CAPÍTULO 4

Métricas en el desarrollo del Software

4.1 Las Métricas y la Calidad de Software

El objetivo primordial de la ingeniería del software es producir un sistema, aplicación o producto de alta calidad. Para lograr este objetivo, los ingenieros de software deben emplear métodos efectivos junto con herramientas modernas dentro del contexto de un proceso maduro de desarrollo del software. Al mismo tiempo, un buen ingeniero del software y buenos administradores de la ingeniería del software deben medir si la alta calidad se va a llevar a cabo. A continuación se verá un conjunto de métricas del software que pueden emplearse a la valoración cuantitativa de la calidad de software

El punto de vista de ¿Qué es calidad? Es diverso, y por lo tanto existen distintas respuestas, tales como se muestra a continuación:

> El American Heritage Dictionary [Pressman ´98] define la calidad como "Una característica o atributo de algo."
>
> La definición estándar de calidad en ISO-8402 es "La totalidad de rasgos y características de un producto, proceso o servicio que sostiene la habilidad de satisfacer estados o necesidades implícitas" [Mcdermid '91].
>
> "Concordar explícitamente al estado funcional y a los requerimientos del funcionamiento, explícitamente a los estándares de documentación de

desarrollo, e implícitamente características que son expectativas de todos los desarrolladores profesionales de software" [Pressman '93].

La calidad de un sistema, aplicación o producto es tan buena como los requisitos que detallan el problema, el diseño que modela la solución, el código que transfiere a un programa ejecutable y las pruebas que ejercita el software para detectar errores. Un buen ingeniero del software emplea mediciones que evalúan la calidad del análisis y los modelos de diseño, así como el código fuente y los casos de prueba que se han establecido al aplicar la ingeniería del software. Para obtener esta evaluación de calidad, el ingeniero debe utilizar *medidas técnicas*, que evalúan la calidad con objetividad, no con subjetividad. Asimismo, un buen administrador de proyectos debe evaluar la calidad objetivamente y no subjetivamente. A medida que el proyecto progresa el administrador del proyecto siempre debe valorar la calidad. Aunque se pueden recopilar muchas medidas de calidad, el primer objetivo en el proyecto es medir errores y defectos. Las métricas que provienen de estas medidas proporcionan una indicación de la efectividad de las actividades de control y de la garantía de calidad en grupos o en particulares. Por ejemplo los errores detectados por hora de revisión y los errores detectados por hora de prueba suministran una visión profunda de la eficacia de cada una de las actividades envueltas en la métrica. Así los datos de errores se pueden utilizar también para calcular la eficiencia de eliminación de defectos en cada una de las actividades del marco de trabajo del proceso.

4.1.1 Visión General de los Factores que Afectan a la Calidad

McCall y Cavano [John A. McDermid ‘91] definieron un juego de factores de calidad como los primeros pasos hacia el desarrollo de métricas de la calidad del software. Estos factores evalúan el software desde tres puntos de vista distintos: (1) operación del producto (utilizándolo), (2) revisión del producto (cambiándolo) y (3) transición del producto (modificándolo para que funcione en un entorno diferente, por ejemplo: “portándolo”) Los autores describen la relación entre estos factores de calidad (lo que llaman un ‘marco de trabajo’) y otros aspectos del proceso de ingeniería del software:

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

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

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

4.1.2 Medida de la calidad

Aunque hay muchas medidas de la calidad de software, la *corrección, facilidad de mantenimiento, integridad* y *facilidad de uso* suministran indicadores útiles para el equipo del proyecto. Gilb [Len O. Ejiogo ‘90] sugiere definiciones y medidas para cada uno de ellos, tales como:

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

***Facilidad de mantenimiento*.** El mantenimiento del software cuenta con más esfuerzo que cualquier otra actividad de ingeniería del software. La facilidad de mantenimiento es la habilidad con la que se puede corregir un programa si se encuentra un error, se puede adaptar si su entorno cambia o optimizar si el cliente desea un cambio de requisitos. No hay forma de medir directamente la facilidad de

mantenimiento; por consiguiente, se deben utilizar medidas indirectas. Una métrica orientada al tiempo simple es el *tiempo medio de cambio* (TMC), es decir, el tiempo que se tarda en analizar la petición de cambio, en diseñar una modificación apropiada, en efectuar el cambio, en probarlo y en distribuir el cambio a todos los usuarios. En promedio, los programas que son más fáciles de mantener tendrán un TMC más bajo (para tipos equivalentes de cambios) que los programas que son más difíciles de mantener.

Hitachi ha empleado una métrica orientada al costo (precio) para la capacidad de mantenimiento, llamada “desperdicios”. El costo estará en corregir defectos hallados después de haber distribuido el software a sus usuarios finales. Cuando la proporción de desperdicios en el costo global del proyecto se simboliza como una función del tiempo, es aquí donde el administrador logra determinar si la facilidad de mantenimiento del software producido por una organización de desarrollo está mejorando y asimismo se pueden emprender acciones a partir de las conclusiones obtenidas de esa información.

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

Para medir la integridad, se tienen que definir dos atributos adicionales: amenaza y seguridad. La a*menaza* es la probabilidad (que se logra evaluar o concluir de la evidencia empírica) de que un ataque de un tipo establecido ocurra en un tiempo establecido. La *seguridad* es la probabilidad (que se puede estimar o

deducir de la evidencia empírica) de que se pueda repeler el ataque de un tipo establecido, en donde la integridad del sistema se puede especificar como:

$$\text{integridad} = Ó[1 - \text{amenaza} \times (1 - \text{seguridad})] \qquad (4.1)$$

donde se suman la amenaza y la seguridad para cada tipo de ataque.

Facilidad de uso. El calificativo “amigable con el usuario” se ha transformado universalmente en disputas sobre productos de software. Si un programa no es “amigable con el usuario”, prácticamente está próximo al fracaso, incluso aunque las funciones que realice sean valiosas. La facilidad de uso es un intento de cuantificar “lo amigable que pude ser con el usuario” y se consigue medir en función de cuatro características: (1) destreza intelectual y/o física solicitada para aprender el sistema; (2) el tiempo requerido para alcanzar a ser moderadamente eficiente en el uso del sistema; (3) aumento neto en productividad (sobre el enfoque que el sistema reemplaza) medida cuando alguien emplea el sistema moderadamente y eficientemente, y (4) valoración subjetiva (a veces obtenida mediante un cuestionario) de la disposición de los usuarios hacia el sistema.

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

4.1.3 Medidas de fiabilidad y de disponibilidad.

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

Considerando un sistema basado en computadora, una medida sencilla de la fiabilidad es el *tiempo medio entre fallos* (TMEF) [Mayrhauser´91], donde:

$$TMEF = TMDF+TMDR \qquad (4.2)$$

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

Muchos investigadores argumentan que el TMDF es con mucho, una medida más útil que los defectos/KLDC, simplemente porque el usuario final se enfrenta a los fallos, no al número total de errores. Como cada error de un programa no tiene la misma tasa de fallo, la cuenta total de errores no es una buena indicación de la fiabilidad de un sistema. Por ejemplo, consideremos un programa que ha estado funcionando durante 14 meses. Muchos de los errores del programa pueden pasar desapercibidos durante décadas antes de que se

detecten. El TMEF de esos errores puede ser de 50 e incluso de 100 años. Otros errores, aunque no se hayan descubierto aún, pueden tener una tasa de fallo de 18 ó 24 meses, incluso aunque se eliminen todos los errores de la primera categoría (los que tienen un gran TMEF), el impacto sobre la fiabilidad del software será muy escaso.

Además de una medida de la fiabilidad debemos obtener una medida de la *disponibilidad.* La disponibilidad (4.1.3.2) del software es la probabilidad de que un programa funcione de acuerdo con los requisitos en un momento dado, y se define como:

$$\text{Disponibilidad} = \text{TMDF}/(\text{TMDF} + \text{TMDR}) \times 100\ \% \qquad (4.3)$$

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

4.1.4 Eficacia de la Eliminación de Defectos

Una métrica de la calidad que proporciona beneficios tanto a nivel del proyecto como del proceso, es la eficacia de la eliminación de defectos (EED) En particular el EED es una medida de la habilidad de filtrar las actividades de la

garantía de calidad y de control al aplicarse a todas las actividades del marco de trabajo del proceso.

Cuando se toma en consideración globalmente para un proyecto, EED se define de la forma siguiente:

$$EED = E / (E + D) \qquad (4.4)$$

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

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

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

donde se puede encontrar o no) Cuando se utilizan en este contexto, el EED se vuelve a definir como:

$$EED = Ei / (Ei + Ei+1) \quad (4.5)$$

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

Un objetivo de calidad de un equipo de ingeniería de software es alcanzar un EED que se aproxime a 1, esto es, los errores se deberían filtrar antes de pasarse a la actividad siguiente. Esto también puede ser utilizado dentro del proyecto para evaluar la habilidad de un equipo, esto con el objetivo de encontrar deficiencias que harán que se atrase el proyecto.

Existen varias métricas de calidad, pero las más importantes y que engloban a las demás, son sólo cuatro: corrección, facilidad de mantenimiento, integridad y facilidad de uso, se explican en la siguiente sección.

4.1.5 Factores de Calidad de McCall

Los factores que perturban la calidad del software se pueden categorizar en dos grandes grupos: (1) factores que se pueden medir directamente (por ejemplo: defectos por puntos de función) y (2) factores que se pueden medir sólo indirectamente (por ejemplo: facilidad de uso o de mantenimiento).

McCall y sus colegas plantearon una categorización de factores que afectan a la calidad de software, que se muestran en la figura 4.1 en donde se centralizan con tres aspectos importantes de un producto de software: sus características operativas, su capacidad de cambio y su adaptabilidad a nuevos entornos.

Refiriéndose a los factores de la figura 4.1, McCall proporciona las siguientes descripciones: [Pressman '98].

- *Corrección*: Hasta dónde satisface un programa su especificación y consigue los objetivos de la misión del cliente.
- *Fiabilidad*: Hasta dónde puede quedarse un programa que lleve a cabo su función pretendida con la exactitud solicitada. Cabe hacer notar que se han propuesto otras definiciones de fiabilidad más completas.
-

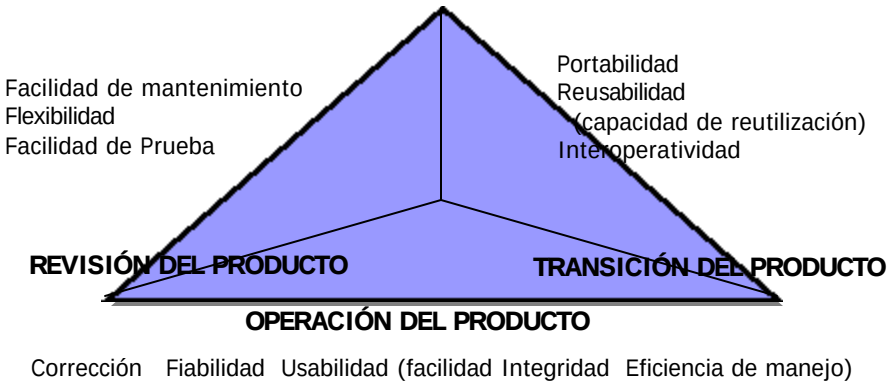

Figura 4.1 Factores que afectan a la calidad del software [Pressman'98]

- *Eficiencia*: El conjunto de recursos informáticos y de código necesarios para que un programa realice su función.
- *Integridad*: Hasta dónde se puede controlar el acceso al software o a los datos por individuos no autorizados.

- *Usabilidad (facilidad de manejo)*: El esfuerzo necesario para aprender, operar, y preparar datos de entrada e interpretar las salida (resultados) de un programa.
- *Facilidad de mantenimiento*: El esfuerzo necesario para localizar y arreglar un error en un programa.
- *Flexibilidad*: El esfuerzo necesario para modificar un programa operativo.
- *Facilidad de prueba*: El esfuerzo necesario para aprobar un programa para asegurarse de que realiza su función pretendida.
- *Portabilidad*: El esfuerzo necesario para trasladar el programa de un entorno de sistema hardware y/o software a otro.
- *Reusabilidad: (capacidad de reutilización):* Hasta dónde se puede volver a utilizar un programa (o partes) en otras aplicaciones con relación al empaquetamiento y alcance de las funciones que ejecuta el programa.
- *Interoperatividad*: El esfuerzo necesario para acoplar un sistema con otro.

Es difícil y en algunos casos improbable, desarrollar medidas directas de los factores de calidad anteriores. Es por eso, que se definen y emplean un conjunto de métricas para desarrollar expresiones para todos los factores de acuerdo con la siguiente relación:

$$F_q = c_1 * m_1 + c_2 * m_2 + \ldots + c_n * m_n \qquad (4.6)$$

Donde F_q es un factor de calidad del software, c_n son coeficientes de regresión y mn son las métricas que afectan al factor de calidad. Lo malo es que las métricas definidas por McCall sólo pueden medirse de manera subjetiva. Las

métricas van en lista de comprobación que se emplean para ‘apuntar’ atributos específicos del software. El esquema de puntuación presentado por McCall es una escala del 0 (bajo) al 10 (alto) En donde se emplean las siguientes métricas en el esquema de puntuación:

- *Facilidad de auditoria*: La facilidad con la que se puede justificar el cumplimiento de los estándares.
- *Exactitud*: La exactitud de los cálculos y del control.
- *Estandarización de comunicaciones*: El nivel de empleo de estándares de interfaces, protocolos y anchos de banda.
- *Complexión*: El grado con que sé a logrado la implementación total de una función.
- *Concisión*: Lo compacto que resulta ser el programa en términos de líneas de código.
- *Consistencia*: El uso de un diseño uniforme y de técnicas de documentación a través del proyecto de desarrollo del software.
- *Estandarización de datos*: El empleo de estructuras y tipos de datos estándares a lo largo del programa.
- *Tolerancia al error*: El deterioro causado cuando un programa descubre un error.
- *Eficiencia de ejecución*: El rendimiento del funcionamiento de un programa.
- *Capacidad de expansión.* El grado con que se pueden aumentar el diseño arquitectónico, de datos o procedimental.
- *Generalidad*: La extensión de aplicación potencial de los componentes del programa.

- *Independencia del hardware*: El grado con que se desacopla el software del hardware donde opera.
- *Instrumentación*: El grado con que el programa vigila su propio funcionamiento e identifica los errores que suceden.

Métrica de la calidad del software / **Factor de calidad**	Corrección	Fiabilidad	Eficiencia	Integridad	Mantenimiento	Flexibilidad	Capacidad de pruebas	Portabilidad	Reusabilidad (capacidad de reutilización)	Interoperabilidad	Usabilidad (facilidad de manejo
Facilidad de auditoria				*			*				
Exactitud		*									
Estandarización de comunicaciones										*	
Compleción		*			*	*					
Complejidad	*										
Concisión	*	*				*	*				
Consistencia				*	*	*					
Estandarización de datos	*	*				*	*			*	
Tolerancia a errores		*									
Eficiencia de ejecución			*								
Capacidad de expansión						*					
Generalidad						*			*	*	*
Independencia del hardware								*	*		
Instrumentación					*	*	*				
Modularidad	*					*	*	*	*	*	
Operatividad			*								*
Seguridad			*								
Autodocumentación					*						
Simplicidad					*	*	*		*	*	
Independencia del sistema		*					*	*	*		
Trazabilidad								*	*		
Facilidad de formación											

Figura 4.2 Relación entre Factores de calidad y métricas de la calidad de software [Fenton'91]

- *Modularidad*: La independencia funcional de componentes de programa.
- *Operatividad*: La facilidad de operación de un programa.

- *Trazabilidad*: La capacidad de alcanzar una representación del diseño o un componente real del programa hasta los requisitos.

Formación: El grado en que el software ayuda a los nuevos usuarios a manejar el sistema.

La relación entre los factores de calidad del software y las métricas de la lista anterior se muestra en la figura 4.2. Convendría decirse que el peso que se asigna a cada métrica depende de los productos y negocios locales.

4.1.6 FURPS

(Funcionality, Usability, Reliability, Performance, Supportability)

Hewlett-Packard ha desarrollado un conjunto de factores de calidad de software al que se le ha dado el acrónimo de FURPS: [Pressman '98]

- *Funcionalidad.* Se aprecia evaluando el conjunto de características y capacidades del programa, la generalidad de las funciones entregadas y la seguridad del sistema global.
- *Usabilidad (facilidad de empleo o uso)* Se valora considerando factores humanos, la estética, consistencia y documentación general.
- *Fiabilidad.* Se evalúa midiendo la frecuencia y gravedad de los fallos, la exactitud de las salidas (resultados), el tiempo medio entre fallos (TMEF), la capacidad de recuperación de un fallo y la capacidad de predicción del programa.
- *Rendimiento.* Se mide por la velocidad de procesamiento, el tiempo de respuesta, consumo de recursos, rendimiento efectivo total y eficacia.

- *Capacidad de soporte.* Combina la capacidad de ampliar el programa (extensibilidad), adaptabilidad y servicios (los tres representan *mantenimiento*), así como capacidad de hacer pruebas, compatibilidad, capacidad de configuración, la facilidad de instalación de un sistema y la facilidad con que se pueden localizar los problemas.

4.2 Métricas del modelo de análisis

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

4.2.1 Métricas basadas en la función

La métrica de punto de función (PF) (capítulo 3.6.2) se puede usar como medio para predecir el tamaño de un sistema que se va a obtener de un modelo de análisis. Para instruir el empleo de la métrica PF, se considerará una sencilla representación del modelo de análisis mostrada por Pressman [‘98]en la figura 4.3

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

El diagrama de flujo de datos se evalúa para determinar las medidas clave necesarias para el cálculo de al métrica de PF.:

- número de entradas de usuario
- número de salidas de usuario
- número de consultas del usuario
- número de archivos
- número de interfaces externas

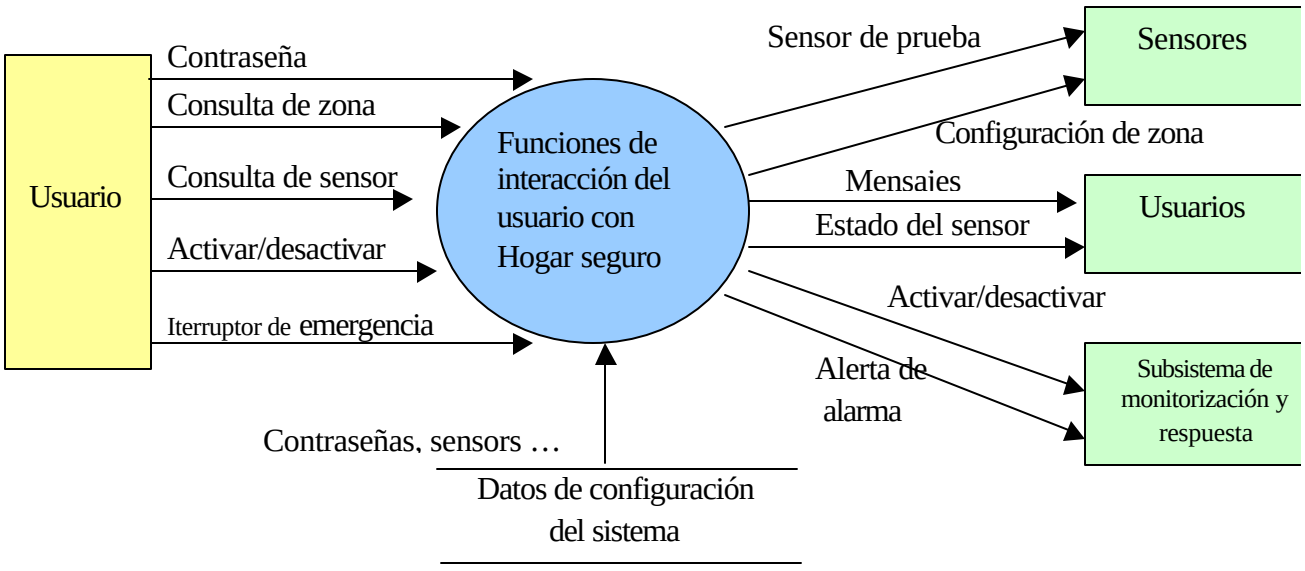

Figura 4.3 Diagrama de flujo de Hogar Seguro [Presmman'98].

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

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

Tabla 4.1 Factor de ponderación [Pressman '98]

La cuenta total mostrada en al figura 4.6.1 b debe ajustarse usando la ecuación:

$$PF = \text{cuenta-total} * (0.65 + 0.01 * ÓF_i)$$

Donde cuenta-total es la suma de todas las entradas PF obtenidas de la figura 4.6.1 b y F_i(i=1 a 14) son "valores de ajuste de complejidad". Para el propósito de este ejemplo, asumimos que $ÓF_i$ es 46 (un producto moderadamente complejo) Por lo tanto:

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

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

4.2.2 La métrica Bang

Puede emplearse para desarrollar una indicación del tamaño del software a implementar como consecuencia del modelo de análisis. Desarrollada por Tom DeMarco [Ejiogo '91], la métrica bang es " una indicación, independiente de la implementación, del tamaño del sistema" [Ejiogo '91]. Para calcular la métrica bang, el desarrollador de software debe evaluar primero un conjunto de primitivas (elementos del modelo de análisis que no se subdividen más en el nivel de análisis) Las primitivas se determinan evaluando el modelo de análisis y desarrollando cuentas para los siguientes elementos:

- *Primitivas funcionales (Pfu)* Transformaciones (burbujas) que aparecen en el nivel inferior de un diagrama de flujo de datos.
- *Elementos de datos (ED)* Los atributos de un objeto de datos, los elementos de datos no compuestos y aparecen en el diccionario de datos.
- *Objetos (OB)* Objetos de datos.

- *Relaciones (RE)* Las conexiones entre objetos de datos.
- *Transiciones (TR)* El número de transacciones de estado en el diagrama de transición de estado.

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

- *Primitivas modificadas de función manual (PMFu)* Funciones que caen fuera del límite del sistema y que deben modificarse para acomodarse al nuevo sistema.
- *Elementos de datos de entrada (EDE)* Aquellos elementos de datos que se introducen en el sistema.
- *Elementos de datos de salida (EDS)* Aquellos elementos de datos que se sacan en el sistema.
- *Elementos de datos retenidos (EDR)* Aquellos elementos de datos que son retenidos (almacenados) por el sistema.
- *Muestras (tokens) de datos (TC_i)* Las muestras de datos (elementos de datos que no se subdividen dentro de una primitiva funcional) que existen en el l'imite de la i-ésima primitiva funcional (evaluada para cada primitiva).
- *Conexiones de relación (Re_i)* Las relaciones que conectan el i-ésimo objeto en el modelo de datos con otros objetos.

DeMarco [Ejiogo '91] sugiere que la mayoría del software se puede asignar a uno de los dos dominios siguientes, *dominio de función o dominio de datos,* dependiendo de la relación RE/PFu. Las aplicaciones de dominio de función (encontrados comúnmente en aplicaciones de ingeniería y científicas) hacen

hincapié en la transformación de datos y no poseen generalmente estructuras de datos complejas. Las aplicaciones de dominio de datos (encontradas comúnmente en aplicaciones de sistemas de información) tienden a tener modelos de datos complejos [Ejiogo ‘91].

Si RE / Pfu < 0,7 implica una aplicación de dominio de función

Si 0,8 < RE / Pfu < 1,4 implica una aplicación híbrida

Si RE / Pfu > 1,5 implica una aplicación de dominio de datos.

Como diferentes modelos de análisis harán una participación del modelo con mayor o menor grado de refinamiento, DeMarco sugiere que se emplee una cuenta medida de muestras (*token*) por primitiva para controlar la uniformidad de la participación a través de muchos diferentes modelos dentro del dominio de una aplicación [Ejiogo ‘91].

$$TC_{avg} = \quad TC_i/Pfu \qquad (4.7)$$

Una vez que se ha calculado la métrica *bang* , se puede empelar el historial anterior para asociarla con el esfuerzo y el tamaño. DeMarco sugiere que las organizaciones se construyan sus propias versiones de tablas IPFuC y IOBC para calibrar la información de proyectos completos de software.

4.2.3 Métricas de la Calidad de Especificación

Existe una lista de características para poder valorar la calidad del modelo de análisis y la correspondiente especificación de requisitos: Especificidad, corrección, compleción, comprensión, capacidad de verificación, consistencia externa e interna, capacidad de logro, concisión, traza habilidad, capacidad de modificación, exactitud y capacidad de reutilización. Aunque muchas de las características anteriores pueden ser de naturaleza cuantitativa, Davis [Pressman '98] sugiere que todas puedan representarse usando una o más métricas. Por ejemplo asumimos que hay n_i requisitos en una especificación, tal como

$$n_i = n_f + n_{nf} \tag{4.8}$$

Donde n_f es el numero de requisitos funcionales y n_{nf} es el número de requisitos no funcionales (por ejemplo, rendimiento).

Para determinar la especificidad de los requisitos, Davis [Pressman '98] sugiere una métrica basada en la consistencia de la interpretación de los revisores para cada requisito:

$$Q_1 = nu_i / n_r \tag{4.9}$$

Donde nu_i es el numero de requisitos para los que todos los revisores tuvieron interpretaciones idénticas. Cuanto más cerca de uno este el valor de Q_1 menor será la ambigüedad de la especificación.

La compleción de los requisitos funcionales pueden terminarse calculando la relación

$$Q_2 = n_u / (n_i * n_s) \tag{4.10}$$

donde n_u es el número de requisitos de función únicos, n_i es el número de entradas (estímulos) definidos o implicados por la especificación y n_s es el número de estados especificados. La relación Q_2 mide porcentaje de funciones necesarias que se han especificado para un sistema, sin embargo, no trata los requisitos no funciónales. Para incorporarlos a una métrica global completa, debemos considerar el grado de validación de los requisitos:

$$Q_3 = n_c / (n_c + n_{nv}) \qquad (4.11)$$

donde n_c es el número de requisitos que se han validados como correctos y n_{nv} el número de requisitos que no se han validado todavía.

4.3 Métrica del modelo del diseño

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

4.3.1 Métricas de diseño de alto nivel

Éstas se concentran en las características de la estructura del programa dándole énfasis a la estructura arquitectónica y en la eficiencia de los módulos.

Estas métricas son de caja negra, en el sentido de que no se requiere ningún conocimiento del trabajo interno de ningún modo en particular del sistema.

Card y Glass [Pressman '98] proponen tres medidas de complejidad del software: *complejidad estructural, complejidad de datos y complejidad del sistema*.

La complejidad estructural. *S(i),* de un módulo i se define de la siguientes manera.

$$S(i) = f^2_{out}(i) \qquad (4.12)$$

donde $f_{out}(i)$ es la expansión del módulo *i*.

La complejidad de datos. *D(i)* proporciona una indicación de la complejidad en la interfaz interna de un módulo i y se define como :

$$D(i) = v(i) / [f_{out}(i) + 1] \qquad (4.13)$$

donde v(i) es el número de variables de entrada y salida del módulo i.

Finalmente. la complejidad del sistema. C(i), se define como la suma de las complejidades estructural y de datos, y se define como.

$$C(i)=S(i)+D(i) \qquad (4.14)$$

A medida que crecen los valores de complejidad, la complejidad arquitectónica o global del sistema también aumenta. Esto lleva a una mayor probabilidad de que aumente el esfuerzo necesario para la integración y las pruebas.

Una métrica de diseño arquitectónico propuesta por Henry y Kafura [Hamdi '99] también emplea la expansión y la concentración. Los autores definen una métrica de complejidad de la forma:.

$$MHK = \text{longitud}(i) \times [f_in(i) + fout(i)]^2 \qquad (4.15)$$

donde la longitud (i) es el número de sentencias en lenguaje de programación en el módulo (i) y fin (i) es la concentración del módulo i. Henry y Kafura amplían la definición de concentración y expansión no sólo el número de conexiones de control del módulo (llamadas al módulo), sino también el número de estructuras de datos del que el módulo i recoge (concentración) o actualiza (expansión) datos. Para calcular el MHK durante el diseño, puede emplearse el diseño procedimental para estimar el número de sentencias del lenguaje de programación del módulo i. Como en las métricas de Card y Glass mencionadas anteriormente, un aumento en la métrica de Henry-Kafura conduce a una mayor probabilidad de que también aumente el esfuerzo de integración y pruebas del módulo.

Fenton ['91] sugiere varias *métricas de morfología* simples (por ejemplo, forma) que permiten comparar diferentes arquitecturas de programa mediante un conjunto de dimensiones directas. En la Figura 4.4 , se muestra un ejemplo de una arquitectura de software donde puede visualizarse las siguientes métricas:

$$\text{Tamaño} = n + a \tag{4.16}$$

Donde *n* es número de nodos (módulos) y a es el número de arcos (líneas de control) Para la arquitectura mostrada en la Figura 4.4, [Fenton'91]

$$\text{tamaño} = 17 + 18 = 35$$

profundidad = el camino más largo desde el nodo raíz (parte más alta) a un nodo hoja. (4.17)

Para la arquitectura mostrada en la Figura 4.4

$$\text{profundidad} = 4.$$

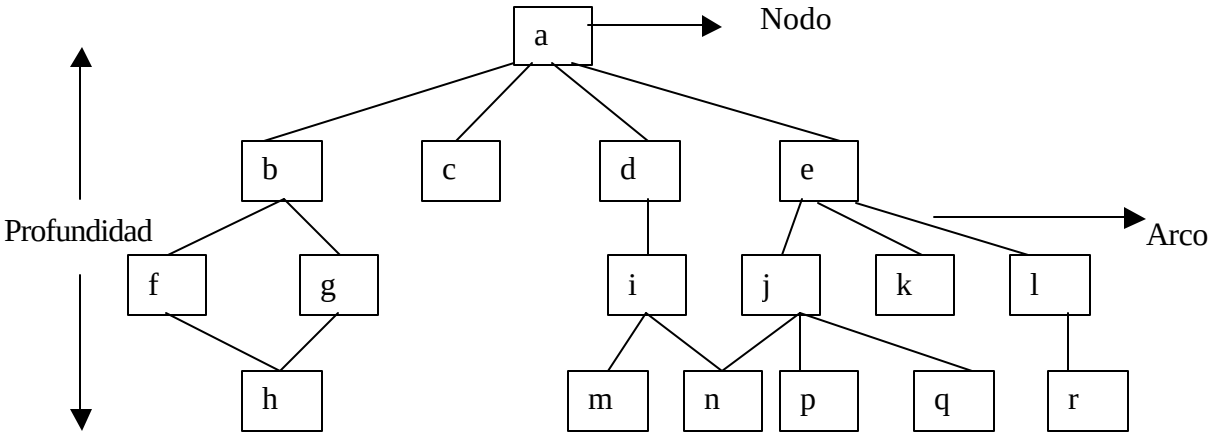

Figura 4.4 Arquitectura de Software [Fenton '91]

anchura = máximo número de nodos de cualquier nivel de la arquitectura. (4.18)

Para la arquitectura mostrada en la Figura 4.4.

anchura = 6.

Relación arco-a-nodo, r = a/n (4.19)

que mide la densidad de conectividad de la arquitectura y puede proporcionar una medida sencilla del acoplamiento de la arquitectura.

Para la arquitectura mostrada en la Figura 4.4:

r = 18/17 = 1,06.

El Mando de sistemas de la Fuerza Aérea Americana [Pressman'98] ha desarrollado varios indicadores de calidad del software basados en características medibles del diseño de un programa de computadora. Empleando conceptos similares a los propuestos en IEEE Std en 1988, la Fuerza Aérea utiliza información obtenida del diseño de datos y arquitectónico para obtener un *índice de calidad de la estructura del diseño* (ICED) que va desde 0 a 1.

Se deben comprobar los siguientes valores para calcular el ICED [Pressman'98]:

S_I = el número total de módulos definidos en la arquitectura del programa

S_2 = el número de módulos que para funcionar correctamente dependen de la fuente de datos de entrada o producen datos que se van a emplear en algún otro lado en general, los módulos de control (entre otros) no formarían parte de S_2.

S_3 = el número de módulos que para que funcionen correctamente dependen de procesos previos

S_4 = el número de elementos de base de datos (incluye todos los objetos de datos y todos los atributos que definen los objetos)

S_5 = el número total de elementos únicos de base de datos

S_6 = el número de segmentos de base de datos (registros diferentes u objetos individuales)

S_7 = el número de módulos con una salida y una entrada (el proceso de excepciones no se considera de salida múltiple)

Una vez que se han determinado los valores S_I a S_7 para un programa de computadora, se pueden calcular los siguientes valores intermedios:

Estructura del programa: D_1, donde D_1 se define como sigue: si el diseño arquitectónico se desarrolló usando un método determinado (por ejemplo, diseño orientado a flujo de datos o diseño orientado a objetos), entonces D_1=1, de otra manera D_1=0.

Independencia del módulo: $D_2 = 1 - (S_2/S_1)$ (4.20)

Módulos no dependientes de procesos previos: $D_3 = 1 - (S_3/S_1)$ (4.21)

Tamaño de la base de datos: D4=1 – (S5/S4) (4.22)

Compartido de la base de datos: D5=1-(S6/S4) (4.23)

Caracterísitcas de entrada/salida del módulo: D6=1 – (S7/S1) (4.24)

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

$$ICED = W_i D_i \quad (4.25)$$

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

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

4.3.2 Métricas de diseño en los componentes

Las métricas de diseño a nivel de componentes se concentran en las características internas de los componentes del software e incluyen medidas de la cohesión, acoplamiento y complejidad del módulo. Estas tres medidas pueden

ayudar al desarrollador de software a juzgar la calidad de un diseño a nivel de componentes.

Las métricas presentadas son de “caja blanca” en el sentido de que requieren conocimiento del trabajo interno del módulo en cuestión. Las métricas de diseño en los componentes se pueden aplicar una vez que se ha desarrollado un diseño procedimental. También se pueden retrasar hasta tener disponible el código fuente.

4.3.2.1 Métricas de cohesión.

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

- *Porción de datos.* Dicho simplemente, una porción de datos es una marcha atrás a través de un módulo que busca valores de datos que afectan a la localización del módulo en el que empezó la marcha atrás. Debería resaltarse que se pueden definir tanto *porciones de programas* (que se centran en enunciados y condiciones) como *porciones de datos*.
- *Símbolos léxicos (tokens) de datos.* Las variables definidas para un módulo pueden definirse como señales de datos para el módulo.
- *Señales de unión.* El conjunto de señales de datos que se encuentran en uno o más porciones de datos.

- *Señales de super-unión.* Las señales de datos comunes a todas las porciones de datos de un módulo.
- *Cohesión.* La cohesión relativa de una señal de unión es directamente proporcional al número de porciones de datos que liga.

Bieman y Ott desarrollaron métricas para cohesiones funcionales fuertes (CFF), cohesiones funcionales débiles (CFD), y pegajosidad (el grado relativo con el que las señales de unión ligan juntas porciones de datos) Estas métricas se pueden interpretar de la siguiente manera [Hamdi'99] .

Todas estas métricas de cohesión tienen valores que van desde 0 a 1. Tienen un valor de 0 cuando un procedimiento tiene más de una salida y no muestra ningún atributo de cohesión indicado por una métrica particular. Un procedimiento sin señales de super-unión, sin señales comunes a todas las porciones de datos, no tiene una cohesión funcional fuerte (no hay señales de datos que contribuyan a todas las salidas) Un procedimiento sin señales de unión, es decir, sin señales comunes a más de una porción de datos (en procedimientos con más de una porción de datos), no muestra una cohesión funcional débil y ninguna adhesividad (no hay señales de datos que contribuyan a más de una salida) La cohesión funcional fuerte y la pegajosidad se obtienen cuando las métricas de Bieman y Ott toman un valor máximo de 1.

Esto es una breve descripción de las métricas de Bieman y Ott. Sin embargo, para ilustrar el carácter de estas métricas, se debe la métrica para la cohesión funcional fuerte:

$$CFF(i) = SU(SA(i))/señales\ (i) \qquad (4.26)$$

donde SU(SA(i)) denota señales de super-unión (el conjunto de señales de datos que se encuentran en todas las porciones de datos de un módulo i) Como la relación de señales de super-unión con respecto al número total de señales en un módulo i aumenta hasta un valor máximo de 1, la cohesión funcional del módulo también aumenta 1.

4.3.2.2 Métricas de acoplamiento.

El acoplamiento de módulo proporciona una indicación de la "conectividad" de un módulo con otros módulos, datos globales y entorno exterior. Dhama [Fenton '91] ha propuesto una métrica para el acoplamiento del módulo que combina el acoplamiento de flujo de datos y de control: acoplamiento global y acoplamiento de entorno. Las medidas necesarias para calcular el acoplamiento de módulo se definen en términos de cada uno de los tres tipos de acoplamiento apuntados anteriormente.

Para el acoplamiento de flujo de datos y de control:

d_i = número de parámetros de datos de entrada

c_i = número de parámetros de control de entrada

d_o = número de parámetros de datos de salida

c_o = número de parámetros de control de salida

Para el acoplamiento global

g_d = número de variables globales usadas como datos

g_c = numero de variables globales usadas como control

Para el acoplamiento de entorno:

w= número de módulos llamados (expansión)

r = número de módulos que llaman al módulo en cuestión (concentración)

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

$$m_c = k/M \qquad (4.27)$$

donde *k* = 1 es una constante de proporcionalidad.

$$M = d_i + a^* c_i + d_o + b^* c_o + g_d + c^* g_c + w + r \qquad (4.28)$$

donde:

$$a=b=c=2$$

Cuanto mayor es el valor de *mc,* menor es el acoplamiento de módulo. Por ejemplo, si un módulo tiene un solo parámetro de entrada y salida de datos, no accede a datos globales y es llamado por un solo módulo entonces:

$$m= 1/(1 +0+1 +0+0+0+0+ l +0)= 1/3=0.33$$

Deberíamos esperar que un módulo como éste presentara un acoplamiento bajo. De ahí que, un valor de *mc* = 0,33 implica un acoplamiento bajo. Por el contrario, si un módulo tiene 5 parámetros de salida y 5 parámetros de entrada de

datos, un número igual de parámetros de control, accede a 10 elementos de datos globales y tiene una concentración de 3 y una expansión de 4,

$$m = 1/(5 + 2.5 + 5 + 2.5 + lO + 0 +,3 + 4) = 0.02$$

el acoplamiento implicado es grande.

Para que aumente la métrica de acoplamiento a medida que aumenta el grado de acoplamiento, se puede definir una métrica de acoplamiento revisada:

$$C= 1\text{-}mc \qquad (4.29)$$

donde el grado de acoplamiento no aumenta linealmente un valor mínimo en el rango de 0,66 hasta un máximo que se aproxima a 1,0.

4.3.2.3 Métricas de complejidad.

Se pueden calcular una variedad de métricas del software para determinar la complejidad del flujo de control del programa. Muchas de éstas se basan en una representación denominada grafo de flujo, un grafo es una representación compuesta de nodos y enlaces (también denominados filos) Cuando se dirigen los enlaces (aristas), el grafo de flujo es un grafo dirigido.

McCabe [Pressman ‘93] identifica un número importante de usos para las métricas de complejidad, donde pueden emplearse para predecir información sobre la fiabilidad y mantenimiento de sistemas software, también realimentan la información durante el proyecto de software para ayudar a controlar la actividad de

diseño, en las pruebas y mantenimiento, proporcionan información sobre los módulos software para ayudar a resaltar las áreas de inestabilidad.

La métrica de complejidad más ampliamente usada (y debatida) para el software es la *complejidad ciclomática,* originalmente desarrollada por Thomas McCabe. La métrica de McCabe proporciona una medida cuantitativa para probar la dificultad y una indicación de la fiabilidad última. Estudios experimentales indican una fuerte correlación entre la métrica de McCabe y el número de errores que existen en el código fuente, así como el tiempo requerido para encontrar y corregir dichos errores. McCabe [Pressman ´98] también defiende que la complejidad ciclomática puede emplearse para proporcionar una indicación cuantitativa del tamaño máximo del módulo. Recogiendo datos de varios proyectos de programación reales, ha averiguado que una complejidad ciclomática de 10 parece ser el límite práctico superior para el tamaño de un módulo, Cuando la complejidad ciclomática de los módulos excedían ese valor, resultaba extremadamente difícil probar adecuadamente el módulo.

4.3.3 Métricas de diseño de interfaz

Aunque existe una significativa cantidad de literatura sobre el diseño de interfaces hombre-máquina, se ha publicado relativamente poca información sobre métricas que proporcionen una visión interna de la calidad y facilidad de empleo de la interfaz.

Sears [Pressman ´98] sugiere la *conveniencia de la representación* (CR) como una valiosa métrica de diseño para interfaces hombre-máquina. Una IGU (Interfaz Gráfica de Usuario) típica usa entidades de representación, iconos gráficos, texto, menús, ventanas y otras para ayudar al usuario a completar tareas. Para realizar una tarea dada usando una IGU, el usuario debe moverse de una entidad de representación a otra. Las posiciones absolutas y relativas de cada entidad de representación, la frecuencia con que se utilizan y el "costo" de la transición de una entidad de representación a la siguiente contribuirá a la conveniencia de la interfaz.

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

$$\text{Costos} = \acute{O}[\text{frecuencia de transición } (k_i) \times \text{costos de transición } (k_i)] \qquad (4.30)$$

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

La conveniencia de la representación se define como:

$$\text{CR} = 100 \times [(\text{costo de la representación óptima CR})/(\text{costo de la representación propuesta})] \; . \qquad (4.31)$$

donde CR = para una representación óptima.

Para calcular la representación óptima de una IGU, la superficie de la interfaz (el área de la pantalla) se divide en una cuadrícula. Cada cuadro de la cuadrícula representa una posible posición de una entidad de la representación. Para una cuadrícula con *N* posibles posiciones y *K* diferentes entidades de representación para colocar, el número posible de distribuciones se representa de la siguiente manera [Pressman '98]:

$$\text{Número posible de distribuciones} = [N\ !/ (K! * (N - K)!] * K! \tag{4.32}$$

La CR se emplea para valorar diferentes distribuciones propuestas de IGU y la sensibilidad de una representación en particular a los cambios en las descripciones de tareas (por ejemplo, cambios en la secuencia y/o frecuencia de transiciones) Es importante apuntar que la selección de un diseño de IGU puede guiarse con métricas tales como CR, pero el árbitro final debería ser la respuesta del usuario basada en prototipos de IGU. Nielsen Levy [Pressman '98] informa; que puede haber una posibilidad de éxito si se prefiere la interfaz basándose exclusivamente en la opinión del usuario ya que el rendimiento medio de tareas de usuario y su satisfacción con la IGU están altamente relacionadas.

4.4 Métricas de código fuente

La teoría de Halstead de la *ciencia del software* es 'probablemente la mejor conocida y más minuciosamente estudiada... medidas compuestas de la

complejidad (software)' [Ejiogo '91]. La ciencia software propuso las primeras 'leyes' analíticas para el software de computadora.

La ciencia del software asigna leyes cuantitativas al desarrollo del software de computadora. La teoría de Halstead se obtiene de un supuesto fundamental [Pressman '98]: 'el cerebro humano sigue un conjunto de reglas más rígido (en el desarrollo de algoritmos) de lo que se imagina...'. La ciencia del software usa un conjunto de medidas primitivas que pueden obtenerse una vez que se ha generado o estimado el código después de completar el diseño. Estas medidas se listan a continuación.

n1: el número de operadores diferentes que aparecen en el programa

n2: el número de operandos diferentes que aparecen en el programa

N1: el número total de veces que aparece el operador

N2: el número total de veces que aparece el operando

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

Halstead expone que la longitud N se puede estimar como:

$$N = n1 \log_2 n1 + n2 \log_2 n2 \qquad (4.33)$$

y el volumen de programa se puede definir como:

$$V = N \log_2 (n1 + n2) \qquad (4.34)$$

Se debería tener en cuenta que *V* variará con el lenguaje de programación y representa el volumen de información

Teóricamente debe existir un volumen mínimo para un algoritmo. Halstead define una relación de volumen L como la relación volumen de la forma más compacta de un programa respecto al volumen real del programa. Por tanto L, debe ser siempre menor de uno. En términos de medidas primitivas, la relación de volumen se puede expresar como:

$$L = 2/n1*n2/N2 \qquad (4.35)$$

Halstead propuso que todos los lenguajes pueden categorizarse por nivel de lenguaje, I, que variará según los lenguajes. Halslead creó una teoría que suponía que el nivel de lenguaje es constante para un lenguaje dado, pero otro trabajo [Pressman '98] indica que el nivel de lenguaje es función tanto del lenguaje como del programador. Los siguientes valores de nivel de lenguaje se muestran en la tabla 4.2.

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

Tabla 4.2 Valores de Nivel de Lenguaje [Pressman '98]

Parece que el nivel de lenguaje implica un nivel de abstracción en la especificación procedimental. Los lenguajes de alto nivel permiten la

especificación del código a un nivel más alto de abstracción que el lenguaje ensamblador (orientado a máquina).

4.5 Métricas para pruebas

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

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

La métrica bang puede proporcionar una indicación del número de casos de prueba necesarios para examinar las medidas primitivas. El número de primitivas funcionales (Pfu), elementos de datos, (ED), objetos (0B), relaciones (RE), estados (ES) y transiciones (TR) pueden emplearse para predecir el número y tipos de pruebas de la caja negra y blanca del software Por ejemplo, el número de pruebas asociadas con la interfaz hombre-máquina se puede estimar examinando (1) el número de transiciones (TR) contenidas en la representación estado-transición del IHM y evaluando las pruebas requeridas para ejecutar cada

transición, (2) el número de objetos de datos (0B) que se mueven a través de la interfaz y (3) el número de elementos de datos que se introducen o salen.

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

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

$$NP=1 / [(n1 / 2) * (N2 / n2)] \quad (4.36)$$

$$e = V / NP \quad (4.37)$$

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

$$\text{porcentaje de esfuerzo de pruebas } (k) = e(k)/ Óe(i) \quad (4.38)$$

donde *e(k)* se calcula para el módulo *k* usando las ecuaciones (4.35 y 4.36) y suma en el denominador de la ecuación (4.37) es la suma del esfuerzo de la ciencia del software a lo largo de todos los módulos del sistema.

A medida que se van haciendo las pruebas tres medidas diferentes proporcionan una indicación de la complexión de las pruebas. Una medida de la *amplitud de las pruebas* proporciona una indicación de cuantos (del número total de ellos) se han probado. Esto proporciona una indicación de la complexión del plan de pruebas. La *profundidad de las prueba* es una medida del porcentaje de los caminos básicos independientes probados en relación al número total de estos caminos en el programa. Se puede calcular una estimación razonablemente exacta del número de caminos básicos sumando la complejidad ciclomática de todos los módulos del programa. Finalmente, a medida que se van haciendo las pruebas y se recogen los datos de los errores, se pueden emplear los *perfiles de fallos* para dar prioridad y categorizar los errores descubiertos. La prioridad indica la severidad del problema. Las categorías de los fallos proporcionan una descripción de un error, de manera que se puedan llevar a cabo análisis estadísticos de errores.

4.6 Métricas de mantenimiento

Se han propuesto métricas diseñadas explícitamente para actividades de mantenimiento. El estándar IEEE 982.1-1988 [Pressman ‘98] sugiere un *índice de madurez del software* (IMS) que proporciona una indicación de la estabilidad de un

producto de software (basada en los cambios que ocurren con cada versión del producto) Se determina la siguiente información:

M_r = número de módulos en la versión actual

F_c = número de módulos en la versión actual que se han cambiado

F_a = número de módulos en la versión actual que se han añadido

F_d = número de módulos de la versión anterior que se han borrado en la versión actual

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

$$IMS = [M_r – (F_a + F_c + F_d)]/ M_r \quad (4.39)$$

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