TÉCNICAS DE EVALUACIÓN DE SOFTWARE

Versión: 12.0

Fecha: 17 de octubre de 2006

Autoras: Natalia Juristo, Ana M. Moreno, Sira Vegas

TABLA DE CONTENIDO

1. INTRODUCCIÓN A LA EVALUACIÓN DE SOFTWARE

1.1 ¿ESTAMOS CONSTRUYENDO SOFTWARE SIN DEFECTOS?: ESTADO ACTUAL DE LA PRÁCTICA

Todos, alguna vez, hemos sufrido algún error informático, ya sea una factura indebidamente cargada o la destrucción del trabajo de todo un día, por culpa de un fallo misterioso en el software. Tales problemas nacen de la complejidad del software. La extrema dificultad para construir sistemas software multiplica la probabilidad de que persistan errores aún después de haberse finalizado y entregado el sistema, manifestándose cuando éste es utilizado por el cliente.

La construcción de un sistema software tiene como objetivo satisfacer una necesidad planteada por un cliente. ¿Cómo puede saberse si el producto construido se corresponde exactamente con lo que el cliente deseaba? y ¿Cómo se puede estar seguro de que el producto que ha construido va a funcionar correctamente?

Desgraciadamente, nuestra capacidad para medir la fiabilidad del software es muy inferior a lo que sería necesario[1]. Sería deseable que los informáticos pudieran demostrar matemáticamente la corrección de sus programas, al estilo de los otros ingenieros. Los otros ingenieros recurren a análisis matemáticos para predecir cuál será el comportamiento de sus creaciones en el mundo real. Esa predicción permite descubrir defectos antes de que el producto esté operativo. Por desdicha, las matemáticas tradicionales, aptas para la descripción de sistemas físicos (los tipos de sistemas tratados por las otras ingenierías), no son aplicables al universo sintético binario de un programa de ordenador. Es la matemática discreta, una especialidad mucho menos madura, y casi no estudiada hasta la aparición de las computadoras, la que gobierna el campo de los sistemas software.

Dada la imposibilidad de aplicar métodos matemáticos rigurosos, el modo que tienen los informáticos para respaldar la confianza de los programas es la verificación empírica. La fiabilidad de los programas irá creciendo a lo largo de este proceso. Se hacen funcionar los programas, observando directamente su comportamiento y depurándolos cada vez que aparece una deficiencia una vez el sistema a construir ha sido terminado. Sin embargo, este modo de actuar no proporciona una solución definitiva debida, principalmente, a dos razones:

1. Si descubrimos en el código errores muy graves que afectan a productos anteriores (requisitos, diseño,…) debemos volver atrás en el desarrollo. Sin embargo, estamos ya al final del proyecto (en la etapa de codificación), ya se ha gastado casi la totalidad del tiempo y del presupuesto. ¿Qué hacer? ¿Entregamos tarde el sistema y repetimos el desarrollo? ¿Le pedimos al cliente un aumento del presupuesto?

2. Por otra parte, la comprobación que empírica no sirve para garantizar que no hay errores en el software, puesto que ello depende, por un lado, de la porción del programa que se esté ejecutando en el momento de esta comprobación, y por otro, de las entradas que se le hayan proporcionado al código. Por lo tanto, pueden existir errores en otras partes del programa que no se ejecuten en ese momento o con otras entradas que no se hayan usado en la prueba.

Por lo tanto, lo recomendable es que producto software vaya siendo evaluado a medida que se va construyendo. Como veremos posteriormente, se hace necesario llevar cabo, en paralelo al proceso de desarrollo, un proceso de evaluación o comprobación de los distintos productos o modelos que se van generando, en el que participarán desarrolladores y clientes.

[1] W. Wayt Gibbs. Software's Chronic Crisis. *Scientific American*. Number 218. November 1994.

No obstante, la calidad que se puede obtener mediante este procedimiento artesanal es bastante baja. De ahí que, a pesar de haber sido ensayados rigurosa y sistemáticamente, la mayoría de los programas grandes contengan todavía defectos cuando son entregados. Ello se debe a la complejidad del software. Un programa de apenas unos centenares de líneas de código puede contener decenas de decisiones, lo que permite millares de rutas de ejecución alternativas, resultando materialmente imposible el ensayo exhaustivo de todas las posibles rutas alternativas. Para alcanzar una confianza de no más de 10^{-9} fallos por hora tendría que ejecutarse un programa durante muchísimos múltiplos de 10^9 horas, esto es, durante muchos múltiplos de 100.000 años[2].

Así pues, la ambición de conseguir programas perfectos sigue siendo una cima inaccesible. Existe, hoy por hoy, la imposibilidad práctica de conseguir software totalmente libre de defectos[2] y, debemos, por tanto, aceptar las actuales limitaciones que padece la construcción de sistemas software. De hecho, algunos autores[3] sugieren que, dada la entidad no física del software, los defectos en los programas son inherentes a su naturaleza.

Es difícil establecer cuál es la cantidad media de defectos que un sistema software "normal" contiene. Hay estimaciones, como la del Software Engineering Institute[4], que dicen que un programador experto introduce un defecto por cada 10 líneas de código; suponiendo que se detectasen el 99% de los defectos introducidos (lo cual resulta tremendamente optimista) aún permanecerían 1 defecto por cada 1.000 líneas de código (KLOC[5]).

No obstante, la depuración de los sistemas software obedece a la ley del rendimiento decreciente. Esto es, según se avanza en el proceso de búsqueda de defectos, el coste de detección de fallos y eliminación de las faltas que los provocan empieza a rebasar con mucho las mejoras conseguidas en la fiabilidad del sistema. Nótese que la fiabilidad de un software no se mide como la cantidad de faltas que quedan en un programa, sino como el tiempo medio entre fallos[6]. Así pues, el objetivo de las técnicas de evaluación del software no es tanto la eliminación total de las faltas existentes en los programas, como la eliminación de las faltas que provocan fallos frecuentes. Si se persevera durante muchísimo tiempo en la depuración de un software, acabamos descubriendo faltas que producirán fallos tan infrecuentes que su enmienda no incide en la fiabilidad percibida del sistema.

De hecho, hasta el software más depurado y considerado de alta fiabilidad contiene defectos remanentes. Edward N. Adams de IBM analizó empíricamente[7] los "tamaños" de las faltas en una base de datos de cobertura mundial que suponía el equivalente de miles de años de uso de un sistema informático particular. El descubrimiento más extraordinario consistió en que alrededor de la tercera parte de las faltas contenidas en un programa son quinquemilenarias. Esto es, faltas que producirían un fallo tan sólo una vez cada 5.000 años. Estas faltas excepcionales sumaban una porción considerable del total de faltas remanentes, pues las faltas responsables de de fallos más frecuentes habían sido descubiertas y consiguientemente eliminadas durante la fase de evaluación y en los primeros meses de operación del sistema. Obviamente, emplear tiempo en detectar faltas que producen fallos cada más allá de 75 años es malgastar recursos.

---

[2] Bev Littlewood and Lorenzo Strigini. The Risks of Software. *Scientific American.* Volume 268, Number 1, January, 1993

[3] Y. Huang, P. Jalote and C. Kintala. Two Techniques for Transient Software Error Recovery. *Lecture Notes in Computer Science*, Vol. 774, pages 159-170. Springer Verlag, Berlin, 1994.

[4] Software Engineering Institute. Information Week, Jan. 21, 2002

[5] KLOC es el acrónimo inglés de Kilo Lines Of Code, esto es, 1.000 líneas de código.

[6] Una *falta* en el código es la causante de un *fallo* en el funcionamiento del sistema. Los *fallos* se perciben como errores por los usuarios del sistema. Las *faltas*, mientras no se manifiestan como fallo durante el funcionamiento del sistema, no pueden ser apreciadas por los usuarios. El término *defecto* se utiliza cuando no es necesario la exactitud de diferencias entre falta y fallo.

[7] E. Adams. *Optimizing Preventive Service of Software Products*. IBM Research J., vol. 28, no. 1, pages. 2-14, 1984.

Si hablamos de valores reales, en lugar de estimaciones, unos buenos ejemplos de cuántos fallos son "normales" en un software pueden serlo los sistemas operativos. La tasa de defectos de Linux es 0,1 defectos/KLOC[8]. Las distintas versiones de Unix tienen entorno a 0,6-0,7 defectos/KLOC[4]. Los sistemas operativos con interfaz gráfico como Windows 95 o MacOS poseían una tasa de defectos/KLOC tan elevada que fallaban cada tres horas, o incluso menos, de funcionamiento ininterrumpido[9]. Otro ejemplo, Siemens sufrió una tasa de 6-15 defectos /KLOC en el desarrollo de alguno de sus sistemas operativos[10].

Fuera del ámbito de los sistemas operativos existen pocas, aunque elocuentes, referencias a las tasas de defectos del software. Así, Unisys alcanzó una tasa de 2-9 defectos/KLOC en el desarrollo de software de comunicaciones. IBM, en el desarrollo normal de software, puede llegar a sufrir una tasa de 30 defectos/KLOC. Como se ve la variabilidad entre unos casos y otros es muy alta, lo que impide sacar reglas sobre qué es "normal"

Incluso utilizado técnicas muy avanzadas (las que se usan en sistemas de alto riesgo como la conocida *Cleanroom Development*[11]) es imposible lograr un software totalmente libre de defectos. Así, por ejemplo, la misma IBM no consiguió rebajar su tasa de defectos de 2,3-3,4 defectos/KLOC utilizando la técnica *Cleanroom*[12].

La *UK Civil Aviation* obtuvo una tasa de defectos de 0,81 / KLOC en el desarrollo del sistema de control de tráfico aéreo del Reino Unido. Nótese que, en este caso, la necesidad de fiabilidad y robustez del sistema es enorme y, sin embargo, casi se alcanzó 1 defecto/KLOC, lo que parece indicar que 1 puede considerarse el límite inferior de la tasa de defectos alcanzable. Sin embargo, en otros casos de sistemas software también críticos y que requieren programas especialmente fiables, se ha superado con creces este límite. Así, según los datos publicados hasta la fecha, la NASA[13] ha sufrido tasas de defectos en el rango 4-12 defectos /KLOC.

Cabe preguntarse, en consecuencia, qué tasa de defectos debe considerarse normal en un proyecto de desarrollo. Existen autores que afirman que es habitual encontrar en el software comercial entre 25-30 defectos/KLOC[14]. No obstante, aplicando una visión exigente tenemos que:

- Lo mejor que se puede conseguir es 0,5-1 defectos/KLOC
- En un software comercial, es esperable encontrar entre 3-6 defectos/KLOC

---

[8] Stephen Shankland CNET News.com February 19, 2003

[9] L. Hatton. Keynote presentation, COMPASS'97, 16-19 June, 1997. http://guinness.cs.stevens-tech.edu/~lbernste/presentations/Defects_ala_Hatton.ppt

[10] L. Hatton. Programming Languages and Safety-Related Systems. *Proc. Safety-Critical Systems Symp.*, Springer-Verlag, New York, 1995, pp. 48-64, citado en S. L. Pfleeger and L. Hatton. Investigating the Influence of Formal Methods. *IEEE Computer*, Volume 30, Issue 2 (February 1997), pp. 33-43.

[11] R. C. Linger. Cleanroom Process Model. *IEEE Software*, Volume 11, issue 2 (March 1994), pp 50-58.

[12] *Cleanroom Development* es una de las más depuradas, avanzadas y contrastada para desarrollar sistemas software con baja tasa de defectos. La razón de su no amplia utilización es su altísimo coste, que hace dispararse los costos de los proyectos. Así pues, únicamente se aplica cuando el cliente lo exige y acepta el aumento que produce en el precio del desarrollo.

[13] La NASA es considerado uno de los centros de desarrollo de software más avanzado a nivel mundial. Durante años sus investigaciones en el *Software Engineering Laboratory*, en Maryland EE.UU, han marcado tendencias en la construcción de software. La razón para esto es clara: los fallos en los vehículos de la NASA son irreversibles, pues no hay opción para que un desarrollador se acerque al sistema y resuelva el problema. En otras palabras, no puede permitirse el lujo de tener fallos.

[14] M. Dyer. *The Cleanroom Approach to Software Quality*. John Wiley & Sons, New York, 1992.

- Puede considerarse que un software posee alta calidad cuando su tasa de defectos es menor de 15 defectos/KLOC.

1.2 CONTROL DE CALIDAD DEL SOFTWARE

El interés por la calidad crece de forma continua, a medida que los clientes se vuelven más selectivos y comienzan a rechazar los productos poco fiables o que realmente no dan respuesta a sus necesidades.

Como primera aproximación es importante diferenciar entre la calidad del PRODUCTO software y la calidad del PROCESO de desarrollo. Las metas que se establezcan para la calidad del producto van a determinar las metas a establecer para la calidad del proceso de desarrollo, ya que la calidad del producto va a estar en función de la calidad del proceso de desarrollo. Sin un buen proceso de desarrollo es casi imposible obtener un buen producto.

También es importante destacar que la calidad de un producto software debe ser considerada en todos sus estados de evolución a medida que avanza el desarrollo de acuerdo al ciclo de vida seleccionado para su construcción (especificaciones, diseño, código, etc.). No basta con tener en cuenta la calidad del producto una vez finalizado, cuando los problemas de mala calidad ya no tienen solución o la solución es muy costosa.

Los principales problemas a los que se enfrenta el desarrollo de software a la hora de tratar la calidad de un producto software son la definición de calidad y su comprobación:

Con respecto a la definición de la calidad del software: ¿Es realmente posible encontrar un conjunto de propiedades en un producto software que nos den una indicación de su calidad? Para dar respuesta a estas preguntas aparecen los Modelos de Calidad. En los Modelos de Calidad, la misma se define de forma jerárquica. Resuelven la complejidad mediante la descomposición. La calidad es un concepto que se deriva de un conjunto de sub-conceptos.

En el caso de la calidad del software, el término es difícil de definir. Con el fin de concretizar a qué nos referimos con calidad de un sistema software, se subdivide en atributos:

- Funcionalidad – Habilidad del software para realizar el trabajo deseado.
- Fiabilidad – Habilidad del software para mantenerse operativo (funcionando).
- Eficiencia – Habilidad del software para responder a una petición de usuario con la velocidad apropiada.
- Usabilidad – Habilidad del software para satisfacer al usuario.
- Mantenibilidad – Habilidad del software para poder realizar cambios en él fácilmente y con una adecuada proporción cambio/costo.
- Portabilidad – Habilidad del software para operar en diferentes entornos informáticos.

A su vez, cada una de estas características del software puede subdividirse en atributos aún más concretos. La Tabla 1 muestra una posible subdivisión. Aunque existen muchas otras descomposiciones de la calidad del software, ésta es una de las más aceptadas.

CHARACTERISTICS AND SUBCHARACTERISTICS		DESCRIPTION
Functionality		**Characteristics relating to achievement of the basic purpose for which the software is being engineered**
	Suitability	The presence and appropriateness of a set of functions for specified tasks
	Accuracy	The provision of right or agreed results or effects
	Interoperability	Software's ability to interact with specified systems
	Security	Ability to prevent unauthorized access, whether accidental or deliberate, to programs and data
	Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols.
Reliability		**Characteristics relating to capability of software to maintain its level of performance under stated conditions for a stated period of time**
	Maturity	Attributes of software that bear on the frequency of failure by faults in software
	Fault tolerance	Ability to maintain a specified level of performance in cases of software faults or unexpected inputs
	Recoverabiltiy	Capability and effort needed to re-establish level of performance and recover affected data after possible failure
	Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols
Usability		**Characteristics relating to the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users**
	Understandability	The effort required for a user to recognize the logical concept and its applicability
	Learnability	The effort required for a user to learn its application, operation, input and output
	Operability	The ease of operation and control by users
	Attractiveness	The capability of the software to be attractive to the user
	Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols
Efficiency		**Characteristics related to the relationship between the level of performance of the software and the amount of resources used, under stated conditions**
	Time behavior	The speed of response and processing times and throughput rates in performing its function
	Resource utilization	The amount of resources used and the duration of such use in performing its function
	Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols
Maintainability		**Characteristics related to the effort needed to make modifications, including corrections, improvements or adaptation of software to changes in environment, requirements and functional specifications**
	Analyzability	The effort needed for diagnosis of deficiencies or causes of failures, or for identification parts to be modified
	Changeability	The effort needed for modification fault removal or for environmental change
	Stability	The risk of unexpected effect of modifications
	Testability	The effort needed for validating the modified software
	Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols
Portability		**Characteristics related to the ability to transfer the software from one organization or hardware or software environment to naother**
	Adaptability	The opportunity for its adaptation to different specified environments
	Installability	The effort needed to install the software in a specified environment
	Co-existence	The capability of a software product to co-exist with other independent software in common environment
	Replaceability	The opportunity and effort of using it in the place of other software in a particular environment
	Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols

Tabla 1. Descomposición de la calidad del software por ISO 9126-1998

Independientemente de la descomposición de calidad que se elija, el nivel de propensión de faltas de un sistema software afecta siempre a varios de los atributos de calidad. En particular, fiabilidad y funcionalidad son siempre los más afectados. No obstante, no existe una relación bien establecida entre las faltas y la fiabilidad y funcionalidad. O dicho de otro modo, entre las faltas y los fallos. Todas las faltas de un producto software no se manifiestan como fallos. Las faltas se convierten en fallos cuando el usuario de un sistema software nota un comportamiento erróneo. Para que un sistema software alcance un nivel alto de calidad se requiere que el número de *fallos* sea bajo. Pero para mantener los fallos a niveles mínimos, las faltas necesariamente deben también estar en niveles mínimos.

Resumiendo, la calidad es difícil de definirse. Para facilitar su comprensión la calidad se ha descompuesto en atributos. Controlar y corregir las faltas existentes en un producto software afecta positivamente algunos atributos de calidad. En particular, si se trabaja en detectar y eliminar las faltas y los fallos, la funcionalidad y la fiabilidad mejoran.

En términos generales, se pueden distinguir dos tipos de evaluaciones durante el proceso de desarrollo: Verificaciones y Validaciones. Según el IEEE Std 729-1983 éstas se definen como:

- *Verificación*: Proceso de determinar si los productos de una cierta fase del desarrollo de software cumplen o no los requisitos establecidos durante la fase anterior.

- *Validación*: Proceso de evaluación del software al final del proceso de desarrollo para asegurar el cumplimiento de las necesidades del cliente.

Así, la verificación ayuda a comprobar si se ha construido el producto correctamente, mientras que la validación ayuda a comprobar si se ha construido el producto correcto. En otras palabras, la verificación tiene que ver típicamente con errores en la transformación entre productos (de los requisitos de diseño, del diseño al código, etc.). Mientras que la validación tiene que ver con errores al malinterpretar las necesidades del cliente. Así la única persona que puede validar el software, ya sea durante su desarrollo con una vez finalizado, es el cliente, ya que será quién pueda detectar si se interpretaron adecuadamente.

La calidad siempre va a depender de los requisitos o necesidades que se desee satisfacer. Por eso, la evaluación de la calidad de un producto siempre va a implicar una comparación entre unos requisitos preestablecidos y el producto realmente desarrollado.

El problema es que, por lo general, una parte de los requisitos van a estar explícitos (se encontrarán en la ERS - Especificación de Requisitos Software, tanto los funcionales como otros requisitos), pero otra parte van a quedar implícitos (el usuario sabe lo que quiere, pero no siempre es capaz de expresarlo). Hay que intentar que queden implícitos la menor cantidad de requisitos posible. No se podrá conseguir un producto de buena calidad sin una buena ERS.

Teniendo esto en cuenta, en un producto software vamos a tener diferentes visiones de la calidad:

- Necesaria o Requerida: La que quiere el cliente.

- Programada o Especificada: La que se ha especificado explícitamente y se intenta conseguir.

- Realizada: La que se ha conseguido.

Nuestro objetivo es conseguir que las tres visiones coincidan. A la intersección entre la calidad Requerida y la calidad Realizada se le llama calidad Percibida, y es la única que el cliente valora. Toda aquella calidad que se realiza pero no se necesita es un gasto inútil de tiempo y dinero.

Tanto para la realización de verificaciones como de validaciones se pueden utilizar distintos tipos de técnicas. En general, estas técnicas se agrupan en dos categorías:

- *Técnicas de Evaluación Estáticas*: Buscan **faltas** sobre el sistema en reposo. Esto es, estudian los distintos modelos que componen el sistema software buscando posibles faltas en los mismos. Así pues, estas técnicas se pueden aplicar, tanto a requisitos como a modelos de análisis, diseño y código.

- *Técnicas de Evaluación Dinámicas*: Generan entradas al sistema con el objetivo de detectar **fallos**, cuando el sistema ejecuta dichas entradas. Los fallos se observan cuando se detectan incongruencias entre la salida esperada y la salida real. La aplicación de técnicas dinámicas es también conocida como *pruebas* de software o *testing* y se aplican generalmente sobre código puesto que es, hoy por hoy, el único producto ejecutable del desarrollo.

Veamos en la siguiente sección cómo estas técnicas de evaluación han de aplicarse durante todo el proceso de desarrollo. Pero antes recordemos que todo proceso de evaluación además de la posible detección de defectos conlleva un proceso de depuración, esto es la corrección de los mismos. Ambas tareas (detección y corrección) pueden realizarse por una misma persona o por personas distintas según la organización y el modelo de desarrollo sobre el que se esté aplicando la técnica. Las técnicas de evaluación, tanto las estáticas como las dinámicas, no aportan ayuda en la corrección de los defectos encontrados.

Bien es cierto, que en el caso de las técnicas estáticas, dado que detectan faltas su corrección es más directa. Mientras que las técnicas dinámicas, como se centran en los fallos su proceso de depuración asociado es mucho más complejo, puesto que se debe, primero, buscar la falta que provoca el fallo (lo cual, no es en absoluto inmediato como sabe cualquier programador) y posteriormente corregirlo.

2. EVALUACIÓN DE SOFTWARE Y PROCESO DE DESARROLLO

Tal como se ha indicado anteriormente, es necesario evaluar el sistema software a medida que se va avanzando en el proceso de desarrollo de dicho sistema. De esta forma se intenta que la detección de defectos se haga lo antes posible y tenga menor impacto en el tiempo y esfuerzo de desarrollo. Ahora bien ¿cómo se realiza esta evaluación?

Las técnicas de evaluación estática se aplican en el mismo orden en que se van generando los distintos productos del desarrollo siguiendo una filosofía *top-down*. Esto es, la evaluación estática acompaña a las actividades de desarrollo, a diferencia de la evaluación dinámica que únicamente puede dar comienzo cuando finaliza la actividad de codificación, siguiendo así una estrategia *botom-up*. La evaluación estática es el único modo disponible de evaluación de artefactos para las primeras fases del proceso de desarrollo (análisis y diseño), cuando no existe código. Esta idea se muestra en la Figura 1 en la que como se observa la evaluación estática se realiza en el mismo sentido en que se van generando los productos del desarrollo de software, mientras que la dinámica se realiza en sentido inverso.

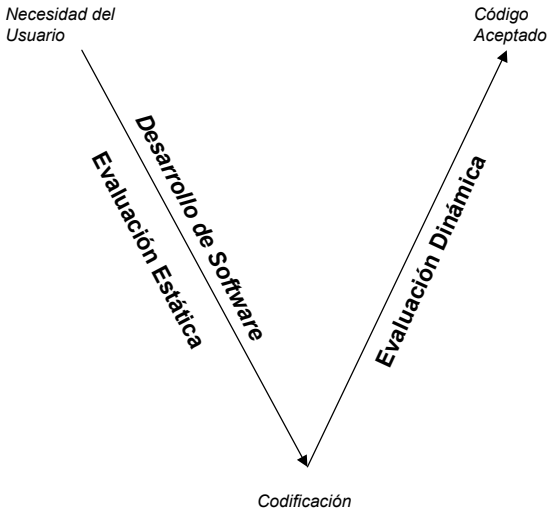

Figura 1. Abstracción de la Relación entre Evaluación y Proceso Software

Más concretamente, la Figura 2 muestra en detalle la aplicación de las técnicas estáticas y dinámicas para evaluar software. La evaluación estática (conocida con el nombre genérico de Revisiones) se realiza en paralelo al proceso de construcción, constando de una actividad de evaluación emparejada con cada actividad de desarrollo. Es decir, la actividad de Definición de Requisitos de Usuario va acompañada de una actividad de Revisión de Requisitos de Usuario, la actividad de Definición de Requisitos Software va emparejada con su correspondiente actividad de revisión y así, sucesivamente.

Las actividades de revisión marcan el punto de decisión para el paso a la siguiente actividad de desarrollo. Es decir, la actividad de requisitos interactúa con la actividad de revisión de requisitos en un bucle de mejora iterativa hasta el momento en que la calidad de los requisitos permite abordar la subsiguiente fase de desarrollo. Lo mismo ocurre con el diseño arquitectónico: sufrirá una mejora iterativa hasta que su nivel de calidad permita pasar al diseño detallado y así, sucesivamente. Nótese que esto también ocurre en la fase de codificación. La actividad siguiente a

la de implementación es la fase de pruebas unitarias. No obstante, antes de pasar a ella, los programas deberán evaluarse estáticamente. Del mismo modo que se ha hecho con los otros productos.

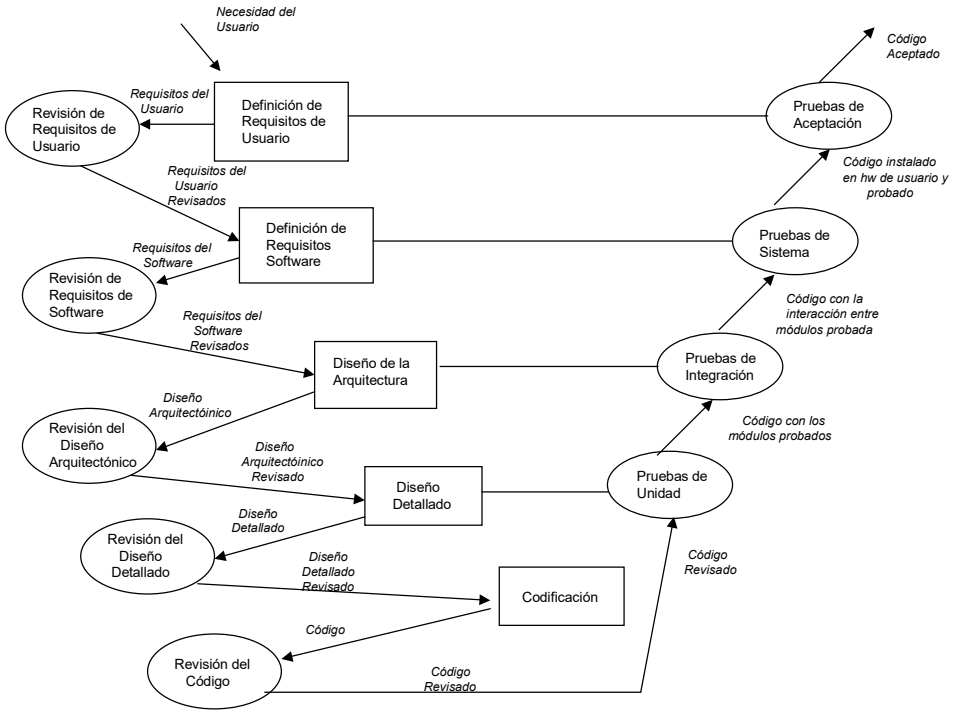

Figura 2. Modelo en V de Evaluación de Software

En otras palabras, las actividades de revisión acompañan las actividades del modelo de desarrollo de software que guía el proyecto. En los modelos de desarrollo de software tradicionales, las actividades de evaluación tanto estáticas como dinámicas tienen una inmersión clara dentro de cada una de las fases del proceso. Es decir, se puede diferenciar claramente donde se introducen las actividades de revisión pues cada fase de desarrollo está claramente diferenciada. En modelos de proceso de software recientes como es el caso de los Modelos de Proceso Ágiles y particularmente en XP, no sucede claramente de la misma manera. La necesidad de hacer liberaciones de código a intervalos cortos de tiempo (totalmente probadas) permite involucrar la evaluación en cada una de las actividades diarias que acompañan el proceso de desarrollo. Las pruebas de aceptación, por ejemplo son pruebas definidas por el cliente con ayuda de un miembro del equipo de desarrollo bajo el rol de Tester o Verificador. Estas buscan medir la funcionalidad de la característica seleccionada por el cliente para ser implementada en esa liberación. Las pruebas se establecen como fundamento del desarrollo, del control de cambios y del trabajo conjunto del cliente con el desarrollador a través de todo el proceso de desarrollo.

En general y por tanto, las actividades de evaluación estática constituyen los puntos de control o revisión utilizados por los gestores de proyectos y las organizaciones para evaluar tanto la calidad de los productos como el progreso del proyecto. Es decir, las actividades de revisión son una herramienta de control para el producto software.

Una vez realizadas estas revisiones se procede con la evaluación dinámica, que como ya se ha indicado se realiza sobre el código. Aunque más adelante se estudiarán en detalle los distintos tipos de pruebas dinámicas, se puede indicar que la primera prueba a realizar es la denominada Prueba de Unidad en la que se buscan errores en los componentes más pequeños del programa (módulos). Estos errores se detectan cuando dichos componentes no actúan como se ha especificado en el diseño detallado.

En el caso de XP, a consecuencia de la refactorización, es necesario correr una sesión de pruebas para verificar que, los cambios no han afectado el comportamiento del sistema, es decir, que no han introducido defectos. En la "Programación por Pares" (uno de los principios de XP), todo el código debe escribirse por pares de programadores. En forma conjunta, dos personas escriben código sentados frente a un ordenador, turnándose en el uso del ratón y el teclado. Mientras uno piensa desde un punto de vista más estratégico y realiza lo que podría llamarse código en tiempo real, el otro programador escribe directamente el código, alternándose en los roles varias veces al día. Las "Pruebas de Unidad" son escritas por cada par de programadores cuando se escribe el código. Las pruebas se ejecutan bajo un proceso de integración y construcción continua que brinda una plataforma estable para el desarrollo.

Seguidamente, se prueban los distintos componentes que constituyen el software en la denominada Prueba de Integración. Esta prueba está orientada a detectar fallos provocados por una incorrecta (no acorde con la especificación de diseño de alto nivel) comunicación entre módulos. El software se puede ejecutar en un contexto hardware concreto, por lo que la Prueba de Sistema es la que se encarga de buscar errores en este ensamblaje sofware/hardware. Finalmente, el usuario ha de realizar la Prueba de Aceptación final sobre el sistema completo.

Para XP, el principio de "Pruebas de Cliente", consiste en que el cliente define una o más pruebas de aceptación. Estas pruebas se automatizan para mostrar que la característica que el cliente desea está funcionando correctamente. El equipo construye esas pruebas y las usa para probarse así mismos y para mostrarle al cliente que la característica ha sido implementada correctamente. La automatización de las pruebas es importante puesto que para XP, la liberación de código en cortos plazos de tiempo es indispensable. Por ello, las pruebas estáticas al código no vendrían siendo la mejor opción.

Nótese cómo la evaluación de los productos software mediante revisiones permite contar con una estimación temprana de la calidad con que se está llevando a cabo el desarrollo. Esto es así porque las revisiones encuentran faltas, pero la cantidad de faltas encontradas en un producto dan una idea de las faltas que aún pueden quedar así como de la calidad del trabajo de desarrollo de dicho producto. La experiencia parece indicar que donde hay un defecto hay otros. Es decir, la probabilidad de descubrir nuevos defectos en una parte del software es proporcional al número de defectos ya descubiertos. Es en este principio sobre el que se basan los métodos de estimación de los defectos que quedan en un software; ya sean los modelos de fiabilidad (que utilizan como entrada los fallos encontrados durante las pruebas) ya sean los métodos de estimación del contenido de faltas (que utilizan como entrada las faltas encontradas mediante revisiones). No obstante, es gracias a la evaluación estática que se puede realizar esta estimación de la calidad del software de manera temprana, puesto que los modelos de fiabilidad requieren el código ya desarrollado para dar una indicación de los posibles fallos que quedan remanentes en dicho código.

En XP, con el principio de “Propiedad Colectiva de Código”, un par de programadores pueden mejorar un código a la vez. Esto significa que, todo el código en general obtiene el beneficio de la atención de muchos programadores. Esto incrementa la calidad del código y disminuye los defectos.

Así pues, la importancia de las técnicas estáticas de evaluación a la hora de controlar el nivel de calidad con el que se está llevando a cabo el desarrollo es crucial. Los modelos que utilizan los datos de las técnicas de testing, ayudan a predecir la fiabilidad del software que se está entregando (cuántas fallos quedan en el sistema sin encontrar), pero poco se puede hacer ya, excepto seguir probando el sistema hasta elevar el nivel de fiabilidad del mismo. Sin embargo, la estimación de faltas que aún quedan en un producto utilizando datos de las revisiones permite dos acciones que ayudan a prevenir futuros defectos en el proyecto:

- Seguir revisando el producto para disminuir el número de faltas remanentes. Por tanto esta detección temprana previene encontrar estas faltas en estadios más avanzados del desarrollo. Es decir, la falta que detectemos en los requisitos estaremos evitando contagiarla al diseño y al código.
- Tomar medidas correctivas del desarrollo si las estimaciones indican que se está llevando a cabo un trabajo pobre. Es decir, si las estimaciones de faltas remanentes indican que un determinado producto contiene más faltas de las habituales, algo se está haciendo mal (hay problemas en el equipo de desarrollo, algún miembro del equipo tiene problemas que está afectando a su trabajo, hay problemas con las técnicas que se están utilizando, quizás el equipo no las conoce bien, etc.) y deben tomarse acciones que remedien o palien estos problemas antes de que afecten al resultad final del proyecto.

En las secciones 3 y 4 se detallan las técnicas estáticas y dinámicas respectivamente.

3. TÉCNICAS DE EVALUACIÓN ESTÁTICA

3.1 BENEFICIOS DE LAS REVISIONES

La razón para buscar defectos en productos tempranos es porque éstos se traducen en defectos en el producto final. Es decir, defectos en los requisitos se traducirán en defectos en el sistema final. Veamos una analogía con la arquitectura de edificios. Si en un plano el color de una línea indica su significado, una confusión en el color se traducirá en un error en el edificio. Por ejemplo, si el azul indica tuberías de agua y el amarillo cables eléctricos y el arquitecto comete un error usando el azul en una conducción eléctrica, los electricistas que usen el plano como guía para su trabajo no colocarán cables eléctricos mientras que los fontaneros colocarán tuberías de agua donde no debían ir. El plano de un edificio es el artefacto equivalente al diseño de un producto software. Si un diseño contiene defectos, seguramente estos defectos se trasmitirán al código cuando los programadores usen ese diseño como guía para su trabajo.

La detección temprana de errores acarrea grandes beneficios. Si las revisiones únicamente se aplican al código mejoran la calidad y producen ahorros en los costos del proyecto. Pero los ahorros son mayores si se inspeccionan artefactos tempranos del desarrollo. Estudiando los resultados publicados sobre ahorros con las revisiones, puede afirmarse que la utilización de inspecciones de código produce un ahorro del 39% sobre el coste de detectar y corregir defectos, frente a únicamente utilizar la evaluación dinámica. Sin embargo, el ahorro es del 44% si se inspecciona también el diseño.

La experiencia demuestra que entre el 30% y el 70% de los defectos, de diseño y código son detectados por las técnicas estáticas. Esto supone un gran ahorro, pues la corrección es más fácil y menos costosa durante la evaluación estática que durante la dinámica. Nótese que cuando durante la evaluación dinámica del sistema se detecta un fallo en un programa, lo que se detecta es el *fallo*, no la falta que lo provoca. Es decir, tras la detección del fallo, se requiere una labor de localización en el programa de la falta que provocó el fallo. Sin embargo, con las técnicas estáticas, lo que se detecta son directamente *faltas*. Por tanto, una vez detectada, se puede pasar a la fase de corrección. Es decir, desaparece la tarea de localización de la falta. Esto significa, que las técnicas estáticas son más baratas por falta que las dinámicas.

Las revisiones también proporcionan beneficios más generales. Entre éstos se pueden citar están:

- Evaluación del progreso del proyecto
- Potencia las capacidades de los participantes
- Mejoran la comunicación entre el equipo de desarrollo, aumentando su motivación, pues los productos pasan a ser documentos públicos.
- Proporciona aprendizaje, retroalimentación y prevención
- Forma y educa a los participantes

En el caso concreto de las revisiones de código, éstas, además, permiten localizar secciones críticas, lo que permitirá dedicar un mayor esfuerzo a ellas en la fase de pruebas.

3.2 OBJETIVOS DE LA EVALUACIÓN ESTÁTICA

La evaluación estática de los distintos artefactos o productos que se generan en el desarrollo de software (especificación de requisitos, modelos conceptuales, diseño, código, etc.) pretende comprobar su calidad.

La calidad significa una cosa distinta para cada producto, precisamente porque son artefactos distintos. Del mismo modo que la calidad de un plano y la calidad de una casa significa cosas distintas. En un plano de un futuro edificio se desea que sea claro (se entienda suficientemente bien como para servir de guía a la construcción del edificio), que sea correcto (por ejemplo, que las líneas que identifican paredes indiquen, a escala, efectivamente el lugar donde se desea que vayan las paredes), que no tenga inconsistencias (por ejemplo, entre las distintas hojas que forman el plano; si una página se focaliza, digamos, en una habitación que en otra página aparecía sólo sus cuatro paredes, que las medidas de las líneas en ambas páginas se correspondan con la misma medida de la realidad), etc.. Sin embargo, de una casa se espera que sea robusta (por ejemplo, que no se caiga), usable (por ejemplo, que los peldaños de las escaleras no sean tan estrechos que provoquen caídas) etc. Por tanto, cuando se esté evaluando estáticamente un producto software, es importante que el evaluador tenga en mente qué tipo de defectos está buscando y cuál sería un producto de ese tipo de calidad adecuada. Digamos que si uno no sabe lo que busca (por ejemplo, inconsistencias al revisar la calidad de un plano) es difícil que lo encuentre, aunque lo tenga delante.

Los defectos que se buscan al evaluar estáticamente los productos software son:

- Para los **requisitos**:
    - *Corrección*. Los requisitos especifican correctamente lo que el sistema debe hacer. Es decir, un requisito incorrecto es un requisito que no cumple bien su función. Puesto que la función de un requisito es indicar qué debe hacer el sistema, un requisito incorrecto será aquel que indica incorrectamente lo que debe hacer el sistema. Por ejemplo: el algoritmo indicado para hacer un cálculo está mal; dice que algo debe eliminarse cuando en realidad debe guardarse; etc. En otras palabras, un requisito incorrecto no se corresponde con lo acordado o adecuado; contiene un error.
    - *Compleción*. Especificación completamente el problema. Está especificado todo lo que tiene que hacer el sistema y no incluye nada que el sistema no deba hacer. En dos palabras: no falta nada; no sobra nada
    - *Consistencia*. No hay requisitos contradictorios.
    - *Ambigüedad*. Los requisitos no pueden estar sujetos a interpretación. Si fuese así, un mismo requisito puede ser interpretado de modo distinto por dos personas diferentes y, por tanto, crear dos sistemas distintos. Si esto es así, los requisitos pierden su valor pues dejan de cumplir su función (indicar qué debe hacer el sistema). Las ambigüedades provocan interpretación por parte de la persona que use o lea los requisitos. Por tanto, una especificación debe carecer de ambigüedades.
    - *Claridad*. Se entiende claramente lo que está especificado.
- Para el **diseño**:

- *Corrección*. El diseño no debe contener errores. Los errores de corrección se pueden referir a dos aspectos. Defectos de "escritura", es decir, defectos en el uso de la notación de diseño empleada (el diseño contiene detalles prohibidos por la notación). Defectos con respecto a los requisitos: el diseño no realiza lo que el requisito establece. Hablando apropiadamente, los primeros son los puros defectos de corrección, mientras que los segundos son defectos de validez.

- *Compleción*. El diseño debe estar completo. Ya sea que diseña todo el sistema marcado por los requisitos; ya sea no diseñando ninguna parte no indicada en los requisitos. De nuevo, nada falta, nada sobra.

- *Consistencia*. Al igual que en los requisitos, el diseño debe ser consistente entre todas sus partes. No puede indicarse algo en una parte del diseño, y lo contrario en otra.

- *Factibilidad*. El diseño debe ser realizable. Debe poderse implementar.

- *Trazabilidad*. Se debe poder navegar desde un requisito hasta el fragmento de diseño donde éste se encuentra representado.

- **Código Fuente**:
  - *Corrección*. El código no debe contener errores. Los errores de corrección se pueden referir a dos aspectos. Defectos de "escritura", es decir, lo que habitualmente se conoce por "programa que no funciona". Por ejemplo, bucles infinitos, variable definida de un tipo pero utilizada de otro, contador que se sale de las dimensiones de un *array*, etc. Defectos con respecto al diseño: el diseño no realiza lo que el diseño establece.

De nuevo, hablando apropiadamente, los primeros son los puros defectos de corrección, mientras que los segundos son defectos de validez. Un defecto de corrección es un código que está mal para cualquier dominio. Un defecto de validez es un código que, en este dominio particular (el marcado por esta necesidad de usuario, estos requisitos, y este diseño) hace algo inapropiado. Por ejemplo, define una variable de un tipo (y se usa en el programa con ese tipo, es decir, "a primera vista" no hay nada incorrecto en la definición del tipo y su uso) que no es la que corresponde con el problema; o define un *array* de un tamaño que no es el que se corresponde con el problema. Nótese que para detectar los errores de validez (en cualquier producto) debe entenderse el problema que se pretende resolver, mientras que los defectos de corrección son errores siempre, aún sin conocer el problema que se pretende resolver.

  - *Compleción*. El código debe estar completo. Una vez más, nada falta ni nada sobra (con respecto, en este caso, al diseño)

  .

  - *Consistencia*. Al igual que en los requisitos y diseño, el código debe ser consistente entre todas sus partes. No puede hacerse algo en una parte del código, y lo contrario en otra.

- *Trazabilidad*. Se debe poder navegar desde un requisito hasta el fragmento de código donde éste se ejecute, pasando por el fragmento de diseño.

3.3 TÉCNICAS DE EVALUACIÓN ESTÁTICA

Las técnicas de Evaluación estática de artefactos del desarrollo se las conoce de modo genérico por **Revisiones**. Las revisiones pretenden detectar *manualmente* defectos en cualquier producto del desarrollo. Por manualmente queremos decir que el producto en cuestión (sea requisito, diseño, código, etc.) está impreso en papel y los revisores están analizando ese producto mediante la lectura del mismo, *sin ejecutarlo.*

Existen varios tipos de revisiones, dependiendo de qué se busca y cómo se analiza ese producto. Podemos distinguir entre:

- *Revisiones informales*, también llamadas inadecuadamente sólo Revisiones (lo cual genera confusión con el nombre genérico de todas estas técnicas). Las Revisiones Informales no dejan de ser un intercambio de opiniones entre los participantes.

- Revisiones formales o *Inspecciones*. En las Revisiones Formales, los participantes son responsables de la fiabilidad de la evaluación, y generan un informe que refleja el acto de la revisión. Por tanto, sólo consideramos aquí como técnica de evaluación las revisiones formales, puesto que las informales podemos considerarlas un antepasado poco evolucionado de esta misma técnica.

- *Walkthrough*. Es una revisión que consiste en simular la ejecución de casos de prueba para el programa que se está evaluando. No existe traducción exacta en español y a menudo se usa el término en ingles. Quizás la mejor traducción porque ilustra muy bien la idea es *Recorrido*. De hecho, con los walkthrough se recorre el programa imitando lo que haría la computadora.

- *Auditorias.* Las auditorias contrastan los artefactos generados durante el desarrollo con estándares, generales o de la organización. Típicamente pretenden comprobar formatos de documentos, inclusión de toda la información necesaria, etc. Es decir, no se tratan de comprobaciones técnicas, sino de gestión o administración del proyecto.

3.4 INSPECCIONES

3.4.1 ¿QUÉ SON LAS INSPECCIONES?

Las inspecciones de software son un método de análisis estático para verificar y validar un producto software manualmente. Los términos Inspecciones y Revisiones se emplean a menudo como sinónimos. Sin embargo, como ya se ha visto, este uso intercambiable no es correcto.

Las Inspecciones son un proceso bien definido y disciplinado donde *un equipo de personas* cualificadas analiza un producto software usando una *técnica de lectura* con el propósito de detectar defectos. El objetivo principal de una inspección es detectar faltas antes de que la fase de prueba comience. Cualquier desviación de una propiedad de calidad predefinida es considerada un defecto.

Para aprender a realizar inspecciones vamos a estudiar primero el proceso que debe seguirse y luego las técnicas de lectura.

3.4.2 EL PROCESO DE INSPECCIÓN

Las Inspecciones constan de dos partes: Primero, la *comprensión* del artefacto que se inspecciona; Y en segundo lugar, la *búsqueda de faltas* en dicho artefacto. Más concretamente, una inspección tiene cuatro fases principales:

1. **Inicio** – El objetivo es preparar la inspección y proporcionar la información que se necesita sobre el artefacto para realizar la inspección.

2. **Detección de defectos** – Cada miembro del equipo realiza *individualmente* la lectura del material, compresión del artefacto a revisar y la detección de faltas. Las Técnicas de Lectura ayudan en esta etapa al inspector tanto a comprender el artefacto como a detectar faltas. Basándose en las faltas detectadas cada miembro debe realizar una estimación subjetiva del número de faltas remanentes en el artefacto.

3. **Colección de defectos** – El registro de las faltas encontrada por cada miembro del equipo es compilado en un solo documento que servirá de basa a la discusión sobre faltas que se realizará *en grupo*. Utilizando como base las faltas comunes encontradas por los distintos inspectores se puede realizar una estimación objetiva del número de faltas remanentes. En la reunión se discutirá si las faltas detectadas son falsos positivos (faltas que algún inspector cree que son defectos pero que en realidad no lo son) y se intentará encontrar más faltas ayudados por la sinergia del grupo.

4. **Corrección y seguimiento** – El autor del artefacto inspeccionado debe corregir las faltas encontradas e informar de las correcciones realizadas a modo de seguimiento.

Estas fases se subdividen además en varias subfases:

1. Inicio
   1.1 Planificación
   1.2 Lanzamiento
2. Detección de defectos
3. Colección de defectos
   3.1 Compilación
   3.2 Inspección en grupo
4. Corrección y seguimiento
   4.1 Corrección
   4.2 Seguimiento

Veamos cada una de estas fases;

Durante **La Planificación** se deben seleccionar los participantes, asignarles roles, preparar un calendario para la reunión y distribuir el material a inspeccionar. Típicamente suele haber una persona en la organización o en el proyecto que es responsable de planificar todas las actividades de

inspección, aunque luego juegue además otros papeles. Los papeles que existen en una inspección son:

- *Organizador*. El organizador planifica las actividades de inspección en un proyecto, o incluso en varios proyectos (o entre proyectos, porque se intercambian participantes: los desarrollados de uno son inspectores de otro).

- *Moderador*. El moderador debe garantizar que se sigan los procedimientos de la inspección así como que los miembros del equipo cumplan sus responsabilidades. Además, modera las reuniones, lo que significa que el éxito de la reunión depende de esta persona y, por tanto, debe actuar como líder de la inspección. Es aconsejable que la persona que juegue este rol haya seguido cursos de manejo de reuniones y liderazgo

- *Inspector*. Los inspectores son los responsables de detectar defectos en el producto software bajo inspección. Habitualmente, todos los participantes en una inspección actúan también como inspectores, independientemente de que, además, jueguen algún otro papel.

- *Lector/Presentador*. Durante la reunión para la inspección en grupo, el lector dirigirá al equipo a través del material de modo completo y lógico. El material debe ser parafraseado a una velocidad que permita su examen detallado al resto de los participantes. Parafrasear el material significa que el lector debe explicar e interpretar el producto en lugar de leerlo literalmente.
- *Autor*. El autor es la persona que ha desarrollado el producto que se esta inspeccionando y es el responsable de la corrección de los defectos durante la fase de corrección. Durante la reunión, el autor contesta a las preguntas que el lector no es capaz de responder. El autor no debe actuar al mismo tiempo ni de moderador, ni de lector, ni de escriba.

- *Escriba*. El secretario o escriba es responsable de incorporar todos los defectos en una lista de defectos durante la reunión.

- *Recolector*. El recolector recoge los defectos encontrados por los inspectores en caso de no haber una reunión de inspección.

Es necesario hacer ciertas consideraciones sobre *el número de participantes*. Un equipo de inspección nunca debería contar con más de cinco miembros. Por otro lado, el número mínimo de participantes son dos: el autor (que actúa también de inspector) y un inspector. Lo recomendable es comenzar con un equipo de tres o cuatro personas: el autor, uno o dos inspectores y el moderador (que actúa también como lector y escriba). Tras unas cuantas inspecciones la organización puede experimentar incorporando un inspector más al grupo y evaluar si resulta rentable en términos de defectos encontrados.

Sobre el tema de cómo *seleccionar las personas adecuadas* para una inspección, los candidatos principales para actuar como inspectores es el personal involucrado en el desarrollo del producto. Se pueden incorporar inspectores externos si poseen algún tipo de experiencia específica que enriquezca la inspección. Los inspectores deben tener un alto grado de experiencia y conocimiento.

La fase de **Lanzamiento** consiste en una primera reunión donde el autor explica el producto a inspeccionar a los otros participantes. El objetivo principal de esta reunión de lanzamiento, es facilitar la comprensión e inspección a los participantes. No obstante, este meeting no es

completamente necesario, pues en ocasiones puede consumir más tiempo y recursos de los beneficios que reporte. Sin embargo, existen un par de condiciones bajo las cuales es recomendable realizar esta reunión. En primer lugar, cuando el artefacto a inspeccionar es complejo y difícil de comprender. En este caso una explicación por parte del autor sobre el producto a inspeccionar facilita la comprensión al resto de participantes. En segundo lugar, cuando el artefacto a inspeccionar pertenece a un sistema software de gran tamaño. En este caso, se hace necesario que el autor explique las relaciones entre el artefacto inspeccionado y el sistema software en su globalidad.

La fase de **Detección de Defectos** es el corazón de la inspección. El objetivo de esta fase es escudriñar un artefacto software pare obtener defectos. Localizar defectos es una actividad en parte individual y en parte de grupo. Si se olvida la parte individual de la inspección, se corre el riesgo de que los participantes sean más pasivos durante la reunión y se escuden en el grupo para evitar hacer su contribución. Así pues, es deseable que exista una fase de detección individual de defectos con el objetivo explícito de que cada participante examine y entienda en solitario el producto y busque defectos. Este esfuerzo individual garantiza que los participantes irán bien preparados a la puesta en común.

Los defectos detectados por cada participante en la inspección deben ser reunidos y documentado. Es más, debe decidirse si un defecto es realmente un defecto. Esta recolección de defectos y la discusión sobre falsos positivos se realizan, respectivamente, en la fase de **Compilación y Reunión**. La recolección de defectos debe ayudar a tomar la decisión sobre si es necesaria una reinspección del artefacto o no. Esta decisión dependerá de la cantidad de defectos encontrados y sobre todo de la coincidencia de los defectos encontrados por distintos participantes. Una coincidencia alta de los defectos encontrados por unos y por otros (y un numero bajo de defectos encontrados) hace pensar que la cantidad de defectos que permanecen ocultos sea baja. Una coincidencia pobre (y un numero relativamente alto de defectos encontrados) hace pensar que aun quedan muchos defectos por detectar y que, por tanto, es necesaria una reinspección (una vez acabada ésta y corregidas las faltas encontradas).

Dado que una reunión consume bastantes recursos (y más cuantos más participantes involucre) se ha pensado en una alternativa para hacer las reuniones más ligeras. Las llamadas reuniones de deposición, donde sólo asisten tres participantes: moderador, autor, y un representante de los inspectores. Este representante suele ser el inspector de más experiencia, el cual recibe los defectos detectados por los otros inspectores y decide él, unilateralmente, sobre los falsos positivos. Algunos autores incluso han dudado del efecto sinergia de las reuniones y han aconsejado su no realización. Parece que lo más recomendable es que las organizaciones comiencen con un proceso de inspección tradicional, donde la reunión sirve para compilar defectos y discutir falsos positivos, y con el tiempo y la experiencia prueben a variar el proceso eliminando la reunión y estudiando si se obtienen beneficios equivalentes en términos de defectos encontrados.

Es importante resaltar, que la reunión de una inspección no es una sesión para resolver los defectos u otros problemas. No se deben discutir en estas reuniones ni soluciones digamos radicales (otras alternativas de diseño o implementación, que el autor no ha utilizado pero podría haberlo hecho), ni cómo resolver los defectos detectados, y, mucho menos, discutir sobre conflictos personales o departamentales.

Finalmente, el autor corrige su artefacto para resolver los defectos encontrados o bien proporciona una explicación razonada sobre porqué cierto defecto detectado en realidad no lo es. Para esto, el autor repasa la lista de defectos recopilada y discute o corrige cada defecto. El autor deberá enviar al moderador un informe sobre los defectos corregidos o, en caso de no haber corregido alguno, porqué no debe corregirse. Este informe sirve de seguimiento y cierre de la inspección, o, en caso

de haberse decidido en la fase de recopilación que el artefacto necesitaba reinspección, se iniciará de nuevo el proceso.

3.4.3 ESTIMACIÓN DE LOS DEFECTOS REMANENTES

A pesar de que el propósito principal de las Inspecciones es detectar y reducir el número de defectos, un efecto colateral pero importante es que permiten realizar desde momentos muy iniciales del desarrollo predicciones de la calidad del producto. Concretamente, las estimaciones de las faltas remanentes tras una inspección deben utilizarse como control de la calidad del proceso de desarrollo.

Hay varios momentos de estimación de faltas remanentes en una inspección. Al realizar la búsqueda individual de faltas, el inspector puede tener una idea de las faltas remanentes en base a las siguientes dos heurísticas:

- Encontrar muchas faltas es sospechoso. Muchas faltas detectadas hacen pensar que debe haber muchas más, puesto que la creencia de que queden pocas faltas sólo se puede apoyar en la confianza en el proceso de inspección y no en la calidad del artefacto (que parece bastante baja puesto que hemos encontrado muchas faltas)
- Encontrar muy pocas faltas también resulta sospechoso, especialmente si es la primera vez que se inspecciona este artefacto. Pocas faltas hacen pensar que deben quedar muchas más, puesto que esta situación hace dudar sobre la calidad del proceso de inspección: No puede saberse si se han encontrado pocas debido a la alta calidad del artefacto o a la baja calidad de la inspección.

La estimación más fiable sobre el número de faltas remanentes que se puede obtener en una Inspección es la coincidencia de faltas entre los distintos inspectores. Los métodos que explotan coincidencias para estimar se llaman Estimaciones Captura-Recaptura y no son originarias de la Ingeniería del Software. En concreto lo usó por primera vez Laplace para estimar la población de Francia en 1786, y se utilizan a menudo en Biología para estimar el tamaño de poblaciones de animales. En el caso de las Inspecciones, el nivel de coincidencia de las faltas detectadas por los distintos revisores es usado como estimador[15].

La idea sobre la que se basa el uso de las coincidencias es la siguiente: Pocas coincidencias entre los revisores, significa un número alto de faltas remanentes; Muchas coincidencias, significará pocas faltas remanentes. Los casos extremos son los siguientes. Supongamos que todos los revisores han encontrado exactamente el mismo conjunto de faltas. Esto debe significar que deben quedar muy pocas faltas, puesto que el proceso de inspección parece haber sido bueno (los inspectores han encontrado las mismas faltas) parece poco probable que queden más faltas por ahí que ningún revisor ha encontrado. Supongamos ahora que ningún revisor ha encontrado las mismas faltas que otro revisor. Esto debe querer decir que quedan muchas más por encontrar, puesto que el proceso de revisión parece haber sido pobre (a cada revisor se le han quedado ocultas n faltas –todas las encontradas por los otros revisores-) vaya usted a saber cuántas faltas más han quedado ocultas a todos los revisores. Esta situación debería implicar una nueva inspección del artefacto (tanto para mejorar la calidad del mismo, como para hacer un intento de mejorar la propia inspección).

[15] Un estimador en una fórmula usada para predecir el número de faltas que quedan. Un modelo de estimación es el paraguas para denominar a un conjunto de estimadores basados en los mismos prerrequisitos.

3.4.4 TÉCNICAS DE LECTURA

Las técnicas de lectura son guías que ayudan a detectar defectos en los productos software. Típicamente, una técnica de lectura consiste en una serie de pasos o procedimientos cuyo propósito es que el inspector adquiere un conocimiento profundo del producto software que inspecciona. La comprensión de un producto software bajo inspección es un prerrequisito para detectar defectos sutiles y, o, complejos. En cierto sentido, una técnica de lectura puede verse como un mecanismo para que los inspectores detecten defectos en el producto inspeccionado.

Las técnicas de lectura más populares son la lectura Ad-hoc y la lectura basada en listas de comprobación. Ambas técnicas pueden aplicarse sobre cualquier artefacto software, no solo sobre código. Además de estas dos técnicas existen otras que, aunque menos utilizadas en la industria, intentan abordar los problemas de la lectura con listas y la lectura ad-hoc: Lectura por Abstracción, Revisión Activa de Diseño y Lectura Basada en Escenarios. Esta última se trata de una familia de técnicas a la que pertenecen: Lectura Basada en Defectos y Lectura Basada en Perspectivas. Veamos en qué consisten cada una.

3.4.4.1 LECTURA SIN CHECKLISTS Y CON CHECKLISTS

En la técnica de **lectura Ad-hoc**, el producto software se entrega a los inspectores sin ninguna indicación o guía sobre cómo proceder con el producto ni qué buscar. Por eso la denominamos también cómo Lectura sin Checklists.

Sin embargo, que los participantes no cuenten con guías de qué buscar no significa que no escudriñen sistemáticamente el producto inspeccionado, ni tampoco que no tengan en mente el tipo de defecto que están buscando. Como ya hemos dicho antes, si no se sabe lo que se busca, es imposible encontrarlo. El término "ad-hoc" sólo se refiere al hecho de no proporcionar apoyo a los inspectores. En este caso la detección de los defectos depende completamente de las habilidades, conocimientos y experiencia del inspector.

Típicamente, el inspector deberá buscar secuencialmente los defectos típicos del producto que esté leyendo (y que hemos indicado más arriba). Por ejemplo, si se está inspeccionando unos requisitos, el inspector, buscará sistemática y secuencialmente defectos de corrección, de completud, de ambigüedad, etc.

Para practicar esta técnica, en el Anexo A aparece unos requisitos con defectos. Intenta buscarlos de acuerdo a lo indicado en el párrafo anterior. Sin guía alguna, simplemente utilizando la lista de criterios de calidad que debe cumplir unos requisitos que hemos indicado anteriormente.

La **lectura basada en Listas de Comprobación** (checklists, en inglés) proporciona un apoyo mayor mediante preguntas que los inspectores deben de responder mientras leen el artefacto. Es decir, esta técnica proporciona listas que ayudan al inspector a saber qué tipo de faltas buscar. Aunque una lista supone más ayuda que nada, esta técnica presenta la debilidad de que las preguntas proporcionan poco apoyo para ayudar a un inspector a entender el artefacto inspeccionado. Además, las preguntas son a menudo generales y no suficientemente adaptadas a un entorno de desarrollo particular. Finalmente, las preguntas en una lista de comprobación están limitadas a la detección de defectos de un tipo determinado, típicamente de corrección, puesto que las listas establecen errores universales (independientes del contexto y el problema).

3.4.4.1.1 Checklists para Requisitos y Diseño

Las listas de comprobación para requisitos contienen preguntas sobre los defectos típicos que suelen aparecer en los requisitos. Preguntas típicas que aparecen en las checklists de requisitos son:

- ✓ ¿Existen contradicciones en la especificación de los requisitos?
- ✓ ¿Resulta comprensible la especificación?
- ✓ ¿Está especificado el rendimiento?
- ✓ ¿Puede ser eliminado algún requisito? ¿Pueden juntarse dos requisitos?
- ✓ ¿Son redundantes o contradictorios?
- ✓ ¿Se han especificado todos los recursos hardware necesarios?
- ✓ ¿Se han especificado las interfaces externas necesarias?
- ✓ ¿Se han definido los criterios de aceptación para cada una de las funciones especificadas?

Nótese que las cinco primeras preguntas, corresponden simplemente a los criterios de calidad de los requisitos. Mientras que las tres últimas tratan olvidos típicos al especificar requisitos. Las dos que aparecen en primer lugar son comprobaciones sobre la especificación del hardware sobre el que correrá el futuro sistema y sobre cómo deberá interactuar con otros sistemas. La última comprueba que los requisitos contienen criterios de aceptación para cuando se realicen las pruebas de aceptación.

Los requisitos necesitan ser evaluados de forma crítica para prevenir errores. En esta fase radica la calidad del producto software desde la perspectiva del usuario. Si la evaluación en general es difícil, la de los requisitos en particular lo es más, debido a que lo que se evalúa es la definición del problema.

Con respecto al diseño, los objetivos principales de su evaluación estática son:

- Determinar si la solución elegida es la mejor de todas las opciones; es decir, si la opción es la más simple y la forma más fácil de realizar el trabajo.
- Determinar si la solución abarca todos los requisitos descritos en la especificación; es decir, si la solución elegida realizará la función encomendada al software.

Al igual que la evaluación de requisitos, la evaluación de diseño es crucial, ya que los defectos de diseño que queden y sean transmitidos al código, cuando sean detectados en fases más avanzadas del desarrollo, o incluso durante el uso, implicará un rediseño del sistema, con la subsiguiente re-codificación. Es decir, existirá una pérdida real de trabajo.

Veamos un ejemplo de preguntas para el diseño:

- ✓ ¿Cubre el diseño todos los requisitos funcionales?
- ✓ ¿Resulta ambigua la documentación del diseño?
- ✓ ¿Se ha aplicado la notación de diseño correctamente?
- ✓ ¿Se han definido correctamente las interfaces entre elementos del diseño?
- ✓ ¿Es el diseño suficientemente detallado como para que sea posible implementarlo en el lenguaje de programación elegido?

En el Anexo B aparecen listas de comprobación para diferentes productos del desarrollo proporcionadas por la empresa Construx. Intenta revisar ahora de nuevo los requisitos del Anexo A, esta vez usando las listas de comprobación del Anexo B. Esta misma especificación de requisitos se

usa más tarde con otra técnica de lectura. Será entonces cuando aportemos la solución sobre qué defectos contienen estos requisitos.

3.4.4.1.2 Checklists para Código

Las listas para código han sido mucho más estudiadas que para otros artefactos. Así hay listas para distintos lenguajes de programación, para distintas partes de código, etc.

Una típica lista de comprobación para código contendrá varias partes (una por los distintos tipos de defecto que se buscan) cada una con preguntas sobre defectos universales y típicos. Por ejemplo:

- Lógica del programa:
  - ✓ ¿Es correcta la lógica del programa?
  - ✓ ¿Está completa la lógica del programa?, es decir, ¿está todo correctamente especificado sin faltar ninguna función?
- Interfaces Internas:
  - ✓ ¿Es igual el número de parámetros recibidos por el módulo a probar al número de argumentos enviados?, además, ¿el orden es correcto?
  - ✓ ¿Los atributos (por ejemplo, tipo y tamaño) de cada parámetro recibido por el módulo a probar coinciden con los atributos del argumento correspondiente?
  - ✓ ¿Coinciden las unidades en las que se expresan parámetros y argumentos? Por ejemplo, argumentos en grados y parámetros en radianes.¿Altera el módulo un parámetro de sólo lectura?¿Son consistentes las definiciones de variables globales entre los módulos?
- Interfaces Externas:
  - ✓ ¿Se declaran los ficheros con todos sus atributos de forma correcta?
  - ✓ ¿Se abren todos los ficheros antes de usarlos?
  - ✓ ¿Coincide el formato del fichero con el formato especificado en la lectura?¿Se manejan correctamente las condiciones de fin de fichero? ¿Se los libera de memoria?
  - ✓ ¿Se manejan correctamente los errores de entrada/salida?
- Datos:
  - o Referencias de datos. Se refieren a los accesos que se realizan a los mismos. Ejemplos típicos son:
    - ✓ Utilizar variables no inicializadas
    - ✓ Salirse del límite de las matrices y vectores
    - ✓ Superar el límite de tamaño de una cadena
  - o Declaración de datos. El propósito es comprobar que todas las definiciones de los datos locales son correctas. Por ejemplo:
    - ✓ Comprobar que no hay dos variables con el mismo nombre
    - ✓ Comprobar que todas las variables estén declaradas
    - ✓ Comprobar que las longitudes y tipos de las variables sean correctos.
  - o Cálculo. Intenta localizar errores derivados del uso de las variables. Por ejemplo:
    - ✓ Comprobar que no se producen overflow o underflow (valores fuera de rango, por encima o por debajo) en los cálculos o divisiones por cero.
  - o Comparación. Intenta localizar errores en las comparaciones realizadas en instrucciones tipo *If-Then-Else*, *While*, etc. Por ejemplo:
    - ✓ Comprobar que no existen comparaciones entre variables con diferentes tipos de datos o si las variables tienen diferente longitud.

- ✓ Comprobar si los operadores utilizados en la comparación son correctos, si utilizan operadores booleanos comprobar si los operandos usados son booleanos, etc.

En el Anexo C se proporcionan distintas listas de comprobación para diversas partes y características del código. En el Anexo D tienes un programa y una pequeña lista de comprobación de código para que te ejercites buscando defectos. Deberías detectar al menos un par de defectos, al menos. Más adelante usaremos este mismo programa para practicar con otra técnica y será entonces cuando proporcionaremos la lista de defectos de este programa.

3.4.4.2 LECTURA POR ABSTRACCIÓN SUCESIVA

La **Lectura por Abstracción Sucesiva** sirve para inspeccionar código, y no otro tipo de artefacto como requisitos o diseño. La idea sobre la que se basa esta técnica de lectura para detectar defectos es en la comparación entre la especificación del programa (es decir, el texto que describe lo que el programa debería hacer) y lo que el programa hacer realmente. Naturalmente, todos aquellos puntos donde no coincida lo que el programa debería hacer con lo que el programa hace es un defecto.

Dado que comparar código con texto (la especificación) es inapropiado pues se estarían comparando unidades distintas (peras con manzanas), se hace necesario convertir ambos artefactos a las mismas "unidades". Lo que se hace, entonces, es convertir el programa en una especificación en forma de texto. De modo que podamos comparar especificación (texto) con especificación (texto).

Obtener una especificación a partir de un código significa recorrer el camino de la programación en sentido inverso. El sentido directo es obtener un código a partir de una especificación. Este camino se recorre en una seria de pasos (que a menudo quedan ocultos en la mente del programador y no se hacen explícitos, pero que no por eso dejan de existir). El recorrido directo del camino de la especificación al código consta de los siguientes pasos:

1. Leer la especificación varias veces hasta que el programador ha entendido lo que el código debe hacer.

2. Descomponer la tarea que el programa debe hacer en subtareas, que típicamente se corresponderán con las funciones o módulos que compondrán el programa. Esta descomposición muestra la relación entre funciones, que no siempre es secuencial, sino típicamente en forma de árbol: Funciones alternativas que se ejecutarán dependiendo de alguna condición; Funciones suplementarias que se ejecutaran siempre una si se ejecuta la otra; etc.

3. Para cada uno de estas subtareas (funciones):
   3.1. Hacer una descripción sistemática (típicamente en pseudocódigo) de cómo realizar la tarea. En esta descripción se pueden ya apreciar las principales estructuras de las que constará la función (bucles, condicionales, etc.)
   3.2. Programar cada línea de código que compone la función

Como puede observarse en este proceso, el programador trabaja desde la especificación por descomposiciones sucesivas. Es decir, dividiendo una cosa compleja (la especificación) en cosas cada vez mas sencillas (primero las funciones, luego las estructuras elementales de los programas,

finalmente las líneas de código). Este tipo de tarea se realiza de arriba hacia abajo, partiendo de la especificación (arriba o nodo raíz) y llegando a n líneas de código (abajo o nodos hojas).

Pues bien, si queremos obtener una especificación a partir de un código deberemos hacer este mismo recorrido pero en sentido contrario: de abajo hacia arriba. Por tanto, deberemos comenzar con las líneas de código, agruparlas en estructuras elementales, y éstas en funciones y éstas en un todo que será la descripción del comportamiento del programa. En este caso no estamos trabajando descomponiendo, sino componiendo. La tarea que se realiza es de abstracción. De ahí el nombre de la técnica: abstracción sucesiva (ir sucesivamente –ascendiendo en niveles cada vez superiores- abstrayendo qué hace cada elemento –primero las líneas, luego las estructuras, finalmente las funciones).

Esta técnica requiere que el inspector lea una serie de líneas de código y que abstraiga la función que estas líneas computan. El inspector debe repetir este procedimiento hasta que la función final del código que se está inspeccionando se haya abstraído y pueda compararse con la especificación original del programa.

Más concretamente, el proceso que se debe seguir para realizar la abstracción sucesiva es el siguiente:

1. Leer por encima el código para tener una idea general del programa.

2. Determinar las dependencias entre las funciones individuales del código fuente (quién llama a quién). Para esto puede usarse un árbol de llamadas para representar tales dependencias comenzando por las hojas (funciones que no llaman a nadie) y acabando por la raíz (función principal).

3. Comprender qué hace cada función. Para ello se deberá: Entender la estructura de cada función individual identificando las estructuras elementales (secuencias, condicionales, bucles, etc.) y marcándolas; Combinar las estructuras elementales para formar estructuras más grandes hasta que se haya entendido la función entera. Es decir, para cada función y comenzando desde las funciones hoja y acabando por la raíz:
    3.1. Identificar las estructuras elementales de cada función y marcarlas de la más interna a la más externa.
    3.2. Determinar el significado de cada estructura comenzando con la más interna. Para ello pueden seguirse las siguientes recomendaciones:
        - Usar los números de línea (líneas x-y) para identificar las líneas que abarca una estructura e indicar a su lado qué hace.
        - Evitar utilizar conocimiento implícito que no resida en la estructura (valores iniciales, entradas o valores de parámetros).
        - Usar principios generalmente aceptados del dominio de aplicación para mantener la descripción breve y entendible (“búsqueda en profundidad” en lugar de describir lo que hace la búsqueda en profundidad).
    3.3. Especificar el comportamiento de la función entera. Es decir, utilizar la información de los pasos 3.1 y 3.2 para entender qué hace la función y describirlo en texto.

4. Combinar el comportamiento de cada función y las relaciones entre ellas para entender el comportamiento del programa entero. El comportamiento deberá describirse en texto. Esta descripción es la especificación del programa obtenida a partir del código.
5. Comparar la especificación obtenida con la especificación original del programa. Cada punto donde especificación original y especificación generada a partir del código no coincidan es un defecto. Anotar los defectos en una lista.

Veamos un breve ejemplo para entender mejor la técnica. El programa "count" ya lo conoces pues lo has usado para practicar con la técnica de lectura con listas de comprobación. Centrémonos en el siguiente trozo de especificación original:

> Si alguno de los ficheros que se le pasa como argumento no existe, aparece por la salida de error el mensaje de error correspondiente y se continúa procesando el resto de los ficheros.

Que se implementa mediante las siguientes líneas de código entresacadas del programa "count".

```
if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
  fprintf (stdout, "can't open %s\n", argv[i]);
  exit(1)
 }
```

La abstracción de estas líneas sería:

> **Línea 1** Si no hay ficheros que coincidan con el argumento (fp=fopen(argv[i], "r")) == NULL)
>
> **Línea 2** Se imprime por la salida estándar (stdout)el mensaje de que no se puede abrir el fichero (indicando el nombre del fichero)
>
> **Línea 3** Se sale del programa

Que se corresponde con la siguiente descripción:

> Se proporcionan argumentos, pero no hay ficheros con nombres correspondientes a los argumentos. En este caso, el programa se para con un mensaje de error que sale por la salida estándar.

Nótese que las especificaciones no coinciden en algunos puntos. En la Tabla 2 se ve la comparación y aparecen en negrita señaladas las diferencias.

ESPECIFICACIÓN ORIGINAL	ESPECIFICACIÓN OBTENIDA POR ABSTRACCIÓN
Si alguno de los ficheros que se le pasa como argumento no existe, aparece ***por la salida de error*** el mensaje de error correspondiente y **se continúa** procesando el resto de los ficheros.	Se proporcionan argumentos, pero no hay ficheros con nombres correspondientes a los argumentos. En este caso, **el programa se para** con un mensaje de error que sale por la ***salida estándar***.

Tabla 2. Diferencias entre especificación original y especificación abstraida

Por tanto, se detecta la siguiente falta en el código:

> **Falta en línea 2**: La llamada a "fprintf" usa "stdout" en lugar de "stderr".
>
> **Causa fallo**: Los mensajes de error aparecen en la salida estándar (stdout) en lugar de la salida estándar de errores (stderr).

En el Anexo E se muestra la aplicación de esta técnica al programa completo "count". Intenta practicar con este ejercicio realizando tú mismo la abstracción y detección de faltas antes de mirar la solución proporcionada. Además, el Anexo F y el Anexo G contienen dos programas más y su lista de defectos para que el lector se ejercite hasta dominar esta técnica de lectura.

3.4.4.3 LECTURA ACTIVA DE DISEÑO

La **Revisión Activa de Diseño** sólo es aplicable sobre el diseño, y no sobre código o requisitos. Esta técnica propone una cierta variación metodológica al proceso básico de inspección. En concreto requiere que los inspectores tomen un papel más activo del habitual, solicitando que hagan aseveraciones sobre determinadas partes del diseño, en lugar de simplemente señalar defectos. La Revisión Activa de Diseño considera que la inspección debe explotar más la interacción entre autor e inspector, y que la inspección tradicional limita demasiado esta interacción. En esta técnica sólo se definen dos roles: un inspector que tiene la responsabilidad de encontrar defectos, y un diseñador que es el autor del diseño que se esta examinando.

El proceso de la Inspección Activa de Diseño consta de tres pasos. Comienza con una fase de inicio donde el diseñador presenta una visión general del diseño que se pretende inspeccionar y también se establece el calendario. La segunda fase es la de detección, para la cual el autor proporciona un cuestionario para guiar a los inspectores. Las preguntas sólo deben poderse responder tras un estudio detallado y cuidadoso del documento de diseño. Esto es, los inspectores deben elaborar una respuesta, en lugar de simplemente responder sí o no. Las preguntas refuerzan un papel activo de inspección puesto que deben realizar afirmaciones sobre decisiones de diseño. Por ejemplo, se le puede pedir al inspector que escriba un segmento de programa que implemente un diseño particular al inspeccionar un diseño de bajo nivel. El ultimo paso es la recolección de defectos que se realiza en una reunión de inspección. Sin embargo, el meeting se subdivide en pequeñas reuniones especializadas, cada una se concentra en una propiedad de calidad del artefacto. Por ejemplo, comprobar la consistencia entre las asunciones y las funciones, es decir, determinar si las

asunciones son consistentes y detalladas lo suficiente para asegurar que las funciones puedan ser correctamente implementadas y usadas.

3.4.4.4 LECTURA BASADA EN ESCENARIOS

La Lectura Basada en Escenarios proporciona guías al inspector (escenarios que pueden ser preguntas pero también alguna descripción más detallada) sobre cómo realizar el examen del artefacto. Principalmente, un escenario limita la atención de un inspector en la detección de defectos particulares definidos por la guía. Dado que inspectores diferentes pueden usar escenarios distintos, y como cada escenario se centra en diferentes tipos de defectos, se espera que el equipo de inspección resulte más efectivo en su globalidad .Existen dos técnicas de lectura basada en escenarios: Lectura Basada en Defectos y Lectura Basada en Perspectivas. Ambas técnicas examinan documentos de requisitos.

La **Lectura Basada en Defectos** focaliza cada inspector en una clase distinta de defecto mientras inspecciona un documento de requisitos. Contestar a las preguntas planteadas en el escenario ayuda al inspector a encontrar defectos de determinado tipo.

La **Lectura Basada en Perspectiva** establece que un producto software debería inspeccionarse bajo las perspectivas de los distintos participantes en un proyecto de desarrollo. Las perspectivas dependen del papel que los distintos participantes tienen en el proyecto. Para cada perspectiva se definen uno o varios escenarios consistentes en actividades repetibles que un inspector debe realizar y preguntas que el inspector debe responder.

Por ejemplo, diseñar los casos de prueba es una actividad típicamente realizada por el validador. Así pues, un inspector leyendo desde la perspectiva de un validador debe pensar en la obtención de los casos de prueba. Mientras que un inspector ejerciendo la perspectiva del diseñador, deberá leer ese mismo artefacto pensando en que va atener que realizar el diseño.

En el Anexo H, Anexo I y Anexo J se muestran respectivamente las perspectivas del diseñador, validador y usuario respectivamente para que las uses con los requisitos del Anexo A. Cada perspectiva descubre defectos distintos. En dichos anexos te proporcionamos también la solución de qué defectos son encontrados desde cada perspectiva. Finalmente, y a modo de resumen el Anexo K puedes encontrar una tabla con todos los defectos de los requisitos del Anexo A.

4. TÉCNICAS DE EVALUACIÓN DINÁMICA

4.1 CARACTERÍSTICAS Y FASES DE LA PRUEBA

Como se ha indicado anteriormente, a la aplicación de técnicas de evaluación dinámicas se le denomina también *prueba* del software.

La Figura 2 muestra el contexto en el que se realiza la prueba de software. Concretamente la Prueba de software se puede definir como una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas (configuración de la prueba), registrándose los resultados obtenidos. Seguidamente se realiza un proceso de Evaluación en el que los resultados obtenidos se comparan con los resultados esperados para localizar fallos en el software. Estos fallos conducen a un proceso de Depuración en el que es necesario identificar la falta asociada con cada fallo y corregirla, pudiendo dar lugar a una nueva prueba. Como resultado final se puede obtener una determinada Predicción de Fiabilidad, tal como se indicó anteriormente, o un cierto nivel de confianza en el software probado.

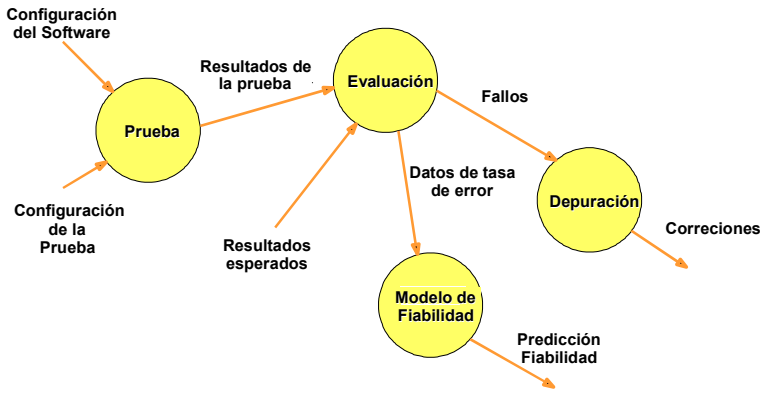

Figura 2. Contexto de la Prueba de Software

El objetivo de las pruebas no es asegurar la ausencia de defectos en un software, únicamente puede demostrar que existen defectos en el software. Nuestro objetivo es pues, diseñar pruebas que sistemáticamente saquen a la luz diferentes clases de errores, haciéndolo con la menor cantidad de tiempo y esfuerzo.

Para ser más eficaces (es decir, con más alta probabilidad de encontrar errores), las pruebas deberían ser realizadas por un equipo independiente al que realizó el software. El ingeniero de software que creó el sistema no es el más adecuado para llevar a cabo las pruebas de dicho software, ya que inconscientemente tratará de demostrar que el software funciona, y no que no lo hace, por lo que la prueba puede tener menos éxito.

Una prueba de software, comparando los resultados obtenidos con los esperados. A continuación se presentan algunas características de una buena prueba:

- Una buena prueba ha de tener una alta probabilidad de encontrar un fallo. Para alcanzar este objetivo el responsable de la prueba debe entender el software e intentar desarrollar una *imagen mental* de cómo podría fallar.

- Una buena prueba debe centrarse en dos objetivos: 1) *probar si el software no hace lo que debe hacer*, y 2) *probar si el software hace lo que no debe hacer*.
- Una buena prueba no debe ser redundante. El tiempo y los recursos son limitados, así que *todas las pruebas deberían tener un propósito diferente*.
- Una buena prueba debería ser la "mejor de la cosecha". Esto es, se debería emplear la prueba que tenga la *más alta probabilidad de descubrir una clase entera de errores*.
- Una buena prueba no debería ser ni demasiado sencilla ni demasiado compleja, pero si se quieren combinar varias pruebas a la vez se pueden enmascarar errores, por lo que en general, *cada prueba debería realizarse separadamente*.

Veamos ahora cuáles son las tareas a realizar para probar un software:

1. *Diseño de las pruebas*. Esto es, identificación de la técnica o técnicas de pruebas que se utilizarán para probar el software. Distintas técnicas de prueba ejercitan diferentes criterios como guía para realizar las pruebas. Seguidamente veremos algunas de estas técnicas.
2. *Generación de los casos de prueba*. Los casos de prueba representan los datos que se utilizarán como entrada para ejecutar el software a probar. Más concretamente los casos de prueba determinan un conjunto de entradas, condiciones de ejecución y resultados esperados para un objetivo particular. Como veremos posteriormente, cada técnica de pruebas proporciona unos criterios distintos para generar estos casos o datos de prueba. Por lo tanto, durante la tarea de generación de casos de prueba, se han de confeccionar los distintos casos de prueba según la técnica o técnicas identificadas previamente. La generación de cada caso de prueba debe ir acompañada del resultado que ha de producir el software al ejecutar dicho caso (como se verá más adelante, esto es necesario para detectar un posible fallo en el programa).
3. *Definición de los procedimientos de la prueba*. Esto es, especificación de cómo se va a llevar a cabo el proceso, quién lo va a realizar, cuándo, …
4. *Ejecución de la prueba,* aplicando los casos de prueba generados previamente e identificando los posibles fallos producidos al comparar los resultados esperados con los resultados obtenidos.
5. *Realización de un informe de la prueba, c*on el resultado de la ejecución de las pruebas, qué casos de prueba pasaron satisfactoriamente, cuáles no, y qué fallos se detectaron.

Tras estas tareas es necesario realizar un proceso de depuración de las faltas asociadas a los fallos identificados. Nosotros nos centraremos en el segundo paso, explicando cómo distintas técnicas de pruebas pueden proporcionar criterios para generar distintos datos de prueba.

4.2 TÉCNICAS DE PRUEBA

Como se ha indicado anteriormente, las técnicas de evaluación dinámica o prueba proporcionan distintos criterios para generar casos de prueba que provoquen fallos en los programas. Estas técnicas se agrupan en:

- *Técnicas de caja blanca o estructurales*, que se basan en un minucioso examen de los detalles procedimentales del código a evaluar, por lo que es necesario conocer la lógica del programa.

– *Técnicas de caja negra o funcionales*, que realizan pruebas sobre la interfaz del programa a probar, entendiendo por interfaz las entradas y salidas de dicho programa. No es necesario conocer la lógica del programa, únicamente la funcionalidad que debe realizar.

La Figura 3 representa gráficamente la filosofía de las pruebas de caja blanca y caja negra. Como se puede observar las pruebas de caja blanca necesitan conocer los detalles procedimentales del código, mientras que las de caja negra únicamente necesitan saber el objetivo o funcionalidad que el código ha de proporcionar.

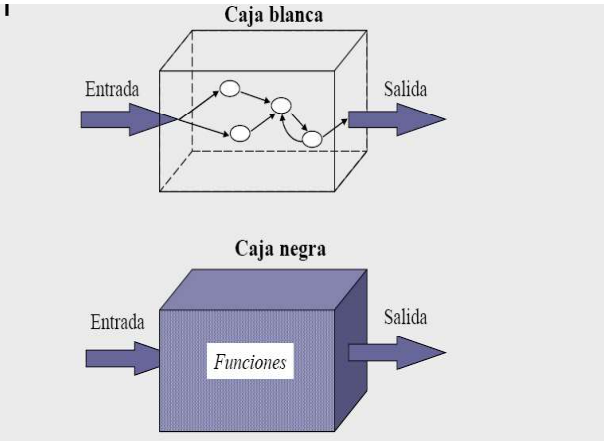

Figura 3. Representación de pruebas de Caja Blanca y Caja Negra

A primera vista parecería que una prueba de caja blanca completa nos llevaría a disponer de un código perfectamente correcto. De hecho esto ocurriría si se han probado todos los posibles caminos por los que puede pasar el flujo de control de un programa. Sin embargo, para programas de cierta envergadura, el número de casos de prueba que habría que generar sería excesivo, nótese que el número de caminos incrementa exponencialmente a medida que el número de sentencias condicionales y bucles aumenta. Sin embargo, este tipo de prueba no se desecha como impracticable. Se pueden elegir y ejercitar ciertos caminos representativos de un programa.

Por su parte, tampoco sería factible en una prueba de caja negra probar todas y cada una de las posibles entradas a un programa, por lo que análogamente a como ocurría con las técnicas de caja blanca, se seleccionan un conjunto representativo de entradas y se generan los correspondientes casos de prueba, con el fin de provocar fallos en los programas.

En realidad estos dos tipos de técnicas son técnicas complementarias que han de aplicarse al realizar una prueba dinámica, ya que pueden ayudar a identificar distintos tipos de faltas en un programa.

A continuación, se describen en detalle los procedimientos propuestos por ambos tipos de técnicas para generar casos de prueba.

4.2.1 PRUEBAS DE CAJA BLANCA O ESTRUCTURALES

A este tipo de técnicas se le conoce también como Técnicas de Caja Transparente o de Cristal. Este método se centra en cómo diseñar los casos de prueba atendiendo al *comportamiento interno y la*

estructura del programa. Se examina así la lógica interna del programa sin considerar los aspectos de rendimiento.

El objetivo de la técnica es diseñar casos de prueba para que se ejecuten, al menos una vez, todas las sentencias del programa, y todas las condiciones tanto en su vertiente verdadera como falsa.

Como se ha indicado ya, puede ser impracticable realizar una prueba exhaustiva de todos los caminos de un programa. Por ello se han definido distintos criterios de cobertura lógica, que permiten decidir qué sentencias o caminos se deben examinar con los casos de prueba. Estos criterios son:

- *Cobertura de Sentencias*: Se escriben casos de prueba suficientes para que cada sentencia en el programa se ejecute, al menos, una vez.
- *Cobertura de Decisión*: Se escriben casos de prueba suficientes para que cada decisión en el programa se ejecute una vez con resultado verdadero y otra con el falso.
- *Cobertura de Condiciones*: Se escriben casos de prueba suficientes para que cada condición en una decisión tenga una vez resultado verdadero y otra falso.
- *Cobertura Decisión/Condición*: Se escriben casos de prueba suficientes para que cada condición en una decisión tome todas las posibles salidas, al menos una vez, y cada decisión tome todas las posibles salidas, al menos una vez.
- *Cobertura de Condición Múltiple*: Se escriben casos de prueba suficientes para que todas las combinaciones posibles de resultados de cada condición se invoquen al menos una vez.
- *Cobertura de Caminos*: Se escriben casos de prueba suficientes para que se ejecuten todos los caminos de un programa. Entendiendo camino como una secuencia de sentencias encadenadas desde la entrada del programa hasta su salida.

Este último criterio es el que se va a estudiar.

4.2.1.1 COBERTURA DE CAMINOS

La aplicación de este criterio de cobertura asegura que los casos de prueba diseñados permiten que todas las sentencias del programa sean ejecutadas al menos una vez y que las condiciones sean probadas tanto para su valor verdadero como falso.

Una de las técnicas empleadas para aplicar este criterio de cobertura es la **Prueba del Camino Básico**. Esta técnica se basa en obtener una medida de la complejidad del diseño procedimental de un programa (o de la lógica del programa). Esta medida es la complejidad ciclomática de McCabe, y representa un límite superior para el número de casos de prueba que se deben realizar para asegurar que se ejecuta cada camino del programa.

Los pasos a realizar para aplicar esta técnica son:

- Representar el programa en un grafo de flujo
- Calcular la complejidad ciclomática
- Determinar el conjunto básico de caminos independientes
- Derivar los casos de prueba que fuerzan la ejecución de cada camino.

A continuación, se detallan cada uno de estos pasos.

4.2.1.1.1 Representar el programa en un grafo de flujo

El grafo de flujo se utiliza para representar flujo de control lógico de un programa. Para ello se utilizan los tres elementos siguientes:

- *Nodos*: representan cero, una o varias sentencias en secuencia. Cada nodo comprende como máximo una sentencia de decisión (bifurcación).
- *Aristas*: líneas que unen dos nodos.
- *Regiones*: áreas delimitadas por aristas y nodos. Cuando se contabilizan las regiones de un programa debe incluirse el área externa como una región más.
- *Nodos predicado*: cuando en una condición aparecen uno o más operadores lógicos (AND, OR, XOR, ...) se crea un nodo distinto por cada una de las condiciones simples. Cada nodo generado de esta forma se denomina nodo predicado. La Figura 4 muestra un ejemplo de condición múltiple.

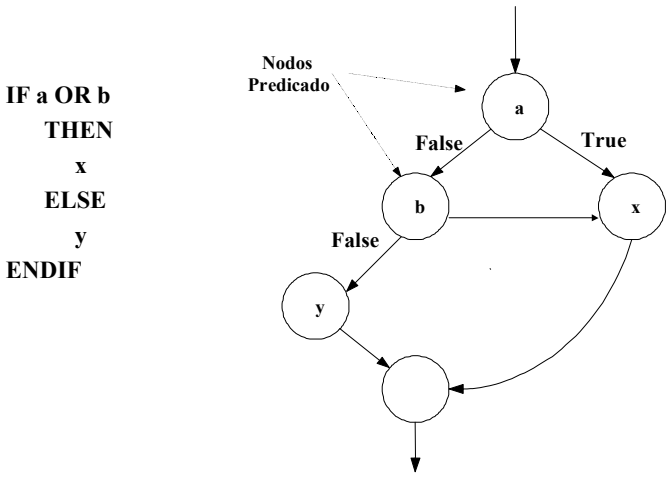

Figura 4. Representación de condición múltiple

Así, cada construcción lógica de un programa tiene una representación. La Figura 5 muestra dichas representaciones.

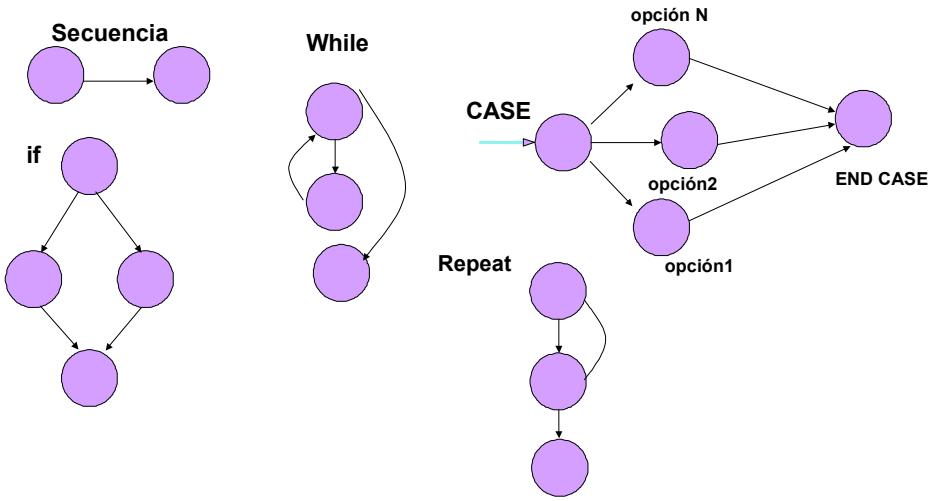

Figura 5. Representación en grafo de flujo de las estructuras lógicas de un programa

La Figura 6 muestra un grafo de flujo del diagrama de módulos correspondiente. Nótese cómo la estructura principal corresponde a un *while*, y dentro del bucle se encuentran anidados dos constructores *if*.

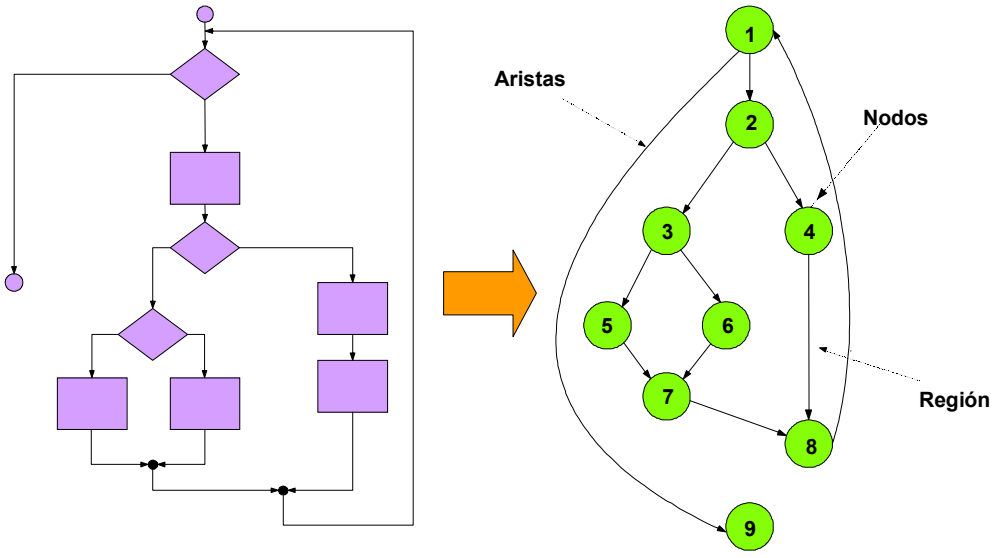

Figura 6. Ejemplo de grafo de flujo correspondiente a un diagrama de módulos

4.2.1.1.2 Calcular la complejidad ciclomática

La complejidad ciclomática es una métrica del software que proporciona una medida cuantitativa de la complejidad lógica de un programa. En el contexto del método de prueba del camino básico, el valor de la complejidad ciclomática define el número de caminos independientes de dicho programa, y por lo tanto, el número de casos de prueba a realizar. Posteriormente veremos cómo se identifican esos caminos, pero primero veamos cómo se puede calcular la complejidad ciclomática a partir de un grafo de flujo, para obtener el número de caminos a identificar.

Existen varias formas de calcular la complejidad ciclomática de un programa a partir de un grafo de flujo:

1. El número de regiones del grafo coincide con la complejidad ciclomática, V(G).
2. La complejidad ciclomática, V(G), de un grafo de flujo G se define como

   V(G) = Aristas – Nodos + 2
3. La complejidad ciclomática, V(G), de un grafo de flujo G se define como

   V(G) = Nodos Predicado + 1

La Figura 7 representa, por ejemplo, las cuatro regiones del grafo de flujo, obteniéndose así la complejidad ciclomática de 4. Análogamente se puede calcular el número de aristas y nodos predicados para confirmar la complejidad ciclomática. Así:

V(G) = Número de regiones = 4

V(G) = Aristas – Nodos + 2 = 11-9 + 2 = 4

V(G) = Nodos Predicado + 1 = 3 +1 = 4

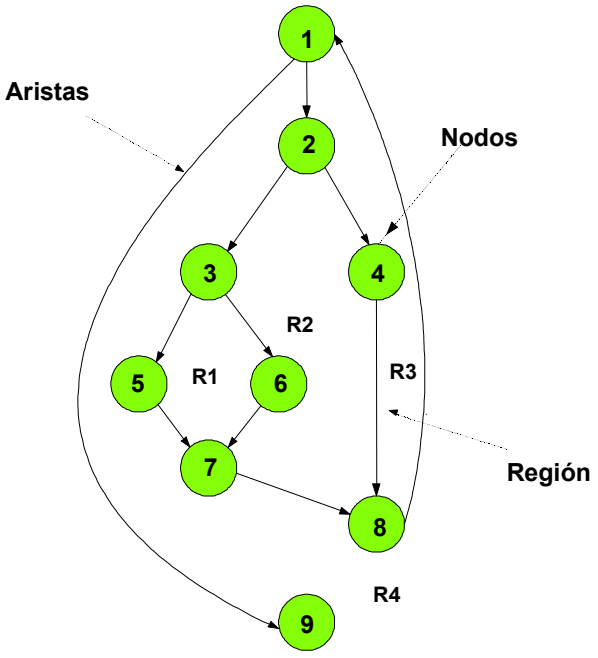

Figura 7. Número de regiones del grafo de flujo

Esta complejidad ciclomática determina el número de casos de prueba que deben ejecutarse para garantizar que todas las sentencias de un programa se han ejecutado al menos una vez, y que cada condición se habrá ejecutado en sus vertientes verdadera y falsa. Veamos ahora, cómo se identifican estos caminos.

4.2.1.1.3 Determinar el conjunto básico de caminos independientes

Un camino independiente es cualquier camino del programa que introduce, por lo menos, un nuevo conjunto de sentencias de proceso o una condición, respecto a los caminos existentes. En términos del diagrama de flujo, un camino independiente está constituido por lo menos por una arista que no haya sido recorrida anteriormente a la definición del camino. En la identificación de los distintos caminos de un programa para probar se debe tener en cuenta que cada nuevo camino debe tener el mínimo número de sentencias nuevas o condiciones nuevas respecto a los que ya existen. De esta manera se intenta que el proceso de depuración sea más sencillo.

El conjunto de caminos independientes de un grafo no es único. No obstante, a continuación, se muestran algunas heurísticas para identificar dichos caminos:

(a) Elegir un *camino principal* que represente una función válida que no sea un tratamiento de error. Debe intentar elegirse el camino que atraviese el máximo número de decisiones en el grafo.

(b) Identificar el segundo camino mediante la localización de la *primera decisión* en el camino de la línea básica alternando su resultado mientras se mantiene el máximo número de decisiones originales del camino inicial.

(c) Identificar un tercer camino, colocando la primera decisión en su valor original a la vez que se altera la *segunda decisión* del camino básico, mientras se intenta mantener el resto de decisiones originales.

(d) Continuar el proceso hasta haber conseguido *tratar todas las decisiones*, intentando mantener como en su origen el resto de ellas.

Este método permite obtener V(G) caminos independientes cubriendo el criterio de cobertura de decisión y sentencia.

Así por ejemplo, para la el grafo de la Figura 7 los cuatro posibles caminos independientes generados serían:

Camino 1: 1-10

Camino 2: 1-2-4-8-1-9

Camino 3: 1-2-3-5-7-8-1-9

Camino 4: 1-2-5-6-7-8-1-9

Estos cuatro caminos constituyen el *camino básico* para el grafo de flujo correspondiente.

4.2.1.1.4 Derivar los casos de prueba que fuerzan la ejecución de cada camino.

El último paso es construir los casos de prueba que fuerzan la ejecución de cada camino. Una forma de representar el conjunto de casos de prueba es como se muestra en la Tabla 3.

Número del Camino	Caso de Prueba	Resultado Esperado

Tabla 3. Posible representación de casos de prueba para pruebas estructurales

En el Anexo L se encuentra un posible ejemplo de pruebas de Caja Blanca para que los alumnos trabajen con él junto con su solución. En el Anexo N se muestra un ejercicio propuesto para que los alumnos se ejerciten en esta técnica de pruebas. El código correspondiente ha sido ya utilizado para la evaluación con técnicas estáticas.

4.2.2 PRUEBAS DE CAJA NEGRA O FUNCIONALES

También conocidas como Pruebas de Comportamiento, estas pruebas se basan en la *especificación* del programa o componente a ser probado para elaborar los casos de prueba. El componente se ve como una "Caja Negra" cuyo comportamiento sólo puede ser determinado estudiando sus entradas y las salidas obtenidas a partir de ellas. No obstante, como el estudio de todas las posibles entradas y salidas de un programa sería impracticable se selecciona un conjunto de ellas sobre las que se realizan las pruebas. Para seleccionar el conjunto de entradas y salidas sobre las que trabajar, hay que tener en cuenta que en todo programa existe un conjunto de entradas que causan un comportamiento erróneo en nuestro sistema, y como consecuencia producen una serie de salidas que revelan la presencia de defectos. Entonces, dado que la prueba exhaustiva es imposible, el objetivo final es pues, encontrar una serie de datos de entrada cuya probabilidad de pertenecer al conjunto de entradas que causan dicho comportamiento erróneo sea lo más alto posible.

Al igual que ocurría con las técnicas de Caja Blanca, para confeccionar los casos de prueba de Caja Negra existen distintos criterios. Algunos de ellos son:

- Particiones de Equivalencia.
- Análisis de Valores Límite.
- Métodos Basados en Grafos.
- Pruebas de Comparación.
- Análisis Causa-Efecto.

De ellas, las técnicas que estudiaremos son las dos primeras, esto es, Particiones de Equivalencia y Análisis de Valores Límite.

4.2.2.1 PARTICIONES DE EQUIVALENCIA

La partición de equivalencia es un método de prueba de Caja Negra que divide el campo de entrada de un programa en *clases de datos* de los que se pueden derivar casos de prueba. La partición equivalente se dirige a una definición de casos de prueba que descubran *clases de errores*, reduciendo así el número total de casos de prueba que hay que desarrollar.

En otras palabras, este método intenta dividir el dominio de entrada de un programa en un número finito de *clases de equivalencia.* De tal modo que se pueda asumir razonablemente que una prueba realizada con un valor representativo de cada clase es equivalente a una prueba realzada con cualquier otro valor de dicha clase. Esto quiere decir que si el caso de prueba correspondiente a una clase de equivalencia detecta un error, el resto de los casos de prueba de dicha clase de equivalencia deben detectar el mismo error. Y viceversa, si un caso de prueba no ha detectado ningún error, es de esperar que ninguno de los casos de prueba correspondientes a la misma clase de equivalencia encuentre ningún error.

El diseño de casos de prueba según esta técnica consta de dos pasos:

1. Identificar las clases de equivalencia.
2. Identificar los casos de prueba.

4.2.2.1.1 Identificar las clases de equivalencia

Una clase de equivalencia representa un conjunto de estados válidos y no válidos para las condiciones de entrada de un programa. Las clases de equivalencia se identifican examinando cada condición de entrada (normalmente una frase en la especificación) y dividiéndola en dos o más grupos. Se definen dos tipos de clases de equivalencia, las *clases de equivalencia válidas*, que representan entradas válidas al programa, y las *clases de equivalencia no válidas*, que representan valores de entrada erróneos. Estas clases se pueden representar en una tabla como la Tabla 4.

Condición Externa	Clases de Equivalencia Válidas	Clases de Equivalencia No Válidas

Tabla 4. Tabla para la identificación de clases de equivalencia

En función de cuál sea la condición de entrada se pueden seguir las siguientes pautas identificar las clases de equivalencia correspondientes:

- Si una condición de entrada especifica un *rango de valores*, identificar una clase de equivalencia válida y dos clases no válidas. Por ejemplo, si un contador puede ir de 1 a 999, la clase válida sería "1 <= contador <= 999. Mientras que las clases no válidas serían "contador < 1" y "contador > 999"
- Si una condición de entrada especifica un *valor* o *número de valores,* identificar una clase válida y dos clases no válidas. Por ejemplo, si tenemos que puede haber desde uno hasta seis propietarios en la vida de un coche. Habrá una clase válida y dos no válidas: "no hay propietarios" y "más de seis propietarios".

- Si una condición de entrada especifica un conjunto de valores de entrada, identificar una clase de equivalencia válida y una no válida. Sin embargo, si hay razones para creer que cada uno de los miembros del conjunto será tratado de distinto modo por el programa, identificar una clase válida por cada miembro y una clase no válida. Por ejemplo, el tipo de un vehículo puede ser: autobús, camión, taxi, coche o moto. Habrá una clase válida por cada tipo de vehículo admitido, y la clase no válida estará formada por otro tipo de vehículo.
- Si una condición de entrada especifica una situación que debe ocurrir, esto es, es lógica, identificar una clase válida y una no válida. Por ejemplo, el primer carácter del identificador debe ser una letra. La clase válida sería "identificador que comienza con letra", y la clase inválida sería "identificador que no comienza con letra".
- En general, si hay alguna razón para creer que los elementos de una clase de equivalencia no se tratan de igual modo por el programa, dividir la clase de equivalencia entre clases de equivalencia más pequeñas para cada tipo de elementos.

4.2.2.1.2 Identificar los casos de prueba

El objetivo es minimizar el número de casos de prueba, así cada caso de prueba debe considerar tantas condiciones de entrada como sea posible. No obstante, es necesario realizar con cierto cuidado los casos de prueba de manera que no se enmascaren faltas. Así, para crear los casos de prueba a partir de las clases de equivalencia se han de seguir los siguientes pasos:

1. Asignar a cada clase de equivalencia un número único.
2. Hasta que todas las clases de equivalencia hayan sido cubiertas por los casos de prueba, se tratará de escribir un caso que cubra tantas clases válidas no incorporadas como sea posible.
3. Hasta que todas las clases de equivalencia no válidas hayan sido cubiertas por casos de prueba, escribir un caso para cubrir una única clase no válida no cubierta.

La razón de cubrir con casos individuales las clases no válidas es que ciertos controles de entrada pueden enmascarar o invalidar otros controles similares. Por ejemplo, si tenemos dos clases válidas: "introducir cantidad entre 1 y 99" y "seguir con letra entre A y Z", el caso *105 1* (dos errores) puede dar como resultado *105 fuera de rango de cantidad*, y no examinar el resto de la entrada no comprobando así la respuesta del sistema ante una posible entrada no válida.

4.2.2.2 ANÁLISIS DE VALORES LÍMITE

La experiencia muestra que los casos de prueba que exploran las condiciones límite producen mejor resultado que aquellos que no lo hacen. Las condicione límite son aquellas que se hayan en los márgenes de la clase de equivalencia, tanto de entrada como de salida. Por ello, se ha desarrollado el *análisis de valores límite* como técnica de prueba. Esta técnica nos lleva a elegir los casos de prueba que ejerciten los valores límite.

Por lo tanto, el análisis de valores límite complementa la técnica de partición de equivalencia de manera que:

- En lugar de seleccionar cualquier caso de prueba de las clases válidas e inválidas, se eligen los casos de prueba en los extremos.

- En lugar de centrase sólo en el dominio de entrada, los casos de prueba se diseñan también considerando el dominio de salida.

Las pautas para desarrollar casos de prueba con esta técnica son:

- Si una condición de entrada especifica un rango de valores, se diseñarán casos de prueba para los dos límites del rango, y otros dos casos para situaciones justo por debajo y por encima de los extremos.
- Si una condición de entrada especifica un número de valores, se diseñan dos casos de prueba para los valores mínimo y máximo, además de otros dos casos de prueba para valores justo por encima del máximo y justo por debajo del mínimo.
- Aplicar las reglas anteriores a los datos de salida.
- Si la entrada o salida de un programa es un conjunto ordenado, habrá que prestar atención a los elementos primero y último del conjunto.

El Anexo M presenta un ejemplo de prueba de caja negra con Particiones de Equivalencia y Análisis de Valores Límite para que los alumnos practiquen con la técnica. En el Anexo O se muestra un ejercicio propuesto para que los alumnos ejerciten. El código correspondiente ha sido ya utilizado para la evaluación con técnicas estáticas.

4.2.3 ESTRATEGIA DE PRUEBAS

La estrategia que se ha de seguir a la hora de evaluar dinámicamente un sistema software debe permitir comenzar por los componentes más simples y más pequeños e ir avanzando progresivamente hasta probar todo el software en su conjunto. Más concretamente, los pasos a seguir son:

1. Pruebas Unitarias. Comienzan con la prueba de cada módulo.
2. Pruebas de Integración. A partir del esquema del diseño, los módulos probados se vuelven a probar combinados para probar sus interfaces.
3. Prueba del Sistema. El software ensamblado totalmente con cualquier componente hardware que requiere se prueba para comprobar que se cumplen los requisitos funcionales.
4. Pruebas de Aceptación. El cliente comprueba que el software funciona según sus expectativas.

4.2.4 PRUEBAS UNITARIAS

La prueba de unidad es la primera fase de las pruebas dinámicas y se realizan sobre cada módulo del software de manera independiente. El objetivo es comprobar que el módulo, entendido como una unidad funcional de un programa independiente, está correctamente codificado. En estas pruebas cada módulo será probado por separado y lo hará, generalmente, la persona que lo creo. En general, un módulo se entiende como un componente software que cumple las siguientes características:

- Debe ser un bloque básico de construcción de programas.
- Debe implementar una función independiente simple.
- Podrá ser probado al cien por cien por separado.
- No deberá tener más de 500 líneas de código.

Tanto las pruebas de caja blanca como las de caja negra han de aplicarse para probar de la manera más completa posible un módulo. Nótese que las pruebas de caja negra (los casos de prueba) se pueden especificar antes de que módulo sea programado, no así las pruebas de caja blanca.

4.2.5 PRUEBAS DE INTEGRACIÓN

Aún cuando los módulos de un programa funcionen bien por separado es necesario probarlos conjuntamente: un módulo puede tener un efecto adverso o inadvertido sobre otro módulo; las subfunciones, cuando se combinan, pueden no producir la función principal deseada; la imprecisión aceptada individuamente puede crecer hasta niveles inaceptables al combinar los módulos; los datos pueden perderse o malinterpretarse entre interfaces, etc.

Por lo tanto, es necesario probar el software ensamblando todos los módulos probados previamente. Ésta es el objetivo de la pruebas de integración.

A menudo hay una tendencia a intentar una integración *no incremental*; es decir, a combinar todos los módulos y probar todo el programa en su conjunto. El resultado puede ser un poco caótico con un gran conjunto de fallos y la consiguiente dificultad para identificar el módulo (o módulos) que los provocó.

En contra, se puede aplicar la integración *incremental* en la que el programa se prueba en pequeñas porciones en las que los fallos son más fáciles de detectar. Existen dos tipos de integración incremental, la denominada *ascendente* y *descendente*. Veamos los pasos a seguir para cada caso:

Integración incremental ascendente:

1. Se combinan los módulos de bajo nivel en grupos que realicen una subfunción específica
2. Se escribe un *controlador* (un programa de control de la prueba) para coordinar la entrada y salida de los casos de prueba.
3. Se prueba el grupo
4. Se eliminan los controladores y se combinan los grupos moviéndose hacia arriba por la estructura del programa.

La Figura 8 muestra este proceso. Concretamente, se forman los grupos 1, 2 y 3 de módulos relacionados, y cada uno de estos grupos se prueba con el controlador C1, C2 y C3 respectivamente. Seguidamente, los grupos 1 y 2 son subordinados de Ma, luego se eliminan los controladores correspondientes y se prueban los grupos directamente con Ma. Análogamente se procede con el grupo 3 eliminando el controlador C3 y probando el grupo directamente con Mb. Tanto Ma y Mb se integran finalmente con el módulo Mc y así sucesivamente.

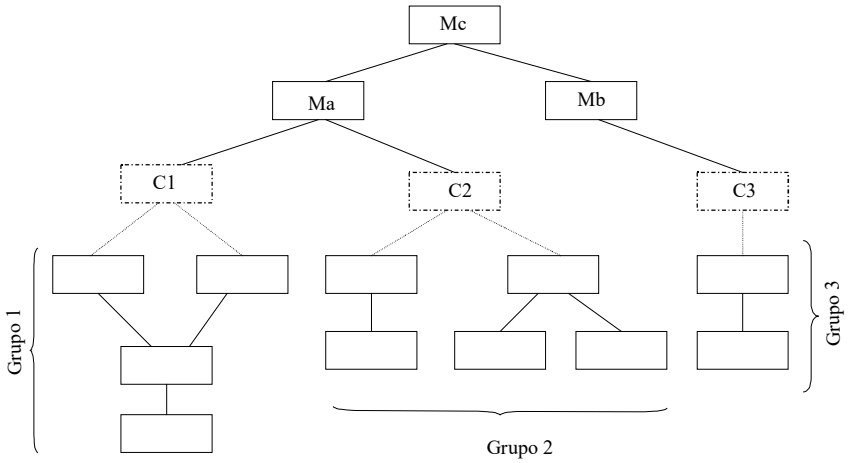

Figura 8. Integración ascendente

Integración incremental descendente:

1. Se usa el módulo de control principal como controlador de la prueba, creando *resguardos* (módulos que simulan el funcionamiento de los módulos que utiliza el que está probando) para todos los módulos directamente subordinados al módulo de control principal.
2. Dependiendo del enfoque e integración elegido (es decir, primero-en-profundidad, o primero-en-anchura) se van sustituyendo uno a uno los resguardos subordinados por los módulos reales.
3. Se llevan a cabo pruebas cada vez que se integra un nuevo módulo.
4. Tras terminar cada conjunto de pruebas, se reemplaza otro resguardo con el módulo real.

La Figura 9 muestra un ejemplo de integración descendiente. Supongamos que se selecciona una integración descendiente por profundidad, y que por ejemplo se prueba M1, M2 y M4. Sería entonces necesario preparar resguardos para M5 y M6, y para M7 y M3. Estos resguardos se ha representado en la figura como R5, R6, R7 y R4 respectivamente. Una vez realizada esta primera prueba se sustituiría R5 por M5, seguidamente R6 por M6, y así sucesivamente hasta probar todos los módulos.

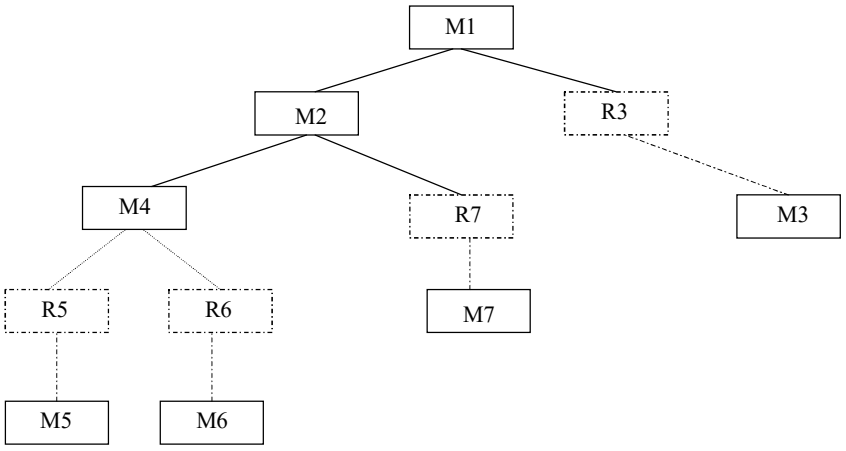

Figura 9. Integración descendiente

Para la generación de casos de prueba de integración, ya sea descendente o ascendente se utilizan técnicas de caja negra.

4.2.6 PRUEBAS DEL SISTEMA

Este tipo de pruebas tiene como propósito ejercitar profundamente el sistema para verificar que se han integrado adecuadamente todos los elementos del sistema (hardware, otro software, etc.) y que realizan las funciones adecuadas. Concretamente se debe comprobar que:

- Se cumplen los requisitos funcionales establecidos.
- El funcionamiento y rendimiento de las interfaces hardware, software y de usuario.
- La adecuación de la documentación de usuario.
- Rendimiento y respuesta en condiciones límite y de sobrecarga.

Para la generación de casos de prueba de sistema se utilizan técnicas de caja negra.

Este tipo de pruebas se suelen hacer inicialmente en el entrono del desarrollador, denominadas *Pruebas Alfa*, y seguidamente en el entrono del cliente denominadas *Pruebas Beta*.

4.2.7 PRUEBAS DE ACEPTACIÓN

A la hora de realizar estas pruebas, el producto está listo para implantarse en el entorno del cliente. El usuario debe ser el que realice las pruebas, ayudado por personas del equipo de pruebas, siendo deseable, que sea el mismo usuario quien aporte los casos de prueba.

Estas pruebas se caracterizan por:

- Participación activa del usuario, que debe ejecutar los casos de prueba ayudado por miembros del equipo de pruebas.
- Están enfocadas a probar los requisitos de usuario, o mejor dicho a demostrar que no se cumplen los requisitos, los criterios de aceptación o el contrato. Si no se consigue demostrar esto el cliente deberá aceptar el producto
- Corresponden a la fase final del proceso de desarrollo de software.

Es muy recomendable que las pruebas de aceptación se realicen en el entorno en que se va a explotar el sistema (incluido el personal que lo maneje). En caso de un producto de interés general, se realizan pruebas con varios usuarios que reportarán sus valoraciones sobre el producto.

Para la generación de casos de prueba de aceptación se utilizan técnicas de caja negra.

4.2.8 PRUEBAS DE REGRESIÓN

La regresión consiste en la repetición selectiva de pruebas para detectar fallos introducidos durante la modificación de un sistema o componente de un sistema. Se efectuarán para comprobar que los cambios no han originado efectos adversos no intencionados o que se siguen cumpliendo los requisitos especificados.

En las pruebas de regresión hay que:

- Probar íntegramente los módulos que se han cambiado.
- Decidir las pruebas a efectuar para los módulos que no han cambiado y que han sido afectados por los cambios producidos.

Este tipo de pruebas ha de realizarse, tanto durante el desarrollo cuando se produzcan cambios en el software, como durante el mantenimiento.

5. PRUEBAS ORIENTADAS A OBJETOS

En las secciones anteriores se ha presentado el proceso de pruebas orientado al concepto general de módulo. Sin embargo, en el caso de la orientación a objetos (OO) es el concepto de clase y objeto el que se utiliza. Veamos a continuación, algunas particularidades de las pruebas para el caso de la OO.

5.1 PRUEBA DE UNIDAD

Al tratar software OO cambia el concepto de unidad. El encapsulamiento dirige la definición de clases y objetos. Esto significa que cada clase e instancia de clase (objeto) empaqueta los atributos (datos) y las operaciones (también conocidas como métodos o servicios) que manipulan estos datos. Por lo tanto, en vez de módulos individuales, la menor unidad a probar es la clase u objeto encapsulado. Una clase puede contener un cierto número de operaciones, y una operación particular puede existir como parte de un número de clases diferentes. Por tanto, el significado de prueba de unidad cambia ampliamente frente al concepto general visto antes.

De esta manera, la *prueba de clases* para el software OO es el equivalente a la prueba de unidad para software convencional. A diferencia de la prueba de unidad del software convencional, la cual tiende a centrarse en el detalle algorítmico de un módulo y los datos que fluyen a lo largo de la interfaz de éste, la prueba de clases para software OO está dirigida por las operaciones encapsuladas en la clase y el estado del comportamiento de la clase. Así, la prueba de una clase debe haber probado mediante las correspondientes técnicas de caja blanca y caja negra el funcionamiento de cada uno de los métodos de dicha clase. Además, se deben haber generado casos de prueba para probar valores representativos de los atributos de dicha clase (esto puede realizarse aplicando la técnica de clases de equivalencia y análisis de valores límite).

5.2 PRUEBA DE INTEGRACIÓN

Debido a que el software OO no tiene una estructura de control jerárquica, las estrategias convencionales de integración ascendente y descendente poseen un significado poco relevante en este contexto.

Generalmente se pueden encontrar dos estrategias diferentes de pruebas de integración en sistemas OO. La primera, *prueba basada en hilos (o threads),* integra el conjunto de clases necesario para responder a una entrada o evento del sistema. Cada hilo se integra y prueba individualmente. El segundo enfoque para la integración, *prueba basada en el uso*. Esta prueba comienza la construcción del sistema integrando y probando aquellas clases (llamadas clases independientes) que usan muy pocas de las clases. Después de probar las clases independientes, se comprueba la próxima capa de clases, llamadas clases dependientes, que usan las clases independientes. Esta secuencia de capas de pruebas de clases dependientes continúa hasta construir el sistema por completo.

Nótese cómo la prueba basada en hilos proporciona una estrategia más ordenada para realizar la prueba que la prueba basada en el uso. Esta prueba basada en hilos, suele aplicarse utilizando los diagramas de secuencia de objetos que diseñan cada evento de entrada al sistema.

Concretamente, se pueden realizar los siguientes pasos para generar casos de prueba a partir de un diagrama de secuencias:

1. Definir el conjunto de secuencias de mensajes a partir del diagrama de secuencia. Cada secuencia ha de comenzar con un mensaje *m* sin predecesor (habitualmente, un mensaje enviado al sistema por un actor), y estará formada por el conjunto de mensajes cuya ejecución dispara *m*.
2. Analizar sub-secuencias de mensajes a partir de posibles caminos condicionales en los diagramas de secuencia.
3. Identificar los casos de prueba que se han de introducir al sistema para que se ejecuten las secuencias de mensajes anteriores, en función de los métodos y las clases afectadas por la secuencia. Tanto valores válidos como inválidos deberían considerarse.

Nótese cómo el conjunto de casos de prueba puede aumentar exponencialmente si se trabaja sobre un sistema OO con un número elevado de interacciones. Por lo tanto, es necesario tener en cuenta este factor a la hora de realizar el diseño.

5.3 PRUEBA DE SISTEMA

En el nivel de prueba del sistema, los detalles de las conexiones entre clases no afectan. El software debe integrarse con los componentes hardware correspondientes y se ha de comprobar el funcionamiento del sistema completo acorde a los requisitos. Como en el caso del software convencional, la validación del software OO se centra en las acciones visibles del usuario y las salidas del sistema reconocibles por éste. Para asistir en la determinación de casos de prueba de sistema, el ejecutor de la prueba debe basarse en los casos de uso que forman parte del modelo de análisis. El caso de uso brinda un escenario que posee una alta probabilidad con errores encubiertos en los requisitos de interacción del cliente. Los métodos convencionales de prueba de caja negra, pueden usarse para dirigir estas pruebas.

5.4 PRUEBA DE ACEPTACIÓN

La prueba de aceptación en un sistema OO es semejante a la prueba de aceptación en un software tradicional. El motivo es que el objetivo de este tipo de prueba es comprobar si el cliente está satisfecho con el producto desarrollado y si este producto cumple con sus expectativas, en términos de los errores que genera y de la funcionalidad que suministra. Al igual que las pruebas convencionales serán los clientes quienes realicen estas pruebas y suministren los casos de prueba correspondientes.

6. HERRAMIENTAS DE PRUEBA

A continuación se muestran algunas herramientas que permiten automatizar en cierta medida el proceso de prueba.

6.1 HERRAMIENTA PARA EL ANÁLISIS ESTÁTICO DE CÓDIGO FUENTE

Jtest

Las características de esta herramienta son las siguientes:

- Jtest comprueba automáticamente la construcción del código fuente ("pruebas de caja blanca"), la funcionalidad del código ("pruebas de caja negra"), y mantiene la integridad del código (pruebas de regresión).
- Se aplica sobre clases Herram Java y JSP.

6.2 HERRAMIENTAS PARA PRUEBAS DE CARGA Y STRESS

OpenLoad Tester

Las características de esta herramienta son las siguientes:

- OpenLoad Tester es una herramienta de optimización de rendimiento basada en navegador para pruebas de carga y stress de sitios web dinámicos.
- Permite elaborar escenarios de ejecución y ejecutarlos de forma repetida, simulando la carga de un entorno de producción real de nuestra aplicación como si múltiples usuarios estuvieran usándola

Benchmark Factory

Las características de esta herramienta son las siguientes:

- Benchmark Factory es una herramienta de prueba de carga y capacity planning, capaz de simular el acceso de miles de usuarios a sus servidores de bases de datos, archivos, internet y correo, localizando los posibles cuellos de botella y aislar los problemas relacionados con sobrecargas del sistema

6.3 HERRAMIENTA PARA LA AUTOMATIZACIÓN DE LAS PRUEBAS FUNCIONALES

DataFactory

Las características de esta herramienta son las siguientes:

- DataFactory, ayuda a la creación automática de juegos de ensayo o casos de prueba basados en la funcionalidad de las aplicaciones (casos de uso), que facilitan la labor de tener que crearlos manualmente y típicamente, se utilizan junto a las herramientas de pruebas de carga.

6.4 HERRAMIENTAS DE DIAGNÓSTICO

PerformaSure

Las características de esta herramienta son las siguientes:

- PerfomaSure, es una herramienta de diagnóstico de rendimiento para el análisis en entornos distribuidos J2EE, que permite el seguimiento de los problemas de rendimiento detectados en tiempo real, desde la transacción del usuario final en fase de producción, hasta la línea de código fuente que genera el problema.

Spotlight

Las características de esta herramienta son las siguientes:

- Spotlight, en sus versiones para Servidores de Aplicaciones, Servidores Web, Bases de datos, Sistemas Operativos, etc. es la herramienta visual para la detección en tiempo real, de los cuellos de botella en estos componentes. Una vez que identifica la causa de estos problemas, proporciona la información y consejos necesarios, para su resolución.

6.5 HERRAMIENTA DE RESOLUCIÓN Y AFINADO

JProbe

- Esta herramienta, permite detectar los 'puntos calientes' de los componentes de una aplicación JAVA, tales como el uso de la memoria, el uso del CPU, los hilos de ejecución,... y a partir de ellos, bajar al nivel del código fuente que los provoca ofreciendo una serie de consejos o buenas prácticas de codificación para la resolución del problema.