Trabajo Fin de Grado
Grado en Ingeniería de Organización Industrial

Estudio sobre los principales modelos de fiabilidad del software

Autor: Jesús Benítez Rodríguez
Tutor: Ester Gutiérrez Moya

Dep. Organización Industrial y Gestión de Empresas I
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Trabajo Fin de Grado
Grado en Ingeniería de Organización Industrial

Estudio sobre los principales modelos de fiabilidad del software

Autor:
Jesús Benítez Rodríguez

Tutor:
Ester Gutiérrez Moya

Dep. Organización Industrial y Gestión de Empresas I
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Trabajo Fin de Grado: Estudio sobre los principales modelos de fiabilidad del software

Autor: Jesús Benítez Rodríguez

Tutor: Ester Gutiérrez Moya

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

A mi familia

Agradecimientos

Este trabajo que aquí presento es la culminación de cinco largos años de esfuerzo y trabajo duro, que han traido consigo un gran aprendizaje y desarrollo personal.

Sería injusto no agradecer a todos los que me han animado a terminarlo, pues sin su empuje no habría encontrado la motivación para hacerlo. Pero por encima de todo, sería injusto no reconocer el mérito de quienes tanto se han esforzado para que llegue hasta aquí.

Así pues, gracias amigos y padres de amigos, y gracias a mis padres.

Por último, me gustaría dar gracias a la profesora Ester Gutiérrez Moya por su tiempo y dedicación durante todo este trabajo.

Jesús Benítez Rodríguez

Sevilla, 2016

Resumen

El presente estudio constituye un análisis de las técnicas existentes para la modelización de la fiabilidad de software, aplicables durante la etapa de prueba de dichos sistemas con el fin de evaluar la evolución de su fiabilidad, y poder determinar el momento óptimo para la finalización de dicha etapa.

Se han examinado las hipótesis y los fundamentos matemáticos de los modelos más característicos hoy en uso, estableciéndose un esquema para su clasificación sistemática, que permite una rápida y sencilla elección del modelo más apropiado a cada caso en base a las hipótesis básicas sobre las que se formulan los distintos modelos.

Finalmente se desarrollan dos casos prácticos, donde se aplican algunos de los modelos introducidos a dos baterías de datos obtenidos en proyectos reales. El análisis se lleva a cabo mediante una aplicación informática estadística, R.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

Según la Real Academia Española, el término *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, las aplicaciones informáticas están totalmente integradas en la sociedad en la que vivimos. A todos nos extrañaría ver una fábrica sin máquinas ni procesos automatizados, una empresa, independientemente de su dimensión, sin una gestión informatizada, un sistema de control de vuelos manual o incluso una tienda sin un sistema de gestión de inventario.

La informática, incluida en el término tecnologías de la información, ha evolucionado a un ritmo vertiginoso hasta resultar imprescindible en nuestras vidas, siendo un elemento altamente potente, llegando a ser peligroso si no está bajo control. Este hecho requiere gran cuidado por parte de los informáticos, no siendo ya suficiente realizar un sistema eficiente, rápido y sencillo para el usuario, sino que tiene que ser un sistema a prueba de fallos.

La industria del software está entrando en un periodo de madurez, al mismo tiempo que el software se esta convirtiendo en un componente esencial de muchos de los productos actuales.

El propósito de este proyecto es examinar los métodos existentes para medir la fiabilidad de un software y ponerlos en práctica a partir de un conjunto de datos obtenidos a partir de bibliografía científica.

Por lo anteriormente expuesto, se va a realizar un análisis en lo referente a fiabilidad de 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 del software.

Los modelos tratados permitirán evaluar la evolución o estado de dicha fiabilidad, además de poder predecir de esta forma los posibles problemas, fallos o errores que puedan producirse a lo largo de la vida útil de dicho software.

En un segundo apartado se expondrán las métricas del software, qué miden, cuál es su propósito y cuáles son aplicadas exclusivamente a la fiabilidad.

Este estudio utilizará como herramienta la librería de `R`, empleando el paquete `"Reliability"`.

1.1. Objetivos

En un primer acercamiento se definirá el concepto de fiabildiad, así como el objeto de estudio de este trabajo, el software. Es necesario introducir el proceso de desarrollo del mismo, prestando atención a sus diferentes etapas, pues según el enfoque de análisis que se quiera seguir, cobrará más importancia una u otra etapa.

El principal objetivo de este trabajo, la aplciación práctica de los modelos teóricos estudiados, se desarrollará tras la introducción de algunos de los modelos de fiabildiad del software existentes. Esta aplicación práctica incluirá un análisis de los resultados obtenidos al ejecutar una serie de funciones en la herramienta estadística `R`.

2 INGENIERÍA DEL SOFTWARE

Un software se desarrolla con éxito cuando satisface las necesidades de las personas que lo utilizan; cuando funciona de forma impecable durante mucho tiempo; cuando es fácil de modificar o fácil de utilizar. Cuando estas condiciones no se dan, se considera que el software ha fallado y pueden ocurrir, y de hecho ocurren, verdaderos desastres.

Todos queremos desarrollar un software que haga bien las cosas, evitando que esas cosas malas aparezcan. Muchas veces, seguir una determinada metodología ayuda a tener éxito al diseñar y construir una aplicación informática.

2.1. Software

Antes de pasar a definir qué es la fiabilidad del software, es importante entender bien el concepto de software. Existen varias definiciones similares aceptadas para software, pero probablemente la más formal sea la que se encuentra en el estándar 729 del IEEE:

Es el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados, que forman parte de las operaciones de un sistema de computación.

Considerando esta definición, el concepto de software incluye no solo los programas o instrucciones de computación en sus distintos estados: código fuente, binario o ejecutable; sino que también recoge su documentación, los datos a procesar e incluso la información de usuario. Así, el software estaría compuesto por programas o instrucciones, datos y documentación.

2.1.1. Características del Software

El software es, esencialmente, un elemento lógico y se diferencia del hardware, un elemento físico, en sus características. Una clara diferencia con este es que el software se desarrolla, no se fabrica en

sentido clásico. Aunque existen similitudes entre el desarrollo de una aplicación informática y la construcción de un hardware, ambas actividades son fundamentalmente distintas.

Cada producto software es diferente porque se construye para cumplir los requisitos únicos de un cliente. Cada software necesita, por lo tanto, ser construido siguiendo una metodología formal. Esto implica entender qué es necesario, diseñar el producto para que cumpla los requisitos, implementar el diseño usando un lenguaje de programación y comprobar que el producto cumple con los requisitos. Todas estas actividades se llevan a cabo mediante la ejecución de un proyecto y requiere un equipo trabajando de una forma coordinada.

Otra característica del software es que se deteriora. Los defectos no detectados harán que falle el programa durante las primeras etapas de su vida. Una vez que se corrigen estos defectos, los fallos disminuyen. Sin embargo, conforme se realizan modificaciones en el programa, es probable que se introduzcan nuevos defectos, lo que hace que el software vaya deteriorándose a lo largo del tiempo debido a estos cambios.

2.1.2. Tipos de software

A fines prácticos se puede clasificar el software en tres grandes categorías:

- Software de sistema. Su objetivo es desvincular al usuario de los detalles del sistema informático que se esté usando, aislándolo del procesamiento interno del equipo. El software de sistema facilita también herramientas y utilidades para el mantenimiento global del sistema. Algunos ejemplos de este tipo de software son los sistemas operativos, herramientas de diagnóstico o controladores de dispositivos.
- Software de programación. Es el conjunto de herramientas que permiten desarrollar programas informáticos, usando diferentes lenguajes de programación. Aquí se incluyen los editores de texto, compiladores o depuradores.
- Software de aplicación. Se usan para llevar a cabo tareas específicas, en actividades que pueden ser automatizadas, como los negocios. Entre este tipo de software se encuentra el software empresarial, aplicaciones ofimáticas, bases de datos o videojuegos.

2.2. Ingeniería del Software

El término ingeniería del software apareció por primera vez en la conferencia de ingeniería de software de la OTAN en 1968 y fue mencionado para probar el pensamiento sobre la crisis de

software del momento. Desde entonces, ha continuado como una profesión y campo de estudio dedicado a la creación de software de alta calidad, barato, con capacidad de mantenimiento y rápido de construir. Debido a que el campo es todavía relativamente reciente comparado con otros campos de la ingeniería, hay aún mucho trabajo y debate sobre qué es realmente la ingeniería de software, y si merece el título de ingeniería.

El IEEE describe la ingeniería del software como:

La aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación (funcionamiento) y mantenimiento del software.

El objetivo primario de la ingeniería del software es construir un producto informático de alta calidad de una manera oportuna.

La ingeniería implica un conjunto de principios fundamentales que deberían seguirse siempre. Incluye también aquellas actividades explícitas para el entendimiento del problema y la comunicación con el cliente, métodos definidos para representar un diseño, mejores prácticas para la implementación de la solución y estrategias y tácticas sólidas para las pruebas. Si se siguen los principios básicos, esto resulta en productos de alta calidad, que es lo que nos se analizará más adelante.

Para conseguir dicho objetivo, dentro de la ingeniería del software se emplean una serie de prácticas para:

- Entender el problema
- Diseñar una solución
- Implementar la solución correctamente
- Probar la solución
- Gestionar las actividades anteriores para conseguir alta calidad

La ingeniería del software representa pues, un proceso formal que incorpora una serie de métodos bien definidos para el análisis, diseño, implementación y pruebas de software y sistemas. Además, abarca una amplia colección de métodos y técnicas de gestión de proyectos para el aseguramiento de la calidad y la gestión de la configuración del software.

2.2.1. Etapas

La ingeniería de software requiere llevar a cabo numerosas tareas, dentro de las siguientes etapas:

2.2.1.1. Análisis de requisitos

Extraer los requisitos de un producto software es la primera etapa para crearlo. Mientras que los clientes piensan que ellos saben lo que el software tiene que hacer, se requiere habilidad y experiencia en la ingeniería de software para reconocer requisitos incompletos, ambiguos o contradictorios. El resultado del análisis de requisitos con el cliente se plasma en el documento Especificación de Requisitos. Asimismo, se define un diagrama de entidad/relación en el que se plasman las principales entidades que participaran en el desarrollo de software.

La captura, análisis y especificación de requisitos, es una parte crucial; de esta etapa depende en gran medida el logro de los objetivos finales. Se han ideado modelos y diversos procesos de trabajo para estos fines.

2.2.1.2. Especificación

Es la tarea de escribir detalladamente el software a ser desarrollado, en una forma matemáticamente rigurosa. En la realidad, la mayoría de las buenas especificaciones han sido escritas para entender y afinar aplicaciones que ya estaban desarrolladas. Las especificaciones son más importantes para las interfaces externas, que deben permanecer estables.

2.2.1.3. Diseño y arquitectura

Se refiere a determinar cómo funcionará el software de forma general sin entrar en detalles. Consisten en incorporar consideraciones de la implementación tecnológica, como el hardware, la red, etc. Se definen los casos de uso para cubrir las funciones que realizará el sistema, y se transformarán las entidades definidas en el análisis de requisitos en clases de diseño, obteniendo un modelo cercano a la programación orientada a objetos.

2.2.1.4. Programación

Reducir un diseño a código puede ser la parte más obvia del trabajo de ingeniería del software, pero no necesariamente es la que demanda mayor trabajo ni la más complicada. La complejidad y la duración de esta etapa está íntimamente relacionada al o a los lenguajes de programación utilizados, así como al diseño previamente realizado.

2.2.1.5. Prueba

Consiste en comprobar que el software realice correctamente las tareas indicadas en la especificación del problema. Una técnica de prueba es probar por separado cada módulo del software y luego probarlo de forma integral, para así poder llegar al objetivo. Se considera una buena práctica que las pruebas sean efectuadas por alguien distinto a quién las desarrolló.

2.2.1.6. Mantenimiento

Mantener y mejorar el software para solventar errores descubiertos y tratar con nuevos requisitos. El mantenimiento puede ser de cuatro tipos: perfectivo (mejorar la calidad interna de los sistemas), evolutivo (incorporaciones, modificaciones y eliminaciones necesarias para cubrir la expansión o cambio en las necesidades del usuario), adaptativo (modificaciones que afectan a los entornos en los que el sistema opera, por ejemplo, cambios de configuración del hardware o gestores de bases de datos) y correctivo (corrección de errores).

3 FIABILIDAD DEL SOFTWARE

Los conceptos de Calidad, Seguridad y Fiabilidad son aspectos de lo que genéricamente se denomina "Calidad" pero no deberían confundirse. Para el usuario de un bien, los tres términos citados son importantes, siendo lo que se ha llamado calidad el grado de aprecio o idoneidad que el usuario tiene por el bien. El usuario, consumidor o cliente, se encuentra en un estado privilegiado ya que elegirá aquel producto o servicio que mejor satisfaga sus necesidades. El usuario optará por aquel que, dentro de un rango de precios, resulte más idóneo para sus necesidades. Sin embargo, desde el punto de vista de la empresa ese concepto de calidad resulta insuficiente, ya que será necesario algo más tangible para poder desarrollarla. En la figura 3.1 se refleja la relación existente entre el usuario y el productor de un bien o servicio.

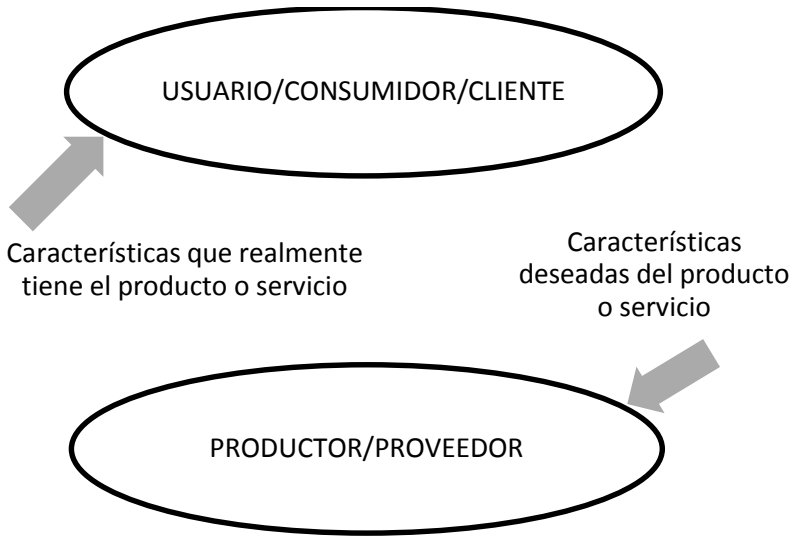

Figura 3.1. Relación entre usuarios y productores

La calidad se ha considerado tradicionalmente desde el punto de vista técnico, es decir, referido a la fabricación de un bien material, aunque actualmente esta interpretación de la calidad ha evolucionado. De forma general, en sentido amplio, por "calidad" de un bien o servicio se entiende actualmente el grado de aprecio que el usuario tiene por el mismo. Respecto a la Seguridad de un

producto, en primer lugar hay que tener en cuenta que se debe garantizar la seguridad ante todo tipo de daño que puedan sufrir las personas en todas las fases de la vida del producto, desde que se fabrica hasta que el usuario lo utiliza, aunque para este usuario el concepto de seguridad está relacionado, muchas veces con el uso particular que él le da al producto, no con el proceso de fabricación que se ha seguido hasta llegar a manos del usuario.

Por último, el usuario incluye la idoneidad del diseño como otro componente de la "calidad". El concepto de fiabilidad suele ser percibido por el usuario como garantía de funcionamiento correcto del producto durante su utilización, en unas condiciones determinadas, a lo largo de un período especificado. La fiabilidad de un producto va unida de forma inseparable a su diseño, ya que durante la fase de diseño es cuando se ponen de manifiesto todos los requisitos que debe cumplir el producto para satisfacer las necesidades del cliente, tanto en rendimiento, prestaciones y durabilidad como en seguridad. Es importante señalar que la fase de diseño de un producto no solo considera su fiabilidad, sino otras muchas características del mismo. La fiabilidad y calidad no son conceptos sinónimos. La calidad de un producto se centra más en su grado de concordancia con las descripciones técnicas y comerciales que lo definen, abarcando tales especificaciones técnicas tanto los requisitos del pedido como la documentación completa de fabricación.

3.1. Fiabilidad de sistemas

Un sistema se define como un conjunto de elementos independientes que interaccionan entre sí para realizar una tarea común. La fiabilidad de un sistema se estudia para poder determinar si el sistema va a realizar su tarea cuando así se requiere, o por el contrario va a fallar.

La evaluación de la fiabilidad de un sistema lógico o software se realiza mediante distribuciones de probabilidad, lo que se explica intuitivamente al considerar como fallan elementos de un mismo tipo, que han sido fabricados, y que trabajan en las mismas condiciones. El tiempo de funcionamiento correcto es específico para cada uno, es decir, que todos los elementos no fallan después del mismo tiempo de operación. Si se registran los tiempos hasta el fallo de cada uno de los elementos observados se obtendrá la distribución de probabilidad de fallo de este tipo de elementos cuando se fabrican y operan en las condiciones definidas. La distribución de probabilidad de fallo de otro grupo de elementos del mismo tipo pero fabricado según otro proceso, o que trabaja en otras condiciones posiblemente será distinta a la anterior.

Según se describe en el párrafo anterior, la ocurrencia de fallos en un grupo de elementos de un mismo tipo e iguales características de fabricación y operación es aleatoria en el tiempo. En consecuencia, el tiempo hasta el fallo T es una variable aleatoria.

La probabilidad de que un elemento falle en el tiempo t viene dada por:

$$P[T \leq t] = F(t) = \int_0^t f(x)dx \quad para\ t \geq 0$$

Siendo $F(t)$ la función de distribución de probabilidad de fallo correspondiente al grupo de elementos en estudio, y $f(t)$ la función de densidad de probabilidad de fallo, en el supuesto que $F(t)$ sea diferenciable.

La función de probabilidad de fallo, $F(t)$ se define como:

Función de distribución de la variable aleatoria T, definida como la probabilidad de fallo de un producto en condiciones preestablecidas, en el intervalo de tiempo t, dado que se encontraba en condición operativa en el instante inicial.

La definición para la función densidad de fallos es:

Probabilidad no condicionada de fallo de un producto en un tiempo unitario contado a partir de t, dado que se encontraba en condición operativa en el instante inicial, verificándose que:

$$f(t)dt = P[t \leq T \leq t + dt] \qquad para\ t \geq 0$$

La probabilidad de que el elemento funcione correctamente hasta el tiempo t es:

$$R(t) = P[T \geq t] = 1 - F(t) = \int_t^\infty f(t)dt$$

Siendo $R(t)$ la función que representa la fiabilidad del elemento. Por tanto, la función de fiabilidad de un elemento no reparable se define como:

La probabilidad de que realice su función sin fallo en un tiempo dado y en unas condiciones especificadas.

Es decir, la fiabilidad es la probabilidad de supervivencia del elemento.

Las expresiones anteriores indican que la función de distribución de probabilidad de fallo y la fiabilidad son complementarias. Por tanto, y dado que la función densidad de probabilidad de fallo es la derivada de la función de distribución de probabilidad de fallo, se tienen las siguientes relaciones

$$f(t) = \frac{dF(t)}{dt} = -\frac{dR(t)}{dt}$$

y

$$R(t) = 1 - \int_0^t f(t)dt$$

Si se representa una función de densidad de probabilidad de fallo hipotética, el área comprendida entre la función y los ejes de coordenadas debe ser igual a uno, en consecuencia dicho área representa la suma de las dos funciones complementarias. Si se quiere calcular sus valores concretos para un instante de tiempo determinado, bastará trazar una paralela al eje de ordenadas en dicho instante, de forma que el área que queda a la izquierda de la línea representa la probabilidad de que el elemento falle durante ese tiempo, y el área que queda en la parte derecha representa la probabilidad de supervivencia del elemento durante ese tiempo, es decir, es la fiabilidad del elemento.

Por tanto, cuanto mayor sea el tiempo de operación del elemento, lo que equivale a decir que cuanto más alejado esté ese instante del origen de coordenadas, la probabilidad de que el elemento falle aumenta y su fiabilidad disminuye.

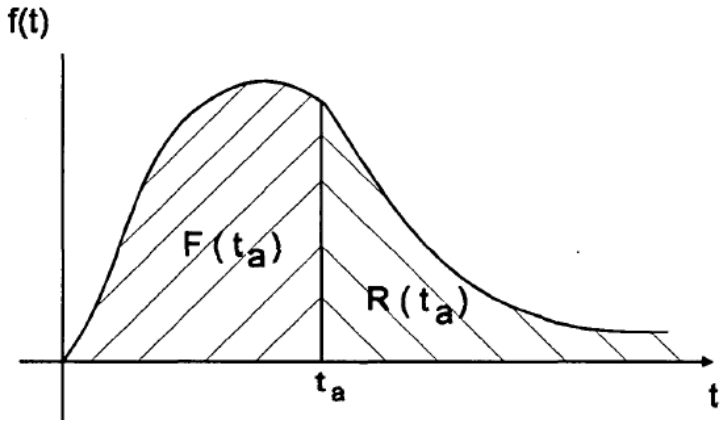

Figura 3.2. Función de densidad de la probabilidad de fallo

Otra función muy importante en un estudio de fiabilidad es la tasa de fallos, o medida de la velocidad de ocurrencia de fallos.

La tasa de fallos $h(t)$ se define según:

Probabilidad condicionada de fallo de un producto en un tiempo unitario contado a partir de t, dado que el producto se encontraba en condición operativa en los instantes t e inicial.

$$h(t)dt = \Pr(t \leq T \leq t + dt) = \frac{\Pr(t \leq T \leq t + dt)}{\Pr(T \geq t)} = \frac{f(t)dt}{R(t)}$$

Lo que permite definir la tasa de fallos de un producto con la expresión

$$h(t) = \frac{f(t)}{R(t)}$$

No se trata de un calculo directo del numero de fallos ocurridos en un grupo de elementos idénticos durante un cierto tiempo t_a, ya que si la muestra observada es de 100 elementos, el número de fallos que ocurren durante ese tiempo será menor que el numero de fallos que ocurren durante el mismo

tiempo t_a cuando la muestra es de 1000 elementos. Sin embargo, al tratarse de muestras de elementos idénticos, la tasa de fallos debe ser la misma. Por tanto, la tasa de fallos depende del número de fallos que ocurren en un tiempo dado y del número de elementos con posibilidad de fallo.

Teniendo en cuenta la definición dada para la tasa de fallos, se ve que para el instante t=0, la tasa de fallos coincide con la función de densidad de fallos, ya que en dicho instante los elementos supervivientes son todos los de la muestra, pero en instantes posteriores, los elementos con posibilidad de fallo son los que siguen funcionando, es decir, los que han sobrevivido durante el tiempo t, por lo tanto la tasa de fallos para un instante t cualquiera será

$$h(t) = \frac{1}{N_s(t)} \frac{\Delta N_f(t)}{\Delta t}$$

donde

$$N_s(t) - \ número\ de\ elementos\ que\ sobreviven\ despues\ del\ tiempo\ t$$

$$N_f(t) - \ número\ de\ elementos\ que\ han\ fallado\ durante\ el\ tiempo\ t$$

siendo

$$N_s(t) + N_f(t) = N_0$$

Desarrollando la expresión de la tasa de fallos se llega a la vista inicialmente:

$$h(t) = \frac{1}{N_s(t)} \frac{\Delta N_f(t)}{\Delta t} \frac{N_0}{N_0} = \frac{N_0}{N_s(t)} \frac{1}{N_0} \frac{\Delta N_f(t)}{\Delta t} = \frac{f(t)}{R(t)}$$

Es decir, que la tasa de fallos esta directamente relacionada con la función de densidad de probabilidad de fallo, y con la fiabilidad.

Esta relación confirma la coincidencia de la tasa de fallos y la función de densidad cuando $t = 0$, ya que en dicho instante la fiabilidad es $R(0) = 1$.

Teniendo en cuenta la definición de la tasa de fallos y la de la función de densidad de fallos

$$f(t) = \frac{dF(t)}{dt} = -\frac{dR(t)}{dt}$$

$$R(t) = 1 - \int_0^t f(t)dt$$

se tiene que

$$h(t) = \frac{-\frac{dR(t)}{dt}}{R(t)}$$

Por tanto se obtiene la siguiente expresión de la fiabilidad en función únicamente de la tasa de fallos:

$$R(t) = exp\left[-\int_0^t h(t)dt\right]$$

En el caso especial en que la tasa de fallos es constante e independiente del tiempo, esto es $h(t) = \lambda$, la ecuación se simplifica:

$$R(t) = e^{-\lambda t}$$

La esperanza de vida de un producto, o tiempo medio hasta el fallo en los sistemas no reparables (MTTF – Mean Time To Failure); o tiempo medio entre fallos en los sistemas reparables referido al tiempo de funcionamiento (MTBF – Mean Time Between Failure), se define como el valor esperado de la variable aleatoria T, es decir:

$$MTTF = E(T) = \int_0^\infty tf(t)dt = -R(t)t|_0^\infty + \int_0^\infty R(t)$$

y dado que

$$\lim_{t\to\infty} R(t)t\big|_0^\infty \to 0$$

se tiene la siguiente expresión para el tiempo medio hasta el fallo:

$$MTTF = \int_0^\infty R(t)\,dt$$

Resumiendo, las funciones que describen la fiabilidad de un elemento son, según se ha indicado, las siguientes:

- Función de distribución de probabilidad de fallo – $F(t)$
- Función de densidad de probabilidad de fallo – $f(t)$
- Función de fiabilidad – $R(t)$
- Tasa de fallos – $h(t)$

Estas cuatro funciones están relacionadas entre sí, de forma que basta la definición de una de ellas para que las otras tres queden determinadas mediante la aplicación de las formulas anteriores.

La evaluación de la fiabilidad de los elementos que componen un sistema es el primer paso para el estudio de la fiabilidad del sistema. El procedimiento que se sigue habitualmente es descomponer el sistema en los elementos que lo constituyen, definir la fiabilidad de cada uno de los elementos y, por último, combinar las fiabilidades de los componentes, según técnicas especificas, con el fin de calcular la fiabilidad de la totalidad del sistema. El nivel de descomposición necesario es aquel que

permite definir la fiabilidad individual de todos los elementos en los que se ha descompuesto el sistema con una cierta precisión, por lo que puede ser suficiente descomponer el sistema en un grupo de subsistemas, sin tener que llegar a un alto nivel de detalle.

Normalmente un sistema se representa como una red de elementos conectados entre sí en serie, en paralelo, en serie-paralelo, o según una topología de red más compleja que estas tres, si bien es cierto, esta complejidad puede ser simplificada hasta obtener una configuración serie-paralelo.

Los elementos que componen un sistema se identifican de acuerdo a la función especifica que realizan, siendo la representación del sistema un diagrama lógico del mismo que indica la interacción existente entre sus distintos elementos constituyentes. El diagrama de bloques de fiabilidad y el diagrama de árboles de fallo son los que se utilizan más frecuentemente.

Un sistema tiene una configuración en serie cuando todos sus componentes deben realizar su función correctamente para que así lo haga el sistema, El fallo de un único

componente causa el fallo del sistema entero, por lo que la fiabilidad del sistema en serie vendrá dada por la combinación de la fiabilidad de todos sus elementos, esto es

$$R_s(t) = \prod_{i=1}^{n} exp\left[-\int_0^t h_i(t)dt\right]$$

Un sistema se dice que es un sistema en paralelo cuando basta que uno de sus componentes realice correctamente su función para que así lo haga el sistema. Por tanto, todos los componentes deben fallar para que falle el sistema en paralelo. La fiabilidad de dicho sistema será

$$R_s(t) = 1 - \prod_{i=1}^{n}\left(1 - exp\left[-\int_0^t h_i(t)dt\right]\right)$$

Según las definiciones de las dos configuraciones básicas, se puede considerar que un sistema serie es un sistema no redundante, mientras que un sistema paralelo es un sistema redundante.

La fiabilidad de un sistema serie-paralelo, compuesto por n subsistemas serie, a su vez formados por $m(i)$) componentes en paralelo será:

$$R_s(t) = \prod_{i=1}^{n}\left\{1 - \prod_{i=1}^{m(i)}\left(1 - exp\left[-\int_0^t h_i(t)dt\right]\right)\right\}$$

3.2. Aspectos característicos del software

Los sistemas de información son cada vez más importantes en la sociedad moderna. Se podría decir que prácticamente la totalidad de las actividades diarias están controladas, o necesitan el apoyo de un sistema software.

A medida que las funciones de este tipo de sistemas son más esenciales y complejas, su fiabilidad es más crítica e importante, por lo que cada vez se dedican más recursos para conseguir sistemas de información con un grado de fiabilidad elevado.

Todo sistema de información se descompone, en un primer paso, en dos bloques funcionales: el sistema físico (hardware) y el sistema lógico (software). Ambos bloques deben funcionar correctamente para que así lo haga la totalidad del sistema, por lo que su fiabilidad será el producto de la fiabilidad de cada uno de los bloques.

Los sistemas físicos tienen un alto grado de fiabilidad, siendo más habitual que el origen de los fallos de un sistema informático sea el fallo de su software y no el de su hardware. Esto es debido a que la fiabilidad del hardware ha sido ampliamente estudiada, siendo aplicables los principios de la teoría tradicional de fiabilidad de componentes.

La fiabilidad del software requiere un estudio más específico, ya que no encaja perfectamente en la teoría tradicional de fiabilidad dadas sus características tan especiales, que los diferencian considerablemente de otros tipos de productos, y que son el motivo por el que es más problemático y costoso conseguir que un sistema lógico cualquiera tenga un grado de fiabilidad elevado.

Mientras que es muy razonable suponer que cualquier producto de ingeniería, una vez terminado, probado y vendido, su diseño es correcto y va a funcionar de manera fiable, en el caso de un sistema lógico es habitual encontrar errores importantes, lo que impide un funcionamiento fiable para determinados usuarios.

Los problemas que suelen presentar los sistemas software pueden persistir durante versiones consecutivas del sistema en concreto, llegando algunas veces a empeorar a medida que aumenta el número de actualizaciones realizadas.

El origen de los fallos de un sistema 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, por lo que se podría pensar que los participantes en la creación de los sistemas lógicos no realizan correctamente su trabajo, pero lo cierto es que hasta el programador más preparado no puede evitar cometer errores. No es cuestión, por tanto, de que los métodos y herramientas empleados no sean los correctos, o que los

desarrolladores sean incompetentes, la razón fundamental para que el desarrollo de sistemas software sea tan crítica es su propia complejidad conceptual.

Los sistemas lógicos o software se desarrollan, no se fabrican en el sentido clásico. Son productos inmateriales, que muy frecuentemente se diseñan a medida, por lo que sus cualidades dependen de las características del equipo técnico encargado de su desarrollo. Es muy importante la experiencia y habilidad de cada una de las personas involucradas en el desarrollo del sistema lógico, así como su capacidad de comunicación, la estabilidad y tamaño del equipo, y por supuesto la capacidad de organización, control y decisión de la persona encargada de la dirección del equipo, en el cual pueden llegar a participar un gran número de personas que deben trabajar conjuntamente.

La naturaleza de la aplicación a la que está destinado un sistema lógico es un factor muy influyente en su grado de complejidad. Se abarca un amplio ámbito de aplicaciones independientemente de su naturaleza, desde aplicaciones en la vida cotidiana, tales como seguros, operaciones bancadas, reservas de viajes, llamadas telefónicas, hasta aplicaciones más sofisticadas en los diferentes sectores industriales, variando ampliamente en cada caso el nivel de complejidad y también el nivel de fiabilidad exigido. Hay aplicaciones en las que está involucrada la seguridad de vidas humanas.

Las características propias del proceso de desarrollo de los sistemas lógicos 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 de los sistemas lógicos, 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 experiencia muestra que las fases en las que aparecen las dificultades más importantes son las de especificación y concepción.

Otra característica muy importante de las aplicaciones informáticas es su carácter evolutivo, consecuencia de la necesidad de que dicha aplicación permanezca adaptada al entorno cambiante para el que se diseñó. Es decir, en un determinado software se introducen modificaciones

continuamente y un cambio en uno de sus módulos puede afectar a muchos módulos, incluso de forma imprevista.

Por último, cabría citar la importancia del coste del desarrollo de un sistema software en relación al coste de la totalidad del sistema, que además va aumentando, dado que cada vez son mayores y más complejas las funciones que soporta el software, y que por tanto se exigen niveles más altos de fiabilidad

Se estima que el coste del proceso de desarrollo de un software tiene la estructura indicada en la siguiente tabla.

Tabla 3.1. Coste desarrollo de un software

CONCEPTO	% COSTE	
	Sistemas Pequeños	Sistemas grandes
1. Corrección de defectos	35	40
2. Codificación	25	10
3. Análisis/Diseño	15	3
4. Reuniones/Discusión/Viajes	10	12
5. Documentación	5	15
6. Garantía de Calidad	5	2
7. Soporte Administrativo	4	5
8. Planificación/Control/Seguimiento	1	5
9. Educación/Entrenamiento	0	1
10. Integración	0	7

Elaboración. Propia

Como se puede observar en la Tabla 3.1, el coste estimado para la corrección de errores es el más importante. Esto es debido a que es necesario dedicar una gran cantidad de tiempo y de recursos para detectar y corregir errores en todas las fases del proceso de desarrollo del sistema lógico, y que a pesar de ello se llega a la fase de explotación con un número importante de errores que no han sido detectados en la fase en la que se cometieron, y se han arrastrado a lo largo de las fases posteriores.

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.

3.3. Calidad del software

Actualmente existe un interés y una sensibilidad especiales por la aspiración a la calidad en todos los procesos de producción y en la actividad empresarial. 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.

Esta filosofía también está presente en el proceso de desarrollo del software, ya que en este campo también es de máximo interés que los productos tengan un alto nivel de calidad.

La calidad de un software se define mediante un conjunto de características que no son fácilmente medibles. Estas características son relativas por una parte a la operación del sistema, y por otra a su evolución.

Los atributos de calidad del software están categorizados en seis características (funcionalidad, fiabilidad, usabilidad, eficiencia, mantenibilidad y portabilidad), que se subdividen a su vez en subcaracteristicas.

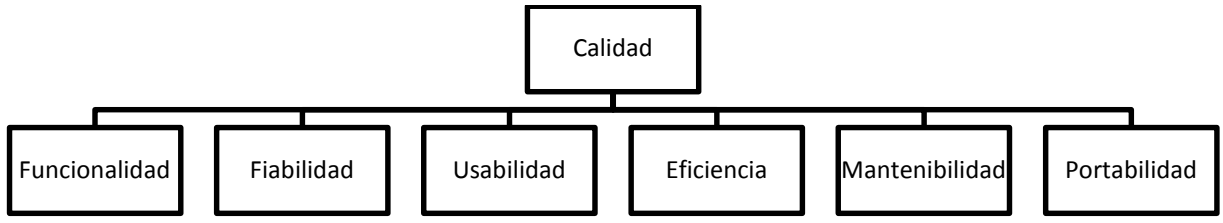

Figura 3.3. Características de la calidad del software

Elaboración. Propia

La calidad de un software será mayor cuanto más elevados sean los niveles de cada una de las características enumeradas. Fijándose el umbral de calidad mediante un nivel mínimo para cada una de ellas.

Para poder alcanzar el umbral de calidad requerido, el diseño y desarrollo del software debe realizarse siguiendo una metodología orientada a la calidad y una vez obtenido el producto se evalúan sus características en relación a los objetivos de calidad fijados.

La primera dificultad que se presenta es como 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 forma poco concreta y es necesario un mayor conocimiento 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.

Se proporcionan a continuación definiciones para cada característica de calidad del software que influyen en la característica de la calidad.

Funcionalidad

La capacidad del producto software para proporcionar funciones que satisfacen necesidades declaradas e implícitas cuando se usa bajo condiciones especificadas. Las subcaracterísticas son la adecuación, exactitud, interoperabilidad y seguridad de acceso.

Fiabilidad

La capacidad del producto software para mantener un nivel especificado de prestaciones cuando se usa bajo condiciones especificadas. Las subcaracterísticas son la madurez, la tolerancia a fallos y la capacidad de recuperación.

Usabilidad

La capacidad del producto software para ser entendido, aprendido, usado y ser atractivo para el usuario, cuando se usa bajo condiciones especificas. Las subcaracterísticas son la capacidad para ser entendido, aprendido, operado y controlado y capacidad de atracción.

Eficiencia

La capacidad para proporcionar prestaciones apropiadas, relativas a la cantidad de recursos usados, bajo condiciones determinadas. Las subcaracterísticas son el comportamiento temporal (tiempos de respuesta, de proceso y potencia, bajo unas condiciones determinadas) y la utilización de recursos.

Mantenibilidad

La capacidad del software para ser modificado. Las modificaciones podrán incluir correcciones, mejoras o adaptación del software a cambios en el entorno, y requisitos y especificaciones funcionales. Entre las subcaracterísticas de la mantenibilidad se encuentra la capacidad para ser analizado, capacidad para ser cambiado, la estabilidad y la capacidad para ser probado.

Portabilidad

Es la capacidad para ser transferido de un entorno a otro. Las subcaracterísticas son la adaptabilidad, la coexistencia y la capacidad para reemplazar a otro producto software.

4 PROCESO DE DESARROLLO DEL SOFTWARE

4.1. Ciclo de vida del software

Un software falla porque durante su diseño y desarrollo, los técnicos cometen errores que dan lugar a la existencia de defectos en el sistema. Así mismo, al aumentar el número de defectos, aumenta también la probabilidad de fallo del sistema. Por tanto, para conseguir un software fiable, es muy importante reducir al máximo el número de defectos presentes. Esto se consigue realizando una serie de pruebas a lo largo del ciclo de vida del software que permitan detectar los posibles defectos presentes en el mismo.

Dado que un software determinado se diseña y desarrolla a lo largo de una serie de fases, es muy importante identificar dichas fases para poder aplicar en cada una de ellas las técnicas más apropiadas para disminuir los errores que se cometen y detectar también los defectos ya existentes en el sistema.

Por ciclo de vida de software se entiende la sucesión de etapas o fases por las que pasa el software desde que nace la idea inicial hasta que el software es retirado o reemplazado (muere). Estas fases están compuestas por tareas que se pueden planificar. Según el modelo de ciclo de vida, la sucesión de fases puede ampliarse con bucles de realimentación, de manera que lo que conceptualmente se considera una misma fase se pueda ejecutar más de una vez a lo largo de un proyecto, recibiendo realimentación.

De forma general, se diferencian tres fases básicas:

Fase de especificación

Esta fase está compuesta por las siguientes actividades:

a. **Adopción e identificación del sistema:** es importante conocer el origen del sistema, así

como las motivaciones que impulsaron el desarrollo del sistema.

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

c. **Especificación:** los requerimientos se realizan con 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.

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

Fase de diseño y desarrollo

Esta fase está compuesta por las siguientes actividades:

a. **Diseño:** en esta etapa se divide el sistema en partes manejables o módulos, y se analizan los elementos que las constituyen. Esto permite afrontar proyectos de muy alta complejidad.

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

c. **Integración y prueba de 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.

d. **Documentación:** generación de documentos necesarios para el uso y mantenimiento del software.

Fase de mantenimiento

Esta fase está compuesta por las siguientes actividades:

a. **Entrenamiento y uso:** instrucciones y guías para los usuarios detallando las posibilidades y limitaciones del sistema.

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

La primera fase es la tarea de escribir detalladamente el software a ser desarrollado, en una forma matemáticamente rigurosa. En la realidad, la mayoría de las buenas especificaciones han sido escritas

para entender y afinar aplicaciones que ya estaban desarrolladas. Las especificaciones son más importantes para las interfaces externas, que deben permanecer estables.

En la fase de diseño y desarrollo se crea físicamente el software. En primer lugar se establece cómo se va a 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 implica mantener y mejorar el software para solventar errores descubiertos y tratar con nuevos requisitos. El mantenimiento puede ser de cuatro tipos: perfectivo (mejorar la calidad interna de los sistemas), evolutivo (incorporaciones, modificaciones y eliminaciones necesarias en un producto software para cubrir la expansión o cambio en las necesidades del usuario), adaptativo (modificaciones que afectan a los entornos en los que el sistema opera), y correctivo (corrección de errores).

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

A continuación se explican los ciclos de vida de desarrollo del software.

4.2. Tipos de modelo de ciclo de vida

Las principales diferencias entre los distintos modelos de ciclo de vida están en:

- El alcance del ciclo dependiendo de hasta donde llegue el proyecto correspondiente. Un proyecto puede comprender un simple estudio de viabilidad del desarrollo de un producto, o su desarrollo completo en el extremo, toda la historia del producto con su desarrollo, fabricación y modificaciones posteriores hasta su retirada del mercado.
- Las características (contenidos) de las fases en que dividen el ciclo. Esto puede depender del propio tema al que se refiere el proyecto, o de la organización.
- La estructura y la sucesión de las etapas, si hay realimentación entre ellas y si se puede iterar.

4.3. Modelos de ciclo de vida

La ingeniería del software establece y se vale de una serie de modelos que establecen y muestran las distintas etapas y estados por los que pasa un producto software, desde su concepción inicial,

pasando por su desarrollo, puesta en marcha y posterior mantenimiento hasta la retirada del producto. A estos modelos se les denomina "modelos de ciclo de vida del software". El primer modelo concebido fue el modelo de Royce o modelo en cascada. Este modelo establece que las diversas actividades que se van realizando al desarrollar un producto software se suceden de forma lineal.

Los modelos de ciclo de vida del software describen las fases del ciclo de software y el orden en que se ejecutan las fases, así como también los criterios de transición asociados entre etapas.

Existen distintos modelos de ciclo de vida, y la elección de un modelo para un determinado tipo de proyecto es realmente importante; el orden es uno de estos puntos importantes. Existen varias alternativas. A continuación se muestran algunos de los modelos tradicionales y más utilizados.

4.3.1. Modelo en cascada

El modelo en cascada propuesto por Roycee 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 que son procesadas de forma lineal. La idea principal es que las diferentes etapas de desarrollo van "fluyendo" hacia las etapas sucesivas.

Es un modelo muy importante puesto que ha sido la base de muchos otros que le han sucedido. Se considera un modelo de desarrollo rígido, por lo que ha quedado desactualizado para muchos proyectos modernos.

En la figura 4.1 se puede observar lo que sería un esquema genérico de este tipo de modelo.

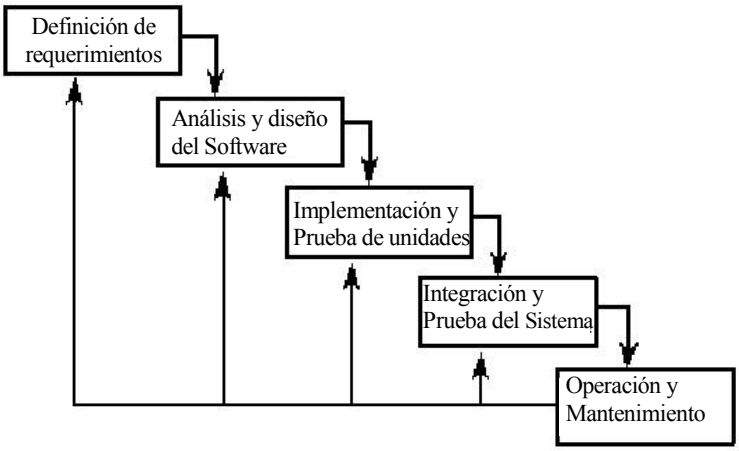

Figura 4.1. Modelo en cascada

Como se recoge en la figura, existen generalmente cinco etapas en este modelo de ciclo de vida:

1. **Análisis y definición de requerimientos.**

   En esta etapa y a partir de un análisis de las necesidades de los usuarios finales del software,

se establecen los requerimientos del producto que se desea desarrollar. Estos consisten usualmente en los servicios que debe proveer el mismo u objetivos así como sus limitaciones. Una vez establecido esto, los requerimientos deben ser definidos de una manera apropiada para ser útiles así en la siguiente etapa. Esta etapa incluye también un estudio de la viabilidad del proyecto y es vista como el comienzo del ciclo de vida.

2. **Diseño del sistema.** El diseño del software es un proceso complejo que se centra en cuatro atributos diferentes de los programas: estructura de datos, arquitectura del software, detalle del proceso y caracterización de las interfaces. El proceso de diseño representa los requerimientos de tal 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 perfectamente documentado y se convierte en parte del producto software.

3. **Diseño del programa.**

   En esta fase se realizan los algoritmos necesarios para el cumplimiento de los requerimientos del usuario así como también los análisis necesarios para saber qué herramientas han de usarse en la posterior etapa de codificación. A menudo suele incluirse una evaluación individual de las unidades de código producidas, antes de su paso a la etapa de integración y prueba global del sistema.

4. **Codificación.**

   En esta fase se implementa el código fuente. Una vez concluida la codificación se realizan pruebas y ensayos para corregir errores, empleando prototipos en el estudio. El proceso se centra en dos puntos principales: las lógicas internas del software y las funcionalidades externas, es decir, se solucionan errores y se asegura que las entradas definidas produzcan resultados reales que coinciden con los requerimientos especificados.

5. **Mantenimiento.**

   Esta etapa consiste en la corrección de aquellos errores no detectados previamente, así como la implementación de mejoras funcionales. Aun no perteneciendo estrictamente a la fase de desarrollo, esta es una de las etapas más críticas y a la que se destina aproximadamente un 75% de los recursos.

Ventajas e inconvenientes

El modelo en cascada es apropiado para proyectos estables donde los diseñadores pueden predecir totalmente áreas de problema del sistema y producen un diseño correcto antes de que empiece la

implementación. Funciona bien en proyectos pequeños donde los requisitos están bien entendidos.

Es un modelo en el que todo está bien organizado y no se mezclan las fases. A pesar de esta rigidez, el modelo es fácil de gestionar puesto que todos los entregables de cada fase están correctamente especificados y están sometidos a un estricto proceso de revisión.

Sin embargo, en la vida real un proyecto rara vez sigue una secuencia lineal, motivo por el que acaban en fracaso si se sigue este modelo. Difícilmente un cliente va a establecer al principio de la creación del software todos y cada uno de los requisitos necesarios por lo que provoca un atraso trabajando con este modelo, al no permitir movimiento entre fases fuera de lo estipulado.

Otro inconveniente es que las mejoras no son visibles progresivamente, el producto se ve cuando ya está finalizado, lo cual provoca una gran inseguridad en el cliente que quiere ir viendo los avances en el producto.

El modelo cascada se aplica bien en situaciones en las que el software es simple y en las que se conocen todos los requerimientos al inicio del desarrollo, la tecnología usada en el desarrollo es accesible y los recursos están disponibles.

4.3.2. Modelo en V

El modelo en V se desarrolló para solucionar algunos de los problemas que presenta la aplicación del modelo en cascada explicado anteriormente: los defectos eran encontrados demasiado tarde puesto que las pruebas no se introducían hasta el final del proyecto.

Este modelo muestra cómo se relacionan las actividades de prueba con el análisis y el diseño. La codificación forma el vértice de la V. La parte izquierda de la V representa la descomposición de los requisitos y la creación de las especificaciones del sistema. El lado derecho representa la integración de partes y su verificación.

La unión de ambos lados ofrece doble información; por un lado sirve para indicar en qué fase de desarrollo se deben definir las pruebas correspondientes y también nos indica a qué fase de desarrollo hay que volver si se encuentran fallos en las pruebas correspondientes.

Mientras el foco del modelo en cascada se sitúa en los documentos y productos desarrollados, el modelo en V se centra en las actividades y la corrección.

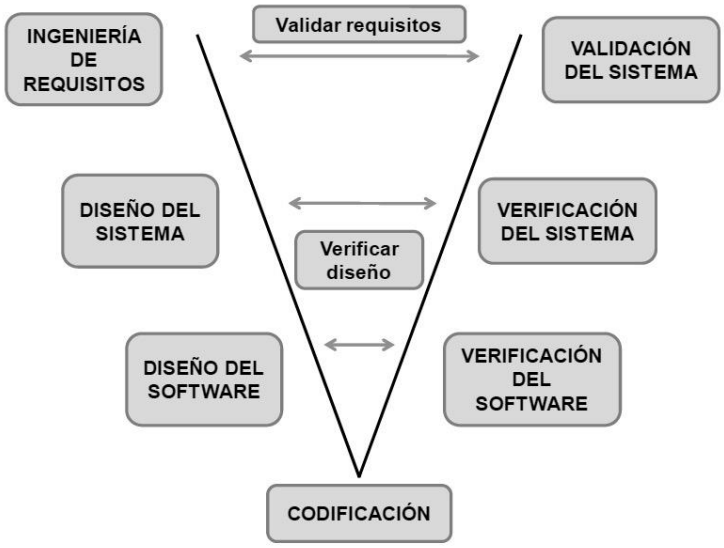

Figura 4.2. Modelo en V

Ventajas e inconvenientes

Las ventajas que se pueden destacar de este modelo son su simplicidad y su uso intuitivo. Además, los entregables de cada fase están especificados de antemano. Tiene una alta oportunidad de éxito sobre el modelo en cascada debido al desarrollo de planes de prueba en etapas tempranas del ciclo de vida. Es un modelo que suele funcionar bien para proyectos pequeños donde los requisitos son entendidos fácilmente.

Por otra parte, uno de los inconvenientes ligados a este modelo es su rigidez, al igual que la del modelo en cascada. Esta poca flexibilidad hace que ajustar el alcance del mismo sea difícil y costoso. Un último inconveniente es que el software se desarrolla durante la fase de implementación, sin que se produzcan prototipos del software.

4.3.3. Modelo Iterativo

El modelo iterativo es también derivado del ciclo de vida en cascada. Este modelo busca reducir el riesgo de que se produzcan malentendidos durante la etapa de recogida de requisitos de los usuarios.

Consiste en la iteración de varios ciclos de vida en cascada. Al final de cada iteración se le entrega al cliente una versión mejorada o con mayores funcionalidades del producto y es él mismo quien evalúa y corrige el producto o propone mejoras. Estas iteraciones se repetirán hasta obtener un producto que satisfaga las necesidades del cliente.

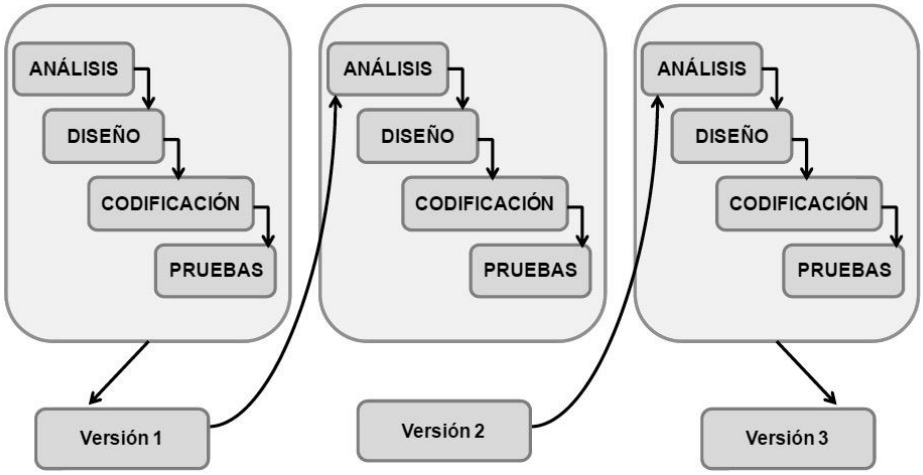

Figura 4.3. Modelo iterativo

Este modelo se suele utilizar en proyectos en los que los requisitos no están claros por parte del usuario, por lo que es necesario fabricar distintos prototipos para presentarlos y conseguir la conformidad del cliente.

Ventajas e Inconvenientes

Una de las principales ventajas que ofrece este modelo es que no hace falta que los requisitos estén totalmente definidos al inicio del desarrollo, sino que se pueden ir refinando en cada una de las iteraciones. Si bien es cierto, esto también puede verse como un inconveniente, ya que pueden surgir problemas relacionados con la arquitectura.

Igual que otros modelos similares, tiene las ventajas propias de realizar el desarrollo en pequeños ciclos, lo que permite gestionar mejor los riesgos o gestionar mejor las entregas.

4.3.4. Modelo Incremental

El modelo incremental combina elementos del modelo en cascada con la filosofía interactiva de construcción de prototipos. La novedad es que este modelo aplica secuencias lineales de forma escalonada mientras progresa el tiempo. Cada secuencia lineal produce un "incremento" del software. El primer incremento es a menudo un producto esencial denominado núcleo y cada incremento agrega funcionalidad adicional o mejorada sobre el sistema

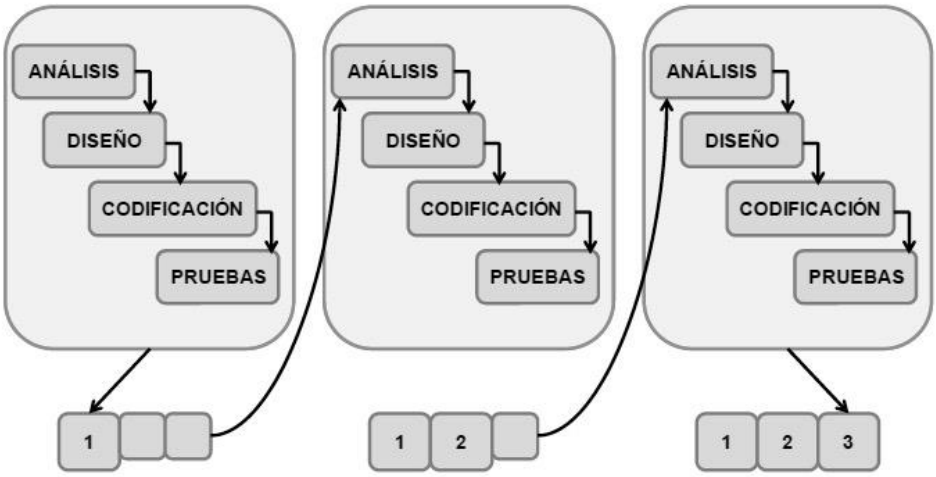

Figura 4.4. Modelo incremental

Ventajas e inconvenientes

Este modelo es iterativo por naturaleza, pero a diferencia de la construcción de prototipos, se centra en la entrega de un producto operacional con cada intento. Así, los primeros incrementos son versiones "incompletas" del producto final que proporcionan al usuario la funcionalidad que precisa y también una plataforma para la evaluación. Al ser un modelo más flexible, cualquier cambio de alcance o requisitos comporta un menor aumento de coste.

El desarrollo incremental es particularmente útil cuando la dotación de personal no está disponible para una implementación completa en la fecha límite que se haya establecido para el proyecto, pues los primeros incrementos se pueden implementar con menos personas.

En cambio, para el uso de este modelo se requiere una experiencia importante para definir los incrementos y distribuir en ellos las tareas de forma proporcionada. Entre los inconvenientes que aparecen en el uso de este modelo se puede destacar que cada fase de una iteración es rígida y no se superpone con otras.

4.3.5. Modelo Prototipado

Este también es un modelo basado en la construcción de prototipos que en cierta medida palia algunas de las deficiencias que presenta el modelo en cascada, descrito anteriormente.

Un cliente, a menudo, define un conjunto de objetivos generales para el software que desea, pero no identifica los requisitos detallados de entrada, proceso o salida. En otros casos, el responsable del desarrollo del software puede no estar seguro de la eficacia de un algoritmo, de la capacidad de adaptación de un sistema operativo o de la forma en que debería tomarse la interacción hombre-máquina. En estas situaciones, el modelo prototipado es el mejor enfoque que se puede plantear.

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.

El paradigma de construcción de prototipos comienza con la recolección de requisitos. El desarrollador y el cliente se encuentran y definen los objetivos globales para el software, identifican los requisitos conocidos y las áreas del esquema en donde es obligatoria más definición. Entonces aparece un "diseño rápido". Este se centra en una representación de aquellos aspectos del software que serán visibles para el usuario. El diseño rápido lleva a la construcción de un prototipo. El prototipo es evaluado por el cliente, lo que sirve para refinar los requisitos del software a desarrollar.

Figura 4.5. Modelo prototipado

Ventajas e inconvenientes

Este modelo ofrece visibilidad del producto desde el inicio del ciclo de vida con el primer prototipo. Esto puede ayudar al cliente a definir mejor los requisitos y a ver las necesidades reales del producto. Permite introducir cambios en las iteraciones siguientes del ciclo, así como la realimentación continua del cliente.

El cliente reacciona mucho mejor ante el prototipo, sobre el que puede experimentar, que no sobre una especificación escrita de ahí que este modelo reduzca el riesgo de construir productos que no satisfagan las necesidades de los usuarios.

Entre los inconvenientes que presenta este modelo está el hecho de que puede ocasionar un desarrollo lento. Es necesario también realizar fuertes inversiones en un producto desechable, pues los prototipos son posteriormente descartados, aumentando así el coste total de desarrollo.

4.3.6. Modelo en Espiral

El modelo en Espiral propuesto originalmente por Barry Boehm, es un modelo de proceso de software evolutivo que conjuga la naturaleza iterativa de construcción de prototipos con los aspectos controlados y sistemáticos del modelo en Cascada.

Las actividades de este modelo se conforman en una espiral donde cada bucle representa un conjunto de actividades. Las actividades no están fijadas a priori, sino que las siguientes se eligen en función del análisis de riesgos, comenzando por el bucle anterior.

En el modelo espiral, el software se desarrolla en una serie de versiones incrementales. Durante las primeras iteraciones, la versión incremental podría ser un modelo en papel o un prototipo, mientras que durante las últimas iteraciones, se producen versiones cada vez más completas del sistema diseñado.

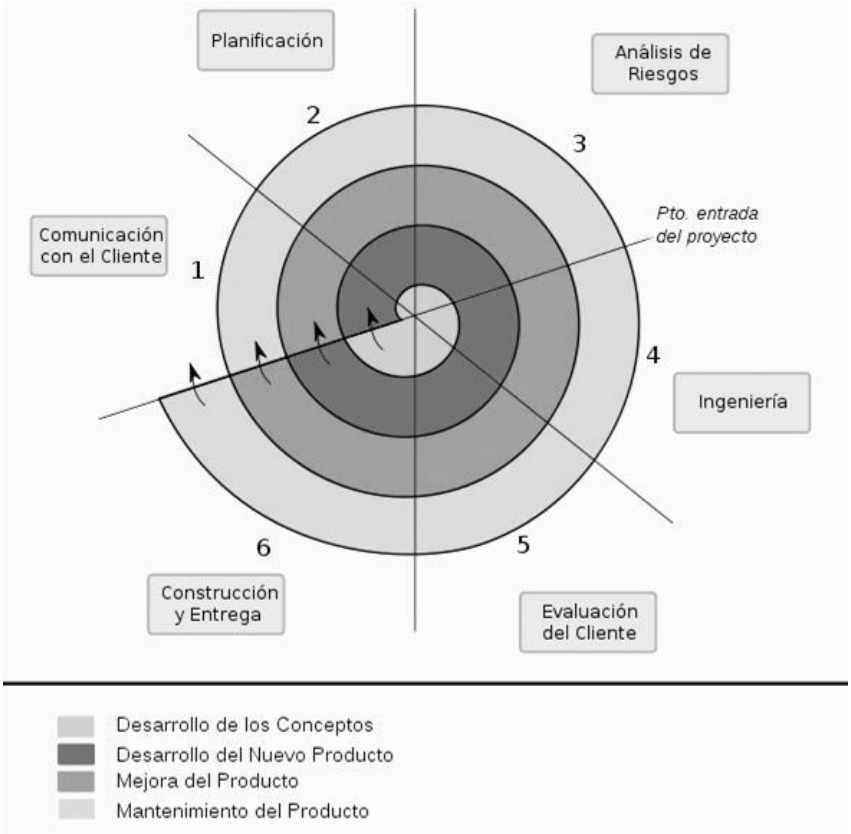

Figura 4.6. Modelo en espiral

El modelo en espiral se divide en un número de actividades o regiones de tareas. Generalmente, existen entre tres y seis regiones de tareas, según se quiera que sea el nivel de división. La figura anterior representa un modelo en espiral que contiene las seis regiones de tareas. Estas son:

1. **Comunicación con el cliente** – las tareas requeridas para establecer comunicación entre el desarrollador y el cliente.

2. **Planificación** – las tareas requeridas para definir recursos, tales como el tiempo u otra información relacionada con el proyecto.
3. **Análisis de riesgos** – las tareas requeridas para evaluar riesgos técnicos y de gestión.
4. **Ingeniería** – las tareas requeridas para construir una o más representaciones de la aplicación.
5. **Construcción y entrega** – las tareas requeridas para construir, probar, instalar y proporcionar soporte al usuario, por ejemplo: documentación y ayuda práctica.
6. **Evaluación del cliente** – las tareas requeridas para obtener la opinión del cliente sobre el software creado.

El conjunto de tareas del trabajo de cada región, llamado *conjunto de tareas*, se adapta a las características del proyecto que va a emprenderse. Así, para proyectos pequeños el número de tareas de trabajo y su formalidad es bajo mientras que en proyectos de mayor envergadura y más críticos el conjunto de tareas es definido para lograr un nivel mas alto de formalidad.

Cuando empieza este proceso evolutivo, el equipo de ingeniería del software gira alrededor de la espiral en la dirección de las agujas del reloj, comenzando por el centro. El primer circuito de la espiral puede producir el desarrollo de una especificación de productos; los pasos siguientes en la espiral se podrían utilizar para desarrollar un prototipo y progresivamente versiones más sofisticadas del software. Cada paso por la región de planificación produce ajustes en el plan del proyecto. El coste y la planificación se ajustan con la realimentación ante la evaluación del cliente. Además, el gestor del proyecto ajusta el número planificado de iteraciones requeridas para completar el software.

Ventajas e inconvenientes

Entre las principales ventajas que posee este modelo es que puede adaptarse y aplicarse a lo largo de la vida del software, intenta eliminar errores en las fases tempranas y permite una la reevaluación después de cada fase, recibiendo así posibles cambios en las percepciones de los usuarios, avances tecnológicos y modificaciones en las perspectivas financieras.

Proveer más flexibilidad que la conveniente para la mayoría de las aplicaciones o la ausencia de guía explicito para determinar objetivos, limitaciones y alternativas son dos puntos débiles también a tener en cuenta.

4.4. Utilización de la fiabilidad durante el desarrollo del software

Cuando se está desarrollando un software, es común realizar pruebas para detectar fallos. Las pruebas que siguen a las actividades de desarrollo deben seguir una planificación cuidadosa. Existen

dos etapas diferenciadas: en primer lugar se establecen las especificaciones de las pruebas que se realizarán, de tal forma que quede definido lo que se va a probar; seguidamente, sobre las bases de las especificaciones de pruebas, se establece el modo de proceder en la prueba.

El resultado de una de estas pruebas es un informe relativo a los defectos encontrados. En base a estos informes se decide si el producto que se ha probado puede pasar a la siguiente fase, o es necesario seguir trabajando en él. Hay también situaciones en las que por determinados motivos el producto pasa a la siguiente fase con defectos conocidos. En tal caso, se tendrá como output de la fase en la que se encuentre un documento explicativo de la anomalía.

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, sin olvidarnos de 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.

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.

Tras analizar una población suficientemente grande, la curva ideal de fallos de software cobra la forma siguiente:

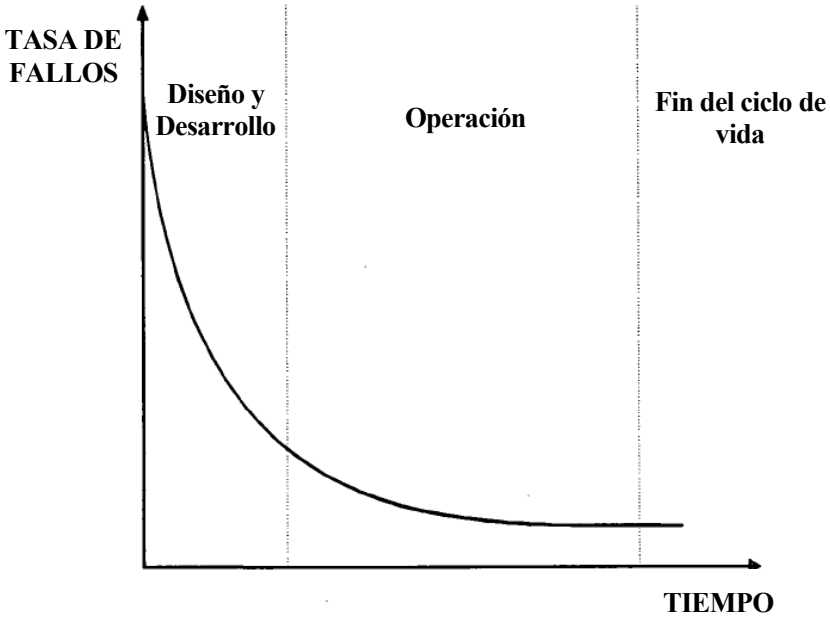

Figura 4.7. Curva ideal de fallos de software

Si se compara esta gráfica con aquella obtenida para Hardware, saltan a la vista diferencias entre software y hardware. La probabilidad de que un software falle decrece de manera continua en el tiempo, lo contrario que ocurre en el hardware. En la gráfica de este último se aprecian tres etapas: en primer lugar está la etapa infantil, correspondiente a las fases de diseño y pruebas del sistema. Aquí, la probabilidad de fallo va disminuyendo hasta que llega a ser prácticamente constante, permaneciendo así durante la vida útil del sistema, para finalmente crecer al término de su ciclo de vida.

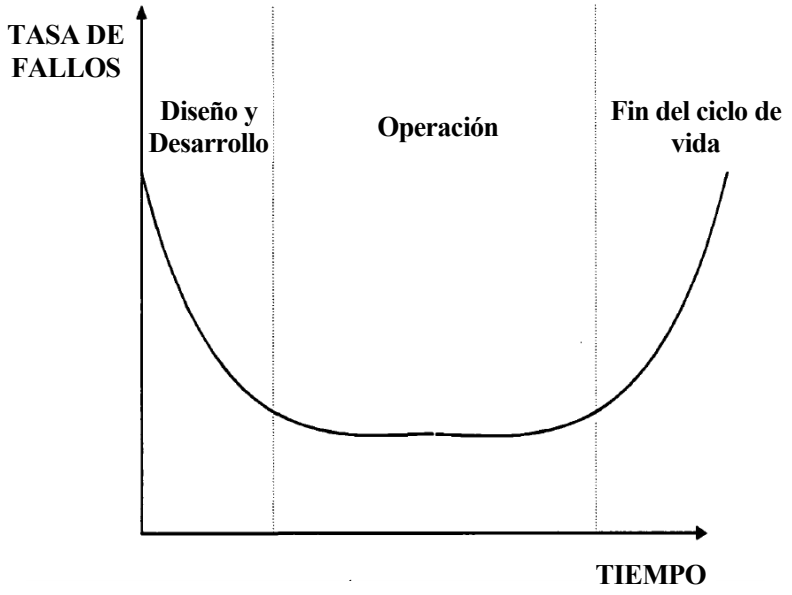

Figura 4.8. Curva ideal de fallos de hardware

Esta diferencia de comportamientos se debe a que el mecanismo de fallo en un software es distinto al de un hardware, si bien ambos tienen el mismo resultado: la degradación del sistema. El hardware falla debido a concentraciones de esfuerzos, desgastes físicos u otros factores atmosféricos o ambientales; mientras que el software falla como consecuencia de errores humanos cometidos durante las fases del ciclo de vida del sistema.

En la figura correspondiente a la tasa de fallos de software, se ha considerado que siempre que ocurre un fallo, se detecta su causa y se corrige, disminuyendo por tanto de manera progresiva la existencia de defectos en el sistema y en consecuencia la probabilidad de fallo. Sin embargo, en la práctica muchas veces no es posible detectar la causa de un fallo o el mismo 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, que se manifiestan en forma de picos en la curva de fallos.

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

Por todo ello, como se puede ve 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.

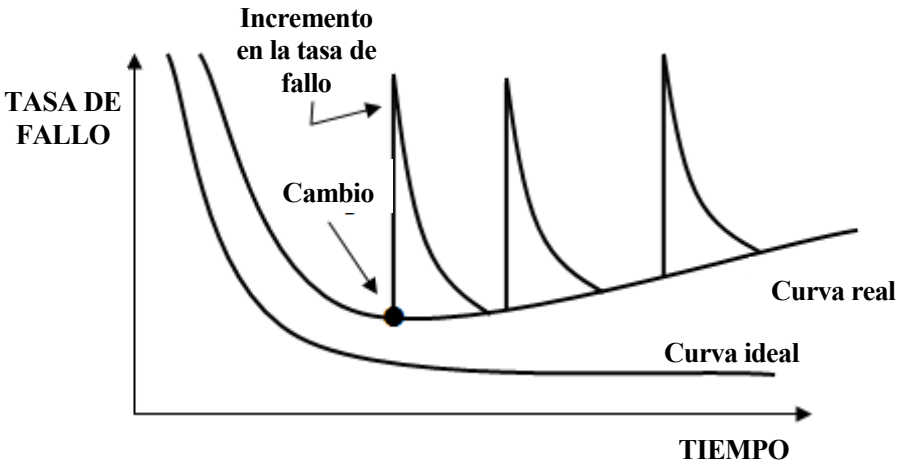

Figura 4.9. Curva de fallos de software cuando se producen correcciones

5 MODELIZACIÓN DE LA FIABILIDAD

5.1. Introducción

Con el papel cada vez más importante que el software ha cobrado en la mayoría de nuestros sistemas, la preocupación por la calidad del mismo ha aumentado de manera significativa a lo largo de la segunda mitad del siglo XX hasta nuestros días. Puesto que el aspecto mas importante de la calidad es la fiabilidad, la Ingeniería de Calidad del Software, SRE (por sus siglas en inglés) ha generado una gran cantidad de interés e investigaciones en la comunidad científica. Un aspecto particular de la SRE que ha recibido gran atención es la modelización de la fiabilidad del software, que tiene por objeto dar una descripción precisa en términos probabilísticos de la fiabilidad de dichos sistemas en base a hipótesis sobre los factores que pueden afectarle y a datos empíricos.

Después de seleccionar algunos de los modelos más significativos que han aparecido en la literatura, en este capítulo se tendrá un primer acercamiento teórico a los mismos, presentando las asunciones de cada uno así como los datos requeridos para poder llevar a cabo su implementación. Finalmente se mostrará la forma del modelo que será empleada en capítulos posteriores para llevar a cabo un análisis empleando dichos modelos.

La estimación que se obtiene será la basada en el método estará basada en el procedimiento para obtener el Estimador de Máxima Verosimilitud, en adelante denotado por sus siglas EMV. Este estimador ofrece muchas propiedades deseables en un estimador, como es la invarianza, consistencia, insesgadez asintótica y normalidad asintótica.

Antes de considerar los modelos es conveniente conocer la perspectiva histórica del desarrollo de este campo y algunos apuntes sobre teoría de fiabilidad, que posteriormente serán usados en el desarrollo de cada modelo. En primer lugar, se considerarán los modelos de clase exponencial, puesto que son estos los mas importantes. Otras formas de distribución de los datos de fallos, como las distribuciones Weibull y gamma serán también consideradas. Es necesario también tratar aquellos

modelos basados en una perspectiva Bayesiana, señalando las diferencias con aquellos basados en teorías más tradicionales.

5.2. Perspectiva histórica e implementación

La modelización de la fiabilidad del software comienza, aunque sorprenda a muchos, a principios de la década de 1970. Este primer acercamiento consiste en modelizar datos de errores pasados para predecir el comportamiento futuro. Para ello se emplea bien el número de fallos encontrados por período de tiempo o el tiempo de observación entre los fallos del software. Los modelos se dividen por tanto en dos clases dependiendo de cual sea el tipo de datos usados.

1. Fallos por período de tiempo
2. Tiempo entre fallos

Estas clases no son sin embargo mutuamente excluyentes, puesto que hay modelos que pueden ser empleados con ambos tipos de datos. Además, muchos de los modelos para un tipo de datos pueden seguir siendo utilizados incluso si el usuario emplea datos del otro tipo, ayudándose eso sí, de algún procedimiento de transformación de datos.

5.3. Principios de la modelización

Para conseguir que un software tenga un alto grado de fiabilidad existen tres caminos que no son excluyentes entre si. Si durante el diseño y desarrollo del sistema se sigue una metodología normalizada, se garantiza la fiabilidad del producto que se obtiene. Un segundo enfoque es el desarrollo de sistemas tolerantes a fallo, capaces de funcionar correctamente incluso existiendo defectos en el sistema, pues se han diseñado con las redundancias adecuadas o contienen procedimientos de actuación en caso de error. El tercer camino consiste en la realización de pruebas del sistema durante la etapa de desarrollo, con el fin de detectar los defectos existentes y corregirlos.

A pesar del interés creciente en los dos primeros enfoques, la realización de pruebas sigue siendo el método más utilizado para alcanzar un determinado nivel de fiabilidad en un software y normalmente los períodos de prueba del mismo representan un porcentaje de tiempo importante en su ciclo de vida.

Durante los períodos de prueba descritos se observa que la fiabilidad experimenta un crecimiento, siendo resultado de las acciones que se toman para corregir los defectos que contiene el sistema en pruebas y que se detectan al manifestarse como fallos.

Mediante una modelización adecuada del software se pretende disponer de valores concretos de su fiabilidad durante la etapa de pruebas, para poder predecir los instantes en los que se producirán nuevos fallos. Es necesario no obstante saber la evolución del proceso de prueba.

En otras palabras, se pretende estimar los valores de las variables aleatorias $(T_{k+1}, T_{k+2} \dots)$ relativas a futuros tiempos de fallo, cuando se conoce que han ocurrido fallos en los momentos $(t_1, t_2, \dots, t_k)$. Para ello es necesario disponer de un modelo probabilístico que especifique la distribución de probabilidad de las variables aleatorias T_k condicionadas sobre unos determinados parámetros, ya que el modelo especifica la forma general de dependencia del proceso de fallo con respecto a las variables que influyen sobre él. Para cada caso concreto se particulariza el modelo determinando los valores específicos de los parámetros, mediante un procedimiento de inferencia estadística que requiere disponer de datos sobre como van evolucionando las pruebas.

Al modelizar el proceso de fallo, una hipótesis muy generalizada es suponer que cuando se produce un fallo del software, inmediatamente se detecta y corrige el defecto que lo originó, es decir, considerar siempre que las correcciones efectuadas son perfectas. Algunos modelos tienen en cuenta la posibilidad de que no se consiga detectar o corregir el defecto que ocasiona un fallo, dejando el sistema en el mismo estado en el que estaba antes de que se produjese el fallo en cuestión. No se tendrá en cuenta el caso en que al intentar corregir un defecto que ha ocasionado un fallo del software, se cometan nuevos errores y por tanto se introduzcan nuevos defectos, lo que conlleva una disminución de la fiabilidad del sistema que se está probando.

Los modelos que se definen suponiendo una corrección perfecta de cada error que se manifiesta también se basan en la hipótesis de que cada defecto tiene como consecuencia un único fallo del software y por tanto, el número de fallos que ocurren durante un tiempo t es igual al número de defectos que se corrigen.

Otra hipótesis muy generalizada en la modelización del proceso de fallo es suponer que los fallos son independientes, lo que se modeliza considerando que los tiempos de fallo son variables aleatorias independientes entre si. Esta hipótesis se basa en que los fallos ocurren como consecuencia de dos procesos aleatorios: uno es la introducción de defectos en el sistema como consecuencia de los errores que se cometen durante su diseño y desarrollo, y otro la selección de los valores correspondientes a las variables de entrada. Dado que ambos procesos son aleatorios, la posibilidad de que un fallo influya sobre otro es muy remota. Esta hipótesis ha sido cuestionada alguna vez como consecuencia de considerar que los programadores pueden tener patrones de error, pero eso daría lugar a la existencia de defectos dependientes entre si, lo cual no tiene porque implicar una dependencia de fallos. Esta hipótesis ha sido constatada empíricamente en un estudio de correlación

entre tiempos de fallo correspondientes a pruebas realizadas en quince proyectos.

Hay un número importante de modelos que consideran que todos los defectos que contiene un software tienen la misma probabilidad de manifestarse, lo cual se traduce en que la intensidad del proceso de fallo del programa en estudio presenta discontinuidades de valor constante en los instantes de fallo como consecuencia de las acciones tomadas para corregir el defecto que ocasiona el fallo.

Otros modelos consideran que en todo programa existen partes de código que se ejecutan más frecuentemente, por lo que los defectos contenidos en dichas secciones deben tener una probabilidad mayor de manifestarse. También puede ocurrir que determinados errores puedan manifestarse con un número mayor de casos de prueba, por lo que su probabilidad de ocasionar un fallo también es mayor. Los defectos que mayor probabilidad tienen de manifestarse se van detectando normalmente al comienzo de las pruebas y según van avanzando estas, los tiempos entre fallos son cada vez mayores ya que los defectos que aún quedan en el sistema son los que tienen una probabilidad menor de manifestarse.

Generalmente todos los modelos suponen una distribución de probabilidad para las variables aleatorias correspondientes a los tiempos de fallo. La más común es la distribución exponencial, pero también son muy frecuentes la distribución de Weibull, Gamma, Pareto, etc..

5.4. Clasificación de los modelos

El primer estudio sobre fiabilidad de software es el publicado por Hudson en 1967. En él se considera el proceso de desarrollo de dichos sistemas como un proceso de nacimiento y muerte, es decir, un tipo de proceso de Markov. Durante el desarrollo del software, la introducción de defectos se considera como nacimientos, y su correlación como muertes, siendo el número de defectos presentes en el sistema en un instante determinado la variable que define el estado del proceso. Las probabilidades de transición entre estados dependen de las probabilidades de que en un momento determinado ocurra un nacimiento o una muerte. Con el fin de que las ecuaciones matemáticas que representan el modelo no fueran excesivamente complejas, el problema se simplifico considerando únicamente la probabilidad de muerte, la cual se supone proporcional al numero de defectos existentes en el programa y a una potencia positiva del tiempo. El estudio también incluye una comparación entre datos reales correspondientes al periodo de pruebas de un sistema determinado y las estimaciones proporcionadas por el modelo que demostraba un cierto acuerdo entre ambos conjuntos de datos, especialmente durante una parte concreta de dicho periodo.

Este primer estudio da comienzo a una etapa que dura unas dos décadas, en la que se dedican verdaderos esfuerzos a conseguir formular un modelo matemático, ampliamente aplicable, que describa fielmente el proceso de fallo de los sistemas lógicos, permitiendo cuantificar su fiabilidad en cualquier instante de tiempo.

El resultado de estos esfuerzos es la existencia hoy en día de unos 200 modelos, muchos de ellos desarrollados tomando como base aquellos modelos anteriores. Es muy difícil, sin embargo, elegir aquel modelo que reproduce más fielmente el proceso de fallo de un software cualquiera. La exactitud de las estimaciones que proporcionan los distintos modelos es muy variable, según se adapten las hipótesis sobre las que se basa el modelo al caso concreto que se este estudiando.

Como consecuencia de la proliferación de modelos y de las discrepancias en sus estimaciones, nació la preocupación de cómo elegir el modelo más adecuado para cada caso, y cómo evaluar los resultados obtenidos en la aplicación de un determinado modelo. Con el fin de facilitar la elección, se han hecho muchos esfuerzos a la hora de conseguir una buena clasificación de los modelos existentes.

La primera clasificación fue propuesta por Musa y Okumoto. Esta permite establecer relaciones entre aquellos modelos del mismo grupo y muestra donde se ha producido el desarrollo del modelo. Para este esquema, Musa y Okumoto clasificaron los modelos en función de cinco atributos diferentes. Estos son:

1. **Dominio del tiempo.** Según la medición del tiempo sea en tiempo de ejecución o de calendario.
2. **Categoría del modelo,** definida como el numero finito o infinito de fallos que, según la ecuación del mismo , pueden llegar a ocurrir en un tiempo infinito.
3. **Tipo del modelo**, dado por la distribución del número de fallos hasta el tiempo t, considerando únicamente distribuciones binomiales y distribuciones no homogéneas de Poisson, satisfaciendo además las siguientes condiciones:
    a. Cada fallo se debe a un defecto a del programa y ocurre de forma aleatoria e independiente, siendo el tiempo hasta el fallo T_a una variable aleatoria de función de distribución F_a, igual para todos los fallos.
    b. Al iniciarse las pruebas de un programa, el número de defectos que contiene es un número determinado n_0 o ω_0, según se trate de tipo binomial o Poisson.
    c. Cada vez que ocurre un fallo se corrige el defecto que lo ha originado de manera instantánea (tipo binomial).
4. **Clase del modelo**, para los modelos de número finito de fallos. Forma funcional de la

tasa de fallos, expresada en función del tiempo.

5. **Familia del modelo**, para los modelos de número infinito de fallos. Forma funcional de la intensidad de fallo expresada en términos del número esperado de fallos experimentados.

La clasificación de los modelos para el análisis de la fiabilidad de software, en base a los criterios desarrollados por Musa conduce, sobre todo el tercer criterio, a una desfiguración de los propios modelos e introduce una mayor dificultad para su comprensión.

En trabajos posteriores se han presentado otros esquemas de clasificación de los modelos de fiabilidad de software que eliminan algunas de las dificultades encontradas en la primitiva clasificación de Musa. Una de estas es la clasificación que se expone a continuación, basada en que el proceso de fallo del software es un proceso aleatorio que cumple la propiedad de Markov y también de Poisson, y es independiente del proceso de corrección, por lo que se considera que los modelos siempre obedecen a un proceso de Poisson, clasificándose además según los siguientes criterios:

1. **Correcciones perfectas o imperfectas**. Los modelos pueden considerar que, o bien siempre se realizan correcciones perfectas, o bien hay veces que no se consigue eliminar correctamente el defecto que ha ocasionado el fallo del software. Ambos tipos de modelos siempre consideran que el proceso de corrección es instantáneo.
2. **Número finito o infinito de fallos.** Las ecuaciones que definen los modelos pueden permitir que el número de fallos posibles en un tiempo infinito sea finito o infinito.
3. **Intensidad del proceso de fallo.** La intensidad del proceso de fallo $\lambda(t)$, que es un proceso de Poisson, puede ser una función del tiempo (lineal, exponencial, logarítmica, etc.) o bien una constante.

5.5. Descripción de los modelos más significativos

El objetivo de este apartado es presentar los modelos que se han considerado más significativos durante el estudio de los métodos que actualmente se utilizan en la evaluación de la fiabilidad del software durante sus periodos de prueba.

Para cada modelo elegido se analizan las hipótesis sobre las que se ha desarrollado el modelo y se indica cual es la intensidad del proceso de fallo que corresponde a dichas hipótesis.

Los modelos se han elegido procurando presentar con cada uno de ellos una hipótesis diferenciadora (forma de comportamiento), teniéndose en cuenta la clasificación propuesta en el apartado anterior,

con el fin de que los modelos que se abordan cubran todo el espectro de posibilidades.

5.5.1. Modelos con tiempos de fallo siguiendo distribución exponencial

En la literatura sobre fiabilidad de software, esta clase es sobre la que más artículos se han escrito. Siguiendo el esquema de clasificación de Musa y Okumoto, este grupo esta compuesto por modelos con número infinito de fallos con la función de intensidad de fallo de forma exponencial.

5.5.1.1. Modelo de Jelinski-Moranda

En 1972, mientras trabajaban en proyectos de la Armada estadounidense para la compañía McDonell Douglas, Jelinski y Moranda desarrollaron un modelo para medir la fiabilidad del software. Este modelo es considerado el primero de todos y es uno de los más estudiados y aplicados, aún hoy en día.

Está basado en varías suposiciones entre ellas que la tasa de fallo de cada defecto no cambia a lo largo del tiempo, sino que permanece constante. Como indica el grupo en el que se recoge, el tiempo transcurrido entre fallos sigue una distribución exponencial, con un parámetro proporcional al número de defectos residuales en el software.

El tiempo medio entre fallos en el instante t es $1/(\phi(N-(i-1))$. Aquí, t es cualquier punto en el tiempo entre la ocurrencia del $(i-1)$-ésimo e i-ésimo fallo. La cantidad ϕ es una constante de proporcionalidad y N es el número total de defectos en el sistema en el momento inicial en el que el software es observado.

La figura 5.1 muestra el impacto que el hecho de encontrar un defecto tiene en la tasa de fallo. Se puede apreciar que cada vez que un defecto es descubierto la tasa de fallo se reduce en la constante de proporcionalidad ϕ. Esto indica que el impacto de cada defecto corregido es el mismo.

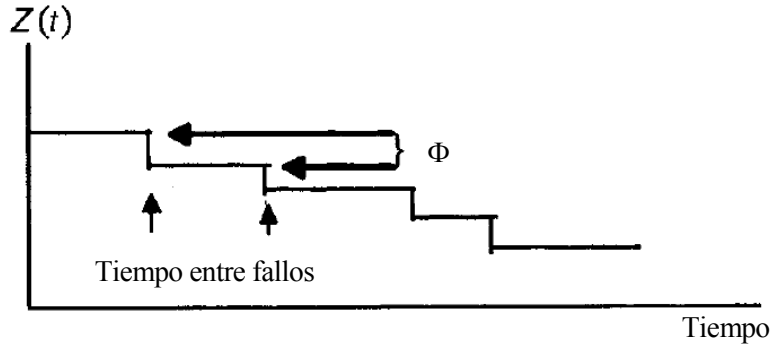

Figura 5.1. Tasa de fallo para modelo Jelinski-Moranda

El modelo Jelinski-Moranda supone las siguientes condiciones:

- Al principio de las pruebas, el código del software contiene un número fijo de defectos desconocido, N.
- Durante el período de pruebas, los defectos del código son independientes y tienen la misma probabilidad de causar un fallo en el programa.
- Los tiempos entre fallos son independientes y siguen una distribución exponencial.
- Cuando se detecta un fallo, el defecto que lo causa es corregido sin que se introduzcan nuevos defectos en el proceso de corrección.
- La tasa de detección de fallos es constante y proporcional al número de defectos que permanecen en el software.

Puesto que las tres últimas suposiciones son poco realistas, es difícil aplicar este modelo a numerosas baterías de datos si bien es cierto que esto no ha impedido, como se comentaba anteriormente, que este modelo si no el que más, de los mas conocidos de todos los modelos de fiabilidad del software.

Forma del Modelo

La tasa de fallos del modelo Jelinski-Moranda para el intervalo $i\text{-}\acute{e}simo$ viene dada por:

$$\lambda(t_i) = N\phi e\,\mathrm{xp}(-\phi t_i) = \phi[N - \mu(t_i)]$$

donde, como se ha explicado antes, la cantidad ϕ es una constante de intensidad de fallo y N es el número total de defectos en el sistema en el momento anterior a la fase de pruebas.

Las funciones de valor medio e intensidad de fallo para este modelo pueden ser obtenidas multiplicando el numero de defectos inherentes por las funciones de fallos acumulados y densidad de probabilidades, respectivamente:

$$\mu(t_i) = N(1 - \exp(-\phi t_i)$$

y

$$\xi(t_i) = N\exp(-\phi t_i)$$

Estimación y predicción de fiabilidad

El estimador de máxima verosimilitud es la solución a estas ecuaciones

$$\hat{\phi} = \frac{n}{\widehat{N}(\sum_{i=1}^{n} X_i) - \sum_{i=1}^{n}(i-1)X_i}$$

$$\sum_{i=1}^{n} \frac{1}{\widehat{N} - (i-1)} = \frac{n}{\widehat{N} - \left(\frac{1}{\sum_{i=1}^{n} X_i}\right)(\sum_{i=1}^{n}(i-1)X_i)}$$

Tras resolver la segunda ecuación primero, se incluye la solución en la primera para encontrar el estimador de máxima verosimilitud de ϕ.

5.5.1.2. Modelo de Goel-Okumoto

En 1979 Goel y Okumoto proponen modelizar la ocurrencia de fallos mediante un proceso no homogéneo de Poisson, dado que se considera demostrada la independencia entre fallos. La diferencia principal entre este modelo y el propuesto por Jelinski y Moranda es la suposición de que el numero esperado de fallos observados en un tiempo t sigue una distribución de Poisson con función de valor medio no decreciente $\mu(t)$. Si se observa el software durante un tiempo infinito, el numero esperado de fallos es el valor finito N

Este modelo también hace las siguientes suposiciones:

- El número de fallos que ocurren en el intervalo $(t, t + \Delta t)$ es proporcional al valor esperado de defectos no detectados, $N - \mu(t)$.
- Los números de fallos detectados en los intervalos $(0, t_1), (t_1, t_2), \dots, (t_{n-1}, t_n)$ no están correlacionados.
- La tasa de intensidad de fallo es una constante invariante ϕ.
- El proceso de corrección cada vez que se produce un fallo es instantáneo y perfecto.

Estas suposiciones generan la siguiente función de valor medio (para la intensidad de fallo):

$$\mu(t_i) = N(1 - \exp(-\phi t_i)$$

$$\lambda(t_i) = N\phi \exp(-\phi t_i) = \phi[N - \mu(t_i)]$$

Se puede apreciar que este modelo es matemáticamente equivalente al modelo Jelinski-Moranda . Las diferencias entre ambos son las suposiciones y la interpretación de que es N. En el caso del modelo J-M, el valor de N es conocido y fijo, mientras que en el modelo G-O no es así, sino que es un valor esperado.

5.5.2. Modelos con tiempos de fallo siguiendo distribuciones Weibull y Gamma

Estas son distribuciones importantes gracias al alto grado de flexibilidad dado para el modelado de fallos, debido a los parámetros de forma y escala que las definen. Muchos procesos de fallo en hardware son modelados usando también estas distribuciones.

5.5.2.1. Modelo Weibull

Uno de los modelos más ampliamente utilizados para el modelado de fiabilidad del software es la distribución Weibull. Puede aplicarse tanto para tasas de fallo crecientes, decrecientes y constantes gracias a su gran flexibilidad. Este modelo pertenece a la categoría de fallos finitos y es de tipo binomial, siguiendo la clasificación de Musa vista anteriormente.

Incluyendo las suposiciones vistas en el modelo de Jelinski-Moranda, las que este modelo añade son:

- El tiempo hasta el fallo producido por el defecto a, denotado por T_a, sigue una distribución Weibull con parámetros α y β. Esto es, la función densidad de T_a es $f_a(t) = \alpha\beta t^{\alpha-1}\exp(-\beta t^{\alpha})$ with $\alpha, \beta > 0$ and $t \geq 0$. Puesto que la función de densidad de fallo es f_a, la tasa de fallo es $z_a(t) = \alpha\beta t^{\alpha-1}$.
- El número de fallos $f_1, f_2, \dots, f_n$ detectados en los respectivos intervalos $[(t_0 = 0, t_1), (t_1, t_2), \dots, (t_{i-1}, t_i), \dots, (t_{n-1}, t_n)]$ son independientes.

Los datos requeridos para implementar este modelo son:

1. Los fallos producidos en cada intervalo de prueba.
2. El tiempo de finalización de cada período en los que el software está bajo observación.

Forma del Modelo

Como se ha indicado, este modelo pertenece al grupo binomial. Al igual que en el modelo de Jelinski-Moranda, se puede hacer uso de las propiedades de este tipo de modelos, expresadas con las siguientes ecuaciones:

$$\mu(t) = \alpha F_a(t)$$

y

$$\lambda(t) = \mu'(t) = \alpha F_a(t)$$

Empleando además la función de distribución acumulativa para una distribución Weibull, se obtienen las funciones de intensidad de fallo y valor medio:

$$\lambda(t) = N f_a(t) = N\alpha\beta t^{\alpha-1}\exp(-\beta t^{\alpha})$$

y

$$\mu(t) = N F_a(t) = N(1 - \exp(-\beta t^{\alpha}))$$

El límite $\lim_{t\to\infty} \mu(t) = N$ es el número total de defectos en el sistema en el momento inicial. También,

a partir de las asunciones hechas se tiene que si $\alpha = 1$ la distribución f_a se hace exponencial y si es igual a 2 la distribución sigue la propuesta por Rayleigh, otro modelo de fallos de la teoría de fiabilidad del hardware.

En el caso de que el parámetro esté comprendido entre 0 y 1, esto es, $0 < \alpha < 1$, la tasa de fallo es decreciente con respecto del tiempo; si $\alpha = 1$ (exponencial) es constante; y si $\alpha > 1$ es creciente.

La forma de la tasa de fallo condicional es:

$$z(t|t_{i-1}) = (N - i + 1)\alpha\beta(t + t_{i-1})^{\alpha-1} \qquad for \;\; t_{i-1} \leq t + t_{i-1} < t_i$$

Esta función esta representada en la figura que se muestra a continuación para $0 < \alpha < 1$, para contrastarla con el modelo exponencial ilustrado en la figura del modelo Jelinski- Moranda. En dicho caso, se produjo un cambio copn cada defecto detectado, y era constante. Para la distribución Weibull, el cambio ocurre cuando se detecta un defecto también, pero tal cambio no es constante. El efecto en la tasa de fallo decrece con el tiempo.

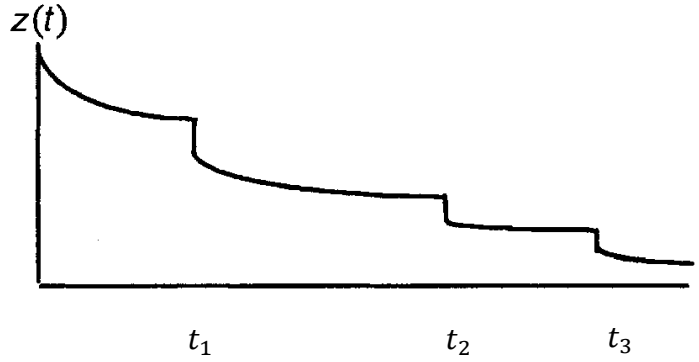

Figura 5.2. Función de la tasa de fallo para la distribución Weibull

La función de fiabilidad es obtenida a partir de la función de distribución acumulativa empleando la relación $R(t) = 1 - F(t) = \exp(-\beta t^{\alpha})$ y por consiguiente, el tiempo medio hasta el fallo MTTF es:

$$MTTF = \int_0^{\infty} R(t)dt = \Gamma\left(\frac{1}{\alpha} + 1\right)/\beta^{\frac{1}{\alpha}}$$

donde $\Gamma(\bullet)$ es la función gamma.

Estimación y predicción de fiabilidad

Los parámetros α y β pueden ser estimados usando el método de los momentos, el de mínimos cuadrados, con estimador de máxima verosimilitud o incluso procedimientos gráficos. Para el caso de los estimadores de mínimos cuadrados, suponiendo el valor de $b = \ln(\beta)$, $Y_i = ln\big[ln[1/(1 - F(i)]\big]$ con $F(i) = \sum_{j=1}^{i} f_j \,/\, \sum_{j=1}^{n} f_j$, el numero de fallos acumulado,

encontrados a lo largo del i-ésimo período, y $X_i = \ln(t_i)$.

Empezando con la función de distribución acumulada, se puede obtener la ecuación para una línea recta de la forma $Y = \alpha X + b$. Usando los puntos (X_i, Y_i), obtenidos a partir de los datos, los estimadores de mínimos cuadrados para la pendiente son fácilmente calculados. Los estimadores de α y β se calculan como la pendiente estimada y $\hat{\beta} = \exp(\hat{b})$ respectivamente.

5.5.2.2. Modelo de Duane

Originalmente propuesto para estudiar la fiabilidad del hardware, uno de los primeros modelos fue el modelo de Duane. Mientras trabajaba en General Electric, Duane se dio cuenta que si la tasa de fallo acumulativa era representada frente al tiempo acumulado de prueba, tendía a seguir una línea recta. Crow observo que este comportamiento podía ser representado como un proceso Weibull, es decir, la función de intensidad de fallo tiene la misma forma que la tasa de fallo.

Este mismo comportamiento ha sido observado para sistemas de software y ha sido utilizado para desarrollar varios estimadores de fiabilidad basados en este resultado. Este modelo es denominado algunas veces como el "modelo de fuerza", puesto que la función de valor medio para el numero de fallos acumulados en un tiempo t es tomado como fuerza de t, esto es, $\mu(t) = \alpha t^{\beta}$ para algún $\beta > 0$ y $\alpha > 0$ (En el caso en el que $\beta = 1$, se tratará de un modelo de proceso Poisson homogéneo). Este modelo es un modelo de fallo infinito pues $\lim_{t\to\infty} \mu(t) = \infty$.

La suposición básica, aparte de aquellas mencionadas en el aparatado del modelo Jelinski-Moranda es:

- El numero acumulado de fallos en un tiempo t, $M(t)$, sigue una proceso de Poisson con la función de valor medio $\mu(t) = \alpha t^{\beta}$ para algún $\beta > 0$ y $\alpha > 0$.

Para poder implementar este modelo, es necesario saber los instantes reales en los que el software falla, $t_1, t_2, \dots, t_n$, o bien los tiempos transcurridos entre fallos $x_1, x_2, \dots, x_n$, donde $x_i = t_i - t_{i-1}$ y $t_0 = 0$.

Forma del Modelo

Se trata de un proceso de Poisson con un valor medio de la función igual a $\mu(t) = \alpha t^{\beta}$. Si T es el tiempo total en el que el software es estudiado, entonces:

$$\frac{\mu(T)}{T} = \frac{\alpha T^{\beta}}{T} = \frac{numero\ de\ fallos\ esperados\ hasta\ T}{tiempo\ total\ de\ test}$$

Aplicando logaritmo a ambos lados de la igualdad, queda:

$$Y = \ln\left(\frac{\mu(T)}{T}\right) = \ln\left(\frac{\alpha T^{\beta}}{T}\right) = \ln(\alpha) + (\beta - 1)\ln(T)$$

Obtenido esto, la segunda ecuación puede ser representada como una línea recta.

La función de intensidad de fallo se obtiene derivando la función de valor medio, que es $\lambda(t) = d\mu(t)/dt = \alpha\beta T^{\beta-1}$. En esta función se comprueba que la función de intensidad de fallo es estrictamente creciente para $\beta > 1$, una constante en el caso de un proceso de Poisson homogéneo (($\beta = 1$), y estrictamente creciente para el caso en el que $0 < \beta < 1$. Puede ocurrir, que para el caso $\beta > 1$ no haya crecimiento de la fiabilidad.

Estimación y predicción de fiabilidad

Los estimadores de máxima verosimilitud se obtienen como:

$$\hat{\alpha} = \frac{n}{t_n^{\hat{\beta}}} \quad y \quad \hat{\beta} = \frac{n}{\sum_{i=1}^{n-1} \ln(\frac{t_n}{t_i})}$$

Estos estimadores de máxima verosimilitud son derivados para obtener las funciones de valor medio e intensidad de fallo $\mu(t)\ y\ \lambda(t)$, cambiando los parámetros $\alpha\ y\ \beta$ de acuerdo a su estimador de máxima verosimilitud (MLEs).

El estimador de máxima verosimilitud para el valor medio hasta el fallo (Mean Time To Failure, MTTF) es $\widehat{MTTF} = t_n/n\hat{\beta}$ para el fallo $n + 1$.

5.5.2.3. Modelo logarítmico de Musa-Okumoto

En este modelo, publicado en 1984 por Musa y Okumoto, también se considera que el proceso de fallo de un software es un proceso de Poisson no homogéneo, cuya función intensidad decrece exponencialmente a medida que se producen los fallos en un tiempo de ejecución τ. La tasa exponencial decreciente refleja que los fallos descubiertos antes tienen un mayor impacto en reducir la función de intensidad de fallo que aquellos encontrados más tarde. Se llama logarítmico debido a que el numero de fallos esperados a lo largo del tiempo es una función logarítmica

Incluyendo, al igual que otros modelos vistos, las asunciones estándar, las asunciones básicas de este modelo son:

- La intensidad de fallo decrece exponencialmente con el numero esperado de fallos experimentados, esto es, $\lambda(\tau) = \frac{d\mu(\tau)}{d\tau} = \lambda_0 \exp[-\theta\mu(\tau)]$, donde $\mu(\tau)$ es la función de

valor medio, $\theta > 0$ una constante de proporcionalidad que indica la disminución progresiva de la intensidad de fallo y λ_0 es la tasa inicial de fallo.

- El numero acumulado de fallos en el tiempo t, $M(t)$, es un proceso de Poisson.

La integración de la ecuación que aparece en la primera asunción conduce a:

$$\mu(\tau) = \frac{1}{\theta}\ln(\lambda_0\theta\tau + 1)$$

que es una función logarítmica del tiempo de ejecución τ. La intensidad del proceso es por tanto:

$$\lambda(\tau) = \frac{\lambda_0}{(\lambda_0\theta\tau + 1)}$$

Claramente, este es un modelo de fallos infinitos.

Los datos requeridos son tanto los tiempos reales en los que el software falla $t_1, t_2, \ldots, t_n$, o el tiempo transcurrido entre fallos $x_1, x_2, \ldots, x_n$, donde $x_i = t_i - t_{i-1}$.

Forma del Modelo

La representación de la función de intensidad de fallo frente al tiempo puede verse en la figura 5.3 Esta ilustra bien el comportamiento decreciente exponencial y el hecho de que los fallos encontrados al principio tienen un impacto mayor que los descubiertos más tarde. El parámetro θ controla la forma de la curva.

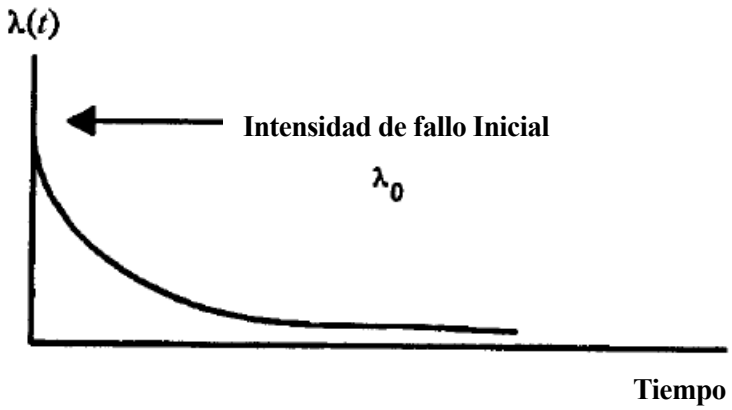

Figura 5.3. Función de la intensidad de fallo a lo largo del tiempo

Según la ecuación del valor medio del número de fallos, se deduce que para un tiempo de ejecución infinito el número de fallos podría ser infinito, sin embargo no es real que un software pueda contener un número infinito de defectos. El modelo supone que un número finito de defectos puede originar un número infinito de fallos en un tiempo infinito, debido a que un defecto puede originar más de un fallo, y que cuando se corrige el defecto que ocasiona un fallo cabe la posibilidad de no

eliminarlo en su totalidad.

Estimación y predicción de fiabilidad

Empleando el modelo reparametrizado, los estimadores de máxima verosimilitud para β_0 y β_1 según Musa, son la solución al siguiente par de ecuaciones:

$$\hat{\beta}_0 = \frac{n}{\ln(1 + \hat{\beta}_1 t_n)}$$

$$\frac{1}{\hat{\beta}_1} \sum_{i=1}^{n} \frac{1}{1 + \hat{\beta}_1 t_n} = \frac{n t_n}{1 + \hat{\beta}_1 t_n \ln(1 + \hat{\beta}_1 t_n)}$$

Este modelo es especialmente aplicable cuando se sabe con antelación que el uso operacional del programa no será uniforme. La no uniformidad tiende a hacer que los fallos encontrados antes tengan un impacto mayor que los fallos encontrados más tarde (como se ve en la figura 5.3).

5.5.3. Modelos Bayesianos

Todos los modelos presentados hasta ahora asumen que los datos de fallos están disponibles para su estudio. Aplican también todos ellos técnicas estadísticas clásicas como el estimador de máxima verosimilitud, donde los parámetros del modelo son fijos pero desconocidos y son estimados a partir de los datos disponibles. El inconveniente de este enfoque es que los parámetros del modelo no pueden ser estimados cuando los datos de fallos no están disponibles. Incluso cuando hay unos pocos datos disponibles, las técnicas basadas en el estimador de máxima verosimilitud no son muy fiables puesto que pueden ofrecer estimaciones incorrectas.

Este grupo de modelos considera el desarrollo de la fiabilidad y su predicción desde un marco Bayesiano, en vez del planteamiento tradicional que presentan los modelos anteriores. Tales modelos permiten un cambio en la fiabilidad solo cuando se produce un error. La mayoría de ellos también ven el impacto de cada defecto como si fuera de la misma magnitud. Un modelo Bayesiano, por su parte, toma un punto de vista subjetivo en el que si no se produce ningún fallo mientras el software es observado, entonces la fiabilidad debería aumentar, reflejando el aumento de confianza en el software por parte del usuario. La fiabilidad es por lo tanto, un reflejo tanto del numero de defectos que son detectados y la cantidad de operación que se realiza sin que aparezcan fallos. Este reflejo es expresado en términos de una distribución a priori que representa los datos pasados y una a posteriori, que incorpora a estos últimos los datos actuales.

Los modelos Bayesianos también plasman la creencia en que distintos tipos de defectos tienen

distintos impactos en la fiabilidad del programa. El numero de defectos no es tan importante como su impacto. Nos surge la pregunta: sea un programa que tiene un determinado número de defectos en una parte del software que apenas es utilizada, ¿es menos fiable este programa que otro que solo tiene un defecto en la parte de código que es utilizada normalmente? El enfoque bayesiano respondería negativamente a esta cuestión, argumentando que es mas importante estudiar el comportamiento del software que estimar el número de defectos en él. La media de tiempo en fallar sería pues, un estadístico muy importante cuando se sigue este planteamiento.

Los datos históricos, a partir de los que se obtiene la distribución a priori, son parte esencial de esta metodología, pues reflejan la idea de que se debe incorporar información pasada, por ejemplo proyectos de naturaleza similar, a la hora de estimar la fiabilidad del presente y futuro. Esta distribución es simultáneamente una fortaleza y una debilidad de los modelos Bayesianos. Es correcto incluir en nuestro análisis lo que ha pasado anteriormente, la cuestión es de qué forma.

5.5.3.1. Modelo logarítmico de Littlewood-Verrall

El modelo de Littlewood-Verrall es probablemente el mejor ejemplo de modelos de tipo bayesiano. Es también el único recomendado en el caso de que se busque un candidato inicial de entre los modelos bayesianos que se adapte a nuestros datos.

El modelo trata de representar un modelo de generación de defectos en el proceso de corrección de los mismos, permitiendo la posibilidad de que el programa software se convierta en menos fiable que antes. Con cada corrección de defectos, una secuencia de programas software es generada. Cada una es obtenida de su predecesor, intentando arreglar el defecto. Debido a la incertidumbre, la nueva versión podría ser mejor o peor que su predecesora, por lo que se incluye así una nueva fuente de variación. Esto queda reflejado en los parámetros que definen las distribuciones de los tiempos de fallos, que son aleatorias. La distribución de tiempos de fallos es, como en los modelos anteriores, considerada exponencial con una determinada tasa de fallo, pero, y aquí se encuentra la diferencia con los otros modelos, es esta tasa de fallo la considerada aleatoria y no constante como anteriormente. La tasa de fallos correspondiente a cada defecto ϕ_i al comenzar el proceso de pruebas del sistema tiene una distribución gamma -G- de parámetros α y β o distribución "a priori".

Las asunciones básicas de este modelo son:

- Los sucesivos tiempos de ejecución entre fallos, X_i, son considerados variables aleatorias exponencialmente independientes, con parámetros $\xi_i, i = 1, \dots, n$.

- Los parámetros ξ_i forman una secuencia de variables aleatorias independientes, cada una con una distribución gamma de parámetros α y $\psi(i)$. Este último parámetro es una función creciente de i que describe la calidad del parámetro y la dificultad de la tarea, de tal forma que un buen programador tendría una función con un crecimiento más rápido que uno malo.

Siempre que la función ψ sea creciente, se cumple la condición $P\{\xi_j < L\} \geq P\{\xi_{j-1} < L\}\ \forall j$. Esto refleja la la intención de hacer que el programa mejore tras el descubrimiento y la corrección de un defecto, pero esto es algo que no puede ser asegurado. Los datos requeridos son los tiempos entre ocurrencia de fallos, x_i.

Forma del modelo

Para calcular la distribución a posteriori, primero se necesita la distribución marginal de los tiempos entre ocurrencia de fallos, x_i. La distribución a priori tiene la forma:

$$g(\xi_i|\psi(i),\alpha) = \frac{\psi(i)^\alpha}{\Gamma(\alpha)}\xi_i{}^{\alpha-1}e^{-\psi(i)\xi_i};\quad \xi_i \geq 0$$

La distribución marginal, ayudándonos de la distribución a priori, queda:

$$f\left(x_i\middle|\alpha,\psi(i)\right) = \frac{\alpha[\psi(i)]^\alpha}{[x_i+\psi(i)]^{\alpha+1}}\ \ para\ x_i > 0$$

La distribución a posteriori para ξ_i es obtenida entonces:

$$h(\xi_1,\xi_2,\dots,\xi_n) = \frac{\prod_{i=1}^n \xi_i^\alpha \exp(-\sum_{i=1}^n \xi_i(x_i+\psi(i)))}{[\Gamma(\alpha+1)]^n \sum_{i=1}^n (x_i+\psi(i))^{\alpha+1}}\ para\ \xi_i > 0, i = 1,\dots,n$$

Littlewood y Verrall sugieren una forma lineal y cuadrática para la función $\psi(i)$,

$$\psi(i) = \beta_0 + \beta_1 i\ (lineal)$$

$$\psi(i) = \beta_0 + \beta_1 i^{\,2}(cuadrática)$$

Las funciones de intensidad de fallo para estas dos formas puede ser calculada de la siguiente forma:

$$\lambda_{lineal}(t) = \frac{\alpha-1}{\sqrt{\beta_0^2 + 2\beta_1 t(\alpha-1)}}$$

$$\lambda_{cuadrática}(t) = \frac{v_1}{\sqrt{t^2+v_2}}\left[\left(t+(t^2+v_2)^{\frac{1}{2}}\right)^{\frac{1}{3}} - \left(t-(t^2+v_2)^{\frac{1}{2}}\right)^{\frac{1}{3}}\right]$$

dónde $v_1 = ((\alpha-1)/18\beta_1)^{1/3}$ y $v_2 = 4\beta_0^3/(9(\alpha-1)^2\beta_1)$

Estimación y predicción de fiabilidad

Usando la función de distribución marginal, los estimadores de máxima verosimilitud de $\alpha, \beta_0 \; y \; \beta_1$ son las soluciones al siguiente sistema de ecuaciones:

$$\frac{n}{\hat{\alpha}} + \sum_{i=1}^{n} \log\left(\hat{\psi}(i)\right) - \sum_{i=1}^{n} \log\left(x_i + \hat{\psi}(i)\right) = 0,$$

$$\hat{\alpha} \sum_{i=1}^{n} \frac{1}{\hat{\psi}(i)} - (\hat{\alpha} + 1) \sum_{i=1}^{n} \frac{1}{x_i + \hat{\psi}(i)} = 0$$

$$\hat{\alpha} \sum_{i=1}^{n} \frac{\mathrm{i}}{\hat{\psi}(i)} - (\hat{\alpha} + 1) \sum_{i=1}^{n} \frac{\mathrm{i}'}{x_i + \hat{\psi}(i)} = 0$$

donde $\psi(i) = \beta_0 + \beta_1 i'$, pudiendo ser i', tanto i como i^2.

Comparativa entre modelos

La tabla que se detalla a continuación es un esquema que recoge las principales funciones de fiabildiad para cada modelo estudiado.

Tabla 5.1. Funciones principales de los modelos estudiados

	Modelo	Intensidad de fallos	Función de fiabilidad	MTTF
Número finito de fallos	Jelinski-Moranda	$\lambda_0(n_0 - k)$	$\exp[\lambda_0(n_0 - k)t']$	$\frac{1}{\lambda_0(n_0 - k)}$
	Goel-Okumoto	$n_0\lambda_0\exp(-\lambda_0 t)$	$\exp[-n_0(e^{-\lambda_0 t_k} - e^{\lambda_0 t''})]$	Indefinido
	Littlewood	$\Lambda = \phi_i + \cdots + \phi_{n_0-k}$	$\left[\frac{\beta + t_k}{\beta + t_k + t'}\right]^{(n_0-k)\alpha}$	$\frac{(\beta + t_k)^{(n_0-k)\alpha}}{(n_0 - k)\alpha + 1}$
	Duane	$2\lambda_0(n_0 - k)t$	$\exp[\lambda_0(n_0 - k)t'^2]$	$\frac{\sqrt{\pi}}{2\lambda_0(n_0 - k)}$
Numero infinito de fallos	Musa-Okumoto	$\frac{\lambda_0}{\lambda_0\theta\tau + 1}$	$\left[\frac{\lambda_0\theta\tau + 1}{\lambda_0\theta(\tau_k + \tau') + 1}\right]^{\frac{1}{\theta}}$	$\frac{1}{1 - \theta\lambda_0}(1 + \lambda_0\theta\tau_k$

Elaboración. Propia

Para finalizar, se incluye una tabla con las ventajas e inconvenientes de cada modelo presentado

Tabla 5.2. Ventajas e inconvenientes de los modelos estudiados

Modelo de Estimación	Ventajas	Inconvenientes
Modelo Jelinski-Moranda	Facilidad de cálculo	Parte de hipótesis paramétricas que no tienen porqué cumplirse
Modelo Goel-Okumoto	Contempla que los fallos pueden causar otros fallos lo que es mucho más real	Muy sensible a las desviaciones en las hipótesis
Modelo Littlewood-Verrall	Se ajusta a cambios de tamaño del programa	Muy costoso en tiempo calcular todos los parámetros
Modelo Duane	Se puede aplicar en diferentes fases del ciclo de vida del proyecto	Si las hipótesis no se cumplen provoca grandes desviaciones en los resultados
Modelo Musa-Okumoto	Facilidad de uso	Falla en programas muy grandes

Elaboración. Propia

6 ANÁLISIS DE LA FIABILIDAD DEL SOFTWARE

6.1. Introducción

Una vez se han visto algunos de los modelos teóricos más importantes en análisis de fiabilidad del software el siguiente paso es poder calcular dichos modelos, para lo que es conveniente emplear métodos informáticos que faciliten su cálculo.

Las técnicas de fiabilidad del software se utilizan para estimar la tasa de fallos y la fiabilidad de una determinada aplicación informática. La necesidad de este tipo de medidas se enmarca dentro del objetivo de cuantificar la confiabilidad de un sistema software, que hace referencia a la "calidad del servicio prestado" de forma que se pueda confiar de una forma justificada en su servicio. La confiabilidad no se mide directamente, sino a través de sus atributos, los cuales, expresados desde una perspectiva RAMS (por sus siglas en inglés) son: fiabilidad, disponibilidad, mantenimiento y seguridad.

<table>
<tr><td colspan="4">Confiabilidad</td></tr>
<tr><td>R</td><td>A</td><td>M</td><td>S</td></tr>
<tr><td>Fiabilidad
(Reliability)</td><td>Disponibilidad
(Availability)</td><td>Mantenimiento
(Maintainability)</td><td>Seguridad
(Safety)</td></tr>
<tr><td colspan="3">Continuidad de servicio</td><td>Sin accidentes</td></tr>
</table>

Figura 6.1. Atributos de la confiabilidad (se detalla la expresión inglesa para cada uno de los atributos)

Una medida de confiabilidad importante es la tasa de fallos (esto es el número de fallos por unidad de tiempo) que sirve para evaluar la frecuencia de fallos de un sistema tal y como es percibida por el usuario.

Como se ha expresado anteriormente, el objetivo de la fase de pruebas de un software es detectar y corregir en un tiempo razonable los defectos que pueda contener el sistema. Para ello, lo que se hace

es permitir su funcionamiento hasta que se produzca un fallo del sistema. Una vez interrumpido el software se intenta detectar y corregir el defecto que ha originado el fallo. Cuando se han llevado a cabo todas las acciones posibles para corregir el defecto se vuelve a poner en marcha el sistema, repitiéndose este proceso hasta dar por concluidas las pruebas.

En consecuencia, el número de defectos que contiene el sistema va variando durante el período de tiempo correspondiente a las pruebas. Su valor en un instante determinado depende del número de fallos que se hayan producido, ya que el número de defectos varía como consecuencia de la ocurrencia de fallos; pero también depende del resultado de las acciones tomadas para detectar y corregir los defectos que ocasiona cada fallo.

6.2. Introducción al entorno R

R es un lenguaje y entorno de programación para análisis estadístico y gráfico. Es un proyecto de software libre enmarcado dentro del proyecto GNU. En su origen, R es una implementación, con ciertas modificaciones, de un lenguaje estadístico anterior, denominado S y que fue desarrollado en los Laboratorios Bell por John Chambers y sus colaboradores. Aunque hay algunas diferencias importantes, la mayor parte de código escrito para S funciona con normalidad al emplear R.

Al contrario de S, R es de libre distribución y ello ha contribuido a su enorme desarrollo. La comunidad que emplea R es muy dinámica y el número de nuevos y mejores paquetes crece exponencialmente.

R ofrece una gran variedad de técnicas gráficas y estadísticas (modelos lineales y no lineales, análisis de series temporales, test estadísticos clásicos, algoritmos de clasificación, etc.) y puede ser altamente extensible por los usuarios. El lenguaje S es a menudo el vehículo elegido para la investigación en metodología estadística y R ofrece una ruta de fuente abierta para la participación en dicha actividad. Una de las fortalezas de R es su capacidad gráfica, pues permite generar gráficos de alta calidad.

6.2.1 El entorno R

R es un juego integrado de servicios de software para la manipulación de datos, cálculo y representación gráfica.

Incluye:

- Servicio efectivo de manipulación y almacenaje de datos.

- Conjunto de operadores para calcular en matrices.
- Gran colección coherente e integrada de herramientas intermedias para el análisis de datos.
- Servicio gráfico para el análisis y la representación de datos en pantalla o en papel.
- Desarrollado lenguaje de programación simple y efectivo.

El término "entorno" pretende ser caracterizado como un sistema totalmente planeado y coherente, en lugar de un aumento progresivo de herramientas muy específicas e inflexibles, como frecuentemente ocurre con otras aplicaciones de análisis de datos.

R, como S, permite a los usuarios añadir funciones adicionales mediante por medio de su definición. Gran parte del sistema está escrito en el dialecto R de S, lo que hace fácil a los usuarios seguir las elecciones hechas por el algoritmo. Para tareas más intensas computacionalmente, los lenguajes C/C++ o Fortran pueden ser conectados y llamados mientras se esta ejecutando el código.

Muchos usuarios creen que R es un sistema estadístico pero es más correcto pensar en R como un entorno dentro del cual las técnicas estadísticas son implementadas. R puede ser extendido fácilmente mediante los llamados *"packages"* o paquetes, que pueden ser obtenidos en la página web de CRAN *("The Comprehensive R Archive Network")*.

A continuación se detalla como se traducen en R algunos de los modelos estudiados en este trabajo.

Duane en "R"

Este modelo se recoge en el paquete *"Reliability"* del software R, cuyo autor es Andreas Wittmann.

Así, la función *"duane"* tiene por objetivo calcular el estimador de máxima verosimilitud de la función de valor medio para el modelo Duane. Se calcula para los parámetros $\hat{\alpha}\ y\ \hat{\beta}$ que se han expuesto anteriormente. Sin embargo, el autor los denomina rho y theta. Es por ello que se denominarán así a la hora de llevar a cabo el análisis.

La forma en que se usará esta función ha de seguir la siguiente estructura.

$$duane(t, init = c(1,1), method = \text{Nelder-Mead}, maxit = 10000, \dots)$$

donde:

t	dato de tiempo entre fallos
$init$	valor inicial para el estimador de máxima verosimilitud fijado en el valor medio de la función para el modelo Duane
$method$	el método empleado para encontrar una solución óptima
$maxit$	el número máximo de iteraciones que se realizarán

... parámetros opcionales de control y de representación.

El propósito del modelo es minimizar la diferencia entre los parámetros rho y theta en sus respectivas ecuaciones, esto es:

$$ecuación_1 = \rho - \frac{n}{t_n^{\theta}} = 0$$

$$ecuación_2 = \theta - \frac{n}{\sum_{i=1}^{n-1} \ln\left(\frac{t_n}{t_i}\right)} = 0$$

donde t es el tiempo entre fallos (dato) y n es la longitud o, en otras palabras, el tamaño de los datos de tiempo entre fallos. La minimización simultánea de estas ecuaciones tiene lugar al minimizar la ecuación

$$ecuación_1^2 + ecuación_2^2 = 0$$

Al aplicar esta función, se obtienen los siguientes resultados:

rho	Estimador de máxima verosimilitud para rho
$theta$	Estimador de máxima verosimilitud para theta

Moranda-Geométrico en "R"

La función computa el estimador de máxima verosimilitud para el parámetro D y $theta$ de la función de valor medio para el modelo Moranda-Geométrico.

$$moranda.geometric(t, init = c(0{,}1), tol = .Machine\$double.eps\text{^}0.25)$$

donde:

t	tiempo entre fallos
$init$	valores iniciales de máxima verosimilitud adecuados para la función de valor medio del modelo de Moranda-Geométrico
tol	exactitud deseada

Esta función estima los parámetros D y $theta$ de la función de valor medio para el modelo de Moranda-Geométrico. Con el método EMV, se llega a la siguiente ecuación, que debe ser minimizada para obtener phi.

Esta expresión es:

$$\frac{\sum_{i=1}^{n} i\phi^i t_i}{\sum_{i=1}^{n} \phi^i t_i} - \frac{n+1}{2} = 0$$

La solución obtenida debe ser introducida en la siguiente ecuación para hallar D

$$D = \frac{\phi n}{\sum_{i=1}^{n} \phi^i t_i}$$

donde t es el tiempo entre fallos y n es la longitud, o en otras palabras el tamaño de los datos de tiempos entre fallos.

Littlewood Verrall en "R"

La función littlewood.verrall calcula el EMV para los parámetros $theta\theta, theta1$ y rho de la función de valor medio para el modelo de Littlewood-Verrall.

La forma en que se usará esta función ha de seguir la siguiente estructura.

$littlewood.verrall(t, linear = T, init = c(1,1,1),\ method =$ Nelder-Mead, $maxit = 10000, \dots)$

donde:

t	dato de tiempo entre fallos.
$linear$	lógico. ¿Debería la linealidad de la forma cuadrática de la función de valor medio para el modelo Littlewood-Verrall ser usada en el cálculo? Si es verdad (caso estándar), la forma lineal de la función de valor medio es utilizada.
$init$	valor inicial para el estimador de máxima verosimilitud fijado en el valor medio de la función para el modelo Littlewood-Verral.
$method$	el método empleado para encontrar una solución óptima
$maxit$	el número máximo de iteraciones que se realizarán
...	parámetros opcionales de control y de representación.

Nota:

Esta función estima los parámetros $theta\theta, theta1$ y rho de la función de valor medio para el modelo de Littlewood-Verrall.

En primer lugar, se explica el cálculo de la forma lineal. Con el estimador de máxima verosimilitud se obtienen la siguientes ecuaciones, que deben ser minimizadas.

$$equation_1 = \frac{n}{p} + \sum_{i=1}^{n} \log(\theta_0 + \theta_1 i) - \sum_{i=1}^{n} \log(\theta_0 + \theta_1 i + t_i) = 0,$$

$$equation_2 = \rho \sum_{i=1}^{n} \frac{1}{\theta_0 + \theta_1 i} - \rho + 1 \sum_{i=1}^{n} \frac{1}{\theta_0 + \theta_1 i + t_i} = 0$$

y

$$equation_3 = \rho \sum_{i=1}^{n} \frac{\mathrm{i}}{\theta_0 + \theta_1 i} - \rho + 1 \sum_{i=1}^{n} \frac{\mathrm{i}}{\theta_0 + \theta_1 i + t_i} = 0$$

Donde t es el dato de tiempo entre fallos y n es la longitud, o en otras palabras la magnitud del tiempo entre fallos. Así, la minimización simultanea de estas tres ecuaciones se da con la minimización de la ecuación

$$equation_1^2 + equation_2^2 + equation_3^2 = 0.$$

Y con un resultado para cada variable:

$theta\theta$	Estimador de máxima verosimilitud para $theta\theta$
$theta1$	Estimador de máxima verosimilitud para $theta1$
rho	Estimador de máxima verosimilitud para rho

La forma cuadrática sigue el mismo procedimiento con la salvedad de que las ecuaciones que emplea son estas que se detallan:

$$equation_1 = \frac{n}{p} + \sum_{i=1}^{n} \log(\theta_0 + \theta_1 i^2) - \sum_{i=1}^{n} \log(\theta_0 + \theta_1 i^2 + t_i) = 0$$

$$equation_2 = \rho \sum_{i=1}^{n} \frac{1}{\theta_0 + \theta_1 i^2} - \rho + 1 \sum_{i=1}^{n} \frac{1}{\theta_0 + \theta_1 i^2 + t_i} = 0$$

y

$$equation_3 = \rho \sum_{i=1}^{n} \frac{i^2}{\theta_0 + \theta_1 i^2} - \rho + 1 \sum_{i=1}^{n} \frac{\mathrm{i}}{\theta_0 + \theta_1 i^2 + t_i} = 0$$

Modelos con tiempos de fallo infinitos

Para este tipo de modelos, el límite $\lim_{t \to \infty} \mu(t)$ es ∞ para la función de valor medio del proceso. Esto significa que el software nunca estará completamente libre de fallos. Esto podría ser causado por la introducción de nuevos errores en el software en el proceso de corrección de que aquellos ya existentes o por la imposibilidad de depurar por completo la aplicación, entre otros.

Musa-Okumoto en "R"

La función *musa.okumoto* computa el EMV para los parámetros parámetros $theta\theta$ y $theta1$

de la función de valor medio del Modelo de Musa-Okumoto.

La forma en que se usará esta función ha de seguir la siguiente estructura.

$$musa.okumoto(t, init = c(0,1), tol = .\text{Machine\$double.eps}^{0.25})$$

donde:

- t dato de tiempo entre fallos
- $init$ valor inicial para el estimador de máxima verosimilitud fijado en el valor medio de la función para el modelo Musa-Okumoto.
- tol precisión deseada

Internamente, lo que se lleva a cabo es la minimización de la siguiente ecuación, con el fin de obtener el parámetro $theta1$.

$$\frac{1}{\theta_1}\sum_{i=1}^{n}\frac{1}{1+\theta_1 t_i} - \frac{nt_n}{(1+\theta_1 t_n)\log(1+\theta_1 t_n)} = 0$$

Una vez resuelta esta expresión, se introduce la solución en la siguiente ecuación con el fin de obtener $theta\theta$

$$\theta_0 = \frac{n}{\log(1+\theta_1 t_n)}$$

donde t es el tiempo entre fallos y n es la longitud o tamaño de los datos de tiempos de fallos.

Al aplicar esta función se dispondrá de los estimadores de máxima verosimilitud para ambos parámetros.

6.3. Aplicación práctica

Para finalizar el análisis de la fiabilidad del software se realiza una aplicación práctica de los modelos recogidos en el paquete "Reliability" de R.

La utilización de un modelo matemático para el análisis de la evolución de la fiabilidad de un software durante su etapa de pruebas exige una minuciosa recogida de datos relativos a la realización de las variables aleatorias para posteriormente, mediante algún modelo de inferencia estadística poder estimar los parámetros del modelo. Una vez determinados todos los parámetros, se sustituyen sus valores en las ecuaciones del modelo y se calculan las cantidades buscadas.

Los cuatro modelos que se van a aplicar requieren una muestra que incluya para cada fallo k el instante de tiempo en el que se ha producido el fallo del sistema, tomando como origen de tiempo el

fallo anterior.

La recogida de este tipo de datos no plantea más problemas que el de mantener una constancia en su registro. Sin embargo, conseguir registros sistemáticos de tiempos de fallo es difícil debido fundamentalmente a la falta de consideración de las mejoras que se pueden obtener mediante la aplicación de este tipo de metodologías durante las pruebas del software. Tras una intensa búsqueda, se ha encontrado un conjunto de datos que será objeto de nuestro análisis. Pertenecen a una gran compañía de desarrollo de software en España.

A modo de apunte, se describen las dificultades encontradas en la aplicación práctica de las técnicas de Fiabilidad. El entorno de trabajo fue la fase de pruebas de integración de dos proyectos de sistemas, previamente a su paso a producción. La dificultad de la aplicación de estas técnicas residió fundamentalmente en los siguientes tres factores:

- El gran dinamismo de las actividades comerciales de la compañía. Esto implica que de forma muy frecuente y rápida se lanzan al mercado nuevos productos o ampliaciones de los ya existentes, de forma que los proyectos bajo prueba no solían ser demasiado grandes y, por tanto, los tiempos planificados para llevar a cabo las pruebas eran extremadamente ajustados y los equipos e prueba solían trabajar en varios proyectos de forma simultánea. La consecuencia directa era el riesgo de disponer de pocos datos para realizar el estudio.
- La complejidad de la estructura de proyectos. Normalmente eran varios los equipos involucrados en las pruebas. De hecho, teóricamente había una metodología de pruebas común, pero en la práctica cada equipo era dirigido de manera independiente por un jefe de equipo con sus propios conocimientos, experiencias y forma de trabajo. Además, cuanto mayor era el número de sistemas implicados en el proyecto, más difícil resultaba conocer por ejemplo la cobertura funcional de las pruebas, el grado de impacto del fallo de un sistema en el resto, el tiempo de ejecución de un caso de prueba o la fiabilidad global.
- La dificultad para recoger los datos de prueba relacionados con el tiempo de uso del sistema. Este hecho implica un alto riesgo de imprecisión en los resultados del análisis, ya que las dos principales claves del éxito en el uso de las técnicas de crecimiento de fiabilidad son por una parte, realizar las pruebas en un entorno lo más similar posible al de producción, contando con un diseño de casos de prueba que ejerciten el sistema en la forma más probable a ser usada por el usuario y por otra parte, recoger los tiempos exactos de ejecución del sistema.

Las dificultades se siguieron encontrando en la escasez de datos proporcionados por los proyectos con un tiempo de pruebas corto. De hecho, por regla general, los proyectos que no estuvieran, al menos, dos semanas bajo prueba, no tenían posibilidad de estimación alguna y, normalmente aquellos que no estuvieran al menos cuatro semanas no aseguraban una bondad de ajuste con un nivel de confianza suficientemente alto. Una de las posibles soluciones a este problema sería utilizar técnicas de estadística Bayesiana para la estimación de los parámetros de los modelos, ya que permiten obtener resultados con un conjunto de datos menor que el utilizado por las técnicas convencionales de inferencia estadística.

Los datos fueron recogidos de la forma más automatizada posible, minimizando en lo posible el factor humano, que es una fuente muy probable de introducción de imprecisiones. Al término de este proceso se disponía de un conjunto datos conocidos que aportaban el tiempo entre las ocurrencias de los fallos (TBF).

6.3.1. Conjunto de datos

Este conjunto de datos fue recogido por la empresa Métodos y Tecnología de Sistemas y Procesos (MTP) y son datos de tiempos de fallos obtenidos de una aplicación software de control y comando. El tamaño del software era de aproximadamente 21.700 instrucciones y fue desarrollado por una empresa del sector de las telecomunicaciones española. Hicieron falta 21 semanas y 9 programadores para completar el test. Durante la fase de prueba, se consumieron 25,3 horas de CPU y fueron eliminados 136 defectos.

Estos son los tiempos entre fallos obtenidos:

```
t <- c(3, 30, 113, 81, 115, 9, 2, 20, 20, 15, 138, 50, 77, 24, 108, 88,
670, 120, 26, 114, 325, 55, 242, 68, 422, 180, 10, 1146, 600, 15, 36, 4,
0, 8, 227, 65, 176, 58, 457, 300, 97, 263, 452, 255, 197, 193, 6, 79,
816, 1351, 148, 21, 233, 134, 357, 193, 236, 31, 369, 748, 0, 232, 330,
365, 1222, 543, 10, 16, 529, 379, 44, 129, 810, 290, 300, 529, 281, 160,
828, 1011, 445, 296, 1755, 1064, 1783, 860, 983, 707, 33, 868, 724, 2323,
2930, 1461, 843, 12, 261, 1800, 865, 1435, 30, 143, 108, 0, 3110, 1247,
943,700, 875, 245, 729, 1897, 447, 386, 446, 122, 990, 948, 1082, 22, 75,
482, 5509, 100, 10, 1071, 371, 790, 6150, 3321, 1045, 648, 5485, 1160,
1864, 4116)
```

Empleando el software R, se verá como son las gráficas que se generan tras la aplicación de los distintos modelos vistos.

Modelo de Duane

La gráfica de la figura 6.2 muestra el número acumulado de fallos por tiempo acumulado.

La pendiente de la curva que representa el número total de fallos encontrados es bastante pronunciada hasta que se produce el sexagésimo fallo. Es también durante ese primer período de tiempo cuando el modelo representa mejor el comportamiento real. Una vez alcanzado este punto y según el modelo, el ritmo de crecimiento ira decreciendo hasta hacerse nulo en el momento en el que se hayan encontrado todos los fallos esperados. La gráfica real queda por encima de la que se obtiene con el modelo por lo que se puede hablar de que es una estimación optimista.

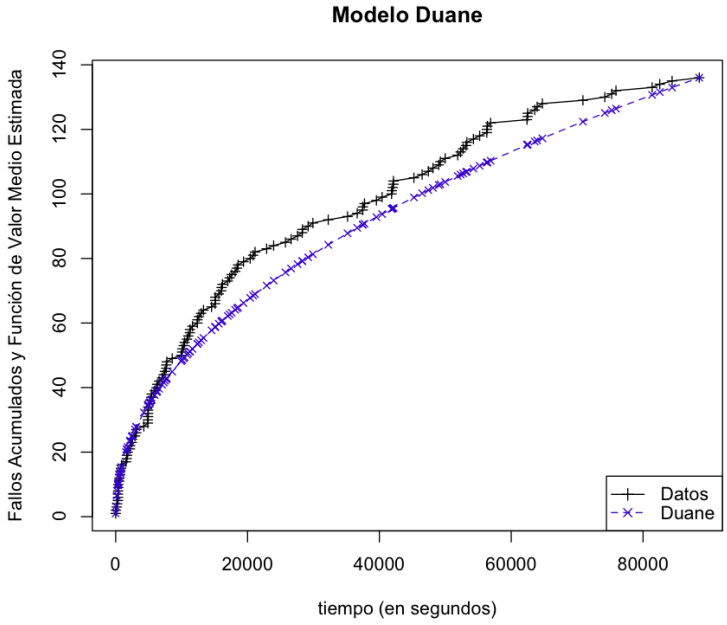

Figura 6.2. Modelo Duane. Gráfico acumulado de fallos por tiempo acumulado

Con el método EMV, los estimadores `rho` y `theta` son:

- `rho = 0.6122738`
- `theta = 0.4743446`

Prestando atención al código ejecutado en `R`, es posible cambiar el método de optimización que se sigue para llegar a estos valores.

```
"duane" <- function(t, init = c(1, 1), method = "Nelder-Mead", maxit =
 10000, ...)
```

El método que se sigue por defecto es el que se recoge en las líneas de código anteriores, una implementación del Nelder y Mead (1965), un método robusto pero relativamente lento. Funcionará razonablemente bien realizando los cálculos que se requieren.

Sin embargo, hay también otros métodos:

- Método “BFGS” es un método alternativo al de Newton, de ahí que reciba el nombre de

Quasi-Newton. Publicado en 1979 por Broyden, Fletcher, Goldfarb y Shanno, construye una superficie que es optimizada.

- Método "CG" es un método de gradiente conjugado, esto es, un algoritmo para resolver problemas de optimización sin restricciones. Estos métodos son generalmente más frágiles que los métodos BFGS, pero en aquellos problemas más grandes pueden ser exitosos.

Se puede profundizar más sobre estos métodos en Nocedal and Wright (1999).

A continuación se recogen en una tabla los valores obtenidos para los parámetros rho y $theta$.

Tabla 6.1. Valores de los parámetros rho y theta para distintos métodos de optimización

	Nelder y Mead	*BFGS*	*CG*
rho	0,6122738	0.6107038	0.6107005
theta	0,4743446	0.4745686	0.4745691

Como se puede apreciar, la diferencia en los distintos resultados es mínima, por lo que hay razones para pensar que el valor obtenido se acerca mucho al óptimo. Gráficamente, la diferencia entre las distintas gráficas correspondientes a cada modelo es inapreciable.

Modelo Moranda-Geométrico

La gráfica 6.3 muestra el número acumulado de fallos por tiempo acumulado, así como la estimación del modelo Moranda-Geométrico.

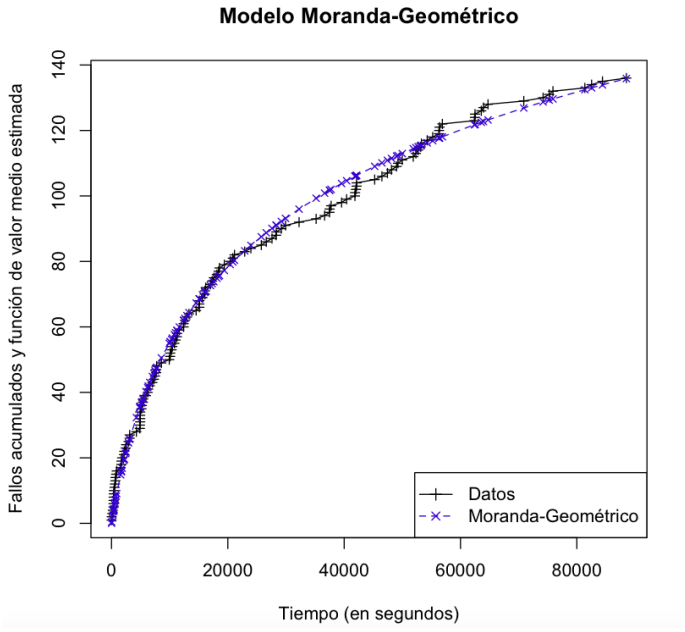

Figura 6.3. Modelo Moranda-Geométrico. Gráfico acumulado de fallos por tiempo acumulado

Comparando con la gráfica anterior, se puede ver que este modelo proporciona una mejor

estimación, pues la diferencia entre las dos curvas, esto es el error, es menor.

En este caso, lo que permite la función es reajustar la exactitud deseada, no el modelo de optimización seguido, como ocurría en el caso anterior. Como se vió al estudiar el modelo, la estructura de la función es:

```
moranda.geometric" <- function(t, init = c(0, 1), tol = .Machine$double.eps^0.25
```

Aparece en negrita el valor de la exactitud que se espera del modelo. ".Machine" es una variable que contiene información sobre las características numéricas de la máquina que está ejecutando el programa `R`, como el tamaño o longitud de palabra, medido en bits, o la precisión de la misma.

`double.eps` implica que se toma como valor de x el valor tipo float (decimal) positivo y más pequeño tal que $1 + x! = 1$. Normalmente este valor es $2.220446 \cdot e^{-16}$

En la tabla 6.2 se recogen los diferentes valores de rho y $theta$ que se generan para distintos valores de exactitud.

Tabla 6.2. Valores de los parámetros rho y theta para distintos niveles de exactitud

	.Machine$double.eps^0.25	*0.1*	*1*
rho	0.6122738	0.6122738	0.6122738
theta	0.02360818	0.02360818	0.02360818

Como se puede observar, aumentar o disminuir el valor de exactitud deseada no trae consecuencias a las estimaciones de los parámetros rho y $theta$. Sin embargo, tras ejecutar la función para representar el modelo se comprueba que sí influye este valor de exactitud. Cuando toma valor unitario, la gráfica queda de la siguiente forma:

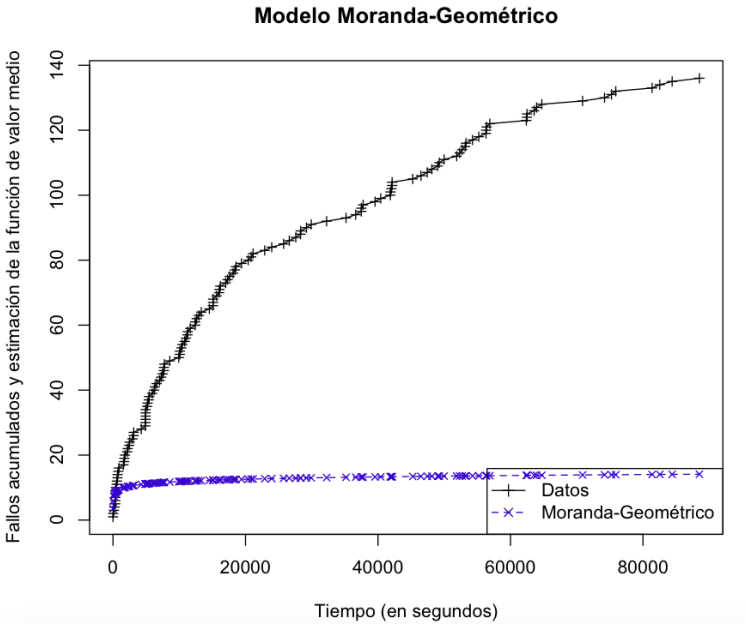

Figura 6.4. Modelo Moranda-Geométrico con nivel de exactitud muy pequeño.

Se evidencia que al ser 1 mucho mayor que el valor incluido en el modelo por defecto, la exactitud del mismo será mucho menor. Esto se ve en la gráfica midiendo la diferencia de las dos curvas: el error del primer caso es mínimo mientras que el de los dos casos con menor exactitud es relativamente grande.

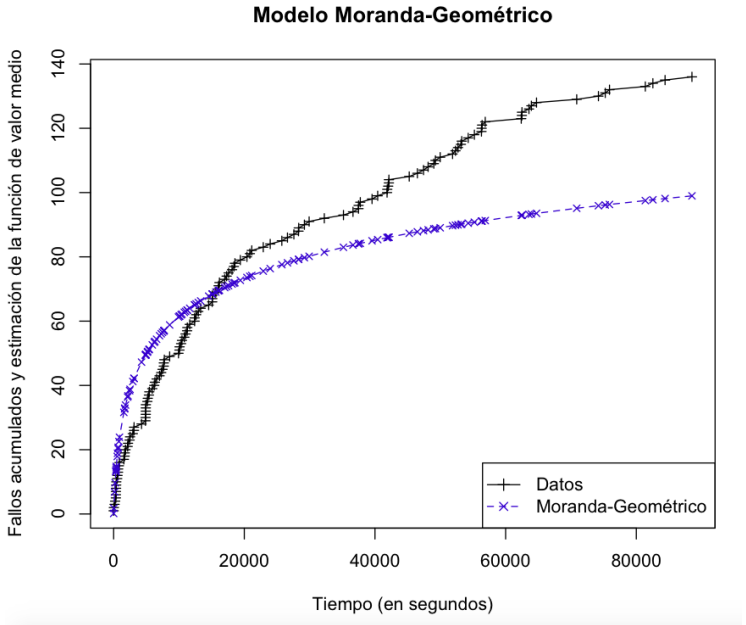

Figura 6.5. Modelo Moranda-Geométrico para un nivel de exactitud pequeño

A continuación se pondrá de manifiesto lo que se comentaba del error

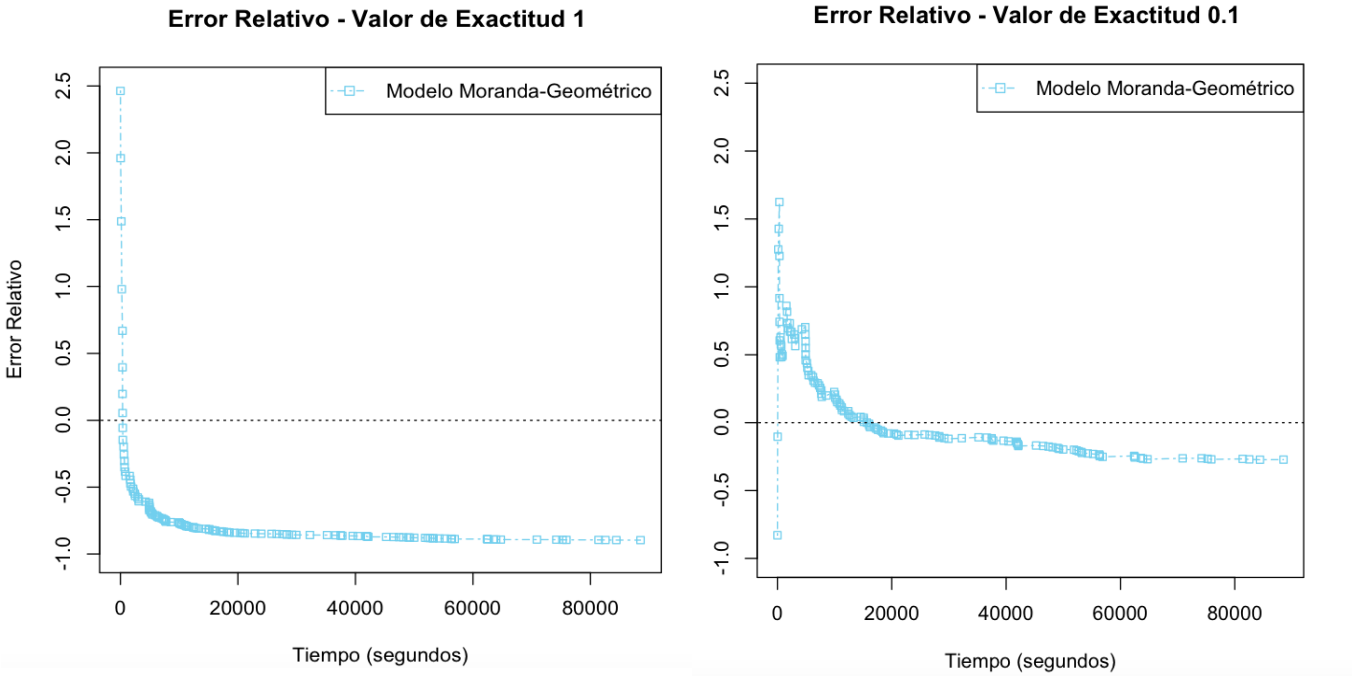

Figura 6.6. Error relativo para distintos niveles de exactitud

Como se ha observado en las gráficas anteriores, el error producido a un menor nivel de exactitud es mucho mayor. Esto queda reflejado en las dos gráficas comparadas de la figura 6.6. El error de la estimación del modelo ejecutado con los datos iniciales es el de la gráfica 6.7.

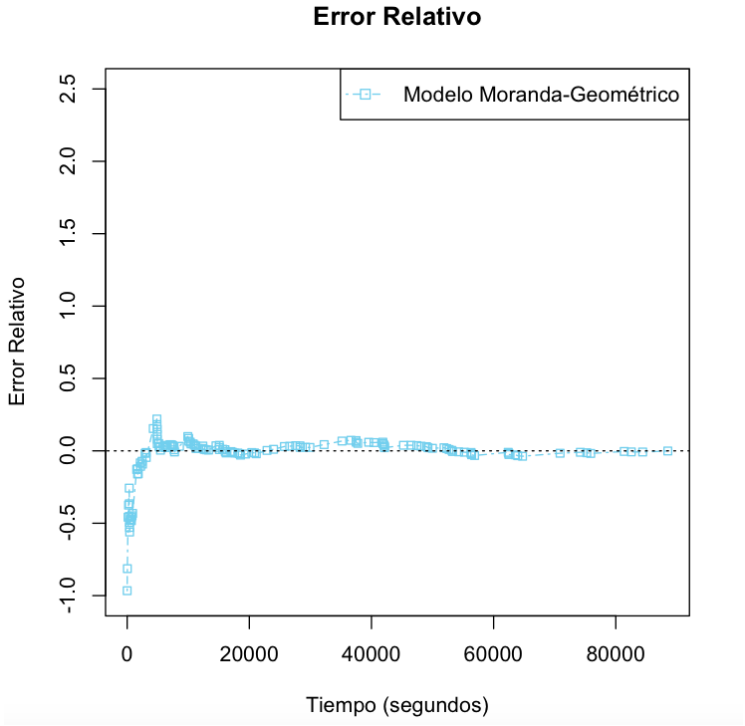

Figura 6.7. Error Relativo del Modelo Moranda Geométrico

De esta gráfica se observa que el nivel de exactitud de este modelo es muy alto, pues el error es casi cero a partir de un pequeño instante de tiempo.

Littlewood Verrall

Analizando el conjunto de datos empleando la función $littlewood.verrall$ en `R`, se obtienen las gráficas que se muestra en las figuras 6.8 y 6.10.

Con el método del EMV y para el modelo lineal se obtienen los estimadores `rho`, `theta0` y `theta1`:

- `rho = 6.219767`
- `theta0 = 44.86406`
- `theta1 = 44.54946`

Al calcular los valores de estos estimadores para la forma cuadrática, se obtiene:

- `rho = 358159111`
- `theta0 = 52855664`
- `theta1 = 140508630`

La diferencia abismal entre los valores llama la atención por lo que es sensato representar el modelo para así ver gráficamente lo que ocurre. Ni tan siquiera se aproxima a la curva de datos reales. Cambiando el método usado por defecto (Nelder-Mead) se ejecuta la función empleando los otros dos métodos de optimización vistos anteriormente, BFGS y CG. Con este último ocurre lo mismo, pero no así con el método BFGS, que genera los siguientes valores para los estimadores:

- `rho = 1334.171`
- `theta0 = 8.372515`
- `theta1 = 408.8465`

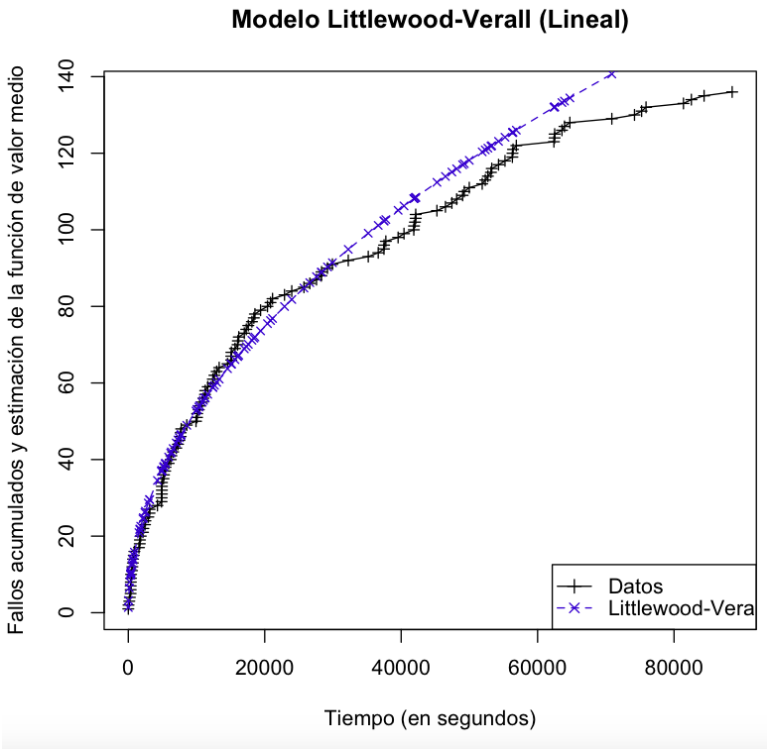

Figura 6.8. Modelo Littlewood-Verrall (lineal). Gráfico acumulado de fallos por tiempo acumulado

Empleando este modelo, la estimación obtenida es algo pesimista conforme pasa el tiempo, al contrario que el modelo Duane.

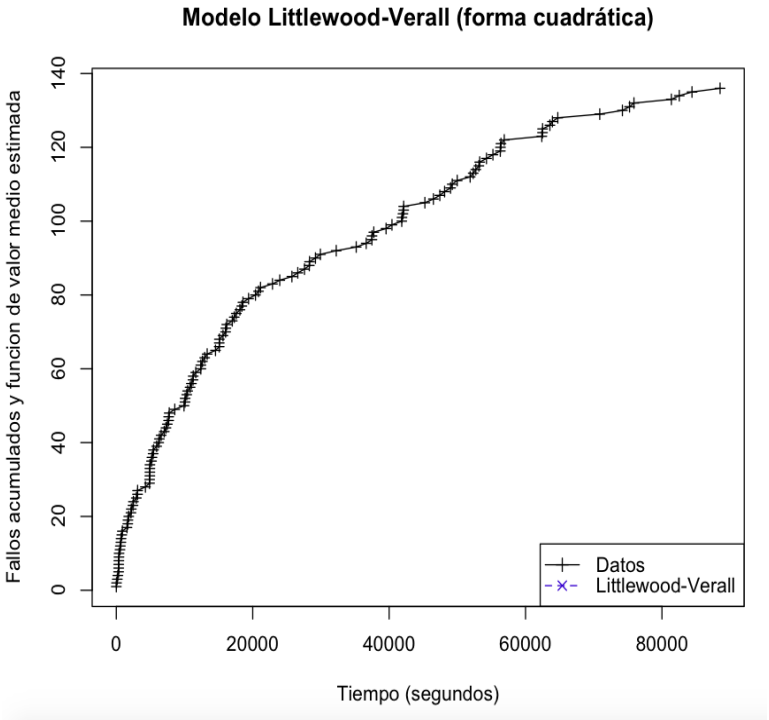

Figura 6.9. Modelo Littlewood-Verrall (cuadrático). Gráfico acumulado de fallos por tiempo acumulado empleando el método Nelder-Mead

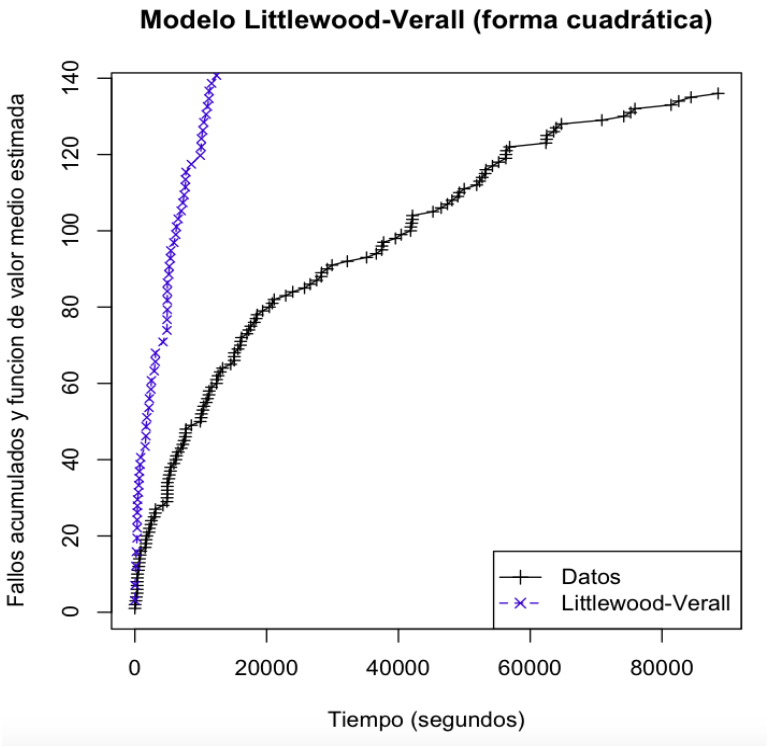

Figura 6.10. Modelo Littlewood-Verrall (cuadrático). Gráfico acumulado de fallos por tiempo acumulado empleando el método BFGS

No parece más idóneo utilizar el modelo Littlewood-Verrall en su forma cuadrática en proyectos del tamaño estudiado, pues la función de valor medio estimada no se asemeja en absoluto a la realidad.

En el caso del modelo en su forma lineal, el error de la estimación es el que se muestra en la siguiente gráfica

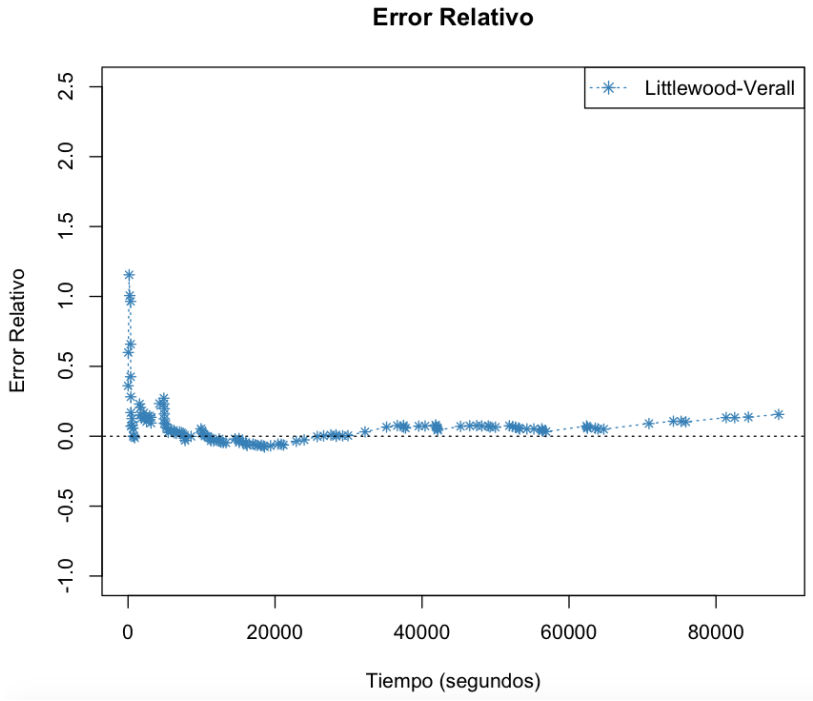

Figura 6.11. Error relativo de la estimación del Modelo Littlewood-Verrall

Modelo Musa-Okumoto

El modelo Musa-Okumoto se estudia en el software `R` con la función

$$musa.okumoto(t, init = c(0,1), tol = .\text{Machine\$double.eps}^{0.25})$$

Al igual que en el modelo Moranda-Geométrico, el parámetro objeto de estudio para ver el comportamiento del modelo es tol, esto es la exactitud deseada.

Para el valor incluido por defecto en el paquete "Reliability" .Machine$double.eps^0.25, explicado anteriormente, se obtienen los siguientes valores para los estimadores `theta0` y `theta1`.

- `theta0 = 11.93933`
- `theta1 = 0.9999431`

La gráfica en la que se incluye esta estimación corresponde con la figura 6.12. Como se puede ver en ella, la función de valor medio estimada no se corresponde con los datos reales hasta casi el final del tiempo de prueba, cuando la pendiente es muy relajada. A diferencia de lo que ocurre en la realidad, esta estimación sugiere que la aparición de errores durante los primeros instantes es mucho más pronunciada.

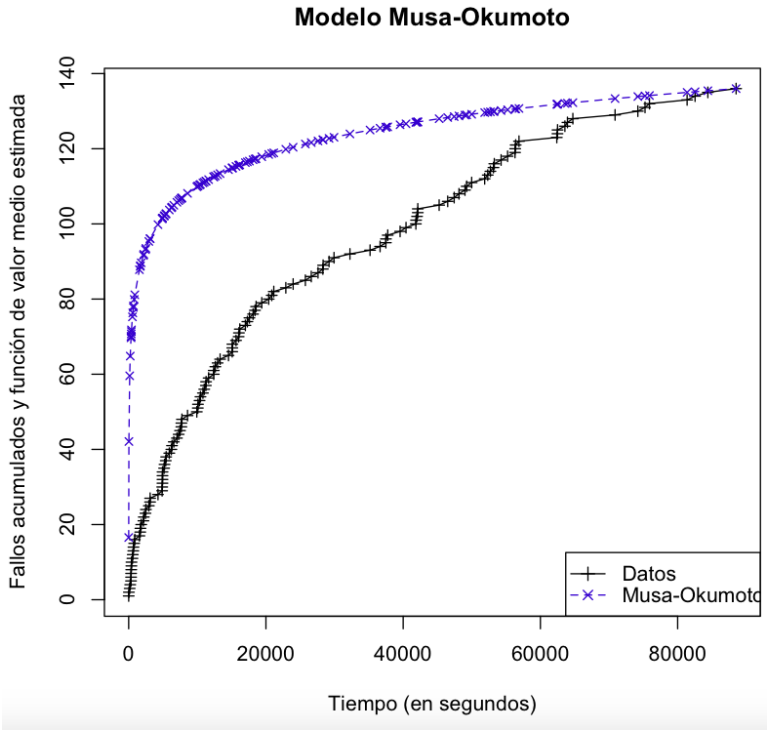

Figura 6.12. Modelo Musa Okumoto. Gráfico acumulado de fallos por tiempo acumulado

Al cambiar el parámetro de la exactitud, no se obtiene variación en la gráfica salvo cuando el nivel de exactitud es mínimo.

El error relativo de este modelo se muestra a continuación:

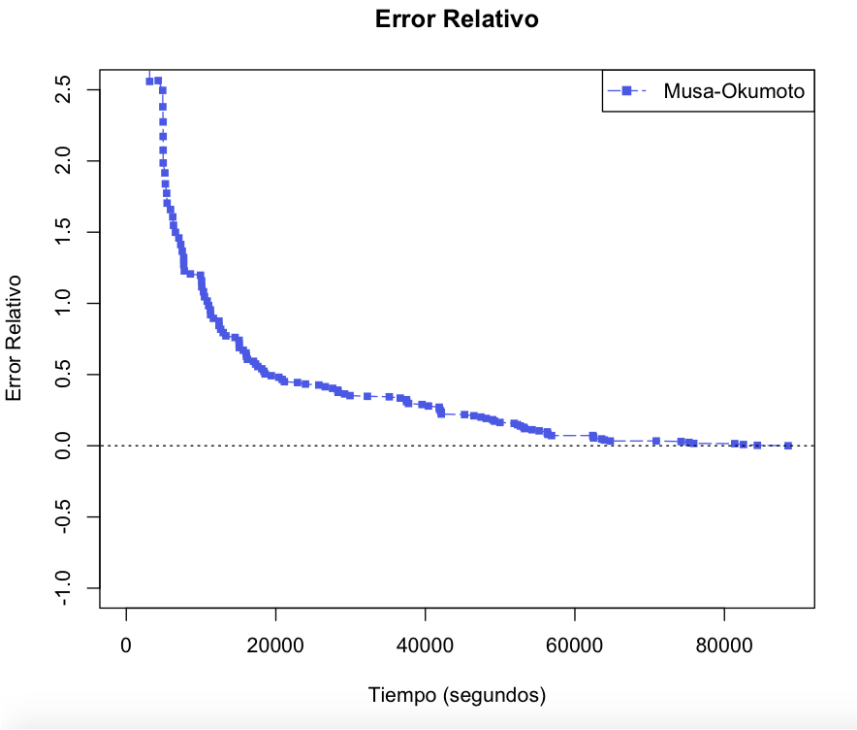

Figura 6.13. Error relativo del modelo Musa-Okumoto

Seguidamente, y a modo de conclusión, se recogen dos gráficas. La primera de ellas incluye una representación de todos los modelos junto con los datos para poder compararlos entre sí y ver cual se acerca más a la realidad. En la segunda se puede ver el error de las estimaciones de cada modelo.

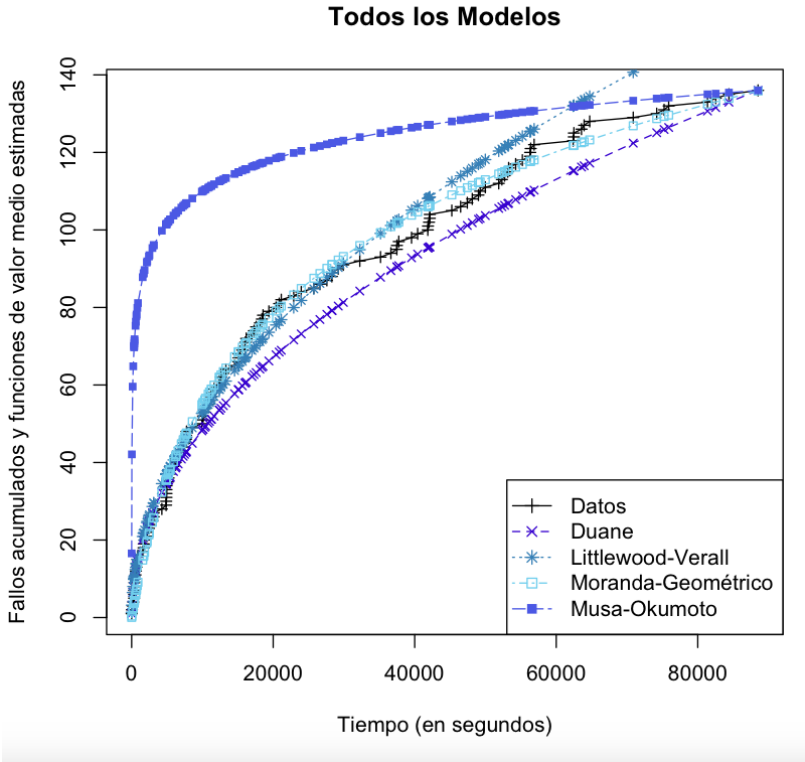

Figura 6.14. Representación grafica de los fallos acumulados y estimaciones de todos los modelos

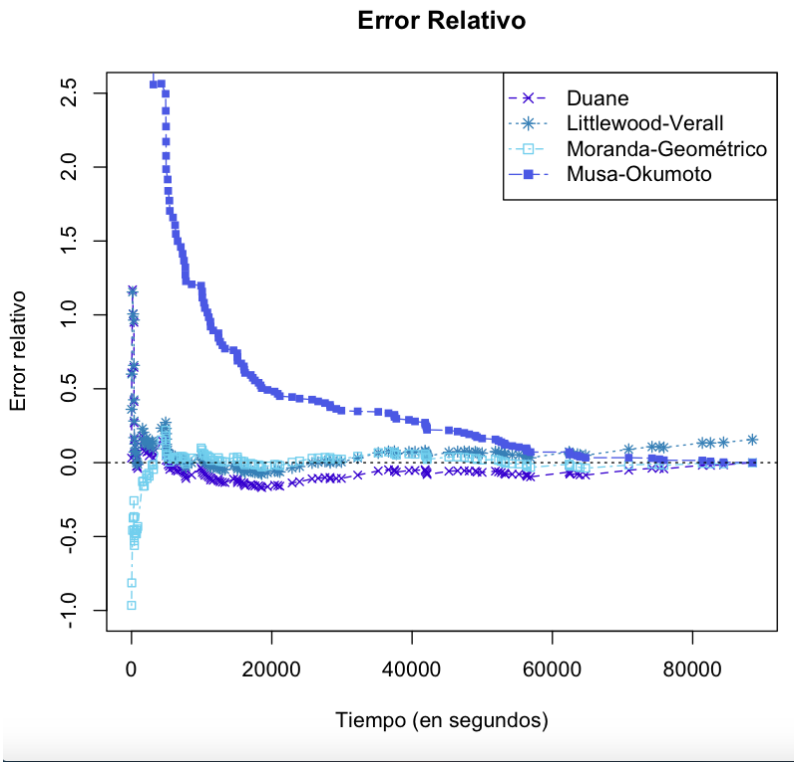

Figura 6.15. Representación gráfica del error relativo de cada modelo estudiado

7 CONCLUSIONES

La fiabilidad del software es una pieza fundamental en la calidad del software. Su estudio puede ser categorizado en tres partes, modelado, medición y mejora.

El modelado de la fiabilidad del software ha ido madurando durante los últimos años hasta el punto de que se puede obtener información muy útil a la hora de desarrollar una aplicación informática, mediante la aplicación de un modelo determinado a cada problema.

Como se vio en el apartado de modelos existentes, son muchos los modelos que pueden ser empleados en el análisis de la fiabilidad de una aplicación informática. El principal problema de esta disciplina es el hecho de que no se puede encontrar un modelo que funcione bien en cualquier situación. Es tarea de los ingenieros adaptar un método o componer uno nuevo a partir de los ya existentes para conseguir una solución que se amolde al problema en particular que se está tratando. En definitiva, es el problema el que propone la solución (o al menos el camino a la solución).

Es importante no obstante considerar las hipótesis de cada modelo antes de aplicarlo a los datos que tengamos. Por ejemplo, si una distribución Weibull o Gamma representa bien unos determinados tiempos de fallo, es más probable que las predicciones obtenidas por un modelo que asuma una distribución de tiempos de fallo similar sean mejores que las que podamos obtener a partir de un modelo que asume una distribución de los tiempos de fallo exponencial.

A menudo se da el caso de un grupo de modelos que tienen hipótesis similares difieren en sus predicciones para el mismo set de datos y también se da el caso de que todos los modelos hacen las mismas predicciones erróneas. En tal caso, es conveniente usar estos modelos como estimadores de la fiabildiad más que como ayuda a la hora de predecirla.

La medición de la fiabilidad del software es fuente de disputa, pues a diferencia de lo que ocurre en otras ramas de la ingeniería no existe un acuerdo generalizado de, por ejemplo, ¿cómo de bueno es un software, cuantitativamente hablando? Si bien puede parecer una pregunta fácil, no existe aún una buena respuesta a la misma. Puesto que la fiabilidad del software no puede ser medida directamente,

para llevar a cabo su estudio estudiar nos ayudamos de factores tales como defectos o fallos del mismo. Estos nos ayudan a comparar diferentes productos.

La última parte de aquellas en las que se categoriza el estudio de la calidad del software era la mejora. La mejora es un aspecto difícil pues aún hoy en día no hay un gran conocimiento sobre la materia y las grandes ventajas que puede comportar el hecho de implantar un estudio de la fiabilidad cuando se desarrolla una aplicación informática. Hasta ahora no hay una buena forma para superar la complejidad del problema del software. Una prueba completa de cualquier software medianamente complejo es inviable y productos libre de defectos no pueden ser asegurados. Salvo en aquellos casos en los que se están desarrollando 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 o salud humana, las restricciones de tiempo y presupuesto son las que limitan el avance o mejora de esta disciplina.

A la hora de llevar a cabo la labor de documentación para realizar este trabajo, se ha aprendido que la predicción de la fiabildiad durante las fases de diseño no permite gran retroalimentación con el proceso de diseño, puesto que están relativamente desacopladas dentro del ciclo de desarrollo del software. Es por ello, que un marco unificado que emplee algunas de los conceptos aprendidos durante las fases más tempranas del ciclo, así como los tiempos de fallos tan pronto estén disponibles para estimar o predecir la fiabilidad aportará más valor en el sentido de una temprana validación de los requerimientos de caldiad del producto. Debido a que este marco unificado no esta desarrollado en la mayoría de empresas de desarrollo de aplicaciones informáticas, es un área que merece y será sin duda, estudiada con más detenimiento.

Se puede concluir que llevar a cabo un estudio de fiabilidad durante el diseño y desarrollo de una aplicación informática otorgará al producto un valor añadido que sin duda será reconocido por los clientes del mismo.

REFERENCIAS

[1] IEEE Std (1993). IEEE Software Engineering Standard: Glossary o Software Engineering Terminology. IEEE Computer Society Press.

[2] Real Academia Española. *Diccionario de la Lengua Española, XXIII Edición*. Consultado el 17 de febrero de 2016.

[3] Andreas Wittmann (2009). Reliability: Functions for estimating parameters in software reliability models. R package version 0.0-2.

[4] R Core Team (2015). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/

[5] Ganesh Pai, A Survey of Software Reliability Models. Department of ECE. University of Virginia, 2002.

[6] M. Grottke, Software reliability model study, PETS Project Technical Report. University of Erlangen-Nuremberg, 2001.

[7] Chin-Yu Huang, Performance analysis of software reliability growth models with testing-effort and change-point. Department of Computer Science, National Tsing Hua University, 2004.

[8] Ana María Hernández Domínguez, Análisis Estadístico de Datos de Tiempos de Fallo en R. Departamento de Estadística, Universidad de Granada, 2010.

[9] Laboratorio Nacional de Calidad del Software, Ingeniería del Software: Metodologías y Ciclos de Vida, 2009.

[10] Mitsuru Ohba (1984). Software reliability analysis models, IBM Journal.

[11] Bárbara Saiz de Bustamante, Tesis Doctoral: El error como proceso de Markov no homogéneo. Aplicación al análisis de la fiabilidad del software, Universidad Politécnica de Madrid, 1995.

[12] Jesús Plaza Rubio, Modelos de Fiabilidad del Software. Trabajo Fin de Grado, Universidad de Valladolid, 2014.

[13] J.D. Musa, A. Iannino, and K. Okumoto. Software Reliability: Measurement, Prediction, Application. McGraw-Hill, 1987.

[14] Michael R. Lyu. Handbook of Software Realibility Engineering. IEEE Computer Society Press, 1996. http://www.cse.cuhk.edu.hk/~lyu/book/reliability/

ANEXO – CÓDIGO EN R

A continuación se detalla el código ejecutado en el programa R.

```
# Función para estimar los parametros del Modelo Duane
"duane" <- function(t, init = c(1, 1), method = "Nelder-Mead", maxit =
 10000, ...)
{
    eq1 <- function(rho, theta, t)
    {
        n <- length(t)
        t <- cumsum(t)
        i <- seq(along = t)
        tn <- t[length(t)]
        rho - n / (tn^theta)
    }

    eq2 <- function(rho, theta, t)
    {
        i <- seq(along = t[1:length(t) - 1])
        n <- length(t)
        t <- cumsum(t)
        tn <- t[length(t)]
        theta - n / (sum(log(tn / t[i])))
    }

    # Unimos las dos ecuaciones
    global <- function(vec, t)
    {
        rho <- vec[1]
        theta <- vec[2]
        v1 <- eq1(rho, theta, t)^2
        v2 <- eq2(rho, theta, t)^2
        v1 + v2
    }
```

```
    # Búsqueda del mínimo para rho y theta
    res <- optim(init, global, t = t, method = method,
                 control = list(maxit = maxit, ...))
    rho <- res$par[1]
    theta <- res$par[2]

    return(list(rho = rho, theta = theta))
}

# Funcion para representar gráficamente la función de valor medio del Modelo
 Duane

"duane.plot" <- function(rho, theta, t, xlab = "time",
                         ylab = "Cumulated failures and estimated mean value
 function",
                         main = NULL)
{
    plot(cumsum(t), 1:length(t), type = "o", pch = 3, cex = 0.8, xlab =
 xlab,
         ylab = ylab)
    lines(cumsum(t), mvf.duane(rho, theta, cumsum(t)), lty = 2, type = "o",
          pch = 4, cex = 0.8, col = "mediumblue")
    legend("bottomright",    c("Data",    "Duane"),    col    =    c("black",
 "mediumblue"), pch = c(3, 4),
           lty = c(1, 2))

    if(!is.null(main))
    {
        title(main)
    }
}

# Función para estimar los parámetros del Modelo Littlewood-Verall

"littlewood.verall" <- function(t, linear = T, init = c(1, 1, 1), method =
 "Nelder-Mead",
                                maxit = 10000, ...)
```

```
{
    if(linear == T)
    {
        eq1 <- function(theta0, theta1, rho, t)
        {
            n <- length(t)
            i <- seq(along = t)
            s1 <- sum(log(theta0 + theta1 * i))
            s2 <- sum(log(theta0 + theta1 * i + t[i]))
            (n / rho) + s1 - s2
        }

        eq2 <- function(theta0, theta1, rho, t)
        {
            i <- seq(along = t)
            s1 <- rho * sum(1 / (theta0 + theta1 * i))
            s2 <- (rho + 1) * sum(1 / (theta0 + theta1 * i + t[i]))
            s1 - s2
        }

        eq3 <- function(theta0, theta1, rho, t)
        {
            i <- seq(along = t)
            s1 <- rho * sum(i / (theta0 + theta1 * i))
            s2 <- (rho + 1) * sum(i / (theta0 + theta1 * i + t[i]))
            s1 - s2
        }

        # Unimos las tres expresiones
        global.linear <- function(vec, t)
        {
            theta0 <- vec[1]
            theta1 <- vec[2]
            rho <- vec[3]
            v1 <- eq1(theta0, theta1, rho, t)^2
            v2 <- eq2(theta0, theta1, rho, t)^2
```

```
            v3 <- eq3(theta0, theta1, rho, t)^2
            v1 + v2 + v3
        }

        # Búsqueda del mínimo para los parámetros theta0, theta1 y rho
         res <- optim(init, global.linear, t = t, method = method,
                      control = list(maxit = maxit, ...))
        theta0 <- res$par[1]
        theta1 <- res$par[2]
        rho <- res$par[3]

        # Repetimos la búsqueda del mínimo para los parámetros theta0,
theta1 y rho
        res <- optim(c(theta0, theta1, rho), global.linear, t = t, method =
method,
                      control = list(maxit = maxit, ...))
    }
    else
    {
        eq1 <- function(theta0, theta1, rho, t)
        {
            n <- length(t)
            i <- seq(along = t)
            s1 <- sum(log(theta0 + theta1 * i^2))
            s2 <- sum(log(theta0 + theta1 * i^2 + t[i]))
            (n / rho) + s1 - s2
        }

        eq2 <- function(theta0, theta1, rho, t)
        {
            i <- seq(along = t)
            s1 <- rho * sum(1 / (theta0 + theta1 * i^2))
            s2 <- (rho + 1) * sum(1 / (theta0 + theta1 * i^2 + t[i]))
            s1 - s2
        }

        eq3 <- function(theta0, theta1, rho, t)
```

```
        {
            i <- seq(along = t)
            s1 <- rho * sum((i^2) / (theta0 + theta1 * i^2))
            s2 <- (rho + 1) * sum((i^2) / (theta0 + theta1 * i^2 + t[i]))
            s1 - s2
        }

        # Unimos las tres ecuaciones
        global.quadratic <- function(vec, t)
        {
            theta0 <- vec[1]
            theta1 <- vec[2]
            rho <- vec[3]
            v1 <- eq1(theta0, theta1, rho, t)^2
            v2 <- eq2(theta0, theta1, rho, t)^2
            v3 <- eq3(theta0, theta1, rho, t)^2
            v1 + v2 + v3
        }

        # Búsqueda del mínimo de theta0, theta1 and rho
        res <- optim(init, global.quadratic, t = t, method = method,
                     control = list(maxit = maxit, ...))
        theta0 <- res$par[1]
        theta1 <- res$par[2]
        rho <- res$par[3]

        # Repetimos búsqueda del mínimo para theta0, theta1 and rho
        res <- optim(c(theta0, theta1, rho), global.quadratic, t = t, method
= method,
                     control = list(maxit = maxit, ...))
    }

    theta0 <- res$par[1]
     theta1 <- res$par[2]
     rho <- res$par[3]

     return(list(theta0 = theta0, theta1 = theta1, rho = rho))
```

```
}


# Función para representar la función de valor medio del Modelo Littlewood-
Verall


"littlewood.verall.plot" <- function(theta0, theta1, rho, t, linear = T,
xlab = "time",
                         ylab = "Fallos acumulados y función de valor medio
estimada", main = NULL)
{
    if(linear == T)
    {
        mvf.ver <- mvf.ver.lin(theta0, theta1, rho, cumsum(t))


        plot(cumsum(t), 1:length(t), type = "o", pch = 3, cex = 0.8,
xlab=xlab,
             ylab = ylab)
    }
    else
    {
        mvf.ver <- mvf.ver.quad(theta0, theta1, rho, cumsum(t))


        plot(cumsum(t), 1:length(t), type = "o", pch = 3, cex = 0.8, xlab =
xlab,
             ylab = ylab)
    }


    lines(cumsum(t), mvf.ver, lty=2, type = "o", pch = 4, cex = 0.8,
col="mediumblue")
    legend("bottomright", c("Datos", "Littlewood-Verall"), col=c("black",
"mediumblue"),
           pch=c(3, 4), lty=c(1, 2))


    if(!is.null(main))
    {
        title(main)
    }
```

```
}


# Fúnción para estimar los parámetros del Modelo Moranda-Geométrico

"moranda.geometric"    <-    function(t,    init    =    c(0,    1),    tol    =
.Machine$double.eps^0.25)
{
    # Función para calcular el valor de D
    Dhat <- function(phi, t)
    {
        n <- length(t)
        i <- seq(along = t)
        return(phi * n / (sum(phi^i * t[i])))
    }

    # Estimador de Máxima Verosimilitud de phi
    phihat <- function(phi, t)
    {
        i <- seq(along = t)
        n <- length(t)
        return((sum(i * (phi^i) * t[i]) / sum((phi^i) * t[i]) - (n + 1) /
2)^2)
    }

    # Búsqueda del mínimo de phi
    min <- optimize(phihat, init, tol = tol, t = t)
    phi <- min$minimum
    D <- Dhat(phi, t)
    theta<- -log(phi)

    return(list(D = D, theta = theta))
}


# Función para representar gráficamente la función de valor medio del Modelo
Moranda-Geometrico
```

```
"moranda.geometric.plot" <- function(D, theta, t, xlab = "Tiempo",
                                     ylab = "Fallos acumulados y función de
valor medio estimada", main = NULL)
{
    plot(cumsum(t), 1:length(t), type = "o", pch = 3, cex = 0.8, xlab =
xlab,
         ylab = ylab)
    lines(cumsum(t), mvf.mor(D, theta, cumsum(t)), lty = 2, type = "o", pch
= 4,
          cex = 0.8, col = "mediumblue")
    legend("bottomright", c("Data", "Moranda-Geometric"), col=c("black",
"mediumblue"),
           pch=c(3, 4), lty=c(1, 2))

    if(!is.null(main))
    {
        title(main)
    }
}


# Función para estimar los parámetros del Modelo Musa-Okumoto

"musa.okumoto" <- function(t, init = c(0, 1), tol =
.Machine$double.eps^0.25)
{
    # Function for computation of theta0hat
    theta0hat <- function(theta1, t)
    {
        n <- length(t)
        t <- cumsum(t)
        tn <- t[length(t)]
        return(n / log(1 + theta1 * tn))
    }

    # Estimador de Maxima Verosimilitud para theta1
    theta1hat <- function(theta1, t)
```

```
    {
        i <- seq(along = t)
        n <- length(t)
        t <- cumsum(t)
        tn <- t[length(t)]
        return((1 / theta1 * sum(1 / (1 + theta1 * t[i])) - n * tn /
               ((1 + theta1 * tn) * log(1 + theta1 * tn)))^2)
    }

    # Búsqueda del mínimo de theta1
    min <- optimize(theta1hat, init, tol = tol, t = t)
    theta1 <- min$minimum
    theta0 <- theta0hat(theta1, t)

    return(list(theta0 = theta0, theta1 = theta1))
}


# Función para representar la función de valor medio del Modelo Musa-Okumoto

"musa.okumoto.plot" <- function(theta0, theta1, t, xlab = "time",
                                ylab = "Cumulated failures and estimated
 mean value function", main = NULL)
{
    plot(cumsum(t), 1:length(t), type = "o", pch = 3, cex = 0.8, xlab =
 xlab,
         ylab = ylab)
    lines(cumsum(t), mvf.musa(theta0, theta1, cumsum(t)), lty = 2, type =
 "o",
          pch = 4, cex = 0.8, col = "mediumblue")
    legend("bottomright", c("Data", "Musa-Okumoto"), col = c("black",
 "mediumblue"),
           pch = c(3, 4), lty = c(1, 2))

    if(!is.null(main))
    {
        title(main)
    }
```

```
}

# Función para representar todas las funciones de valor medio

"total.plot" <- function(duane.par1, duane.par2, lit.par1, lit.par2,
lit.par3, mor.par1, mor.par2, musa.par1,
                        musa.par2, t, linear = T, xlab = "time",
                        ylab = "Cumulated failures and estimated mean value
functions", main = NULL)
{
    if(linear == T)
    {
        mvf.ver <- mvf.ver.lin(lit.par1, lit.par2, lit.par3, cumsum(t))
    }
    else
    {
        mvf.ver <- mvf.ver.quad(lit.par1, lit.par2, lit.par3, cumsum(t))
    }

   plot(cumsum(t), 1:length(t), type = "o", pch = 3, cex = 0.8, xlab =
xlab,
         ylab = ylab)
    lines(cumsum(t), mvf.duane(duane.par1, duane.par2, cumsum(t)), lty = 2,
          type = "o", pch = 4, cex = 0.8, col = "mediumblue")
    lines(cumsum(t), mvf.ver, lty = 3, type = "o", pch = 8, cex = 0.8,
          col = "steelblue")
    lines(cumsum(t), mvf.mor(mor.par1, mor.par2, cumsum(t)), lty = 4,
          type = "o", pch = 0, cex = 0.8, col = "skyblue")
    lines(cumsum(t), mvf.musa(musa.par1, musa.par2, cumsum(t)), lty = 5,
          type = "o", pch = 15, cex = 0.8, col = "royalblue")
    legend("bottomright", c("Data", "Duane", "Littlewood-Verall", "Moranda-
Geometric", "Musa-Okumoto"),
           col = c("black", "mediumblue", "steelblue", "skyblue",
"royalblue"), pch = c(3, 4, 8, 0, 15),
           lty = c(1, 2, 3, 4, 5))

    if(!is.null(main))
    {
```

```
        title(main)
    }
}

# Función para representar todos los errores relativos

"rel.plot" <- function(duane.par1, duane.par2, lit.par1, lit.par2, lit.par3,
 mor.par1, mor.par2, musa.par1,
                        musa.par2, t, linear = T, ymin, ymax, xlab = "time",
                        ylab = "relative error", main = NULL)
{
    l <- 1:length(t)
    rel.duane <- (mvf.duane(duane.par1, duane.par2, cumsum(t)) - l) / l

    if(linear == T)
    {
        rel.ver <- (mvf.ver.lin(lit.par1, lit.par2, lit.par3, cumsum(t)) -
 l) / l
    }
    else
    {
        rel.ver <- (mvf.ver.quad(lit.par1, lit.par2, lit.par3, cumsum(t)) -
 l) / l
    }

    rel.mor <- (mvf.mor(mor.par1, mor.par2, cumsum(t)) - l) / l
    rel.musa <- (mvf.musa(musa.par1, musa.par2, cumsum(t)) - l) / l

    if(missing(ymin))
    {
        ymin <- min(c(rel.duane, rel.ver, rel.mor, rel.musa))
    }

    if(missing(ymax))
    {
        ymax <- max(c(rel.duane, rel.ver, rel.mor, rel.musa))
    }
```

```
    plot(cumsum(t), rel.duane, type = "o", pch = 4, cex = 0.8, col =
"mediumblue",
         xlab = xlab, ylim = c(ymin, ymax), lty = 2, ylab = ylab)
    lines(cumsum(t), rel.ver, lty = 3, type = "o", pch = 8, cex = 0.8, col =
"steelblue")
    lines(cumsum(t), rel.mor, lty = 4, type = "o", pch = 0, cex = 0.8, col =
"skyblue")
    lines(cumsum(t), rel.musa, lty = 5, type = "o", pch = 15, cex = 0.8, col
= "royalblue")
    segments(-max(cumsum(t)) * 1.05, 0, max(cumsum(t)) * 1.05, 0, lty = 3)
    legend("topright", c("Duane", "Littlewood-Verall", "Moranda-Geometric",
"Musa-Okumoto"),
           col = c("mediumblue", "steelblue", "skyblue", "royalblue"), pch =
c(4, 8, 0, 15),
           lty = c(2, 3, 4, 5))

    if(!is.null(main))
    {
        title(main)
    }
}



# Función de valor medio para el Modelo Duane

"mvf.duane" <- function(rho, theta, t)
{
    if(rho <= 0 || theta <= 0)
    {
        stop("rho and theta should be larger than 0")
    }

    if(length(rho) != 1 || length(theta) != 1)
    {
        stop("rho and theta should have length 1")
    }
```

```
    return(rho * t^theta)
}


# Función de valor medio para el Modelo Moranda-Geométrico model

"mvf.mor" <- function(D, theta, t)
{
    if(theta <= 0)
    {
        stop("theta should be larger than 0")
    }

    if(length(D) != 1 || length(theta) != 1)
    {
        stop("D and theta should have length 1")
    }

    return(1 / theta * log((D * theta * exp(theta)) * t + 1))
}


# Función de valor medio para el Modelo Musa-Okumoto

"mvf.musa" <- function(theta0, theta1, t)
{
    if(length(theta0) != 1 || length(theta1) != 1)
    {
        stop("theta0 and theta1 should have length 1")
    }

    return(theta0 * log(theta1 * t + 1))
}


# Función de valor medio para el Modelo Littlewood-Verall model lineal
```

```
"mvf.ver.lin" <- function(theta0, theta1, rho, t)
{
    if(theta1 == 0)
    {
        stop("theta1 should not be equal 0")
    }

    if(length(theta0) != 1 || length(theta1) != 1 || length(rho) != 1)
    {
        stop("theta0, theta1 and rho should have length 1")
    }

    return(1 / theta1 * sqrt(theta0^2 + 2 * theta1 * t * rho))
}

# Función de valor medio para el Modelo Littlewood-Verall modelo cuadrático

"mvf.ver.quad" <- function(theta0, theta1, rho, t)
{
    if(theta1 == 0)
    {
        stop("theta1 should not be equal 0")
    }

    if(length(theta0) != 1 || length(theta1) != 1 || length(rho) != 1)
    {
        stop("theta0, theta1 and rho should have length 1")
    }

    v1 <- (rho - 1)^(1 / 3) / ((18 * theta1)^(1 / 3))
    v2 <- 4 * (theta0^3) / (9 * (rho - 1)^2 * theta1)
    Q1 <- (cumsum(t) + (cumsum(t)^2 + v2)^(1 / 2))^(1 / 3)
    Q2 <- (cumsum(t) - (cumsum(t)^2 + v2)^(1 / 2))^(1 / 3)

    return(3 * v1 * (Q1 + Q2))
```

```
}
```