Universidad de Valladolid

E. U. DE INFORMÁTICA (SEGOVIA)

Ingeniería Técnica en Informática de Gestión

Modelos de Fiabilidad del Software

Alumno: Jesús Plaza Rubio

Tutores: Pablo Fernández Blanco

Pilar Grande González

ÍNDICE DE CONTENIDOS

1. VISIÓN GENERAL DEL DOCUMENTO

Según la Real Academia Española fiabilidad significa “Probabilidad de buen funcionamiento de algo” por tanto fiabilidad del software significa “Probabilidad de buen funcionamiento de un sistema informático”.

Hoy en día los sistemas informáticos están totalmente integrados en la sociedad en la que vivimos. A todos nos extrañaría ver una fábrica sin máquinas ni procesos automatizados, una empresa, independientemente de su tamaño, sin una gestión informatizada, un sistema de control de vuelos manual o incluso, una tienda sin un sistema de gestión de inventario.

La informática (o también denominada tecnologías de la información) ha evolucionado a un ritmo vertiginoso convirtiéndose, no sólo en imprescindibles en nuestra vida, sino en un elemento altamente potente y a la vez peligroso si no está en control. Este hecho requiere necesariamente una evolución de los informáticos no siendo ya suficiente el realizar un sistema eficiente, rápido, sencillo para el usuario sino que tiene que ser un sistema a prueba de fallos.

Pero, ¿realmente es necesario que todos los sistemas que se generen sean a prueba de fallos? ¿Tiene la misma consecuencia que genere resultados incorrectos un software de gestión de inventario que un software que controla el tráfico aéreo? Y, ¿cómo se mide la tolerancia a fallos de un sistema informático?

El propósito de este proyecto es indagar en los métodos actuales que hay para medir lo fiable que es un software, ponerlo en práctica partiendo de un software de prueba para poder determinar cuál sería el más apropiado para medir la fiabilidad de este tipo de software.

Por todo lo anteriormente expuesto, se va a realizar un análisis en lo referente a fiabilidad del software, esto es:

- Qué se entiende por fiabilidad del software.
- Los modelos existentes para analizar dicha fiabilidad.
- A qué escenarios son aplicables, según la situación:
    - Etapa de desarrollo.
    - Pruebas.
    - Explotación de dicho software.

Gracias a estos modelos se podrá evaluar la evolución y/o estado de dicha fiabilidad, además, de poder predecir de esta manera los posibles problemas, fallos, errores, etc, que se pueden dar a lo largo de la vida útil de dicho programa.

Se han examinado las hipótesis y los fundamentos matemáticos de los modelos más característicos en uso hoy día, estableciéndose un esquema para su clasificación sistemática.

En resumen, se van a estudiar modelos de predicción, modelos de estimación clásica, y modelos de estimación bayesianos.

En un segundo apartado se expondrán las métricas del software, qué miden, cual es su pretensión y cuales son aplicadas exclusivamente a la fiabilidad.

Por último, se estudiará un caso práctico, en donde se han aplicado estos modelos, llegando a una serie de conclusiones acerca de utilizar uno u otro modelo según el caso.

2. OBJETIVOS

Como se ha comentado en la introducción, el objeto de este proyecto es el de realizar un análisis de los métodos que se pueden emplear actualmente en la evaluación de la fiabilidad de software, y como resultado final, presentar un estudio sobre un caso práctico aplicando estos diferentes modelos para poder discernir cual puede ser el mejor método a utilizar y los diferentes motivos que nos llevan a ello.

Los objetivos que se pretenden cubrir en este proyecto son los siguientes:

1. Definir el concepto de "fiabilidad" del software.

2. Definir el proceso de desarrollo de software y las diferentes etapas que lo componen.

3. Definir cada uno de los métodos existentes (predictivo y de estimación) que se pueden emplear actualmente en la evaluación de la fiabilidad de software.

4. Definir cada uno de los modelos que componen los diferentes métodos (predictivo y de estimación).

5. Presentar un estudio sobre un caso práctico aplicando todos los modelos incluidos en el presente documento.

6. Analizar los resultados obtenidos con cada uno de los modelos y, en base a dicho análisis, identificar el modelo más adecuado para el estudio presentado.

7. Presentar las ventajas e inconvenientes de cada uno de los modelos estudiados agrupados en su correspondiente método.

8. Establecer recomendaciones de uso de los diferentes modelos existentes en base a las diferentes etapas del desarrollo de software.

3. FIABILIDAD DEL SOFTWARE

3.1 - DEFINICIONES

Generalmente, para empezar a evaluar la fiabilidad de un sistema software, es necesario primero evaluar los diferentes módulos/elementos por los que esta compuesto.

Por tanto, es necesario empezar descomponiendo el sistema software en los diferentes módulos que lo componen hasta un nivel en que se pueda definir el nivel de fiabilidad que queremos alcanzar en cada uno de ellos, y en ultima instancia, evaluar la fiabilidad de la totalidad del sistema en base a los resultados obtenido anteriormente.

No es necesario llegar a un nivel de detalle extremo, solo dividirlo en diferentes subsistemas con un cierto nivel de precisión.

Los elementos que componen un sistema software se identifican de acuerdo a la función específica que realizan, siendo la representación del sistema software un diagrama lógico del mismo modo que indica la interacción existente entre sus distintas partes funcionales ó elementos. El diagrama de bloques de fiabilidad y el diagrama de árboles de fallo son los que se utilizan más frecuentemente.

3.1.1 - DEFINICIÓN DE FIABILIDAD

Dado que uno de los objetivos del proyecto es el estudio de los diferentes modelos que se utilizan para medir la fiabilidad del software, y en particular, la revisión de un caso concreto que permita cuantificar dicha fiabilidad durante sus periodos de prueba.
Por todos estos motivos es imprescindible dar una definición para la fiabilidad del software.

La IEEE [4] define la fiabilidad como "*la habilidad que tiene un sistema o componente de realizar sus funciones requeridas bajo condiciones específicas en periodos de tiempo determinados*".

La probabilidad es una función que depende de las variables de entrada al sistema, del uso del sistema, y también de la existencia de defectos en el software. Las variables de entrada al sistema determinan si se activan, en caso de existir, defectos del software.

Otros autores como Musa [6] dan definiciones parecidas: "*la probabilidad o la capacidad de que un sistema funcione sin fallos en un periodo de tiempo y bajo condiciones o un medio ambiente también específico*".

Por tanto, lo que se pretende predecir es el momento en que se va a producir el siguiente fallo, basándonos en el estudio de los momentos en que se produjo el último fallo de dicho software.

Si se compara la definición de fiabilidad de un software, con la definición de fiabilidad de un sistema físico o hardware, puede observarse que es específica del primer caso la importancia del instante en el cual ocurre el último fallo, siendo igualmente necesario conocer la distribución de probabilidad de fallo. Dicha distribución depende de una gran variedad de factores, y es específica de cada sistema.

La probabilidad de fallo de un software depende fundamentalmente de dos factores: la existencia de errores en el código o la existencia de variables de entrada incorrectas que el sistema no espera.
Precisamente, una de las características mas importantes a la hora de definir la robustez y fiabilidad de un sistema software es comprobando que sigue realizando su función a pesar de recibir valores erróneos en sus ejecución.
Lo que un sistema puede recibir como valor correcto o no se suele determinar a la hora de realizar su toma de requerimientos y quedan definidos en el perfil operacional del sistema. Es decir, depende de los errores humanos cometidos durante el diseño y desarrollo del software.

3.1.2 - DEFINICIÓN DE PROGRAMA

Un programa es un conjunto de instrucciones u órdenes que indican a la máquina las operaciones que ésta debe realizar con unos datos determinados. En general, todo programa indica a la computadora cómo obtener unos datos de salida, a partir de unos datos de entrada.
Este programa, además, puede ejecutarse de manera independiente del resto del sistema software en el que se encuentra.

3.1.3 - DEFINICIÓN DE PERFIL OPERACIONAL

A la hora de realizar la toma de requerimientos de un programa informático, se define lo que se conoce como el perfil operacional de un programa, que no es más que el conjunto de posibles valores de las variables de entrada al programa, y su probabilidad de ocurrencia.

Tal y como explica la figura 3.1 [1], hay un conjunto definido con todos los posibles valores de entrada que el programa P esta esperando.

Dicho programa, realizará una conversión de estas variables de entrada transformándolos en el conjunto O de valores de salida.

Mientras el programa se ejecute correctamente, el programa P siempre transformará los valores I por los valores O según sea la misión para la que se haya diseñado dicho programa P.

Cuando no se obtienen los valores esperados, hay un subconjunto de valores de variables de entrada IF del conjunto I para los cuales el programa falla y se obtienen los valores de salida OF del conjunto O, que no son los adecuados.

Por tanto, existe un subconjunto IF que contiene todos los valores que entrada que hacen que el programa P falle.
Dado que el programa P falla, éste debe corregirse con el fin de eliminar el subconjunto IF, de manera que solamente se obtengan valores correctos de las variables de salida. Cada vez que se efectúa una corrección del programa P lo que realmente se hace es sustituirlo por otro programa P', que pretende modificar el programa P para que el

subconjunto IF sea lo más pequeño posible o desaparezca, siendo necesario muchas veces realizar modificaciones sucesivas hasta su eliminación completa.

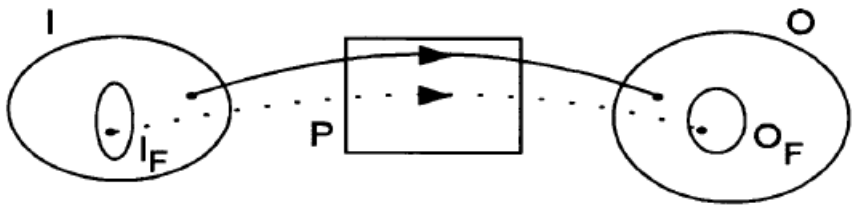


Figura 3.1 Funcionamiento básico de un programa

3.1.4 - DEFINICIÓN DE FALLO

Cuando estamos hablando de software y una vez visto las anteriores definiciones, podemos decir que un fallo se produce cuando al ejecutarse el programa P no se obtienen las variables de salida especificadas en el conjunto O. Es imprescindible por tanto, que el sistema esté ejecutándose para que ocurra un fallo.

Otra definición que se puede dar al fallo es el siguiente [3]:

"La incapacidad de un sistema o de alguno de sus componentes para realizar las funciones requeridas dentro de los requisitos de rendimiento especificados."

Lo que tenga consideración de fallo o no debe determinarlo el perfil operacional que se haya establecido a la hora de diseñar el programa P, puesto que dependiendo de la criticidad del sistema, podemos considerar como fallo la parada del sistema software o la más minima variación en las variables de salida esperadas.

En todos los proyectos de Ingeniería del Software, existe un el cliente con una serie de ideas acerca de cómo quiere un determinado programa software. A este cliente se le realiza una toma de requerimientos funcionales que quiere que tenga el programa, y como consecuencia de ello, este mismo cliente es quien generalmente debe establecer cuales son los límites en los que una determinada ejecución puede ser considerada fallo o no.

Por este motivo, una toma de requerimientos mal realizada, puede acabar en una discusión sobre lo que puede considerarse fallo o no y por tanto ser susceptible de corrección.

Según sea la función del software, necesitaremos definir el impacto de cada uno de los fallos. Evidentemente, para sistemas críticos como el utilizado en proyectos militares, tráfico aéreo .....etc., un fallo de software, por pequeño que sea estará clasificado como grave o muy grave, mientras que en otros con menores exigencias de fiabilidad podría ser leve.

Existen por lo menos tres criterios de clasificación de la severidad de los fallos [2]: el impacto económico, la seguridad humana, y la calidad de servicio.

En los sistemas dirigidos al mundo de los negocios, es decir a los bancos, la bolsa, etc.. es de interés primordial la clasificación de la severidad de los fallos de acuerdo a su impacto económico. Es decir, qué coste supone un fallo determinado en términos de lo que cuesta reparar sus daños, como pérdidas del negocio u otras alteraciones.

La clasificación de la severidad de los fallos en términos de su impacto en la seguridad de vidas humanas se utiliza para los sistemas nucleares, militares, ó a para cualquier aplicación en que la seguridad sea de máxima prioridad.

Por último, clasificar la severidad de los fallos de un software de acuerdo a su impacto en la calidad de servicio del sistema, es típico de los sistemas en que los usuarios interaccionan directamente con ellos, como por ejemplo servicios de información, telefónicos, venta, etc..

La severidad de los fallos se mide generalmente según la siguiente escala [2]:

1. ***Catastrófico.*** Un fallo es catastrófico cuando sus consecuencias son graves e irreversibles.

2. ***Crítico.*** Un fallo crítico es el que impide que se realice la función del sistema durante un tiempo inaceptable, ó un fallo que genere pérdida de datos.

3. ***Moderado.*** Un fallo es moderado cuando la función del sistema se suspende durante un cierto tiempo no crítico, ó cuando la función se realiza de manera parcial.

4. ***Leve****.* Un fallo leve es el que permite que el sistema siga realizando plenamente su función.

Para cuantificar la fiabilidad de un sistema es necesario indicar cómo se mide la ocurrencia de fallos. Existen cuatro maneras de caracterizarla:

1. Instante de tiempo en que ocurre el fallo.

2. Intervalo de tiempo entre fallos.

3. Número acumulado de fallos hasta un instante de tiempo dado.

4. Número de fallos durante un intervalo de tiempo.

Siendo las cuatro cantidades variables aleatorias, es decir que pueden tomar más de un valor, dentro de un rango de posibles valores, cada uno asociado con una probabilidad.

3.1.5 - DEFINICIÓN DE DEFECTO

Un defecto de un programa provoca que el programa no cumpla de manera efectiva la misión con la que se creo. Un defecto es, por tanto, una característica concreta y objetiva, se puede identificar, describir y contabilizar. Los defectos son consecuencia de errores humanos en las distintas fases del ciclo de vida de software y tienen un coste asociado.

Definiciones admitidas como defecto son las siguientes:

«Un paso, proceso o definición de dato incorrecto en un programa de computadora. El resultado de una equivocación.» [4]

«Una variación de una característica deseada del producto.» [5]

De acuerdo con [5], un defecto puede dividirse en dos categorías:

- Defectos en la especificación del producto: el producto construido varía del producto especificado.
- Variaciones en las expectativas del cliente/usuario: estas variaciones son algo que el usuario quiere que no esté en el producto final, pero éstas además no fueron especificadas para ser incluidas en el producto.

Para poder detectar los defectos que contiene un programa hay dos posibles caminos. Uno directo siguiendo procedimientos que no implican la ejecución del programa:

- Revisión de requisitos.
- Revisión del diseño.
- Revisión del código.
- Utilización de diagnósticos de compilación.

Los defectos se detectan de forma indirecta como consecuencia de la ocurrencia de fallos. Los síntomas del fallo pueden sugerir las áreas de código donde sea más probable que se encuentre el defecto que lo ha ocasionado.

Estos son los principales tipos de defecto que suele haber:

1. Documentación Sintaxis.
2. Organización (gestión de cambios, librerías, control de versiones).
3. Asignación (declaraciones, ámbitos, nombres duplicados).
4. Interfaz (dentro del mismo sistema o con otros externos).

5. Chequeo (mensajes de error, trazas, validaciones).
6. Datos (estructura, contenido).
7. Función (errores lógicos, bucles, recursividad).
8. Sistema (configuración, instalación, explotación).
9. Entorno (diseño, compilación, pruebas).

3.1.6 - DEFINICIÓN DE ERROR

El error puede ser definido como [3]:

"*La diferencia entre un valor calculado, observado o medido y el valor verdadero, especificado o teóricamente correcto.*".

De las definiciones de fallo, defecto y error, se deduce que un software falla porque durante las distintas fases de su ciclo de vida, los equipos técnicos cometen errores que dan origen a la presencia de defectos, y éstos a su vez causan fallos durante la operación del sistema.

Los errores se deben a una gran variedad de causas, pudiendo agruparse la mayoría en alguna de las siguientes categorías [6]:

1. Comunicación.
2. Conocimientos.
3. Análisis incompleto.
4. Transcripción.

Los errores de comunicación son probablemente los más numerosos, y su proporción aumenta considerablemente con el tamaño del proyecto.

Esto se debe a que hay muchos tipos de comunicación durante el desarrollo de un software. Existe la comunicación entre personas que realizan distintas funciones en el proyecto: Cliente-diseñador, diseñador-programador, programador-equipo de pruebas, y la comunicación entre las personas que realizan el mismo tipo de función: diseñadores, programadores, y técnicos que integran el equipo de pruebas.

Los errores de conocimiento se deben a la inexperiencia de los técnicos involucrados en el proyecto. La fallo de conocimiento puede ser sobre el área de aplicación del producto que se desarrolla, sobre las metodologías de diseño, ó sobre el lenguaje de programación utilizado.

También pueden cometerse errores como resultado de un análisis incompleto del tema que se pretende abordar, no considerando todas las posibles condiciones que pueden ocurrir en un punto determinado del programa.

Los errores se pueden cometer **cuando** se traslada información entre la mente humana y el papel ó la máquina, ó al revés, es decir cuando la mente humana interpreta información escrita.

La siguiente figura muestra la relación entre error, defecto y fallo [3]:

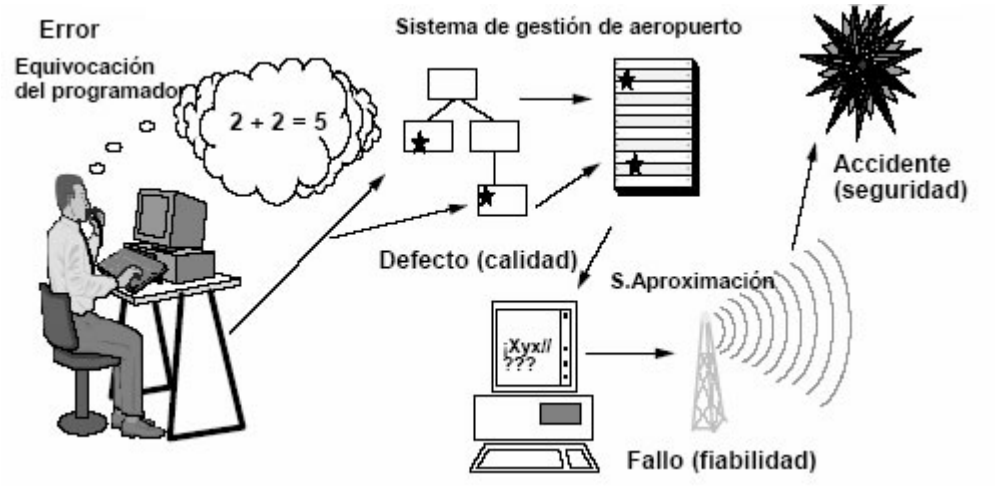


Figura 3.2: Relación entre error, defecto y fallo.

3.1.7 - DEFINICIÓN DE PROCESO DE FALLO

Según se ha indicado en los apartados anteriores, durante las distintas fases del ciclo de vida de un software cualquiera se cometen errores que tienen como consecuencia la presencia de defectos en el sistema, y dichos defectos se manifiestan durante su operación en forma de fallos.

Un fallo ocurre cuando el programa deba ejecutarse pasando por una zona en la que hay algún defecto, y además los valores asignados a las variables de entrada permiten que dicho defecto se manifieste.

Por tanto los fallos ocurren de manera aleatoria. Antes de comenzar la operación de un software se tiene una familia de variables aleatorias Ti, T2....Tn, que representa el tiempo hasta el primer fallo, segundo fallo, n-ésimo fallo del sistema, ó de variables aleatorias T'j, T'2, T'n^ que representa el tiempo del primer intervalo entre fallos, segundo intervalo entre fallos. ,n-ésimo intervalo entre fallos.

Los tiempos son variables aleatorias ya que no se conoce qué defectos tiene el programa, ni cuales son los que se van a manifestar en un momento dado, por tanto no se sabe cuando va a ocurrir un fallo del sistema.

Cuando se somete un software ó sus componentes a una fase de pruebas, lo que se hace es permitir el funcionamiento del programa hasta que se produce un fallo, en ese caso se analiza el fallo y se intenta corregirlo. Una vez que se corrige el motivo de fallo, ó que supuestamente se ha corregido, se vuelve a poner el programa en operación hasta que se produce un nuevo fallo. Este proceso se repite continuamente hasta que se estime que ya se han eliminado todos los defectos, ó que el número de defectos que aún quedan está por debajo del umbral establecido.

El resultado de las acciones que se toman para detectar y corregir el defecto que ha ocasionado el fallo de un programa no es siempre satisfactorio, puede ocurrir que no se detecte el origen del fallo, ó que al intentar corregir el defecto se cometa un nuevo error.

En consecuencia pueden existir tres posibles resultados:

1. Disminuye en uno el número de defectos que contiene el programa, ya que se ha corregido el que ha causado el último fallo.

2. El número de defectos no varía ya que no se ha conseguido detectar o corregir el motivo del último fallo.

3. Aumenta en uno el número de defectos que contiene el programa, ya que al intentar corregir el que ha causado el último fallo se ha cometido un nuevo error.

Normalmente la probabilidad de que la corrección sea perfecta es mayor que la probabilidad de efectuar una corrección imperfecta – segundo caso -, o de que se cometa un nuevo error durante la corrección, por tanto lo habitual es que la fiabilidad de un programa aumente durante su periodo de pruebas, aunque la rapidez del crecimiento de la fiabilidad dependerá del número y tipo de las correcciones que se efectúen.

Según se avanza en el proceso de pruebas, se van obteniendo datos concretos en cuanto a tiempos de fallo, es decir se dispondrá de los valores tj, *t^,* , tj,, - ó t'j, t'j, , t'k "» correspondientes a los tiempos en que se han producido k fallos del sistema - realización muestral - .

Valores que pueden utilizarse para estimar la fiabilidad del programa, el número de posibles defectos aún presentes en el programa, y el tiempo adicional que deben durar las pruebas.

Las medidas de tiempo se pueden realizar de tres formas diferentes:

- ***Tiempo de ejecución-->*** Cuando se mide el tiempo de ejecución lo que realmente se está midiendo es el tiempo que ha empleado el procesador en ejecutar instrucciones del programa hasta el instante en que se produce el fallo.

- ***Tiempo de calendario-->*** En este caso se cuenta la totalidad de tiempo transcurrido hasta la ocurrencia del fallo en la medida de tiempo que habitualmente se utiliza para las actividades cotidianas. Por tanto en este

caso también se cuenta el tiempo que transcurre, aunque el programa no esté en operación.

- ***Tiempo de operación**:* En este caso se mide el tiempo de reloj que utiliza el programa durante su ejecución hasta que se produce el fallo. Incluye los tiempos de espera y los tiempos que el procesador dedica a la ejecución de otros programas antes de concluir la ejecución del programa de interés.

La ventaja de utilizar como medida de tiempo el tiempo de ejecución, es que se trata de una medida uniforme que tiene en cuenta el hecho de que el esfuerzo dedicado a pruebas no es el mismo a lo largo de una fase dada. La desventaja es que no se tiene en cuenta el tiempo que se necesita para corregir los defectos, es decir el tiempo que hay que invertir revisando código, aislando problemas y cambiando ó añadiendo código.

El tiempo de calendario es la medida más sencilla de utilizar, pero es una medida que no tiene en cuenta el hecho de que unas maneras de realizar las pruebas son más eficientes que otras, y también que unas personas son más eficientes realizando las pruebas y codificando que otras.

El tiempo de operación tampoco tiene en cuenta las posibles diferencias de eficiencia en las distintas formas de realizar las pruebas, y tampoco considera el tiempo invertido en el trabajo necesario para la corrección de los defectos.

3.1.8 - DEFINICIÓN DE CALIDAD DEL SOFTWARE

"Que el desarrollo del software sea un éxito implica mucho más que escribir código. La mayoría de las organizaciones de desarrollo del software comprenden ahora que el éxito de los procesos depende de la organización y estructura de la empresa, el control de proyectos y los procedimientos y estándares software".
(Dutta, Lee, & Van Wassenhove, 1999)

Actualmente existen un interés y una sensibilidad especiales por la aspiración a la calidad en todos los procesos de producción y en la actividad empresarial tal y como refleja la frase anterior. Se ha aceptado la Calidad Total como un instrumento de mejora de la competitividad, y en consecuencia se ha iniciado un cambio de actitud y de comportamiento dirigido a obtener una mejora continua de la calidad en el trabajo.

La calidad de un software se puede definir como [7] la concordancia con los requisitos funcionales debidamente establecidos, con los estándares de desarrollo explícitamente documentados y con las características implícitas que se espera que todo software desarrollado profesionalmente.

Los requisitos de software son la base de la medida de calidad. Si no se cumple con los requisitos establecidos, no será un software de calidad.
Los estándares especificados definen un conjunto de criterios de desarrollo que guían la forma en que se aplica la ingeniería de software.

Existe un conjunto de requisitos implícitos que no se mencionan, pero que están presentes en todo desarrollo profesional, como el buen mantenimiento o la facilidad de uso.

De todas formas, profundizaremos sobre la Calidad del Software y la manera de medirlo (métricas) en un capitulo posterior.

3.2 - ESTADO DEL ARTE DE LA MATERIA

El hardware, al contrario que un sistema software, es mucho más fiable, y por tanto, los fallos suelen provenir casi en su totalidad, por el software instalado. Esto es debido a que la fiabilidad del hardware ha sido estudiada durante muchos años antes de que empezara a tener repercusión el estudio de la fiabilidad del software.

Por otro lado, decir que debido a la complejidad de los actuales sistemas informáticos, el desarrollo del software no es nada fácil, haciendo necesario en muchas ocasiones proyectos con decenas de miles de líneas de códigos. No se puede programar sin más, es necesario analizar lo que tenemos que hacer, cómo lo vamos a hacer y cómo se van a coordinar las distintas personas que intervienen en el proyecto para llegar a obtener los resultados inicialmente esperados.

A medida que los sistemas software se van haciendo más y más complejos, su fiabilidad es más critica e importante, por lo que cada vez se dedican más recursos para conseguir sistemas de información con un grado de fiabilidad elevado.

Por lo visto anteriormente, podemos decir que el software, es el componente cuyo desarrollo presenta mayores dificultades, ya sea por su planificación, el no cumplimiento de las estimaciones de costes iniciales, etcétera. Pero todo es debido a que es una actividad reciente si la comparamos con otras actividades de ingeniería, y aún es más reciente la disciplina que estudia su fiabilidad dentro de las diferentes etapas del desarrollo de estos sistemas de software.

La fiabilidad del software requiere un estudio más específico para conseguir su fiabilidad, ya que dadas sus características tan especiales (un software se desarrolla, no se fabrica), que los diferencian considerablemente de otros tipos de productos, es más problemático y costoso conseguir que un software cualquiera tenga un grado de fiabilidad elevado.

Por ejemplo, dentro de un sistema software, hay fallos que no solo perduran en el tiempo, sino que su propia corrección puede llegar a veces a provocar otro tipo de fallos y a empeorar su fiabilidad.

El origen de los fallos de un software no es el desgaste que sufre el sistema como consecuencia de su funcionamiento, como es el caso de los sistemas físicos, sino los errores humanos cometidos durante el diseño y desarrollo del sistema. La razón fundamental para que el desarrollo de software sea tan crítico es su propia complejidad conceptual.

Las características propias del proceso de desarrollo del software hacen que el resultado no sea un producto sencillo. Es un proceso que se caracteriza por la dificultad en detectar los errores que se van cometiendo durante sus distintas fases. Los errores cometidos por los equipos técnicos a lo largo del desarrollo dan lugar a la presencia de defectos en el sistema que no son detectados hasta que determinadas condiciones de operación hacen que se manifiesten.

Es necesario, por tanto, efectuar pruebas del software, durante las cuales se intenta simular el máximo número posible de situaciones que pueden originar la activación de los defectos existentes. Ni es posible cubrir en un tiempo razonable todo el abanico de posibilidades, ni el coste sería asumible. Por este motivo es necesario especificar un conjunto de pruebas que garanticen la detección del máximo número posible de defectos, de una manera eficiente. Lo habitual es terminar cada fase del proceso de desarrollo con unas pruebas específicas para dicha fase con el fin de pasar de una fase a la siguiente con el mínimo número de defectos. Los defectos que siguen estando presentes según avanza el proceso de desarrollo son los más difíciles de detectar.

La figura siguiente muestra donde se suelen encontrar el grueso de los errores de un sistema software según el momento en el que nos encontremos:

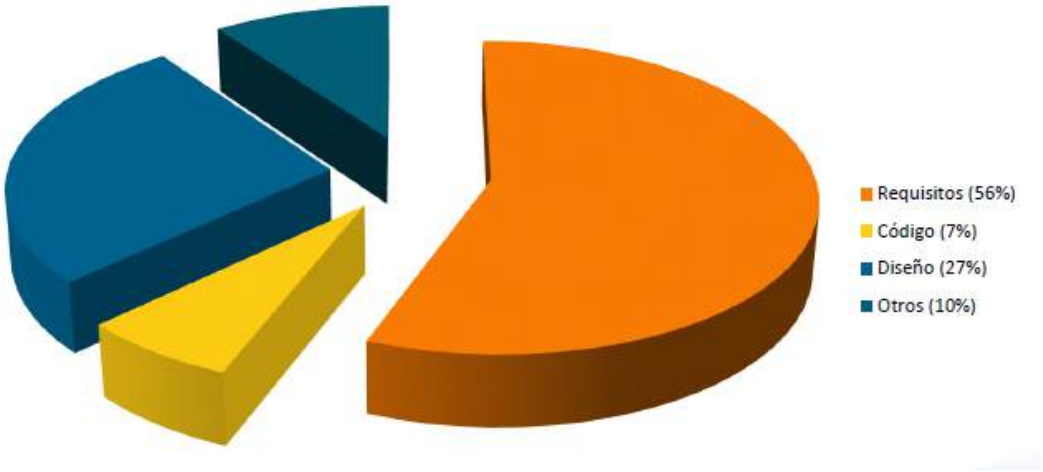


Figura 3.5: Distribución típica del origen de los errores.
Fuente: Lazic L., Mastorakis N., "Cost Effective Software Test Metrics", WSEAS TRANSACTIONS on COMPUTERS, Issue 6, Volume 7, June 2008, pp. 599-619

La experiencia muestra que la fase en la que se producen más errores es la de requisitos, es decir, justo donde se esta recogiendo y especificando como debe comportarse un sistema, una vez más, vemos la importancia de la comunicación a la hora de abordar cualquier proyecto.

Otra característica esencial de un software es que a diferencia del hardware, se puede hacer evolucionar, es decir, es susceptible de sufrir modificaciones a lo largo de su vida útil, no hablamos ya de corregir errores o de insertar nuevas funcionalidad, sino de cambiar incluso el propósito para el que fue creado.

Así mismo, el esfuerzo para corregir los errores, se produce en su mayoría en esta misma fase de requisitos, tal y como muestra la siguiente figura:

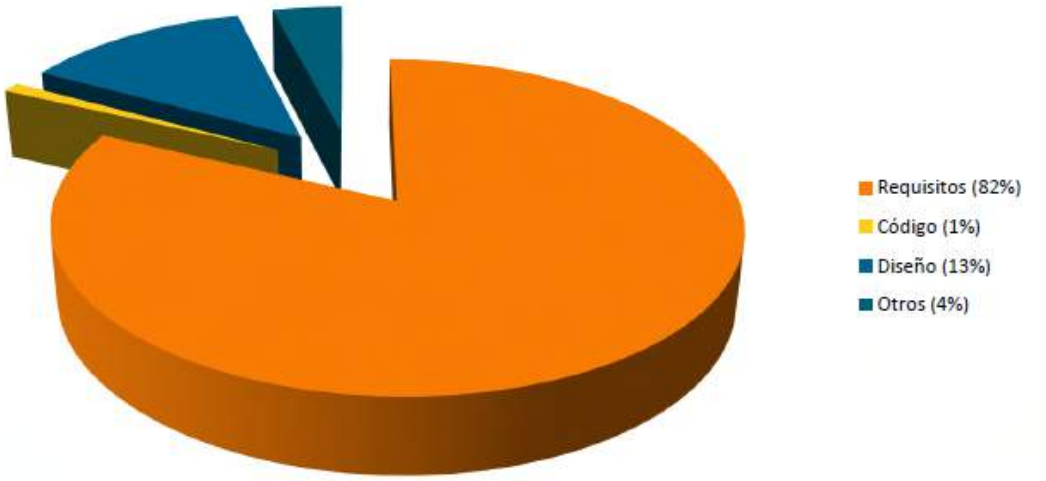


Figura 3.6: Distribución tipica del esfuerzo para corregir errores.
Fuente: Lazic L., Mastorakis N., "Cost Effective Software Test Metrics", WSEAS TRANSACTIONS on COMPUTERS, Issue 6, Volume 7, June 2008, pp. 599-619

Por tanto, es muy importante dedicar gran parte de los recursos en detectar y corregir los errores en la fase más temprana del desarrollo de software, que es la de requisitos, puesto que casi siempre llegaremos a la fase de explotación con algún error imprevisto, pero el coste de corregir el error en dicho momento se multiplica.

Es más costoso corregir un error cuando se detecta en una fase posterior a la fase en la que se ha cometido, ya que dicha corrección suele requerir un seguimiento hacia atrás del problema, hasta llegar a su origen, y repetir el desarrollo desde ese punto

Datos recogidos en proyectos concretos indican que el orden de magnitud del aumento del coste de corrección de errores por cada fase por la que pasa inadvertido un error es un factor diez. Del gráfico siguiente se deduce lo importante que es detectar los defectos en la misma fase en la que se cometieron los errores que los generaron, es decir lo antes posible, con el fin de reducir los costes de producción del software.

4. PROCESO DE DESARROLLO DE SOFTWARE

4.1 FASES DE CICLO DE VIDA DEL SOFTWARE

Por todo lo visto en los episodios anteriores, sabemos que un software falla porque durante su desarrollo se producen errores humanos que dan lugar a defectos en el sistema. Así mismo, cuanto más grande es el sistema software, mayor será el número posible de errores y por tanto, la probabilidad de fallo también será mayor.

Para conseguir un software lo más fiable posible, es necesario reducir a la minima expresión los defectos existentes, y además, es necesario depurar dichos errores durante cada una de las fases del ciclo de vida del sistema realizando las pruebas más adecuadas a cada fase.

Es muy importante identificar dichas fases para poder aplicar en cada una de ellas las técnicas más apropiadas que ayuden a disminuir el número de errores que se cometen, y a detectar los defectos ya existentes, los cuales pueden ser consecuencia de errores cometidos durante la fase actual ó en fases previas.

Por ciclo de vida del software [7], entendemos la sucesión de etapas por las que pasa el software desde que un nuevo proyecto es concebido hasta que se deja de usar. Estas etapas representan el ciclo de actividades involucradas en el desarrollo, uso y mantenimiento de sistemas de software, además de llevar asociadas una serie de documentos que serán la salida de cada una de estas fases y servirán de entrada en la fase siguiente.

Las fases son las siguientes:

1. **Fase de Especificación**: compuesta por las siguientes actividades:

    - **Adopción e identificación del sistema:** es importante conocer el origen del sistema, así como las motivaciones que impulsaron el desarrollo del sistema (por qué, para qué, etcétera.).

    - **Análisis de requerimientos**: identificación de las necesidades del cliente y los usuarios que el sistema debe satisfacer.

    - **Especificación**: los requerimientos se realizan en un lenguaje más formal, de manera que se pueda encontrar la función de correspondencia entre las entradas del sistema y las salidas que se supone que genera. Al estar completamente especificado el sistema, se pueden hacer estimaciones cuantitativas del coste, tiempos de diseño y asignación de personal al sistema, así como la planificación general del proyecto.

    - **Especificación de la arquitectura**: define las interfaces de interconexión y recursos entre módulos del sistema de manera apropiada para su diseño detallado y administración.

2. **Fase de Diseño y Desarrollo**: compuesto por las siguientes actividades:

- **Diseño**: en esta etapa, se divide el sistema en partes manejables que, como anteriormente hemos dicho se llaman módulos, y se analizan los elementos que las constituyen. Esto permite afrontar proyectos de muy alta complejidad.

- **Desarrollo e implementación**: codificación y depuración de la etapa de diseño en implementaciones de código fuente operacional.

- **Integración y prueba del software**: ensamble de los componentes de acuerdo a la arquitectura establecida y evaluación del comportamiento de todo el sistema atendiendo a su funcionalidad y eficacia.

- **Documentación**: generación de documentos necesarios para el uso y mantenimiento.

3. **Fase de Mantenimiento**: compuesta de las siguientes actividades:

    - **Entrenamiento y uso**: instrucciones y guías para los usuarios detallando las posibilidades y limitaciones del sistema, para su uso efectivo.

    - **Mantenimiento del software**: actividades para el mantenimiento operativo del sistema. Se clasifican en: evolución, conservación y mantenimiento propiamente dicho.

Durante la fase de especificación se realiza un análisis de los requisitos, tanto del sistema como del usuario, se estudian las posibilidades que presenta el sistema, y se establece un plan de diseño para el proyecto.

En la fase de diseño y desarrollo se crea físicamente el software. En primer lugar se establece cómo se va realizar lo que se ha especificado y a continuación se desarrolla el sistema según el diseño definido. A medida que se va avanzando en el desarrollo, se van realizando pruebas de los distintos componentes generados, con el fin de comprobar que cumplen las especificaciones definidas en la fase inicial del ciclo de vida, y también para detectar errores que se hayan cometido durante su desarrollo.

Finalmente, la fase de mantenimiento corresponde al sistema en operación. En teoría, dado que un software no sufre desgastes, podría estar en operación indefinidamente sin problemas, pero la realidad es que estos sistemas también necesitan acciones de mantenimiento como consecuencia de errores que se cometieron en las fases anteriores y no se detectaron, o porque se requieren nuevas funcionalidades.

Existen diversos modelos de ciclo de vida, pero cada uno de ellos va asociado a unos métodos, herramientas y procedimientos que debemos usar a lo largo de un proyecto.

A continuación vamos a ver los diferentes modelos que existen para definir el ciclo de vida de un Sistema Software.

4.1.1 MODELO EN V

La figura siguiente [9] muestra un modelo que representa el ciclo de vida de los sistemas lógicos. Es el modelo en V presentado por la guía STARTS del Centro de Cálculo Nacional del Reino Unido.

Este modelo además de presentar las fases básicas del ciclo de vida de un software, analiza en profundidad la fase de diseño y desarrollo.

El modelo se define tomando como referencia tres ejes, uno horizontal y dos verticales. El eje horizontal indica, de izquierda a derecha, el tiempo, y los ejes verticales muestran la realización de tareas. El eje vertical de la izquierda, en sentido descendente, corresponde a las tareas de especificación y diseño, mientras que el de la derecha, en sentido ascendente corresponde a tareas de integración y pruebas. Los dos tipos de tareas confluyen en el vértice del diagrama que corresponde a la codificación y pruebas de cada uno de los elementos básicos, ó módulos especificados.

En el diagrama las tareas se representan mediante rectángulos, mientras que el resultado ó productos de las tareas se representan mediante elipses. Puede observarse que a cada producto resultante de una tarea de especificación y diseño le corresponde un producto obtenido como resultado de una acción de integración ó de pruebas, por lo que el modelo enfatiza que cada documento de especificación debe corresponderse con un elemento del software ya probado y que cumple la totalidad de las especificaciones descritas en el documento, permitiendo, por tanto, una mejor identificación de los productos intermedios y en consecuencia un mejor control de las pruebas intermedias que se realizan para minimizar el número de errores que se transmiten entre fase y fase.

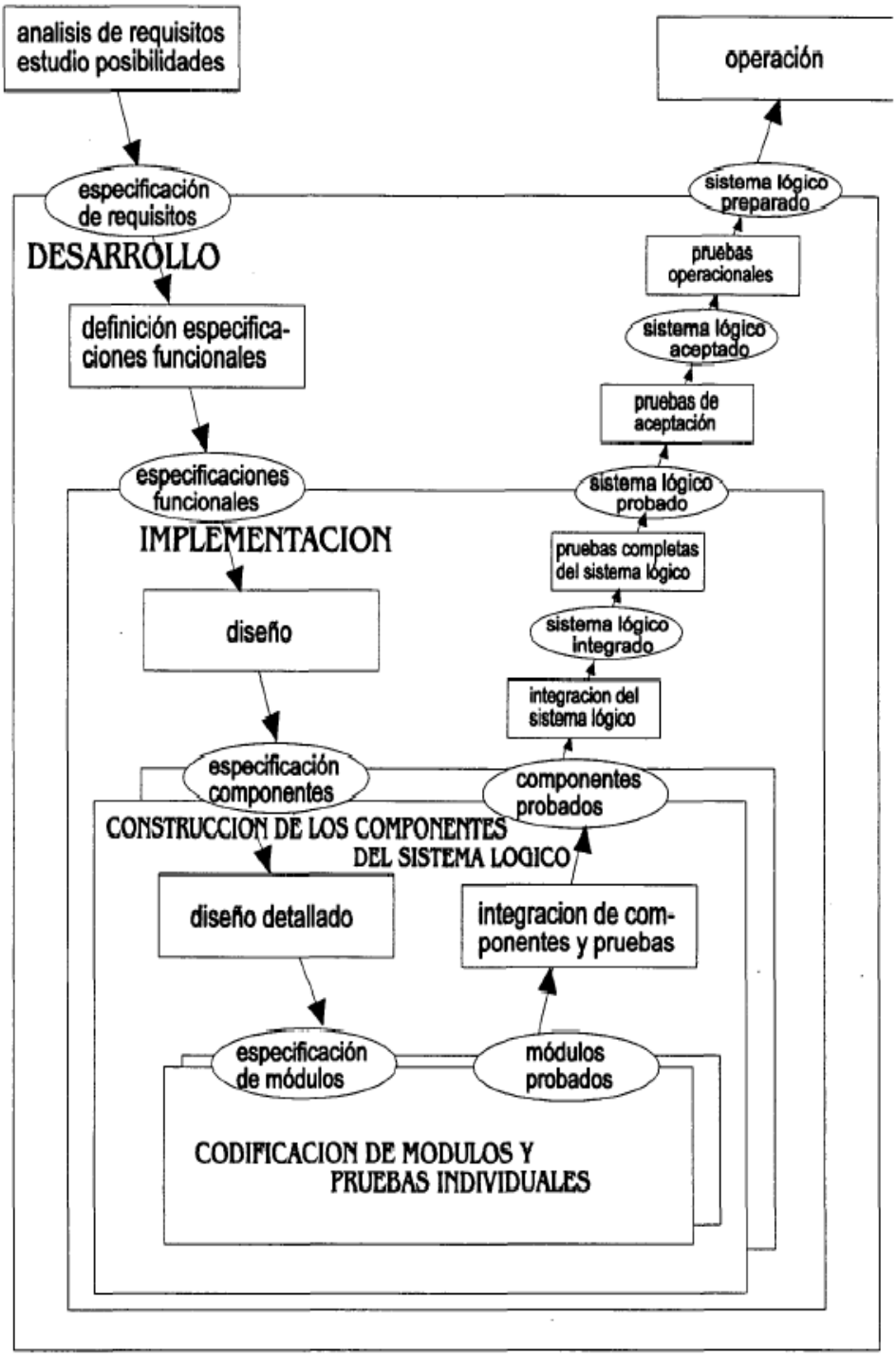


Figura 4.1 Figura en V del Ciclo de Vida de un Sistema Software.

En el diagrama también aparecen unos rectángulos mayores, ó bloques, contenidos unos dentro de otros, y que abarcan una ó varias tareas y sus productos resultantes. Estos bloques representan las etapas necesarias en el diseño y desarrollo de un software, y muestran claramente el emparejamiento de especificaciones y de elementos, validados y verificados, que componen el sistema.

Cada bloque puede verse como una etapa ó fase que recibe unas especificaciones de requisitos, y produce un elemento componente del software, acorde a dichas especificaciones. Antes de comenzar una fase hay una actividad que establece la especificación de requisitos, y una vez finalizada la fase hay una actividad que utiliza el producto/s de la fase/s para a su vez producir algo que se utilizará en el siguiente escalón ó que comprueba hasta que punto los productos de la fase cumplen los requisitos establecidos.

El bloque que identifica la totalidad de la fase de desarrollo comienza con el documento de especificación de requisitos de la totalidad del sistema, y finaliza con el software preparado para comenzar las pruebas de entrada en operación. Este bloque contiene el correspondiente a la etapa de implementación, que a su vez contiene a la etapa de construcción de los elementos básicos del sistema. La fase de implementación comienza con el documento de especificaciones funcionales, y finaliza con el software probado, mientras que la fase de construcción comienza con los documentos de especificación de los módulos, y finaliza con todos los módulos probados individualmente.

Todas las etapas tienen la misma forma funcional.

4.1.2 MODELO EN CASCADA

El modelo cascada (waterfall), propuesto por Royce en 1970, fue derivado de modelos de actividades de ingeniería con el fin de establecer algo de orden en el desarrollo de grandes productos de software. Consiste en diferentes etapas, las cuales son procesadas en una manera lineal. Comparado con otros modelos de desarrollo de software es más rígido y mejor administrable. El modelo cascada es un modelo muy importante y ha sido la base de muchos otros modelos, sin embargo, para muchos proyectos modernos, ha quedado un poco desactualizado.

El modelo cascada es un modelo de ingeniería diseñado para ser aplicado en el desarrollo de software. La idea principal es la siguiente: existen diferentes etapas de desarrollo, la salida de la primera etapa “fluye” hacia la segunda etapa y esta salida “fluye” hacia la tercera y así sucesivamente.

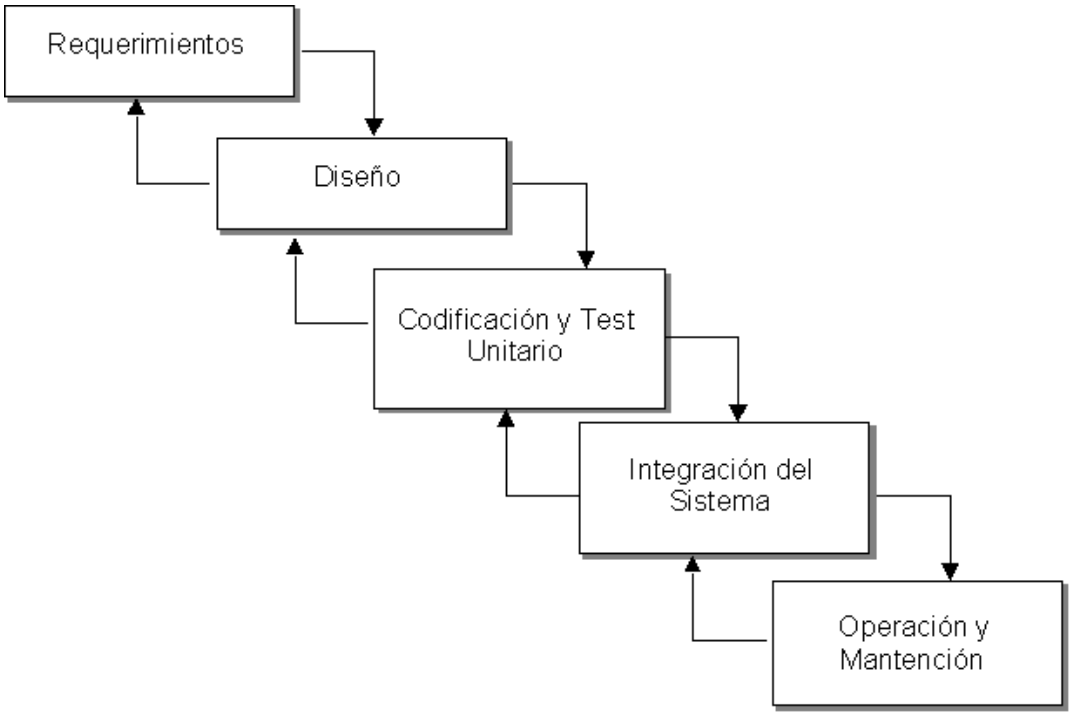


Figura 4.2 Modelo en Cascada.

Existen generalmente cinco etapas en este modelo de desarrollo de software:

1. **Análisis y definición de requerimientos:** en esta etapa, se establecen los requerimientos del producto que se desea desarrollar. Éstos consisten usualmente en los servicios que debe proveer, limitaciones y metas del software. Una vez que se ha establecido esto, los requerimientos deben ser definidos en una manera apropiada para ser útiles en la siguiente etapa. Esta etapa incluye también un estudio de la factibilidad y viabilidad del proyecto con el fin de determinar la conveniencia de la puesta en marcha del proceso de desarrollo. Puede ser tomada como la concepción de un producto de software y ser vista como el comienzo del ciclo de vida.

2. **Diseño del sistema**: el diseño del software es un proceso multipaso que se centra en cuatro atributos diferentes de los programas: estructura de datos, arquitectura del software, detalle del proceso y caracterización de las interfases. El proceso de diseño representa los requerimientos en una forma que permita la codificación del producto (además de una evaluación de la calidad previa a la etapa de codificación). Al igual que los requerimientos, el diseño es documentado y se convierte en parte del producto de software.

3. **Implementación**: esta es la etapa en la cual son creados los programas. Si el diseño posee un nivel de detalle alto, la etapa de codificación puede implementarse mecánicamente. A menudo suele incluirse un testeo unitario en

esta etapa, es decir, las unidades de código producidas son evaluadas individualmente antes de pasar a la etapa de integración y testeo global.

4. **Testeo del sistema**: una vez concluida la codificación, comienza el testeo del programa. El proceso de testeo se centra en dos puntos principales: las lógicas internas del software; y las funcionalidades externas, es decir, se solucionan errores de "comportamiento" del software y se asegura que las entradas definidas producen resultados reales que coinciden con los requerimientos especificados.

5. **Mantenimiento**: esta etapa consiste en la corrección de errores que no fueron previamente detectados, mejoras funcionales y de performance, y otros tipos de soporte. La etapa de mantenimiento es parte del ciclo de vida del producto de software y no pertenece estrictamente al desarrollo. Sin embargo, mejoras y correcciones pueden ser consideradas como parte del desarrollo.

Estas son las etapas principales. También existen sub-etapas, dentro de cada etapa, pero éstas difieren mucho de un proyecto a otro. También es posible que ciertos proyectos de software requieran la incorporación de una etapa extra o la separación de una etapa en otras dos. Sin embargo, todas estas variaciones al modelo cascada poseen el mismo concepto básico: la idea de que una etapa provee salidas que serán usadas como las entradas de la siguiente etapa (un flujo lineal entre las etapas). Por lo tanto, el progreso del proceso de desarrollo de un producto de software, usando el modelo cascada, es simple de conocer y controlar.

Existen actividades que son llevadas a cabo en cada una de las etapas del desarrollo del software. Éstas son documentación, verificación y administración. La documentación es intrínseca al modelo cascada puesto que la mayoría de las salidas que arrojan las etapas son documentos.

El ciclo de vida clásico es el paradigma más viejo y el más ampliamente usado en la ingeniería del software. Sin embargo, su aplicabilidad en muchos campos ha sido cuestionada. Entre los problemas que aparecen cuando se aplica el modelo cascada están:

- Los proyectos raramente siguen el flujo secuencial que el modelo propone. La iteración siempre es necesaria y está presente, creando problemas en la aplicación del modelo.

- A menudo es difícil para el cliente poder especificar todos los requerimientos explícitamente. El modelo de vida del software clásico requiere esto y presenta problemas acomodando la incertidumbre natural que existe al principio de cualquier proyecto.

- El cliente debe ser paciente. Una versión funcional del sistema no estará disponible hasta tarde en la duración del desarrollo. Cualquier error o malentendido, si no es detectado hasta que el programa funcionando es revisado, puede ser desastroso.

- Cada uno de estos problemas es real. Sin embargo, el modelo clásico del ciclo de vida del software tiene un lugar bien definido e importante en los trabajos de ingeniería del software. Provee un patrón dentro del cual encajan métodos para el análisis, diseño, codificación y mantenimiento.

El modelo cascada se aplica bien en situaciones en las que el software es simple y en las que el dominio de requerimientos es bien conocido, la tecnología usada en el desarrollo es accesible y los recursos están disponibles.

4.1.3 MODELO PROTOTIPADO

Dos de las críticas que se hacían al modelo de ciclo de vida en cascada eran que es difícil tener claros todos los requisitos del sistema al inicio del proyecto, y que no se dispone de una versión operativa del programa hasta las fases finales del desarrollo, lo que dificulta la detección de errores y deja también para el final el descubrimiento de los requisitos inadvertidos en las fases de análisis. Para paliar estas deficiencias se ha propuesto un modelo de ciclo de vida basado en la construcción de prototipos.

En primer lugar, hay que ver si el sistema que tenemos que desarrollar es un buen candidato a utilizar el paradigma de ciclo de vida de construcción de prototipos o al modelo en espiral. En general, cualquier aplicación que presente mucha interacción con el usuario, o que necesite algoritmos que puedan construirse de manera evolutiva, yendo de lo mas general a lo más específico es una buena candidata. No obstante, hay que tener en cuenta la complejidad: si la aplicación necesita que se desarrolle una gran cantidad de código para poder tener un prototipo que enseñar al usuario, las ventajas de la construcción de prototipos se verán superadas por el esfuerzo de desarrollar un prototipo que al final habrá que desechar o modificar mucho. También hay que tener en cuenta la disposición del cliente para probar un prototipo y sugerir modificaciones de los requisitos. Puede ser que el cliente ‘no tenga tiempo para andar jugando’ o ‘no vea las ventajas de este método de desarrollo’.

También es conveniente construir prototipos para probar la eficiencia de los algoritmos que se van a implementar, o para comprobar el rendimiento de un determinado componente del sistema en condiciones similares a las que existirán durante la utilización del sistema. Es bastante frecuente que el producto de ingeniería desarrollado presente un buen rendimiento durante la fase de pruebas realizada por los ingenieros antes de entregarlo al cliente pero que sea muy ineficiente, o incluso inviable, a la hora de almacenar o procesar el volumen real de información que debe manejar el cliente. En estos casos, la construcción de un prototipo de parte del sistema y la realización de pruebas de rendimiento, sirven para decidir, antes de empezar la fase de diseño, cuál es el modelo más adecuado de entre la gama disponible para el soporte hardware o cómo deben hacerse los accesos para obtener buenas respuestas en tiempo cuando la aplicación esté ya en funcionamiento.

En otros casos, el prototipo servirá para modelar y poder mostrar al cliente cómo va a realizarse la E/S de datos en la aplicación, de forma que éste pueda hacerse una idea de como va a ser el sistema final, pudiendo entonces detectar deficiencias o errores en la especificación aunque el modelo no sea más que una cáscara vacía.

Según esto un prototipo puede tener alguna de las tres formas siguientes:

- Un prototipo, en papel o ejecutable en ordenador, que describa la interacción hombre-máquina y los listados del sistema.

- Un prototipo que implemente algún(os) subconjunto(s) de la función requerida, y que sirva para evaluar el rendimiento de un algoritmo o las necesidades de capacidad de almacenamiento y velocidad de cálculo del sistema final.

- Un programa que realice en todo o en parte la función deseada pero que tenga características (rendimiento, consideración de casos particulares, etc.) que deban ser mejoradas durante el desarrollo del proyecto.

La secuencia de tareas del paradigma de construcción de prototipos puede verse en la siguiente figura.

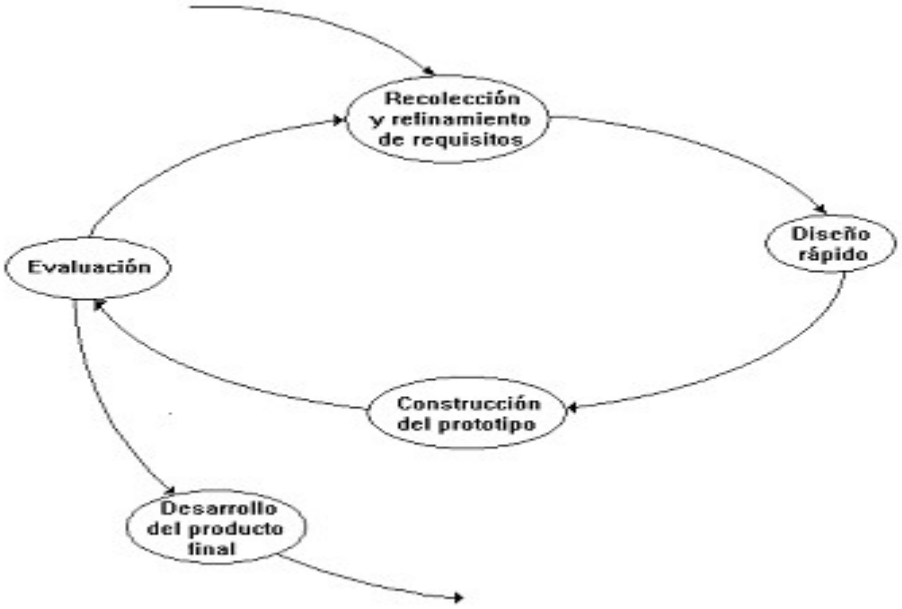


Figura 4.3 Modelo Prototipado.

Si se ha decidido construir un prototipo, lo primero que hay que hacer es realizar un modelo del sistema, a partir de los requisitos que ya conozcamos. En este caso no es necesario realizar una definición completa de los requisitos, pero sí es conveniente determinar al menos las áreas donde será necesaria una definición posterior más detallada.

Luego se procede a diseñar el prototipo. Se tratará de un diseño rápido, centrado sobre todo en la arquitectura del sistema y la definición de la estructura de las interfaces más que en aspectos de procedimiento de los programas: nos fijaremos más en la forma y en la apariencia que en el contenido.

A partir del diseño construiremos el prototipo. Existen herramientas especializadas en generar prototipos ejecutables a partir del diseño. Otra opción sería utilizar técnicas de cuarta generación. La posibilidad más reciente consiste en el uso de especificaciones formales, que faciliten el desarrollo incremental de especificaciones y permitan la prueba de estas especificaciones.

En cualquier caso, el objetivo es siempre que la codificación sea rápida, aunque sea en detrimento de la calidad del software generado.

Una vez listo el prototipo, hay que presentarlo al cliente para que lo pruebe y sugiera modificaciones. En este punto el cliente puede ver una implementación de los requisitos que ha definido inicialmente y sugerir las modificaciones necesarias en las especificaciones para que satisfagan mejor sus necesidades.

A partir de estos comentarios del cliente y los cambios que se muestren necesarios en los requisitos, se procederá a construir un nuevo prototipo y así sucesivamente hasta que los requisitos queden totalmente formalizados, y se pueda entonces empezar con el desarrollo del producto final.

Por tanto, el prototipado es una técnica que sirve fundamentalmente para la fase de análisis de requisitos.

4.1.4 MODELO EN ESPIRAL

El problema con los modelos de procesos de software es que no se enfrentan lo suficiente con la incertidumbre inherente a los procesos de software. Importantes proyectos de software fallaron porque los riegos del proyecto se despreciaron y nadie se preparó para algún imprevisto.

Boehm reconoció esto y trató de incorporar el factor “riesgo del proyecto” al modelo de ciclo de vida, agregándoselo a las mejores características de los modelos Cascada y Prototipado. El resultado fue el Modelo Espiral.

La dirección del nuevo modelo fue incorporar los puntos fuertes y evitar las dificultades de otros modelos desplazando el énfasis de administración hacia la evaluación y resolución del riesgo. De esta manera se permite tanto a los desarrolladores como a los clientes detener el proceso cuando se lo considere conveniente.

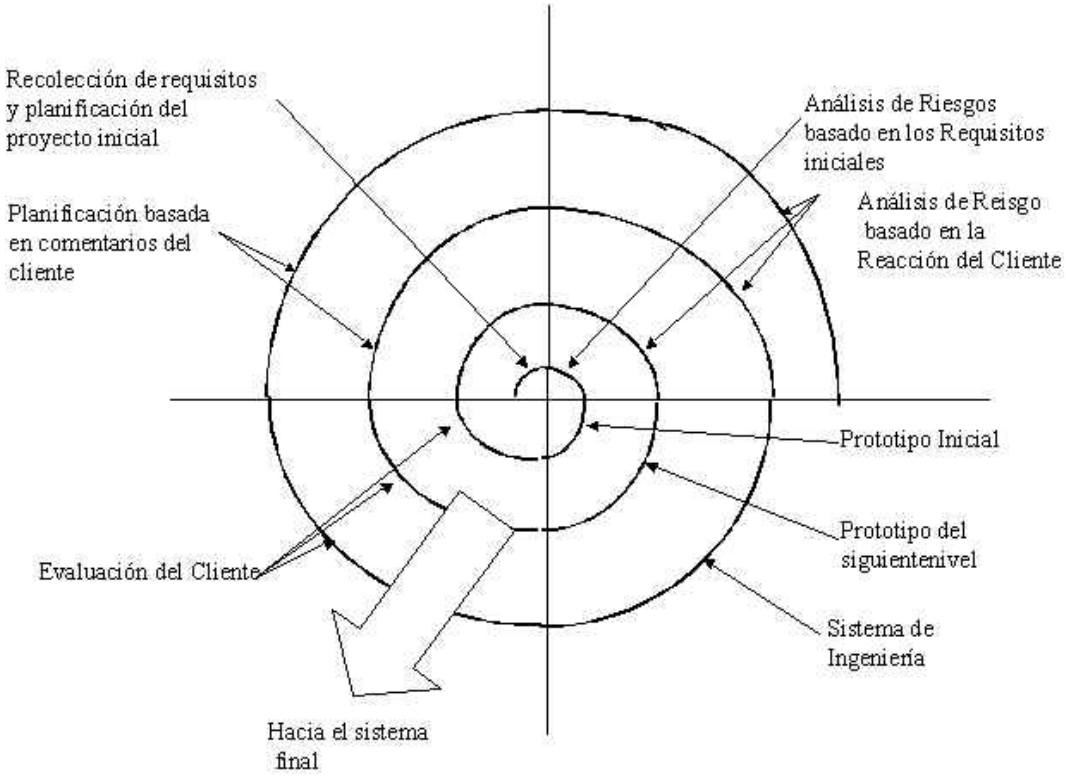


Figura 4.4 Modelo en Espiral.

Básicamente, la idea es desarrollo incremental usando el modelo Cascada para cada paso, ayudando a administrar los riesgos. No se define en detalle el sistema completo al principio; los diseñadores deberían definir e implementar solamente los rasgos de mayor prioridad. Con el conocimiento adquirido, podrían entonces volver atrás y definir e implementar más características en trozos más pequeños.

El modelo Espiral define cuatro actividades principales en su ciclo de vida:

1) **Planeamiento**: determinación de los objetivos, alternativas y limitaciones del proyecto.

2) **Análisis de riesgo**: análisis de alternativas e identificación y solución de riesgos.

3) **Ingeniería**: desarrollo y testeo del producto.

4) **Evaluación del cliente**: tasación de los resultados de la ingeniería.

El modelo está representado por una espiral dividida en cuatro cuadrantes, cada una de las cuales representa una de las actividades arriba mencionadas.

Los puntos fuertes de este modelo son:

- Evita las dificultades de los modelos existentes a través de un acercamiento conducido por el riesgo.
- Intenta eliminar errores en las fases tempranas.
- Es el mismo modelo para el desarrollo y el mantenimiento.
- Provee mecanismos para la aseguración de la calidad del software.
- La reevaluación después de cada fase permite cambios en las percepciones de los usuarios, avances tecnológicos o perspectivas financieras.
- La focalización en los objetivos y limitaciones ayuda a asegurar la calidad.

Los puntos débiles son:

- Falta un proceso de guía explícito para determinar objetivos, limitaciones y alternativas.
- Provee más flexibilidad que la conveniente para la mayoría de las aplicaciones.
- La pericia de tasación del riesgo no es una tarea fácil. El autor declara que es necesaria mucha experiencia en proyectos de software para realizar esta tarea exitosamente.

Su aplicación se centra sobre todo en proyectos complejos, dinámicos, innovadores, ambiciosos, llevados a cabo por equipos internos (no necesariamente de software).

4.1.5 MODELO CLEANROOM

Cleanroom es un proceso de administración e ingeniería para el desarrollo de software de alta calidad con fiabilidad certificada. Focaliza la atención en la prevención en lugar de la corrección de errores, y la certificación de la fiabilidad del software para el entorno de uso para cual fue planeado. En lugar de realizar pruebas de unidades y módulos, se especifican formalmente componentes de software los cuales son verificados matemáticamente en cuanto son construidos.

A continuación se muestra una figura donde se esquematiza el modelo:

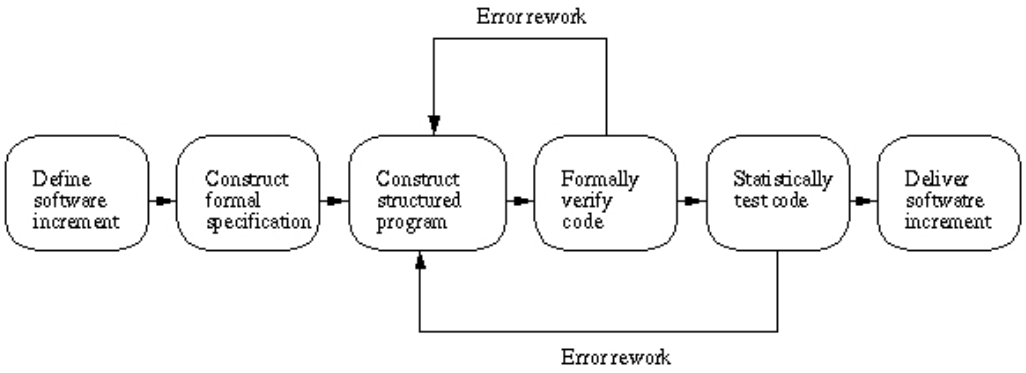


Figura 4.5 Modelo Cleanroom.

El modelo tiene las siguientes características:

- **Desarrollo incremental**: el software es particionado en incrementos los cuales se desarrollan utilizando el proceso cleanroom.

- **Especificación formal**: el software a desarrollarse es formalmente especificado.

- **Verificación estática**: el software desarrollado es verificado matemáticamente (los desarrolladores no pueden ejecutar el código) utilizando argumentos de correctitud basados en matemáticas. No hay pruebas a nivel unidad o módulo.
- **Pruebas estadísticas**: el incremento de software integrado es examinado estadísticamente para determinar su fiabilidad.

Hay tres equipos involucrados en el proceso de cleanroom:

- El equipo de especificación: este equipo es responsable del desarrollo y mantenimiento de las especificaciones del sistema. Ambas especificaciones, las orientadas al cliente.

- El equipo de desarrollo: este equipo tiene la responsabilidad de desarrollar y verificar el software.

- El equipo de certificación: este equipo es responsable del desarrollo de un conjunto de pruebas estadísticas para ejercitar el software una vez que fue construido.

Con respecto a los equipos, estos deben ser pequeños: típicamente de seis a ocho integrantes. El testeo del software debe ser llevado a cabo por un equipo independiente.

Las ventajas de utilizar este modelo son:

- Cleanroom provee las prácticas de administración e ingeniería que permiten a los equipos lograr cero fallos en el campo de uso, cortos ciclos de desarrollo, y una larga vida del producto.

- Reduce los fallos encontrados durante el proceso de certificación a menos de cinco fallos por cada mil líneas de código en la primera ejecución del código del primer proyecto.

- Equipos nuevos deberían experimentar un incremento del doble en la productividad con respecto a su primer proyecto. La productividad seguirá incrementándose con la experiencia adquirida.

- Cleanroom lleva a una inversión en bienes tales como especificaciones detalladas y modelos que ayudan a mantener un producto viable durante una vida más larga.

- Todos los beneficios técnicos se trasladan en beneficios económicos significantes.

El método cleanroom es mandatorio para cualquier sistema de software de misión crítica; sin embargo es apto para cualquier proyecto de software, en especial si el proyecto requiere detección de errores en fases tempranas.

4.1.6 CONCLUSIONES

La siguiente tabla muestra una comparación entre los cuatro modelos más utilizados:

Criterio	Cascada	Prototipado	Cleanroom	Espiral
Disponibilidad de recursos	Todos	Algunos	Algunos	Algunos
Complejidad del proyecto	Baja	Media	Alta	Alta
Entendimiento de los requerimientos	Específico	Vago	Vago	Vago
Tecnología del producto	Vieja	Nueva	Nueva	Nueva
Volatilidad de requerimientos	No	Si	Si	Si
Manejo de la perspectiva del riesgo	No	Si	No	Si
Conocimiento del dominio del problema	Alto	Regular	Alto	Pobre

Figura 4.6 Comparación entre modelos de Ciclo de Vida.

Los métodos presentados tienen un campo de aplicación bien definido; sin embargo, los problemas reales con los que un equipo de desarrollo puede enfrentarse no calzan de manera perfecta en los dominios para los cuales los métodos fueron pensados. Es tarea de los ingenieros adaptar un método o componer uno nuevo a partir de los métodos existentes para conseguir una solución que se amolde al problema en particular que se está atacando. En definitiva, es el problema el que propone la solución (o al menos el camino a la solución).

Independientemente del paradigma que se utilice, del área de aplicación, y del tamaño y la complejidad del proyecto, el proceso de desarrollo de software contiene siempre una serie de fases genéricas, existentes en todos los paradigmas. Estas fases son la definición, el desarrollo y el mantenimiento.

4.2 UTILIZACION DE LA FIABILIDAD DURANTE EL DESARROLLO SOFTWARE

Si se decide seguir trabajando en el producto, las acciones a tomar son identificar los orígenes de los problemas y realizar los cambios necesarios para corregirlos. Se debe documentar estas acciones.

Durante el progreso de cada fase se debe ir reuniendo todo tipo de información relativa a su evolución, como por ejemplo estadísticas acerca de los defectos detectados y sus orígenes, perfil de los distintos equipos involucrados, o cumplimientos de objetivos. Esta información se analiza periódicamente con el fin de mejorar el proceso.

El modelo presentado muestra que, continuamente a lo largo del ciclo de vida de un software, se dedica una cantidad importante de recursos a la tarea de detectar posibles causas de fallo del sistema, con el fin de que durante su fase de operación el número de fallos sea mínimo, idealmente nulo.

Las causas de fallo se detectan mediante la realización de pruebas que consisten en poner en operación el sistema, o sus componentes individualmente, hasta que ocurra un fallo, entonces se busca el motivo de dicho fallo y se intenta corregir. A continuación se vuelve a poner el sistema en operación repitiéndose el proceso descrito hasta que se den por finalizadas las pruebas. Normalmente ocurren más fallos durante las pruebas iniciales, y según se va avanzando en el ciclo de vida el número de fallos va disminuyendo ya que los defectos que pueda contener el sistema se van detectando a medida que se van realizando las pruebas.

Gráficamente, la curva de fallos del software se representa de la siguiente forma:

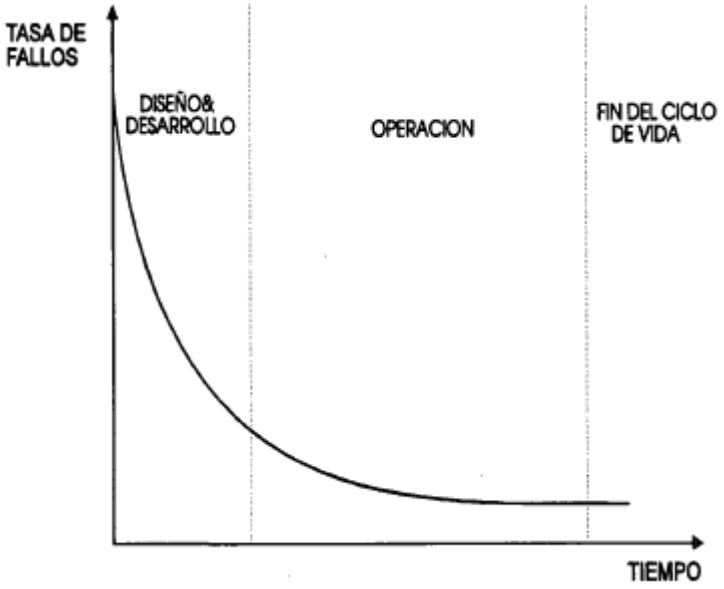


Figura 4.7 Curva ideal de fallos del software.

Inicialmente la tasa de fallos es alta, por errores no detectados durante su desarrollo, pero que una vez corregidos y dado que no está expuesto a factores de deterioro externos, la tasa de fallos debería alcanzar un nivel estacionario y mantenerse definitivamente.

La aplicación de la teoría de la fiabilidad al software se ha apoyado en los conocimientos adquiridos en la aplicación de dicha disciplina a los sistemas físicos, cuya tasa de fallos genérica se muestra en la figura 4.8, también conocida como “curva de la bañera”.

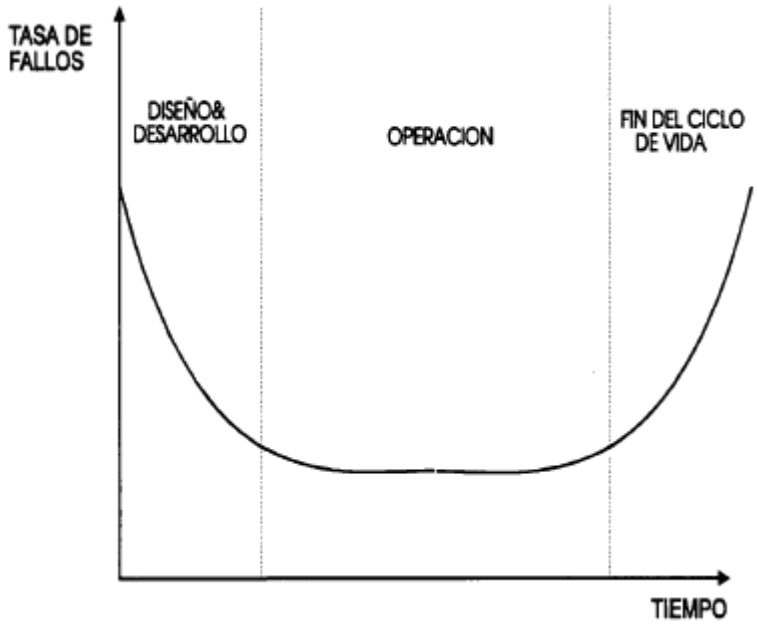


Figura 4.8 Curva de la bañera.

La comparación de ambas tasas de fallos pone de manifiesto las diferencias entre el software y el hardware.

Al principio de su vida se puede ver que tiene muchos fallos, debidos en gran parte a los defectos de diseño o a la baja calidad inicial de la fase de producción. Dichos defectos se van subsanando hasta llegar a un nivel estacionario, pero los fallos vuelven a incrementarse debido al deterioro de los componentes (suciedad, vibraciones u otros factores externos) por lo que hay que sustituir los componentes defectuosos por otros nuevos para que la tasa de fallos vuelva a estar a un nivel estacionario.

La diferencia de comportamientos se debe a que el mecanismo de fallo de los sistemas físicos no es el mismo que el del software, aunque tienen el mismo resultado, la degradación del sistema.

Los sistemas físicos fallan debido a concentraciones de esfuerzos, desgastes físicos, y a factores atmosféricos u otros factores de tipo ambiental, mientras que el software fallo cómo consecuencia de errores humanos cometidos durante las fases del ciclo de vida del sistema.

Todo esto no es más que una simplificación del modelo real de fallos de un producto software. Durante su vida, el software va sufriendo cambios debido a su mantenimiento, el cual puede orientarse tanto a la corrección de errores como a cambios en los requisitos iniciales del producto. Al realizar los cambios es posible que se produzcan nuevos errores con lo cual se manifiestan picos en la curva de fallos.

Estos errores pueden corregirse, pero los sucesivos cambios, hacen que el producto se aleje cada vez más de las especificaciones iniciales de acuerdo a las cuales fue desarrollado, conteniendo cada vez más errores. Además, a veces, se solicita un nuevo cambio antes de haber corregido todos los errores producidos por el cambio anterior.

Por todo ello, como vemos en la figura siguiente, el nivel estacionario que se consigue después de un cambio es algo superior al que había antes de efectuarlo, degradándose poco a poco el funcionamiento del sistema. De esta manera, podemos decir que el software no se estropea, pero se deteriora.

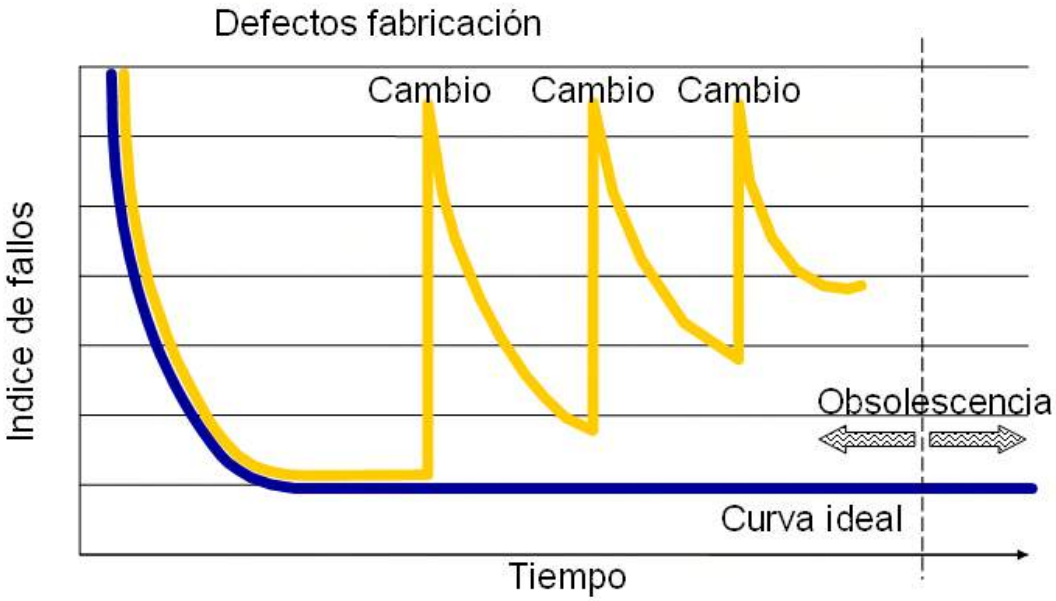


Figura 4.9 Curva real de fallos del software vs. curva ideal.

5. MÉTRICAS DE SOFTWARE: CALIDAD Y FIABILIDAD

5.1 LAS MÉTRICAS Y LA CALIDAD DE SOFTWARE

El proceso de planificación del desarrollo de cualquier sistema debe hacerse partiendo de una estimación del trabajo a realizar. Sólo a partir de ello es factible conocer los recursos necesarios y el tiempo necesario para su realización.

Una métrica es una medida efectuada sobre algún aspecto del sistema en desarrollo o del proceso empleado que permite, previa comparación con unos valores (medidas) de referencia, obtener conclusiones sobre el aspecto medido con el fin de adoptar las decisiones necesarias. Con esta premisa, la definición y aplicación de una métrica no es un objetivo en sí mismo sino un medio para controlar el desarrollo de un sistema de software.

El objetivo primordial de la ingeniería del software es producir un sistema, aplicación o producto de alta calidad. Para lograr este objetivo, los ingenieros de software deben emplear métodos efectivos junto con herramientas modernas dentro del contexto de un proceso maduro de desarrollo del software.

Al mismo tiempo, un buen ingeniero del software y buenos administradores de la ingeniería del software deben medir si la alta calidad se va a llevar a cabo.
El punto de vista de ¿Qué es calidad? Es diverso, y por lo tanto existen distintas respuestas, tales como se muestra a continuación:

- El American Heritage Dictionary define la calidad como “Una característica o atributo de algo.”

- La definición estándar de calidad en ISO-8402 es “La totalidad de rasgos y características de un producto, proceso o servicio que sostiene la habilidad de satisfacer estados o necesidades implícitas” [10].

- “Concordar explícitamente al estado funcional y a los requerimientos del funcionamiento, explícitamente a los estándares de documentación de desarrollo, e implícitamente características que son expectativas de todos los desarrolladores profesionales de software” [7].

Para obtener esta evaluación de calidad, el ingeniero debe utilizar medidas técnicas, que evalúan la calidad con objetividad, no con subjetividad. Asimismo, un buen administrador de proyectos debe evaluar la calidad objetivamente y no subjetivamente.

A medida que el proyecto progresa el administrador del proyecto siempre debe valorar la calidad. Aunque se pueden recopilar muchas medidas de calidad, el primer objetivo en el proyecto es medir errores y defectos. Las métricas que provienen de estas medidas proporcionan una indicación de la efectividad de las actividades de control y de la garantía de calidad en grupos o en particulares.

Por ejemplo los errores detectados por hora de revisión y los errores detectados por hora de prueba suministran una visión profunda de la eficacia de cada una de las actividades envueltas en la métrica. Así los datos de errores se pueden utilizar también para calcular

la eficiencia de eliminación de defectos en cada una de las actividades del marco de trabajo del proceso.

A continuación se verá un conjunto de factores de calidad y métricas del software que pueden emplearse a la valoración cuantitativa de la calidad de software.

5.1.1 VISIÓN GENERAL DE LOS FACTORES QUE AFECTAN A LA CALIDAD

McCall y Cavano [12] definieron un juego de factores de calidad como los primeros pasos hacia el desarrollo de métricas de la calidad del software. Estos factores evalúan el software desde tres puntos de vista distintos:

1. Operación del producto (utilizándolo).

2. Revisión del producto (cambiándolo).

3. Transición del producto (modificándolo para que funcione en un entorno diferente, por ejemplo: "portándolo").

Los autores describen la relación entre estos factores de calidad (lo que llaman un 'marco de trabajo') y otros aspectos del proceso de ingeniería del software:

En primer lugar el marco de trabajo proporciona al administrador identificar en el proyecto lo que considera importante, como: facilidad de mantenimiento y transportabilidad, atributos del software, además de su corrección y rendimiento funcional teniendo un impacto significativo en el costo del ciclo de vida. En segundo lugar, proporciona un medio de evaluar cuantitativamente el progreso en el desarrollo de software teniendo relación con los objetivos de calidad establecidos. En tercer lugar, proporciona más interacción del personal de calidad, en el esfuerzo de desarrollo. Por último, el personal de calidad puede utilizar indicaciones de calidad que se establecieron como "pobres" para ayudar a identificar estándares "mejores" para verificar en el futuro.

Es importante destacar que casi todos los aspectos del cálculo han sufrido cambios radicales con el paso de los años desde que McCall y Cavano hicieron su trabajo, teniendo gran influencia, en 1978. Pero los atributos que proporcionan una indicación de la calidad del software siguen siendo los mismos.

Si una organización de software adopta un juego de factores de calidad como una "lista de comprobación" para evaluar la calidad del software, es probable que el software construido hoy siga exhibiendo la buena calidad dentro de las primeras décadas del siglo XXI. Incluso, cuando las arquitecturas del cálculo sufran cambios radicales (como seguramente ocurrirá), el software que exhibe alta calidad en operación, transición y revisión continuará sirviendo a sus usuarios.

Estos son los factores que determinan la calidad del software según el modelo de McCall [8]:

Aspecto que trata	Factor Calidad	Relacionado con …
Operación del Producto	Corrección	Grado en que un programconsigue los objetivos de la misión encomendada por el cliente
	Fiabilidad	Probabilidad de operación libre de fallos de un programa de computadora en un entorno determinado durante un tiempo específico
	Eficiencia	Cantidad de recursos de computadora y de código requeridos por un programa para llevar a cabo sus funciones
	Integridad	Grado en que puede controlarse el acceso al software o a los datos, por personal no autorizado
	Facilidad de uso	Esfuerzo requerido para aprender un programa, trabajar con él, preparar su entrada e interpretar su salida
Revisión del producto	Facilidad de mantenimiento	Esfuerzo requerido para localizar y arreglar un error en un programa
	Flexibilidad	Esfuerzo requerido para modificar un programa operativo
	Facilidad de prueba	Esfuerzo requerido para probar un programa de forma que se asegure que realiza la función requerida
Transición del producto	Portabilidad	Esfuerzo requerido para transferir el programa desde un hardware y/o entorno de sistemas de software a otro
	Reusabilidad	Grado en que un programa (o partes de un programa) puede reusarse en otras aplicaciones
	Facilidad de interoperación	Esfuerzo requerido para acoplar un sistema a otro

Figura 5.1 Factores de Calidad según McCall.

Los factores desarrollados según el modelo de McCall, se centra en tres aspectos importantes de un producto de software:

- Sus características operativas.
- Su capacidad para soportar los cambios.
- Su adaptabilidad a nuevos entornos.

Estos factores de calidad de McCall, a pesar de haberse definido hace mucho tiempo, conservan en gran medida su vigencia (salvo para el caso de la programación orientada a objetos).

Es difícil (y en algunos casos imposible) desarrollar medidas directas de los anteriores factores de calidad. Por tanto, se define un conjunto de métricas usadas para evaluar los factores de calidad. Según de qué factor se trate, se utilizarán unas determinadas métricas ponderadas para determinar ese factor. Las métricas (algunas de ellas medidas subjetivas) que utiliza McCall para evaluar los distintos factores son los siguientes:

Métrica	Relacionado con ...
Facilidad de auditoría	La facilidad con que se puede comprobar la conformidad con los estándares
Exactitud	La precisión de los cálculos y del control
Normalización de las comunicaciones	El grado en que se usa el ancho de banda, los protocolos y las interfaces estándar
Completitud	El grado en que se ha conseguido la total implantación de las funciones requeridas
Concisión	Lo compacto que es el programa en términos de líneas de código
Consistencia	El uso de un diseño uniforme y de técnicas de documentación a lo largo de todo el programa
Estandarización en los datos	El uso de estructuras de datos y de tipos estándar a lo largo de todo el programa
Tolerancia de errores	El daño que se produce cuando el programa detecta una situación erronea
Eficiencia en la ejecución	El rendimiento en tiempo de ejecución de un programa
Facilidad de expansión	El grado en que se puede ampliar el diseño arquitectónico, de datos o procedimental
Generalidad	La amplitud de aplicación potencial de los componentes del programa
Independencia del hardware	El grado en que el software es independiente del hardware sobre el que opera
Instrumentación	El grado en que el programa muestra su propio funcionamiento e identifica errores que aparecen
Modularidad	La independencia funcional de los componentes del programa
Facilidad de operación	La facilidad de utilización de un programa

Seguridad	La disponibilidad de mecanismos que controlen o protejan los programas o los datos
Autodocumentación	El grado en que el código fuente proporciona documentación significativa
Simplicidad	El grado en que un programa puede ser entendido sin dificultad
Independencia del sistema de software	El grado en que el programa es independiente de características no estándar del lenguaje de programación, de las características del sistema operativo y de otras restricciones del entorno
Facilidad de traza	La posibilidad de seguir la pista a la representación del diseño o de los componentes reales del programa hacia atrás, hacia los requisitos
Formación	El grado en que el software ayuda a permitir que nuevos usuarios empleen el sistema

Figura 5.2 Métricas de Calidad de McCall.

El segundo modelo de calidad más conocido es el de Barry Boehm. Se centra en:

1. Sus características operativas.
2. Su capacidad para soportar los cambios.
3. Su adaptabilidad a nuevos entornos.
4. La evaluación del desempeño del hardware.

El modelo comienza con la utilidad general del software, afirmando que el software es útil, evitando pérdida de tiempo y dinero.
La utilidad puede considerarse en correspondencia a los tipos de usuarios que quedan involucrados. El primer tipo de usuarios queda satisfecho si el sistema hace lo que el pretende que haga; el segundo tipo es aquel que utiliza el sistema luego de una actualización y el tercero, es el programador que mantiene el sistema.

Aunque parezcan similares, la diferencia está en que McCall focaliza en medidas precisas de alto nivel, mientras que Boehm presenta un rango más amplio de características primarias. La Mantenibilidad está más desarrollada en Boehm

El modelo FURPS (Funcionality, Usability, Reliability, Performance, Supportability) es otro modelo desarrollado por Hewlett-Packard, cuyos factores son:

- Funcionalidad: se aprecia evaluando el conjunto de características y capacidades del programa, la generalidad de las funciones entregadas y la seguridad del sistema global.

- Usabilidad (facilidad de empleo o uso): se valora considerando factores humanos, la estética, consistencia y documentación general.

- Fiabilidad: se evalúa midiendo la frecuencia y gravedad de los fallos, la exactitud de las salidas (resultados), el tiempo medio entre fallos (TMEF), la capacidad de recuperación de un fallo y la capacidad de predicción del programa.

- Rendimiento: se mide por la velocidad de procesamiento, el tiempo de respuesta, consumo de recursos, rendimiento efectivo total y eficacia.

Durante muchos años se buscó en la Ingeniería de Software un modelo único para expresar calidad. La ventaja era obvia: poder comparar productos entre sí.

En 1992, una variante del modelo de McCall fue propuesta como estándar internacional para medición de calidad de software: ISO 9126 Software Product Evaluation: Quality Characteristics and Guidelines for their Use es el nombre formal.

El modelo ISO 9126 consiste en un modelo jerárquico con seis atributos especiales.

La diferencia con McCall y Boehm es que la jerarquía es estricta, es decir, que cada característica de la derecha solo está relacionada con un solo atributo del modelo. Las características de la derecha se relacionan con la visión del usuario.

- Funcionalidad → Adaptación, Exactitud, Interoperación, Seguridad.

- Confiabilidad → Madurez, Tolerancia a Defectos, Facilidad de Recuperación.

- Eficiencia → Comportamiento en el Tiempo, de los Recursos.

- Facilidad de Uso → Facilidad de Comprensión, de Aprendizaje, de Operación.

- Facilidad de Mantenimiento→ Facilidad de Análisis, de Cambios, de Pruebas, Estabilidad.

- Portabilidad → Adaptabilidad, Facilidad de Instalación, de Reemplazo.

Básicamente, a la hora de verificar la calidad del software en un proyecto, deberemos decidir empezar un proyecto, deberemos decidir si adoptar un modelo fijo en el que se supone que todos los factores de calidad importantes son un subconjunto de los de un modelo publicado y se acepta el conjunto de criterios y métricas asociados al modelo o desarrollar un modelo propio de calidad y se acepta que la calidad está compuesta por varios atributos, pero no se adopta lo impuesto por modelos existentes.

En este último caso, se debe consensuar el modelo con los clientes antes de empezar el proyecto, se deciden cuáles atributos son importantes para el producto, y cuáles medidas específicas los componen.

La primera dificultad que se presenta es cómo cuantificar el nivel requerido para cada una de las características que definen la calidad de un software, ya que la mayor parte de ellas se definen de una manera poco concreta y se necesita conocerlas con mayor profundidad antes de poder cuantificarlas rigurosamente.

Si no se definen de manera clara los objetivos de calidad requeridos es muy difícil elegir la metodología de diseño y desarrollo más adecuada, y realizar una validación correcta de las características del producto obtenido.

Existen diversas técnicas que permiten seguir un proceso normalizado en el diseño y desarrollo del software. Ninguna de ellas destaca como la solución perfecta, y además no hay forma de evaluar las ventajas e inconvenientes que presentan cada una de ellas, por tanto es conveniente analizar en cada caso cuáles son las técnicas más adecuadas.

Entre las más importantes se podrían citar:

- Análisis de requisitos y traducción en términos de diseño y código.
- Metodologías de diseño.
- Metodologías de codificación.
- Metodologías para la realización de pruebas, tanto modulares como de la totalidad del sistema.
- Revisiones y comprobaciones.
- Prevención de errores y tolerancia de defectos.

La validación final de un software se realiza mediante la aplicación de los siguientes conceptos:

1. **Métricas:** conjunto de atributos del software, los cuales pueden ser definidos de manera precisa (ejemplo: número de instrucciones).
2. **Medidas:** formas objetivas y mecánicas de determinar los valores de las métricas.
3. **Modelos:** leyes matemáticas que relacionan entidades de una métrica. Pueden utilizarse para realizar predicciones.

Por tanto, el estudio de la fiabilidad del software es un tema de gran interés y amplia aplicación en la actualidad.

5.1.2 MEDIDA DE LA CALIDAD

Aunque hay muchas medidas de la calidad de software, la corrección, facilidad de mantenimiento, integridad y facilidad de uso suministran indicadores útiles para el equipo del proyecto [11].

1. **Corrección**: A un programa le corresponde operar correctamente o suministrará poco valor a sus usuarios. La corrección es el grado en el que el software lleva a cabo una función requerida. La medida más común de corrección son los defectos por KLDC, en donde un defecto se define como una falla verificada de conformidad con los requisitos.

2. **Facilidad de mantenimiento**: El mantenimiento del software cuenta con más esfuerzo que cualquier otra actividad de ingeniería del software. La facilidad de mantenimiento es la habilidad con la que se puede corregir un programa si se encuentra un error, se puede adaptar si su entorno cambia o optimizar si el cliente desea un cambio de requisitos. No hay forma de medir directamente la facilidad de mantenimiento; por consiguiente, se deben utilizar medidas indirectas. Una métrica orientada al tiempo simple es el tiempo medio de cambio (TMC), es decir, el tiempo que se tarda en analizar la petición de cambio, en diseñar una modificación apropiada, en efectuar el cambio, en probarlo y en distribuir el cambio a todos los usuarios. En promedio, los programas que son más fáciles de mantener tendrán un TMC más bajo (para tipos equivalentes de cambios) que los programas que son más difíciles de mantener.

3. **Integridad:** En esta época de intrusos informáticos y de virus, la integridad del software ha llegado a tener mucha importancia. Este atributo mide la habilidad de un sistema para soportar ataques (tanto accidentales como intencionados) contra su seguridad. El ataque se puede ejecutar en cualquiera de los tres componentes del software, ya sea en los programas, datos o documentos.

4. **Facilidad de uso**: El calificativo “amigable con el usuario” se ha transformado universalmente en disputas sobre productos de software. Si un programa no es “amigable con el usuario”, prácticamente está próximo al fracaso, incluso aunque las funciones que realice sean valiosas. La facilidad de uso es un intento de cuantificar “lo amigable que pude ser con el usuario” y se consigue medir en función de cuatro características:

    - Destreza intelectual y/o física solicitada para aprender el sistema.
    - El tiempo requerido para alcanzar a ser moderadamente eficiente en el uso del sistema.
    - Aumento neto en productividad (sobre el enfoque que el sistema reemplaza) medida cuando alguien emplea el sistema moderadamente y eficientemente.

- Valoración subjetiva (a veces obtenida mediante un cuestionario) de la disposición de los usuarios hacia el sistema.

Los cuatro factores anteriores son sólo un ejemplo de todos los que se han propuesto como medidas de la calidad del software.

5.1.3 MEDIDAS DE FIABILIDAD Y DE DISPONIBILIDAD

Los trabajos iniciales sobre fiabilidad buscaron extrapolar las matemáticas de la teoría de fiabilidad del hardware a la predicción de la fiabilidad del software. La mayoría de los modelos de fiabilidad relativos al hardware van más orientados a los fallos debidos al desajuste, que a los fallos debidos a defectos de diseño, ya que son más probables debido al desgaste físico (p. ej.: el efecto de la temperatura, del deterioro, y los golpes) que los fallos relativos al diseño. Desgraciadamente, para el software lo que ocurre es lo contrario. De hecho, todos los fallos del software, se producen por problemas de diseño o de implementación.

Considerando un sistema basado en computadora, una medida sencilla de la fiabilidad es el tiempo medio entre fallos (TMEF) [13], donde:

TMEF = TMDF+TMDR

(TMDF (tiempo medio de fallo) y TMDR (tiempo medio de reparación)).

Muchos investigadores argumentan que el TMDF es con mucho, una medida más útil que los defectos/KLDC, simplemente porque el usuario final se enfrenta a los fallos, no al número total de errores. Como cada error de un programa no tiene la misma tasa de fallo, la cuenta total de errores no es una buena indicación de la fiabilidad de un sistema. Por ejemplo, consideremos un programa que ha estado funcionando durante 14 meses. Muchos de los errores del programa pueden pasar desapercibidos durante décadas antes de que se detecten. El TMEF de esos errores puede ser de 50 e incluso de 100 años. Otros errores, aunque no se hayan descubierto aún, pueden tener una tasa de fallo de 18 ó 24 meses, incluso aunque se eliminen todos los errores de la primera categoría (los que tienen un gran TMEF), el impacto sobre la fiabilidad del software será muy escaso.

Además de una medida de la fiabilidad debemos obtener una medida de la disponibilidad. La disponibilidad del software es la probabilidad de que un programa funcione de acuerdo con los requisitos en un momento dado, y se define como:

Disponibilidad = TMDF/(TMDF + TMDR) x 100 %

La medida de fiabilidad TMEF es igualmente sensible al TMDF que al TMDR. La medida de disponibilidad es algo más sensible al TMDR ya que es una medida indirecta de la facilidad de mantenimiento del software.

5.1.4 EFICACIA DE LA ELIMINACIÓN DE DEFECTOS

Una métrica de la calidad que proporciona beneficios tanto a nivel del proyecto como del proceso, es la eficacia de la eliminación de defectos (EED) En particular el EED es una medida de la habilidad de filtrar las actividades de la garantía de calidad y de control al aplicarse a todas las actividades del marco de trabajo del proceso. Cuando se toma en consideración globalmente para un proyecto, EED se define de la forma siguiente:

$$EED = E / (E + D)$$

donde E= es el número de errores encontrados antes de la entrega del software al usuario final y D= es el número de defectos encontrados después de la entrega.

El valor ideal de EED es 1, donde simbolizando que no se han encontrado defectos en el software. De forma realista, D será mayor que cero, pero el valor de EED todavía se puede aproximar a 1 cuando E aumenta. En consecuencia cuando E aumenta es probable que el valor final de D disminuya (los errores se filtran antes de que se conviertan en defectos) Si se utiliza como una métrica que suministra un indicador de la destreza de filtrar las actividades de la garantía de la calidad y el control, el EED alienta a que el equipo del proyecto de software instituya técnicas para encontrar los errores posibles antes de su entrega.

Del mismo modo el EED se puede manipular dentro del proyecto, para evaluar la habilidad de un equipo en encontrar errores antes de que pasen a la siguiente actividad, estructura o tarea de ingeniería del software. Por ejemplo, la tarea de análisis de requerimientos produce un modelo de análisis qué se puede inspeccionar para encontrar y corregir errores. Esos errores que no se encuentren durante la revisión del modelo de análisis se pasan a las tareas de diseño (en donde se puede encontrar o no) Cuando se utilizan en este contexto, el EED se vuelve a definir como:

$$EED = Ei / (Ei + Ei+1)$$

Donde Ei = es el número de errores encontrados durante la actividad iesima de: ingeniería del software i, el Ei + 1 = es el número de errores encontrado durante la actividad de ingeniería del software (i + 1) que se puede seguir para llegar a errores que no se detectaron en la actividad i.

Un objetivo de calidad de un equipo de ingeniería de software es alcanzar un EED que se aproxime a 1, esto es, los errores se deberían filtrar antes de pasarse a la actividad siguiente. Esto también puede ser utilizado dentro del proyecto para evaluar la habilidad de un equipo, esto con el objetivo de encontrar deficiencias que harán que se atrase el proyecto. Existen varias métricas de calidad, pero las más importantes y que engloban a las demás, son sólo cuatro: corrección, facilidad de mantenimiento, integridad y facilidad de uso.

5.2 TIPOS DE METRICAS

5.2.1 MÉTRICAS DE ANÁLISIS

En esta fase se obtendrán los requisitos y se establecerá el fundamento para el diseño. Y es por eso que se desea una visión interna a la calidad del modelo de análisis. Sin embargo hay pocas métricas de análisis y especificación, se sabe que es posible adaptar métricas obtenidas para la aplicación de un proyecto, en donde las métricas examinan el modelo de análisis con el fin de predecir el tamaño del sistema resultante, en donde resulte probable que el tamaño y la complejidad del diseño estén directamente relacionados. Es por eso que se verán en las siguientes secciones las métricas orientadas a la función, la métrica bang y las métricas de la calidad de especificación.

5.2.1.1 MÉTRICAS BASADAS EN LA FUNCIÓN

La métrica de punto de función (PF) se puede usar como medio para predecir el tamaño de un sistema que se va a obtener de un modelo de análisis. Para instruir el empleo de la métrica PF, se considerará una sencilla representación del modelo de análisis mostrada [7] en la figura siguiente.

En dicha figura se representa un diagrama de flujo de datos, de una función de una aplicación de software llamada Hogar Seguro. La función administra la interacción con el usurario, aceptando una contraseña de usuario para activar/ desactivar el sistema y permitiendo consultas sobre el estado de las zonas de seguridad y varios censores de seguridad. La función muestra una serie de mensajes de petición y envía señales apropiadas de control a varios componentes del sistema de seguridad. El diagrama de flujo de datos se evalúa para determinar las medidas clave necesarias para el cálculo de la métrica de PF:

- Número de entradas de usuario.
- Número de salidas de usuario.
- Número de consultas del usuario.
- Número de archivos.
- Número de interfaces externas.

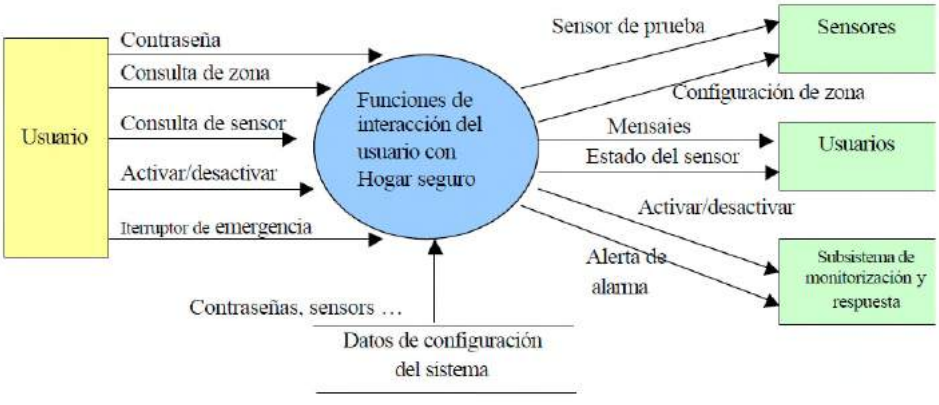


Figura 5. Diagrama de flujo de Hogar Seguro.

Hay tres entradas del usuario: contraseña, interruptor de emergencias y activar/desactivar aparecen en la figura con dos consultas: consulta de zona y consulta de sensor. Se muestra un archivo (archivo de configuración del sistema) También están presentes dos salidas de usuarios (mensajes y estado de del sensor) y cuatro interfaces externas (sensor de prueba, configuración de zona, activar/desactivar y alerta de alarma).

Parámetro de medición	Cuenta	Simple	Media	Compleja	Total
Número de entradas de usuario	3 *	3	4	6	=9
Número de salidas de usuario	2 *	4	5	7	=8
Número de consultas del usuario	2 *	3	4	6	=6
Número de archivos	1 *	7	10	15	=7
Número de interfaces externas	4 *	5	7	10	= 20
Cuenta total					**50**

Figura 5.4: Factor de ponderación.

La cuenta total mostrada en la figura debe ajustarse usando la ecuación:

PF = cuenta-total * (0.65 + 0.01 * ÓFi)

Donde cuenta-total es la suma de todas las entradas PF obtenidas de la figura anterior y Fi(i=1 a 14) son “valores de ajuste de complejidad”. Para el propósito de este ejemplo, asumimos que ÓFi es 46 (un producto moderadamente complejo).

Por lo tanto:

PF = 50* [0.65+(0.01*46)]=56

Basándose en el valor previsto de PF obtenido del modelo de análisis, el equipo del proyecto puede estimar el tamaño global de implementación de las funciones de Hogar Seguro. Asuma que los datos de los que se disponen indican que un PF supone 60 líneas de código (si se usa un lenguaje orientado a objetos) y que en un esfuerzo de un mes-persona se producen 12 PF. Estos datos históricos proporcionan al administrador del proyecto una importante información de planificación basada en el modelo de análisis en lugar de en estimaciones preliminares.

5.2.1.2 LA MÉTRICA BANG

Puede emplearse para desarrollar una indicación del tamaño del software a implementar como consecuencia del modelo de análisis. Desarrollada por Tom DeMarco [11], la métrica bang es “una indicación, independiente de la implementación, del tamaño del sistema” [11].

Para calcular la métrica bang, el desarrollador de software debe evaluar primero un conjunto de primitivas (elementos del modelo de análisis que no se subdividen más en el nivel de análisis) Las primitivas se determinan evaluando el modelo de análisis y desarrollando cuentas para los siguientes elementos:

- Primitivas funcionales (Pfu) Transformaciones (burbujas) que aparecen en el nivel inferior de un diagrama de flujo de datos.

- Elementos de datos (ED) Los atributos de un objeto de datos, los elementos de datos no compuestos y aparecen en el diccionario de datos.
- Objetos (OB) Objetos de datos.

- Relaciones (RE) Las conexiones entre objetos de datos.

- Transiciones (TR) El número de transacciones de estado en el diagrama de transición de estado.

Además de las seis primitivas nombradas arriba, se determinan medidas adicionales para:

- Primitivas modificadas de función manual (PMFu) Funciones que caen fuera del límite del sistema y que deben modificarse para acomodarse al nuevo sistema.

- Elementos de datos de entrada (EDE) Aquellos elementos de datos que se introducen en el sistema.

- Elementos de datos de salida (EDS) Aquellos elementos de datos que se sacan en el sistema.

- Elementos de datos retenidos (EDR) Aquellos elementos de datos que son retenidos (almacenados) por el sistema.

- Muestras (tokens) de datos (TCi) Las muestras de datos (elementos de datos que no se subdividen dentro de una primitiva funcional) que existen en el l’imite de la i-ésima primitiva funcional (evaluada para cada primitiva).

- Conexiones de relación (Rei) Las relaciones que conectan el i-ésimo objeto en el modelo de datos con otros objetos.

5.2.1.3 MÉTRICAS DE LA CALIDAD DE ESPECIFICACIÓN

Existe una lista de características para poder valorar la calidad del modelo de análisis y la correspondiente especificación de requisitos: especificidad, corrección, compleción, comprensión, capacidad de verificación, consistencia externa e interna, capacidad de logro, concisión, traza habilidad, capacidad de modificación, exactitud y capacidad de reutilización.

5.2.2 MÉTRICAS DE DISEÑO

Las métricas para software, como otras métricas, no son perfectas; muchos expertos argumentan que se necesita más experimentación hasta que se puedan emplear bien las métricas de diseño. Sin embargo el diseño sin medición es una alternativa inaceptable. A continuación se mostraran algunas de las métricas de diseño más comunes. Aunque ninguna es perfecta, pueden proporcionarle al diseñador una mejor visión interna y así el diseño evolucionara a un mejor nivel de calidad.

5.2.2.1 MÉTRICAS DE DISEÑO DE ALTO NIVEL

Éstas se concentran en las características de la estructura del programa dándole énfasis a la estructura arquitectónica y en la eficiencia de los módulos. 83 Estas métricas son de caja negra, en el sentido de que no se requiere ningún conocimiento del trabajo interno de ningún modo en particular del sistema. Card y Glass [7] proponen tres medidas de complejidad del software: complejidad estructural, complejidad de datos y complejidad del sistema.

El Mando de sistemas de la Fuerza Aérea Americana [7] ha desarrollado varios indicadores de calidad del software basados en características medibles del diseño de un programa de computadora. Empleando conceptos similares a los propuestos en IEEE Std en 1988, la Fuerza Aérea utiliza información obtenida del diseño de datos y arquitectónico para obtener un índice de calidad de la estructura del diseño (ICED) que va desde 0 a 1. Se deben comprobar los siguientes valores para calcular el ICED [7]:

- SI = el número total de módulos definidos en la arquitectura del programa.
- S2 = el número de módulos que para funcionar correctamente dependen de la fuente de datos de entrada o producen datos que se van a emplear en algún otro lado en general, los módulos de control (entre otros) no formarían parte de S2.
- S3 = el número de módulos que para que funcionen correctamente dependen de procesos previos.
- S4 = el número de elementos de base de datos (incluye todos los objetos de datos y todos los atributos que definen los objetos).
- S5 = el número total de elementos únicos de base de datos.

- S6 = el número de segmentos de base de datos (registros diferentes u objetos individuales).

- S7 = el número de módulos con una salida y una entrada (el proceso de excepciones no se considera de salida múltiple).

Una vez que se han determinado los valores SI a S7 para un programa de computadora, se pueden calcular los siguientes valores intermedios:

- Estructura del programa: D1, donde D1 se define como sigue: si el diseño arquitectónico se desarrolló usando un método determinado (por ejemplo, diseño orientado a flujo de datos o diseño orientado a objetos), entonces D1=1, de otra manera D1=0.

- Independencia del módulo: D2=1 – (S2/S1).

- Módulos no dependientes de procesos previos: D3=1 – (S3/S1).

- Tamaño de la base de datos: D4=1 – (S5/S4).

- Compartido de la base de datos: D5=1-(S6/S4).

- Características de entrada/salida del módulo: D6=1 – (S7/S1).

Con estos valores intermedios calculados, el ICED se obtiene de la siguiente manera:

ICED=_ Wi Di (4.25)

donde i = 1 a 6, Wi es el peso relativo de la importancia de cada uno de los valores intermedios y_W i =1 (si todos los Di tiene el mismo peso, entonces Wi=0.167).

Se puede determinar y compartir el valor de ICED de diseños anteriores con un diseño actualmente en desarrollo. Si el ICED es significativamente inferior a la medida, se aconseja seguir trabajando en el diseño y revisión. Igualmente, si hay que hacer grandes cambios a un diseño ya existente, se puede calcular el efecto de estos cambios en el ICED.

5.2.2.2 MÉTRICAS DE DISEÑO EN LOS COMPONENTES

Las métricas de diseño a nivel de componentes se concentran en las características internas de los componentes del software e incluyen medidas de la cohesión, acoplamiento y complejidad del módulo. Estas tres medidas pueden ayudar al desarrollador de software a juzgar la calidad de un diseño a nivel de componentes. Las métricas presentadas son de “caja blanca” en el sentido de que requieren conocimiento del trabajo interno del módulo en cuestión. Las métricas de diseño en los componentes se pueden aplicar una vez que se ha desarrollado un diseño procedimental. También se pueden retrasar hasta tener disponible el código fuente.

5.2.2.2.1 MÉTRICAS DE COHESIÓN

Bieman y Ott [Hamdi '99] definen una colección de métricas que proporcionan una indicación de la cohesión de un módulo. Las métricas se definen con cinco conceptos y medidas:

- **Porción de datos**: dicho simplemente, una porción de datos es una marcha atrás a través de un módulo que busca valores de datos que afectan a la localización del módulo en el que empezó la marcha atrás. Debería resaltarse que se pueden definir tanto porciones de programas (que se centran en enunciados y condiciones) como porciones de datos.

- **Símbolos léxicos (tokens) de datos**: las variables definidas para un módulo pueden definirse como señales de datos para el módulo.

- **Señales de unión**: el conjunto de señales de datos que se encuentran en uno o más porciones de datos.

- **Señales de super-unión**: las señales de datos comunes a todas las porciones de datos de un módulo.

- **Cohesión**: la cohesión relativa de una señal de unión es directamente proporcional al número de porciones de datos que liga.

Todas estas métricas de cohesión tienen valores que van desde 0 a 1. Tienen un valor de 0 cuando un procedimiento tiene más de una salida y no muestra ningún atributo de cohesión indicado por una métrica particular. Un procedimiento sin señales de super-unión, sin señales comunes a todas las porciones de datos, no tiene una cohesión funcional fuerte (no hay señales de datos que contribuyan a todas las salidas).

Un procedimiento sin señales de unión, es decir, sin señales comunes a más de una porción de datos (en procedimientos con más de una porción de datos), no muestra una cohesión funcional débil y ninguna adhesividad (no hay señales de datos que contribuyan a más de una salida) La cohesión funcional fuerte y la pegajosidad se obtienen cuando las métricas de Bieman y Ott toman un valor máximo de 1.

5.2.2.2.2 MÉTRICAS DE ACOPLAMIENTO

El acoplamiento de módulo proporciona una indicación de la "conectividad" de un módulo con otros módulos, datos globales y entorno exterior. Dhama [15] ha propuesto una métrica para el acoplamiento del módulo que combina el acoplamiento de flujo de datos y de control: acoplamiento global y acoplamiento de entorno. Las medidas necesarias para calcular el acoplamiento de módulo se definen en términos de cada uno de los tres tipos de acoplamiento apuntados anteriormente. Para el acoplamiento de flujo de datos y de control:

di = número de parámetros de datos de entrada.
ci = número de parámetros de control de entrada.

do = número de parámetros de datos de salida.
co = número de parámetros de control de salida.

Para el acoplamiento global gd = número de variables globales usadas como datos gc = numero de variables globales usadas como control.

Para el acoplamiento de entorno:

w = número de módulos llamados (expansión).
r = número de módulos que llaman al módulo en cuestión (concentración).

Usando estas medidas, se define un indicador de acoplamiento de módulo, .mc, de la siguiente manera:

mc = k/M

donde k = 1 es una constante de proporcionalidad.

M = di + a* ci + do + b*co +gd + c *gc + w + r

donde: a=b=c=2 Cuanto mayor es el valor de mc, menor es el acoplamiento de módulo.

5.2.2.2.3 MÉTRICAS DE COMPLEJIDAD

Se pueden calcular una variedad de métricas del software para determinar la complejidad del flujo de control del programa. Muchas de éstas se basan en una representación denominada grafo de flujo, un grafo es una representación compuesta de nodos y enlaces (también denominados filos) Cuando se dirigen los enlaces (aristas), el grafo de flujo es un grafo dirigido. McCabe [14] identifica un número importante de usos para las métricas de complejidad, donde pueden emplearse para predecir información sobre la fiabilidad y mantenimiento de sistemas software, también realimentan la información durante el proyecto de software para ayudar a controlar la actividad de diseño, en las pruebas y mantenimiento, proporcionan información sobre los módulos software para ayudar a resaltar las áreas de inestabilidad.

La métrica de complejidad más ampliamente usada (y debatida) para el software es la complejidad ciclomática, originalmente desarrollada por Thomas McCabe. La métrica de McCabe proporciona una medida cuantitativa para probar la dificultad y una indicación de la fiabilidad última. Estudios experimentales indican una fuerte correlación entre la métrica de McCabe y el número de errores que existen en el código fuente, así como el tiempo requerido para encontrar y corregir dichos errores. McCabe [7] también defiende que la complejidad ciclomática puede emplearse para proporcionar una indicación cuantitativa del tamaño máximo del módulo. Recogiendo datos de varios proyectos de programación reales, ha averiguado que una complejidad ciclomática de 10 parece ser el límite práctico superior para el tamaño de un módulo, Cuando la complejidad ciclomática de los módulos excedían ese valor, resultaba extremadamente difícil probar adecuadamente el módulo.

5.2.3 MÉTRICAS DE INTERFAZ

Aunque existe una significativa cantidad de literatura sobre el diseño de interfaces hombre-máquina, se ha publicado relativamente poca información sobre métricas que proporcionen una visión interna de la calidad y facilidad de empleo de la interfaz. 94 Sears [7] sugiere la conveniencia de la representación (CR) como una valiosa métrica de diseño para interfaces hombre-máquina. Una IGU (Interfaz Gráfica de Usuario) típica usa entidades de representación, iconos gráficos, texto, menús, ventanas y otras para ayudar al usuario a completar tareas. Para realizar una tarea dada usando una IGU, el usuario debe moverse de una entidad de representación a otra. Las posiciones absolutas y relativas de cada entidad de representación, la frecuencia con que se utilizan y el "costo" de la transición de una entidad de representación a la siguiente contribuirán a la conveniencia de la interfaz.

Para una representación específica (p. ej.: un diseño de una IGU específica), se pueden asignar costos a cada secuencia de acciones de acuerdo con la siguiente relación:

Costos = Ó[frecuencia de transición (ki) x costos de transición (ki)]

donde k es la transición i específica de una entidad de representación a la siguiente cuando se realiza una tarea específica. Esta suma se da con todas las transiciones de una tarea en particular o conjunto de tareas requeridas para conseguir alguna función de la aplicación. El costo puede estar caracterizado en términos de tiempo, retraso del proceso o cualquier otro valor razonable, tal como la distancia que debe moverse el ratón entre entidades de la representación.

La conveniencia de la representación se define como:

CR = 100 x [(costo de la representación óptima CR)/ (costo de la representación propuesta)] . (4.31)

donde CR = para una representación óptima.

Para calcular la representación óptima de una IGU, la superficie de la interfaz (el área de la pantalla) se divide en una cuadrícula. Cada cuadro de la cuadrícula representa una posible posición de una entidad de la representación. Para una cuadrícula con N posibles posiciones y K diferentes entidades de representación para colocar, el número posible de distribuciones se representa de la siguiente manera [7]:

Número posible de distribuciones = [N !/ (K! * (N - K)!] * K! (4.32)

La CR se emplea para valorar diferentes distribuciones propuestas de IGU y la sensibilidad de una representación en particular a los cambios en las descripciones de tareas (por ejemplo, cambios en la secuencia y/o frecuencia de transiciones) Es importante apuntar que la selección de un diseño de IGU puede guiarse con métricas tales como CR, pero el árbitro final debería ser la respuesta del usuario basada en prototipos de IGU. Nielsen Levy [7] informa; que puede haber una posibilidad de éxito si se prefiere la interfaz basándose exclusivamente en la opinión del usuario ya que el

rendimiento medio de tareas de usuario y su satisfacción con la IGU están altamente relacionadas.

5.2.4 MÉTRICAS DE CÓDIGO FUENTE

La teoría de Halstead de la ciencia del software es 'probablemente la mejor conocida y más minuciosamente estudiada... medidas compuestas de la complejidad (software)' [11]. La ciencia software propuso las primeras 'leyes' analíticas para el software de computadora. La ciencia del software asigna leyes cuantitativas al desarrollo del software de computadora. La teoría de Halstead se obtiene de un supuesto fundamental [7]: 'el cerebro humano sigue un conjunto de reglas más rígido (en el desarrollo de algoritmos) de lo que se imagina...'.

La ciencia del software usa un conjunto de medidas primitivas que pueden obtenerse una vez que se ha generado o estimado el código después de completar el diseño. Estas medidas se listan a continuación.

n1: el número de operadores diferentes que aparecen en el programa.
n2: el número de operandos diferentes que aparecen en el programa.
N1: el número total de veces que aparece el operador.
N2: el número total de veces que aparece el operando.

Halstead usa las medidas primitivas para desarrollar expresiones para la longitud global del programa; volumen mínimo potencial para un algoritmo; el volumen real (número de bits requeridos para especificar un programa); el nivel del programa (una medida de la complejidad del software); nivel del lenguaje (una constante para un lenguaje dado); y otras características tales como esfuerzo de desarrollo, tiempo de desarrollo e incluso el número esperado de fallos en el software. Halstead expone que la longitud N se puede estimar como:

$$N = n1 \log_2 n1 + n2 \log_2 n2$$

y el volumen de programa se puede definir como:

$$V = N \log_2 (n1 + n2)$$

Se debería tener en cuenta que V variará con el lenguaje de programación y representa el volumen de información Teóricamente debe existir un volumen mínimo para un algoritmo. Halstead define una relación de volumen L como la relación volumen de la forma más compacta de un programa respecto al volumen real del programa. Por tanto L, debe ser siempre menor de uno. En términos de medidas primitivas, la relación de volumen se puede expresar como:

$$L = 2/n1*n2/N2$$

Halstead propuso que todos los lenguajes pueden categorizarse por nivel de lenguaje, I, que variará según los lenguajes. Halslead creó una teoría que suponía que el nivel de lenguaje es constante para un lenguaje dado, pero otro trabajo [7] indica que el nivel de

lenguaje es función tanto del lenguaje como del programador. Los siguientes valores de nivel de lenguaje se muestran en la tabla siguiente:

Lenguaje	Media/.I
Inglés prosaico	2.16
PL/1	1.53
ALGOL/68	2.12
FORTRAN	1.14
Ensamblador	0.88

Figura 5.5: Valores de Nivel de Lenguaje.

Parece que el nivel de lenguaje implica un nivel de abstracción en la especificación procedimental. Los lenguajes de alto nivel permiten la especificación del código a un nivel más alto de abstracción que el lenguaje ensamblador (orientado a máquina).

5.2.5 MÉTRICAS PARA PRUEBAS

Aunque se ha escrito mucho sobre métricas del software para pruebas, la mayoría de las métricas propuestas se concentran en el proceso de pruebas, no en las características técnicas de las pruebas mismas. En general, los responsables de las pruebas deben fiarse del análisis, diseño y código para que les guíen en el diseño y ejecución los casos de prueba.

Las métricas basadas en la función pueden emplearse para predecir el esfuerzo global de las pruebas Se pueden juntar y correlacionar varias características a nivel de proyecto (p ej.: esfuerzo y tiempo las pruebas, errores descubiertos, número de casos de prueba producidos) de proyectos anteriores con el número de Pf producidos por un equipo del proyecto. El equipo puede después predecir los valores ‘esperados’ de estas características del proyecto actual.

La métrica bang puede proporcionar una indicación del número de casos de prueba necesarios para examinar las medidas primitivas. El número de primitivas funcionales (Pfu), elementos de datos, (ED), objetos (0B), relaciones (RE), estados (ES) y transiciones (TR) pueden emplearse para predecir el número y tipos de pruebas de la caja negra y blanca del software.

Las métricas de diseño de alto nivel proporcionan información sobre facilidad o dificultad asociada con la prueba de integración y la necesidad de software especializado de prueba (p. ej.: matrices y controladores) La complejidad ciclomática (una métrica de diseño de componentes) se encuentra en el núcleo de las pruebas de caminos básicos, un método de diseño de casos de prueba. Además, la complejidad ciclomática puede usarse para elegir módulos como candidatos para pruebas más profundas. Los módulos con gran complejidad ciclomática tienen más probabilidad de tendencia a errores que los módulos con menor complejidad ciclomática.

Por esta razón, el analista debería invertir un esfuerzo extra para descubrir errores en el módulo antes de integrarlo en un sistema. El esfuerzo de las pruebas también se puede estimar usando métricas obtenidas de medidas de Halstead. Usando la definición del volumen de un programa V, y nivel de programa, NP, el esfuerzo de la ciencia del software, puede calcularse como:

NP=1 / [(n1 / 2) * (N2 / n2)]

e = V / NP

El porcentaje del esfuerzo global de pruebas a asignar un módulo k para estimar usando la siguiente relación:

porcentaje de esfuerzo de pruebas (k) = e(k)/ Óe(i)

donde e(k) se calcula para el módulo k usando las ecuaciones y suma en el denominador de la ecuación es la suma del esfuerzo de la ciencia del software a lo largo de todos los módulos del sistema.

A medida que se van haciendo las pruebas tres medidas diferentes proporcionan una indicación de la complexión de las pruebas. Una medida de la amplitud de las pruebas proporciona una indicación de cuantos (del número total de ellos) se han probado. Esto proporciona una indicación de la complexión del plan de pruebas. La profundidad de las prueba es una medida del porcentaje de los caminos básicos independientes probados en relación al número total de estos caminos en el programa. Se puede calcular una estimación razonablemente exacta del número de caminos básicos sumando la complejidad ciclomática de todos los módulos del programa.

Finalmente, a medida que se van haciendo las pruebas y se recogen los datos de los errores, se pueden emplear los perfiles de fallos para dar prioridad y categorizar los errores descubiertos. La prioridad indica la severidad del problema. Las categorías de los fallos proporcionan una descripción de un error, de manera que se puedan llevar a cabo análisis estadísticos de errores.

5.2.6 MÉTRICAS DE MANTENIMIENTO

Se han propuesto métricas diseñadas explícitamente para actividades de mantenimiento. El estándar IEEE 982.1-1988 [7] sugiere un índice de madurez del software (IMS) que proporciona una indicación de la estabilidad de un producto de software (basada en los

cambios que ocurren con cada versión del producto) Se determina la siguiente información:

Mr = número de módulos en la versión actual.
Fc = número de módulos en la versión actual que se han cambiado.
Fa = número de módulos en la versión actual que se han añadido.
Fd = número de módulos de la versión anterior que se han borrado en la versión actual.

El índice de madurez del software se calcula de la siguiente manera:

IMS = [Mr – (Fa + Fc + Fd)]/ Mr

A medida que el IMS se aproxima a 1.0 el producto se empieza a estabilizar. El IMS puede emplearse también como métrica para la planificación de las actividades de mantenimiento del software. El tiempo medio para producir una versión de un producto software puede correlacionarse con el IMS desarrollándose modelos empíricos para el mantenimiento.

6. MODELOS DE FIABILIDAD DEL SOFTWARE

La ingeniería de fiabilidad de software fue primero introducida como una disciplina durante la Segunda Guerra Mundial para evaluar la probabilidad de éxito de los misiles y demás artefactos de guerra. En los años 50's empezaron a conocerse métodos avanzados para estimar la vida útil de componentes mecánicos, electrónicos y eléctricos utilizados en la defensa y la industria aeroespacial. Para los 60's, la ingeniería de fiabilidad se convirtió a sí misma como una parte integral en el desarrollo de productos comerciales enfocados en el concepto de usuario final.

La disciplina de la fiabilidad de software es mucho más joven y reciente, comenzando a mediados de los 70's cuando el ambiente de desarrollo de software se encontraba en un punto estable y equilibrado.

Muchos de los modelos de fiabilidad de software fueron desarrollados durante este tiempo de estabilidad aunque años después, es decir en los 80's comenzaron a surgir nuevas tecnologías, nuevos paradigmas, nuevos conceptos de análisis estructurados y nuevas alternativas de desarrollo que hasta la fecha siguen evolucionando y revolucionando todo lo que tiene que ver con calidad y desarrollo de software.

Cada día más y más sistemas se hacen parte de la vida diaria y de ellos pueden depender elementos muy importantes como la vida humana. Todos estos sistemas tienen una gran dependencia sobre el software que los maneja, lo que constituye una prioridad que éste sea lo más confiable posible dentro de su desarrollo y operación. En el proceso de abordar este tema se han tratado de reducir los "errores" del software en general, pasando por conceptos como fallos, defectos y normas, que la calidad de software impone para este tipo de desarrollos.

Los Ingenieros de software no tienen puntos de vista análogos al tratar el tema de fiabilidad. Algunos definen confiable como "lograr el mismo resultado de éxito en pruebas sucesivas", esta definición es un poco simple si se tienen puntos de vista como el ciclo de vida útil del sistema y las condiciones en las que operará.

En general, las métricas para la fiabilidad de software son usadas para describir la probabilidad de operación del software en un ambiente determinado, con un rango de entradas designadas sin que se produzca un fallo. La fiabilidad de software es definida entonces como la probabilidad de que el software no causará fallos al sistema sobre un tiempo determinado y bajo unas condiciones específicas, que son definidas por parte del usuario.

Esta probabilidad es una función de las entradas y uso que se tiene del sistema, teniendo en cuenta la presencia latente de fallos. Las entradas del sistema pueden ayudar a determinar la posible aparición de fallos durante la operación del sistema.

Es importante destacar la importancia que tiene el levantamiento de requisitos de un sistema y los planes de contingencia que se logren idear durante la ejecución del mismo. Se debe tener claro que no es posible que exista un software 100% confiable, ya que si partimos del supuesto que es hecho por humanos siempre existirá una probabilidad, aunque sea pequeña de que falle.

El potencial de la fiabilidad, consiste en que permite plantear mejoras en la gestión del software antes de empezar con la generación de código (coding), y las pruebas (el testing).

El objetivo del proceso de estimación es determinar el número de fallos residentes en el software, por lo que solo a priori de la duración/complejidad del test pueden sacarse fallos.

Los modelos de fiabilidad de software han existido desde la época de los 70's; alrededor de 200 han sido desarrollados. Muchos de los modelos nuevos son construidos con base en la teoría de los modelos viejos. Lo que se trata con la fiabilidad de software es seleccionar los mejores modelos que se puedan aplicar en el día de hoy y ayuden de manera importante a la ingeniería de fiabilidad.

En la siguiente figura tenemos una comparación entre los modelos de estimación y de predicción.

COMPARANDO MODELOS DE ESTIMACION Y PREDICCION

Elementos	Modelos de predicción	Modelos de estimación
Datos de Referencia	Usa datos históricos	Usa datos en “pruebas de esfuerzo” del software que se esta desarrollando
Cuando se usa en el ciclo de desarrollo	Normalmente se realiza antes del desarrollo o en fases de test. El modelo escogido puede ser usado desde el mismo momento en que se concibe la idea	Normalmente se realiza mas tarde, en el ciclo de vida (después de que algunos datos han sido recogidos); no se suele usar en las fases de la idea o desarrollo
Franja de Tiempo	Predice la fiabilidad en algún tiempo futuro	Estima la fiabilidad tanto en el presente como en algún momento futuro

Figura 6.1: Comparación Modelos Estimación vs. Predicción.

6.1 MODELOS DE PREDICCIÓN

Los modelos más básicos de predicción utilizan como premisa fundamental la organización interna de los datos, basada en experiencias obtenidas y pruebas durante el desarrollo de las predicciones.

Hasta la época, cuatro modelos fundamentales han sido desarrollados, estos son: Modelo de Tiempo de Ejecución de Musa, Modelo de Putnam, y dos modelos desarrollados en el laboratorio de Roma, denotados por sus números técnicos que son: el TR-92-52 y TR-92-15.
Cada modelo de predicción, sus capacidades y la descripción de sus salidas esta recogido en la siguiente figura:

Modelo de	Capacidades	Descripción de Resultados
Modelo de colección de datos históricos	Puede ser muy exacto si hay un amplio compromiso por parte de la organización.	Produce una predicción de la tasa de fallos del software liberado basada sobre datos históricos que existen en la empresa
Modelo de Musa	Predice la tasa de fallos al comenzar las pruebas del sistema, lo anterior puede ser utilizado después en modelos de crecimiento de fiabilidad.	Produce una predicción de la tasa de fallos al comenzar las pruebas del sistema.
Modelo de Putnam	Predice fallos sobre el tiempo y este resultado puede ser utilizado con otros modelos de predicción.	Produce una predicción de los fallos sobre el tiempo de vida del proyecto.
Modelo TR-92-52	Predice la densidad de fallos durante el ciclo de vida del proyecto	Produce una predicción en términos de densidad de fallos o numero estimado de fallos inherentes.
Modelo TR-92-15	Posee factores para la estimación del numero de fallos	Estima fallos durante cada fase del desarrollo.

Figura 6.2: Modelos de predicción.

6.1.1 MODELO DE COLECCIÓN DE DATOS HISTÓRICOS INTERNOS

Pocas organizaciones predicen la fiabilidad de software mediante la colección y el uso de la información acumulada en las bases de datos de todos los proyectos que se han realizado. Métricas que han sido empleadas incluyen el producto, la dirección del proyecto y los indicadores de fallos.

El análisis de regresión estadística es típicamente utilizado para desarrollar la ecuación de predicción para cada característica importante del proyecto. Toda la información que se puede obtener de estos estudios es utilizada por la dirección para tratar de hacer cada vez productos más confiables y de mejor calidad.

6.1.2 MODELO DE TIEMPO DE EJECUCIÓN DE MUSA [6]

Desarrollado por John Musa a mediados de los 70's, fue uno de los modelos más tempranos para la predicción de la fiabilidad de software. Este modelo trata de predecir la tasa inicial de fallo (intensidad) de un sistema de software hasta el momento en que las pruebas del producto comienzan. La intensidad inicial de fallos (fallos por unidad de tiempo), es una función desconocida pero estimada mediante un número total de fallos esperadas en un periodo de tiempo infinito, N. La ecuación de predicción se ilustra a continuación, los términos son explicados en la tabla.

$$\lambda_0 = k \times p \times w_o$$

Símbolo	Representación	Valor
K	Constante que acumula para estructuras dinámicas de los programa	K=4.2E-7
P	Estimación del numero de ejecuciones por unidad de tiempo	p=r/SL* C/ER
R	Promedio de la tasa de ejecución de instrucciones, determinada por el constructor	Constante
SLOC	Líneas de código fuente (no se incluye código reusado)	
ER	Razón de expansión, una constante que depende del lenguaje	Assembler, C++, Java, VB
W o	Estimación del numero inicial de fallos en el programa	Puede ser calculado usando W o = N * B * un valor por defecto de 6(fallos/(000)SL* C
N	Numero total de ocurrencia de fallos	Estimación basada en el juicio o experiencia pasada
B	Fallos no corregidas antes de que el producto sea liberado.	Se asume B=0.95

Figura 6.3: Términos en el modelo de tiempo de ejecución de Musa [6].

Por ejemplo, un programa de 100 líneas de FORTRAN, con un ratio de ejecución aproximada de 150 líneas por segundo, tiene un ratio de fallo esperado, cuando el sistema empieza, de:

λo= k x p x wo=(4.2E-7) x (150/100/3) x (6/100)=.012E-7=1.26E-9 fallos por segundo (o una fallo por 79365E8 segundos, lo que es equivalente a 1 fallo cada 25,17 años).

Es importante hacer constar, que la medida del tiempo es tiempo de ejecución, no de calendario, el modelo de Musa esta pensado exclusivamente para tiempo de ejecución.

Este tiempo se inicia con la puesta en memoria principal del programa, por lo que el sistema operativo comienza a ejecutar sus instrucciones. El intervalo finaliza en el momento en que éste envía al sistema operativo la señal de terminación, sea ésta una terminación normal, en que el programa tuvo la posibilidad de concluir sus instrucciones satisfactoriamente, o una terminación anormal, en el que el programa produjo algún error y el sistema debió forzar su finalización.

Los modelos de fiabilidad del hardware son normalmente en términos de calendario, no es factible usar el Modelo de predicción de Musa en el desarrollo de la estimación de todo un sistema de fiabilidad tanto de hardware como de software, a menos que se quiera asumir que el tiempo de calendario y el tiempo de ejecución es el mismo (esta presunción no suele ser valida).

6.1.3 MODELO DE PUTNAM [17]

Este modelo fue desarrollado mediante el estudio de las historias de los defectos que se iban tratando y documentando de los diversos proyectos, variando por el tipo de aplicación que se realizaba.

Basado en lo anterior Putnam asignó una distribución general normalizada, donde K y a son constantes que provienen de entradas de los datos y t es el tiempo en meses; la ecuación es la siguiente:

$$R(t) = k \exp(-at^2)$$

La función de densidad de probabilidad, f (t), la derivada de R (t) con respecto a t, es de la forma general:

$$f(t) = 2ak\ t \exp(-at^2)$$

Surge en 1978 como solución a un requerimiento de la marina de EEUU para proveer un método para estimar esfuerzo y tiempo. Fue desarrollado por Putnam y lo llamó modelo SLIM.

Se utiliza para proyectos con más de 70.000 LOC, aunque puede ser ajustado para proyectos más pequeños.

Asume que el esfuerzo para proyectos de desarrollo de software es distribuido de forma similar a una colección de curvas de Rayleigh, una para cada una de las actividades principales del desarrollo.

Predice el ratio de fallo inicial (intensidad) de un sistema software en el momento en que el testeo del sistema de software comienza (por ejemplo cuando el tiempo t=0).

Putnam asigno la distribución normalizada de Rayleigh para describir la fiabilidad observada, donde k y a son constantes ajustadas desde los datos y t es el tiempo, en meses:

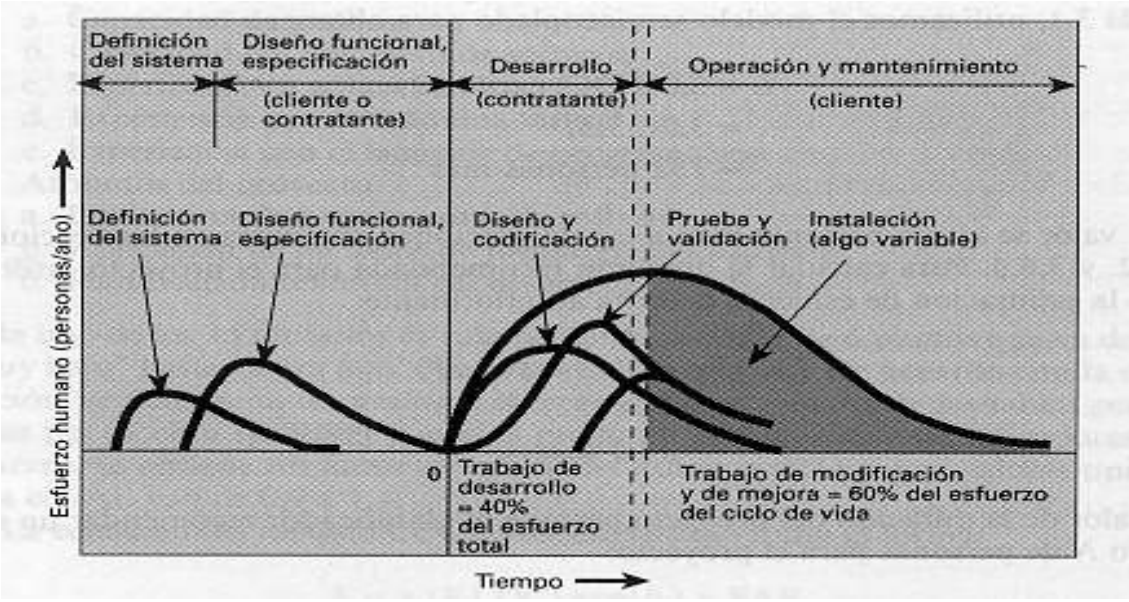


Figura 6.4: Distribución normalizada de Rayleigh.

Putnam desarrolló una escala ordinal mediante peldaños para representar cualquier proceso de desarrollo, estimado para esos tiempos y aplicable a la fecha (ilustrada en la tabla (2). Es de especial interés el peldaño 7, denotado por td (tiempo de desarrollo), correspondiente al final de las fases de desarrollo y el comienzo de la capacidad operacional del producto.

Este punto fue definido como la ocurrencia en el 95th percentil (es decir que el 95% de todos los defectos han sido detectados en ese punto durante el desarrollo de software).

Utilizando td como la referencia básica, Putnam desarrolló las expresiones de las constantes para el modelo, a y k, en términos de N y td. La ecuación final predice el número esperado de defectos por mes como una función del cronograma en meses y el número total de defectos inherentes, N, que es tomado por:

$$f(t) = (6N/t_d^2)\, t \exp(-3t^2/t_d^2)$$

INDICADOR #	INDICADOR
0	ESTUDIO DE VIABILIDAD
1	ANALISIS PRELIMINAR DEL DISEÑO, FUNCION DE DISEÑO COMPLETA
2	ANALISIS DE DISEÑO CRITICO, DISEÑO COMPLETADO
3	PRIMER CODIGO COMPLETO
4	PRUEBA DE INTEGRACION DE INICIO DEL SISTEMA
5	PRUEBA DE USUSARIOS DEL SISTEMA
6	CAPACIDAD OPERACIONAL INICIAL, INSTALACION
7	cAPACIDAD OPERACIONAL TOTAL; CONFIABILIDAD DEL 95% EN LAS RUTINAS USADAS
8	ALCANZAR EL 99% DE CONFIABILIDAD POR MEDIO DE ACENTUAMIENTO EN LAS PRUEBAS
9	ALCANZAR EL 99.9% DE CONFIABILIDAD, ASUMIENDO ELIMINACION DE ERRORES

Figura 6.5 Indicador de niveles de Putnam.

Por ejemplo, supongamos un programa que esta siendo desarrollado en FORTRAN, el plan es que este sea plenamente operacional (escalón 7) en 10 meses de calendario, resultando td^2 en 10^2 o 100.

Los defectos esperados por mes durante el desarrollo son calculados usando la expresión:

$$f(t) = .06\ N\ t \exp(-.03t^2)$$

El calculo del resultado se muestra en la siguiente figura, donde t es el numero del mes, f(t) es la proporción esperada del numero total de defectos observados en el mes t, y F(t) representa la proporción acumulativa.

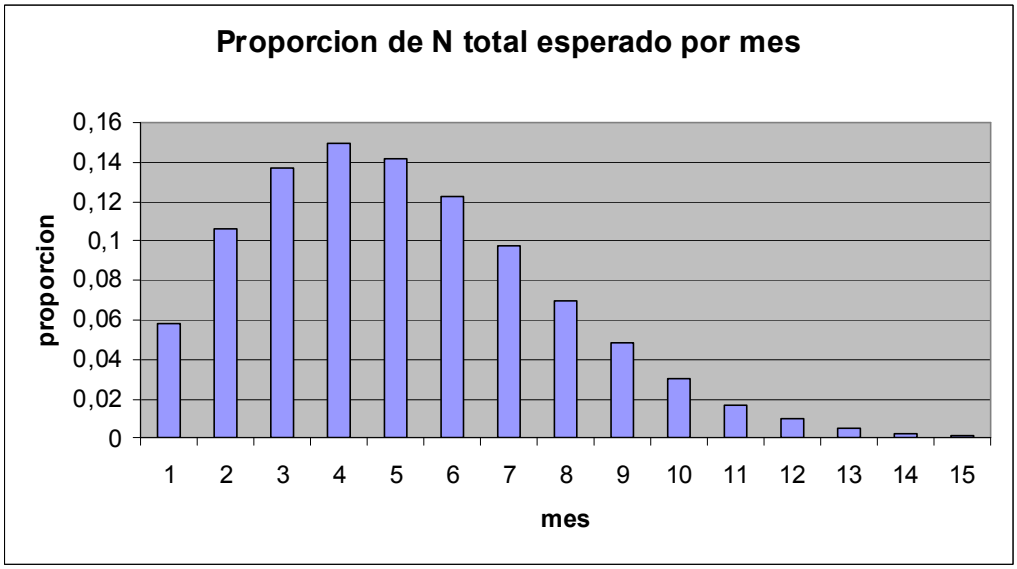


Figura 6.6: Proporción esperada del número total de defectos.

t	f(t)	F(t)	Nivel
1	0,058	0,058	
2	0,106	0,165	1
3	0,137	0,302	
4	0,149	0,451	2
5	0,142	0,592	
6	0,122	0,715	3
7	0,097	0,811	4
8	0,07	0,881	5
9	0,048	0,929	6
10	0,03	0,959	7
11	0,017	0,976	
12	0,01	0,986	
13	0,005	0,991	8
14	0,002	0,993	
15	0,001	0,994	

Figura 6.7: Escala numérica según valores.

La escala numérica, basado en la agenda del desarrollo planeado es también mostrado para comparar.

La escala 7, correspondiente al percentil 95, es, en realidad, el mes 10, la escala 8, al percentil 99, que se espera que ocurra en el mes 13 del calendario, y escala 9 al 99,99, no se espera conseguir hasta el fin del mes 15 del calendario.

Uno de los puntos fuertes de este modelo es que el numero esperado de fallos puede ser previsto desde varios puntos en el proceso de desarrollo comparado con el modelo MUSA, que proporciona la predicción solo cuando el testeo del sistema empieza (escala 4).

Otro corolario a este modelo es que el tiempo principal hasta el siguiente defecto (MTTD) es proporcionado por l/f(t). Esto es solo significativo después de la escala 4 (desde antes de este punto el sistema no ha sido desarrollado así que los defectos podrían no ser detectados). Dado que progresa el desarrollo (por ejemplo se incrementa la t), el MTTD aumenta ya que los defectos se están eliminando.

6.1.4 MODELO DE PREDICCION DEL LABORATORIO ROME [18]

6.1.4.1 MODELO DE PREDICCIÓN DEL LABORATORIO DE ROMA: RL-TR-92-52

Este es un método para predecir la densidad de las faltas en el tiempo de entrega, y, subsecuentemente utilizar esta densidad de faltas para predecir el número total de faltas inherentes, N, y la tasa de fallos.

El modelo RL-TR-92-52 contiene datos empíricos recogidos de multitud de fuentes incluyendo el Laboratorio de Ingeniería de Software. Hay un total de 33 fuentes de datos de 59 proyectos diferentes.

El aspecto fundamental de esta metodología es que la fuente de las líneas de código (SLOC) es una medida métrica valida.

Este modelo provee un mecanismo para la asignación de la fiabilidad de software como una función de las características del mismo. La terminología de este modelo es presentada en la tabla:

Términos	Descripción
A	Factor seleccionado basado en el tipo de aplicación; representa la línea base de densidad de fallos
D	Factor seleccionado que refleja el entorno de desarrollo
S	Factor calculado de varios sub-factores que reflejan las características del software
SLOC	El numero de líneas de código fuente ejecutables; las líneas que están en blanco o contienen comentarios son excluidas
FD	Densidad de fallo: para el propósito de este modelo es definida como la razón de fallos por línea de código
N	Estimación del numero total de fallos en el sistema; la predicción es derivada de la densidad de fallos y del tamaño del sistema
C	Es factor representa una conversión de la razón de fallos asociada con cada tipo de aplicación ; los valores son determinados por la división de la tasa promedio de fallos operacionales y el promedio de la densidad de fallos en la línea base del conjunto de ejemplo

Figura 6.8: Terminología del RL-TR-92-52.

Este modelo se reconoce como uno de los pocos de predicción disponible públicamente que esta basado sobre extensa información histórica. Las predicciones están basadas sobre datos recopilados y almacenados de varios tipos de sistemas de software desarrollados por las fuerzas aéreas. Lo anterior se ilustra en la tabla siguiente:

Cantidad de datos históricos incluidos

TIPO DE APLICACION	# DE SISTEMAS	Total SLOC
TRANSPORTE AEREO	7	540,617
ESTRATEGICOS	21	1,793,831
TACTICOS	5	88,252
CONTROL DE PROCESOS	2	140,090
CENTROS DE PRODUCCION	12	2,575,427
DE DESARROLLO	6	193,435
TOTAL	53	5,331,652

Figura 6.9: Cantidad de datos históricos incluidos.

El modelo consiste en 9 factores para predecir la densidad del fallo de la aplicación software.

Factor	Medida	Rango de valores	Fase usada	Intercambio rango
A Aplicación	Dificultad en el desarrollo Varios tipos de aplicación	2 a 14	A — T	No arreglado
D Desarrollo organizacional	Desarrollo organizado, métodos, herramientas, técnicas, documentación	0.5 a 2.0	Si sabe en A, D - T	El rango mas amplio
SA Administración de anomalias de software	Indicación de diseño con tolerancia a fallos	0.9 a 1.1.	Normalmente C - T	Pequeño
ST Trazabilidad de software	Trazabilidad en los diseños, codigos y requerimientos	0.9 a 1.0	Normalmente C — T	Grande
SQ Calidad de softwsare	Codificacion estandar	1.0 a 1.1	Normalmente C — T	Pequeño
SL Lenguaje de software	Normalizacion en la densidad de fallos por el tipo de lenguaje	N/A	C — T	N/A

SX Complejidad de software	Unidad de complejidad	0.8 a 1.5	C — T	Grande
SM Modularidad del software	Unidad de tamaño	0.9 a 2.0	C — T	Grande
SR Revision estandar de software	Cumplimiento con las reglas de diseño	0.75 a 1.5	C — T	Grande

Figura 6.10: Resumen del modelo RL-TR-92-52.

Este es el significado de las iniciales de la Fase:

- A - Fase de análisis.
- D - Diseño Detallado.
- C - Codificación.
- T – Pruebas.

Los beneficios de utilizar este modelo:

Puede ser utilizado en cuanto el concepto del software es conocido.

- Durante la fase de concepto, el análisis es hecho para determinar el impacto del entorno del desarrollo sobre la densidad de fallos.
- Durante la fase de diseño, el análisis es elaborado para determinar el impacto de las características del software sobre la densidad de fallos.
- Puede ser muy útil para la asignación de la fiabilidad de un sistema de software ya que puede ser aplicado únicamente a cada tipo de aplicación, comprometiendo todo el sistema de software.
- La predicción puede ser realizada utilizando valores únicos para los factores A, S y D basados sobre datos históricos de software provenientes del ambiente especifico de desarrollo en la organización.

Esto significa que a pesar de que las ecuaciones fundamentales son las siguientes:

Densidad de Fallos = FD = A x D x S (fallos/línea).
Número Estimado de Fallos Inherentes = N = FD x SLOC.
Tasa de Fallos = FD x C (fallos/tiempo).

Cambiaran sustancialmente dependiendo de la fase en la que nos encontremos dentro de la aplicación software.

Si hay políticas de desarrollo de software que están definidos en el proyecto y se sabe de manera cierta el impacto de los mismos (a pesar de que en el código todavía no existen), entonces, dichos elementos pueden ser utilizados en la fase de predicción en vez de cuando les toque.

Sin embargo, el analista debe tener la certeza de que la predicción refleja lo que realmente se esta realizando en la ingeniería de software.

Hay ciertos parámetros en este modelo de predicción que tienen capacidad de modificación. Esto significa que hay una gran diferencia entre los valores máximos y mínimos previstos para ese factor en particular.

Realizar una modificación sobre estos valores significa que el analista determina donde se pueden realizar modificaciones en el proceso de ingeniería de software o producto para experimentar predicción de la densidad del fallo mejorada.
Una modificación por tanto, será valiosa sólo si el analista tiene la capacidad de impactar en el proceso de desarrollo de software.

El análisis de las modificaciones también se puede utilizar para llevar a cabo un análisis de coste. Por ejemplo, una predicción puede llevarse a cabo utilizando un conjunto básico de parámetros de desarrollo.

A continuación, la predicción se puede realizar de nuevo utilizando un conjunto de parámetros de desarrollo agresivo. La diferencia en la densidad de fallo se puede medir para determinar el beneficio en términos de densidad de fallo que se pueden conseguir mediante la optimización del desarrollo.

Un análisis de costes también se puede realizar multiplicando la diferencia en el número total esperado de defectos por cualquiera de un parámetro de coste relativo o fijo.
La salida de este modelo es de una densidad de fallo en términos de fallos por KSLOC. Esto se puede utilizar para calcular el número total estimado de los defectos inherentes simplemente multiplicando por el número total de KSLOC.

Si se utilizan puntos de función, pueden ser convertidos a KSLOC mediante el uso de la siguiente tabla:

Programming Language	Expansion Ratio	Mean Source Statements/Function Point
Basic Assembly	1.0	320
Macro Assembly	1.5	320
C	2.5	128
Interpreted Basic	2.5	128
2nd Generation language	3.0	107
Fortran	3.0	107
ALGOL	3.0	107
COBOL	3	107
CMS2	3	107
JOVIAL	3	107
Pascal	3.5	91
3rd Generation language	4.0	80
PL/I	4.0	80
Modula 2	4.0	80
Ada 83	4.5	71
Prolog	5.0	64
Lisp	5.0	64
Forth	5.0	64
Quick Basic	5.5	58
C++	6.0	53
Ada 9X	6.5	49
Database Default	8.0	40
Visual Basic	10.0	32
APL	10.0	32
SMALLTALK	15.0	21
Generators	20.0	16
Screen Painters	20.0	16
SQL	27.0	12
Spreadsheet Default	50.0	6

Figura 6.11: Conversión puntos de función a miles de líneas de código.

La densidad de fallos también se puede convertir a ratio de fallo mediante el uso de uno de los siguientes elementos:

- Recogida de datos de prueba.
- Los datos históricos de otros proyectos de la organización, y / o
- La tabla de transformación aportada con el modelo, que se muestra en esta tabla.

Application type	Conversion from fault density to failure rate
Airborne	6.28
Strategic	1.2
Tactical	13.8
Process control	3.8
Production center	23
Developmental	132.6
Average	10.6

Figura 6.12: Tabla de transformación de densidad de fallo a ratio de fallo.

Éstos se enumeran en el orden de preferencia. Lo ideal sería que la organización determine el ratio de conversión entre la densidad de fallo y el ratio fallo. Si ese dato no esta disponible, esta técnica proporciona una tabla de ratio de conversión. Esta tabla se basa en los datos generados durante la elaboración del informe RL-TR-92-52.
La salida de la densidad de fallo predicho a partir de este modelo también se puede usar como una entrada para el modelo de predicción Musa.

Los valores de muchos de los parámetros de este modelo pueden cambiar a medida que avanza el desarrollo. Los valores más actualizados siempre se deben utilizar al hacer una predicción. Las predicciones tenderán a ser mas y mas precisas en el momento en que las métricas de cada fase sucesiva estén disponibles y los valores se actualicen para reflejar mejor las características del diseño y la implementación final. Los detalles de este modelo no se incluyen en este trabajo al desviarse del proyecto original.

PREDICCION DE LA FASE DE REQUISITOS

.Este método sólo requiere que la información sobre el tipo de aplicación y la organización del desarrollo se conozca. Cuanto menor es el valor calculado, menor será la densidad de fallo y pronosticaron tasa de fracaso.

Factor	Measure	Range of values
A - Application	Difficulty in developing various application types	2 to 14
D - Development organization*	Development organization, methods, tools, techniques, documentation	.5 to 2.0

Figura 6.13. Factores a utilizar en la fase de requisitos.
* Puede no ser conocido durante esta fase.

D = fallos esperados por KSLOC = A si D no es conocido o

FD = fallo esperados por KSLOC = A*D si D es conocido

N = numero inherente estimado de fallos = FD*KSLOC

PREDICCION EN LA FASE DE DISEÑO

Este método solo requiere que la información sobre el tipo de aplicación, la organización, del desarrollo, y los requisitos de diseño sean conocidos. Cuanto menor es el valor calculado, menor será la densidad de fallo y el ratio de fallo esperado.

Factor	Measure	Range of values
A - Application	Difficulty in developing various application types	2 to 14
D - Development organization	Development organization, methods, tools, techniques, documentation	.5 to 2.0
SA - Software anomaly management*	Indication of fault tolerant design	.9 to 1.1
ST - Software traceability*	Traceability of design and code to requirements	.9 to 1.0
SQ - Software quality*	Adherence to coding standards	1.0 to 1.1

Figura 6.14: Factores a utilizar en la fase de diseño.

A pesar de que normalmente no hay código en la fase de diseño, sin embargo, en la fase de diseño, los parámetros pueden medirse en base a las prácticas de codificación que se están utilizando.

FD = fallos esperados por lineas de código = A*D

Si hay politicas de realización de código que se este teniendo en cuenta entonces
FD = fallos esperados por KSLOC = A*D*SA*ST*SQ

N = numero estimado de fallos inherentes = FD*KSLOC

Durante las fases de diseño de los factores A y D deben ser conocidos.

D: si es necesario, convierta la densidad de fallo a ratio de fallo utilizando un factor de conversión.

PREDICCION EN LA FASE DE CODIFICACIÓN, UNITARIAS E INTEGRACION

Este método requiere que la información sobre el tipo de aplicación, la organización deL desarrollo, los requisitos de diseño y las prácticas de codificación deben ser conocidos..

Cuanto menor es el valor calculado, menor será la densidad de fallo y el ratio de fallo esperado.

Factor	Measure	Range of values
A - Application	Difficulty in developing various application types	2 to 14
D - Development organization	Development organization, methods, tools, techniques, documentation	.5 to 2.0
SA - Software anomaly management	Indication of fault tolerant design	.9 to 1.1
ST - Software traceability	Traceability of design and code to requirements	.9 to 1.0
SQ - Software quality	Adherence to coding standards	1.0 to 1.1
SL - Software language	Normalizes fault density by language type	n/a
SX - Software complexity	Unit complexity	.8 to 1.5
SM - Software modularity	Unit size	.9 to 2.0
SR - Software standards review	Compliance with design rules	.75 to 1.5

Figura 6.15: Factores a utilizar en las fases de codificación, unitarias e integración.

FD = fallos esperados por líneas de código = A*D*SA*ST*SQ*SL*SX*SM*SR

N = numero estimado de fallos inherentes = FD*KSLOC

E: si es necesario, convertir la densidad de fallo en ratio de fallo usando una técnica de conversión.

F: realiza intercambios con los factores D y S aplicables para determinar qué técnicas serían necesarias para alcanzar una densidad de fallo objetiva o un ratio de fallo.

6.1.4.2 MODELO DE PREDICCIÓN DEL LABORATORIO DE ROMA: RL-TR-92-15

El uso de KSLOC como tipo métrico se vuelve más y más irrelevante con los cambios recientes en la tecnología de desarrollo de software, así como el desarrollo de un sistema GUI (Graphical User Interface), y el uso de software fuera de la plataforma.

Esta técnica de informes, producido por Hughes Aircraft para Laboratorio Rome, es examinado en muchos sistemas software. El resultado en una predicción de media de ratio de fallo de 6/1000 SLOC. (este fue el valor utilizado por defecto en la tasa de fallos, utilizado en el modelo de tiempo de ejecución de Musa).

Para el informe concreto de Hughes Aircraft se definieron un conjunto de 24 factores predictores que se listan en la tabla siguiente, y se utilizan para estimar las tres principales variables de interés, que son:

- Numero de fallos detectados durante cada fase de desarrollo (DP).

- Horas-Hombre utilizados para cada fase (UT).
- Tamaño del producto (S).

El resultado de las ecuaciones fue:

$$(1) \quad f(DP) = 18.04 + .05 \times (.009\,X_1 + .99\,X_2 + .10\,X_3 - .0001\,X_4 + .0005\,X_5)$$

$$(2) \quad f(UT) = 17.90 + .04 \times (.007\,X_1 + .796\,X_2 + .08\,X_3 - .0003\,X_4 + .0003\,X_5 + .00009\,X_6 + .0043\,X_7 + .013\,X_8 + .6\,X_9 + .003\,X_{10})$$

$$(3) \quad f(S) = 17.88 + .04 \times (.0007\,X_1 + .8\,X_3 + .01\,X_8 + .6\,X_9 + .008\,X_{23} + .03\,X_{25})$$

Donde los coeficientes y descripción de las variables del modelo de regresión están listados en la tabla.

X	DESCRIPCION DE VARIABLES	COEFICIENTES		
		EQ 1	EQ 2	EQ 3
1	NUMERO DE FALLAS EN LA ESPECIFICACION DE REQUERIMIENTOS	.009	.007	.007
2	REQUERIMIENTOS EVIDENCIADOS EN LA ESPECIFICACIÓN	.99	.796	NA
3	PAGINAS EN ESPECIFICACION	.10	.08	.80
4	MESES-HOMBRE GASTADOS EN EL ANALISIS DE REQUERIMIENTOS	.0001	-.0003	NA
5	REQUERIMIENTOS CAMBIADOS DE LA BASE INICIAL	.0005	.0003	NA
6	FALLAS EN EL ANALISIS DEL DOCUMENTO PRELIMINAR	NA	.00009	NA
7	NUMERO DE CSCS	NA	.0043	NA
8	NUMERO DE UNIDADES EN DISEÑO	NA	.013	.01
9	PAGINAS EN EL DOCUMENTO DE DISEÑO	NA	.6	.6
10	MESES-HOMBRE GASTADOS EN EL DISEÑO PRELIMINAR	NA	.003	NA
11	NUMERO DE FALLAS EN EL DOCUMENTO DE DISEÑO	NA	NA	NA
12	MESES HOMBRE GASTADOS EN EL DETALLE DEL DISEÑO	NA	NA	NA
13	FALLAS DE DISEÑO IDENTIFICADAS EN LAS LINEA BASE	NA	NA	NA
14	FALLAS DE DISEÑO DESPUES DE LA REVISION . INTERNA	NA	NA	NA
15	NUMERO DE EJECUTABLES	NA	NA	NA
16	FALLAS ENCONTRADAS EN LA REVISION DE CODIGO	NA	NA	NA
17	PROMEDIO EN AÑOS DE LA EXPERIENCIA DEL PROGRAMADOR	NA	NA	NA
18	NUMERO DE UNIDADES BAJO REVISION	NA	NA	NA
19	NUMERO PROMEDIO DE SLOC POR UNIDAD	NA	NA	NA
20	NUMERO DE SECCIONES POR UNIDAD	NA	NA	NA
21	NUMERO DE SECCIONES CUBIERTAS	NA	NA	NA
22	COVARIANZA	NA	NA	NA
23	NUMERO DE VECES QUE UNA UNIDAD ES PROBADA	NA	NA	.008
24	MESES-HOMBRE PARA CODIFICAR UNA UNIDAD PROBADA	NA	NA	NA
25	TOTAL (X13 + X14 + X16)	NA	NA	.03

Figura 6.16: Coeficientes de la ecuación de regresión

Los resultados indican que 13 de los 24 factores listados no tienen efectos sobre las tres variables de interés. En desarrollos más a futuro, importantes y destacados estimadores complicaron la especificación de los requerimientos de software, al incluir la declaración del número de requerimientos, número de fallos en esas declaraciones y el número total de páginas en la especificación.

Los beneficios de este modelo son:

- Puede ser usado antes para testear el sistema para estimar la fiabilidad.
- Incluye parámetros de coste y producto así como las fallos y el tiempo.

Las desventajas de este modelo son:

- Estaba basado en una colección de datos para una organización en un tipo de industria con unas aplicaciones específicas.
- No revela la unidad de medida para el tamaño de la especificación.

6.2 MODELOS DE ESTIMACIÓN

La cuenta de fallos y los modelos de ratio de fallos son los tipos más comunes de técnicas de estimación. Cada uno realiza una presunción sobre como pueden llegar los fallos (detectadas) y como pueden corregirse.

Los modelos de cuenta de fallos que vamos a ver son:

- Exponencial.
- Técnicas Weibull y Bayesianas.
- También están incluidos en los escenarios de modelos de estimación los test de cobertura y los métodos de marcado de fallos.

6.2.1 MODELOS DE DISTRIBUCIÓN EXPONENCIAL

En general, los modelos exponenciales asumen que el software esta en un estado operacional y que todos los fallos son independientes de otros. El tiempo de fallo, t, de un fallo individual sigue una distribución exponencial.

Para el caso de que λ (t) sea constante nos encontramos ante una distribución de fallos de tipo exponencial y la fiabilidad tendrá la expresión:

R (t) = exp (-λt) para t>=0

Matemáticamente podremos escribir la función exponencial de densidad de probabilidad de fallo:

f(t)=λexp(-λt) cuando t>=0

y f(t) = 0 cuando t>0

El tiempo medio hasta un fallo (MTTF→ mean time to next failure) expresado como:

MTTF = 1/λ

El Tiempo medio entre fallos (MTBF--> mean time beetwen failure) esta expresado como:

Se demuestra que para la distribución exponencial el MTBF es igual a la inversa de la tasa de fallos y por lo tanto igual al MTTF o sea:

MTBF = m = 1/λ = MTTF

En las expresiones anteriores la constante λ (tasa de fallos) tiene las dimensiones de (tiempo)^-1, exp es la base de los logaritmos neperianos o naturales (2,71828...) t es el tiempo de funcionamiento para el que se desea conocer la fiabilidad.

La tasa de fallos se expresa, según se ha visto, en unidades inversas a las que expresan el tiempo t, generalmente en horas. La fiabilidad R (t) representa en éste caso la probabilidad de que el programa, caracterizado por una tasa de fallos constante, no se averíe durante el tiempo de funcionamiento t.

Las notaciones usadas en el caso general del modelo de distribución exponencial son mostradas como sigue en la tabla:

Notation	Explanation
N	NUMERO TOTAL DE DEFECTOS
n	NUMERO DE DEFECTOS A LA FECHA
c	NUMERO DE DEFECTOS CORREGIDOS A LA FECHA
N-n	DEFECTOS AUN NO MANIFESTADOS
N-c	DEFECTOS PARA SER CORREGIDOS
n_f	CONTEO DE FALLAS
λ_f	TASA DE FALLAS
t_f	TIEMPO FUTURO
n_p	CONTEO DE FALLAS EN EL TIEMPO PRESENTE
λ_p	TASA DE FALLAS EN EL TIEMPO PRESENTE
t_p	TIEMPO PRESENTE

Figura 6.17: Tabla de notaciones del modelo exponencial.

Así mismo, la representación gráfica del modelo exponencial es el siguiente:

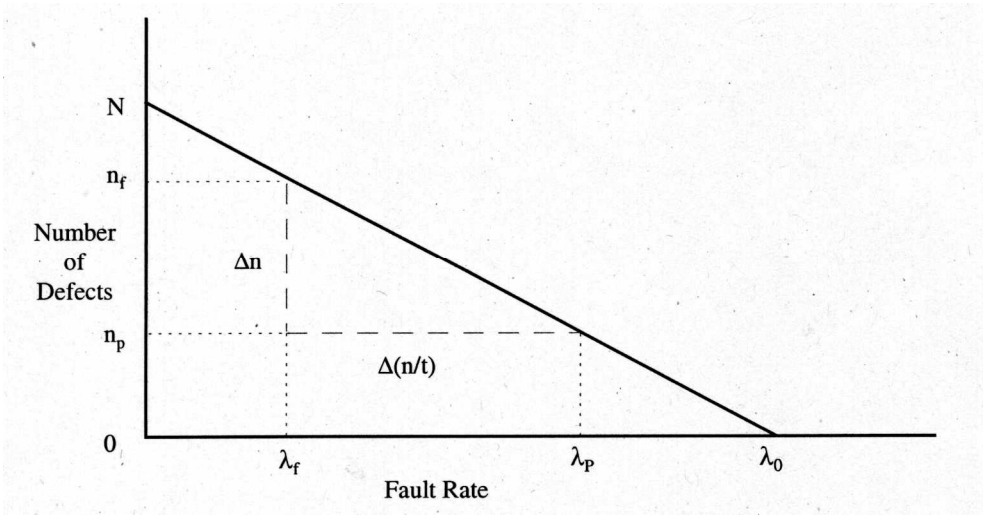


Figura 6.18: Gráfica de modelo exponencial

Los defensores del modelo exponencial denotan su simplicidad y su paralelismo con el marco de fiabilidad del hardware. La mayor desventaja es que no puede ser usado tan pronto como este en desarrollo, sino que el producto debe ser operacional antes de que

el modelo sea usado. Por lo tanto, no puede ser usado para la primera evaluación de fiabilidad.

La siguiente tabla resume varios modelos exponenciales incluyendo suposiciones y comentarios:

Modelo	MTTF	Dn	Dt	COMENTARIOS
Exponencial general	$1/[k(N - c)]$	$k^{-1} \lambda_p / \lambda_f$	$k^{-1} ln (\lambda_p / \lambda_f)$	Las fallas son iguales en severidad y en probabilidad de deteccion. La tasa de fallas son directamente proporcionales al numero de de fallas encontradas
Lloyd-Lipow	$1/[k(N - n)]$	$k^{-1} \lambda_p / \lambda_f$	$k^{-1} ln (\lambda_p / \lambda_f)$	La tasa de fallas es directamente proporcional al numero de fallas encontradas
Basica de Musa		$N/\lambda_0 (\lambda_p - \lambda_f)$	$N/\lambda_0 ln (\lambda_p - \lambda_f)$	Referencia a una tasa de fallas inicial en el tiempo "0" (inicio de la prueba)
Logaritmica de Musa		$\Gamma^1 ln (\lambda_p / \lambda_f)$	$\Gamma^1 (1/\lambda_f - 1/\lambda_p)$	Algunas fallas son encontradas mas rapidamente que otras. la tasa de detección de fallas decrese exponencialmente
Shooman	$1/[kSLOC - ((N/SLOC) - (C/SLOC))]$	$k^{-1} \lambda_p / \lambda_f$	$k^{-1} ln (\lambda_p / \lambda_f)$	ajustes por cambios en el tamaño del producto; Cada parametro es normalizado por lineas de codigo
Goel-Okumoto				Las fallas pueden ocasionar otras fallas. Las faltas tienen que ser removidas

Cuando "K" es constante de proporcionalidad, "N" es el numero de ocurrencia de fallas, "c" es el numero fallas corregidas y "n " es el numero de fallas detectadas

Figura 6.19: Resumen modelos exponenciales.

Claves

- K es una constante de proporcionalidad.
- N es el número inherente de fallos.
- c es el número de fallos corregidas.
- n es el número de fallos detectadas.

6.2.1.1 MODELO EXPONENCIAL GENERAL

En el caso general el modelo asume que todos los fallos son iguales en severidad y probabilidad de detección y cada una de ellas es inmediatamente corregida una vez detectada. El ratio de fallos λ, se asume que esta directamente relacionado con el numero de fallos restantes en el software. Esto es, λ es una función del número de fallos corregidas, c, como se muestra a continuación:

$\lambda = k(N-c)$

Donde K es una constante de proporcionalidad. En actualidad, K esta normalmente estimado del alcance de los argumentos observados de la tasa de fallos contra el numero de fallos corregidas.

La proyección del número de fallos necesitados para ser detectadas para alcanzar el ratio de fallo final λf esta dado por:

$\Delta n = (1/k)\ \lambda p/\lambda f$

Donde K es la misma constante de proporcionalidad usada anteriormente.

La proyección del tiempo necesario para alcanzar un ratio de fallo esta dado por:

$\Delta t = (1/K)\ln[\lambda p/\lambda f]$

La mayor desventaja de esta aproximación esta en que no solo los defectos deben ser detectados, sino que también deben ser corregidos.

6.2.1.2 LLOYD-LIPOW MODEL

El modelo Exponencial de Lloyd-Lipow también asume que todos los fallos son iguales en severidad y probabilidad de detección. La diferencia con el modelo previo es que en la aproximación de Lloyd-Lipow, el ratio de fallos λ esta relacionado directamente con el numero de fallos pendientes de ser detectadas (no corregidas) en el software. Esto es, λ es una función del número de fallos detectadas n:

$\lambda = K(N-n)$

La expresión para el MTTF, Δn e Δt son las mismas que en el modelo exponencial general.

Esta fórmula del modelo exponencial no requiere corrección de defectos, solo detección. Sin embargo, la validez del uso del modelo exponencial en esta situación ha sido cuestionada por personas que trabajan con estos modelos.

6.2.1.3 MODELO BÁSICO DE MUSA

El modelo básico de Musa es otra fórmula del modelo exponencial general.

Utiliza el ratio de fallo inicial (por ejemplo al comienzo del testeo de software), $\lambda 0$, donde sea λo estimado desde el dato o calculado ($\lambda 0 = N/k$) basado en la suposición para N y la estimación de k. Hay que tener en cuenta que el alcance de los valores debe ser previamente referenciado.

En este modelo, el ratio de fallo después de n fallos ha sido detectado en una fracción del ratio de fallo original de la siguiente forma:

$\lambda n = \lambda 0(1 - n/v)$

Donde n es normalmente expresado como μ y v es normalmente representado como υ
Mientras la expresión para el ratio de fallo en el tiempo t esta dado por:

$$\lambda_t = \lambda_0 \exp[-(\lambda_0/\upsilon)\tau]$$

Donde:

υ = N/B donde N es el numero de fallos inherentes y B es la proporción de reducción de fallos, usualmente asumida a ser de 95% (95 % de las fallos que no se detectan en la entrega del producto se convierten en fallos después).

τ = Tiempo de test del sistema

La proyección del numero de fallos que necesitan ser detectados para alcanzar una ratio de fallo final λf esta dado por:

$\Delta n = N/\lambda 0(\lambda p - \lambda f)$

La proyección del tiempo necesario para alcanzar un ratio de fallo proyectado esta dado por:

$\Delta t = N/\lambda 0 \ln(\lambda p - \lambda f)$

La desventaja de este modelo es que es muy sensible a desviaciones de las suposiciones. Por añadidura, como se anoto en los trabajos anteriores del modelo de Musa, las unidades son tiempo de ejecución, y no tiempo de calendario.

6.2.1.4 MODELO LOGARÍTMICO DE MUSA

El modelo Logarítmico de Musa tiene diferentes hipótesis que los otros modelos exponenciales:

- Algunas fallos son mas fáciles de encontrar antes que otras.
- El ratio de detección de fallo no es constante, pero decrece exponencialmente.

En este modelo, la tasa de fallos después de n fallos ha sido detectada mediante una función de la tasa de fallo original, mediante la siguiente fórmula:

$$\lambda_n = \lambda_0 \exp(-ft)$$

Mientras la expresión para la tasa de fallos en función del tiempo es dada por:

$$\lambda_t = \lambda_0 / (\lambda_0 \, f \, t + 1)$$

Donde:

f = parámetro de reducción de la intensidad del fallo, es decir, el cambio relativo de n/t sobre n.

La proyección del numero de fallos que necesitan ser detectados para alcanzar un ratio de fallo final λf esta dado por:

Δn = 1/f ln (λp/ λf)

La proyección del tiempo necesario para alcanzar un ratio de fallo proyectado esta dado por:

Δt = 1/f (1/ λf – 1/ λp)

El mayor beneficio de este modelo es que no requiere una estimación para N. Dado que el valor para f puede ser estimado antes que la ocurrencia de datos reales, el modelo puede ser usado antes en el ciclo de desarrollo para estimar la fiabilidad.

La desventaja de este modelo, típica de los modelos mas exponenciales, es que las hipótesis del modelo deben ser validas para que los resultados lo sean. En particular, la hipótesis de que el ratio de fallo de detección de fallos decrece exponencialmente no ha sido confirmado con mucho conjuntos de datos reales.

Por añadidura, como se anoto en los trabajo previos con el modelo de Musa, las unidades son tiempo de ejecución, no tiempo de calendario, haciendo comparación directa con la dificultad de fiabilidad del hardware.

6.2.1.5 MODELO DE SHOOMAN

El modelo de Shooman es similar al modelo exponencial general, excepto que cada fallo contabilizada es normalizada para las líneas de código en ese punto del tiempo.

Antes, λ = k(N – c); aquí esta dado por:

λ = k SLOC (N/SLOC – c/SLOC)

La ecuación de MTTF = 1/ λ que usa la expresión de Shooman para λ. Las ecuaciones para Δn e Δt son las mismas que en el modelo exponencial general.

La ventaja del modelo de Shooman es que se ajusta a cambios del tamaño del producto software. La desventaja es que podría ser utilizado en desarrollos donde los supuestos pueden no aplicar a las especificaciones que se tienen, provocando resultados erróneos.

6.2.1.6 MODELO GOEL-OKUMOTO

Este modelo es diferente de otros modelos exponenciales porque asume que los fallos pueden causar otros fallos y que pueden no ser eliminadas inmediatamente. Se requiere una solución iterativa.

Este modelo esta expresado como:

λ t = ab exp(-bt)

Donde a y b son resueltos iterativamente desde la siguiente fórmula:

n/a = 1 – exp(-bt) y n/b = a t exp(-bt) + Σ tj,

donde el sumatorio es sobre i = 1,…..,n

Se usa N y K como puntos de comienzo para resolver estas dos ecuaciones simultáneamente.

El mayor beneficio de este modelo es que puede ser usado antes que otros modelos exponenciales mientras que su mayor desventaja es que es muy sensible a desviaciones de las hipótesis.

6.2.2. MODELO DE DISTRIBUCIÓN DE WEIBULL

El modelo Weibull es uno de los primeros modelos aplicados al software. Tiene la misma fórmula que se usa para la fiabilidad del hardware. Hay dos parámetros:

Es de los modelos de distribución mas frecuentemente usados en fiabilidad. Es normalmente usada para calcular el MTTF (Mean Time To Fail) y el MTTR (Mean Time To Repair).

Su función de densidad de probabilidad viene dada por:

$$f(x) = \left(\frac{\beta}{\theta} \left(\frac{x-\delta}{\theta} \right)^{(\beta-1)} \right) \left(e^{-\left[\left(\frac{x-\delta}{\theta} \right) \right]^{\beta}} \right), x \geq \delta$$

Para tratar problemas de tiempos de vida, la fórmula vendrá dada por:

$$f(t) = \lambda\beta(\lambda t)^{\beta-1} \exp\{-(\lambda t)^{\beta}\}$$

Donde

- λ es el parámetro escala (λ >0) y
- beta β es el parámetro forma que refleja el incremento (b>1), decrecimiento (b<1) o constante (b=1) del ratio de fallo.
- d parámetro de ubicación.

El parámetro forma es lo que le da a la distribución de Weibull su flexibilidad, ya que, cambiando el valor del parámetro forma, la distribución de Weibull puede modelar una amplia variedad de datos.

Si β es igual a 1 la distribución de Weibull se aproxima a una distribución exponencial, si es igual a 2, es idéntica a una distribución de Rayleigh; si β esta entre 3 y 4 la distribución de Weibull se aproxima a una distribución normal.

La distribución de Weibull se aproxima a una distribución log normal para varios valores de b. Para la mayoría de los estudios se requieren más de 50 muestras para diferenciar entre la distribución de Weibull y la log normal.

En la siguiente imagen se muestra la flexibilidad de la distribución de Weibull con λ =100 y diferentes valores de β:

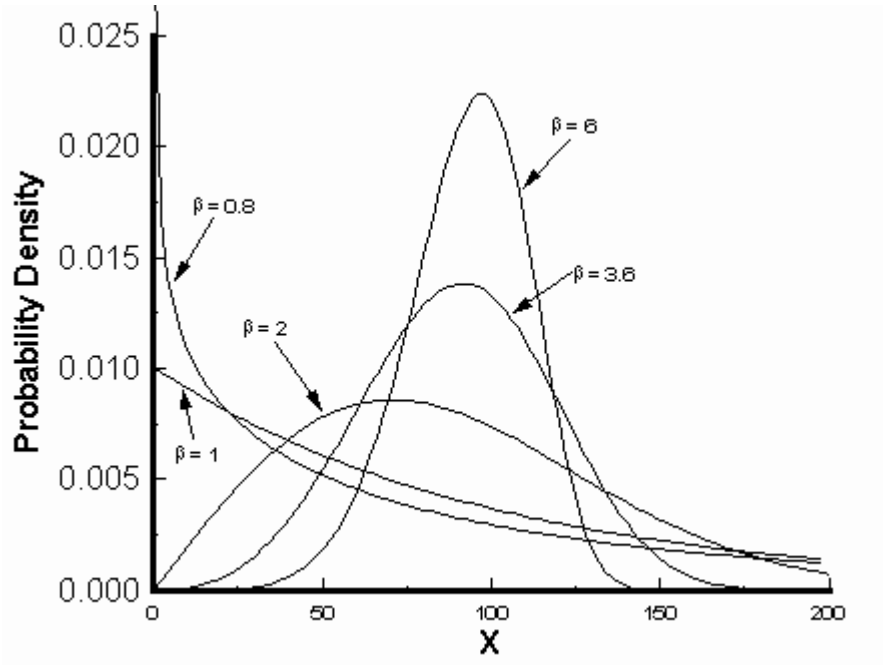


Figura 6.20 Gráfica de distribución de Weibull

El parámetro escala determina el rango de la distribución y también se conoce como vida característica si el parámetro de localización es igual a cero.

Si d no es igual a cero, la vida característica es igual a $\lambda + d$

El valor $1/\lambda$ es aproximadamente el percentil 63,2% y se interpreta como el valor de la variable del tiempo de vida en el que han fallado el 63,2% de las unidades, independientemente del valor del parámetro de forma.

El efecto del parámetro de escala de la función de densidad de probabilidad se muestra en la siguiente figura.

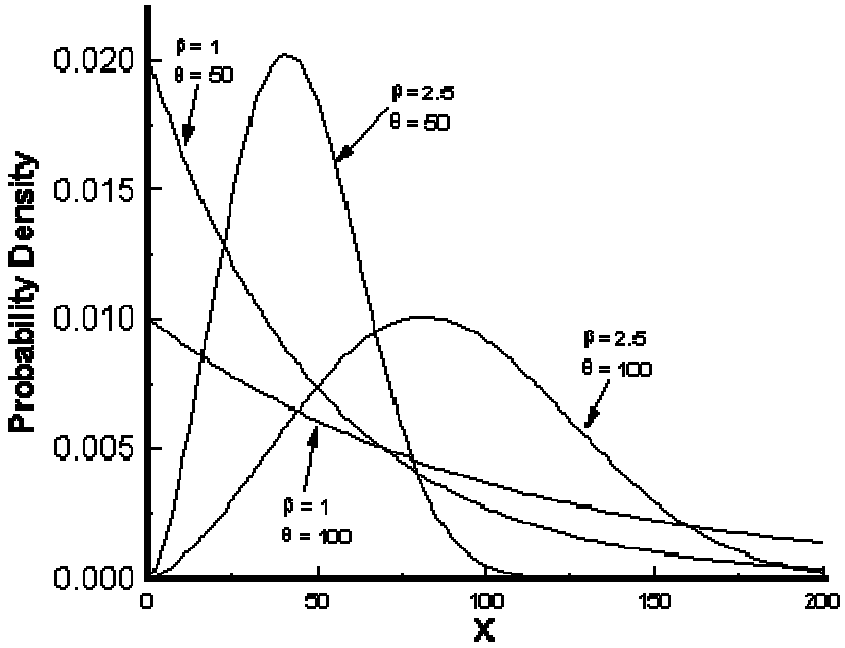


Figura 6.21: Gráfica de función de densidad.

El parámetro de localización se utiliza para definir una zona libre de fallos. La probabilidad de fallo cuando x es menor que d es cero. Cuando d> 0, hay un período en el que no se puedan producir fallos.

Cuando d <0, los fallos se han producido antes de que el tiempo sea igual a 0.

Al principio esto parece ridículo, pero un parámetro de localización negativo es causado por el envío de datos erróneos, fallos en ejecución y los fallos de la vida útil. En general, se supone que el parámetro de ubicación es cero. El efecto del parámetro de ubicación se muestra en la figura siguiente.

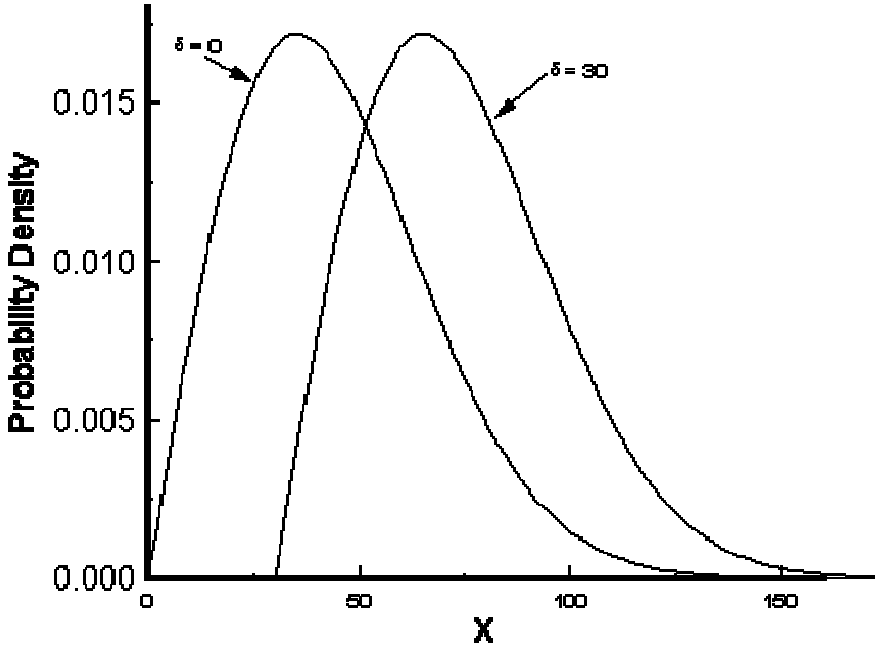


Figura 6.22: Efecto del parámetro de ubicación.

La función de riesgo de Weibull se determina por el valor del parámetro de forma.

$$h(x) = \frac{\beta}{\theta}\left(\frac{x-\delta}{\theta}\right)^{(\beta-1)}$$

- Si β < 1 la función de riesgo es decreciente, es decir, la tasa de fallo disminuye al aumentar el tiempo esto se conoce como el periodo de la mortalidad infantil.
- Si β = 1 la función de riesgo es constante, por lo que no depende del tiempo. En este caso, la distribución Weibull coincide con la Exponencial.
- Si β > 1 la función de riesgo es creciente, lo que se conoce como el periodo de desgaste natural. En particular, si 1 < β < 2 la función de riesgo crece rápido en el origen y muy poco a medida que t crece.
- Para β = 2 el riesgo crece linealmente con el tiempo y para β > 2 crece un poco con t próximo a cero y después rápido.
- Es oportuno considerar la posibilidad β = 3, ya que en este caso, la distribución Weibull se parece a la Normal.

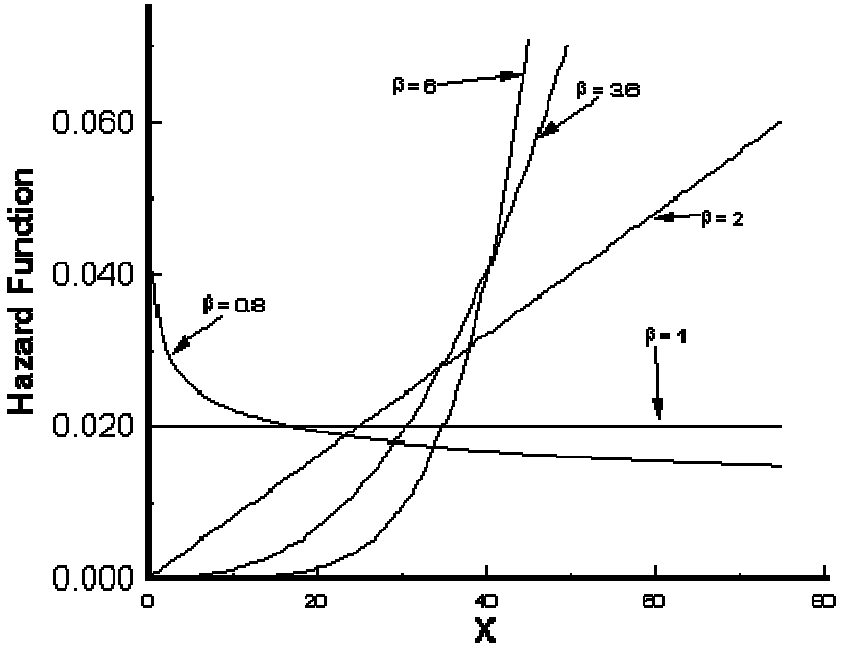


Figura 6.23: Función de riesgo dependiendo de β.

La función de fiabilidad y de distribución acumulada son:

$$R(x) = e^{-\left[\left(\frac{x-\delta}{\theta}\right)\right]^{\beta}}$$

$$F(x) = 1 - e^{-\left[\left(\frac{x-\delta}{\theta}\right)\right]^{\beta}}$$

El MTTF esta dado por:

MTTF = (b/a) Γ (1/a)

Donde Γ es la función Gamma completa = $\int_0^{\infty} y^{c-1} e^{-y} dy$

La fiabilidad en el tiempo esta dada por:

R(t) = exp [-(t/λ)^ β]

Los beneficios del modelo de Weibull es la flexibilidad para tener en cuenta incrementos y decrementos de los ratios de fallo. La desventaja de este modelo es que se requiere más trabajo en estimar los parámetros sobre el modelo exponencial.

6.2.3 MODELO DE ESTIMACIÓN DEL RATIO DE FALLO BAYESIANO

Tiene mucho sentido práctico tener toda la información disponible, antigua o nueva, objetiva o subjetiva, al tomar decisiones bajo incertidumbre. Esto es especialmente cierto cuando las consecuencias de las decisiones pueden tener un impacto significativo, financiero o de otro tipo. La mayoría de nosotros hace decisiones personales todos los días de esta manera, mediante un proceso intuitivo basado en nuestra experiencia y juicios subjetivos.

Los análisis estadístico, sin embargo, busca objetividad restringiendo generalmente la información utilizada en un análisis al que se obtiene de un conjunto actual de datos claramente relevantes.

El conocimiento previo no se utiliza excepto para sugerir la elección de un modelo particular de muestra para "adaptarse" a los datos, y esta elección se comprueba más adelante con los datos razonables.
Modelos de ciclo de vida o de reparación, como vimos antes, cuando buscamos modelos de fiabilidad para sistemas reparables y no reparables fiabilidad, tienen uno o más parámetros desconocidos.

El enfoque estadístico clásico considera estos parámetros fijos pero también como constantes desconocidas para ser estimadas (es decir, utilizando el "supongamos que...") utilizando datos tomados al azar de los sistemas de interés de la muestra. En sentido estricto, no se pueden hacer afirmaciones de probabilidad sobre el parámetro verdadero ya que es fijo, no al azar.

El enfoque bayesiano, por el contrario, trata estos parámetros de la muestra como al azar, no fija los valores. Antes de mirar los datos actuales, utilizamos información antigua o juicios incluso subjetivos, para construir un modelo de distribución previa de estos parámetros.

Este modelo expresa nuestra evaluación inicial sobre como son determinados valores probabilidades son varios valores de los parámetros desconocidos. Entonces hacemos uso de los datos actuales (a través de la fórmula de Bayes) para revisar esta evaluación derivando a lo que llamaremos el modelo de distribución.

Los parámetros estimados, junto con intervalos de confianza (conocidos como intervalos de credibilidad), se calculan directamente de la distribución posterior. Intervalos de credibilidad son declaraciones de probabilidad legítimos sobre los parámetros desconocidos, ya que estos parámetros ahora considerados al azar, no fijos.

Es poco probable que en la mayoría de las aplicaciones, los datos que se van a elegir para validar, sigan un modelo de distribución previo elegido. Los modelos paramétricos bayesianos previos son elegidos debido a su flexibilidad y conveniencia matemática. En particular, los previos conjugados (definidos a continuación) son una opción natural y popular de modelos de distribución previa bayesianos.

6.2.3.1 FÓRMULA DE BAYES, MODELOS DE DISTRIBUCIÓN PREVIOS Y POSTERIORES Y PREVIOS CONJUGADOS

La Fórmula de Bayes es una ecuación muy útil de teoría de la probabilidad que expresa la probabilidad condicional de que ocurra un evento A, dado que se ha producido el evento B (escrito P (A|B)), en términos de probabilidades no condicionales y la probabilidad que el evento B ha ocurrido, dado que A ha ocurrido. En otras palabras, la fórmula de Bayes calcula cual de los eventos es el condicionado. La fórmula es:

$$P(A \mid B) = \frac{P(A,B)}{P(B)} = \frac{P(A) \bullet P(B \mid A)}{P(B)}$$

y P(B) en el denominador es conocido mucho como el llamado "Ley de Probabilidad Total" se escribe:

$$P(B) = \sum_{i=1}^{n} P(B \mid A_i) P(A_i)$$

con los acontecimientos Ai siendo mutuamente excluyentes y agotando todas las posibilidades, incluyendo el evento A como uno de Ai..

La misma fórmula, escrita en términos de modelos de la función de densidad de probabilidad, toma la forma:

$$g(\lambda \mid x) = \frac{f(x \mid \lambda) g(\lambda)}{\int_{0}^{\infty} f(x \mid \lambda) g(\lambda) d\lambda}$$

donde f(x|λ) es el modelo de probabilidad, o función de verosimilitud, para los datos observados x dado el parámetro desconocido (o parámetros) λ, g (λ) es el modelo de distribución previa para λ y g(λ|x) es el modelo de distribución posterior para λ una vez se han observado los datos x.

Cuando g(λ|x) y g(λ) ambos pertenecen a la misma familia de distribución, g(λ) y f(x|λ) se llaman conjugadas distribuciones y g(λ) es el conjugado previo para f(x|λ).

Por ejemplo, el modelo de distribución Beta es un conjugado previo para la proporción de éxitos p cuando las muestras tienen una distribución binomial. Y el modelo Gamma (ver

Anexo 1) es un conjugado antes para la tasa de fallo λ cuando se toma como muestra los tiempos de fallo o los tiempos de reparación desde una distribución exponencial.

Este último par conjugado (gamma, exponencial) se utiliza extensivamente en aplicaciones de fiabilidad del sistema bayesiano.

Vamos a comparar el paradigma clásico y el paradigma Bayesiano cuando la confiabilidad del sistema sigue el modelo exponencial (es decir, la parte plana de la Curva de la bañera).

PARADIGMA CLÁSICO PARA LA EVALUACIÓN DE LA FIABILIDAD

- El MTBF es un valor fijo de desconocido - no hay ninguna "probabilidad" asociada.
- Datos de fallo de un período de prueba o de observación permiten realizar inferencias sobre el valor del verdadero MTBF desconocido.
- No hay otros datos que se utilizan y no hay "juicio" - el procedimiento es objetivo y basados únicamente en los datos de prueba y el supuesto modelo exponencial.

PARADIGMA BAYESIANO PARA LA EVALUACIÓN DE LA FIABILIDAD

- El MTBF es una cantidad aleatoria con una distribución de probabilidad.
- La pieza particular de equipo o sistema que está probando "elige" un MTBF de esta distribución y observas datos de falla que siguen un modelo HPP con eso MTBF.
- Antes de ejecutar la prueba, ya tienes una idea de lo que parece la distribución de probabilidad de tiempo medio entre fallos basados en datos de prueba previo o un consenso de juicio de Ingeniería.

VENTAJAS Y DESVENTAJAS DEL USO DE LA METODOLOGÍA DE BAYES

Mientras que la principal motivación para utilizar métodos de confiabilidad bayesianos es típicamente un deseo para ahorrar en costos de tiempo y materiales de prueba, existen otros factores que deben tenerse en cuenta. La siguiente tabla resume algunos pros y contras del modelo.

Ventajas:

- Utiliza información previa - esto "tiene sentido".
- Si la información anterior es buena, serán necesarias menos pruebas nuevas para confirmar un MTBF deseado en un grado de confianza.

- Intervalos de confianza son realmente intervalos para el MTBF (al azar) - a veces llamado "intervalos de credibilidad".

Desventajas

- La información previa puede no ser precisa - generando conclusiones engañosas.
- Manera de introducir información previa (opción de antes) no puede ser correcto.
- Los clientes pueden no aceptar la validez de datos previas o juicios de ingeniería.
- No hay "forma correcta" de introducir información previa y enfoques diferentes pueden dar resultados diferentes.
- Resultados no son objetivos y no se mantienen por sí mismos.

Paradigma Bayesiano: Ventajas y desventajas

Aquí consideramos sólo los modelos y supuestos que son comunes al aplicar la metodología bayesiana para evaluar la confiabilidad del sistema.

SUPUESTOS:

1. Los tiempos de fallo para un sistema bajo estudio se pueden modelar adecuadamente por la distribución exponencial. Para sistemas reparables, esto significa el modelo exponencial se aplica y el sistema está funcionando en la parte plana de la curva de la bañera. Mientras que la metodología bayesiana puede aplicarse también a los componentes no reparables.

2. El tiempo medio entre fallos del sistema pueden considerarse como un modelo de distribución previa que es una representación analítica de nuestra información previa o juicios sobre la confiabilidad de sistema. La forma de este modelo previo es la distribución gamma (el conjugado previo para el modelo exponencial). El modelo previo se define realmente para $\lambda = 1/MTBF$, puesto que es más fácil hacer los cálculos de esta manera.

3. Nuestro conocimiento previo se usa para elegir los parámetros a y b de Gamma para el modelo de distribución previa para λ. Hay muchas maneras posibles de convertir el "conocimiento" a los parámetros de la gamma, dependiendo de la forma del "conocimiento". En este punto hay tres enfoques.

    a. Si se tienen datos reales de pruebas anteriores hechas en el sistema (o un sistema que cree que tiene la misma fiabilidad que el objeto de investigación), esto es el conocimiento previo más creíble y más fácil de usar. Simplemente se define el parámetro a de gamma igual al número total de errores de todos los datos previos y el parámetro b igual al total de las últimas horas de prueba.

b. Un método de consenso para determinar a y b que funciona bien es el siguiente: montar un grupo de ingenieros que conocen el sistema y sus subcomponentes bien desde un punto de vista de la fiabilidad.

   Con el grupo llegar a un acuerdo sobre un MTBF razonable que esperan que tenga el sistema. Cada uno podría elegir un número por el que estarían dispuestos a apostar dinero en el que el sistema podría comportarse bien o mal. La media de estos números sería el 50% de su mejor suposición. O podría discutir solo los mejores candidatos al MTBF hasta llegar a un consenso.

   Repetir el proceso, esta vez llegar a un acuerdo sobre un MTBF bajo que esperan que supere el sistema. Un valor de "5 %" que significa un "95% de confianza" que el sistema superará (es decir, daría probabilidades de 19-1) es una buena opción. O puede elegir un valor de "10 %" (es decir, le daría probabilidades de 9 a 1 que el MTBF real excede el MTBF bajo).

   Se le llama al valor MTBF razonable MTBF50 y MTBF05 al MTBF bajo en el que se esta al 95% seguro. Estos dos números únicamente determinan gamma parámetros a y b que tienen λ valores de percentil en la ubicación correcta.

   $$\lambda_{50} = 1/MTBF_{50} \text{ and } \lambda_{95} = 1/MTBF_{05}$$

   Llamamos este método para especificar parámetros gamma previos el método 50/95 (o el método de 50/90 si usamos MTBF10 por ejemplo). Una forma sencilla para calcular a y b para este método se describe a continuación.

c. Una tercera manera de elegir parámetros previos comienza del mismo modo como el segundo método. Se realiza un consenso de un razonable MTBF, MTBF50. A continuación, sin embargo, el grupo decide que quieren algo previo débil que cambiará rápidamente, basado en nueva información de las pruebas. Si el parámetro previo "a" se establece en 1, la gamma tiene una desviación estándar igual a su media, que le hace "débil".

   Para asegurar el percentil 50 se establece en λ50 = 1 /MTBF50 de MTBF, tenemos que escoger b = ln 2 × MTBF50, que es aproximadamente 0,6931 × MTBF50.

   Nota: como vamos a ver cuándo estamos con un plan de pruebas Bayesiano, esta debilidad previa es realmente un muy amigable en términos de ahorro de tiempo de prueba.

Muchas variaciones son posibles, partiendo de los tres métodos anteriores. Por ejemplo, podría tener datos anteriores de fuentes en que no se confía completamente. O se podría

cuestionar si los datos se aplican realmente al sistema bajo investigación. Se podría decidir el "peso" de los datos anteriores en 0.5, para "debilitarlo". Esto se puede implementar mediante el establecimiento de un a=0,5 x el número de fallos en los datos previos y b = 0,5 veces el número de horas de prueba.
Se extiende la distribución previa más y le permite reaccionar más rápidos a los nuevos datos de prueba.

Las consecuencias son que no importa cómo llegar a los valores de los parámetros previos de Gamma a y b, el método para incorporar nueva información de la prueba es el mismo. La nueva información se combina con el modelo previo para producir una versión actualizada o el modelo de distribución posterior para λ.

Bajo supuestos 1 y 2, cuando una nueva prueba se ejecute con sistema T horas de sistema operativo y r fallos, la distribución posterior para λ sigue siendo una gamma, con nuevos parámetros:

a' = a + r, b' = T + b

En otras palabras, se añade al parámetro “a” el número de nuevos fallos y al parámetro “b” el número de nuevas horas de pruebas para obtener los nuevos parámetros de la distribución posterior.

Aproximación del modelo Bayesiano a la Fiabilidad del Software.

La aproximación del modelo bayesiano no se enfoca sobre la estimación inherente de contabilizar los fallos, N, pero se concentra sobre la tasa de fallos sobre fallos.

Las aproximaciones clásicas asumen que la fiabilidad y la tasa de fallo son una función de la detección de fallos, mientras en otro sentido, la aproximación bayesiana asume que un programa de software el cual ha tenido operación libre de fallos tiene más probabilidad de ser fiable.
La aproximación bayesiana también difiere de los otros modelos porque incluye suposiciones de "conocimiento previo", es decir utiliza aproximaciones que pueden ser interpretadas algunas veces como subjetivas.

Este modelo asume que:

- El software es operacional.
- Las fallos de software ocurren con una tasa desconocida λg y asumen una distribución Gamma con parámetros Xi y fi + 1.
- Los fallos son corregidos entre los periodos de pruebas, pero no durante estos periodos.
- El número total de fallos observados en un periodo de pruebas simple de longitud ti, sigue una distribución Poisson con parámetro λti.

El modelo asume que hay i periodos de pruebas, cada uno con longitud, ti (no es igual para todos los periodos); donde el número de fallos detectados durante ese periodo es representada por: fi.

La información subjetiva es insertada como una ocurrencia en el periodo O (tO), y fO representa la información previa. Si no hay información previa, ese conjunto de valores son colocados en cero. El valor del número previo de fallos también depende de experiencias pasadas y es independiente del tiempo previo.

El término de Ti representa el total acumulado de periodos de pruebas sobre el rango completo, desde el periodo O a i y Fi representa el total acumulado de fallos, fi, sobre el rango completo, desde el periodo O a i.

Entonces, la fiabilidad en función del tiempo (en el intervalo i) es expresada como una función de los valores desde el intervalo previo (i - () hasta el intervalo ith actual de datos. Lo anterior se expresa como:

$$R(t) = [T_{i-1}/(T_{i-1} + t)]^{F_{i-1}}$$

La tasa de fallo estimada para el tiempo t (en el intervalo i) es dada por:

$$\lambda(t) = (F_{i-1} + 1)/T_{i-1}$$

Los beneficios de este modelo se encuentran relacionados con sus supuestos: N no es asumida contradictoria, la fiabilidad de software no es asumida directamente una función de N y para los fallos, no es necesaria la corrección inmediata de ellas. La desventaja general de este modelo es que no es universalmente aceptado, desde que existe la premisa de la inclusión de información previa, reflejando desacuerdos en el análisis de lo que se cree acerca de la tasa de fallos.

7. COMPARACIÓN DE LA APLICACIÓN DE LOS DIFERENTES MODELOS A UN CASO CONCRETO

7.1 INTRODUCCIÓN

En este capitulo vamos a centrar nuestros esfuerzos en utilizar los diferentes métodos de estimación de la fiabilidad en un sistema software concreto.

Hemos escogido un Portal Web perteneciente a una empresa de reciente creación, que se dedica al desarrollo de aplicaciones orientadas al entretenimiento on line.

Actualmente la aplicación se encuentra en Integración, es decir, esta totalmente estructurada a nivel de módulos y flujos de información, y se han realizado diversos casos de prueba para verificar el correcto funcionamiento del core de la aplicación, pero todavía no esta puesto al publico para su explotación comercial esta terminado y funcionando.

7.2 AMBITO DE LA APLICACIÓN

La empresa desarrolladora, dedicada al entretenimiento digital, ha desarrollado una aplicación a través de un Portal Web.
Este site ofrece las funciones de simulación de una Liga de Futbol entre los usuarios que entren y se validen en la Página Web.

Debido al volumen de información recogido en este portal, se está estudiando adaptar la totalidad de las páginas.

El ámbito de aplicación es a nivel nacional, puesto que la simulación de partidos sería con equipos españoles y los usuarios de la página estarían entre 15 y 45 años.

El ámbito de aplicación en este sentido, estaría orientado sobre todo a un uso doméstico y privado.

Hemos escogido una aplicación comercial de este tipo, sobre todo por dos razones:

- Hemos asistido en primera persona a las sesiones del propio concept del producto, su diseño y su estructuración, lo que nos permite tener una visión muy amplia de en que consiste la aplicación, la metodología utilizada y como se han decidido cuestiones como el diseño.

- No se suele utilizar ningún modelo de fiabilidad en las aplicaciones comerciales de este tipo, ya que se basan sobre todo en prueba/error/corrección. Por este motivo, nos pareció útil aplicar los diferentes modelos estudiados.

7.3 PERSPECTIVAS DEL PROYECTO

Al ser una aplicación concebida para el entretenimiento entre diferentes usuarios conectados al portal, las perspectivas son conseguir una masa crítica de gente suficiente que pueda hacer rentable la aplicación.

Cuando se consigue un número elevado de conexiones al site, entonces es cuando la aplicación es interesante en términos económicos, puesto que a mayor tráfico generado, mayor es la posibilidad de conseguir patrocinadores, inversores....etc.

El proyecto esta dividido en tres fases claramente diferenciadas que prácticamente lo convierten en un producto diferente:

1. Una primera fase en la que se busca conseguir un comportamiento exacto al de la Liga Española de Futbol, esto es, sistema de puntuaciones, reglas, arbitraje.....etc.

   Nota: En esta fase, se busca la rentabilidad mediante masa crítica de usuarios y el tráfico hacia el site.

2. Una segunda fase en la que se busca ampliar el comportamiento del producto, creando personalizaciones a nivel de usuario y una monetización de la misma.

3. La tercera fase estaría compuesta de su adaptación en plataformas móviles Android y IPhone.

La puesta en marcha de la primera fase esta prevista para finales de 2013, la segunda para mediados de 2014 y la tercera para finales de 2014.

7.4 HERRAMIENTAS EMPLEADAS

La aplicación ha sido desarrollada mediante las siguientes herramientas, toda ellas son software libre y se pueden encontrar para sus descargas en sus respectivas páginas Web.

Para la infraestructura Web se decidió por un sistema conocido en el argot como LAMP.
LAMP es el acrónimo usado para describir un sistema de infraestructura de internet que usa las siguientes herramientas:

- **L**inux, el sistema operativo.
- **A**pache el servidor web.
- **M**ySQL, el gestor de bases de datos.
- **P**erl, PHP o Python, los lenguajes de programación.

La combinación de estas tecnologías es usada primariamente para definir la infraestructura de un servidor web, utilizando un paradigma de programación para el desarrollo.

A pesar de que el origen de estos programas de código abierto no han sido específicamente diseñado para trabajar entre sí, la combinación se popularizó debido a su bajo coste de adquisición y ubicuidad de sus componentes (ya que vienen pre-

instalados en la mayoría de las distribuciones linux). Cuando son combinados, representan un conjunto de soluciones que soportan servidores de aplicaciones.

En el caso del lenguaje de programación, hemos optado por PHP, por los siguientes motivos:

- Es un lenguaje multiplataforma.
- Completamente orientado al desarrollo de aplicaciones web dinámicas con acceso a información almacenada en una Base de Datos.
- El código fuente escrito en PHP es invisible al navegador y al cliente ya que es el servidor el que se encarga de ejecutar el código y enviar su resultado HTML al navegador. Esto hace que la programación en PHP sea segura y confiable.
- Capacidad de conexión con la mayoría de los motores de base de datos que se utilizan en la actualidad, destaca su conectividad con MySQL y PostgreSQL.*
- Capacidad de expandir su potencial utilizando la enorme cantidad de módulos (llamados ext's o extensiones).
- Posee una amplia documentación en su página oficial (Sitio Oficial), entre la cual se destaca que todas las funciones del sistema están explicadas y ejemplificadas en un único archivo de ayuda.
- Es libre, por lo que se presenta como una alternativa de fácil acceso para todos. Permite aplicar técnicas de programación orientada a objetos.
- Biblioteca nativa de funciones sumamente amplia e incluida.
- No requiere definición de tipos de variables aunque sus variables se pueden evaluar también por el tipo que estén manejando en tiempo de ejecución.
- Tiene manejo de excepciones (desde PHP5).

Si bien PHP no obliga a quien lo usa a seguir una determinada metodología a la hora de programar (muchos otros lenguajes tampoco lo hacen), aun estando dirigido a alguna en particular, el programador puede aplicar en su trabajo cualquier técnica de programación y/o desarrollo que le permita escribir código ordenado, estructurado y manejable. Un ejemplo de esto son los desarrollos que en PHP se han hecho del Patrón de diseño Modelo Vista Controlador (o MVC), que permiten separar el tratamiento y acceso a los Datos, la Lógica de control y la Interfaz de usuario en tres componentes independientes (Frameworks).

El Framework utilizado para desarrollar toda la estructura de clases es symfony (www.symfony.es) debido a que los programadores encargados del proyecto eran especialistas en este marco.

7.5 ANALISIS Y DISEÑO DEL SISTEMA

La fase de análisis trata de obtener toda la información posible sobre el sistema a desarrollar para expresarla en función de requisitos. Así, en este caso, se puede obtener información observando el funcionamiento de otros simuladores para generar algunos de los requisitos.

Hemos creado un diagrama de casos de uso para mostrar las situaciones en las que se puede encontrar un usuario al encontrarse con la página Web y como interacciona entre los distintos elementos.

En la siguiente hoja hemos puesto todo el diagrama, y luego lo hemos dividido en 3 grandes bloques para posteriormente ir explicándolas.

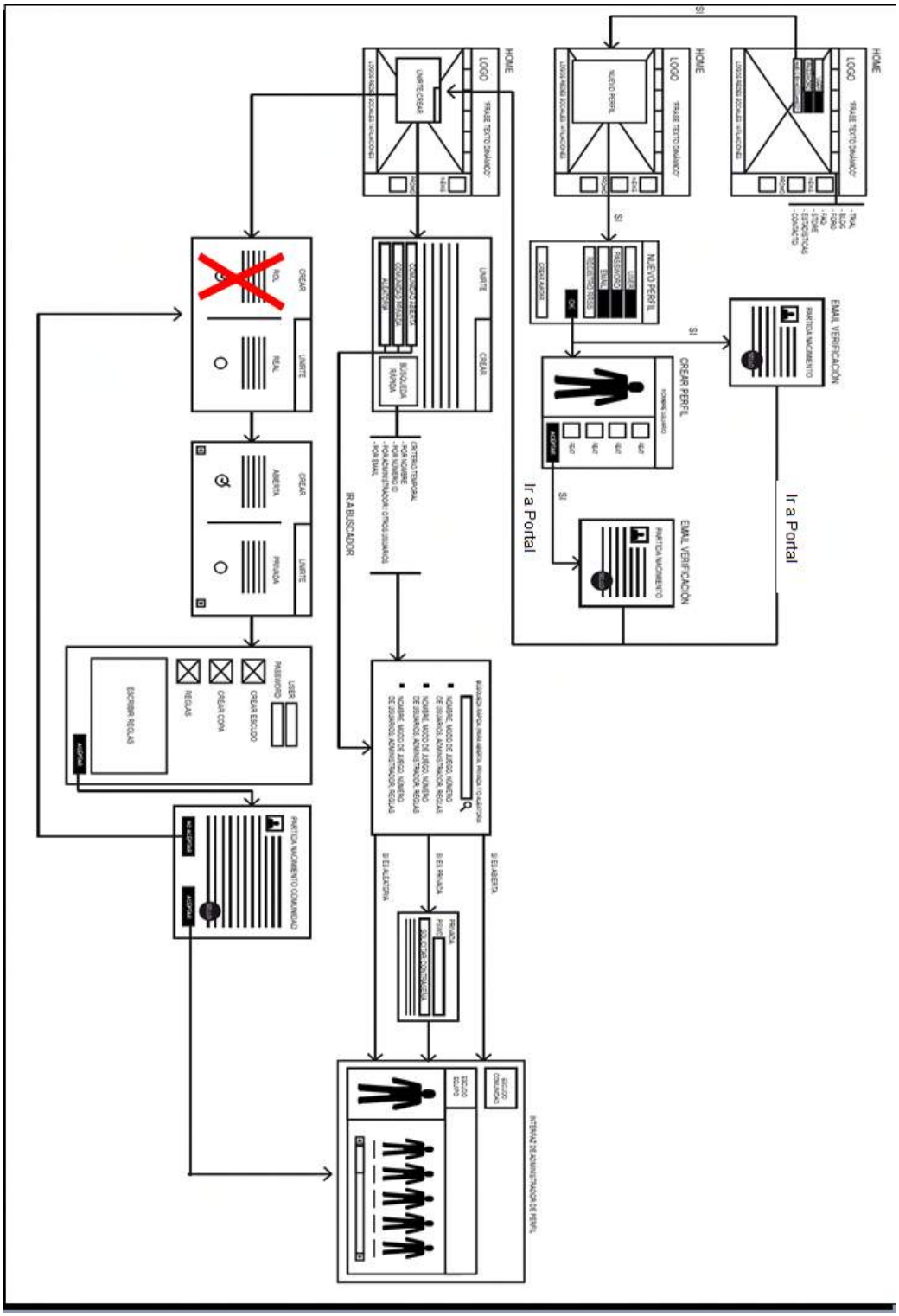


Figura 7.1 Diagrama completo del programa.

7.5.1 PRIMER BLOQUE DE APLICACIÓN--> REGISTRO DE USUARIOS Y COMUNIDADES

Este primer diagrama muestra como al entrar en la página Web nos solicitará que introduzcamos nuestro usuario y contraseña, y en este momento nos podemos encontrar con dos posibilidades:

1. Que no este dado de alta, en cuyo caso, si queremos continuar, nos llevará al diagrama de CREAR PERFIL y como medida de verificación utilizaremos el método de enviarle un mail para comprobar que es la persona que dice ser.

2. Que ya este registrado, en cuyo caso, con nuestro usuario contraseña y nos lleve directamente a la misma página de Inicio, pero con nuestro alta realizada.

A partir de aquí, accederemos al modulo de UNION/CREACION DE COMUNIDAD, este modulo permitirá dar de alta una comunidad nueva o unirse a una ya existente.

Al darse de alta en la aplicación Web, un usuario pasa directamente a pertenecer a la Comunidad Global, que esta compuesta por todos los usuarios registrados de la página, y posteriormente, un usuario puede crear una comunidad nueva o unirse a una ya existente.
Este tipo de comunidades son subgrupos de la Comunidad Global y les permite a los usuarios comparar los resultados de sus alineaciones solo con los usuarios de esa Comunidad.

A su vez, las comunidades tienen una puntuación que les permite compararse con las restantes.

También esta la opción de crear una Comunidad privada que solo permite agregar usuarios determinados que hayan sido invitados previamente y que no se pueda unir cualquiera. Esta opción aparece reflejada en las opciones de Comunidad Abierta, Comunidad Privada y Aleatoria.

7.5.2 SEGUNDO BLOQUE DE APLICACIÓN--> UNIRSE A COMUNIDADES DENTRO DE LA APLICACIÓN

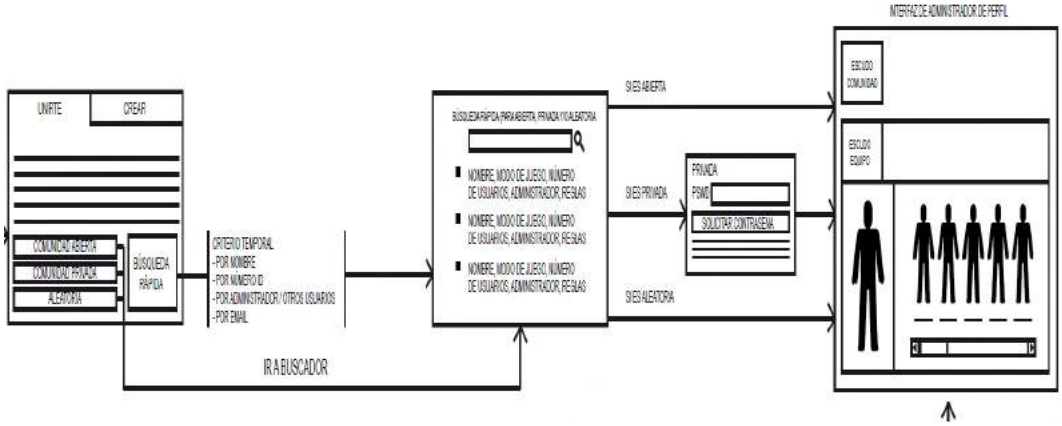


Figura 7.2 Segundo bloque del diagrama.

En este segundo bloque podemos ver como desde el modulo anterior de registro de usuarios accedemos a la opción UNIRTE.

En este punto, el usuario debe decidir a que Comunidad quiere unirse y de que tipo pertenece (Privada, Abierta o Aleatoria).

Una vez elegida la Comunidad, accedemos al modulo INTERFAZ DE ADMINISTRADOR DE PERFIL

CREAR COMUNIDAD

Este modulo será el core de la aplicación, puesto que desde aquí se gestionaran todas las opciones principales de creación y gestión del equipo de futbol, las principales son:

- Selección de escudo del equipo.
- Selección de jugadores.
- Selección de suplentes.
- Edición de jugadores.
- Compra/Venta/Cambio de jugadores.
- Calculo de estadísticas y valoración del equipo.

7.5.3 TERCER BLOQUE DE APLICACIÓN--> CREAR COMUNIDADES DENTRO DE LA APLICACIÓN

El tercer bloque de la aplicación es el que pertenece a la segunda fase del proyecto, y consiste en un modo de juego totalmente diferente al que se esta planteando, con opciones mucho mas abiertas para el usuario, como la creación de reglas personalizadas e incluso campeonatos propios.

Al pertenecer a la segunda fase, todavía no esta desarrollado, pero si contemplado desde donde se debe de acceder en un futuro, que es a través de la opción de CREAR y dentro de esa opción la opción de ROL.

En este caso, la única opción de la que disponemos es la de REAL.

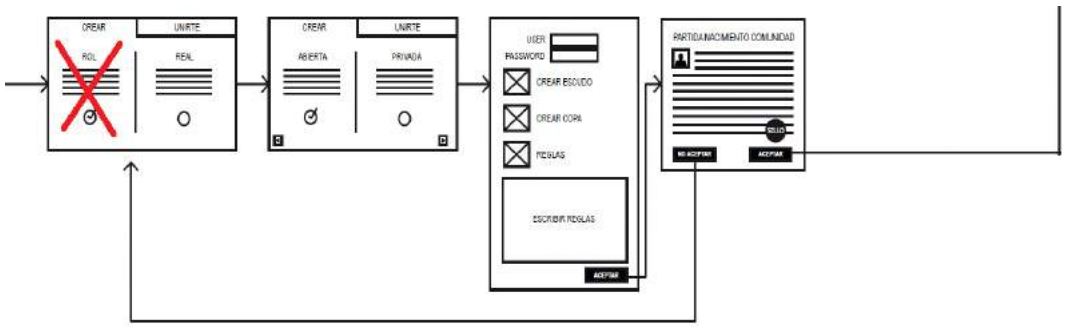


Figura 7.3 Diagrama del tercer bloque.

7.6 REQUISITOS FUNCIONALES

Después de las explicaciones previas, esta claro, que habrá una serie de requisitos funcionales claros, como son que un usuario solo puede darse de alta una vez, que solo se puede pertenecer a una o varias Comunidades concretas, además de la Global.

Al darse de alta e ingresar en una Comunidad, se le permite confeccionar un equipo de futbol con los jugadores que hay actualmente jugando en la Liga de Futbol Profesional de España solo desde la Interfaz de Administrador de Perfil.

En esta fase se le asigna un presupuesto para confeccionar su equipo que será el mismo para todos.
Dentro de un equipo no se pueden repetir jugadores, pero si esta permitido repetir jugadores entre diferentes comunidades.

Estos jugadores seleccionados por el Usuario son valorados en función de las acciones que hayan realizado durante la semana, es decir en función del desempeño real del jugador, y mediante dicha puntuación, se obtiene una valoración del equipo que se ha confeccionado.

Esta aplicación Web, desde el Interfaz de Administrador de Perfil, permite la compra, venta y cambio de jugadores entre usuarios de la Comunidad conforme a unas reglas de juego preestablecidas, pudiéndose cambiar uno por uno, varios jugadores por uno solo o uno por varios jugadores.

7.7 REQUISITOS DE INTERFAZ

La mayor parte del contenido es accesible y ha sido validado por el Test de Accesibilidad Web (TAW) y por el test de validación xhtml del W3C en su nivel estricto.

Aunque esta claro que existen muchas páginas que ofrecen simuladores de futbol, con las mismas normas que la que estamos exponiendo, esta pretende desmarcarse del resto ofreciendo una interfaz mucho más amigable, que es donde estas aplicaciones suelen fallar.

Por ejemplo, la mayoría suele ofrecer un volumen grande de datos pero difícil de gestionar, esto provoca que a la mayoría de los usuarios acaban abandonando la página por ser complicada y molesta tanto la navegabilidad como la forma de mostrar la información.

Así mismo, otro de los puntos por los que pretende conseguir una masa crítica de usuarios es por la creación de una identidad, es decir, una marca reconocible, otro de los puntos donde las páginas suelen fracasar.

Para ellos cuenta con una marca y un diseño propio que a su vez sea fácil de promocionar.

En una sola palabra final, se pretende fidelizar al cliente, ya no solo por el juego, sino por lo que la marca representa.

Y por último, la creadora de la Web pretende ser un jugador más,.poniéndose a disposición del resto de usuarios para poder interactuar con ellos y conocer su grado de satisfacción.

En definitiva, deberá cumplir con tres requisitos fundamentales en su diseño para garantizar una adecuada funcionalidad:

- Navegación: Los visitantes de la página deberán poder tener acceso a toda la información de manera fácil y rápida.

- Interacción: Deberá lograrse una interacción entre la organización y los visitantes de la página, proporcionándoles información valiosa.

- Retroalimentación: la página deberá brindar al visitante la posibilidad de recibir retroalimentación por parte de la organización respecto de las preguntas que aquel tenga sobre los productos y la propia organización.

En definitiva, el diseño es el 90% del trabajo dentro de la planificación de los trabajos.

7.8 ESTIMACIÓN DE TIEMPO Y COSTES

Se ha valorado de manera oficial el tiempo y los costes de este programa yse ha determinado que como equipo de trabajo se necesitan.

2 programadores con experiencia de más de 7 años en programación Web con entornos LAMP

- 1 Maquetador con un mínimo de 5 años de experiencia.
- 1 Diseñador de Páginas Web con un mínimo de 5 años de experiencia.

El tiempo necesario para tener una maqueta funcional es de 4 meses, y 8 meses para tener un producto completo en Producción, esto en horas supone 640 horas y 1280 respectivamente por cada persona.

Es decir, tenemos un total de horas trabajadas de 2560 horas y 5120.

El coste estimado teniendo en cuenta que valoramos la hora a 18 euros es un total de 46080 euros para la maqueta funcional y de 92160 para el producto final terminada.

7.9. CALCULO DE FIABILIDAD MEDIANTE MODELOS DE PREDICCION

7.9.1 MODELO DE DATOS HISTORICOS INTERNOS

Como ya hemos comentado anteriormente, plantear el cálculo de la fiabilidad desde un punto de vista predictivo en el caso de esta aplicación es un error ya de base, puesto que no es posible obtener datos históricos de un programa que se esta desarrollando desde el principio y en donde no se tienen datos recogidos de otros programas parecidos.

La única salvedad sería compararlo con programas parecidos en tamaño de código y/o que ofrezcan funcionalidades parecidas.

Por estos motivos, no es de aplicación este modelo.

7.9.2 MODELO DE MUSA

Al igual que el anterior caso, el Modelo de Musa esta pensado para ofrecer una estimación de la tasa de fallos de un programa, actualmente el programa se encuentra terminado en la fase de definición de clases y de interacción entre las mismas, pero hay mucho código pendiente de completar (actualmente el programa se encuentra en unas 10.000 líneas de código).

Vamos a realizar a modo didáctico una estimación del modulo core de la aplicación.

Vamos a suponer que nos encontramos en tiempo de ejecución, cuando realmente es tiempo de calendario, y que salvo errores hardware (seguimos suponiendo que los sistemas hardware siempre estará funcionando correctamente) el modulo principal este ejecutándose 24 horas al día los 365 días del año las 10.000 líneas que componen el core y que el ratio de ejecución es de 1500 líneas por segundo.

Nuestra tabla quedaría como sigue:

Símbolo	Representación	Valor
K	Constante que acumula para estructuras dinámicas de los programa	K=4.2E-7
P	Estimación del numero de ejecuciones por unidad de tiempo	p=R/SLOC/ER
R	Promedio de la tasa de ejecución de instrucciones, determinada por el constructor	Constante
SLOC	Líneas de código fuente (no se incluye código reusado)	10.000
ER	Razón de expansión, una constante que depende del lenguaje	PHP
W o	Estimación del numero inicial de fallos en el programa	Puede ser calculado usando W o = N * B * un valor por defecto de 6(fallos/(000)SLOC
N	Numero total de ocurrencia de fallos	100 es el número estimado de fallos que puede tener la aplicación
B	Fallos no corregidos antes de que el producto sea liberado.	5

Figura 7.4: Resumen fiabilidad modelo Musa.

$$\lambda_0 = k \times p \times w_o$$

(4.2E-7) x (1500/5000/3) x (5*15)=. 0,00000315 fallos por segundo, o lo que es lo mismo 99 fallos por año.

Esta tasa de fallo, para una aplicación que debe dar servicio en red, es bastante alto, por lo que deberíamos procurar tener un nivel de ocurrencia de fallos mucho más bajo, para que mejorara la tasa.

7.9.3 MODELO DE PUTNAM

Para realizar un ejemplo, en el software que estamos tratando, lleva unos 8 meses de desarrollo (td), si queremos calcular la proporción de fallos esperados por mes, aplicando la fórmula quedaría como sigue:

f(t)=(0,09375N)tEXP(-0,046875t^2)

Con lo que nuestro cuadro mes a mes quedaría como sigue:

Meses	f(t)	F(t)	Nivel
1	0,089456875	0,089456875	
2	0,15544296	0,244899835	1
3	0,184448253	0,429348088	2
4	0,177137457	0,606485545	3
5	0,145211976	0,751697521	4
6	0,104052037	0,855749558	5
7	0,066000797	0,921750355	6
8	0,037340301	0,959090656	7
9	0,018934416	0,978025073	
10	0,008634077	0,986659149	8

Figura 7.5: Valores para tiempo de Putnam

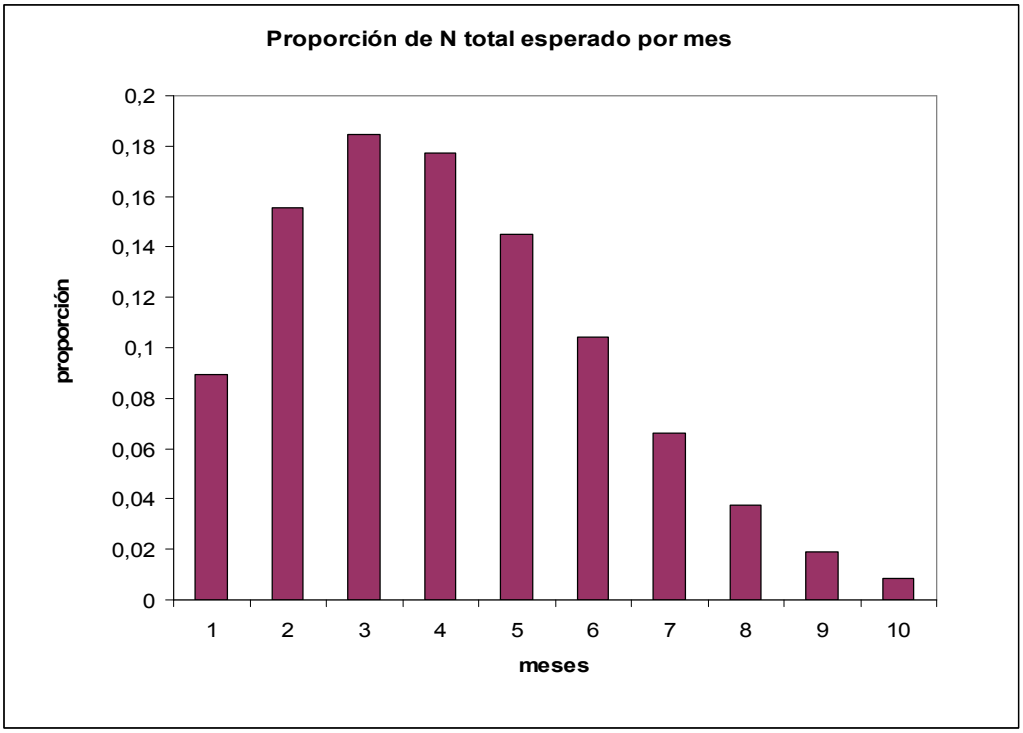


Figura 7.6: Gráfica Putnam.

La proporción de N fallos para el mes 8 es del 0,037, en el que se conseguiría un 95% de fiabilidad.

7.9.4 MODELO ROME RL-TR-92-52

Ya hemos visto que el modelo Rome 92-52 depende en sus fórmulas de la fase en la que se encuentre el proyecto, por tanto, vamos a realizar ese ejercicio para las diferentes fases.

En el caso de asignación de valores para el factor A y D, hemos asignado lo siguiente:

- A: le hemos asignado un valor de 6, ya que se trata de una aplicación que aunque compleja, esta desarrollada dentro de un sector en el que el único peligro que se puede dar debido a una mala fiabilidad, es no dar el servicio adecuado, con la consiguiente perdida de resultado, pero en ningún caso se trata de una aplicación relacionada con al seguridad o la salud.

- D: Le hemos asignado el valor de 0,5, ya que se trata de utilizar técnicas y métodos de programación que en ningún caso requieren de sofisticación extrema, tales como cifrados seguros de nivel elevado, acceso a Bases de Datos con información relevante o un servidor configurado de manera segura.

Ya hemos comentado antes, que el core de la aplicación contiene unas 5000 líneas de código.

PREDICCION DE LA FASE DE REQUISITOS

Este método sólo requiere que la información sobre el tipo de aplicación y la organización del desarrollo se conozca. Cuanto menor es el valor calculado, menor será la densidad de fallo y pronosticaron tasa de fracaso.

Factor	Measure	Range of values
A - Application	Difficulty in developing various application types	2 to 14
D - Development organization*	Development organization, methods, tools, techniques, documentation	.5 to 2.0

Figura 7.7: Factores de la Fase de requisitos.
* Puede no ser conocido durante esta fase.

D = fallos esperados por KSLOC = A si D no es conocido o

FD = fallos esperados por KSLOC = A*D si D es conocido

En nuestro caso: 6*0,5 =3, es el número de fallos esperado por 1000 líneas de código.

N = numero inherente estimado de fallos = FD*KSLOC

En nuestro caso será 3*5=15

PREDICCION EN LA FASE DE DISEÑO

Este método solo requiere que la información sobre el tipo de aplicación, la organización, del desarrollo, y los requisitos de diseño sean conocidos. Cuanto menor es el valor calculado, menor será la densidad de fallo y el ratio de fallo esperado.

Factor	Measure	Range of values
A - Application	Difficulty in developing various application types	2 to 14
D - Development organization	Development organization, methods, tools, techniques, documentation	.5 to 2.0
SA - Software anomaly management*	Indication of fault tolerant design	.9 to 1.1
ST - Software traceability*	Traceability of design and code to requirements	.9 to 1.0
SQ - Software quality*	Adherence to coding standards	1.0 to 1.1

Figura 7.8: Factores de la Fase de Diseño.

A pesar de que normalmente no hay código en la fase de diseño, sin embargo, en esta fase los parámetros pueden medirse en base a las prácticas de codificación que se están utilizando.

Por tanto, de la formula original:

FD = fallos esperados por líneas de código = A*D

Si hay políticas de realización de código que se estén teniendo en cuenta entonces:

FD = fallos esperados por KSLOC = A*D*SA*ST*SQ

En nuestro caso, se ha tomado la siguiente valoración:

SA = 1

ST = 1

SQ = 1,05

En nuestro caso: 6*0,5*1*1*1.05 =3,15 es el número de fallos esperado por 1000 líneas de código.

N = numero estimado de fallos inherentes = FD*KSLOC

Durante las fases de diseño de los factores A y D deben ser conocidos.

PREDICCION EN LA FASE DE CODIFICACIÓN, UNITARIAS E INTEGRACION

Este método requiere que la información sobre el tipo de aplicación, la organización deL desarrollo, los requisitos de diseño y las prácticas de codificación sean conocidas. Cuanto menor es el valor calculado, menor será la densidad de fallo y el ratio de fallo esperado.

Factor	Measure	Range of values
A - Application	Difficulty in developing various application types	2 to 14
D - Development organization	Development organization, methods, tools, techniques, documentation	.5 to 2.0
SA - Software anomaly management	Indication of fault tolerant design	.9 to 1.1
ST - Software traceability	Traceability of design and code to requirements	.9 to 1.0
SQ - Software quality	Adherence to coding standards	1.0 to 1.1
SL - Software language	Normalizes fault density by language type	n/a
SX - Software complexity	Unit complexity	.8 to 1.5
SM - Software modularity	Unit size	.9 to 2.0
SR - Software standards review	Compliance with design rules	.75 to 1.5

Figura 7.9: Factores de la Fase de codificación, unitarias e integración.

FD = fallos esperados por líneas de código = A*D*SA*ST*SQ*SL*SX*SM*SR

En nuestro caso, se ha tomado la siguiente valoración:

SL = 1

SX = 0.9

SM = 1.2

SR= 0.85

En nuestro caso: 6*0,5*1*1*1.05*1*0.9*1.2*0.85 =2,89 es el número de fallos esperado por 1000 líneas de código.

N = numero estimado de fallos inherentes = FD*KSLOC

7.10 CALCULO DE FIABILIDAD MEDIANTE MODELOS DE ESTIMACION

7.10.1 MODELO DE DISTRIBUCION EXPONENCIAL

El programa para el que hemos estado calculando la fiabilidad apenas lleva 6 meses en funcionamiento, y el cálculo de la fiabilidad se suele realizar en periodos anuales, por lo que hemos extrapolado los resultados de estos 6 meses a un año.

Dentro de los perfiles operacionales que se han planteado alrededor del programa, nos hemos encontrado con 20 errores en unas 500 ejecuciones, por tanto, la tasa de fallo sería la siguiente:

λ= 20/500=0,04.

Aplicando la fórmula de Fiabilidad R(t) nos da lo siguiente:

R (t) = exp (-λt) = exp(-0,04*1) = 0,9607 o lo que es lo mismo, tiene una fiabilidad del 96% para un periodo de un año.

La función exponencial de densidad de probabilidad de fallo:

f(t)= 0,04 * exp(-0,04*1) = 0,03843

El tiempo medio hasta un fallo (MTTF→ mean time to next failure) expresado como:

MTTF = 1/0,04 = 25 días

El Tiempo medio entre fallos (MTBF--> mean time beetwen failure) esta expresado como:

MTBF = m = 1/λ = MTTF es decir siguen siendo 25 días

Si quisiéramos saber la probabilidad de que el programa falle antes de alcanzar un tiempo de funcionamiento de 4 meses, tendríamos que recurrir a la fórmula de la infiabilidad, que sería igual a:

Q (t): Q(t)= 1 - exp(-λt)

Con λ = 0,04 y t es el tiempo expresado en años

Luego, para t = 1/3, se tendrá:

Q (t) = 1 - exp (-0,04*1/3) = 1 - 0,9867 = 0,01324

La probabilidad de que el programa falle antes de cuatro meses será del 1,32 %.

Si queremos calcular la probabilidad de que el programa siga ejecutándose sin que se haya producido el fallo, tendremos que recurrir a la fórmula de la fiabilidad, esto es:

R (t) = exp (-λt) = exp (-0,04*1/3) = exp (- 0,002) = 0,9867

Esto quiere decir que existe una probabilidad del 98,67 % de que no se produzcan errores en estos 6 meses.

El gran inconveniente de este modelo es que fue concebido para estimar la fiabilidad del hardware, por tanto, hay una serie de supuestos que no se pueden establecer, uno sería que el software ya se encuentra en estado operacional, y otro es que la tasa de fallos en un modelo exponencial se mantiene constante a lo largo de toda su vida útil, es decir, la tasa de fallos es la misma para el periodo de t=0 hasta t=X, siendo X el final de u vida útil.

Se llama "vida útil" el periodo de vida de un programa durante el cual es válida la fórmula indicada de la fiabilidad, por tanto, es imprescindible que el tiempo t que utilicemos en la fórmula no supere la vida útil del programa.

Así, para un programa con una vida útil de 10 años, la fórmula de la fiabilidad será la misma para el año 1 que para el año 9.
A partir de t>10 la fórmula exponencial no es aplicable porque, terminada la vida útil, la tasa de fallos del dispositivo no es constante y empieza a crecer significativamente.

Esto ya de por si nos plantea un problema importante, y es que los programas no suelen tener un fecha de vida útil como la que pueda tener un dispositivo hardware.
Además, los componentes software suelen tener un mantenimiento evolutivo y correctivo a lo largo de toda su vida, que hace enormemente difícil la estimación de la fiabilidad.

Para no tener problemas con este modelo, es imprescindible tener una estimación más o menos coherente de la vida útil del programa.

7.10.1.1 MODELO EXPONENCIAL GENERAL

Para este modelo, tenemos la siguiente tasa de fallos:

λ= k(N-c)

Si para nuestro programa suponemos un número total de fallos N=100 y hemos conseguido corregir durante las fases de programación y pruebas un total de 70, para alcanzar la misma tasa de fallos de 0,04 tenemos la siguiente fórmula:

λ= k(N-c) --> 0,04 = k (100-70) despejando k nos da una constante de 0,0013

Suponemos la tasa de fallos actual λp= 0,04 y queremos una tasa de fallos final λf de 0,01

La proyección del número de fallos que se necesitan detectar para alcanzar el ratio de fallo final λf esta dado por:

Δn=(1/k) λp/λf --> (1/k) luego 1/0,0013 = 769

La proyección del tiempo necesario para alcanzar un ratio de fallo esta dado por:

Δt = (1/K)ln[λp/λf]

7.10.1.2 LLOYD-LIPOW MODEL

Recordemos que Exponencial de Lloyd-Lipow también asume que todos los fallos son iguales en severidad y probabilidad de detección. La diferencia con el modelo previo es que en la aproximación de Lloyd-Lipow, el ratio de fallos λ esta relacionado directamente con el numero de fallos pendientes de ser detectadas (no corregidas) en el software. Esto es, λ es una función del número de fallos detectadas n:

λ=K(N-n)

Si tenemos que el número de fallos detectados son 80, tenemos que:

λ = 0,0013(100-80) = 0,026

La expresión para el MTTF, Δn e Δt son las mismas que en el modelo exponencial general.

Esta fórmula del modelo exponencial no requiere corrección de defectos, solo detección.

7.10.1.3 MODELO BÁSICO DE MUSA

Recordemos que este modelo tiene en cuenta el tiempote ejecución de la CPU, no el tiempo de calendario, por lo que la interpretación de los datos es muy diferente.

Por poner un ejemplo, suponemos que nuestro programa tiene 100 errores, y el número de fallos experimentados hasta ahora ha sido de 80.

La tasa inicial de fallos es de 2 por cada hora de CPU.

Es decir, nuestro λ0 = 2.

Nuestro ratio de fallo actual sería:

λ n = λ0(1 – n/v) que es igual a 2[1-80/100] = 0,4 fallos a la hora.

Mientras la expresión para el ratio de fallo en el tiempo t esta dado por:

$$\lambda_t = \lambda_0 \exp[-(\lambda_0/\upsilon)\tau]$$

Si queremos saber el ratio de fallo a las 10 horas, entonces la fórmula será

λ 10 = 2 exp[-(2/100)*10] = 1,637 fallos.

La proyección del número de fallos que necesitan ser detectados para alcanzar una ratio de fallo final λf esta dado por:

Δn = N/ λ0(λp – λf)

Si queremos un ratio de fallos final λf del 0,5 y tenemos un ratio de fallos actual de 1, nuestra fórmula quedaría de la siguiente manera:

Δn = 100/2(1-0,5) = 0,01 fallos que hay que detectar.

La proyección del tiempo necesario para alcanzar un ratio de fallo proyectado esta dado por:

Δt = N/ λ0 ln (λp – λf)

Por tanto, tendremos que:

Δt = 100 / 2 ln (1-0,5) =72,13 días para alcanzar este ratio de fallo.

7.10.1.4 MODELO LOGARÍTMICO DE MUSA

Recordemos las fórmulas siguientes:

Tasa de fallos:

$$\lambda_n = \lambda_0 \exp(-ft)$$

Mientras la expresión para la tasa de fallos en función del tiempo es dada por:

$$\lambda_t = \lambda_0/(\lambda_0 f t + 1)$$

$$\lambda(\tau) = \lambda_0/(\lambda_0\theta\tau + 1)$$

Donde:

f = parámetro de reducción de la intensidad del fallo, es decir, el cambio relativo de n/t sobre n.

La proyección del número de fallos que necesitan ser detectados para alcanzar un ratio de fallo final λf esta dado por:

$\Delta n = 1/f \ln (\lambda p/ \lambda f)$

$\mu(\tau) = (1/\theta).\lambda\nu(\lambda_0\theta\tau + 1)$

La proyección del tiempo necesario para alcanzar un ratio de fallo proyectado esta dado por:

$\Delta t = 1/f (1/ \lambda f – 1/ \lambda p)$

Para el programa que nos ocupa hemos decidido que experimentará 100 fallos, y al tratarse de tiempo de ejecución tenemos que suponer que se trata en tiempo indeterminado.

La intensidad de fallo inicial vamos a suponer que ha sido de 4 fallos/hora CPU y que la intensidad de fallo actual es 1 fallo/hora CPU.

Vamos a proponernos el objetivo de 0,0002 fallos/HORA/CPU

El tiempo adicional de pruebas necesario para llegar a este objetivo será:

Con las fórmulas anteriores, tenemos que:

v_0 = 100 fallos.
λ_0 = 4 fallos/ hora CPU.
λ_p = 1 fallo/hora CPU.
λ_f = 0,0002 fallos /hora CPU.

Tiempo de pruebas para alcanzar dicho valor es el siguiente:

$(\tau_2 - \tau_1) = (v_0/ \lambda_0).\ln(\lambda_p/ \lambda_f)$= (100/4)ln(2/0,0002) = 230,25 horas CPU.

7.10.1.5 MODELO DE SHOOMAN

El modelo de Shooman es similar al modelo exponencial general, excepto que cada fallo contabilizada es normalizada para las líneas de código en ese punto del tiempo.

Antes, λ = k(N – c); aquí esta dado por:

λ = k SLOC (N/SLOC – c/SLOC)

Luego:

λ = 0,0013*10000(100/10000-80/10000) = 0,026 tasa de fallos

La ecuación de MTTF = 1/ λ = 38,46.

7.10.1.6 MODELO GOEL-OKUMOTO

Este modelo esta expresado como:

λ t = ab exp(-bt)

Donde a y b son resueltos iterativamente desde la siguiente fórmula:

n/a = 1 – exp(-bt) y n/b = a t exp(-bt) + Σ tj,

donde el sumatorio es sobre i = 1,…..,n

Para el programa que nos ocupa, no ha sido posible estimar a y b, ya que, aunque la hipótesis que la corrección de un fallo puede provocar otro nuevo, es totalmente valida en el mundo informático, la correlación entre a y b es difícil de probar.

Incluso se han llegado a producir discusiones sobre hasta que punto un fallo provoca otro y en que situaciones se puede achacar un fallo a la existencia de otro fallo.

7.10.1.7 MODELO DE DISTRIBUCIÓN DE WEIBULL

Para poder aplicar correctamente el modelo Weibull necesitaríamos una recogida de muestras de fallos en la ejecución del programa lo suficientemente grande poder mostrar dichos fallos en una distribución, y además, que dicha muestra a su vez se pudiera extrapolar a una distribución Weibull.

Debido a las características del programa, no ha sido posible recoger un conjunto fallos lo suficientemente homogéneo para poder determinar que la muestra pertenece a una distribución de Weibull.

Por este motivo, hemos tomado como suposición que sigue esta distribución, con un parámetro de escala β = 1,5 y un parámetro de escala λ de 18 meses, que es el tiempo en el que se cree que estará preparado para funcionar al 99% de fiabilidad.

El valor de β = 1,5 es debido a que el parámetro escala, cuando toma un valor entre 1 y 2 el riesgo decrece a medida que aumenta el tiempo, y esta si es una característica que se da en nuestro programa.

El MTTF será:

MTTF = (β / λ) Γ (1/ λ) = (1,5/18) Γ (1/18) = 2,86 meses es el MTTF.

La fiabilidad en el tiempo esta dada por:

R(t) = exp [-(t/λ)^ β]

Si quisiéramos por ejemplo la fiabilidad a los 9 meses:

R(9) = exp [-(9/18)^1,5] = 0,7021, sería del 70%.

Si quisiéramos el tiempo que necesitamos para obtener un 90% de fiabilidad:

T= λ (-ln R)^(1/ β) = 18 (-ln 0,9)^(1/1,5) = 4,01 meses.

7.10.1.8 MODELO DE ESTIMACIÓN BAYESIANO DEL RATIO DE FALLO

Para realizar una estimación mediante el modelo Bayesiano, se ha decidido seguir el método 50/95, según las experiencias previas que se han tenido con programas de este tipo.

El consenso al que se ha llegado es que el MTBF50 se estima en unas Consenso sobre un probable MTBF50 valor de 2560 horas y un bajo valor MTBF05 de 1150.

RT es 2560/1150 = 2,22

Usando un software parecido para encontrar la raíz de la función de análisis, los parámetros anteriores gamma fueron:

Parámetro de forma a = 2,563
Parámetro de escala b = 1726,53

λ tendrá una probabilidad del 50% de estar por debajo de 1/2560= 0,0004 y una probabilidad del 95% de estar por debajo de 1/1150 = 0,00087.

Las probabilidades se basan en los 0.0004 y 0.00087 percentiles de una distribución gamma con forma parámetro forma a = 2.563 y parámetro escala b = 1726,53).

8. CONCLUSIONES

Después de haber calculado la fiabilidad de nuestro programa para los diferentes modelos, se ha resumido modelo por modelo los resultados obtenidos.

8.1 RESULTADO MODELOS DE PREDICCION

De forma sinóptica, se ha recogido en una tabla las ventajas e inconvenientes de cada modelo de predicción:

Modelo de	Ventajas	Inconvenientes
Modelo de colección de datos históricos	El mejor método si se tienen suficientes datos históricos.	Necesario personal con experiencia al no tener suficiente información.
Modelo de Musa	Predice correctamente rutinas muy concretas.	Solo para tiempo de ejecución.
Modelo de Putnam	Muy completo y compatible con otros modelos de predicción.	Solo para programas muy grandes (más de 70.000 LOC) y muy costoso.
Modelo TR-92-52	Se puede aplicar en diferentes fases del ciclo de vida del proyecto.	Grandes desviaciones si esta mal calculado y es necesaria extensa información histórica.
Modelo TR-92-15	Posee factores para la estimación del numero de fallos.	Es complicado predecir los factores que nos son necesarios.

Figura 8.1: Resumen ventajas e inconvenientes modelos de estimación

8.2 RESULTADO MODELOS DE ESTIMACION

Al igual que en el apartado anterior, se ha recogido en un cuadro las principales características de cada modelo:

Modelo de Predicción	Ventajas	Inconvenientes
Modelo Exponencial General*	Fácil de usar, los fallos se pueden resolver fácilmente.	Parte de hipótesis que no tienen porqué cumplirse, solo es útil si tiene una tasa de fallos constante.
Modelo Lloyd-Lipow	Relaciona el ratio de fallos con el ratio de fallos pendientes de corregir.	Solo sirve para detectar los fallos.
Modelo Básico de Musa	Relaciona ratio de fallo inicial con el final.	Solo vale para tiempo de ejecución y es muy sensible a desviaciones.
Modelo Logarítmico de Musa	El ratio de fallos decrece exponencialmente, es más verosímil.	Solo para tiempos de ejecución y las hipótesis deben ser validas.
Modelo de Shooman	Se ajusta a cambios de tamaño.	Si las hipótesis no se cumplen provoca desviaciones muy grandes.
Modelo Goel-Okumoto	Los fallos pueden causar otros fallos, se trata de una situación mucho más real.	Muy sensible a desviaciones en las hipótesis.
Modelo de Weibull	Muy flexible, ya que se adapta a diferentes modelos.	Muy costoso en tiempo calcular todos los parámetros.
Modelo Estimación Bayesiano	La estimación de la información previa es subjetiva, depende de las experiencias previas.	La estimación de la información previa es subjetiva, depende de las experiencias previas.

Figura 8.2: Resumen ventajas e inconvenientes modelos de predicción

Muchos libros de texto básicos usan la distribución exponencial cuando están tratando con problemas de fiabilidad. Así mismo, mucho de esos libros a menudo empiezan la resolución de un problema de fiabilidad con la frase “asumiendo que el tiempo de fallo sigue una distribución exponencial”.

El motivo de ello es que la distribución exponencial es muy fácil de usar. Los problemas de fiabilidad puede ser resueltos rápida y fácilmente mientras que otras distribuciones están consideradas demasiado difíciles para los principiantes.

Un tema importante en ingeniería de confiabilidad es la estimación de parámetros cuando hay elementos que han sido probados y no han fallado (datos censurados). Para la distribución exponencial, la estimación de parámetros cuando se encuentran datos censurados es una tarea relativamente sencilla.

Para otras distribuciones, la estimación de parámetros es extremadamente difícil, cuando algunos de los datos son censurados.

Debido a esto, muchos textos describen la estimación de parámetros con todo detalle para la distribución exponencial, pero descuidan presentar técnicas de estimación de parámetros para otras distribuciones.

La distribución exponencial no es muy útil en el modelando datos en el mundo real. La distribución exponencial sólo es útil para los elementos que tienen una tasa de fallo constante.
Esto significa que la muestra de datos no tenga ningún problema tanto en su nacimiento como en su vida útil.

8.3 CONCLUSIONES OBTENIDAS DEL ESTUDIO

En nuestro caso, el modelo que nos ha venido mejor es el Bayesiano, que ha coincidido bastante con los tiempos que se han establecido en el proyecto.

La primera razón de todas es que en el mundo real, el dinero manda, y establecer un modelo de fiabilidad viene a encarecer el precio del producto final, esto provoca que la mayoría de los fabricantes de software evadan este apartado y en su lugar ofrecen un producto más barato, que a pesar de tener fallos suele venir con un mantenimiento asociado para ir subsanando los errores que el cliente pueda encontrar.

Eso si, cada vez se esta dando más importancia a la fiabilidad del software, y actualmente se suele vender de cara al cliente como un añadido a la calidad final del producto, aunque la introducción de estos modelos esta siendo muy lenta, ya que se trata de una rama casi independiente que mucha gente asocia que puede ir en contra de las necesidades de negocio.

Según se van realizando proyectos más complejos, se va dando más importancia a la fiabilidad del software, puesto que a mayor precio de producto, el cliente solicita más garantías al desarrollador.

En este caso, se suelen tener reuniones con el cliente para establecer todo un conjunto de casos de uso que pueden darse en el programa y definir un perfil operacional de las personas que utilizan dicho software.

Si el programa pasa con éxito todos estos casos de uso para los diferentes perfiles operacionales, entonces el cliente está dispuesto a recepcionarlo, y aún así, se establece un periodo de garantía para los diferentes errores que se puedan encontrar, ya que se dan dos premisas que siempre se cumplen.

Es casi imposible definir todos los casos de uso en un programa y si se han definido todos, al cliente siempre se le ocurrirás más a medida que el producto este operativo.

Es casi imposible definir todos los perfiles operacionales del programa, y si se consiguiera, al cliente siempre se le puede ocurrir alguno más en medio de los desarrollos.

Esta dinámica es la que ha permitido que aparezcan metodologías para un desarrollo ágil de software como SCRUM, es decir, es más importante el tiempo y el coste pero cumpliendo con unos requisitos mínimos de fiabilidad, ya que el cliente tiene autentica ansiedad por tener algo que funcione relativamente bien, pero en poco tiempo.

Para aplicaciones como la que estamos viendo, no se suele utilizar ningún modelo de fiabilidad, sino que se suele optar por la dinámica de realizar todos los casos de uso que se pueda para intentar “pulirlo” y asumir una tasa de fallos que sea la menor posible con el compromiso de ir realizando “entregables” con los defectos subsanados.

Sólo cuando nos encontramos con aplicaciones no comerciales y de gran calado, como cadenas de montaje con sistemas de redundancia para no parar la producción o con software relacionado con la seguridad y/o salud humana es cuando la aplicación de estos modelos de fiabilidad se hace obligatoria.

Por ejemplo, el software de una báscula milimétrica para investigar compuestos químicos no puede contener errores, ya que podría poner en riesgo la salud de las personas que compren los futuros medicamentos.

En conclusión, es necesario y cada vez más importante añadir estos procesos de cálculo de fiabilidad del software y poco a poco se va engranando dentro del proceso normal de desarrollo de software como un ciclo más.

No se busca solo cumplir con los Requisitos Técnicos Formales de un proyecto, sino que se intenta dar un valor añadido al producto software a través de estas técnicas.

9. BIBLIOGRAFÍA Y REFERENCIAS

[1] Littlewood, B.: "Forecasting Software Reliability ". CSR Technical Report, February 1989. Centre fox Software Reliability, The City University, Northampton Square, London ECl VOHB.

[2] Neufelder, A.: Ensuring Software Reliability. Marcel Dekker, Inc.New York, 1993.

[3] Mario G. Piattini, Jose A. Calvo-Manzano, Joaquin Cervera Bravo y Luis Fernandez Sanz. “Análisis y diseño de aplicaciones informáticas de gestión, una perspectiva de ingeniería del software”, paginas 419-469. Ed. Alfaomega, 2004

[4] IEE90 IEEE. IEEE Standard Glossary of Software Engineering Terminology, 1990

[5] William E. Perry.Effective methods for software testing, pages 1-19. Wiley, second edition, 2000

[6] Musa, J.D., Iannino. A., Okumoto, K.: Software Reliability, Measurement, Prediction, Application. McGraw-Hill Book, New York, 1987.

[7] Pressman, Roger S. "Ingeniería del Software: Un Enfoque Práctico". 4ª Edición. McGraw-Hill. 1998.

[8] McCall, J. (1977). Factors in Software Quality : General Electric.

[9] Rook, P.: Software Reliability Handbook. Centre for Software Reliability, City University, London, U.K. 1990.

[10] McDermid, Donald C. “Software Engineering for Information Systems”. Editorial: Alfred Waller Ltd, 1990

[11] Ejiogo, Lem. “Software Engineering with Formal Metrics”

[12] John A. McDermid “Software Engineering Environments: Automated Support for Software Engineering”. Ed. McGraw Hill

[13] Anneliese Von Mayrhauser Software Engineering: Methods and Management. Ed. Academic Press 1991

[14] Pressman, Roger S. “A Manager´s Guide to Software Engineering” Ed. Mc Graw-Hill

[15] Norman E. Fenton. “Software Reliability and Metrics” Ed. Kluwer Academic Publishers; Edición: 1991. (1 de julio de 1991)

[16] Deparment of Defense. Military Handbook Electronic Reliability Design Handbook (MIL-HDBK-338B)

[17] Putnam, Lawrence H., and Myers, Ware, "Measures for Excellence: Reliable Software on Time, Within Budget", Prentice-Hill

[18] Mc Call, J.A., W. Randell, and J. Dunham, "Software Reliability, Measurement, and Testing," Rome Laboratory, RL-TR-92-52

REFERENCIAS

Paul A. –Tobias, David C. Trindade. “Applied Reliability”. Third Edition. CRC Group 2012

Reliability Engineering Handbook

Ronald E. Walpole, Raymond H. Myers, Sharon L. Myers “Probabilidad y Estadística para Ingenieros” Sexta Edición. Prentice Hall

http://www.itl.nist.gov/div898/handbook/ (Engineering Statistics Handbook)

http://ewh.ieee.org/r1/boston/rl/home.html (IEEE Boston Reliability Chapter)

Kitchenham,B.; Towards a Constructive Quality Model, Software Engineering Journal, Vol.2, N. 4, pp. 105-113, 1987.

Murine, G.E. , Integrating software quality metrics with software QA, Quality Progress vol.21, no.11; pp. 38-43; Nov. 1988.

"Automatición and Technological Change", Heariogs, Subcommittee on Economic stabilitation of the joint committee on the Economic Report, Congress of the United States, Octubre, 1955, Págs. 53-54.

A Re-examination of the Reliability of Unix Utilities and Services por Barton P. Miller <bart@cs.wisc.edu>, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan y Jeff Steidl

10. ANEXOS

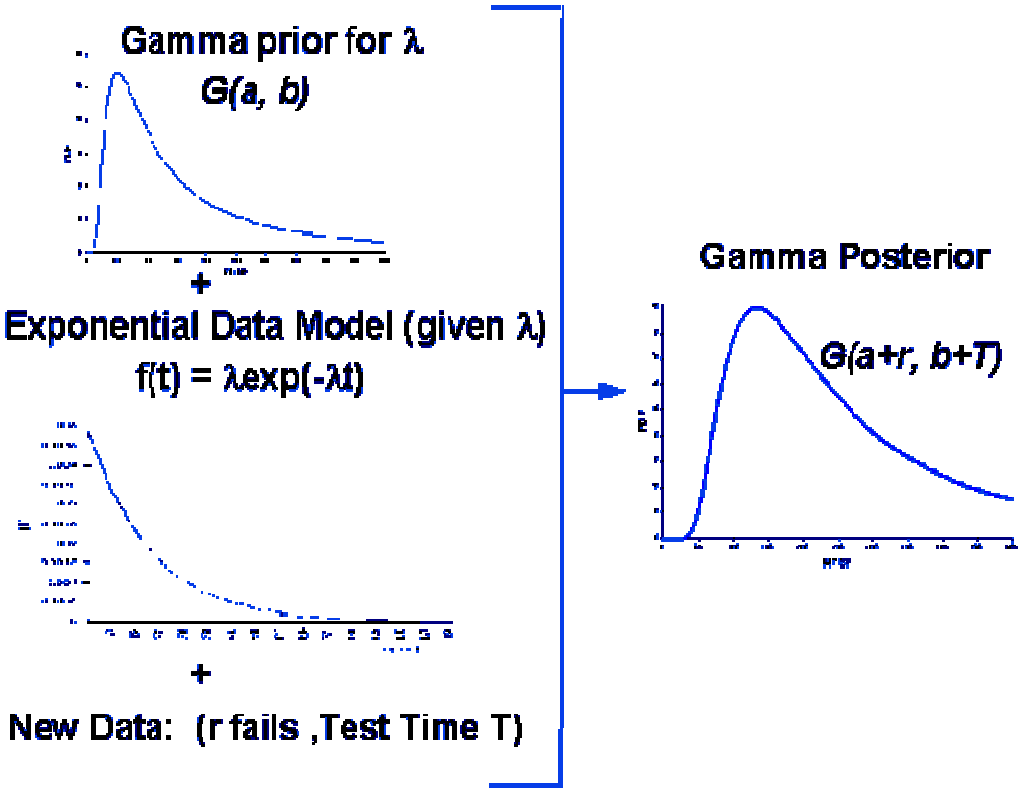
ANEXO 1 EL MODELO GAMMA

El paradigma Bayesiano que se introdujo en el capitulo 6 describía los supuestos que subyacen sobre el modelo gamma/exponencial, incluyendo varios métodos para transformar los datos previos y los juicios subjetivos en parámetros previos gamma “a” y “b”.

Posteriormente, vimos como utilizar este modelo para calcular el tiempo de prueba requerido para confirmar el MTBF en un nivel de confianza dado.

El objetivo de los procedimientos de fiabilidad Bayesianos es obtener una distribución posterior tan precisa como sea posible y luego usar esta distribución para calcular la tasa de tiempo medio entre fallos (o MTBF) estimado con intervalos de confianza (llamados intervalos de credibilidad bayesianos).

La figura siguiente resume los pasos de este proceso:



Una vez que las pruebas ha sido ejecutados y los r fallos recogidos, los parámetros de gamma posterior son:

$a' = a + r,\ b' = b + T$

y una estimación (mediana) para el MTBF se calcula como:

$1 / G^{-1}(0.5, a', (1/b'))$

donde G (q, β, γ) representa la distribución gamma con parámetro de forma γ y parámetro de escala β.

Nota: cuando a = 1, Gamma se reduce a una distribución exponencial con b= λ

Algunas personas prefieren utilizar la media reciproca de la distribución posterior como su estimación para el MTBF. Su significado es el estimador de error mínimo cuadrático (MSE) de λ, pero usando el recíproco de la media para calcular el MTBF siempre será más conservador que un estimador del 50%.

Una banda menor del 80% para el MTBF es obtenida de:

$1 / G^{-1}(0.8, a', (1/b'))$

Y en general, una banda menor al 100(1-α) % está dado por

$1 / G^{-1}((1-\alpha), a', (1/b'))$.
Los dos intervalos de credibilidad 100(1-α) % para el MTBF son:

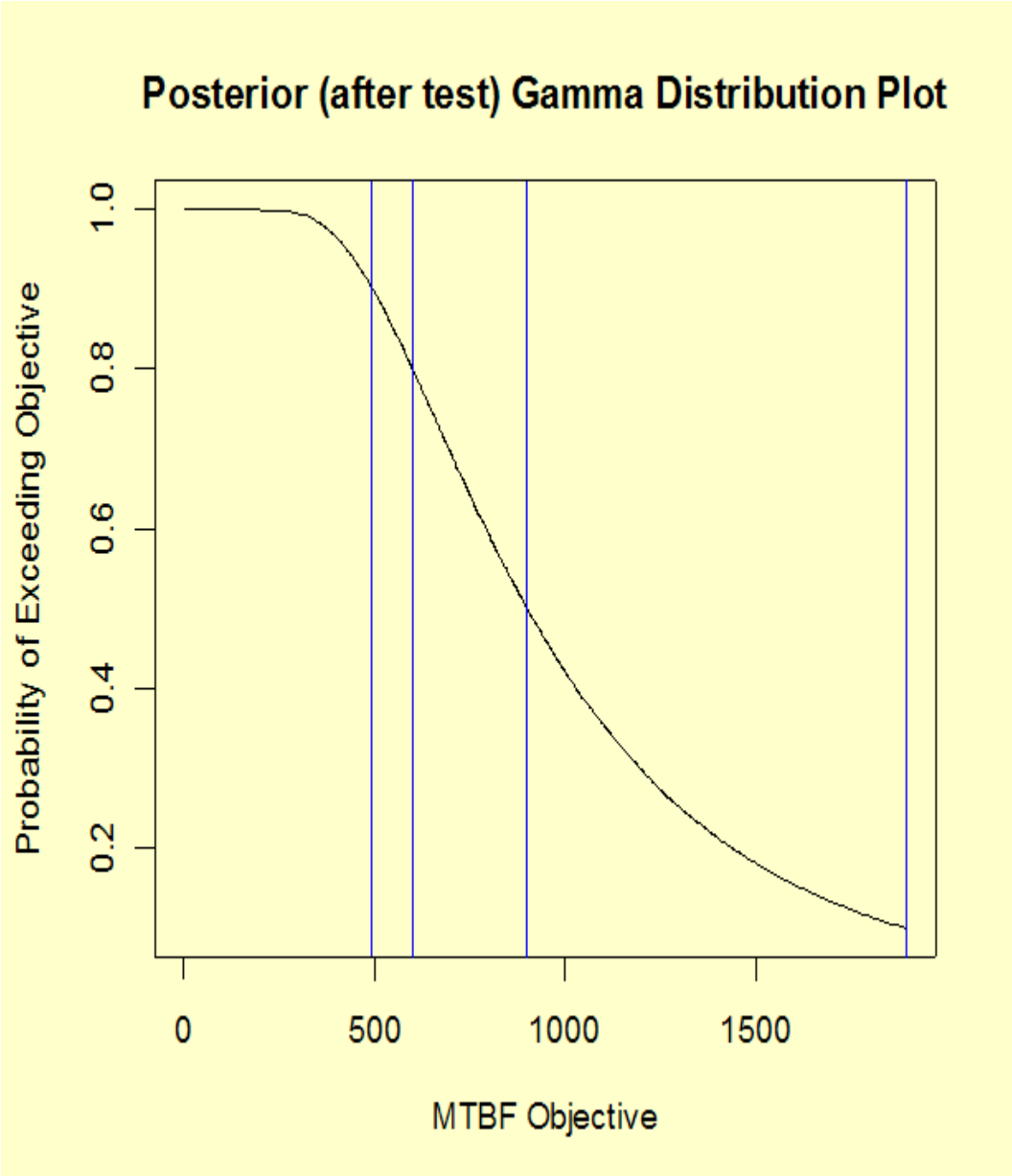
$[1 / G^{-1}((1-\alpha/2), a', (1/b')), 1 / G^{-1}((\alpha/2), a', (1/b'))]$.

Finalmente el $G((1/M), a', (1/b'))$ calcula la probabilidad que el MTBF sea mayor que M.

EJEMPLO

Un sistema ha completado una prueba de fiabilidad dirigido a confirmar 600 horas de MTBF con un nivel de confianza del 80%. Antes de la prueba, se acordó un gamma previo de a = 2, b = 1400 en base a las pruebas en las instalaciones del proveedor. Los cálculos de planificación del Test Bayesiano permiten un máximo de 2 nuevos fallos, para una prueba de 1909 horas. Cuando dicho test se llevó a cabo, hubo exactamente dos fallos. ¿Qué se puede decir sobre el sistema?

La posterior gamma CDF tiene parámetros a '= 4 y b' = 3309. El diagrama siguiente muestra los valores CDF en el eje Y, trazado frente a 1 / λ = MTBF, en el eje X. Al ir desde la probabilidad, en el eje Y,a lo largo de la curva y bajando hasta el MTBF, podemos estimar cualquier punto percentil del MTBF.



Los valores del Tiempo Medio Entre Fallos (MTBF) son mostrados en el siguiente cuadro:

$1 / G^{-1}(0.9, 4, (1/3309))$	= 495 horas
$1 / G^{-1}(0.8, 4, (1/3309))$	= 600 horas (esperado)
$1 / G^{-1}(0.5, 4, (1/3309))$	= 901 horas
$1 / G^{-1}(0.1, 4, (1/3309))$	= 1897 horas

La prueba ha confirmado un MTBF de 600 horas para un 80% de confianza, un MTBF de 495 horas para un 90% de confianza y (495, 1897) es un intervalo de credibilidad 90% para el MTBF. Un solo número (punto) estimado para el sistema MTBF serían 901 horas. Alternativamente, es posible que desee utilizar el recíproco de la media de la distribución posterior (b'/a') = 3309/4 = 827 horas como una sola estimación. La media recíproca es más conservadora, en este caso se trata de un 57% de límite inferior (G ((4/3309), 4, (1/3309))).