectos.

Para saber más

Si quieres conocer la historia de esta programadora puedes visitar esta página de la Wikipedia:

Wikipedia: Grace Murray Hopper

http://es.wikipedia.org/wiki/Grace_Murray_Hopper

Autoevaluación

¿Cuál de los siguientes casos sería un "bug"?

a) Una impresora que deja de imprimir porque se queda sin papel.
b) Un programa que debe actualizarse para usar euros en lugar de pesetas.
c) Un programa que calcula los número primos y responde que 21 es primo.
d) Ninguna respuesta es correcta .

Comprobar

Caso.

***Carmen** y **Víctor** tienen que decidir qué estrategia de prueba van a usar, o si van a usar varias de ellas, según para qué parte de la aplicación. **Carmen** empieza a hablar de que deben usar procedimientos de prueba, casos de prueba y componentes de prueba, lo que hace que **Víctor** la detenga y le pida una explicación. Tras un tiempo aclarándole las cosas, se han puesto a trabajar.*

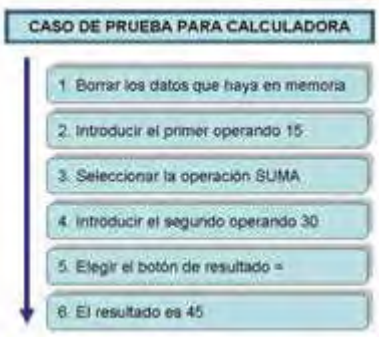
***Víctor** y **Carmen** han elaborado un conjunto de casos de prueba consistente, que **verificará** si el funcionamiento del software es correcto, es decir, que el programa que se haga para **SinUnEuro**, se habrá construido correctamente, pero ¿será el software que realmente querían y necesitaban en **SinUnEuro**? Para ello, contarán con la ayuda de **José**, que hará las pruebas de validación, y de **Julia**, que en representación de la empresa será la encargada de darle el visto bueno, con una serie de pruebas de **aceptación** deese software.*

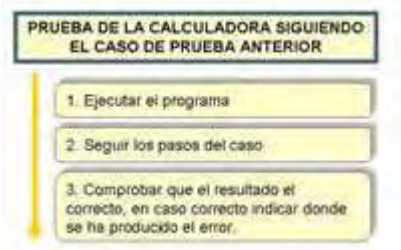
En la introducción hablamos de las **pruebas** como respuesta a los fallos de software, pero ¿qué es exactamente una prueba?

La prueba será el proceso de detección de errores y estará compuesto por estos elementos:

- **Procedimiento de Prueba**: descripción del conjunto de actividades que se realizan en un sistema de software según un plan preestablecido para evaluarlo y detectar posibles errores.
- **Caso de prueba**: Hemos dicho que las pruebas se realizan según un plan. En éste se recogen las condiciones de ejecución, los datos de entrada y los resultados esperados con las que se realizará la prueba. Un caso de prueba será, por tanto, el conjunto de estos datos, condiciones y resultados usados en las pruebas de software.

Por ejemplo, para un programa que implementase una calculadora, un caso de prueba y el procedimiento de prueba correspondiente pueden verse en las siguientes figuras:





- A veces, para llevar a cabo la prueba es necesario

utilizar componentes software auxiliares que nos ayuden. Éstos se denominan **componentes de pruebas** y se estudiarán más adelante en esta unidad.

Cuando hablamos de **pruebas de software** solemos pensar en encontrar errores en los programas, pero si nos fijamos en los fallos que comentamos en la introducción, algunos no eran errores de programación sino de diseño. Si un programa almacena la fecha en una variable de 32 bits no se puede decir que funcione mal o que tenga errores, el hecho de que el programa no pueda funcionar más allá del año 2038 es un problema de diseño y no de error de software. Por tanto, vemos que si queremos evitar fallos debemos revisar el software en todo su proceso y no sólo en la acción de la programación. Esto quiere decir que las pruebas de software deben realizarse a lo largo de todo el **ciclo de vida del software.**

Por otro lado, también en este problema del año 2038 podemos ver que la **solución** era muy sencilla de resolver al principio, bastaba con usar una variable de 64 bits, pero modificar los programas una vez que están terminados y usándose es mucho más costoso. Por tanto, es fundamental **detectar los errores desde el principio**. Según ciertos estudios, un error que no se detecta al inicio del ciclo de vida, necesitará 50 veces más de esfuerzo para solucionarlo si es detectado después de tener el programa implementado. Utilizando el símil del insecto (*bug*), hay que eliminarlos cuando son pequeños y pocos, antes de que crezcan y se reproduzcan convirtiéndose en una plaga.

Autoevaluación

¿Cuándo es mejor realizar las pruebas?

a) Al principio del ciclo de vida para localizar los fallos pronto y ahorrar esfuerzos
b) Al final, cuando esté el programa terminado, pues entonces podrán realizarse los casos de prueba y ver si funciona bien el programa
c) A lo largo de todo el ciclo de vida
d) Ninguna respuesta es correcta

Comprobar

Pruebas del software

Verificación y validación

Hemos visto el concepto de prueba, pero no sólo basta con comprobar que el programa obtiene resultados correctos, también hay que asegurarse de que cumple con las **necesidades del usuario.** Por ejemplo, un Rolls-Royce es un coche que sin duda pasaría las pruebas más exigentes sobre los últimos detalles de su mecánica o su carrocería. Sin embargo, si el cliente desea un todo-terreno, difícilmente va a comprárselo. De forma equivalente, en el

mundo informático, si escribimos un programa para ordenar datos por orden ascendente, pero el cliente los necesita en orden decreciente, no seremos capaces de detectar el fallo buscando errores de programación, debemos revisar el análisis de requisitos que se hizo.

Por tanto la **comprobación del software** debe incluir dos aspectos:

- ¿Estamos elaborando correctamente el software? A lo largo del ciclo de vida debemos comprobar que cada producto intermedio obtenido satisface las condiciones previstas. El proceso que comprueba este aspecto se denomina **Verificación de Software.**
- ¿Estamos elaborando el software correcto? Es decir, el producto final cumple con los requisitos iniciales y las necesidades del cliente. El proceso que comprueba este aspecto se denomina **Validación de Software.**

¿Y qué nos aportan las **pruebas** a estos procesos?

Son el mecanismo fundamental, ya que las **pruebas** realizadas a lo largo de todo el ciclo de vida permiten **validar** y **verificar** el software.

En el resto de este apartado veremos cómo se organizan las pruebas y en qué consisten. En apartados posteriores aprenderemos a diseñar y llevar a cabo pruebas.

Autoevaluación

Si un cliente solicita un software para llevar la contabilidad de su empresa de Almería, pero el producto que se le desarrolla utiliza la legislación fiscal y laboral de Cataluña, ¿cómo detectaríamos el problema?

a) Mediante un diseño exhaustivo de casos de prueba.
b) Se detectaría en los procesos de validación del software.
c) Se detectaría en los procesos de verificación del software.
d) No hay ningún problema que detectar porque es problema del cliente saber lo que compra.

Comprobar

Pruebas del software

Las pruebas en el ciclo de vida

Caso.

***Julia** le pregunta a **María** sobre cuando se realizarán todas las pruebas que se*

*están planificando. Ésta le responde que a lo largo de todo el **ciclo de vida**. Es importante que los desarrolladores hagan pruebas de cada una de las unidades que van desarrollando durante la implementación de la aplicación, pero también es importante que previamente hagan pruebas de integración de esas unidades junto a los ingenieros de pruebas durante la fase de diseño de módulos y componentes. Estos ingenieros, previamente habrán hecho las pruebas de sistema y de validación para verificar y validar el diseño del sisºtema y el análisis de requisitos del mismo. Y antes de todo eso, deberá haberse firmado el contrato con los clientes en la fase de establecimiento de los requisitos.*

*De esta forma, **Julia** y **María** se pusieron de acuerdo en los requisitos del cliente: ¿Qué es lo que la empresa quiere, y qué es lo que necesita para la aplicación que se iba a desarrollar para **SinUnEuro**?*

***José** ha validado y verificado el sistema, realizando un diseño del mismo, y aplicándole una serie de pruebas del sistema.*

*Al mismo tiempo, ha definido los módulos de la aplicación, y con la ayuda de **Víctor** y **María**, ha comprobado mediante unas pruebas de integración que todos los módulos se relacionan debidamente entre sí, y que interactúan de la manera prevista.*

*Finalmente, **Víctor** y **Carmen**, que son los que han programado la aplicación, realizando también las pruebas de unidades, de forma que han verificado que el software que han programado es correcto.*

Hemos estado explicando en el anterior apartado que las **pruebas** deben realizarse a lo largo de todo el proceso para detectar los errores lo antes posible. Cabe entonces preguntarse, ¿cómo encajan las pruebas en el ciclo de vida que hemos estado viendo en unidades anteriores?

Vamos a responder a esta pregunta a través de la siguiente figura donde se representa el **ciclo de vida del software** y el desarrollo de las pruebas. ¿A qué te recuerda la figura?

Por la forma que tiene este esquema se conoce como **modelo de ciclo de vida en uve**.

MODELO DE CICLO DE VIDA EN UVE: FASES DE PRUEBAS

En el modelo anterior puede verse que las **pruebas** se realizan a lo largo de todo el ciclo de vida sobre los diferentes productos que se van obteniendo. A la par que se avanza en el desarrollo del software se van planificando las pruebas que se

realizarán en cada fase del proyecto. Esta **planificación** se concretará en un **Plan de Pruebas** que se aplicará a cada producto desarrollado. Cuando se detectan errores en un producto debe volverse a la fase anterior para depurarlo y corregirlo, lo cual se indica con las flechas de vuelta de la parte izquierda de la figura. Las **herramientas CASE** ayudan a realizar estos procesos de pruebas, en concreto las encargadas de depurar errores de en los programas se conocen como "**debuggers**" (*eliminación de insectos*).

Como vimos en el apartado anterior el proceso de control del desarrollo del producto se llama **verificación** y cubrirá las fases de diseño e implementación del producto. Las personas implicadas en su realización serán los desarrolladores o programadores y el ingeniero de pruebas. Los **desarrolladores** realizarán pruebas sobre el código y los distintos módulos que lo integran, y el ingeniero sobre la integración de los módulos y el diseño del sistema.

Evaluar si el producto desarrollado cumple con los requisitos establecidos en el análisis de denomina **validación**, como ya vimos. La persona encargada de realizar las pruebas de validación serán los ingenieros de pruebas.

Por último, el cliente debe dar su visto bueno al producto para lo que hará las pruebas de **aceptación** en función del contrato que se firmó al principio.

Hay otro modelo de pruebas denominado **construcción y prueba diarias**, que consiste en desarrollar cada día una versión ejecutable del programa a la que se le aplica un plan de pruebas diario. Este esquema se utiliza ampliamente en los modelos de ciclo de vida de desarrollo incrementales. Tienen la ventaja de que mejoran la visibilidad del progreso, con lo cual, se detectan errores en etapas tempranas facilitando su corrección. Además, como siempre hay una versión disponible del sistema producen buenos resultados en los desarrollos rápidos del software.

Para saber más

Puedes conocer más sobre el diseño de las pruebas a lo largo de todo el ciclo de vida leyendo el documento "Aplicación práctica del diseño de pruebas de software a nivel de programación":

Biblioteca Digital

http://dspace.icesi.edu.co/dspace/handle/item/401 **[versión en cache]**

Pruebas del software

Recomendaciones para el desarrollo de las pruebas

Caso.

Víctor *ha visto el trabajo realizado, pero tiene la*

*impresión de que le falta mucho que aprender a la hora de decidir cómo puede elaborar y ejecutar un buen plan de pruebas. En ese sentido sabe que la experiencia es muy valiosa y le pide a **María** que le dé algunos consejos o recomendaciones que le puedan ayudar en el futuro a la hora de hacer nuevos planes de pruebas. **María** accede encantada, puesto que así, **Víctor** podrá pronto hacer una parte considerable del trabajo que hasta ahora venía haciendo ella misma.*

Como hemos visto a lo largo de todo este módulo se han propuesto diferentes **metodologías** de desarrollo de software con la intención de producir software de calidad y libre de errores. Además, en esta unidad hemos hablado de la necesidad de realizar un **plan de prueba** a lo largo de todo el desarrollo de software para detectar errores. Entonces, si seguimos todas estas indicaciones, ¿podemos estar seguros de que nuestro software está libre de errores?

Para asegurarme de que mi sistema no falla, debería **probarlo** completamente en todas las condiciones y situaciones, con todas las entradas de datos posibles y haciendo uso de toda su funcionalidad. ¿Es esto posible?

Imaginemos un programa sencillo que nos dice si una persona es mayor o menor de edad. Al programa le introducimos una edad y el sistema responde que es mayor de edad si su valor es mayor o igual a 18 años. En caso contrario responde que es menor de edad. El algoritmo del programa puede verse en la figura adjunta.

Para **probar el programa** podríamos introducir una edad de 10 años y funcionaría bien, otra edad de 25 años y funcionaría bien también, ¿sería suficiente? No, porque hay muchas más opciones, podríamos meter muchas más edades para asegurarnos, y podríamos probar a meter letras a ver si detecta el error, o podríamos probar a meter números negativos, o números muy grandes. En realidad vemos que aunque el programa es muy sencillo es imposible probarlo completamente, ya que el número de entradas posibles tiende a infinito. Pensemos entonces en si es posible probar una aplicación compleja con cientos de entradas, varias interfaces gráficas, miles de datos en una base de datos, etc. Por tanto, como las pruebas exhaustivas no son posibles debemos diseñar **casos de prueba** que sean lo más útiles posibles para encontrar posibles errores, y aún así, nunca podremos estar completamente seguros de que nuestro software está libre de errores.

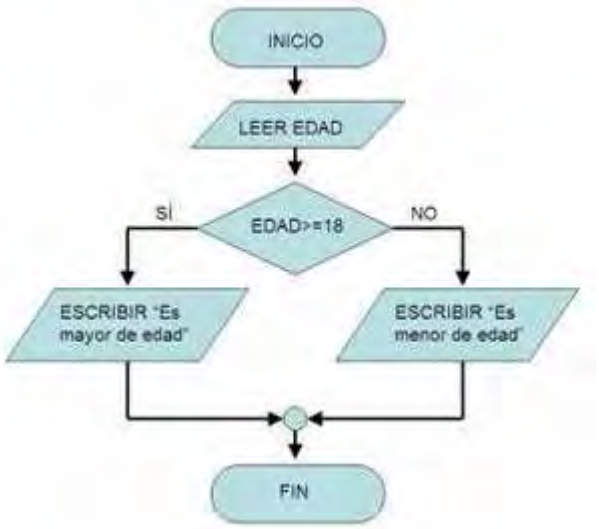
En este contexto, lo que sí podemos hacer es seguir las **recomendaciones** de los expertos e ingenieros experimentados sobre cómo deben realizarse las pruebas. Algunas de estas recomendaciones podemos verlas a continuación:

- Se comienza a evitar errores a través del uso de una **metodología de desarrollo del software**, por lo que es importante seguirla adecuadamente.
- La **finalidad de las pruebas** es encontrar errores, por tanto una prueba tendrá éxito cuando localice un error. En consecuencia los casos de prueba deben diseñarse de manera que provoquen situaciones con mayor

probabilidad de encontrar fallos.

- Las pruebas deben centrarse e insistir más en las partes o módulos que más se utilicen o sean más **críticos** para el sistema.
- No debe verse el proceso de prueba como algo rutinario, sino como un **proceso fundamental**, por lo que para hacerlo se requieren recursos, tiempo, personal experimentado y un proceso creativo. Es una mala práctica encargar las pruebas a los desarrolladores principiantes.

- No debe asociarse el error con la negligencia de un programador, la finalidad de las pruebas debe ser **encontrar fallos** y no desprestigiar la labor de las personas. Además, los errores no siempre son culpa del programador, pueden estar mal hechas las especificaciones del programa, o haber surgido cambios o nuevas necesidades de las que no se informó al programador.
- El programador no debe probar sus propios programas ya que desea (consciente o inconscientemente) demostrar que funcionan sin problemas, es mejor que otros programadores se encarguen de las pruebas. Hay estrategias, como la programación extrema, en la que el trabajo se hace por parejas para evitar errores. En las grandes empresas hay un **equipo de prueba** distinto al de desarrollo encargado de realizar todas las pruebas.
- Es fundamental que el trabajo de la prueba se desarrolle según un **plan** preestablecido donde se indiquen los **casos de prueba**. Los casos de prueba deben incluir tanto entradas correctas como incorrectas para evaluar el comportamiento del sistema en cualquier situación (pensemos en el caso expuesto en la introducción de la máquina de diagnóstico Therac 25).
- Los resultados de las pruebas deben estudiarse a conciencia para descubrir posibles síntomas de defectos. Por ello, es importante comprobar los posibles **efectos colaterales o secundarios** que pueda producir un módulo en otros módulos del sistema.



Autoevaluación

Para saber si un programa es correcto y está libre de errores:

a) Basta con seguir el diseño del algoritmo y saber que funciona.
b) Debo aplicarle distintos casos de prueba, y si los supera sin fallos, entonces está libre de errores.
c) Debo entregarlo al cliente para que lo pruebe y si queda satisfecho es suficiente.
d) Puedo seguir una metodología de desarrollo y aplicar un plan de pruebas con lo que me aseguro un cierto nivel de calidad y seguridad ante el fallo, pero nunca podré saber si está 100% libre de fallos.

Comprobar

Pruebas del software

Caso.

*Cuando **Carmen** y **Víctor** empezaron a diseñar los casos de prueba de la aplicación, éste todavía no tenía claro cómo se diseñaban esos casos ni sus procedimientos.*

__Carmen__ le explicó que deberían aislar cada parte del programa, cada módulo o función no trivial, para así aislar y acotar las posibles causas de error. A cada uno de ellos se le llama __unidad__, y cada unidad será probada por separado para ver que funciona correctamente.

Así, en la empresa __SinUnEuro__, cuando atiende a un cliente, típicamente tiene que comprobar si el cliente está registrado, si no lo está, registrarlo, preguntarle si lo que quiere hacer es una compra o una venta de un inmueble, si es una compra, mostrar ofertas, y si es una venta, registrar la venta.... Todas esas funciones forman parte del proceso de atención al cliente, pero cada una de ellas puede probarse de forma independiente a las demás. Eso nos llevará a definir casos de prueba y procedimientos de prueba independientes para registrar cliente, seleccionar venta o compra, mostrar ofertas, registrar ventas, etc.

A __Víctor__, la tarea le pareció enorme, pero __Carmen__ le tranquilizó: Existen herramientas Case capaces de facilitar el trabajo, para abordar las pruebas unitarias, y las de integración, permitiendo también documentarlas y automatizarlas en gran medida.

¿Qué se evalúa en las pruebas de unidades?

Una aplicación está compuesta por múltiples módulos software, los cuales están formados por unidades de programa. En el nivel de la codificación, como vimos en el apartado anterior, las pruebas a

realizar reciben el nombre de pruebas de unidades
porque se encargan de probar dichas unidades de código. Ya hemos dicho que el plan de pruebas incluirá una serie de estudios, cada uno de los cuales se compondrá de un **caso de prueba** (especifican qué probar) y un **procedimiento de prueba** (especifican cómo realizar los casos de pruebas).

En este apartado vamos a ver cómo se diseñan dichos casos de prueba y sus procedimientos correspondientes.

Como el proceso de prueba puede ser repetitivo y laborioso, es posible automatizar algunos procesos de manera que los procedimientos de prueba puedan hacerse con la ayuda de software. Este software debe ser preparado y recibe el nombre de **Componentes de prueba**. Su utilización es muy frecuente en las pruebas de integración que veremos en el apartado siguiente.

- A estas pruebas de unidades también se las llama **pruebas unitarias** y se definen como el procedimiento para probar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado.
- Una vez comprobado cada código se probarán los módulos combinados en las llamadas **Pruebas de Integración**, que se explicarán en el siguiente apartado.
- Las pruebas unitarias **pueden abarcar varios módulos simples e incluso un programa completo** si es necesario, pero siempre aplicando las mismas estrategias. No se trata de comprobar la integración de los distintos módulos sino de evaluar el proceso de cada módulo.

Las pruebas de unidades pueden ser realizadas también por los propios desarrolladores al mismo tiempo que se realiza la codificación para ayudarles en su trabajo, pero siempre es necesario que el equipo de pruebas realice pruebas posteriores sobre el módulo finalizado según el plan de pruebas diseñado.

- La **idea de las pruebas de unidades** es escribir casos de prueba para cada función no trivial o método del módulo de forma que cada caso sea independiente del resto, es decir, se aísla cada parte del programa para comprobar que es correcta. De esta forma los errores están más acotados y son más fáciles de localizar.

- Para que una **prueba unitaria sea de calidad** debe documentarse adecuadamente. Esto ayuda a elaborar la propia **documentación** del código. Es bueno utilizar componentes de pruebas para que las pruebas sean **automatizadas** y evitar, en la medida de lo posible, la intervención manual. Las pruebas deben cubrir la mayor parte posible del código, y ser **reutilizables**. La realización sistemática de pruebas da seguridad sobre el código desarrollado, proporciona **seguridad a la integración** de los distintos módulos, y por tanto a las pruebas de integración. Además, estas pruebas **fomentan en el programador el cambio y mejora del código** optimizando su estructura puesto que permiten hacer pruebas sobre los cambios y asegurarse de que no han introducido errores. A este proceso de reescritura del código para mejorarlo se le llama refactorización.

Podemos utilizar **herramientas CASE** para ayudarnos en el proceso del diseño, ejecución y

análisis de las pruebas. Una herramienta fundamental para detectar errores en el código es el depurador o **debugger**. Los depuradores pueden usarse para realizar inspecciones rigurosas sobre el comportamiento dinámico de los programas, por ejemplo ofrecen la posibilidad de ejecutar un programa paso a paso, permitiendo conocer dónde está en cada momento, y cuánto valen las variables. Sin embargo antes de utilizar un **depurador** debemos haber hecho las pruebas oportunas según los casos de prueba diseñados, ya que el depurador es el último paso para convencernos de nuestro análisis y afrontar la reparación con conocimiento de causa.

Es importante recordar que el objetivo de las pruebas unitarias no es descubrir todos los tipos de errores del código, ya que prueban unidades por sí solas. Por lo tanto, no descubrirán errores de integración, problemas de rendimiento y otros problemas que afectan a todo el sistema en su conjunto. Para ello, se realizan **otras pruebas** como vimos en el modelo en uve.

Para saber más

Si quieres profundizar en las pruebas de unidades y conocer algunas de las herramientas CASE disponibles para su desarrollo puedes visitar esta página de la Wikipedia:

Wikipedia: Prueba unitaria

http://es.wikipedia.org/wiki/Prueba_unitaria **[versión en cache]**

Pruebas del software

Tipos de pruebas de unidades

Caso.

Carmen *le ha dicho a* ***Víctor*** *que van a elaborar una serie de pruebas de caja negra y otras de caja blanca para las unidades.*

¿Qué diferencia hay? , le pregunta ***Víctor****.*

Las pruebas de caja blanca *sirven para probar la estructura dinámica del programa: Por ejemplo, para la prueba de la unidad de "registrar la venta", las pruebas de caja blanca nos permitirán recorrer todos los posibles caminos del código, viendo qué sucede en cada caso posible. Probaremos qué ocurre con las condiciones y los bucles que se ejecutan, y probaremos con datos que garanticen que se han dado todas las combinaciones posibles para dichas condiciones y bucles (introduciendo inmuebles ya vendidos, o de los que el cliente no sea propietario, o en los que todo sea correcto, junto a otros en los que el precio sea incorrecto, etc. Y para decidir qué valores probar, necesitamos saber cómo está hecho el código, para que no quede rincón sin ejecutar.*

¿Y qué hay de ***las pruebas de caja negra****?,*

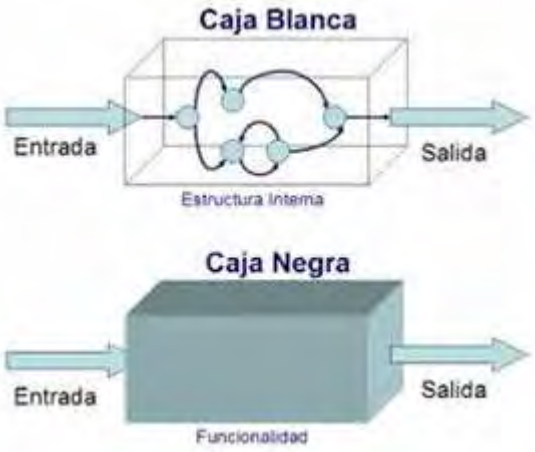
*pregunta de nuevo **Víctor**.*

*Permiten probar la **funcionalidad**. Es decir, lo que me interesa es comprobar qué salidas se producen para determinadas entradas, pero no cómo se obtienen las salidas a partir de las entradas. Se trata de probar todas las entradas posibles, y comprobar que las salidas son las esperadas. Por **ejemplo**, para la unidad de comprobar que el cliente está registrado, se trataría de elaborar un conjunto de entradas de clientes lo más exhaustivo posible, (en este caso es simple, clientes que estén registrados y que no lo estén) sin tener en cuenta para ello cómo los trata el programa, y comprobar que los resultados que obtenemos son los correctos, es decir, que siempre nos confirma el registro para los registrados, y nos contesta que no existe registro para los que no lo están, sin que me importe en absoluto los mecanismos o comprobaciones que haga internamente para llegar a esa conclusión.*

Hemos visto cómo deben ser las **pruebas de unidades**, pero, ¿en qué consisten exactamente?

Fundamentalmente suele haber dos enfoques para realizar las pruebas unitarias:

- **Enfoque estructural o pruebas de caja blanca**: se centran en la implementación de los programas para elegir los casos de prueba. Lo ideal sería buscar casos de prueba que recorran todos los caminos posibles del flujo de control del programa. El nombre de caja blanca se debe a que es posible ver el interior del módulo y comprobar cómo está hecho. (Un nombre más apropiado sería "pruebas de caja transparente").
- **Enfoque funcional o pruebas de caja negra**: se centran en la especificación del programa para estudiar las entradas de datos y las salidas de resultados. En este caso, no interesa la implementación del código sino su funcionalidad, es decir, estudiar los resultados que debo obtener a partir de determinadas entradas de datos. Para este enfoque lo ideal es diseñar casos de prueba para todas las posibles entradas y salidas del programa. Su nombre se debe a que en una caja negra no puedo ver el interior.



Estos dos enfoques se combinan para diseñar un plan de pruebas completo:

- las de **caja negra** prueban la funcionalidad del programa, y
- las de **caja blanca** prueban su estructura y comportamiento dinámico.

Además, también se pueden realizar **revisiones técnicas,** que son reuniones de desarrolladores y equipos de pruebas para inspeccionarel código y buscar errores. Lo que presentaremos en los próximos apartados son las diferentes estrategias y tipos de pruebas de unidades que podemos utilizar.

Autoevaluación

Si para un programa que controla un cajero automático de un banco, diseño casos de prueba a partir de las distintas operaciones accesibles desde el teclado de la máquina, estaría haciendo una prueba:

a) De caja negra.
b) De caja blanca.
c) De enfoque estructural.
d) De caja transparente.

Comprobar

Pruebas del software

Caso:

*En la empresa **SinUnEuro** han hecho una promoción de viviendas dirigidas a colectivos desfavorecidos (jóvenes, ancianos, mujeres maltratadas, parados de larga duración, etc) subvencionadas en gran parte con fondos municipales. Para la adjudicación de estas viviendas, el Excmo. Ayuntamiento ha impuesto una serie de restricciones y valoraciones que deberán tenerse en cuenta para hacer una baremación de las solicitudes, de 0 a 10, de forma que se determine la adjudicación de una vivienda a aquellas personas con baremo superior a 5.*

***Carmen** y **Víctor** deben encargarse de generar los casos de prueba, usando alguna de las distintas técnicas de pruebas de caja negra que existen.*

Como hemos explicado antes, las **pruebas unitarias de caja negra** se centran en la funcionalidad del programa, para lo cual, se diseñan casos de prueba que comprueben la especificación del programa.

Ya dijimos que es imposible probar todas las entradas y salidas del programa. Por tanto, debemos elegir

cuidadosamente los casos de prueba de manera que:

- sean **los menos posibles**, para que la prueba
- **se puedaejecutar en un tiempo razonable**, y al mismo tiempo,
- **que cubra la mayor variedad de entradas y salidas** posibles.

Para lograr esto se han diseñado distintas técnicas:

- **Clases de equivalencia**: se trata de determinar los tipos distintos de entradas y salidas, agruparlos y elegir casos de prueba para cada tipo o conjunto de datos de entrada y salida.
- **Análisis de los valores límite**: estudian los valores iniciales y finales ya que estadísticamente se ha demostrado que tienen más tendencia a detectar errores.
- **Estudio de errores típicos:** la experiencia nos dice que hay una serie de errores que suelen repetirse en muchos programas, se trataría de diseñar casos de prueba que provocasen las situaciones típicas de este tipo de errores.
- **Manejo de interfaz gráfica:** para probar el funcionamiento de las interfaces gráficas deben diseñarse casos de prueba que permitan descubrir errores en el manejo de ventanas, botones, iconos, etc.
- **Datos aleatorios:** se trata de utilizar una herramienta que automatice las pruebas y que genere de forma aleatoria los propios casos de prueba. Esta técnica no optimiza la elección de los casos de prueba. Si se hace durante el tiempo suficiente con muchos datos, podrá llegar a hacer una prueba bastante completa. Esta técnica podría usarse como complementaria a las anteriores o en casos en que no sea posible aplicar otra técnica.

Una ventaja de **las pruebas de caja negra es que son independientes del lenguaje o paradigma de programación utilizado**, de manera que son válidas tanto para programación estructurada como para programación orientada a objetos. A continuación veremos estas técnicas **y pondremos ejemplos** que clarifiquen su estrategia.

Pruebas del software

Clases de equivalencia

Hemos dicho que debo diseñar los casos de prueba de manera que prueben la mayor funcionalidad posible del programa pero que no incluyan demasiados valores, ¿por dónde empiezo?, ¿qué valores elijo?

Realizar particiones en los datos de entrada con clases de equivalencia es el método que proponemos para seleccionar los casos de prueba. Veamos en qué consiste.

Debemos seguir los siguientes **pasos**:

1. **Identificar las condiciones,** restricciones o contenidos de las entradas y las salidas.
2. **Identificar a partir de dichas condiciones las clases de equivalencia de las entradas y salidas**. Para identificar las clases el método propone algunas recomendaciones:
    1. Cada **elemento de clase** ha de ser tratado por el programa de igual manera, pero cada clase debe ser tratada de forma distinta con respecto a otra clase. Esto asegura que es suficiente con probar algún elemento de una clase para comprobar que el programa funciona correctamente para esa clase, y también garantiza que cubrimos diferentes tipos de datos de entrada con cada una de las clases.
    2. Las clases deben recoger tanto **datos válidos como erróneos**, ya que el programa debe estar preparado para cualquier circunstancia. Un programa no puede bloquearse porque el usuario meta un dato mal. Por ejemplo, si en la calculadora metemos una letra en lugar de un número, el programa avisará del error (por ejemplo con un pitido) pero no se detendrá, esperará a una entrada correcta.
    3. Si se especifica un **rango de valores** para los datos de entrada (por ejemplo: si se admiten del 10 al 50) se creará una clase válida $(10 \leq x \leq 50)$ y dos clases no válidas, una para los valores superiores $(x > 50)$ y otra para los inferiores $(x < 10)$.
    4. Si se especifica un **valor válido** de entrada y otros no válidos (por ejemplo, la entrada comienza con mayúscula), se crea una clase válida (con la primera letra mayúscula) y otra no válida (con la primera letra minúscula).
    5. Si se especifica un **número de valores** de entrada (por ejemplo, se deben introducir tres números seguidos), se creará una clase válida (con tres valores), y dos no válidas (una con menos de dos valores y otra con más de 3 valores).
    6. Si hay un conjunto de datos de entrada **concretos** válidos se generará una clase por cada valor válido (por ejemplo, si la entrada debe ser rojo, naranja, verde se generarán tres clases) y otra para un valor no válido (por ejemplo azul).

    7. Si no se han recogido ya con las clases anteriores, debe seleccionarse una clase para cada posible clase de **resultado**.

3. **Crear los casos de prueba** a partir de las clases de equivalencia detectadas. Esto incluye estos pasos:
    1. Elegir un valor que **represente** a cada clase de equivalencia.
    2. Diseñar casos de prueba que incluyan los valores de **todas las clases** de equivalencia identificadas.

Para **automatizar** y sistematizar este proceso podemos desarrollar componentes de prueba. Un inconveniente de las clases de equivalencia es que no tienen en cuenta las posibles relaciones que pueda haber entre los distintas clases, sus distintas combinaciones o el orden en que se ejecutan. La experiencia previa del equipo de pruebas puede ayudar a elegir los casos que más probabilidades tienen

de encontrar errores.

Autoevaluación

Si la entrada de un programa debe ser un número de 2 cifras, un caso de prueba que recoja todas las clases de equivalencia podría ser el siguiente:

a) 1, 60, 5476, pedro, 99
b) 3, 14, 100, A
c) 02, 27, 99, z
d) 1, 10, 100

Comprobar

Pruebas del software

Ejemplo de aplicación de clases de equivalencia

Caso.

***Carmen** y **Víctor** han elaborado los casos de prueba para el programa necesario para efectuar la adjudicación de viviendas subvencionadas aplicando la técnica de clases de equivalencia.*

El programa debe guardar las puntuaciones en el baremo de una serie de solicitantes de vivienda, y nos debe decir si han resultado beneficiados o no con la adjudicación de una vivienda subvencionada por el Ayuntamiento (puntuaciones en el baremo iguales o superiores a 5) o no. El programa admite como entrada el DNI y un número entre 0 y 10 sin decimales.

Para aclarar los conceptos expuestos antes vamos a realizar el ejemplo de aplicación de esta técnica:

1. Lo primero es **determinar las clases de equivalencia**, para lo cual, debemos analizar la entrada del programa siguiendo las recomendaciones anteriores como hacemos aquí:
   - Sabemos que el DNI se compone de forma general de 8 cifras numéricas y una letra, aunque algunos DNI antiguos tienen 7 cifras. Es decir en total la longitud válida será 8 o 9 caracteres. Esto genera estas clases:
     - 2 clases válidas: una con 8 caracteres y otra con 9
     - 2 clases no válidas: una con 7 o menos caracteres y otra con 10 o más.
   - Si nos fijamos en la **combinación de letras** y números las clases serán:
     - 2 clases válidas: una con 7 cifras y una letra y otra con 8 y la letra,

coinciden con las anteriores por lo que no hace falta repetirlas
    - 2 clases no válidas: cualquier combinación diferente de 8 y 9 cifras
- Para la puntuación en el baremo, si nos fijamos en el **rango**, tenemos las siguientes clases:
    - 1 válida: con un valor entre 0 y 10
    - 2 con datos no válidos: una con un valor negativo y otra con un valor superior a 10

- Si nos fijamos en los **decimales** de la puntuación en el baremo tenemos:
    - Una clase no válida: un número entre 0 y 10 con decimales. Podríamos pensar en una clase no válida con un número superior a 10 y decimales pero no es necesaria porque queda cubierta en el apartado anterior. Tampoco necesitamos una clase válida sin decimales porque eso coincide con la clase anterior.
- Debemos estudiar la salida, que puede ser "beneficiado con una adjudicación" o "no beneficiado con adjudicación", lo que genera 2 posibles clases:
    - Una clase con una puntuación en el baremo inferior a 5 y otra con más de 5 (en realidad una de las clases puede coincidir con la anterior por lo que sólo hace falta añadir una nueva clase)

En resumen tendríamos estas clases, que vamos a etiquetar con un número para facilitar su identificación:

CONDICIÓN	CLASES VÁLIDAS	Nº	CLASES NO VÁLIDAS	Nº
DNI	7 cifras más una letra	1	Menos de 7 caracteres	3
	8 cifras más una letra	2	Más de 8 caracteres	4
			8 caracteres que no sean 7 cifras más letra	5
			9 caracteres que no sean 8 cifras más letra	6
PUNTUACIÓN EN EL BAREMO	Un valor x con rango 0 ≤ x < 5	7	Un valor < 0	9
		8	Un valor > 10	10
	Un valor x con rango 5 ≤ x ≤ 10		Un valor con decimales	11

2. Ahora hay que **determinar los casos de prueba a partir de las clases de equivalencia** anteriores. Cada caso válido debe cubrir tantas clases válidas como sea posible, pero cada caso no válido debe cubrir una y sólo una clase no válida. Los representantes de cada clase los podemos elegir al azar. Así, podríamos tener por ejemplo estos casos de prueba que cubren todas las clases anteriores:

Casos de prueba válidos (DNI y puntuación baremo)	Clases de equivalencia cubiertas	Resultado

1234567A 4	1 7	No Adjudicada
12345678A 9	2 8	Adjudicada

Casos de prueba no válidos (DNI y nota)	Clases de equivalencia cubiertas	Resultado
123456A 2	3 7	Dato no válido
1234567890A 2	4 7	Dato no válido
1234ABC5 2	5 7	Dato no válido
ABCD12345 6	6 8	Dato no válido
1234567A -5	1 9	Dato no válido
1234567A 20	1 10	Dato no válido
1234567A 7,5	1 11	Dato no válido

Podemos observar que algunas clases se comprueban varias veces, pero esto no es un problema, ya que es necesario para cumplir con que se prueben todas las clases válidas y no válidas (un caso por prueba).

Autoevaluación

Si el programa de control por contraseña de un sistema admite sólo identificaciones de 6 caracteres, ¿qué casos de prueba tendríamos para las clases de equivalencia de la entrada?:

a) Hola, pepito, CASTAÑAS
b) 1234, 1234567, 1234567890
c) 12, 123456, PASO
d) CAT12P, 12AD, OOO123

Comprobar

Pruebas del software

Análisis de valores límite y errores típicos

Caso.

Víctor *sabe que la técnica que han usado antes no es más que una de las posibles, y sabe que tarde o temprano tendrá que usar otras técnicas, y piensa que debe aprenderlas cuanto antes, ahora que dispone de* ***Carmen*** *para ayudarle. Por eso le pide a* ***Carmen*** *que le explique cómo habrían elaborado los casos de prueba usando otras técnicas.* ***Carmen*** *considera que es una buena idea, y le pide*

*permiso a **María**, ya que tendría que dejar otras cosas para enseñarle esto a **Víctor**. Como hasta ahora, **Víctor** ha demostrado que todo lo que le han enseñado lo ha aplicado diligentemente en mejorar su trabajo en la empresa, le dan permiso a **Carmen**, que decide continuar la formación de **Víctor** con la técnica de análisis de valores límite y la de análisis de errores típicos, aplicadas sobre el mismo ejemplo anterior, la adjudicación de viviendas subvencionadas por el ayuntamiento.*

Hemos visto cómo elegir las **clases de equivalencia**, pero antes comentamos otras estrategias para determinar los casos de prueba del método de caja negra. En este apartado explicaremos estas técnicas que nos van a servir para seleccionar mejor las clases de equivalencia.

Empecemos por el **Análisis de los valores límite.** ¿Por qué es una técnica adecuada fijarse especialmente en esos valores límite?

Se ha podido demostrar que los casos de prueba que se centran en los valores límites producen un mejor resultado para la detección de defectos. De esta manera al elegir el elemento representativo de la clase de equivalencia, en lugar de coger cualquiera, se eligen los valores al límite y uno intermedio. Además, también se intenta que los valores a la entrada provoquen valores límite en los resultados.

Este método, por tanto, seguiría los mismos pasos explicados para las clases de equivalencia, pero a la hora de elegir los representantes de cada clase se seguirían estas recomendaciones:

- En los rangos de valores vamos a elegir **los extremos del rango y el valor medio**.
- Si se especifican una serie de valores, elegiremos **el mayor, el menor, el anterior del menor, y el posterior al mayor**.
- Si el resultado se mueve en un determinado rango, debemos **elegir datos a la entrada para provocar las salidas mínima, máxima y un valor intermedio**.
- Si el programa maneja **una lista o tabla**, debemos **elegir el elemento primero, el último y el intermedio**.

Si aplicamos este esquema al ejemplo anterior los casos de prueba podrían quedar de la siguiente manera:

Casos de prueba válidos	Clases de equivalencia cubiertas	Resultado
1234567A **0**	1 7	No Adjudicada
12345678A **10**	2 8	Adjudicada

Casos de prueba no válidos	Clases de equivalencia cubiertas	Resultado
123456A **-1**	3 9	Dato no válido

1234567890A **11**	4 10	Dato no válido
1234ABC5 7,5	5 11	Dato no válido
ABCD12345 **5**	6 8	Dato no válido

También podemos aprovecharnos de la **experiencia previa**. Sabemos que hay una serie de errores que se repiten muchos en los programas, y podría ser una buena estrategia utilizar casos de prueba que se centrasen en buscar estos fallos. De esta manera mejoramos la elección de los representantes de las clases de equivalencia. Así, podemos seguir estas recomendaciones:

- El valor cero suele provocar errores, por ejemplo, una división por cero bloquea al programa. **Si tenemos la posibilidad de introducir ceros en la entrada deberíamos elegirlos en los casos de prueba.**
- Cuando hay que introducir **una lista de valores** debemos centrarnos en **la posibilidad de no introducir ningún valor, o introducir uno sólo**.
- Debemos pensar que el usuario puede introducir entradas extrañas, por lo que **es recomendable ponerse en el peor lugar a la hora de pensar cómo se manejará el programa**.
- Los **desbordamientos de memoria** son habituales por lo que **podemos probar a introducir valores lo más grandes posibles.**

Siguiendo estas recomendaciones, podríamos añadir un caso de prueba al ejemplo anterior en el que la entrada de datos esté vacía, es decir, que no introduzcamos ninguna nota, por ejemplo. La tabla de casos válidos sería la misma pero la de no válidos tendría una fila más:

Casos de prueba no válidos	Clases de equivalencia cubiertas	Resultado
123456A **-1**	3 9	Dato no válido
1234567890A **11**	4 10	Dato no válido
1234ABC5 7,5	5 11	Dato no válido
ABCD12345 **5**	6 8	Dato no válido
12345678A	**1 (puntuación baremo vacía)**	Dato no válido

Para saber más

El proceso de selección de los casos de prueba puede automatizarse. Si quieres profundizar en el tema de la selección de los casos de prueba con ayuda de herramientas puedes consultar este documento de un profesor de la Universidad de Sevilla:

Propuestas para la generación de casos de prueba

http://www.lsi.us.es/docs/informes/LSI-2005-01.pdf **[versión en cache]**

Manejo de interfaz gráfica

Caso.

*La empresa **SinUnEuro** necesita que haya un control de acceso a sus ordenadores, de forma que cada equipo pueda ser usado únicamente por el personal de la empresa autorizado. **Carmen** y **Víctor** son los encargados de elaborar los casos de prueba para esta aplicación, que tiene una interfaz gráfica, ya que es mediante una ventana como se controla el acceso al sistema mediante la introducción de un nombre de usuario y una contraseña (password). El sistema comprueba si existe una cuenta con ese nombre y contraseña y, si es así, se le da permiso para entrar. Si existe el nombre de usuario pero la contraseña es incorrecta permite volver a introducir la contraseña hasta un máximo de tres veces.*

En los ejemplos anteriores hemos hablado de entradas de texto, pero no hemos comentado nada del manejo de los **entornos gráficos** en los que introducimos estas entradas de texto. ¡Habrá que remediarlo! ¿Piensas que es lo mismo? Vamos a verlo.

En los programas actuales toda la **interconexión** con la máquina se suele hacer con sistemas gráficos que cada vez son más complejos por lo que pueden generar errores. Como ejemplo de interfaz gráfico podemos ver la imagen del programa de retoque de imágenes "Gimp". En este apartado vamos a explicar cómo diseñar casos de prueba para los entornos gráficos.

Las **pruebas de interfaces gráficasde usuario** deben incluir:

- Pruebas sobre **ventanas**: iconos cerrar, minimizar, etc.
- Pruebas sobre **menús** y uso del **ratón**: funcionamiento de menús superiores, botones, etc.
- Pruebas de **entrada de datos**: cuadros de texto, listas desplegable, etc.
- Pruebas de documentación y **ayuda** del programa incluyendo la ayuda contextual.

Obviamente, no podemos revisar todas las combinaciones de usos de una interfaz, pero los casos de prueba deben incluir la revisión del funcionamiento de todos los elementos anteriores, al menos, **una prueba para cada elemento**.

Vamos a fijarnos en la imagen adjunta y vamos a proponer casos de prueba siguiendo el esquema anterior. La ventana controla el acceso a un sistema mediante la introducción de un nombre de usuario y una contraseña (password). El sistema comprueba si existe una cuenta con ese nombre y contraseña y, si es así, se le da permiso para entrar. Si existe el nombre de usuario pero la contraseña es incorrecta permite volver a introducir la contraseña hasta un máximo de tres veces.

Incluimos a continuación algunos de los casos de

prueba que se podrían utilizar con este programa:

- **Caso de Prueba 1**:
  Entrada: Pulsar botón minimizar
  Condiciones de ejecución: la ventana debe estar abierta
  Resultado esperado: la ventana se minimiza a la barra de tareas

- **Caso de Prueba 2**:
  Entrada: usuario "correcto" password "correcta". Pulsar botón "Acceder al sistema"
  Condiciones de ejecución: en la tabla existe ese usuario con ese password y con 1 intento fallido anterior (número inferior a 3)
  Resultado esperado: dar acceso y reflejar que el número de intentos para el usuario "correcto" es cero en la tabla USUARIO (cuenta,password,numIntentos)

- **Caso de prueba 3:**
  Entrada: usuario "incorrecto" password "correcta". Pulsar botón "Acceder al sistema"
  Condiciones de ejecución: en la tabla no existe ese usuario con ese password
  Resultado esperado: no dar acceso

- **Caso de prueba 4:**
  Entrada: usuario "correcto" password "correcta". Pulsar botón "Acceder al sistema"
  Condiciones de ejecución: en la tabla existe ese usuario con ese password y con 4 intentos fallidos (número superior a 3)
  Resultado esperado: no dar acceso y añadir para el usuario "correcto" un número de intentos de 5 en la tabla USUARIO(cuenta,passord,numIntentos)

Autoevaluación

En una interfaz gráfica debo diseñar casos de prueba para los siguientes elementos:

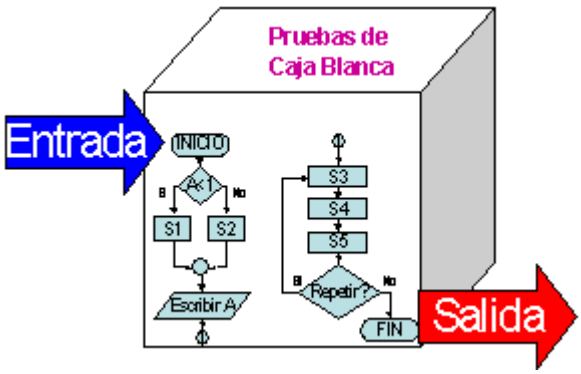
a) Iconos y menús.
b) Cuadros de texto y listas desplegables.
c) Tamaño y movimiento de la ventana.
d) Todos los elementos anteriores y otros que compongan la interfaz gráfica.

Comprobar

Caso.
***Víctor** ha visto claramente cómo elaborar las pruebas*

*de caja negra, pero se pregunta qué tipos de pruebas de caja blanca se podrían usar. **Carmen** ya está ocupada con algunas cosas del trabajo, y le pone algunos ejemplos de pruebas de cobertura de flujo de control y de cobertura lógica. Las primeras se encargan de probar que al menos se pasa una vez por cada camino del programa. Las segundas prueban que el funcionamiento interno del código es el adecuado.*

Como ya comentamos, las pruebas de **caja blanca** se diseñan a partir del código del programa que queremos probar.



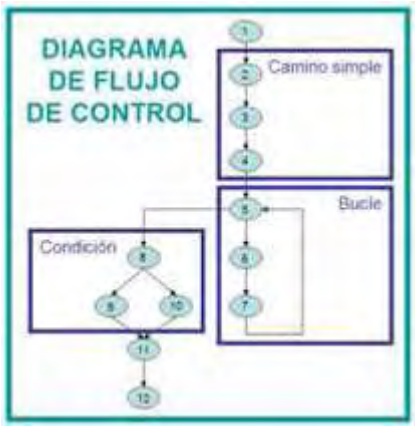
Su objetivo es asegurar que la operativa interna del programa se ajusta a las especificaciones iniciales y que todos los componentes internos se han probado adecuadamente.

Al igual que ocurría con las pruebas de caja negra, **las pruebas exhaustivas son impracticables**, ya que el número de combinaciones en la ejecución de un programa es excesivo. Por tanto, debemos diseñar estrategias que nos ofrezcan una seguridad aceptable para descubrir errores. Veamos esas estrategias en los siguientes apartados de la unidad.

Pruebas del software

Prueba de cobertura de flujo de control

El método que se propone es el de **cobertura de flujo de control** que consiste en utilizar la estructura de control del programa para obtener los casos de prueba, que son diseñados **de manera que se garantice que todos los caminos de ejecución del programa quedan probados**, es decir que, **al menos, se pasa una vez por cada camino del programa.**



Una posible técnica para llevar a cabo este método se muestra en la figura adjunta, y consiste en obtener un diagrama del flujo de control que represente al código y probar todos los **caminos simples**, todas las **condiciones** y todos los **bucles** del programa.

Puede ser imposible cubrir el 100% si el programa es muy complejo, pero podemos tener un mínimo de garantías de eficacia si seguimos estas sugerencias para diseñar los casos de prueba fijándonos en estos tres elementos:

- **Camino simple**: Debemos diseñar un caso de prueba por cada camino independiente de manera que ejecutemos al menos una vez cada sentencia. Para ello es necesario, determinar los posibles "caminos" independientes y preparar suficientes casos de prueba para recorrer todos los caminos.
- **Condiciones**: Debemos diseñar suficientes casos de prueba para que todas las condiciones del programa se evalúen a cierto/falso. Si las condiciones son múltiples, habrá que dividirlas en expresiones simples (una por cada operando lógico o comparación) de manera que se debe probar que se cumpla o no cada parte de cada condición.
- **Bucles**: Para los bucles debemos diseñar casos de prueba de manera que se intente ejecutar un bucle en diferentes situaciones límite. Hay que distinguir el tipo de ciclos:
  - Si el **bucle es simple**, debe probarse para que el bucle no se ejecute (0 veces), se ejecute 1 vez, y se ejecute un número medio de veces si no hay un número límite de ejecuciones. Este número medio se sustituye por ejecutarlo n-1 veces, n veces y n+1 veces si hay un número máximo de ejecuciones (siendo n el número máximo del contador del bucle). Con esto nos aseguramos que los contadores están bien inicializados y que la operativa es correcta.
  - Si hay **bucles anidados**, se deben repetir un número medio (típico) los bucles internos, el mínimo los externos, y variar las repeticiones del bucle intermedio ensayado. El proceso se repite con cada nivel de anidamiento.

Veamos un par de **ejemplos** de bucles:

```
for (contador=0; contador <=10; contador ++)
{ ... }
```

Deberían diseñarse casos de prueba para intentar ejecutar el ciclo el siguiente número de veces: 0, 1, 9, 10, 11

```
while (control == false)
{ ... }
```

Deberían diseñarse casos de prueba para intentar ejecutar el ciclo el siguiente número de veces: 0, 1, 5 (Suponiendo que 5 es un número habitual de veces).

Hemos comentado que la **estrategia de cobertura de flujo de control** requiere diseñar casos de prueba suficientes para recorrer toda la lógica del programa.

¿Sabemos de cuántos casos estamos hablando?, ¿cómo se calcula?

El matemático McCabe llamó al número de caminos independientes de un diagrama de flujo de control, como **complejidad ciclomática** (CC), y propuso la siguiente fórmula para calcularlo:

CC = Número de ramas - Número de nodos + 2

Para aplicar la **fórmula** es necesario que las condiciones múltiples hayan dicho **descompuestas** en expresiones simples. La complejidad ciclomática me dice el número de caminos independientes que tiene el diagrama que representa a mi programa, y habrá que diseñar un caso de prueba para cada camino independiente. La complejidad ciclomática sólo tiene en cuenta una única ejecución de los bucles, es decir, si queremos probarlos 0, n-1, n y n+1 veces hay que añadir más casos de prueba. Una complejidad ciclomática superior a 10 puede indicar que el programa es demasiado complicado y quizás debería modificarse o refactorizarse para hacerlo más simple.

Por **ejemplo**, en el diagrama de la figura anterior tendríamos una complejidad ciclomática de tres: CC = 13 - 12 + 2 = 3. Esto significa que habrá que diseñar tres casos de prueba (pruebas múltiples del bucle aparte). Para elegir los recorridos a realizar debemos comenzar por los más simples. Así, los tres recorridos que deberían tenerse en cuenta a la hora de diseñar los casos de prueba podrían ser: (1 2 3 4 5 8 9 11 12) (1 2 3 4 5 8 10 11 12) (1 2 3 4 5 6 7).

Para ver el resultado de las pruebas y comprobar que efectivamente se están recorriendo estos nodos debemos crear un mecanismo que nos guíe. Una solución simple sería añadir en el código sentencias de salida por pantalla que hagan indicaciones de valores de variables, puntos del programa (nodos) que se están ejecutando, o evaluaciones de las condiciones, etc. Pero, esto obliga a modificar el código. Una solución mucho mejor es utilizar **herramientas CASE** de ayuda como el depurador o debugger que realizan éstas y otras útiles tareas.La mayoría de los entornos integrados de desarrollo, como por ejemplo NetBeans, incluyen herramientas de depuración para este fin.

En cualquier caso, si el programa es muy complejo, es posible que no sea posible diseñar todos los casos de prueba sugeridos. Ayudándonos de la **experiencia** del equipo de pruebas, deberemos evaluar la oportunidad y alcance de las pruebas alcanzando un compromiso, entre el tiempo dedicado a las pruebas, y la garantía de realizar un plan completo de pruebas. Los **componentes de pruebas** nos

permitirán automatizar todos estos procesos.

Pruebas del software

Ejemplo de prueba de cobertura lógica

Caso.

A la hora de buscar los clientes a los que se ha adjudicado una vivienda protegida, se dispone de un array con todos los clientes, ordenados por puntuación en el baremo. El algoritmo de búsqueda empleado es la búsqueda binaria, y ***Víctor*** *y* ***Carmen*** *elaboran un plan de prueba de* ***cobertura lógica*** *para este caso.*

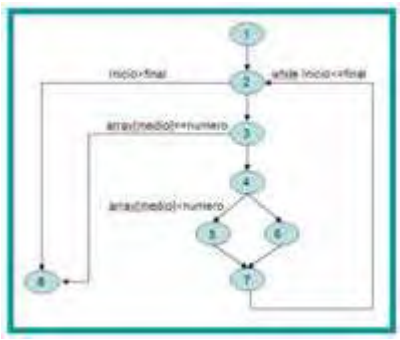
Vamos a aplicar esta técnica de **cobertura lógica** a un problema concreto y podremos aclarar mejor su aplicación. Supongamos un programa para realizar una búsqueda en una lista de elementos ordenada.

Hay diferentes métodos, pero vamos a utilizar la **búsqueda binaria**. La lista va a estar implementada mediante una tabla o "array" que estará ordenado. El proceso de búsqueda consiste en:

- tomar el elemento que se encuentra a la mitad del array (N/2) y compararlo con el elemento buscado.
- Si el elemento buscado es menor, ahora sólo buscaremos del inicio hasta la mitad -1, en caso contrario, buscaremos de la mitad +1 hasta el final. Cabe mencionar que si (N/2) resulta en un número con decimales, sólo se toma la parte entera. Este procedimiento se repite en la mitad del array elegido hasta encontrar el elemento buscado o acabar de dividir el array. A continuación incluimos una implementación en Java de este algoritmo que vamos a usar para aplicar este método de prueba.

Del programa anterior podemos derivar fácilmente el siguiente **diagrama de flujo**

de control, para ello basta con seguir la secuencia lógica del programa (entre paréntesis indicamos en el código los nodos del diagrama). Las **condiciones** son expresiones simples, luego no hace falta descomponerlas y el bucle es sencillo. En la figura siguiente mostramos el diagrama.



Calculemos la **complejidad ciclomática de McCabe**, teniendo en cuenta que el número de nodos es 8, y el número de ramas es 10:

$$CC = 10 - 8 + 2 = 4$$

Por tanto debemos diseñar 4 casos de prueba que realicen estos recorridos:

- 1 2 8
- 1 2 3 8
- 1 2 3 4 5 7
- 1 2 3 4 6 7

A estos recorridos hay que añadir las pruebas del bucle para que se ejecute 0, 1, n-1, n y n+1 veces. Aunque, en este caso el número máximo de veces que se ejecuta el bucle depende del número de elementos, por lo que basta con probarlo 0, 1 y n veces. En los casos de prueba de búsqueda de un elemento en un array de varios elementos se recorren ambos recorridos (1 2 3 4 5 7) (1 2 3 4 6 7). Además, es importante tener en cuenta que al nodo 8 podremos llegar con el número encontrado o no, por lo que hay que probar ambos casos. A continuación detallamos estos cuatro **casos de prueba** para el algoritmo de la búsqueda binaria:

Caso de prueba	Recorrido	Veces que se ejecuta el bucle	Elementos del array	Número a buscar	Resultado esperado (posición del número)
1	1 2 8	0	vacío	x	-1
2	1 2 3 8	1	7 10 15	10	1
3	1 2 3 4 5 7 1 2 3 4 6 7	3	1 3 4 6 10 12	6	3
	1 2 3 4 5 7		1 3 4 6 10		

4 1 2 3 4 6 7 3 12 2 -1

Para saber más

Podemos consultar más información sobre el diseño y desarrollo de casos y componentes de prueba en la siguiente página web del Departamento de Lenguajes y Sistemas Informáticos de la Universidad del País Vasco:

Diseño de casos de prueba

http://siul02.si.ehu.es/~alfredo/

http://siul02.si.ehu.es/~alfredo/iso/Tema3.pdf **[versión en cache]**

Pruebas del software

Caso.

Todos el personal de ***Si Andalucía*** *que está implicado en el proyecto de la empresa* ***SinUnEuro.*** *Se han reunido para hacer una revisión técnica, que es útil para detectar no sólo los síntomas, sino también las causas de algunos fallos. Todos han revisado el código, haciendo bueno el dicho de que cuatro ojos ven más que dos. De esta forma, se han detectado algunos fallos, que se han corregido, de forma previa a las pruebas.* ***Víctor*** *le comenta a* ***Carmen*** *que le ha parecido una reunión bastante informal. Ésta le comenta que efectivamente las reuniones son por lo general bastante informales, pero que a veces usan una técnica ligeramente distinta:* ***José*** *y* ***María****, a modo de expertos, revisan todo el código que ha elaborado algún desarrollador, y le hacen preguntas, y debaten sobre los distintos fallos y errores. Estas reuniones se denominan* ***Walkthroughs.*** *De esta forma se suelen detectar más errores que con la aplicación de las pruebas sin más. De hecho,* ***Carmen*** *recuerda algo así es lo que los profesores hacían con ellos a la hora de corregir los proyectos finales de los módulos de programación, cuando estudiaba en el* ***IES Aguadulce****. Normalmente, cuando se sentaban 3 profesores a revisar el proyecto de un alumno y a preguntarle cosas sobre el mismo, acababan apareciendo errores que ni el alumno ni su profesor habían detectado en todo el proceso de desarrollo. Y todavía existen otras reuniones más formales para revisar los errores del código, las* ***inspecciones,*** *pero como implican a mucha gente, sólo merece la pena aplicarlas en proyectos de gran tamaño, y no son frecuentes en el caso de* ***SI Andalucía.***

Hemos visto diferentes estrategias de prueba de código

basadas en técnicas de caja negra y blanca, pero ¿alguna vez has utilizado alguna otra estrategia para buscar errores en un programa?

Seguro que sí. Todos hemos utilizado otra estrategia de revisión de código que es **la inspección directa del código** individualmente o en grupo. Probablemente cuando hayas querido asegurarte de que un programa funcionaba bien, has pedido a un compañero, profesor o amigo que le echase un vistazo al código. Pues, cuando esta revisión se realiza de forma sistematizada y siguiendo unas normas estaremos hablando de las **revisiones,** que son muy utilizada en las empresas como complemento de las pruebas.

Los **objetivos** fundamentales de estas revisiones coinciden con lo indicado para las pruebas, es decir, **la detección de fallos lo antes posible para su eliminación**. Una ventaja de las revisiones sobre las pruebas, es que las pruebas detectan el síntoma del defecto, pero las revisiones detectan también la causa pues revisan directamente el código. Una desventaja es que la revisión directa no puede hacerse globalmente sobre todo el código del programa sino que se analizan módulos.

En realidad, las revisiones van más allá de la búsqueda de errores en el código, e incluyen todo tipo de evaluaciones para detectar defectos en los requerimientos, diseño, implementación o cualquier otro producto del desarrollo del proyecto software. Es decir, **las revisiones se utilizan tanto en las pruebas como en el control de la calidad,** por ello, se usan en la comprobación de módulos de las pruebas unitarias y en el resto de validaciones y verificaciones del sistema. Cuando se utilizan en la comprobación del código se denominan revisiones técnicas.

Veamos algunas de las estrategias incluidas en las **revisiones técnicas**:

- **Reuniones**: son las menos formales de las distintas revisiones por lo que son muy flexibles y requieren poca preparación previa. En ellas varios desarrolladores revisan un documento o producto para mejorar su calidad y detectar errores antes de la prueba. En el caso de la revisión de la implementación de los módulos de un programa, consistiría en reunirse para leer el código y buscar errores.

- **Walkthroughs**: Es un proceso más formal que las reuniones y normalmente se aplica a la revisión del código, aunque a veces, puede usarse con productos de otras etapas de desarrollo. Para ello, el desarrollador se reúne con revisores expertos. El desarrollador entrega el código a los revisores que lo revisan e indican cualquier error o duda que haya acerca del código. En la sesión de walkthough el desarrollador explica las dudas y se discute sobre los posibles errores. Se utilizan como complemento de las pruebas. Algunos estudios han revelado que los walkthroughs detectan más errores que las pruebas en el mismo tiempo, por lo que, son muy apropiados para el desarrollo rápido de software.

- **Inspecciones**: son las más formales y requieren de preparación previa. Se utilizan para la verificación de todo el proceso de desarrollo. Las

inspecciones están organizadas por un **moderador** que reparte roles y tareas entre los participantes. En el caso de la inspección de código, el moderador y los revisores inspeccionan el software antes de la reunión siguiendo unas listas de control en las que el revisor comprueba y anota si el producto cumple ciertas características**. Durante la reunión de inspección, el moderador organiza, el autor explica el código, los revisores identifican los errores y un secretario recoge lo tratado en la reunión.** En la reunión se analizan los errores y se discuten soluciones. Tras la inspección, el moderador, con ayuda del secretario, elabora un informe describiendo los defectos encontrados y lo que debe hacerse con ellos. Las inspecciones son muy efectivas aunque requieren la implicación de bastante personal.

Además de las revisiones, se realizan **auditorías del software**, cuyo objetivo es asegurar que los productos y procesos del desarrollo se ajustan a los estándares, niveles de calidad, especificaciones y procedimientos previstos en el proyecto. Lo normal, es que las auditorías sean realizadas por personas u organizaciones ajenas a la empresa de desarrollo para asegurar la objetividad.

Autoevaluación

El orden de las revisiones de menos a más formal es:

a) Inspecciones, Walkthroughs, Reuniones
b) Reuniones, Walkthroughs, Inspecciones, Auditorías
c) Reuniones, Walkthroughs, Inspecciones
d) Reuniones, Inspecciones, Walkthroughs

Comprobar

Pruebas del software

Caso.

Carmen *le indica a* ***Víctor*** *que deben realizar las* ***pruebas de integración****.*

¿En qué consisten?, pregunta él.

En probar, una vez que sabemos que cada unidad funciona bien por separado, que todas las unidades se comportan de la manera adecuada al relacionarse unas con otras. Consiste por tanto, en probar la ejecución conjunta de varias unidades, hasta llegar a probar todo el sistema- le comenta ***Carmen****.*

Para que lo entienda mejor, le pone como ejemplo la aplicación que están desarrollando para ***SinUnEuro****. Han probado que el procedimiento de adjudicación de viviendas subvencionadas funciona correctamente, pero ese procedimiento tiene que relacionarse con el de registro de*

clientes, y con el de ventas, que también funcionan correctamente, pero que no sabemos si se comunican de forma adecuada entre sí. Por ejemplo, cuando una vivienda es adjudicada, deberían cogerse los datos adecuadamente del módulo de registro para el cliente, y la casa debería dejar de estar disponible para el módulo ventas. Ese tipo de cosas son las que se comprueban en las pruebas de integración.

¿Y existe un único tipo de pruebas de integración?

Nada de eso, existen varios tipos, como las ascendentes, descendentes, combinadas y en big bang.

¿Y cuál será más adecuada?

Depende del problema. Cada una tiene sus ventajas e inconvenientes.

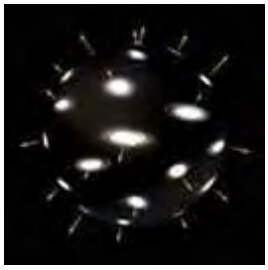
Hemos visto las pruebas de unidades de una aplicación, pero ¿es esto suficiente para asegurarnos que hemos probado el software?

Cuando expusimos el modelo de pruebas del software a lo largo del ciclo de vida comentamos que era necesario **probar también las diferentes unidades combinadas**. En este apartado vamos a revisar las **pruebas de integración**.

Una vez que probamos los componentes individuales del programa y nos hemos asegurado de que no contienen errores, debemos integrarlos para crear un sistema parcial o completo que debe ser probado, éste es el nivel de las pruebas de integración. **Un objetivo importante de las pruebas de integración es localizar errores en las interfaces entre las distintas unidades**. Las interfaces en programación son el método de comunicarse las distintas unidades de una aplicación para enviarse datos, mensajes, órdenes, parámetros, etc.

Pruebas del software

Estrategias de integración de unidades



En **aplicaciones grandes**, donde es necesario integrar muchos componentes, cuando se descubre un error, es difícil localizarlo. ¿Cómo podemos solucionarlo?

Para ello, los elementos no se integran todos al mismo tiempo sino que se utilizan diferentes estrategias de integración incremental, que básicamente, consisten en:

- integrar unos pocos componentes,
- probarlos y
- si no hay errores añadir nuevos componentes,
- si hay errores se solucionan y se vuelven a probar.

Con este proceso se facilita la localización del error cuando se produzca porque se sabe cuáles son los últimos módulos que se han integrado y cuándo se ha producido el error. La organización clásica de los módulos es una **estructura jerárquica por niveles**, en la parte alta estará el módulo o módulos principales que hacen llamadas a los módulos subordinados de nivel inferior, y así sucesivamente cada nivel utilizará módulos de nivel inferior hasta llegar a los módulos terminales. Los módulos superiores serán los más cercanos al usuario, es decir los que incluyen la interfaz de usuario (entorno gráfico, menús, ayudas, etc) y los módulos inferiores son los más cercanos a la estructura física de la aplicación (bases de datos, hardware, etc).

Veamos diferentes **estrategias de desarrollo de las pruebas de integración**:

- **Pruebas de integración ascendente**:
- Comienza combinando en grupos las unidades o módulos de bajo nivel para probarlos (pruebas dobles, triples, cuádruples, etc. según el número de módulos combinados).
- Para realizar las pruebas es necesario diseñar módulos software como componentes de pruebas, denominados manejadores de pruebas o impulsores o "*driver*", que permiten simular el comportamiento de los módulos superiores.
- Los manejadores tienen la misma interfaz del módulo al que sustituyen pero no realiza su función, por tanto, son fáciles de construir ya que no simulan la actividad de los módulos padre sino que sirven simplemente para poder ejecutar el módulo que se va a probar.
- Con los manejadores realizamos las pruebas de las interfaces y los módulos.
- Cuando se ha probado un grupo de módulos se continúa el proceso con el resto de agrupaciones del mismo nivel.
- Una vez probado un nivel, se sustituyen los manejadores por los módulos superiores y se repite el proceso, para lo cual será necesario ir desarrollando nuevos manejadores superiores.
- El proceso se irá repitiendo sucesivamente hasta que se integren todos los niveles,

En la siguiente animación podemos ver ese proceso de forma esquemática:

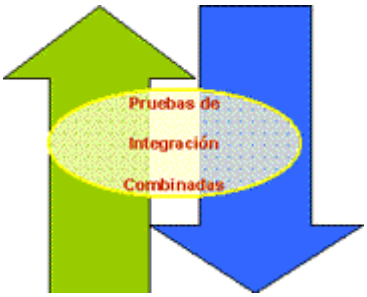
Pruebas de integración ascendente

- **Pruebas de integración descendente**:
- En primer lugar se prueban los módulos de nivel superior y luego se van integrando los componentes de la siguiente capa.
- Una vez probados los componentes de esta capa se prueban los del nivel inferior y así sucesivamente hasta involucrar a todas las capas.
- Para poder realizar las pruebas descendentes sin incluir los módulos inferiores, necesitamos también componentes, en concreto es necesario crear módulos software simuladores de pruebas llamados ficticios o "*stub*" que emulen su comportamiento y tengan su misma interfaz.
- Los módulos ficticios simulan las funciones que deberían hacer los módulos

reales a los que sustituyen por ello son más difíciles de construir que los manejadores.
- En este caso no se necesitan manejadores de pruebas como ocurría en las ascendentes.

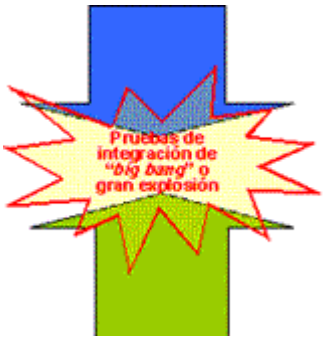
Pruebas de integración descendente

- **Pruebas de integración combinadas**:
- Combinan la integración ascendente y descendente.
- Los módulos individuales se prueban con *drivers* y *stubs* y
- luego se van probando las capas superiores e inferiores sustituyendo los *drivers* y *stubs* por los módulos probados.
- En los niveles o capas superiores se usa una estrategia descendente y en las inferiores ascendente hasta encontrarse en alguna capa intermedia.
- Las pruebas pueden realizarse en paralelo para ahorrar tiempo.



Pruebas de integración combinadas

- **Pruebas de integración de "*big bang*" o gran explosión**:
- Primero se prueban todos los métodos individualmente y después todos los componentes del sistema se integran a la vez y se realizan pruebas.
- Como hemos dicho antes, esta estrategia parece simple de aplicar, pero tiene el importante inconveniente de que cuando se descubre un fallo es muy difícil localizarlo para poder corregirlo.



Pruebas de integración de "*big bang*" o gran explosión

En todas las estrategias debemos decidir el **orden** en que se van integrando los módulos. Un buen criterio es comenzar con los módulos críticos del sistema y los que sean más susceptibles de contener errores por ser más complejos o estar relacionados con más requerimientos. Hemos hablado de los componentes de pruebas, de los manejadores de pruebas (drivers) y de los módulos simuladores (stubs), que son necesarios para realizar las pruebas. Existe **software** que ayuda y automatiza el desarrollo y uso de estos componentes. Las pruebas de los diferentes grupos de módulos deben centrarse en comprobar que funcionan bien los interfaces y que cumplen su cometido, para lo que pueden utilizarse las estrategias de caja negra que comentamos en apartados anteriores. De todo lo indicado, tal y como hemos podido ver en las animaciones, podemos deducir que las pruebas de unidades y las de integración están muy ligadas y pueden realizarse en paralelo según se vaya desarrollando el software.

Para saber más

Si queremos saber más sobre el software de ayuda para el desarrollo de pruebas podemos consultar esta página de la enciclopedia wikipedia:

JUnit

http://es.wikipedia.org/wiki/JUnit **[versión en cache]**

Pruebas del software

Comparación de las distintas estrategias

Hemos visto diferentes estrategias de integración de módulos, ¿qué **ventajas** aportan unas sobre las otras?

En este apartado vamos a **comparar** estas estrategias. La forma de integrar los componentes hasta desarrollar la aplicación completa para realizar las pruebas nos va a permitir centrarnos en la detección de unos tipos de error u otros. También hay que contemplar el **tiempo** consumido en las pruebas, ya que hay que diseñar componentes de pruebas, realizar pruebas, evaluar, etc, y dependiendo de la estrategia de integración podremos simultanear las pruebas de diferentes partes del programa o probar grupos de módulos mientras se desarrollan otros. Con todo ello, queremos decir que **el tipo de integración usada va a provocar una serie de ventajas o desventajas, que debemos conocer para seleccionar la más adecuada para nuestro desarrollo de un proyecto concreto**. Como todas las estrategias tienen sus ventajas y desventajas, no podremos decir que en general una es mejor que las otras, sino que dependiendo del tipo de proyecto y desarrollo necesario será más adecuada una u otra estrategia, de ahí la importancia de conocerlas todas en profundidad.

ESTRATEGIA	VENTAJAS	DESVENTAJAS
Ascendente	▪ Localizan de forma sencilla los errores en las interfaces de los módulos al sustituir sistemáticamente manejadores por módulos desarrollados. ▪ Es ventajoso si los errores están en los niveles inferiores. ▪ Los manejadores son	▪ El diseño de alto nivel, la arquitectura del sistema y la interfaz de usuario se comprueba al final de las pruebas. Si se detecta un error puede invalidar las pruebas realizadas e implicar la necesidad de realizar grandes cambios en el código de muchos módulos inferiores. ▪ La visión funcional

	sencillos de construir	global del sistema no llega hasta el final. ▪ Se necesitan manejadores de pruebas.
Descendente	▪ El diseño de alto nivel, la arquitectura del sistema y la interfaz de usuario se comprueba al inicio de las pruebas, por lo que los errores relacionados con los requerimientos se detectan rápido reduciendo el coste de grandes cambios. ▪ Se tiene un sistema funcional desde el principio aunque muy limitado. ▪ El mismo conjunto de pruebas del nivel superior puede usarse en niveles inferiores. ▪ Es ventajoso si los errores están en los niveles superiores.	▪ Se necesitan stubs que a veces son difíciles de construir. ▪ Es difícil observar los resultados de las pruebas. ▪ Provoca el retraso en la prueba de los módulos inferiores.
Combinada	▪ Se pueden probar desde el principio tanto los módulos superiores como los inferiores, es decir, tanto la arquitectura de sistema y la interfaz de usuario como las unidades finales. ▪ Es posible desarrollar pruebas en paralelo, por lo que, en general necesitan menos tiempo que las pruebas ascendentes y descendentes.	▪ Se necesitan manejadores y stubs
Big Bang	▪ Son fáciles de llevar a cabo y requieren menos tiempo.	▪ Es difícil detectar el motivo y módulo o agrupación de módulos causantes del error, ya que se integran todos los módulos a la vez. ▪ Es difícil distinguir entre fallos de interfaz y fallos de los

módulos.
- Se necesitan manejadores y stubs para hacer las pruebas individuales de los módulos.

Autoevaluación

La mejor estrategia para la integración de módulos en las pruebas es:

a) La descendente.
b) La ascendente.
c) La combinada.
d) Cualquiera de las anteriores, depende del tipo de sistema a desarrollar y la metodología de desarrollo utilizada.

Comprobar

Pruebas del software

En las explicaciones de pruebas unitarias y de integración no hemos distinguido entre unos **paradigmas de programación** y otros, pero **¿las pruebas se diseñan igual para la programación y diseño estructurado y funcional que para la programación y diseño orientado a objetos?**

Para resolver esta duda debemos conocer algunas peculiaridades del **desarrollo orientado a objetos** que afectan al diseño y aplicación de un plan de pruebas:

- La metodología de desarrollo orientada a objetos introduce una serie de conceptos y técnicas (**encapsulación, herencia, polimorfismo**...) que previenen cierto tipo de errores, pero que pueden provocar la aparición de otros, por lo que las pruebas deben centrarse en su búsqueda. Así, por ejemplo, la **herencia** obliga a probar un método y todas las implementaciones de sus clases hijas, y el **polimorfismo** a probar las diferentes implementaciones que se solapan. Esto afecta al tipo de casos de prueba a diseñar.
- Los objetos suelen ser **componentes mayores** que las funciones tradicionales, por lo que las pruebas unitarias y de integración tienden a mezclarse.
- La **reutilización de objetos** puede impedir el conocimiento del código fuente lo que dificultará las pruebas de caja blanca.
- La relación entre las diferentes clases no suele responder a **estructuras** jerárquicas, por lo que los enfoques ascendentes y descendentes para las

pruebas de integración no son apropiados.

- Los objetos incluyen nuevos elementos como **atributos, métodos o eventos** que deben probarse, esto implica que habrá que ampliar el concepto de clases de equivalencia y las pruebas de caja negra.

> ***Para saber más***
>
> ***Si deseas buscar más información sobre las características de la programación orientada a objetos (POO) puedes ir a la enciclopedia wikipedia:***
>
> ***Wikipedia: POO***
>
> ***http://es.wikipedia.org/wiki/POO*** **[versión en cache]**

Pruebas del software

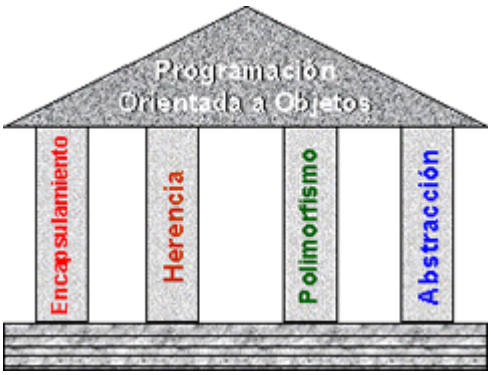
Descripción de las pruebas para POO

Hemos indicado que las características del desarrollo orientado a objetos implican diferencias en el diseño de las pruebas, ¿cómo serán?, ¿habrá muchos cambios con el modelo explicado antes?

Las pruebas para sistemas orientados a objetos deberán incluir estos cuatro **niveles**:

1. **Pruebas de operaciones individuales de los objetos**: incluiría la prueba de los algoritmos que implementan los distintos métodos de las clases y las técnicas a utilizar en las pruebas serían las ya expuestas de caja negra y caja blanca. Aspectos como el análisis de errores típicos, valores límite, condiciones, bucles, etc. son aplicables.

2. **Pruebas de clases de objetos individuales**:
   1. El concepto de **caja negra** para probar la **funcionalidad** no cambia, pero a la hora de diseñar las clases de equivalencia debemos extenderlas a las secuencias de operaciones relacionadas agrupándolas por los atributos utilizados.
   2. En cuanto a la prueba de la **estructura**, incluimos la **caja blanca,** cuya idea fundamental era recorrer todos los caminos independientes del código, pero con los objetos, habrá que extender esta idea para incluir los elementos propios de las clases:
      1. Pruebas de las **operaciones individuales** de los objetos (descritas en A).
      2. Asignación, evaluación y uso de todos los **atributos** de la clase.



3. Prueba de todos los **estados** posibles del objeto en función de los eventos asociados. En la animación que sigue podemos ver un ejemplo de aplicación.

La métrica de McCabe es aplicable al diseño de casos de prueba a partir del análisis de los diagramas UML.

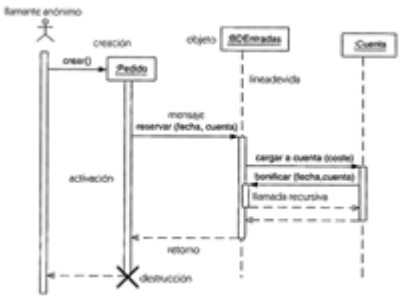
Implementación de un reloj digital mediante objetos

3. **Pruebas de integración:** como carece de sentido hablar de integración ascendente o descendente, se habla de la prueba de "*cluster*" de objetos. Un **cluster** sería un grupo de objetos que actúan combinados para proporcionar un conjunto de servicios relacionados. Durante las pruebas se van probando los diferentes clusters y posteriormente se combinan hasta completar el sistema. La formación y combinación de los clusters puede realizarse según distintas estrategias, aunque la más utilizada es la de **pruebas de casos de uso o basadas en escenarios:** los casos de uso describen un modo de utilización del sistema, los cluster estarán formados por los objetos que intervienen en un determinado caso de uso. La ventaja de este método es que es fácil identificar los escenarios más utilizados para probarlos antes y más intensamente, (lo describimos también en el ejemplo del apartado siguiente). Para completar la información de cara a diseñar las pruebas recurrimos al **diagrama UML de secuencia** como el de la figura inferior.

Por **ejemplo**, si nos fijamos en el diagrama podremos ver que para "hacer un cargo a una cuenta" tras un "pedido de un cliente" es necesaria la siguiente **cadena de métodos** de los distintos objetos:

Pedido:crear() → BDEntradas:reservar → Cuenta:cargar_a_cuenta

Por ello, estos objetos quedarán asociados en un cluster y habrá que crear un caso de prueba para esta cadena de métodos. Obviamente, no es posible probar todas las combinaciones de métodos, pero es importante asegurar que cada método de cada clase se ejecuta al menos una vez. Además, deben incluirse las pruebas de las excepciones en caso de que existiesen.



Otras **estrategias** que podemos usar para formar los clusters es utilizar las distintas **secuencias de eventos** que llegan a un sistema y ver los objetos implicados. O también, a partir de la **interacción entre los objetos** mediante el intercambio de mensajes podemos agrupar objetos. Para todo ello, nos basaremos en el análisis de los correspondientes diagramas UML.

4. **Pruebas del sistema**: la verificación y validación del sistema a partir de la especificación de requerimientos se realizará igual que en otro tipo de desarrollos.

Podemos ayudarnos de sistemas CASE y entornos de desarrollo de pruebas para sistematizar, diseñar y realizar todas estas pruebas.

Para saber más

JUnit es un conjunto de librerías que son utilizadas para desarrollar componentes y hacer pruebas de aplicaciones Java. Antes ya pudimos ver en la Wikipedia su descripción, en su página web oficial podremos conocer con más profundidad este software:

Web Oficial de JUnit

http://www.junit.org/index.htm

Si queremos conocer más sobre la técnica de modelado UML podemos consultar la wikipedia en esta página:

Wikipedia: UML

http://es.wikipedia.org/wiki/Uml **[versión en cache]**

Autoevaluación

En las pruebas estructurales de caja blanca aplicadas a una clase de objetos debemos cubrir:

a) El recorrido de todos los caminos independientes de los métodos del objeto
b) El uso y acceso a todos los atributos del objeto
c) Todos los posibles estados del objeto
d) Todos los elementos anteriores

Comprobar

Pruebas del software

*En **Si Andalucía** piensan que la aplicación de **SinUnEuro** está prácticamente terminada. De hecho, han efectuado todas las pruebas unitarias y de integración, y todo parece funcionar bien. Pero **José** dice que antes de entregar el producto, deben realizarse las pruebas del sistema y las pruebas de aceptación.*

*Para eso, **Víctor** elabora un plan de pruebas que somete a todo el sistema a una **sobrecarga**, para medir cómo afecta al rendimiento de la aplicación. En ese sentido, crea un fichero de clientes con varios cientos de miles de clientes y de ofertas, simulando 10 veces más comerciales vendiendo casas simultáneamente de los que realmente tiene la*

empresa, y midiendo el tiempo de respuesta de la aplicación a cada comercial o cada cliente.

***Carmen** se encarga de elaborar un plan de pruebas de recuperación, en las que hace que falle la red, desconecta algún periféfico, etc. para comprobar cómo reacciona la aplicación, si se bloquea, si produce datos erróneos, o si se recupera de esos errores de forma adecuada, sin estropear nada, y dándole información al usuario. Por ejemplo, corta internet cuando un comercial está en medio de una operación de venta de un inmueble, y comprueba que al comercial se le indica con un mensaje adecuado que la operación no se ha podido realizar y las causas, al mismo tiempo que comprueba que efectivamente, toda la información queda restaurada al estado anterior al inicio de la venta frustrada.*

***Jesús**, el experto en sistemas de la empresa, intenta acceder sin tener clave, ni permisos suficientes para ello, a los ficheros de la empresa, y comprueba que los ataques exteriores que intenta hacer están bien protegidos por una política correcta de permisos y contraseñas. De esta manera ha probado la seguridad del sistema.*

*Como todo ha ido bien, han decidido hacer las pruebas definitivas: Han instalado la aplicación en los ordenadores de la empresa **SinUnEuro**, lo han configurado todo de acuerdo a la red de la empresa, y han vuelto a comprobar que todo funciona correctametne.*

*Ya sólo faltan las **pruebas de aceptación**, que consiste en dejar que los empleados y el propio gerente de **SinUnEuro** prueben la aplicación, y evalúen si realmente satisface sus necesidades, es fiable y fácil de usar, la documentación es clara, dispone de una eficaz ayuda en línea, etc. También han estado usando el sistema un tiempo en paralelo con el antiguo, para compararlo y ver las mejoras que aporta el nuevo sistema. El **resultado final** es que la empresa queda muy satisfecha, por lo que dan por finalizadas las pruebas por el momento, hasta que surjan nuevas necesidades, o hasta que se detecte alguna posible mejoría del programa.*

Hemos indicado que las **pruebas** deben cubrir todo el ciclo de desarrollo del software para asegurar su calidad y el cumplimiento de los requisitos, y hemos estudiado las pruebas sobre el diseño y la implementación del código, pero nos falta por ver las pruebas sobre el sistema global.

¿Cómo serán estas pruebas?

Las técnicas que hemos comentado anteriormente de **caja negra, caja blanca** y las **revisiones** son igualmente válidas para estos niveles de pruebas pero deben adaptarse a las necesidades de los productos a probar. A continuación describiremos estas pruebas.

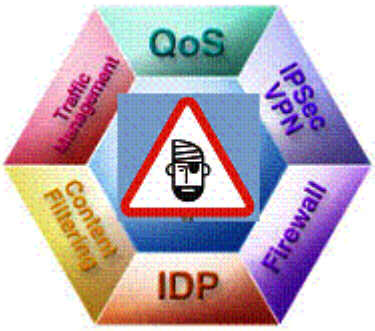
Pruebas del software

Pruebas del sistema

El objetivo de las pruebas del sistema y validación es asegurar que el sistema completo (software y hardware) cumple con los requisitos especificados inicialmente. Incluyen, por tanto, las pruebas de **verificación** y **validación** del sistema a partir de los requerimientos funcionales y no funcionales, y la adecuación de la documentación de usuario.

Algunos autores incluyen sólo a las actividades de verificación dentro de las pruebas del sistema, y las actividades de validación quedan incluidas en las pruebas específicas de validación, pero en realidad, todas estas actividades están muy ligadas, por lo que otros autores las denominan en su conjunto como pruebas del sistema, incluyendo actividades de verificación y validación. Dentro de estas pruebas podemos distinguir las siguientes actividades a desarrollar:

- **Pruebas funcionales:** se utilizan técnicas de caja negra en las que se crean casos de prueba basándose en los casos de uso fijados en los requisitos del sistema. Se trata de comprobar que los requisitos funcionales y no funcionales quedan satisfechos por la aplicación desarrollada. Se aplican las mismas estrategias de caja negra ya vistas, es decir, clases de equivalencia, casos de uso comunes, situaciones límite y excepcionales, etc. También pueden utilizarse estrategias de caja blanca para analizar diagramas UML o DFD (Diagramas de Flujo de Datos) donde se puede utilizar la métrica de McCabe para determinar el número de casos a estudiar.
- **Pruebas de rendimiento:** el sistema se somete a sobrecarga de trabajo como gran volumen de datos, exceso de usuarios, reducción de recursos, etc. provocando situaciones límite que están por encima de la carga normal prevista de trabajo y que nos asegura su fiabilidad y eficacia.



- **Pruebas de recuperación:** consiste en provocar fallos en el sistema (errores en la red, fallo de hardware, etc) y ver cómo reacciona y se recupera ante ellos.
- **Pruebas de seguridad:** se trata de encontrar fallos de seguridad en el sistema. En algunos casos se somete el sistema al ataque de experimentados programadores, hackers y crackers para que descubran posibles agujeros de seguridad.
- **Pruebas de instalación:** una prueba completa del sistema debe realizarse en el entorno real en el que se va a utilizar, para lo cual es necesario instalar el sistema en los equipos de la empresa y realizar allí nuevas pruebas.
- **Pruebas de configuración:** hay que prever los posibles cambios en la configuración del sistema software y hardware, como cambios en las redes, ordenadores, etc. y para ello hay que realizar las pruebas e inspecciones necesarias.
- **Prueba y validación de la documentación de usuario:** es importante que la

documentación sea útil y esté actualizada para lo que pueden usarse técnicas de auditoría.

Autoevaluación

¿Qué actividades se incluyen dentro de las pruebas del sistema?:

a) Pruebas funcionales
b) Pruebas de instalación y configuración
c) Pruebas de rendimiento y recuperación
d) Todas las anteriores forman parte de las pruebas del sistema

Comprobar

Pruebas del software

Pruebas de aceptación

En las **pruebas de aceptación** el cliente evalúa el sistema sobre todo en los aspectos de fiabilidad, facilidad de uso y requisitos previstos para decidir si el sistema responde a sus necesidades y expectativas y es aceptado. A veces, cuando se producen **errores** en un sistema instalado los desarrolladores se preguntan ¿pero cómo es posible que hayan usado así el programa?, y pueden culpar del error al mal uso de la aplicación por parte del usuario. Pero estas situaciones debían haber estado previstas, de ahí la importancia de las pruebas de aceptación. Se realizan en el entorno real del cliente, empresa, trabajadores, equipos, etc. Pueden incluir diferentes actividades:

- **Prueba de usuario:** Se realiza con un plan de prueba que pretende demostrar que se cumplen los requisitos. La ejecución la realiza el usuario ayudado por profesionales participantes en el proyecto.
- **Prueba de comparación:** Si el nuevo sistema sustituye a otro antiguo, se ejecutan ambos sistemas en paralelo para comparar resultados, rendimiento, etc. y demostrar las mejoras ofrecidas por el nuevo.
- **Pruebas alfa:** En ellas se invita a los clientes a probar los productos en el entorno de desarrollo. Aunque los programas no están totalmente terminados ya ofrecen una funcionalidad básica suficiente para realizar algunas pruebas que ayuden al desarrollo y a descubrir defectos en el proyecto. Los usuarios realizan un informe indicando los errores detectados o las mejoras propuestas.
- **Pruebas beta:** se realizan en el entorno del cliente, y en ellas el usuario prueba productos que ya son una versión completa de la aplicación final aunque está en fase de prueba. A esta versión de los productos o programas en fase de pruebas se les suele llamar también programas beta o inestables (frente a las versiones finales que se les llama estables).

Este tipo de pruebas se usan mucho en el

desarrollo de software libre o de código abierto
donde se ponen a disposición de todo el mundo, a través de Internet, las versiones beta de las aplicaciones. Los probadores informan a los desarrolladores de los errores que detectan. La última versión completa del producto preparada para el lanzamiento, a menos que se detecte un error de última hora, se suele denominar **candidata** definitiva (release candidate). En el software comercial también se utilizan pruebas beta, pero normalmente no se distribuyen libremente las versiones beta sino que se entregan a probadores seleccionados. Esta estrategia es muy eficaz para detectar fallos y se utiliza mucho en las distribuciones de software para un mercado amplio de muchos clientes, es decir, cuando no se trata de desarrollos de software por encargo.

Todos los sistemas sufren una evolución a lo largo de su vida activa. En cada nueva versión se supone que o bien se corrigen defectos, o se añaden nuevas funciones, o ambas cosas. Es habitual que los fabricantes de software preparen paquetes de actualización del software, a veces se llaman informalmente "parches".

En cualquier caso, una nueva versión exige nuevas actividades de pruebas que se denominan **Pruebas de Regresión**. Si las pruebas se han sistematizado en las fases anteriores, ahora pueden volver a realizarse automáticamente en la prueba de regresión, simplemente para comprobar que las modificaciones no provocan errores donde antes no los había.

Autoevaluación

Las características de una prueba alfa son:

a) Se utilizan en las pruebas de sistema por parte de los desarrolladores y el equipo de pruebas
b) Se realizan en el entorno del cliente
c) El producto probado es una versión completa
d) Ninguna de las anteriores características corresponde a las pruebas alfa

Comprobar

Pruebas del software

Al igual que ocurre con el resto de productos del ciclo de vida de desarrollo del software, las pruebas como parte integrante del mismo, deben ser perfectamente documentadas.

¿Qué documentos componen dicha documentación?

Algunos contenidos ya los hemos comentado en los apartados previos, plan de pruebas, casos de pruebas, informes, etc. A continuación hacemos una clasificación de estos documentos:

- **Plan de pruebas**: incluye el alcance, objetivos, recursos y calendario de las

actividades de pruebas, las personas responsables, y las actividades concretas a llevar a cabo.

- **Especificación de los casos de pruebas**: cada prueba debe documentarse con un documento de especificación que incluirá la descripción de la prueba, tareas a realizar, procedimiento a seguir, entorno de la prueba, necesidades y recursos para realizar la prueba, datos y eventos de entrada, datos y comportamientos del sistema esperados a la salida, componentes, manejadores y stubs de prueba necesarios.
- **Plan de revisiones, reuniones e inspecciones**: incluye la preparación de estas revisiones y los documentos resultantes.
- **Informes del desarrollo de las pruebas**: cada ejecución de cada prueba debe documentarse para recoger los posibles incidentes que se hayan podido producir.
- **Informe de resumen de las pruebas**: en este documento se detalla un listado de todos los fallos detectados en las pruebas y que deben investigarse. A partir de este documento los desarrolladores analizan cada error y plantean acciones a llevar a cabo para solucionarlos. Al proceso de localizar, analizar y corregir defectos se le denomina depuración y es el resultado de una prueba exitosa que ha encontrado errores. A la hora de depurar errores es importante tratarlos individualmente, ya que unos pueden ser consecuencia de otros. Estos cambios en el sistema provocarán la realización de nuevas pruebas de comprobación.

Autoevaluación

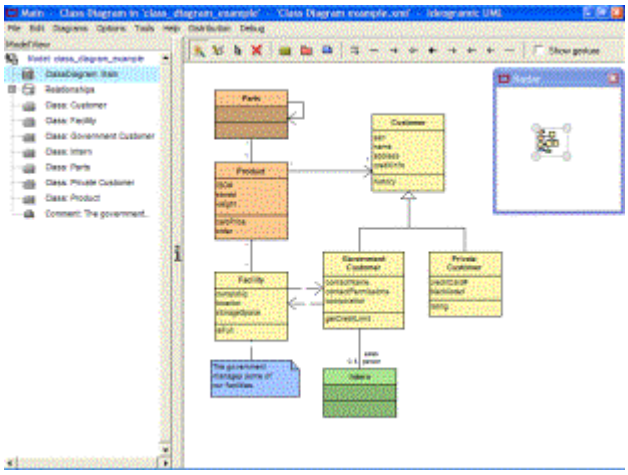
Los stubs deben describirse en el documento de pruebas siguiente:

a) Plan de pruebas
b) Especificación de los casos de prueba
c) Informes de desarrollo de las pruebas
d) Informe de resumen de las pruebas

Comprobar

Pruebas del software

A lo largo del estudio del ciclo de vida del desarrollo del software hemos hablado de la necesidad de ayudarnos de **herramientas CASE** que automaticen, gestionen y faciliten el trabajo. En esta fase del ciclo de vida de pruebas también se ha comentado la existencia y utilidad de estas herramientas.



Pero **no debemos pensar en herramientas específicamente preparadas para el desarrollo de las pruebas, sino que las aplicaciones CASE generales nos suministran herramientas y servicios que pueden utilizarse en las pruebas**.

Además, hay herramientas específicamente diseñadas para la realización de pruebas denominadas **CAST** (Computer Aided Software Testing). A continuación vamos a describir brevemente algunas de estas herramientas:

- **Administrador y gestor de pruebas**: se utilizan para planificar las pruebas, predecir su coste y temporización, monitorizar las pruebas, generar informes, etc.
- **Diseñador de pruebas**: permiten diseñar, generar y documentar los casos de pruebas. Incluyen generadores de datos para las pruebas, generadores de predicciones de resultados esperados, generadores de componentes y elementos de pruebas (manejadores, stubs, etc)
- **Asistente de pruebas**: permiten automatizar la ejecución, seguimiento y registro de las pruebas. Además, nos proporcionan estadística del desarrollo de las pruebas y sus resultados.
- **Simuladores**: proporcionan entornos en los que realizar las pruebas simulando el comportamiento de sistemas software y hardware, esto nos permite realizar pruebas en entornos similares a los reales del cliente aunque no los dispongamos en nuestra empresa. Las simulaciones pueden ir desde la simulación de una interfase de usuario hasta la simulación de una red de ordenadores.
- **Analizadores**: son sistemas utilizados para ayudar en la localización de errores y búsqueda de soluciones, un ejemplo de este tipo de herramienta son los depuradores o debuggers.

Para saber más

Las herramientas software que nos ayudan a desarrollar las pruebas son muy importantes. En esta web podemos ver el paquete software AGEDIS que permite la automatización de la generación y ejecución de pruebas:

AGEDIS software

http://www.agedis.de/

Pruebas del software