CAPÍTULO 4

MEDICIÓN EN LA ORIENTACIÓN A OBJETOS

Daniel Rodríguez y Rachel Harrison [1]
School of Computer Science, Cybernetics & Electronic Engineering
University of Reading, UK.

En este capítulo se examinan un conjunto de métricas de diseño en la orientación a objeto como medio para evaluar la calidad de este tipo de sistemas. La construcción de software mediante el uso de este paradigma, no garantiza por sí mismo la calidad y las métricas tradicionales no se adecuan bien. Por lo tanto, se han definido métricas especificas para este paradigma. El conjunto de métricas aquí descritas son las definidas por Abreu, Chidamber y Kemerer, y un subconjunto de las definidas por Lorenz y Kidd. Éstas han sido agrupadas acorde a la granularidad que desean medir: sistema, acoplamiento, herencia, clase y método. En cada métrica se comenta su intrepretación, guías de uso, problemas y posibles mejoras en algunas de ellas.

4.1. INTRODUCCIÓN

Hay varios tipos de métricas que pueden utilizarse en la en la realización de proyectos de software para gestionar, predecir y mejorar la calidad de software. En este capítulo nos centraremos en métricas relacionadas con la calidad de los diseños Orientados a Objetos (OO). Es sabido que no sólo con el uso de este paradigma se consige la calidad, ya que se dan ejemplos de sistemas OO no robustos, con poca mantenibilidad o donde el grado de reusabilidad es mínimo. La finalidad del uso de métricas es evaluar sistemas para conseguir alta calidad y robustez.

[1] Parte de este trabajo fue realizado en la Universidad de Southampton, UK.

Se describen un conjunto de métricas relevantes definidas por Abreu [Abreu y Melo, 1996], Chidamber y Kemerer [Chidamber y Kemerer, 1994], algunas de Lorenz y Kidd [Lorenz y Kidd, 1994]. Las cuales se explican de acuerdo con la clasificación definida por [Archer y Stinson, 1995], tratando de abarcar todas las posibles características de los sistemas OO. Cada métrica es descrita considerando su definición, cómo se calcula, umbrales o valores a los cuales debería limitarse, su utilidad y cómo es de apropiada. Con los umbrales y guías se detectan clases o métodos que difieren sustancialmente de unos valores medios. Estos valores pueden indicar problemas futuros, abstracciones mal concebidas o malas implementaciones, por lo que esas clases o métodos son candidatos para ser revisados o reescritos. Muchos de estos umbrales han sido recomendados por [Lorenz y Kidd, 1994] basándose en algunos projectos de IBM en C++ y Smalltalk.

Este trabajo está dividido en los siguientes contenidos. En la sección 4.2, se considera el porqué las métricas deben estar basadas en modelos de calidad como medio para determinar sus objetivos. En la sección 4.3, se describen una serie de propiedades que deberían de cumplir las métricas para ser consideradas válidas. En la sección 4.4, se resumen métricas encontradas en la literatura que han tenido relevancia en la orientación a objetos. Estas métricas son comentadas para tener una guía y ayuda a la hora de seleccionarlas y utilizarlas.

4.2. EL USO DE MÉTRICAS

El conjunto de métricas a usar debe dejar claro qué aspectos de la calidad son los que propone medir y a quién van dirigidos. Programadores, gestores y usuarios tienen diferentes puntos de vista de lo que significa calidad por lo que el conjunto de métricas a utilizar debería estar basado en un modelo de calidad bien definido, como por ejemplo, *Goal-Question-Metric* (GQM) [Basili y Rombach, 1988] o *Quality Management System* (QMS) [Kitchenham et al. 1986]). Tanto GQM como QMS ayudan a construir un modelo de requerimientos de calidad basándose en factores como usabilidad, mantenibilidad, reusabilidad, etc. Estos modelos flexibles ayudarán a clarificar qué aspectos de la calidad son considearados y porqué.

Algunas métricas tradicionales o con ciertas modificaciones pueden ser de utilidad en sistemas OO, especialmente a nivel de métodos. Sin embargo, para poder cuantificar características específicas de este paradigma como encapsulación, herencia, polimorfismo, paso de mensajes, etc., ha sido necesaria la creación de nuevas métricas.

4.3. VALIDACIÓN DE LAS MÉTRICAS

Métricas y mediciones, como correspondencias entre entidades del mundo real y números, deben de ser validadas tanto teórica como experimentalmente.

4.3.1. VALIDACIÓN TEÓRICA

Una métrica debe tener claro qué atributos de software van a ser medidos y cómo se va a hacer [Basili et al. 1995][Kitcheham et al. 1995][Fenton, 1994], sólo de esta forma, tendrán sentido y estarán relacionados con lo que se quiere medir.

[Kitchenham et al. 1995] describen una serie de propiedades que las métricas deben cumplir para ser válidas. Para métricas directas, es decir, aquellas para las que no es necesario ningun otro atributo o entidad (por ejemplo, longitud, número de defectos duracion de un proceso, etc.), son las siguientes:

1. Para que un atributo pueda ser medido, debe permitir que diferentes entidades sean distingibles una de la otra.
2. Una métrica debe cumplir la condición de representación.
3. Cada unidad que contribuye en una métrica válida debe ser equivalente.
4. Diferentes entidades pueden tener el mismo valor (dentro de los límites en los errores de medición).

Para métricas indirectas, es decir, aquellas que combinan varias métricas directas (por ejemplo, densidad de defectos en un modulo es igual al número de defectos del módulo entre longitud del módulo, etc.):

1. La métrica debe estar basada en un modelo explícitamente definido de relaciones entre ciertos atributos (generalmente relacionando atributos externos e internos).
2. El modelo debe ser dimensinalmente consistente.
3. La métrica no debe mostrar ninguna discontinuidad inesperada.
4. La métrica debe usar unidades y escalas correctamente.

La condición de representación, como está descrita por [Fenton y Pfleeger, 1997], asegura que la relacion M debe hacer corresponder entidades a números y relaciones empíricas a relaciones numéricas de tal forma que las relaciones empíricas son preservadas por las relaciones numéricas. Por ejemplo, A es mayor que B si y sólo si M(A) > M(B).

4.3.2. Validación experimental

Los métodos empíricos corroboran la validez de las métricas. Usando métodos estadísticos y experimentales se evalúan la utilidad y relevancia de las métricas [Basili et al. 1995] [Harrison et al. 1997] [Schneidewind, 1992].

Aunque en la bibliografía se recojen trabajos validando métricas a través de técnicas estadísticas y experimentales, aún es necesario mucho más para obtener mejores guías e interpretaciones.

4.4. MÉTRICAS ORIENTAS A OBJETOS

A continuación se describe un conjunto representativo de métricas tratando de abordar todos los posibles niveles de granularidad y características en sistemas OO basado en un marco de referencia definido por [Archer y Stinson, 1995].

4.4.1. Métricas a nivel de sistema

El conjunto de métricas MOOD (*Metrics for Object oriented Design*) definido por [Abreu y Melo, 1996] opera a nivel de sistema. Se refieren a mecanismos estructurales básicos en el paradigma de la orientación a objetos como encapsulación (MHF y AHF), herencia (MIF y AIF), polimorfismo (PF) y paso de mensajes (COF). En general, las métricas a nivel de sistema pueden derivarse de otras métricas usando métodos estadísticos como la media, etc. Éstas son utilizadas para identificar características del sistema.Este conjunto de métricas es explicado a continuación.

Proporción de métodos ocultos
(Method Hiding Factor –MHF-) [Abreu y Melo, 1996]

Definición MHF se define como la proporción de la suma de las invisibilidades de los métodos en todas las clases entre el número total de métodos definidos en el sistema. La invisibilidad de un método es el porcentaje sobre el número total de clases desde las cuales un método no es visible. En otras palabras, MHF es la proporción entre los métodos definidos como protegidos o privados y el número total de métodos.

$$MHF = \frac{\sum_{i=1}^{TC}\sum_{m=1}^{Md(Ci)}\left(1-V\left(M_{mi}\right)\right)}{\sum_{i=1}^{TC}M_d\left(C_i\right)}$$

donde:
$M_d(C_I)$ es el número de métodos declarados en una clase,

$$V\left(M_{mi}\right)=\frac{\sum_{j=1}^{TC}es\_visible\left(M_{mi},C_j\right)}{TC-1},$$

$$es\_visible\left(M_{mi},C_j\right)=\begin{cases}1 \Leftrightarrow \mathrm{j}\neq \mathrm{i}\wedge \mathrm{C_j\ puede\ llamar\ a\ M_{mi}}\\ 0 \qquad\qquad \text{caso contrario}\end{cases}$$

Es decir, para todas las clases, C_1, … ,C_n, un método cuenta 0 si puede ser usado por otra clase y 1 en caso contrario.
En lenguages como C++ o Java donde existe el concepto de método protegido, $V(M_{mi})$ se cuenta como una fracción entre 0 y 1:

$$V\left(M_{mi}\right)=\frac{DC(C_i)}{TC-1}$$

TC es el número total de clases en el sistema.

Propósito MHF se propone como una medida de encapsulación, cantidad relativa de información oculta.

Guías y comentarios [Abreu y Melo, 1996] han demostrado empíricamente que cuando se incrementa MHF, la densidad de defectos y el esfuerzo necesario para corregirlos debería disminuir.

Para calcular esta métrica, los métodos heredados no son considerados.

Proporción de atributos ocultos.
(Attribute Hiding Factor –AHF-) [Abreu y Melo, 1996]

Definición AHF se define como la proporción de la suma de las invisibilidades de los atributos en todas las clases entre el número total de atributos definidos en el sistema. La invisibilidad de un atributo es el porcentaje sobre el número total de clases desde las cuales un atributo no es visible. En otras palabras, AHF es la proporción entre los atributos definidos como protegidos o privados y el número total de atributos.

$$AHF = \frac{\sum_{i=1}^{TC}\sum_{m=1}^{Ad(Ci)}\left(1 - V\left(A_{mi}\right)\right)}{\sum_{i=1}^{TC} Ad\left(Ci\right)}$$

donde:

$$V\left(A_{mi}\right) = \frac{\sum_{j=1}^{TC} es\_visible\left(A_{mi}, C_j\right)}{TC - 1}$$

$$es\_visible\left(A_{mi}, C_j\right) = \begin{cases} 1 \Leftrightarrow \text{j} \neq \text{i} \wedge \text{C}_\text{j} \text{ puede llamar a } \text{A}_\text{mi} \\ 0 \qquad\qquad \text{caso contrario} \end{cases}$$

TC es el número total de clases en el sistema.

Propósito AHF se propone como una medida de encapsulación.

Guías y comentarios Idealmente esta métrica debe de ser siempre 100%, intentando ocultar todos los atributos. Las pautas de diseño sugireren que no se debe emplear atributos públicos, ya que se considera que violan los principios de encapsulación al exponer la implementación de las clases.

Para mejorar el rendimiento, a veces se evita el uso de métodos que acceden o modifican atributos (métodos *get/set*) accediendo a ellos directamente. En esta práctica debe extremarse la prudencia y evaluar si realmente los pros son mayores que los contras.

Proporción de métodos heredados
(Method Inheritance Factor –MIF-) [Abreu y Melo, 1996]

Definición MIF se define como proporción de la suma de todos los métodos heredados en todas las clases entre el número total de métodos (localmente definidos más los heredados) en todas las clases.

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

donde:

$M_a(C) = M_d(C_i) + M_i(C)$

y:

$M_d(C_I)$ es el número de métodos declarados en una clase

$M_a(C_I)$ es el número de métodos que pueden ser invocados en relación a C_i.

$M_i(C_i)$ es el número de métodos heredados (y no redefinidos C_i).

TC es el número total de clases en el sistema.

Propósito Sus autores proponen a MIF como una medida de la herencia y como consecuencia, una medida del nivel de reuso. También se propone como ayuda para evaluar la cantidad de recursos necesarios a la hora de testear.

Guías y comentarios El uso de la herencia se ve como un compromiso entre la reusabilidad que proporciona, y la comprensibilidad y mantenimiento del sistema (cf. Métricas a nivel de herencia).

Comentarios Evaluaciones experimentales relacionadas con la herencia se encuentran en [Daly et al. 1996] y [Harrison et al. 1999].

Proporción de atributos heredados.
(Attribute Inheritance Factor –AIF-) [Abreu y Melo, 1996]

Definición AIF se define como la proporción del número de atributos heredados entre el número total de atributos.

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

donde:

$A_a(C_i) = A_d(C_i) + A_i(C_i)$

y:

$A_d(C_i)$ es el número de atributos declarados en una clase.

$A_a(C_i)$ es el número de atributos que pueden ser invocados asociados a C_i.

$A_i(C_i)$ es el número total de atributos heredados (y no redefinidos) en C_i.

TC es el número total de clases en el sistema.

Propósito Al igual que MIF, AIF se considera un medio para expresar el nivel de reusabilidad en un sistema.

Guías y comentarios Demasiado reuso de código a través de herencia hace que el sistema sea más dificil de entender y mantener (cf. Métricas a nivel de herencia).

Proporción de polimorfismo.
Polymorphism Factor (PF) [Abreu y Melo, 1996]

Definición PF se define como la proporción entre el número real de posibles diferentes situaciones polimórficas para una clase C_i entre el máximo número posible de situaciones polimorfícas en C_i. En otras palabras, el número de métodos heredados redefinidos dividido entre el máximo número de situaciones polimórficas distintas.

$$PF = \frac{\sum_{i=1}^{TC} M_0(C_i)}{\sum_{i=1}^{TC}\left[M_n(C_i) \times DC(C_i)\right]}$$

Donde:

$$M_d(C_i) = M_n(C_i) + M_o(C_i)$$

y,
$M_n(C_i)$ es el número de métodos nuevos.
$M_o(C_i)$ es el número de métodos redefinidos.
$DC(C_i)$ es el número de descendientes de C_i.
TC es el número total de clases en el sistema.

Propósito PF es una medida del polimorfismo y una medida indirecta de la asociación dinámica en un sistema.

Guías y comentarios El polimorfismo es debido a la herencia. Abreu indica que en algunos casos sobrecargando métodos se reduce la complejidad y por lo tanto, se incrementa la mantenibilidad y comprensibilidad del sistema.
[Harrison et al. 1998] muestran como esta métrica no cumple todas las propiedades definidas en [Kitchenham et al. 1995] (cf. Validez de las métricas) para ser válida ya que en un sistema sin herencia, el valor de PF resulta indefinido, exhibiendo una discontinudad.

Proporción de acoplamiento.
(Coupling Factor –CF-) [Abreu y Melo, 1996]

Definición CF se define como la proporción entre el máximo número posible de acoplamientos en el sistema y el número real de acoplamientos no imputables a herencia. En otras palabras, indica la comunicación entre clases.

$$CF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} es\_cliente(C_i, C_j)\right]}{TC^2 - TC}$$

donde:

$$es\_cliente = \begin{cases} 1 \Leftrightarrow C_c \Rightarrow C_s \wedge C_c \neq C_s \\ 0 \quad caso\ contrario \end{cases}$$

La relación cliente-proveedor ($C_c \Rightarrow C_s$) representa que la clase cliente (C_c) contiene al menos una referencia no heredada de la clase proveedor (C_s).
TC es el número total de clases en el sistema.

Propósito El acoplamiento se ve como una medida del incremento de la complejidad, reduciendo la encapsulación y el posible reuso. Por tanto, limita la comprensibilidad y mantenibilidad del sistema.

Guías y comentarios CF puede ser una medida indirecta de los atributos con los cuales está relacionado: complejidad, falta de encapsulación, carecia de reuso, comprensibilidad y falta de mantenibilidad. [Abreu y Melo, 1996] han encontrado una correlación positiva: al incrementar el acoplamiento entre clases, se incrementa la densidad de defectos y la dificultad en la mantenibilidad.

4.4.2. Métricas a nivel de acoplamiento

Acoplamiento es el uso de métodos o atributos definidos en una clase que son usados por otra. Las clases tienen que interactuar entre ellas para formar sistemas, y esta interacción puede indicar su complejidad. Métricas representativas de acoplamiento son las siguientes:

Acoplamiento entre objectos
(Coupling Between Objects –CBO-) [Chidamber y Kemerer, 1994]

Definición CBO de una clase es el número de clases a las cuales una clase está ligada. Se da dependencia entre dos clases cuando una clase usa métodos o variables de la otra clase.
Las clases relacionadas por herencia no se tienen en cuenta.

Propósito Chidamber y Kemerer proponen que sea un indicador del esfuerzo necesario para el mantenimiento y en el testeo.

Guías y comentarios Chidamber y Kemerer sugieren que cuanto más acoplamiento se da en una clase, más dificil sera reutilizarla. Además, las clases con excesivo acoplamiento dificultan la comprensibilidad y hacen más dificil el mantenimiento por lo que será necesario un mayor esfuerzo y riguroso testeo.
Las clases deberían de ser lo más independientes posible, y aunque siempre es necesaria una dependencia entre clases, cuando ésta es grande, la reutilizacion puede ser más cara que la reescritura. Al reducir el acoplamiento se reduce la complejidad, se mejora la

modularidad y se promueve la encapsulación.

4.4.3. Métricas a nivel de herencia

La relación de herencia se ve como un compromiso, es decir, un acuerdo entre el reuso que proporciona y la dificultad en el entendimiento y mantenibilidad de los sistemas. Esto se debe a que cambios en la interfaz de una clase son heredados por todas las subclases. La bibliografía muestra un uso mínimo la herencia como norma general. [Cartwright y Shepperd, 1996] muestran escaso uso de la herencia en una compañía de telecomunicaciones en el Reino Unido. También, [Chidamber et al. 1998] muestran la baja utilización consciente de la herencia tras el analisis en un banco europeo.

La herencia puede ser sustituida por *delegación*, además reduce el acoplamiento ya que el cliente no conoce al proveedor y este puede ser reemplazado sobre la marcha. El mecanismo de delegación debe ser tenido en cuenta cuando el objetivo es la reutilización de código.

En general, no se recomienda el uso de *herencia multiple*. Ésta crea conflictos debido a la colisión de nombres en la propagación de atributos y operaciones. Para utilizar herencia múltiple se debe conocer de antemano como la trata el sistema que se está utilizando. Todos estos problemas llevan a que muchos lenguajes como Java, Ada95 o Smalltalk no soporten herencia múltiple.

Métricas representativas relacionadas con la herencia son:

Profundidad en árbol de herencia.
(Depth of Inheritance Tree –DIT-) [Chidamber y Kemerer, 1994]

Definición DIT mide el máximo nivel en la jerarquía de herencia. DIT es la cuenta directa de los niveles en la jerarquía de herencia. En el nivel cero de la jerarquía se encuentra la clase raíz.

Propósito Chidamber y Kemerer proponen DIT como medida de la complejidad de una clase, complejidad del diseño y el potencial reuso. Esto es debido a que cuanto más profunda se encuentra una clase en la jerarquía, mayor será la probabilidad de heredar un mayor número de métodos.

Guías y comentarios El uso de la herencia es visto como un compromiso ya que:

- Altos niveles de herencia indican objetos complejos, los cuales pueden ser difíciles de testear y reusar.
- Bajos niveles en la herencia pueden indicar que el código está escrito en un estilo funcional sin aprovechar el mecanismo de herencia proporcionado por la orientación a objetos.

En general, la herencia es poco utilizada y [Cartwright y Shepperd,

1996] encontraron una correlación positiva entre DIT y el número de problemas emitidos por el usuario, poniendo en duda el efectivo uso de la herencia.

Lorenz y Kidd sugieren un umbral de 6 niveles como indicador de un abuso en la herencia tanto en Smalltalk como en C++.

Sistemas construidos a partir de *frameworks* suelen presentar unos niveles de herencia altos ya que las clases son construidas a partir de una jerarquía existente. En lenguajes como Java o Smalltalk, las clases siempre heredan de la clase *Object*, lo que añade uno a DIT.

Los problemas con esta métrica se deben a las diferentes características de la herencia, ya que DIT no queda claramente definida y no puede verse como una medida de reuso. Puede fácilmente imaginarse clases con gran profundidad en la jerarquía reusando menos métodos que en una clase poco profunda pero muy ancha.

Número de hijos.
(Number of Children –NOC-) [Chidamber y Kemerer, 1994]

Definición NOC es el número de subclases subordinadas a una clase en la jerarquía, es decir, el número de subclases que pertenecen a una clase.

Propósito Según Chidamber y Kemerer, NOC es un indicador del nivel de reuso, la posibilidad de haber creado abstracciones erróneas y es un indicador del nivel de test requerido.

Guías y comentarios Aunque un mayor número de hijos representa una mayor reutilización de código, presenta los siguientes inconvenientes:

- Mayor probabilidad de usar incorrectamente la herencia creando abstracciones erróneas.
- Mayor dificultad para modificar una clase ya que afecta a todos los hijos que tienen dependencia con la clase base.
- Se requiere mayor número de recursos para testear.

Es un potencial indicador de la influencia que una clase puede tener sobre el diseño del sistema. Si el diseño tiene una alta dependencia en la reusabilidad a través de herencia, puede ser mejor dividir la funcionalidad en varias clases.

Índice de especialización por clase.
(Specialisation Index per Class –SIX-) [Lorenz y Kidd, 1994]

Definición El índice de especialización muestra en que medida las subclases redefinen el comportamiento de sus superclases.

$$SIX = \frac{N^o \text{ de metodos redefinidos} \cdot \text{Anidamiento en la jerarquía}}{N^o \text{ total de metodos}}$$

Esta fórmula pondera más las redefiniciones que ocurren en niveles más profundos del árbol de herencia ya que cuanto más especializada es una clase, es menos probable que su comportamiento sea reemplazado.
Cuando se utilizan *frameworks*, algunos métodos deben ser redefinidos, estos métodos no deben ser tenidos en cuenta al calcular esta métrica.

Propósito SIX se propone como medida de la calidad en la herencia.

Guías y comentarios SIX puede indicar cuando hay demasiados métodos redefinidos, tal que las abstracciones pudieron no ser apropiadas y se necesite reemplazar demasiado comportamiento. Generalmente, una subclase debería extender el comportamiento de la superclase con nuevos métodos, más que reemplazar o borrar comportamientos a través de redefiniciones.

Lorenz y Kidd sugieren un valor del 15% para ayudar a identificar superclases que no tienen mucho en común con sus subclases. Cuanto más profundizamos en la jerarquía, la subclase debe de ser más especializada.

4.4.3. Métricas a nivel de clases

Este conjunto de métricas identifica características dentro de las clases, destacando diferentes aspectos de sus abstracciones y ayudando a descubrir clases que podrían necesitar ser rediseñadas.

Respuesta de una clase
(Response For a Class –RFC-) [Chidamber y Kemerer, 1994]

Definición RFC es el cardinal del conjunto de todos los métodos que pueden ser invocados en respuesta a un mensaje a un objeto de la clase o por alguno método en la clase. Esto incluye todos los métodos accesibles dentro de la jerarquía de la clase. En otras palabras, cuenta las ocurrencias de llamadas a otras clases desde una clase particular.

$RFC = |RS|$ donde RS es el conjunto respuesta para la clase.

El conjunto respuesta para la clase puede ser expresado de la siguiente manera:

$RS = \{M\} \bigcup_i \{R_i\}$,

donde $\{R_i\}$ es el conjunto de métodos llamados por el método *i*;
y $\{M\}$ es el conjunto de todos los métodos en la clase.

Propósito Para Chidamber y Kemerer, RFC es una medida de la complejidad de

una clase a través del número de métodos y de la comunicación con otras clases, ya que incluye todos los métodos llamados desde fuera de la clase.

Guías y comentarios RFC es un indicador de los recursos necesarios para testeo y depuración. Cuanto mayor es RFC, más complejidad tiene el sistema ya que se puede invocar un mayor número de métodos como repuesta a un mensaje.
[Harrison et al. 1996] comentan como la definición de esta métrica es ambigua, forzando al usuario a interpretarla.

Métodos ponderados por clase.
(Weighted Methods per Class -WMC-) [Chidamber y Kemerer, 1994]

Definición WMC mide la complejidad de una clase. Si todos los métodos son considerados igualmente complejos, entonces WMS es simplemente el número de métodos definidos en una clase.

$$WMC = \sum_{i=1}^{n} c_i ,$$

donde una clase C_i tiene los métodos $M_1, \ldots M_n$, con su complejidad respectiva, $c_1, \ldots, c_n$.

Propósito Chidamber y Kemerer sugieren que WMC es una medida de la complejidad de una clase.

Guías y comentarios Clases con un gran número de métodos requieren más tiempo y esfuerzo para desarrollarlas y mantenerlas, ya que influirán en las subclases heredando todos sus métodos. Además, estas clases tienden a ser específicas de la aplicación, limitando su posibilidad de reuso.
Lorenz y Kidd sugieren un umbral de 40 o 20, dependiendo si las clases son o no de interface de usuario respectivamente.
En WMC se aprecian los siguientes problemas [Harrison et al. 1996]:
1. WMC presuntamente mide la complejidad, pero no se da ninguna definición de complejidad.
2. WMC no puede verse como un indicador del esfuerzo necesario para desarrollar una clase, ya que es fácil imaginar clases con pocos métodos complicados y clases con un gran número de métodos pero muy simples.
Por lo tanto, esta métrica debería de ser considerada simplemente como una medida del tamaño de una clase.

Falta de cohesión en los métodos.
(Lack of Cohesion in Methods –LCOM-). [Chidamber y Kemerer, 1994]

Definición LCOM establece en qué medida los métodos hacen referencia a atributos.
Considérese una clase C1 con n métodos $M_1, M_2, \ldots, M_n$. Sea $\{I_j\}$ =

el conjunto de variables instancias por el método Mi.
Hay *n* conjuntos tales que $\{I_1\},... \{I_n\}$; Sea $P = \{(I_i, I_j) \mid I_i \cap I_j = \varnothing\}$, y $Q = \{(I_i, I_j) \mid I_i \cap I_j \_ \varnothing\}$. Si todos los conjuntos *n* $\{I_1\},... \{I_n\}$ son $\varnothing$, entonces $P = \varnothing$.

$$LCOM = \begin{cases} |P| - |Q| & si\ |P| > |Q| \\ 0 & \text{caso contrario} \end{cases}$$

Propósito LCOM es una medida de la cohesión de una clase midiendo el número de atributos comunes usados por diferentes métodos, indicando la calidad de la abstracción hecha en la clase.

Guías y comentarios Un valor alto de LCOM implica falta de cohesión, es decir, escasa similitud de los métodos. Esto puede indicar que la clase está compuesta de elementos no relacionados, incrementando la complejidad y la probabilidad de errores durante el desarrollo.
Es deseable una alta cohesión en los métodos dentro de una clase ya que ésta no puede ser dividida fomentando la encapsulación.

[Henderson-Sellers, 94] destaca dos problemas con esta métrica:
1. Dos clases pueden tener LCOM=0, mientras una tiene más variables comunes que la otra.
2. No se dan guías para la interpretación de esta métrica.

Por lo tanto, Henderson-Sellers ha sugerido una nueva medida para LCOM, LCOM*:
Considérese un conjunto de métodos $\{M_i\}$ $(i=1,\ldots,m)$ accediendo a un conjunto de atributos $\{A_j\}$ $(j=1,\ldots,a)$ y el número de métodos que acceden a cada atributo como $\mu(A_j)$. Entonces se define *LCOM** como:

$$LCOM^* = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right) - m}{1-m}$$

Esta métrica sólo puede calcularse cuando $m>1$.
Cuando todos los métodos acceden a todos los atributos, entonces $\Sigma\mu(A_j) = ma$, y por lo tanto $LCOM^* = 0$. Esto indica una cohesión perfecta. Valores cercanos a 0 indican que la mayoría de los métodos accede a la mayoría de las instancias. Por el contrario, cuando cada método accede solo a un atributo, entonces $\Sigma\,\mu(A_j) = a$ y por lo tanto $LCOM^* = 1$, lo cual indica una falta de cohesión.

4.4.4 Métricas a nivel de métodos

Métodos y atributos se encuentran en el nivel de granularidad más bajo, por lo que las características son bien conocidas por los métodos tradicionales. A este nivel es donde se suelen aplicar las métricas tradicionales como *Líneas de Código*, *cyclomatic complexity* [McCabe, 1976], etc. Métodos y atributos son equivalentes a funciones y variables en la programación funcional. Aunque en los sistemas OO estas métricas no son tan importantes ya que los métodos tienden a distribuir la complejidad entre las llamadas a objetos más que dentro de los propios métodos.

Líneas de código
(Lines of Code per method –LOC-). [Lorenz y Kidd, 1994]

Definición LOC es el número de líneas activas de código (líneas ejecutables) en un método.

Propósito El tamaño de un método es usado para evaluar la comprensibilidad, reusabilidad y mantenibilidad del código.

Guías y comentarios Los umbrales para evaluar el tamaño dependen del lenguaje de programación y de la complejidad de los métodos. En sistemas OO el número de líneas de código de los métodos debería ser bajo. Lorenz y Kidd sugieren un umbral de 24 LOC para métodos en C++ y de 8 en Smalltalk para descubrir que métodos son candidatos a ser divididos en varios.
No es una métrica recomendada para sistemas OO, pero es fácil de recoger y utilizar. Es fácil ver que LOC para la misma funcionalidad, varían con el lenguaje de programación y el estándar de codificación utilizado.

Número de mensajes enviados
(Number of Messages Send –NOM-) [Lorenz y Kidd, 1994]

Definición NOM mide el número de mensajes enviados en un método, segregados por el tipo de mensaje. Los tipos incluyen:

- Unarios, mensajes sin argumentos.
- Binarios, mensajes con un argumento que pertenecen a tipos especiales (por ejemplo concatenación y funciones matemáticas).
- Clave, mensajes con uno o más argumentos.

Propósito NOM cuantifica el tamaño del método de una manera relativamente no sesgada.

Guías y comentarios Lorenz y Kidd sugieren un umbral de 9. Un valor alto puede indicar un estilo funcional y/o una colocación pobre de responsabilidades.
Lenguajes como C++ pueden llamar a subsistemas no OO y estas llamadas, no deberían ser contadas en el número de mensajes enviados.

4.5. CONCLUSIÓN

Se han descrito un conjunto de métricas para diseños orientados a objeto, dando guías y umbrales para su interpretación como un medio para producir sistemas de calidad. Se comentan también algunos de los problemas que podemos encontrarnos al analizar las métricas, como definiciones ambiguas que dan lugar a varias interpretaciones, la falta de concordancia para el propósito que fueron concebidas e incluso la falta de validez teórica.

Todo esto hace ver que es necesario más trabajo en el campo de las métricas orientadas a objetos y en su validez tanto teórica como experimental. Así mismo, es necesario tener unas normas más claras para la interpretación de las métricas basándose en el sentido común y en la experiencia.

4.6. REFERENCIAS

[Abreu y Melo, 1996] Brito e Abreu F., Melo W. "Evaluating the impact of Object-Oriented Design on Software Quality". *Proceedings of 3rd International Software Metrics Symp*., Berlin, 1996.

[Archer y Stinson, 1995] Archer C., Stinson M. *"Object-Oriented Software Measures"*. Technical Report CMU/SEI-95-TR-002, Abril 1995.

[Basili y Rombach, 1988] Basili V. R., Rombach H. D. "The TAME Project: Towards Improvement-Oriented Software Environments", *IEEE Transactions on Software Engineering*, Vol. 14, No. 6, Junio 1988.

[Basili et al. 1995] Basili V. R., Briy L., Melo W. *"A Validation of OO Design Metrics as Quality Indicators"*, Technical Report CS-TR-3443, 1995.

[Cartwright y Shepperd, 1996] Cartwright, M., Shepperd, M. *"An Empirical Investigation of Object Oriented Software in Industry"*, Dept. of Computing, Talbot Campus, Bournemouth University Technical Report TR 96/01, 1996.

[Chidamber y Kemerer, 1994] Chidamber S. R., Kemerer C. F. "A metric suite forsz object oriented design", *IEEE Transactions on Software Engineering*, pp. 467–493, 1994.

[Chidamber et al. 1998] Chidamber, S. R., Darcy D. P., Kemerer C.F. "Guides of object oriented software metrics: an exploratory analysis". *IEEE Transactions on Software Engineering*, 24(8), pp. 629-639, Agosto 1998.

[Daly et al. 1992] Daly J., Brooks A., Miller J., Roper M., Wood M. "Evaluating inheritance Depth on the Maintainability of Object-Oriented *Software"*. *Empirical Software Engineering* 1(2), Febrero 1996.

[Fenton y Pfleeger, 1997] Fenton N. E., Pfleeger S. L. *Software Metrics. A rigorous and Practical Approach. 2nd Edition*. ITP, International Thomson Computer Press, 1997.

[Fenton, 1994] Fenton N. E. "Software measurement: a necessary scientific basis," *IEEE Transactions on Software Engineering*, Vol. 20, no. 3, pp. 199–206, 1994.

[Harrison et al. 1997] Harrison R., Counsell S., Nithi, R. "An Overview of Object-Oriented Design Metrics". *Proceedings of 8th IEEE International Workshop*

on Software Technology and Engineering Practice - (STEP'97), London, UK, Julio 1997.

[Harrison et al. 1998] Harrison R., Counsell S., Nithi R. “An evaluation of the MOOD Set of Object-Oriented Software Metrics”. *IEEE Transactions on Software Engineering*, Vol. 24, No. 6, Junio 1998.

[Harrison et al. 1999] Harrison R., Counsell S., Nithi R. “Experimental Assessment of the effect of inheritance on the maintainability of Object-Oriented systems”. *Proceedings of Empirical Assessment in Software Engineering (EASE'99), Keele, UK, 1999*.

[Henderson-Sellers, 1996] Henderson-Sellers, B. *Object-Oriented Metrics Measures of Complexity*. Upper Saddle River, NJ: Prentice Hall, 1996.

[Kitchenham et al. 1986] Kitchenham B. A., Walker J. D., y Domville I. “*Test specification and quality management - design of a QMS sub-systemfor quality requirements specification”*, Project Deliverable A27, Alvey Project SE/031, Noviembre 1986.

[Kitchenham et al. 1995] Kitchenham B. A., Pleeger S. L., y Fenton N. “Towards a Framework for Software Measurement Validation”. *IEEE Transactions on Software Engineering*, Vol. 21, No. 12, pp 929-944, Diciembre 1995.

[Lorenz y Kidd, 1994] Lorenz M., Kidd J. *Object Oriented Metrics*. Englewood, NJ: Prentice Hall, 1994.

[McCabe, 1976] McCabe, T. J. "A Complexity Measure", *IEEE Transactions on Software Engineering*, 2(4), 308-320, 1976.

[Schneidewind, 1992] Schneidewind N. F. “Methodology for validating software metrics,” *IEEE Transactions on Software Engineering*, Vol. 18(5), pp. 410–422, 1992.