Medición
en la ingeniería del software

Daniel Rodríguez (daniel.rodriguezg@uah.es)

Índice

1. Introducción

El glosario IEEE de términos de ingeniería del software define la ingeniería del software como la *"aplicación de una aproximación sistemática, disciplinada y* ***cuantificable*** *al desarrollo del software..."*. La medición está pues inexorablemente ligada a nuestra disciplina como una actividad necesaria a lo largo de todo el ciclo de vida del software. En aspectos clave tales como la planificación y gestión de proyectos, la medición resulta fundamental para la estimación de recursos, coste y esfuerzo, la evaluación del personal o el cómputo de la productividad. La medición permite además, durante la ejecución de un proyecto, conocer el estado del mismo para realizar ajustes o mejoras en los procesos, si fuera necesario. Finalmente, la medición de los productos y sus características hace posible la mejora de su calidad. Por ejemplo, se pueden estudiar qué características del código fuente se dan más en los módulos con errores.

Es importante resaltar que al igual que ocurre con otras áreas de la ingeniería del software, la medición está todavía madurando. Por ello, es frecuente objeto de críticas, muchas de las cuales están relacionadas con la falta de adherencia a la teoría.

1.1. Conceptos básicos

La teoría de la representación de la medición establece los principios generales de la medición y su validez. Esta teoría trata de expresar de forma numérica (mundo formal) las entidades del mundo real (o mundo empírico) y la correspondencia entre ambos mundos. Como veremos más adelante, las entidades en la ingeniería del software son los procesos, los recursos (personal, oficinas, etc.) y todos los artefactos (código, documentación, etc.) generados durante el ciclo de vida del software. El estándar ISO/IEC 15939 define entidad del siguiente modo:

Definición
Se denomina **entidad** a un objeto que va a ser caracterizado mediante una medición de sus atributos.

Los atributos, por otra parte, son las características de las entidades. Por ejemplo, algunos atributos del código fuente pueden ser las líneas de código o su complejidad. Por tanto, definiremos atributo del siguiente modo:

Definición
Un **atributo** es una característica medible de una entidad.

Además de la definición de estos dos conceptos, entidad y atributo, centrales para el estudio de la medición, es importante clarificar la diferencia entre *medición* y *medida*:

Definición
Medición [4] es el proceso por el que se asignan números o símbolos a atributos de entidades del mundo real para describirlos según unas reglas definidas de antemano.

Definición
Medida [4] es la asignación de un símbolo o número resultado de una medición a una entidad para caracterizar un atributo.

Otra definición más específica de nuestra disciplina para el término *medida* es la que proporciona el glosario IEEE de términos de ingeniería del software:

Definición
Medida es la evaluación cuantitativa del grado en el cual un producto o proceso software posee un atributo determinado.

Nos encontramos por tanto con elementos reales, por una parte, y con otros formales o matemáticos, por otra, entre los cuales existe una relación. Así por ejemplo, al atributo *número de líneas de código* del código fuente se le puede asignar un número entero. A otro atributo del código fuente, la *facilidad de mantenimiento*, podríamos asignarle subjetivamente el valor

alto, *medio* o *bajo*, etc.

Las medidas han de satisfacer la denominada "condición de representación", la cual establece que las relaciones empíricas deben preservar las relaciones numéricas y viceversa. Por tanto, sólo será la magnitud A mayor que la magnitud B si las medidas que tomemos de A son mayores que las que tomemos de B ($A > B$ si y sólo si $M(A) > M(B)$). La función $DuracionProyecto$ (por ejemplo), definida como "el número de días transcurridos desde el inicio de un proyecto", cumple la condición de representación pues para todo par de proyectos $P1$ y $P2$, siendo $P1$ más corto que $P2$, entonces $DuracionProyecto(P1) < DuracionProyecto(P2)$.

Otro concepto importante en el mundo de la medición es la noción de escala, que podría definirse del siguiente modo:

Definición
Una **escala** de medición es un conjunto de valores que permite establecer relaciones entre medidas. Con frecuencia dicho conjunto es continuo, está ordenado y viene delimitado por un punto inicial y otro final.

Dado que se emplean diferentes escalas de medición según la magnitud que se quiera medir, es lógico que existan también diferentes tipos de escalas tal y como se detalla a continuación.

1.2. Tipos de escalas de medición

Cada tipo de escala engloba a todas las escalas que admiten las mismas *transformaciones admisibles*, es decir, los tipos de operaciones matemáticas que se pueden aplicar a una escala garantizando que se conserva la condición de representación. Cada atributo debe poder ser medido con un tipo de escala determinada y si bien es posible modificar los valores de la misma, el tipo debe mantenerse inalterable. A continuación se describen los diferentes tipos de escalas de menor a mayor complejidad:

- **Escala nominal**. Aquella formada por categorías entre las cuales no

existe ningún orden, por lo que la única relación que se puede aplicar es la de igualdad. Por ejemplo, la escala nominal para determinar el sexo, que tiene únicamente 2 valores: {*masculino*, *femenino*}. O la clasificación de los módulos según su lenguaje de programación: {*Java*, *C++*, *Phyton*, *COBOL*, *Ruby...*}.

- **Escala ordinal**. Aquella en que se definen categorías pero, a diferencia de la escala nominal, existe una relación de orden .es menor que.entre ellas. Un ejemplo de escala nominal es la escala de Likert, muy utilizada en cuestionarios, en la que se asignan valores enteros entre 1 y 5 que se hacen corresponder generalmente con los valores {*Muy poco*, *Poco*, *Medio*, *Bastante*, y *Mucho*}. El tipo de error en un programa, que podría clasificarse como {*Leve*, *Moderado* o *Grave*}, es otro ejemplo de escala ordinal.

- **Escala intervalo**. En este tipo de escala la distancia entre intervalos es conocida y siempre la misma, si bien no tienen un valor inicial de referencia o cero absoluto. El ejemplo clásico es el de la escala centígrada o Celsius para la temperatura. En esta escala la diferencia entre 12ºC y 13ºC es la misma que entre 24ºC y 25ºC, pero no podemos decir que a 24ºC haga "el doble de calor" que a 12ºC. Una escala de este tipo en la ingeniería del software es la *duración de un proyecto* en días: podemos decir que un proyecto está en el día 200, pero no tiene ningún sentido decir que un proyecto va a empezar el "doble de tarde" que otro.

- **Escala de ratio**. Este tipo de escalas es el que más información proporciona y por tanto el que permite llevar a cabo análisis más interesantes y completos. Tienen un valor inicial de referencia o cero absoluto, y permiten definir ratios coherentes con los valores de la escala. O lo que es lo mismo, se pueden comparar los valores estableciendo proporciones. El ejemplo clásico es el de la escala Kelvin de medición de temperaturas. En la ingeniería del software, un ejemplo de este tipo de escalas es la longitud de un programa en líneas de código, que permite decir que un programa es *el doble de largo* o *la mitad de largo* que otro.

Cuadro 1: Resumen de escalas de medición y operaciones permitidas

Escala	Operaciones estadísticas	Operaciones matemáticas
nominal	moda	igualdad (=)
ordinal	mediana	orden (<, >)
intervalo	media, desviación estándar	+, −
ratio	media geométrica, coeficiente de variación	+, −, ×, ÷

- **Escala absoluta**. Las escalas absolutas tienen las características de las escalas anteriores, si bien consisten simplemente en la cuenta sin transformación del número de entidades. El número de programadores involucrado en un desarrollo, algo que sólo se puede medir contando, es un ejemplo de escala absoluta.

En la tabla 1 se muestran los distintos tipos de escalas con las diferentes operaciones matemáticas y estadísticas que se pueden aplicar a cada una de ellas. Téngase en cuenta que para cada una de ellas no sólo se pueden aplicar las operaciones que se indican, sino también todas las aplicables a las escalas de menor complejidad que la preceden.

1.3. Clasificación de las medidas

Una vez definidos los conceptos fundamentales, trataremos las medidas con mayor profundidad. Vaya por delante el hecho de que no existe una única clasificación de las medidas y que, por el contrario, se suelen clasificar según distintos criterios.

Una clasificación habitual consiste en dividir las medidas de un atributo en dos tipos, directas e indirectas:

- **Medidas directas:** Las medidas directas de un atributo son aquellas que pueden ser obtenidas directamente de la entidad sin necesidad de ningún otro atributo. Ejemplos de medidas directas en la ingeniería del software serían la *longitud* del código, el *número de defectos* durante

los 6 primeros meses del sistema en producción o las *horas* de trabajo de un programador en un proyecto.

- **Medidas indirectas:** Son aquellas que se derivan de una o más medidas de otros atributos. Se definen y calculan, por tanto, a partir de otras medidas. Un ejemplo de medida indirecta es la densidad de defectos de un módulo, que se define como el número de defectos del módulo dividido por su tamaño.

Otra clasificación diferencia entre medidas objetivas y subjetivas, pues dependiendo de uno u otro tipo éstas se emplean como base de muy diferentes estudios:

Medidas objetivas: Una medida es objetiva si su valor no depende del observador.

Medidas subjetivas: Son aquellas en las que la persona que realiza la medición puede introducir factores de juicio en el resultado. No obstante, el interés de estas medidas está en

Aunque en alguna literatura sobre medición de software se da un significado específico y distinto a los términos *medida* y *métrica*, habitualmente se usan de manera indistinta. Ambos términos tienen las siguientes acepciones: (i) como medida, es decir la asignación de un número a una entidad; (ii) como escala de medición; y (iii) como atributo de las entidades, por ejemplo, métricas orientadas a objetos. Para los propósitos de este libro, consideraremos medida y métrica términos sinónimos.

1.4. Evaluación de las métricas

La definición de métricas y modelos de medición como correspondencia entre entidades del mundo real y números no es trivial. Una métrica debe medir adecuadamente el atributo de la entidad a medir, pero también definir inequívocamente cómo se va a realizar la medición. Es por tanto necesario,

para que las métricas y modelos de medición cobren sentido, que sean evaluados tanto teórica como experimentalmente.

Desde el punto de vista teórico, las métricas deben cumplir ciertas propiedades para ser consideradas válidas. Aunque existen estudios donde se enumeran detalladamente dichas propiedades, aquí no las estudiaremos en profundidad, pero sí enumeraremos algunas para ilustrar al lector. En el caso de las *métricas directas* (aquellas que se obtienen directamente de la entidad sin ningún otro atributo intermedio) es necesario por ejemplo que la métrica permita distinguir diferentes entidades entre sí, que cumpla la *condición de representación* o que permita que diferentes entidades puedan tener el mismo valor, entre otros.

Pero además de la evaluación teórica, es importante que una métrica sea evaluada de manera experimental. Para corroborar la validez de las métricas desde este punto de vista se emplean métodos empíricos, que pueden clasificarse en encuestas, casos de estudio y experimentación formal. Así, usando métodos estadísticos y experimentales, se evalúan la utilidad y relevancia de las métricas. En la sección 4 se describen en más detalle las distintas posibilidades a la hora de evaluar las métricas de ingeniería del software.

2. ¿Qué medir en la ingeniería del software?

Una vez introducidos los fundamentos de la teoría de la medición, necesitamos definir los tipos de entidades que encontramos en la ingeniería del software, para después definir sus correspondientes atributos, que es sobre los que se llevan cabo las mediciones. En concreto son tres los tipos de entidades:

- **Productos**. Cualquier artefacto, entregable o documento que resulta de cualquiera de las actividades (procesos) del ciclo de vida del software. El código fuente, las especificaciones de requisitos, los diseños, el plan de pruebas y los manuales de usuario son algunos ejemplos de productos.
- **Procesos**. Todas las actividades del ciclo de vida del software: requisi-

Cuadro 2: Ejemplos de métricas de productos, adaptado de [4]

Producto	Atributos internos	Atributos externos
Especificaciones	tamaño, reutilización, etc.	calidad, legibilidad
Diseño	tamaño, complejidad, acoplamiento, cohesión, etc.	calidad, complejidad
Código	tamaño, complejidad, etc.	fiabilidad, facilidad de mantenimiento
Casos de Pruebas	número de casos, %cobertura, etc.	calidad
...	...	...

tos, diseño, construcción, pruebas, mantenimiento, etc. Las mediciones en los procesos van encaminadas, en primer lugar, a conocer el estado de los procesos y cómo se llevan a cabo, para después mejorarlos. Algunas de las métricas relacionadas con los procesos son el tiempo invertido en las actividades o el tiempo para reparar un defecto.

Cuadro 3: Ejemplos de métricas de procesos, adaptado de [4]

Proceso	Atributos internos	Atributos externos
Análisis de requisitos	tiempo, esfuerzo, número de requisitos	calidad, coste, estabilidad
Diseño	tiempo, esfuerzo, número de errores, etc.	calidad, coste, estabilidad, etc.
Pruebas	tiempo, esfuerzo, número de errores, etc.	
...	...	...

- **Recursos**. Cualquier entrada de una actividad. Por ejemplo, el número de personas por actividad o proyecto, las herramientas utilizadas (herramientas para requisitos, compiladores, etc.), oficinas, ordenadores, etc.

Cuadro 4: Ejemplos de métricas de recursos, adaptado de [4]

Recurso	Atributos internos	Atributos externos
Personal	edad, salario	productividad, experiencia, etc
Equipos	número de personas, estructura del equipo, etc	productividad, experiencia, etc
Software	coste, número de de licencias, etc	usabilidad, fiabilidad, etc.
Hardware	marca, coste, especificaciones técnicas, etc.	usabilidad, fiabilidad, etc.
...	...	...

Como hemos comentado anteriormente, a las entidades se les asignan atributos a medir. En la literatura se distingue entre atributos internos y atributos externos:

- **Atributos internos**. Los atributos internos de un producto, proceso o recurso son aquellos que se pueden medir directamente a partir de dicho producto, proceso o recurso. En un módulo software, por ejemplo, podemos medir directamente el número de defectos que se han encontrado en el mismo. Sin embargo, es importante resaltar que el principal uso de los atributos internos es la medición de los atributos externos, como a continuación veremos.

- **Atributos externos**. Los atributos externos de productos, procesos o recursos son aquellos que los relacionan con el entorno. Se miden por medio de métricas indirectas y se deducen en función de atributos internos. Como atributos externos se suelen considerar los relacionados con la calidad. La calidad de suele dividir en factores que no pueden medirse directamente, como por ejemplo, la fiabilidad, la eficiencia, la usabilidad o la facilidad de mantenimiento. A cada uno de estos factores se les asigna una o varias métricas. Por ejemplo, al atributo *facilidad de mantenimiento* de una entidad software, se le pueden asignar métricas como el tiempo medio para reparar un defecto, el número de errores no resueltos, el porcentaje de modificaciones que introducen errores, etc.

En las siguientes secciones se describen ejemplos de métricas generales, aplicables por tanto a muchas de las actividades del ciclo de vida. Como se ha comentado, la medición es una actividad necesaria a lo largo de todo el ciclo del vida del software.

A continuación estudiaremos las diferentes medidas según su objeto. Así, comenzaremos describiendo las medidas del producto (divididas en dos, medidas de atributos internos y de atributos externos) para posteriormente estudiar las medidas de los procesos y finalmente las de los recursos.

2.1. Medidas del producto: atributos internos

2.1.1. Tamaño de sistemas y código fuente

La cuenta de las ***líneas de código*** (*LoC – Lines of Code*) es una de las métricas más usadas, esencialmente por su facilidad y simplicidad de cálculo. Sin embargo, aunque *LoC* puede parecer una métrica sencilla e intuitiva, su definición no es trivial. Y lo que es todavía más importante, su definición y modo de cómputo debe homogeneizarse para que todos los interesados entiendan y apliquen la métrica del mismo modo. Porque por ejemplo, una persona podría considerar las líneas en blanco y/o los comentarios como líneas válidas, mientras que otra podría no tenerlos en cuenta. También es discutible si debe considerarse que las declaraciones de variables hayan de formar parte de la cuenta de instrucciones del lenguaje de programación o no. Por todo lo anterior, es frecuente toparse con variantes de la métrica *LoC* tales como *NCLoC* (*Non-Comment Lines of Code*, líneas de código sin comentarios), *CLoC* (*Comment Lines of Code*, líneas de código con comentarios) o *DSI* (*Delivered Source Instructions*, número de sentencias o instrucciones). Debemos por tanto saber exactamente a qué nos referimos cuando usamos las líneas de código para comparar tamaños de programas y sistemas[1] o cuando hacemos estimaciones de tamaño y productividad (que se pueden medir en LoC/dia, por ejemplo).

Dada la variabilidad de los resultados de utilizar como medida de tamaño las líneas de código (algunos autores han estimado que puede llegar a diferencias del 500 %), y puesto que esta métrica es además muy dependiente del lenguaje de programación usado, han aparecido métricas de tamaño más perfeccionadas. Los ***puntos de función***, originalmente descritos a finales de los años 1970 por Allan Albrecht, miden el tamaño del software por la cantidad de funcionalidad que proporciona a los usuarios (sin considerar el código fuente). Se basan fundamentalmente en la cuenta de entradas, salidas, accesos y modificaciones a las bases de datos y ficheros ponderada por

[1]Para referirse a este tipo de comparaciones se utiliza a menudo el término inglés *benchmarking*.

la complejidad de cada uno de ellos.

Todas las métricas tratadas hasta el momento son métricas directas de tamaño. Un ejemplo de métrica indirecta en el tamaño del software es la ***densidad de comentarios*** de un programa, que se calcula dividiendo el número de líneas con comentarios entre el número de líneas totales, entendiendo aquí LoC en su acepción más común (cualquier sentencia, salvo las de comentario y las líneas en blanco):

$DoC = CLoC/(LoC + CLoC)$

Esta métrica suele utilizarse como indicador de la legibilidad de un código o de su facilidad de mantenimiento ya que, en teoría, cuantos más comentarios haya en el código fuente, más fácil será de entender (y en consecuencia mantener) un cierto software.

Las métricas de tamaño se utilizan a menudo como umbrales e indicadores de posibles problemas. Como ejemplo de ello se pueden citar los estudios que corroboran el límite de líneas a partir del cual un código es menos legible. McCabe, concretamente, establece que si una función excede de 60 líneas, que es más o menos lo que entra en una página impresa, la función es más propensa a tener errores.

2.1.2. Complejidad del software

Aunque en los últimos años se han propuesto métricas de complejidad específicas para el modelo de programación y diseño orientado a objetos en esta sección estudiaremos los dos conjuntos de métricas clásicas para medir la complejidad del software: las **métricas de McCabe** y la **ciencia del Software** de Halstead. Ambas fueron desarrolladas como indicadores para la estimación del coste, esfuerzo, número de defectos y facilidad de mantenimiento del software, entre otros atributos. Hoy en día son aún populares tanto por el hecho de que no son específicas de ningún lenguaje de programación en concreto, como por la existencia de numerosas herramientas software que las implementan y dan soporte.

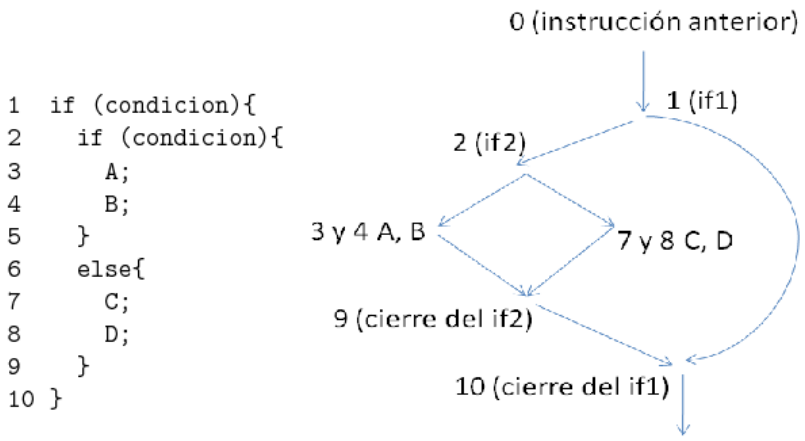

Figura 1: Ejemplo complejidad ciclomática

McCabe y las métricas de complejidad.- La **complejidad ciclomática** de McCabe $v(g)$ se basa en la cuenta del número de caminos lógicos individuales contenidos en un código de programa. Para calcular la complejidad ciclomática, el programa (o fragmento de programa) se representa como un grafo cuyos nodos son las instrucciones, y los posibles caminos sus aristas. Una vez representado de este modo, la complejidad ciclomática se calcula como:

$$v(g) = e - n + 2 \tag{1}$$

donde e representa el número de aristas y n el número de nodos, esto es, el número de posibles caminos del código. Teniendo en cuenta que la psicología establece el límite normal de elementos simultáneos en la memoria de trabajo humana en 7 ± 2, diremos que un módulo es más complejo, y por tanto potencial causa de problemas, en función de la cercanía de su complejidad ciclomática a dicho límite.

La parte izquierda de la figura 1 muestra un fragmento de código con 2 sentencias selectivas (`if`) anidadas, que se representan según el grafo de la parte derecha. Dado que en dicho grafo se identifican 3 posibles caminos, se dice que la complejidad ciclomática es 3 (aplicando la fórmula 1, se obtiene

Cuadro 5: Clasificación de modulos según su complejidad ciclomática

Complejidad ciclomática	Complejidad del código
1-10	Simple, sin riesgos
11-20	Algo complejo, riesgo moderado
21-50	Complejo, riesgo elevado
51+	Muy difícil de probar, riesgo muy alto

$v(g) = 9 - 8 + 2 = 3$).

La complejidad ciclomática se utiliza, por ejemplo, en la *metodología de pruebas estructurada*. Otros usos de la complejidad de McCabe son medir la complejidad de la integración de módulos, evaluar la dificultad de automatizar las pruebas de un código o incluso medir su fiabilidad. En la tabla 5 se muestran los umbrales comúnmente aceptados para clasificar módulos de software, datos basados no sólo en la psicología humana sino también en estudios realizados por el propio McCabe a partir de cierto número de proyectos de software. En dicha tabla se indica qué rangos de complejidad son potencial causa de problemas y qué tipo de módulos software, por tanto, deberían ser probados más cuidadosamente.

Si bien la complejidad ciclomática mide la cantidad del código, poco o nada dice acerca de su calidad. Para medir específicamente dicho atributo, y con el objetivo final de evitar lo que comúnmente se conoce como "código spaghetti" (código no estructurado), McCabe definió la denominada **complejidad esencial**. La complejidad esencial `ev(g)` se calcula de modo similar a la complejidad ciclomática, pero utilizando un grafo simplificado en el que se eliminan las construcciones básicas de la programación estructurada (secuencias, estructuras selectivas e iteraciones). Dijkstra demostró que cualquier programa puede expresarse utilizando únicamente estas construcciones. Por tanto, el grafo resultante una vez eliminadas dichas estructuras mostrará el grado de alejamiento con respecto al canon o, en otras palabras, su grado de imperfección. La figura 2 muestra cómo una estructura selectiva pura (en la parte izquierda de la figura) se simplifica, pero no es posible hacer lo mismo con la estructura selectiva que aparece en la parte inferior de ese mismo grafo ya que existen saltos (`goto`) desde el interior de la misma

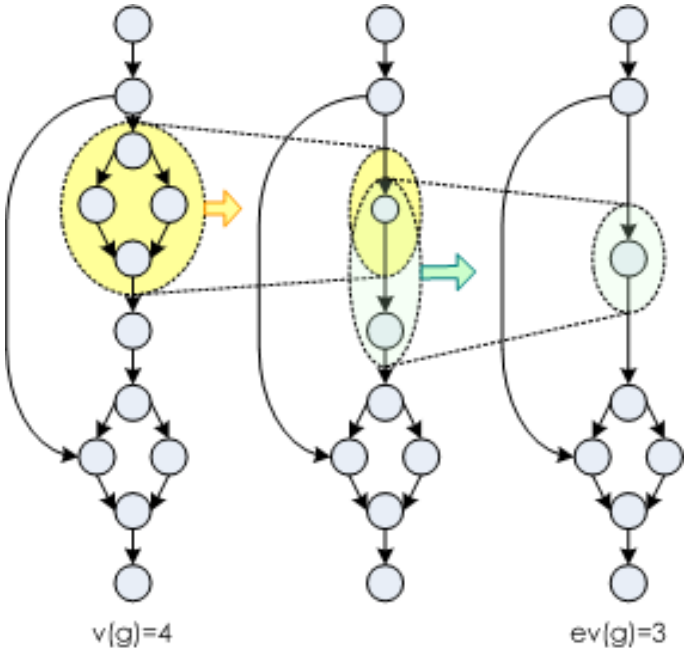

Figura 2: Complejidad ciclomática ($v(g)$) vs. complejidad esencial ($ev(g)$)

a otras zonas del código, incumpliendo así la regla de Dijkstra. El rango de esta métrica es $1 \leq ev(g) \leq v(g)$, por lo que cuanto más cercano a uno sea su valor, más estructurado será el código.

La **complejidad del diseño del módulo** `iv(g)` es otra medida de complejidad que utiliza un grafo simplificado para sus cálculos. En este caso, la reducción se aplica en las llamadas a otros módulos. Se trata de una medida muy útil en las pruebas de integración.

Halstead y su ciencia del software.- A finales de 1970, Halstead desarrolló un conjunto de métricas conocidas en su conjunto como la *ciencia del software de Halstead*. Estas métricas se basan en último término en computar los *operadores* y *operandos* de un programa:

- Los operadores son las palabras reservadas del lenguaje (tales como `if`, `while` o `for`), los operadores aritméticos (`+`, `-`, `*`, etc.), los de asignación (`=`, `+=`, `*=`, etc.) y los operadores lógicos (AND, OR, etc.)

- Los operandos son las variables, los literales y las constantes del programa.

Halstead propone diferentes medidas que basa en el cálculo previo del número de operadores y operandos únicos, y del número total de operadores y operandos. Para calcular todos estos valores utiliza la siguiente notación:

- n_1, número de operadores únicos que aparecen en un código.
- N_1, número total de ocurrencias de operadores.
- n_2, número de operandos únicos que aparecen en un código.
- N_2, número total de ocurrencias de operandos.

Así, si por ejemplo consideramos el siguiente fragmento de programa:

```
if (MAX < 2){
   a = b * MAX;
   System.out.print(a);
}
```

obtendremos los siguientes valores:

- $n1 = 6$ (if, { }, System.out.print(), =, *, <)
- $N1 = 6$ (if, { }, System.out.print(), =, *, <)
- $n2 = 4$ (MAX, a, b, 2)
- $N2 = 6$ (MAX, 2, a, b, MAX, a)

A partir de estos 4 parámetros Halstead elabora diferentes métricas para diversas propiedades de los programas, independientemente –como hemos dicho– del lenguaje de programación utilizado. Las más relevantes de estas métricas son las siguientes:

- **Vocabulario** ($n = n_1 + n_2$). El vocabulario es una medida de la complejidad de las sentencias de un programa a partir del número de operadores y operandos únicos. Se basa en el hecho de que un programa que utiliza un número reducido de elementos muchas veces será, según Halstead, menos complejo que un programa que emplea un mayor número de elementos.

- **Longitud** ($N = N_1 + N_2$). La longitud es una medida del tamaño de un programa: cuanto más grande, mayor será la dificultad para comprenderlo. Se trata de una medida alternativa a la simple cuenta de líneas de código y casi igual de fácil de calcular. N es sin embargo más sensible a la complejidad, porque no asume que todas las instrucciones son igualmente fáciles o difíciles de entender.

- **Volumen** ($V = N \cdot log_2(n)$). El vocabulario se define como el número de bits necesarios para codificar un programa en un alfabeto que utiliza un único carácter para representar todo operador u operando. Mientras que la longitud es una simple cuenta del total de operadores y operandos, el volumen da un peso extra al número de operadores y operandos únicos. Por ejemplo, si dos programas tienen la misma longitud N pero uno de ellos tiene un mayor número de operadores y operandos únicos, que naturalmente lo hacen más difícil de entender y mantener, éste tendrá un mayor volumen.

- **Esfuerzo mental** ($E = V/L$). El esfuerzo ofrece una estimación del trabajo requerido para desarrollar un programa dividiendo su volumen por el nivel del lenguaje (L), siendo este L un indicador que varía en función de si se está utilizando un lenguaje de alto o bajo nivel. El esfuerzo crece por tanto con el volumen, pero decrece a medida que se utiliza un lenguaje de mayor nivel. Así por ejemplo, una llamada a un procedimiento podría tener un valor $L = 1$ en Java, mientras que en COBOL podría ser $0,1$ y en lenguaje ensamblador $0,01$. Según estudios empíricos, E es una mejor medida de la facilidad para comprender un programa de lo que lo es N. La motivación original de Halstead al crear

esta métrica fue representar el esfuerzo mental necesario (en términos de operaciones mentales de discriminación) para escribir un programa de longitud N.

Las métricas de Halstead son tan ampliamente utilizadas como criticadas por su comprometida validación empírica, ya que muchas de las métricas se definen en función del número de discriminaciones mentales, un valor ciertamente difícil de definir y evaluar. Debe tenerse en cuenta además que se trata de métricas pensadas para medir programas una vez se tiene el código completo, lo cual impide utilizarlas para realizar estimaciones, por ejemplo. Son, sin embargo, útiles durante las actividades de prueba pues permiten identificar aquellos módulos potencialmente problemáticos de acuerdo a su complejidad.

Existen muchas otras métricas relacionadas con el tamaño y la complejidad del código. En general estas métricas son utilizadas para predecir esfuerzo en las etapas de construcción y mantenimiento, para detectar posibles módulos defectuosos y por tanto para determinar dónde invertir el esfuerzo de pruebas. Dado que se trata de decenas de métricas, el describirlas todas se escapa a la intención de esta obra. Se recomienda al lector interesado en profundizar en el estudio de estas métricas, acudir a los trabajos de Fenton y Pfleeger o de Zuse incluidos en la sección de referencias bibliográficas.

2.1.3. Medición de la documentación

Además de las métricas estudiadas en las secciones anteriores, centradas en la medición de atributos del propio código en sí, existen otras que miden aspectos relacionados con el mismo, como por ejemplo la documentación. Las métricas de la documentación permiten precisar la documentación generada en cada una de las distintas fases del ciclo de vida. Así, en la fase de requisitos es posible contar el número de requisitos, el número de casos de uso o el número de cambios en los requisitos por mes, entre otros. Además, otras métricas provenientes de la lingüística, más generales y no específicas de la ingeniería del software, permiten medir por ejemplo la legibilidad de un

documento. Entre este tipo de métricas se incluyen las siguientes:

Índice de Gunning-Fog . El índice de Gunning-Fog es una medida del nivel de estudios que una persona debe tener para comprender texto. Esta métrica, que se diseñó para textos en inglés y específicamente para el sistema de educación americano, se basa en el siguiente proceso. En un texto de aproximadamente 100 palabras, se calcula el índice Gunning-Fog con la siguiente ecuación:

$$Fog = 0{,}4 \times \left(\frac{Num\_Palabras}{Num\_Frases} + 100 \times \left(\frac{Num\_Palabras\_Complejas)}{Num\_Palabras} \right) \right)$$

donde se suelen considerar como $Palabras\_Complejas$, aquellas con 3 o más sílabas. El rango de esta métrica está entre 1 y 12, un valor que indica el nivel de educación necesario para comprender un texto dentro del sistema americano de educación. No obstante, desde una perspectiva profana, la interpretación del índice es sencilla: cuanto menor es el índice, más fácil de entender es el texto.

Índice de la facilidad de lectura de Flesch . Señala cómo de fácil es la comprensión de un texto a través del cómputo de la siguiente ecuación:

$$Flesch = 206,835 - 1,015 \times \left(\frac{total\_palabras}{total\_frases} \right) - 84,6 \times \left(\frac{total\_sílabas}{total\_palabras} \right)$$

Este índice, cuyo rango oscila entre 0 y 100, se interpreta del siguiente modo: cuanto menor es el índice, más difícil será la comprensión del texto. Valores por encima de un 80 %-90 % indican que el texto puede ser comprendido por la práctica totalidad de la población.

Aunque existen muchas otras métricas para evaluar la facilidad de comprensión de textos, las dos descritas se encuentran entre las más utilizadas. Son de hecho tan populares, que están implementadas en procesadores de texto ampliamente conocidos y utilizados como Microsoft Word y Google Docs.

2.1.4. Medición de la reutilización

La reutilización de documentación, diseños, código, casos de prueba, etc. es de primordial importancia a la hora de desarrollar nuevos proyectos. Como sabemos, la reutilización afecta de forma positiva a la calidad y productividad de un proyecto. Desde el punto de vista de la gestión de un proyecto de desarrollo resulta esencial conocer el grado de reutilización, para así poder saber cuánto código será producido y cuánto será reutilizado a partir de desarrollos anteriores, de bibliotecas externas, etc. Pese a todo, la definición y medición del grado de reutilización no es trivial.

A menudo se clasifican los módulos, clases, etc. mediante rangos nominales con valores como *completamente reutilizado*, *ligeramente modificado* (si el número de líneas modificado es menos del 25 %), *muy modificado* (si el número de líneas modificadas es más del 25 %) y *nuevo* (cuando el módulo, función o clase es completamente nuevo). Sin embargo, es más común emplear como métrica general de la reutilización el *porcentaje de reutilización* de líneas de código, que se calcula de las siguiente manera:

$$Porcentaje\_reutilizacion = \frac{LoC\_reutilizadas}{Total\_LoC} \times 100$$

2.1.5. Eficiencia

Muchas veces necesitamos medir la eficiencia de un software. En sistemas de tiempo real, por ejemplo, es obligatorio garantizar una respuesta dentro de un determinado rango de tiempo y por tanto, el modo natural y habitual de evaluar estos programas es medir el tiempo que demoran en ejecutarse. Pero el tiempo de ejecución es una métrica externa, ya que un cierto programa puede ejecutarse en más o menos tiempo dependiendo de la entrada que se le proporcione, del compilador utilizado o de la plataforma sobre la que se ejecute. Si lo que se desea es medir la eficiencia examinando únicamente el código del programa (es decir, mediante métricas internas), podemos contar el número de operaciones que efectúa el algoritmo dada una entrada. Esta

métrica se ha formalizado matemáticamente en lo que se denomina ***orden*** (O grande) de un algoritmo. Veamos un ejemplo.

Existen diferentes algoritmos de búsqueda. La *búsqueda lineal* depende sólo del número de elementos a buscar, n, lo que matemáticamente se denota como $O(n)$. Este $O(n)$ –que se lee "complejidad de orden n"– representa el hecho de que, en el peor de los casos, necesitaremos recorrer los n elementos de la entrada para encontrar el elemento buscado. Sin embargo, un algoritmo de *búsqueda binaria* en conjuntos de elementos ordenados –que utiliza un método de búsqueda similar al modo en que los humanos buscamos una palabra en un diccionario– establece la cota del número de elementos analizados necesarios para encontrar el buscado sensiblemente menor: $log_2 n$. Es decir, el orden del algoritmo de búsqueda binaria es $O(log_2 n)$, lo que indica que este tipo de búsqueda es significativamente más eficiente que la búsqueda lineal.

Aunque el análisis de algoritmos se escapa al ámbito de este libro (ya que es parte de lo que se denomina *algorítmica*), desde el punto de vista de la ingeniería del software es importante para estimar y acotar el tiempo de ejecución de los programas. Así, y pese a que existen casos en que con entradas muy pequeñas un algoritmo exponencialmente acotado puede ser más eficiente que uno polinomial (aquellos cuyo tiempo de ejecución depende de una función polinómica), casi siempre preferiremos algoritmos polinomialmente acotados. La figura 3 muestra gráficamente cómo para una entrada pequeña n_1 el tiempo de ejecución de un algoritmo exponencialmente acotado es menor que el de algoritmos menos complejos, si bien para valores grandes (n_2, n_3 y siguientes) los tiempos de respuesta devienen inaceptables conforme se incrementa el tamaño de la entrada.

2.2. Medidas del producto: atributos externos

Como ya se comentó en la sección 2, los atributos externos tratan de medir características que dependen de la visión externa del producto, asociándose generalmente esta visión con la calidad del producto.

Entre los modelos de calidad más conocidos se encuentran los de McCall

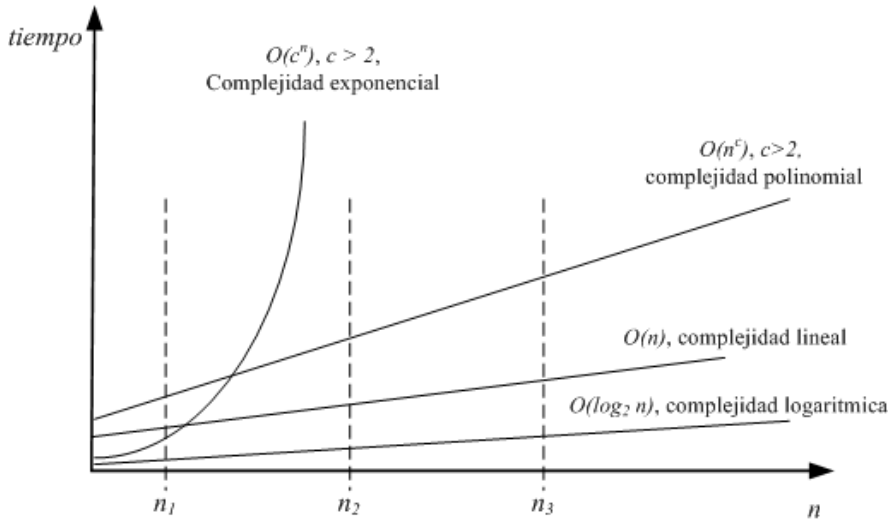

Figura 3: Ejemplos de orden de función.

y Bohem, y el estándar ISO 9126 (y su evolución en el estándar ISO/IEC 14598). En dichos modelos, a cada entidad (producto, proceso o recurso) se asignan unos atributos externos que en en el mundo de la calidad se denominan *factores de calidad*. Algunos de estos factores son la *corrección*, la *fiabilidad*, la *facilidad de uso* o la *eficiencia*. Los factores de calidad se pueden a su vez dividir en otros sub-factores (atributos internos) que se pueden calcular directamente de las entidades, como por ejemplo la legibilidad, el acoplamiento, o la eficiencia, por nombrar sólo unos pocos. La figura 4 resume cómo se definen los modelos de calidad citados en función de sus atributos externos e internos.

Como se muestra en la figura 4, toda entidad generada durante el ciclo de vida puede tener su modelo específico de calidad generado a partir de modelos de calidad como los mencionados anteriormente. Por ejemplo, a la hora de medir requisitos podemos considerar la legibilidad de los documentos, pero no la eficiencia, mientras que a la hora de definir un modelo de calidad para el código, podríamos considerar tan importante la legibilidad como la eficiencia. Además, muchas entidades serán artefactos generados y después refinados en sucesivas iteraciones a lo largo del ciclo de vida. Por ejemplo,

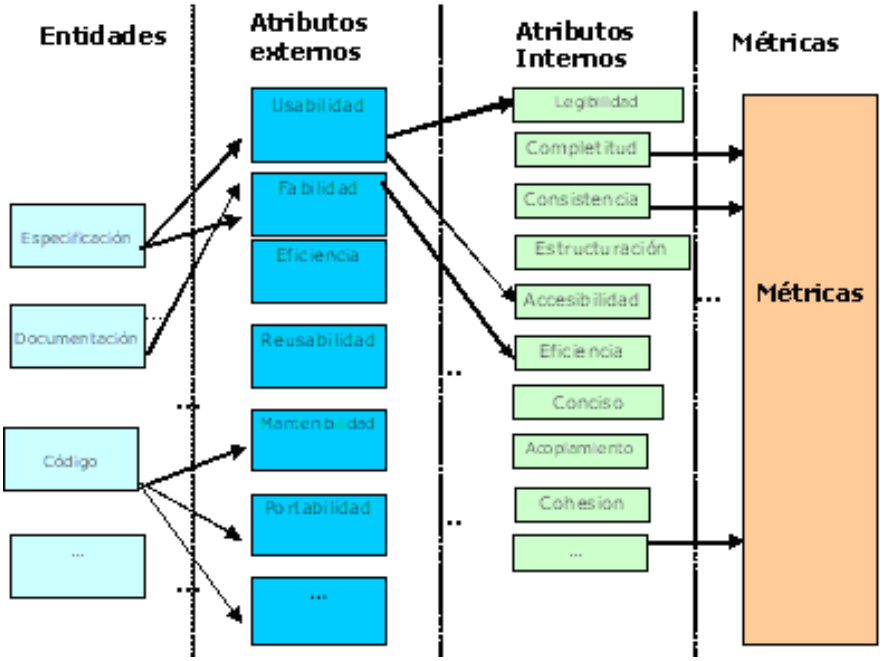

Figura 4: Factores, sub-factores y métricas en modelos de calidad

los modelos de diseño de alto nivel se transformarán en modelos de diseño de bajo nivel y estos últimos pueden tener distintas versiones según se van realizando iteraciones.

2.3. Medidas relacionadas con el proceso

A diferencia de las métricas vistas hasta el momento, basadas en la medición de atributos del producto, las métricas que veremos en esta sección evalúan el proceso de desarrollo de los productos software. Ejemplos de métricas internas de este tipo incluyen el tiempo de desarrollo del producto, el esfuerzo que conlleva dicho desarrollo o el número de incidentes, defectos o cambios en las distintas fases del ciclo. Otras métricas externas del proceso incluyen la facilidad de observación, la estabilidad del proceso o el coste del proceso, por citar sólo algunas.

Hoy en día se dedican muchos esfuerzos a la mejora (y por tanto eval-

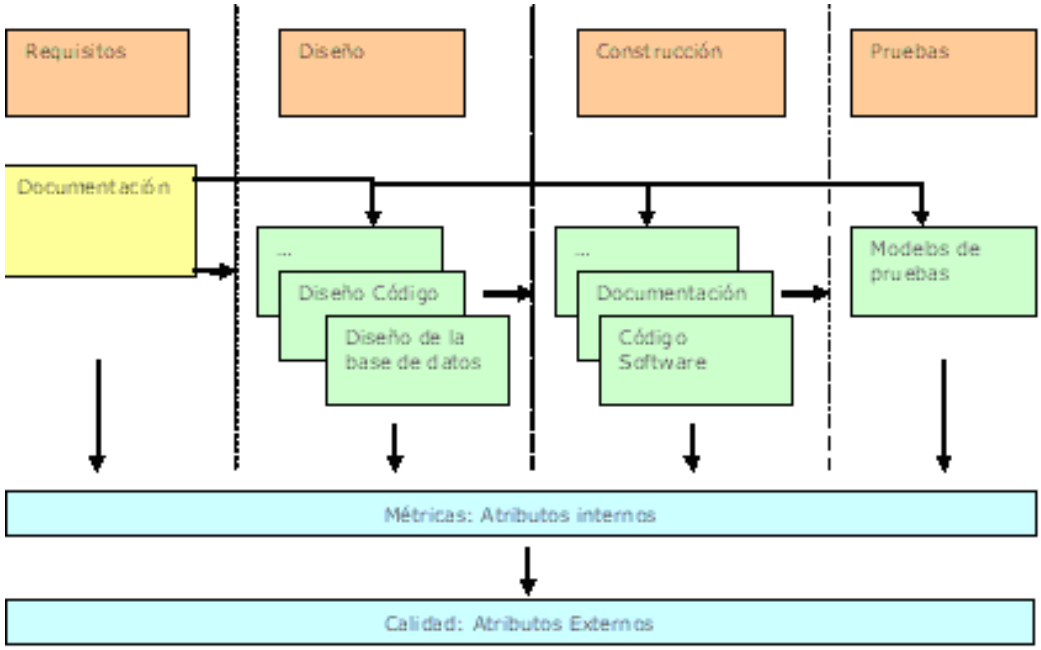

Figura 5: La medición a lo largo del ciclo de vida del software

uación) de procesos, pues mejorando la calidad del proceso, se mejora la calidad del producto. Entre los más conocidos de mejora de procesos se encuentran CMMI, SPICE, ISO 9000, Six Sigma y COBIT. Dentro de estos, CMMI y SPICE por ejemplo, agrupan procesos con sus respectivas métricas en distintos niveles de madurez, con la idea de que una organización vaya progresivamente mejorando su procesos.

2.4. Medidas relacionadas con los recursos

Resultan de gran utilidad en la ingeniería del software las medidas que se llevan a cabo sobre los recursos. Dentro de esta categoría veremos métricas específicas para cada uno de los principales recursos, que son el *personal*, las *herramientas* utilizadas en el proceso de desarrollo, los *materiales* y los *métodos*. Para todos ellos podremos medir el coste, utilizado principalmente por los gestores de proyectos para decidir en qué recursos invertir el presupuesto disponible. Con el propósito de que la inversión lleve a la máxima productivi-

dad –definida genéricamente como la relación entre la cantidad de trabajo obtenido y el esfuerzo invertido– y siempre dependiendo de las circunstancias del proyecto, se invertirá más en un tipo de recurso (por ejemplo en personal) o en otro (por ejemplo herramientas).

El concepto de productividad es muy utilizado en las medidas relacionadas con los recursos. La definición general que hemos esbozado no aspira a definir el término de manera cuantificable, aunque sin embargo, a menudo será necesario definirla de este modo. Determinaremos de aquí en adelante la productividad en la ingeniería del software con la siguiente expresión:

$$Productividad = \frac{Tamaño}{Esfuerzo}$$

donde el $Tamaño$ vendrá dado normalmente bien por el número de líneas de código (LoC), o bien por el número de puntos de función, mientras que el esfuerzo se medirá, en la mayoría de los casos, en personas por mes. En este cálculo de la productividad obviamente influyen todos los tipos de recursos. En el caso particular de las herramientas esto resulta especialmente evidente –por ejemplo diferentes compiladores pueden tardar distinto tiempo en compilar el mismo programa– pero también es cierto para el resto de recursos materiales, sobre todo para el personal. A continuación se describen algunos ejemplos de medidas de recursos.

2.4.1. Medición del personal

Dentro de las métricas relacionadas con el personal podemos distinguir entre aquellas que miden atributos de la persona como unidad individual y aquellas cuyo objeto de medición son los diferentes grupos o equipos que pueden formarse dentro del personal. A nivel individual, las medidas más utilizadas y comunes son el coste (el salario), la productividad individual, y la experiencia. A la hora de medir esta última, por ejemplo, se deben tener en cuenta distintas habilidades, tanto técnicas como de gestión.

Desde una perspectiva colectiva, es posible tomar, dentro de un equipo

de personal, medidas indirectas como la media o la moda de alguna de las métricas individuales. La media de edad de un equipo, por ejemplo, pertenece a este tipo pues se obtiene a partir de medidas directas, en este caso la edad de cada uno de los individuos que conforman el equipo. Además de las métricas indirectas, también son habituales las métricas relacionadas con la *estructura y comunicación de los equipos*, muy a tenerse en cuenta durante la gestión del proyecto. A este respecto, es interesante mencionar las métricas Hewlett-Packard para medir la *complejidad de las comunicaciones*. Dichas métricas se basan en un grafo de comunicaciones en el que se representan los miembros del equipo como nodos y las comunicaciones entre ellos como aristas, el cual sirve de base para la definición de las siguientes métricas:

- El *tamaño* es el número de individuos del equipo.
- La *densidad de comunicaciones* es el ratio entre el número de arcos y nodos, es decir, la proporción entre el tamaño del equipo y el número de comunicaciones que se producen entre ellos.
- El *número de líneas de comunicación* es la cuenta del número de aristas del grafo.
- El *nivel de comunicaciones* es una medida de la impureza del árbol. Esta métrica se apoya en un concepto de la teoría de grafos, la impureza del árbol, que no es sino un indicador de cuán lejos (o cerca) está un cierto grafo de ser un árbol. Matemáticamente, dado un grafo G, su impureza $m(G)$ viene dada por la ecuación:

  $$m(G) = \frac{2 \cdot (e - n + 1)}{(n - 1) \cdot (n - 2)}$$

  donde e es el número de aristas y n el número de nodos.
- El *nivel individual de comunicaciones* es el número de individuos con el que se comunica un determinado miembro del equipo.

- El *nivel medio de comunicaciones* es la media del nivel individual de comunicaciones.

La figura 6 muestra un ejemplo de grafo y las métricas de complejidad de las comunicaciones obtenidas a partir de la información en el mismo.

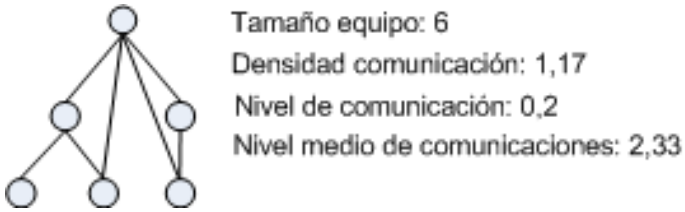

Figura 6: Ejemplo de métricas de complejidad de las comunicaciones

En este tipo de grafos, el incremento del número de vías de comunicación, indica una mayor complejidad. En un grafo con forma de lista, que representaría a un miembro del equipo que sólo se comunica con otro miembro del equipo, el número de líneas de comunicación sería $n - 1$. Según se vayan incrementando las vías de comunicación, se puede llegar al máximo nivel de comunicación, que es aquél en el que todos los miembros del equipo se comunican con el resto de miembros, en cuyo caso las líneas de comunicación serían $n \cdot (n - 1)/2$.

2.4.2. Medición de las herramientas

En los desarrollos modernos se emplean multitud de herramientas. Los gestores de proyecto son los encargados de seleccionar las herramientas hardware y software necesarias para llevar a cabo las distintas actividades del ciclo de vida del software, para lo cual han de tener en cuenta un buen número de variables. El coste de adquisición es sólo una de ellas y, tal vez demasiado a menudo, es la única que se tiene en cuenta cuando se trata de adquirir herramientas. Para asegurarse el éxito en sus decisiones, los gestores de proyecto deberían sopesar también factores tan importantes como la productividad de las herramientas o el tiempo que las personas que van a utilizarlas necesitarán

para aprender a usarlas adecuadamente, todas las cuales pueden (y deben) ser medidas.

Entre las métricas a tener en cuenta a la hora de comprar o seleccionar hardware, se encuentran la potencia de cálculo o la capacidad de almacenamiento, por citar sólo dos de las más habituales. En el caso del software, podríamos tener en cuenta el tipo de licencia (código abierto o propietario), su facilidad de uso, el tiempo de aprendizaje o la posibilidad de reutilizarlo en otros proyectos, entre otros.

2.4.3. Medición de los materiales

Los costes relacionados con las oficinas, el material fungible, los desplazamientos a las sedes de nuestros clientes, y otros como éstos, influyen en el coste final del producto y por tanto, en la productividad. Otras métricas que podrían ser tenidas en cuenta son, por ejemplo, la temperatura ambiente en la oficina trabajo o el tipo de oficina (colectivas vs. individuales), todo lo cual podría utilizarse para analizar de productividad en función de los diferentes entornos.

2.4.4. Medición de los métodos

Dentro de los recursos, es importante tener en cuenta la clasificación de los métodos usados en un proyecto, por ejemplo si se utilizaron métodos formales para los requisitos, o si se siguieron métodos estructurados o métodos orientados a objetos en el diseño. Los datos medidos a este respecto pueden ayudar a la selección de los métodos más apropiadas en futuros proyectos. De hecho, es común que las organizaciones de desarrollo con altos niveles de madurez lleven a cabo lo que comunmente se conoce como análisis *post-mortem*, es decir, un análisis de cómo ha ido el proyecto una vez éste ha finalizado.

3. Metodologías y estándares para la medición

Siempre se mide con la idea de mejorar, de aumentar la calidad de las entidades involucradas en los procesos de producción del software. Sin embargo, es imposible (o al menos poco práctico) medir todos los atributos de todas las entidades. Además, programadores, gestores de proyectos y usuarios pueden tener diferentes puntos de vista de lo que significa *calidad*. Es por ello por lo que debemos determinar qué medir basándonos en metodologías y modelos de calidad bien definidos. En otras palabras, necesitamos saber qué queremos mejorar para saber qué medir. Entre los métodos más conocidos se encuentra GQM (*Goal-Question-Metric*) y el estándar de IEEE 1061-1998, que junto con otros de más reducida difusión, se explican a continuación.

3.1. Método Objetivo-Pregunta-Métrica (GQM)

El método Objetivo-Pregunta-Métrica, o GQM (*Goal-Question-Metric*) como es más conocido, fue desarrollado para alcanzar los objetivos de calidad requeridos por la NASA en los años 70. GQM ayuda a identificar, centrar, documentar y analizar un número reducido de métricas con la intención de mejorar un objetivo que puede ser tanto del producto como del proceso o sus recursos. El método se basa en 3 niveles (ver figura 7):

- El primero es el *nivel conceptual*, en el que se identifica un objetivo de calidad. Dicho objetivo, que puede estar relacionado con la evaluación o la mejora, será el propósito de la medición en relación a una entidad –producto, proceso o recurso– y desde un punto de vista específico (gestor, desarrollador, operador, mantenimiento, etc.).
- El segundo nivel, llamado *nivel operacional*, divide el objetivo en una serie de preguntas que caracterizan a la entidad.
- Finalmente, el *nivel cuantitativo* especifica el conjunto de métricas necesarias para poder responder a las preguntas planteadas en el segundo

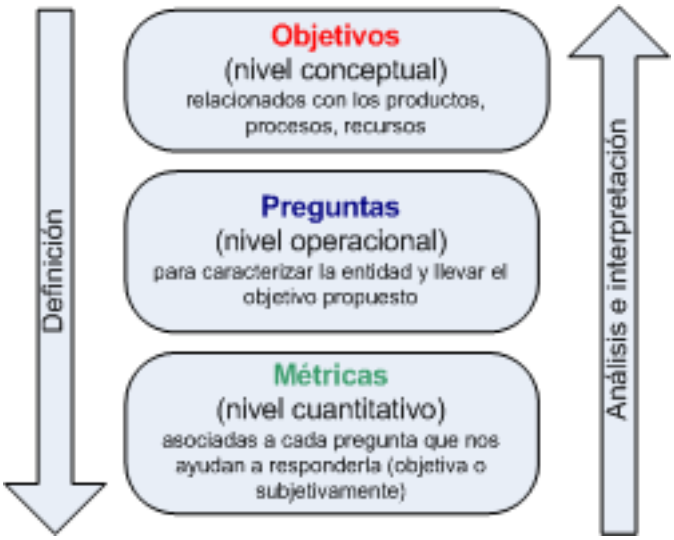

Figura 7: Niveles GQM

nivel.

Estos 3 niveles se pueden resumir en un árbol de objetivos, tal y como se muestra en el ejemplo de la figura 8.

El funcionamiento del método se basa en el refinamiento progresivo de un conjunto de objetivos de negocio (G-Goals) que se establecen como partida. Tomando dichos objetivos como entrada y mediante el planteamiento de preguntas (Q-Questions), se obtienen finalmente un conjunto de métricas (M-Metrics) específicas que permitirán medir los objetivos enunciados. Más en detalle, diremos que es necesario:

1. Desarrollar un conjunto de objetivos tanto de la organización y de sus divisiones organizativas, como del proyecto. Establecer métricas que permitan evaluar la productividad y calidad deseadas para cada uno de ellos.

2. Basándose en modelos de calidad, generar preguntas que cubran los objetivos desarrollados con la mayor compleción posible, y siempre de forma cuantificable.

Figura 8: Ejemplo de árbol GQM

3. Especificar las métricas necesarias a recabar para poder responder a las preguntas y asegurar que los procesos y productos se ajustan a los objetivos.

4. Desarrollar los mecanismos que permitan llevar a cabo las mediciones.

5. Realizar las mediciones y evaluar y analizar los datos durante la ejecución del proyecto para poder llevar a cabo acciones correctivas si fuera necesario.

6. Analizar los datos una vez terminado el proyecto (*análisis post-mortem*), para evaluar así el cumplimiento de los objetivos y recomendar mejoras basadas en las lecciones aprendidas.

Finalmente, para que los objetivos estén bien especificados y puedan ser entendidos por todas las personas involucradas en un proyecto, éstos se suelen especificar en una plantilla con los siguientes puntos:

- *Objeto*: Es la entidad que se va a estudiar. Los posibles valores a marcar en la plantilla serían *proceso*, *producto*, *recurso*, *modelo* o *métrica*, si bien también otros son posibles.

Cuadro 6: Ejemplo plantilla GQM

Objeto	Calidad del software producido
Con el propósito de	Entender
Con respecto a	Mantenibilidad
Desde el punto de vista de	Personal/Equipo mantenimiento
En el contexto de	Proyecto XYZ

- *Propósito*: Indica cuál es la motivación para medir dicho objetivo. El porqué puede ser *analizar*, *evaluar*, *predecir*, *entender*, *mejorar*, etc. Por ejemplo: medición de un cierto modelo (objeto de la medición) con objeto de estudiar posibles mejoras en el mismo (propósito).

- *Perspectiva*: Informa sobre cuál es el atributo de calidad propósito del objeto, o lo que es lo mismo, con respecto a qué se toma la medida. Algunos valores posibles son la *corrección*, la *fiabilidad* o el *coste*. Por ejemplo: medición de la corrección (perspectiva de la medición) del producto (objeto de la medición).

- *Punto de vista*: Hace referencia a la perspectiva desde la que se lleva a cabo el estudio. Así, es posible estudiarlo desde el punto de vista del usuario, del cliente, del gestor del proyecto o de los programadores, por poner algunos ejemplos.

- *Entorno*: Es el contexto en el que se ejecuta. Este entorno puede hacer referencia a factores de personal, del proyecto, de los métodos, de las herramientas, etc.

La tabla 3.1 muestra como ejemplo de todo lo anterior una plantilla para un caso de mejora de la calidad del código.

3.2. El estándar para métricas de calidad del software IEEE 1061-1998

El estándar IEEE 1061-1998, denominado Metodología para métricas de calidad del software (IEEE *Standard for a Software Quality Metrics Method-*

ology), define una metodología para llevar a cabo la identificación, implementación, y evaluación de métricas para procesos y productos software válida para todas las fases del ciclo de vida independientemente del modelo utilizado (cascada, iterativo e incremental, etc.). El estándar comprende cinco pasos que deben realizarse de manera iterativa, ya que los resultados de uno de ellos pueden necesitarse en pasos subsiguientes. El conjunto de tareas a llevar a cabo es el siguiente:

1. Establecer los requisitos de calidad del software. Para lo cual es necesario llevar cabo las siguientes tareas:

   a) Elaborar un listado con los posibles requisitos de calidad, utilizando para ello toda la información disponible: estándares, requisitos de los contratos o compras (tales como garantías o fechas de entrega), etc.

   b) Identificar la lista de requisitos de calidad a partir de la lista de posibles requisitos del paso anterior. Dicha lista se obtendrá evaluando la importancia de cada uno y eliminando posibles contradicciones.

   c) Cuantificar los factores de calidad. A cada factor de calidad se le debe asignar al menos una métrica con sus respectivos valores. Por ejemplo, si un requisito fundamental es la *fiabilidad* en un sistema Web, una métrica podría ser la "Disponibilidad diaria" con un valor de 95 %, o lo que es lo mismo disponibilidad de 1.368 minutos sobre los 1.440 minutos que componen las 24 horas de un día.

2. Identificar las métricas de calidad del software mediante la aplicación de un marco de métricas de calidad, la realización de un análisis sobre costes y beneficios y el establecimiento de las garantías necesarias para implantar dichas métricas.

3. Implementar las métricas de calidad. En este paso se definen los procedimientos de medición, se hace un prototipo del proceso de medición

y finalmente se lleva a cabo la medición en sí.

4. Analizar los resultados. Consiste en interpretarlos, analizando si los requerimientos de calidad se ajustan a los requisitos definidos en el la especificación.

5. Evaluar los resultados. Para ello se aplican criterios y metodologías de evaluación.

El estándar no prescribe, de manera explícita, ninguna métrica específica, limitándose a facilitar y promover el uso de métricas dentro de las organizaciones y en el contexto de proyectos con el objetivo final de mejorar la calidad del software producido.

3.3. PSM y el estándar ISO/IEC 15939

Tanto el estándar ISO/IEC 15939 (Proceso de medición del software) como el PSM (*Practical Software Measurement*, Medición práctica del software) en el cual está basado, establecen las actividades y tareas necesarias del proceso de medición, ayudando a identificar, definir, seleccionar, aplicar y mejorar la medición de software dentro de un proyecto general o de la estructura de medición de una empresa. El estándar se compone de dos aspectos, (i) el modelo de información de la medición y (ii) el proceso de medición propiamente dicho, de forma que pueda ser integrado en procesos generales.

El modelo de información de la medición define los términos de uso común relativos a la medición del software y la relación entre las necesidades de información, los tipos de medidas o métricas (por ejemplo los indicadores necesarios, métricas directas e indirectas) y las entidades a medir (en este estándar son procesos, productos, proyectos y recursos).

El proceso de medición se compone, como muestra la figura 9, de 4 actividades fundamentales:

- **Establecer y mantener el compromiso de medición**. Esta actividad implica el establecimiento de un compromiso empresarial para

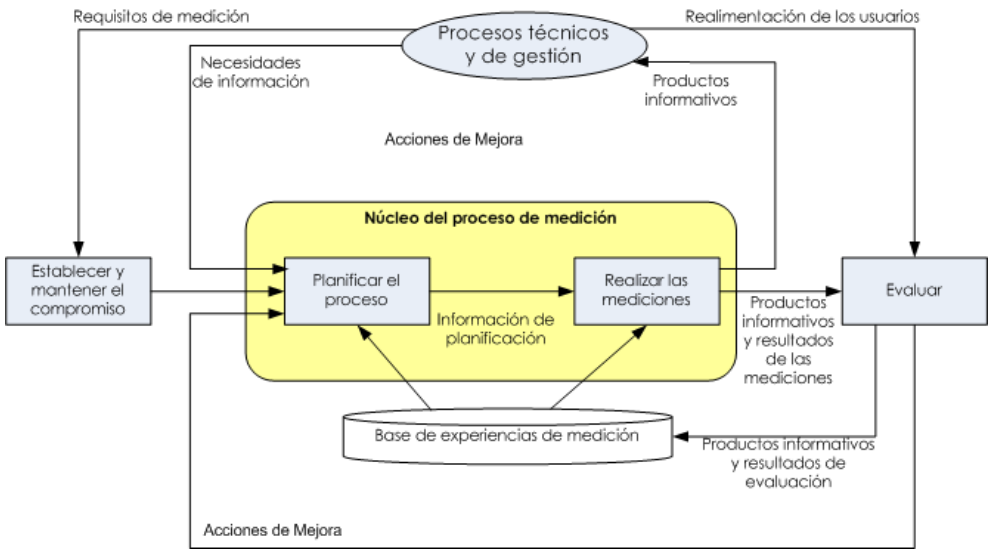

Figura 9: Actividades ISO/IEC 15939

realizar mediciones, compromiso que será necesario establecer al más alto nivel y con pocas o ninguna dependencia de personas concretas. Esto es así porque para recuperar la inversión realizada será necesario mantener este compromiso durante un cierto tiempo durante el que muchas cosas podrían cambiar en la organización.

- **Planificar el proceso de medición**. Esta actividad consiste básicamente en la selección de las medidas de acuerdo con los objetivos y características de la organización, definir los procedimientos de cómo y cuándo se van a realizar las medidas seleccionadas, quién las va a llevar a cabo, la frecuencia de medición de las mismas y cómo se analizarán.

- **Ejecutar el proceso de medición**. En esta actividad se realizan y almacenan las mediciones realizadas, posteriormente se analizan y con los datos obtenidos se desarrollan productos de información que puedan ser utilizados en procesos de decisión.

- **Evaluación de las medidas obtenidas**. Esta actividad consiste en evaluar tanto las mediciones realizadas como el proceso de medición en sí mismo, identificando mejoras potenciales.

3.4. Otras metodologías y estándares para la medición

Además de los dos citados, existen otros modelos y prácticas comúnmente citados en la bibliografía. A continuación se hace una breve reseña de ellos.

3.4.1. Los estándares ISO/IEC 14598 e ISO/IEC 9126

El estándar ISO/IEC 14598 (*Information Technology-Software Product Evaluation*) se compone de una serie de seis volúmenes dedicados a la medición y evaluación de la calidad de productos software desde distintos puntos de vista (desarrolladores, compradores y evaluadores). El estándar incluye actividades de proceso y se basa en un estándar anterior, el ISO/IEC 9126. Actualmente, ambos estándares comparten la misma terminología y están en proceso de convergencia hacia un único estándar que los englobe.

3.4.2. Vocabulario internacional de metrología – VIM

En su última edición, el estándar ISO/IEC Guide 99:2007 (más comunmente conocido como VIM) recopila un conjunto de definiciones y términos relacionados con la ciencia de la medición (metrología) en general y no sólo de la ingeniería del software. Además de esto, incluye diagramas conceptuales que muestran de manera más evidente las relaciones entre términos, ejemplos y todo tipo de información complementaria. El objetivo último es que sirva como referencia para otros estándares y metodologías relacionadas con la medición, armonizando así la dispersa nomenclatura actual.

3.4.3. AMI (*Aplicación de Métricas en la Industria*)

El método AMI, publicado por Kuntzmann-Combelles y sus colaboradores en 1996, combina el modelo de madurez CMM con el método GQM con objeto de crear un marco de trabajo para la mejora de procesos. Se trata de un método de claro enfoque iterativo, orientado a objetivos y cuantitativo, que involucra a todos los miembros de la organización integrando a su vez

a los gestores, de quienes necesita un compromiso firme. Cubre la totalidad del ciclo de mejora del proceso, y utiliza CMM (pero también otras normas de calidad como Bootstrap, SPICE o ISO 9001) para identificar posibles áreas débiles en el proceso de desarrollo. Esta información, junto con información sobre el entorno de la empresa y los objetivos específicos se utiliza para definir objetivos del proceso de software. Estos objetivos se evalúan, se refinan en sub-objetivos, de los cuales finalmente se obtienen métricas. El método establece además un plan de medición para recolectar datos, que serán posteriormente analizados y contrastados con el objetivo original. En función de los datos obtenidos se elabora un plan de acción para mejorar el proceso de desarrollo, para lo cual se definen nuevos objetivos y se vuelve a repetir así el ciclo.

4. Estudios empíricos

Antes de proseguir, recordemos la definición del término *Ingeniería del software* que propone el glosario IEEE de términos de ingeniería del software:

1. Ingeniería del software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, la operación y el mantenimiento del software. Esto es, la aplicación de la ingeniería al software.

2. El estudio de enfoques como los mencionados en el punto (1).

La mayoría de este libro cubre aspectos relacionados con la primera parte de la definición. Sin embargo, en esta sección daremos una visión general de la segunda: el estudio de la ingeniería del software en sí misma. Mostraremos cómo la evaluación de nuevas técnicas, procesos, herramientas o métricas de ingeniería del software se puede llevar a cabo mediante estudios empíricos, algo ampliamente utilizado en otras disciplinas como la Química, la Biología o la Psicología.

Los estudios empíricos nos ayudan a demostrar la mejoría introducida por el uso de una nueva técnica o herramienta y a responder preguntas del

estilo ¿Realmente funciona mejor Java que C++ en entornos de robótica? ¿Con qué método de especificación de requisitos se genera un código con menor cantidad de errores en sistemas de tiempo real, con los casos de uso o con los métodos formales? Lo que buscamos con los métodos empíricos son respuestas cuantitativas a este tipo de preguntas.

La experimentación es una disciplina relativamente joven dentro de la ingeniería del software. Por tanto, no es extraño ver que el conocimiento en la ingeniería del software se adquiere todavía, desde el punto de vista epistemológico, bien por influencia de una cierta autoridad o persona prestigiosa ("si tal persona lo dice, será la mejor forma"), bien por inercia ("si IBM lo hace así, es seguro que funciona") o bien por parecer que se utiliza lo más actual o lo que está de moda ("si todo el mundo ha desechado el enfoque estructurado y adoptado el modelo de orientación a objetos, nosotros también lo haremos aunque no entendamos muy bien las razones para hacerlo"). Sin embargo, esta situación hace un flaco favor a la consolidación de la ingeniería del software como disciplina de ingeniería de pleno derecho. En la ingeniería del software, como en otras disciplinas, las nuevas teorías deberían venir respaldadas por datos experimentales que las sustenten y confirmen su validez. Todo ello, por supuesto, asumiendo las características y limitaciones propias de la ingeniería del software con respecto a disciplinas como la Física o la Biología donde la experimentación tiene siglos de historia.

Poco a poco, pero inexorablemente, la importancia de la experimentación dentro la ingeniería del software está ganando terreno. Ya se han empezado a publicar libros al respecto, y comienzan a celebrarse cada vez con mayor frecuencia conferencias dedicadas. Este hecho queda patente en varios estudios:

- En un estudio llevado a cabo por Zelkowitz y Wallace en 1997, se analizaron aproximadamente 600 artículos científicos del área de la informática. El estudio demostró que sólo el 10 % de los artículos incluían algún tipo de experimentación controlada, y que aproximadamente el 30 % de los artículos no incluía experimentación ninguna.

- Tichy y sus colaboradores realizaron un estudio similar en 1995 sobre 400 artículos. Sus conclusiones indican que aproximadamente un 40 % de los artículos no incluye experimentación, lo cual es muy elevado en comparación con otras disciplinas de ingeniería en las cuales este número desciende hasta el 10-15 %.

Además, y hasta la fecha, se han formulado muy pocas leyes en la ingeniería del software, siendo las más conocidas las leyes de la evolución del software de Lehman. Recientemente se han planteado diversas leyes, teorías e hipótesis para tratar de acercar la ingeniería del software al **método científico**. Según dicho método la adquisición de conocimiento se basa en la obtención de información mediante observación, a lo cual sigue la propuesta de teorías e hipótesis, la evaluación de las hipótesis y si es posible, la replicación de estudios anteriores. No obstante, otros métodos de experimentación son igualmente aplicables a la ingeniería del software, como el método ingenieril, el método empírico y el método analítico.

Para llevar a cabo la evaluación serán necesarios instrumentos que nos permitan adquirir datos, identificar factores clave de estudio y lógicamente obtener resultados. Estos instrumentos son los siguientes:

- **Encuestas**: Son estudios retrospectivos cuya intención es documentar relaciones y resultados, siendo una de las herramientas más útiles para recabar un buen número de datos que posteriormente serán analizados.
- **Casos de estudio**: Son empleados para identificar y documentar factores clave que pueden afectar los resultados de una actividad.
- **Experimentos formales**: Se emplean para, de forma controlada y rigurosa, investigar cuantitativamente aquellos factores que afectan a las actividades a realizar.

Los tres instrumentos enumerados, encuestas, casos de estudio y experimentos formales, se utilizan tanto para estudios cualitativos como cuantitativos. Los estudios cualitativos buscan la interpretación de un fenómeno en

su entorno natural, recabando información entre las personas involucradas. Los estudios cuantitativos, por su parte, buscan medir la influencia de una variable en un fenómeno, es decir la relación causa-efecto entre ambos (por ejemplo la relación entre la profundidad en la jerarquía de clases en un diseño orientado a objetos y el número de errores en el código de una clase).

Los tres tipos de evaluación empírica son por tanto valiosos en la ingeniería del software. A modo de resumen, se puede afirmar que las encuestas cualitativas sirven como base para la formulación de hipótesis, los experimentos formales confirman o rechazan las hipótesis y los casos de estudio sirven como contraste para determinar si las hipótesis pueden o no generalizarse.

Todas estas técnicas se explican brevemente a continuación con el objeto de tratar de identificar cuál de ellas resultará más apropiada en cada momento. Téngase en cuenta que la elección de la técnica apropiada es siempre dependiente no sólo del propósito, sino también de los posibles participantes y del tipo de análisis de resultados que se quiera llevar a cabo.

4.1. Encuestas

Las encuestas son un método que se emplea para recabar datos que están en la memoria de los entrevistados. Por ejemplo, tras introducirse un nuevo método o herramienta en una organización, puede evaluarse su efectividad a través de un cuestionario que deberán cumplimentar los empleados que la utilizan.

Las encuestas forman parte de lo que se conoce como investigación a gran escala (*research in-the-large*), pues sirven para recopilar un gran número de datos acerca de diferentes personas y proyectos. Esto además permite que las conclusiones puedan generalizarse, siempre y cuando la selección de los entrevistados haya sido realmente aleatoria y significativa.

Una vez establecido el objeto (u objetos) de interés, y dependiendo tanto de la población seleccionada como del porcentaje de respuestas esperadas, las entrevistas pueden diseñarse para ser realizadas cara a cara, por correo

o a través de Internet (por Web o email). Dependiendo del método utilizado existen tres maneras de llevar a cabo las entrevistas:

- Entrevistas *estructuradas*, en las que las preguntas corren a cargo del entrevistador. Dichas preguntas están fijadas de antemano, son las mismas para todos los entrevistados y no se modifican durante la entrevista.
- Entrevistas *no estructuradas*, donde el entrevistado puede ser la fuente tanto de respuestas como de preguntas. El objetivo es obtener la mayor información posible del entrevistado.
- Entrevistas *semi-estructuradas*, formadas por un conjunto de preguntas abiertas que facilitan el que el entrevistado pueda ofrecer información no prevista por el entrevistador.

Aunque generalmente las encuestas son cualitativas, el tipo de estudio (cuantitativo o cualitativo) depende de cómo se haya diseñado la encuesta. Además, dependiendo del tipo de encuesta, el análisis de los datos extraídos de las mismas puede utilizarse tanto para generar teorías como para confirmarlas.

Existen guías tanto para la correcta planificación y realización de los cuestionarios como para su análisis. En éste último caso, pueden utilizarse técnicas estadísticas si lo que se desea evaluar son los resultados de cuestionarios cuantitativos. Para datos cualitativos y generación de teorías pueden utilizarse por ejemplo el *método de comparación constante*, que consiste en añadir códigos a los resultados, agrupar la información según los códigos y sintetizar las conclusiones.

Un ejemplo clásico de estudio mediante encuesta, fue el realizado por Beck y Perkins en 1983. En dicho estudio se analizaron las prácticas de la ingeniería del software en la industria y se buscaron correlaciones entre dichas prácticas y determinados problemas que aparecían en las instalaciones de usuario de los sistemas una vez en producción.

4.2. Casos de estudio

Los casos de estudio constituyen una técnica de investigación que se basa en la observación de entornos reales para contrastar o evaluar datos, encontrar relaciones y descubrir tendencias. En la ingeniería del software se usan mucho para comparar métodos o herramientas, pudiendo ser tanto cuantitativos como cualitativos. Así, por ejemplo, si se desea introducir una nueva herramienta en una organización pero no se está seguro de si el incremento en la productividad merecerá la pena el esfuerzo en inversión, podría diseñarse un caso de estudio como parte de un proyecto piloto cuyo único objeto fuese comparar la productividad de la nueva herramienta con la de la herramienta actual.

Los casos de estudio se clasifican como "investigación en lo típico" (*research in the typical*) pues reflejan preferentemente la información de un proyecto representativo en lugar de toda la casuística con la que nos podríamos encontrar. La diferencia entre los casos de estudio y los experimentos formales es que en los experimentos las variables son manipuladas, mientras que en los casos de estudio simplemente se analizan dentro de su contexto. Los casos de estudio tienen la ventaja de que son más fáciles de diseñar que los experimentos formales, aunque son difíciles de replicar pues se realizan generalmente en entornos industriales concretos. Además, al no tener los investigadores ningún control sobre las variables, el establecimiento de relaciones causales entre ellas es más difícil y por tanto la generalización de las conclusiones más complicada.

4.3. Experimentación formal

La experimentación formal busca medir relaciones causales con la mayor exactitud posible, o lo que es lo mismo, establecer el grado de influencia de unas variables en otras. Para ello es necesario configurar entornos en los que se tenga un alto nivel de control sobre las variables para, modificándolas, medir sus efectos. Fenton y Plfeeger clasifican la experimentación formal

como investigación a pequeña escala (*research in the small*) debido a la dificultad de controlar todos los posibles factores. Se suelen realizar en entornos académicos (por ejemplo con estudiantes en un laboratorio) o en organizaciones con un grupo reducido de trabajadores, precisamente por la dificultad mencionada, lo que ha hecho que a estos estudios también se los conozca como experimentos *in vitro*.

Los experimentos formales son por tanto más difíciles de diseñar y más costosos que las encuestas o los casos de estudio. La experimentación formal en la ingeniería del software, al igual que en otras disciplinas como la Física o la Medicina, ayuda a confirmar teorías, explorar relaciones causa-efecto entre variables, corroborar o rechazar creencias sobre métodos, procesos y tecnologías, evaluar métricas, etc. Dado que el factor humano es especialmente importante dentro la ingeniería del software, los tipos de experimentos muchas veces se asemejan a los realizados en las ciencias sociales. Un famoso ejemplo de este tipo de experimentos en la ingeniería del software es el realizado por Scanlan para descubrir si sus estudiantes en la asignatura "Estructuras de datos" tenían diferentes niveles de comprensión de la lógica de un programa en función de la herramienta de representación utilizada. Su estudio demostró que los estudiantes preferían los diagramas de flujo, pues les ayudaban a entender mejor los programas que el pseudocódigo.

La experimentación formal sigue un proceso definido a lo largo de 5 pasos, el cual que se muestra detallado a continuación:

1. Definición: En primer lugar se debe declarar la intención del experimento, es decir, cuáles son sus objetivos, en qué contexto se llevará a cabo, cuál es su propósito, etc. Es frecuente el uso del método GQM (ver sección 3.1) para sistematiza, ordenar y considerar todos los aspectos del experimento.

2. Planificación: En esta fase se diseña el experimento mediante el enunciado de las hipótesis, la selección de variables de estudio (las cuales determinarán los tipos de análisis estadísticos que se pueden aplicar), la selección de los sujetos y finalmente el diseño del experimento propia-

mente dicho. Es especialmente relevante la elección de la instrumentación y la determinación del método de evaluación.

3. Operación: es la fase de preparación del experimento, que consiste en realizar la selección de participantes, formularios y guías para que los participantes entiendan el proceso, entrenar a los participantes si es necesario, y en definitiva realizar todas las acciones necesarias para poner en marcha el experimento. En esta fase suele realizarse un estudio piloto para encontrar errores en el diseño y subsanarlos para que el experimento pueda llevarse a cabo.

4. Interpretación: A la hora de analizar e interpretar los resultados, el tipo de análisis está condicionado por la hipótesis del experimento, el tipo de variables seleccionadas según su escala y el diseño experimental. El primer paso es caracterizar los datos utilizando estadística descriptiva, para posteriormente realizar el *test de hipótesis* con el fin de confirmar o rechazar las hipótesis enunciadas.

5. Conclusiones y presentación de resultados: Al final del experimento se escribe un informe comentado del proceso seguido. En este informe se debe enfatizar por qué es relevante la hipótesis del estudio y cómo se han solventado o minimizado las amenazas a la validez. Dado que a partir de las conclusiones del experimento otros podrían querer poner en marcha nuevos estudios para confirmar o rebatir los hallazgos, es necesario incluir toda la información que permita crear posibles réplicas del experimento.

La figura 10 muestra de manera resumida los diferentes pasos de la experimentación formal que acabamos de enumerar.

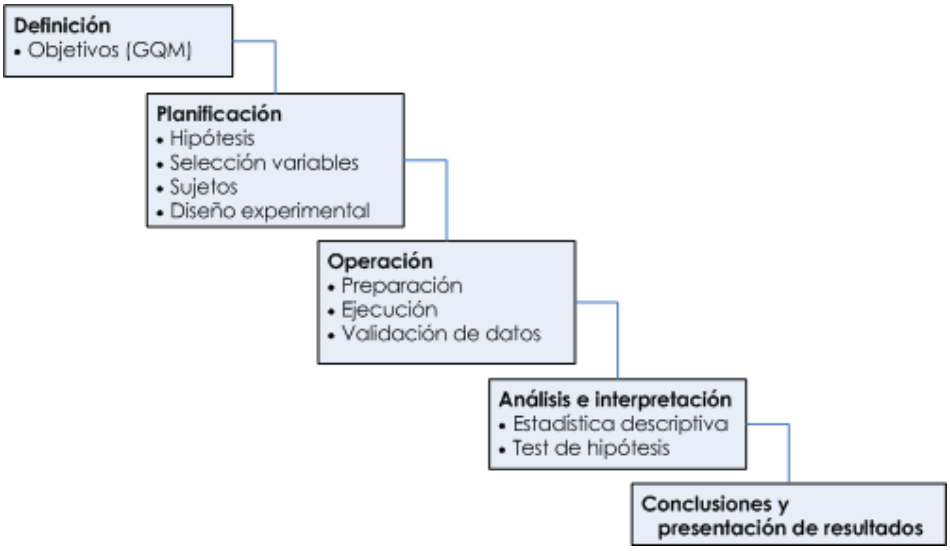

Figura 10: Proceso que sigue la experimentación formal.

Ingeniería del software basada en la evidencia
La ingeniería del software basada en la evidencia es un novedoso modelo de experimentación que intenta acercarse a la forma de proceder de otras áreas, particularmente de la medicina, a través de la determinación de cuáles son las cosas que funcionan, en qué casos y cuándo. El modelo se apoya en la recolección y análisis sistemáticos de todos los datos empíricos disponibles sobre un determinado fenómeno. El objetivo último es conocer dicho fenómeno con mayor profundidad y perspectiva de los que los estudios individuales podrían proporcionar, ya que éstos a menudo están sesgados por factores como el contexto en que se llevan a cabo o las personas que participan. El elemento central de la ingeniería del software basada en la evidencia es la revisión sistemática de la literatura, una forma de estudio muy utilizada en medicina. Según este método, los resultados de estudios basados en un único ensayo no pueden considerarse ni lo suficientemente seguros ni lo suficientemente fiables como para ser generalizados. Este principio se aplica a la ingeniería del software, en palabras de Kitchenham, mediante "la provisión de medios a través de los cuales se integren las mejores evidencias de la investigación, la experiencia práctica y los valores humanos en el proceso de toma de decisiones que se lleva a cabo durante el desarrollo y mantenimiento de software".

5. Resumen

Hemos tratado pues el tema de la medición desde un punto de vista global de la disciplina, desde el cual las entidades principales a medir son los productos, los procesos y los recursos. Para cada entidad hemos distinguido entre métricas directas e indirectas, o lo que es lo mismo, entre aquellas que podemos medir directamente y aquellas que dependen de algún factor externo. Se han visto ejemplos de métricas directas relacionadas con el código, unas relativas a su complejidad, otras sobre la reutilización del código, algunas transversales pero aplicables a la ingeniería del software, como las de legibilidad de los documentos. Entre los ejemplos de métricas indirectas se encuentran los relacionados con la calidad como fiabilidad, usabilidad o portabilidad.

Una parte importante se ha dedicado a proporcionar una visión general de los estándares, métodos y metodologías más relevantes relacionados con la medición, como son el GQM, el estándar IEEE 1061-1998, ISO/IEC 15939 o VIM, entre otros. Estos modelos nos ayudan a saber qué medir, de entre las múltiples magnitudes que podrían ser objeto de medida. Sabiendo que no es posible medir todos los atributos de todas las entidades, hemos decidido determinar qué medir basándonos en metodologías y modelos de calidad bien definidos.

Se ha introducido además la experimentación en la ingeniería del software, que aunque aún es un área joven dentro de las disciplina, su importancia está creciendo rápidamente como mecanismo de justificación y evaluación de muchas de las prácticas de la disciplina. Y por qué no decirlo, también como forma de rechazar otras prácticas que no cuentan con sustento experimental suficiente. No nos cabe la menor duda de que la medición y experimentación es el camino por el que la ingeniería del software transitará en los próximos años y gracias al cual está madurando.

6. Notas bibliográficas

La obra fundamental sobre métricas es *Software Metrics — A Rigorous and Practical Approach* [4]. Este libro ha influido a la mayoría de los libros posteriores que han tratado la medición, incluido el presente. De hecho muchos de los conocimientos descritos se han tomado de esta importante obra.

Otra obra esencial, esta vez sobre experimentación, es *A Framework for Software Measurement* [15], donde se realiza un estudio en gran profundidad y detalle sobre numerosas métricas de software. Cualquiera interesado en ampliar sus conocimientos sobre la medición en este campo debería consultarlo.

En español existe un trabajo muy completo sobre la medición y experimentación en la ingeniería del software titulado "Medición para la gestión en la ingeniería del software" [3]. Muchos de los contenidos de los se tratan en dicho libro con mayor amplitud, por lo que resulta un excelente complemento para aquellos que quieran adquirir un conocimiento más profundo sobre medición.

Sobre evaluación y validación de métricas, el artículo "*Towards a framework for software measurement validation*" [7] resulta aún hoy día la referencia más citada. El apartado que hemos dedicado a la evaluación plasma de hecho la filosofía expresada por sus autores. Otra referencias interesantes son *Software Engineering Measurement* [10] que toca aspectos de la teoría de la medición, y las propiedades de validación de métricas publicadas por Weyuker [12].

A pesar de que ya han transcurrido unos cuantos años desde su publicación, y a pesar de estar pensadas para la programación estructurada, las métricas de complejidad de McCabe [9] y Halstead [**?**] se encuentran implementadas en multitud de herramientas y por tanto, ampliamente disponibles en la literatura de la ingeniería del software.

Para profundizar en el método Objetivo-Pregunta-Métrica (GQM), aparte de la referencia principal de Basili y Rombach [2], existe un libro muy interesante titulado *The Goal/Question/Metric Method: A Practical Guide for*

Quality Improvement of Software Development [11].

Actualmente existe multitud de bibliografía relacionada con la medición y experimentación en la ingeniería del software. Un libro introductorio es el de Juristo y Moreno [6]. Entre los artículos introductorios enfatizando la importancia de los estudios empíricos se encuentra el de R. Glass [5]. Además, el artículo de Zelkowitz y Wallace [14] mencionado en el apartado de experimentación y en el que se reporta un estudio sobre 600 artículos científicos dentro del área de la informática. Como ya dijimos, los resultados obtenidos mostraron que aproximadamente el 30 % de los artículos no incluían experimentación ninguna aunque ésta era necesaria y que sólo el 10 % de los artículos incluían algún tipo de experimentación controlada. Tichy *et al.* [**?**] llegan a conclusiones parecidas con un estudio sobre 400 artículos.

En relación a la experimentación en la ingeniería del software, Basili y sus colaboradores [1] fueron los primeros en iniciar el área y aunque no es un artículo introductorio fácil de leer, es el origen de y cita referenciada en los libros sobre experimentación en la ingeniería del software. Para una estudio más detallado se referencia al lector a la obra ya mencionada de de Fenton y Pfleeger [4]. Otro libro introductorio que trata la experimentación es el de Wholin *et al.* [13].

Relacionados con experimentación, los trabajos de B. Kitchenham son especialmente relevantes, siendo de destacar una serie de artículos sobre experimentación donde se detallan, entre otros, guías para la selección de las técnicas más apropiadas en la evaluación empírica, *Evaluating Software Engineering Methods and Tools* (publicado en 6 volúmenes).

Finalmente, varios autores comúnmente nombrados en el área han publicado una guía para llevar a cabo la experimentación en la ingeniería del software evitando los típicos errores [8].

Referencias

[1] V R Basili, R W Selby, and D H Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engieering*, 12(7):733–743, 1986.

[2] V.R. Basili and H.D. Rombach. The tame project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, Jun 1988.

[3] José Javier Dolado and Luis Fernandez. *Medición para la gestión en la ingeniería del software*. Ra-ma, 2000.

[4] N. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Publishing, 1998.

[5] Robert L. Glass. The realities of software technology payoffs. *Commun. ACM*, 42(2):74–79, 1999.

[6] N. Juristo and A.M. Moreno. *Basics of Software Engineering Experimentation*. Kluwer Academic, 2001.

[7] B. Kitchenham, S.L. Pfleeger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–944, Dec 1995.

[8] B.A. Kitchenham, S.L. Pfleeger, L.M. Pickard, P.W.Jones, D.C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions onSoftware Engineering*, 28(8):721–734, Aug 2002.

[9] T J McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[10] John C. Munson. *Software Engineering Measurement*. CRC Press, Inc., Boca Raton, FL, USA, 2002.

[11] Rini van Solingen and E. Berghout. *The Goal/Question/Metric Method: a Practical Guide for Quality Improvement of Software Development*. McGraw-Hill, Maidenhead, 1999.

[12] E.J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988.

[13] C. Wholin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

[14] M.V. Zelkowitz and D. Wallace. Experimental validation in software engineering. *Information and Software Technology*, 1997.

[15] Horst Zuse. *A Framework for Software Measurement*. DeGruyter Publisher, 1998.