Informe Técnico – Technical Report
DPTOIA-IT-2001-002
Noviembre, 2001

MÉTRICAS ORIENTADAS A OBJETOS

Pedro Jesús Vázquez Escudero
María N. Moreno García
Francisco J. García Peñalvo

Departamento de Informática y Automática
Universidad de Salamanca

Revisado por:

Dr. Juan Manuel Corchado Rodríguez

Departamento de Informática y Automática

Universidad de Salamanca

corchado@usal.es

Dña. Mª Esperanza Manso Martínez

Departamento de Informática

Universidad de Valladolid

manso@infor.uva.es

Aprobado en el Consejo de Departamento de 29 de octubre de 2001

Información de los autores:

Pedro Jesús Vázquez Escudero

Estudiante de doctorado del Departamento de Informática y Automática

Universidad de Salamanca

pedro.vazquez.e@terra.es

Dra. María N. Moreno García

Área de Lenguajes y Sistemas Informáticos

Departamento de Informática y Automática

Facultad de Ciencias - Universidad de Salamanca

Plaza de la Merced S/N – 37008 – Salamanca

mmg@usal.es

Dr. Francisco José García Peñalvo

Área de Ciencia de la Computación e Inteligencia Artificial

Departamento de Informática y Automática

Facultad de Ciencias - Universidad de Salamanca

Plaza de la Merced S/N – 37008 – Salamanca

fgarcia@usal.es

Este trabajo ha sido parcialmente financiado por el proyecto “DOLMEN” de la CICYT, TIC2000-1673-C06-05, Ministerio de Ciencia y Tecnología.

Resumen

El objetivo del presente trabajo es ofrecer una panorámica del estado actual de la medición de productos software y en particular en orientación a objetos, enfoque que ha alcanzado en la actualidad altas cotas de popularidad debido a los beneficios que promete: rápido desarrollo, reutilización, gestión de la complejidad, etc., características que inciden directamente en la mejora de la calidad de los productos software. Se propone un marco de trabajo basado en el lenguaje de marcas XML para facilitar la recolección y análisis de métricas.

Abstract

This work examines the state of art in software products measuring, more specifically in the Object Oriented approach, which has become high popular because of the benefits it promises: rapid development, reuse, complexity management, etc., all of this, characteristics that increase directly the quality of the software products. Also is proposed a metrics framework based on markup language XML which facilitates the activities of collecting and analysing software metrics.

Tabla de Contenidos

1. Introducción

Uno de los objetivos fundamentales del uso de métricas en sistemas software es evaluar los mismos con el fin de controlar y mejorar su calidad

Por otra parte, cuestiones como el tiempo estimado para el desarrollo de un proyecto, los costes de mantenimiento una vez que el producto está acabado y funcionando, etc., no pueden ser fácilmente resueltas sin la ayuda de la medición.

Parte del concepto de la Ingeniería del Software es la idea de que el producto debería ser controlable. Según De Marco, "no se puede controlar lo que no se puede medir" [De Marco, 1982]. Fenton y Pfleeger añaden: "No se puede predecir lo que no se puede medir" [Fenton y Pfleeger, 1997].

Proyectos futuros podrán beneficiarse de la medición efectuada en proyectos pasados de similares características, o comparando componentes similares dentro del mismo proyecto, permitiendo realizar predicciones más ajustadas.

La mayoría de las estrategias de medida tienen por objetivo evaluar las diferentes características de calidad del software, tales como la fiabilidad, la facilidad de uso, la facilidad de mantenimiento, la robustez, etc.

Haciendo un repaso histórico los inicios de la medición pueden datarse en los años 60, en que sólo se recogían medidas muy primitivas tales como las líneas de código (LOC) de los módulos y el tiempo dedicado en cada etapa del ciclo de vida.

Modelos posteriores (COCOMO) en los años 70 intentaron establecer una relación entre el tamaño de los programas y el esfuerzo necesario para realizarlos y mantenerlos [Boehm, 1981].

McCall y Boehm, entre otros, han definido modelos de medidas orientados hacia la evaluación de la calidad de los productos software [MacCall et al., 1977], [Boehm et al., 1978].

En los años 80 se profundizó en métricas de complejidad y de diseño modular y estructurado, modelos de estimación del coste y del tiempo de desarrollo de los proyectos software. Aparecieron las normas ISO 9000 y los modelos CMM [Paulk et al., 1993] y SPICE [SPICE, 1999] centrados en los procesos software.

En los años 90 se busca sentar unas bases bien establecidas frente a la confusión reinante en los ochenta. Los intentos se centran en clasificar, estudiar y validar las métricas del software [Fenton, 1991]. Aparece una revisión del modelo COCOMO (COCOMO II) [USC-CSE, 1997] para reflejar los grandes cambios en las técnicas de desarrollo producidos en más de una década y media: declive de los procesos "*batch*" ejecutados por la noche en "*mainframes*" frente al auge de los procesos en tiempo real, énfasis en la reutilización del software y construcción de nuevos sistemas usando componentes ya existentes y la dedicación de un mayor esfuerzo al diseño y a la gestión de los procesos de desarrollo software.

Este trabajo examina el estado del arte de la medición en orientación a objetos. Después de esta breve introducción histórica, se repasan las propiedades que, idealmente, debe poseer una métrica para que sea de utilidad.

A continuación se hace una breve presentación del enfoque orientado a objetos, que por sus características singulares, requerirán de un conjunto adicional de métricas que completen o sustituyan a las tradicionales.

Conocidas las características principales del software orientado a objetos, se presenta una taxonomía de métricas OO en tres dimensiones: una la propiedad que se quiere medir (cohesión, acoplamiento, herencia, etc.), otra el nivel a que se quiere medir (variable, método, clase,

sistema...) y otra la etapa del ciclo de vida en que se mide (análisis, diseño, implementación, prueba...).

Se hace un estudio en mayor profundidad de las métricas relacionadas con la prueba y la calidad del software.

Por último se propone un marco de trabajo para métricas utilizando el lenguaje de marcas XML para facilitar la recolección y análisis de las métricas.

2. Propiedades de las métricas

2.1. Factores que determinan la calidad del software

Los factores de calidad son normalmente atributos de un alto nivel de abstracción como "fiabilidad", "facilidad de uso", "facilidad de mantenimiento", etc. Los modelos de McCall [McCall et al., 1977] y Boehm [Boehm et al., 1978] centrados en las características del producto software descomponen estos atributos en otros mensurables directamente.

El modelo de calidad del software de McCall define tres aspectos o características importantes de un producto: características operacionales, de modificación y de transición, descomponiendo cada una en *factores de calidad* de alto nivel que determinan la calidad en cada una de ellas. Estos son a su vez descompuestos en *criterios de calidad*. Por último, estos son asociados a un conjunto de atributos directamente mensurables denominados *métricas de calidad.* Esta permite definir unos atributos en términos de otros más fáciles de medir. Aquí se presenta el primer nivel de descomposición:

- **Características operacionales**
  - FIABILIDAD
    - CORRECCIÓN: un programa es correcto cuando efectúa sin errores las tareas que le son encomendadas.
    - ROBUSTEZ: capacidad de un programa para responder con elegancia a situaciones no previstas.
  - EFICIENCIA: ofrecer un máximo de prestaciones requiriendo un mínimo de recursos. En general, compatibilidad es antagónico con eficiencia.
  - FACILIDAD DE USO: esfuerzo que se necesita para aprender un programa, para trabajar con él, preparar la entrada e interpretar su salida.
  - INTEGRIDAD: el grado en que puede controlarse el acceso al software o a los datos por personal no autorizado.

- **Características de modificación**: capacidad de soportar cambios.
  - FACILIDAD DE MANTENIMIENTO: el esfuerzo requerido para modificar un programa operativo.
  - EXTENSIBILIDAD: adición de nuevas funcionalidades de manera sencilla.
  - FACILIDAD DE PRUEBA: el esfuerzo requerido para probar un programa de forma que asegure que realiza su función requerida.

- **Características de transición**: adaptabilidad a nuevos entornos.
  - PORTABILIDAD: capacidad de ejecutarse en diferentes plataformas sin cambios (abstracción).
  - REUSABILIDAD: propiedad de los componentes software por la que pueden aplicarse a tareas distintas de aquellas para las que se construyeron inicialmente.
  - INTEROPERABILIDAD: los sistemas deben ser capaces de interoperar con otros componentes software.

Estas características miden la forma en que el producto se relaciona con su entorno. Por ejemplo, la fiabilidad se define como la operación de un producto libre de fallos; depende del entorno de máquina y del usuario. Para ser medidas, estas características se apoyan en atributos internos del producto:

- Atributos internos: aquellos que se pueden medir en términos del propio producto independientemente de su comportamiento.
- Atributos externos: son aquellos que sólo se pueden medir con respecto a cómo el producto se relaciona con su entorno.

La medición tanto de los atributos internos como externos se consideran el fundamento para mejorar la calidad de los productos software.

2.2. Medición y propiedades de las métricas

Medición es “el proceso por el cual se asignan números o símbolos a atributos (dimensiones) de entidades del mundo real de tal manera que describa dichos atributos de una forma significativa y de acuerdo a unas reglas claramente definidas” [Fenton y Pfleeger, 1997]. Una entidad puede ser una persona, un programa desarrollado o el proceso de desarrollo. Un atributo (dimensión) de una entidad puede ser el peso de una persona o la longitud del proceso de desarrollo. La finalidad del uso de las métricas es evaluar sistemas para conseguir alta calidad, robustez y facilidad de mantenimiento.

Las métricas se utilizan, entre otras cosas, para evaluar en qué grado el software posee las distintas características que definen la calidad de un producto software.

Una métrica puede ser aplicable a una o más etapas del ciclo de vida de desarrollo del software: especificación de requisitos, análisis, diseño, implementación, depuración, prueba e integración, mantenimiento y documentación.

Las medidas no pueden ser aplicadas alegremente a cualquier atributo. Por ejemplo, las líneas de código, siendo una medida válida del tamaño, han sido usadas para medir la complejidad de programas de forma incorrecta.

Una métrica debería, idealmente, poseer las siguientes propiedades [Schulmeyer y MacKenzie, 2000]:

- Simplicidad: la definición y uso de la métrica ha de ser simple.
- Objetividad: diferentes personas han de darle valores idénticos. Esto le da consistencia y evita interpretaciones individuales.
- Facilidad de recolección: el coste y esfuerzo para obtener la medida ha de ser razonable.
- Robustez: la métrica ha de ser insensible a cambios irrelevantes. Esto permite realizar útiles comparaciones.

- Validez: la métrica ha de medir lo que se supone que mide. Esto da fiabilidad a la medida.

Existen dos tipos de métricas:

- Métricas directas: aquellas que para medir un atributo de una entidad sólo precisan de dicho atributo.
- Métricas indirectas: aquellas que no miden un atributo directamente, sino que para medir un atributo de una entidad precisan medir más de un atributo de dicha entidad, es decir, combinan varias métricas directas.

Dado que las dimensiones de mayor interés (por ejemplo, calidad y fiabilidad) son a menudo externas a la entidad que está siendo medida, se suele necesitar de medidas indirectas para alcanzar los resultados deseados. Por ejemplo, el número de defectos conocidos se cuenta para valorar la calidad de un programa.

2.3. Validación de métricas

Informalmente se dice que una medida es válida si caracteriza fielmente el atributo que pretende medir.

Validar una medida software es el proceso de asegurar que la medida es una caracterización numérica apropiada del atributo que pretende medir, demostrando que se cumple la *condición de representación*. En [Fenton y Pfleeger, 1997] se define la condición de representación de la forma siguiente: "La relación de medición *M* debe hacer corresponder entidades a números, así como 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 solo si *M(A) > M(B)*". Es decir, hay que asegurarse de que las medidas que se usan reflejan el comportamiento de las entidades del mundo real.

Una medida debe ser vista en función del contexto en que será usada. La validación debe tener en cuenta el propósito de la medición. Fenton y Pfleeger [Fenton y Pfleeger, 1997] dan dos definiciones de validación:

- Validación en sentido reducido: es el proceso riguroso de asegurar que la medida representa adecuadamente el atributo software que se quiere medir, esto es, verificar que la medida es razonable teóricamente (internamente válida).
- Validación en sentido amplio: es la utilización de una medida internamente válida como un componente de un *sistema de predicción* válido.

Un modelo que mida de forma precisa los atributos es necesario pero no suficiente para construir un sistema de predicción preciso. La medición se utiliza no sólo en la valoración de un atributo actual sino en la predicción de un atributo futuro.

[Kitchenham et al., 1995] da las siguientes propiedades que las métricas deben poseer para ser válidas:

- Métricas directas:
  - Para poder medir un atributo debe permitir que distintas entidades sean distinguibles entre sí (problema de la unicidad). Las afirmaciones hechas que impliquen escalas numéricas tienen sentido sólo si son únicas, esto es, si se mantiene la afirmación cuando la escala es reemplazada por otra escala igualmente admisible.

Por ejemplo, podemos decir que una piedra pesa el doble que otra, pero no que una piedra está el doble de caliente que otra. El peso es una escala de ratio[1] y por tanto se mantiene la afirmación independientemente de que lo expresemos en gramos o en kilos. Las escalas de temperatura en grados Fahrenheit y Celsius son escalas de intervalo y tienen distinto punto de origen. Por ello, el ratio de temperatura medida en una escala es diferente que el medido en la otra.

- La métrica debe cumplir la condición de representación: ha de permitir el establecimiento de una correspondencia entre el sistema observado y un sistema matemático determinado.
- Cada unidad que contribuye en una métrica válida debe ser equivalente. Por ejemplo, en el caso de las escalas de temperatura (escalas de intervalo), un grado contribuye de igual forma para pasar de 20° a 21° que de 30° a 31°.
- Diferentes entidades pueden tener el mismo valor.

- Métricas indirectas:
  - Se debe basar en un modelo explícitamente definido de relaciones entre atributos.
  - El modelo de medición tiene que ser dimensionalmente consistente.
  - La métrica no debe mostrar ninguna discontinuidad inesperada.
  - Las unidades y escalas se han de usar correctamente.

2.4. Escalas

El problema de la unicidad lleva a tener que elegir qué representación es la mejor para el atributo que se va a medir. La teoría de escalas ayuda en esta tarea. La escala a la que pertenece una métrica limita las operaciones y asertos significativos que pueden hacerse sobre ella, así como el tratamiento estadístico que se puede aplicar. Existen 5 tipos principales de escalas:

- Nominal: sólo existen clases diferentes. No hay noción de orden entre las clases. Una medida aceptable es un número o símbolo. No hay noción de magnitud (clasificación).

  Un ejemplo de escala nominal es la clasificación de los programas de software según el lenguaje de programación en que estén escritos: C, Pascal, Java...

- Ordinal: clases ordenadas respecto al atributo. Cualquier mapeo que preserve el orden es aceptable. Los números sólo indican el lugar que ocupan, no se permiten operaciones aritméticas.

  Un ejemplo de escala ordinal es la expresión de forma cuantitativa de la complejidad de un conjunto de programas, utilizando las siguientes categorías en orden de menor a mayor complejidad: trivial, sencilla, moderada, compleja y muy compleja. Aquí no se puede afirmar que un programa es “el doble de complejo” que otro.

- Intervalo: preserva el orden y diferencias. Admite sumas y restas pero no la multiplicación ni división (ratios).

  Un ejemplo típico es la medición de temperatura en grados Celsius, en que el intervalo de un grado a otro es el mismo. Si el mismo día hay 20° en Londres y 30° en Madrid, y se incrementa en un grado en ambos lugares, pasará a haber 21° en Londres y 31° en

[1] Ver los distintos tipos de escalas en el apartado siguiente.

Madrid. Sin embargo, no se puede decir que haya un 50% más de calor en Madrid que en Londres.

- Ratio: preserva el orden, tamaño de los intervalos entre entidades y ratios entre entidades. Existe elemento cero (ausencia el atributo). Admite cualquier operación aritmética.

  Por ejemplo, la longitud de un programa de software. Se puede hablar de un programa el doble de largo que otro, y un programa vacío representaría el elemento cero.

- Absoluta: es la más restrictiva de todas. Sólo admite la transformación identidad entre dos medidas. La medida se obtiene contando el número de elementos del conjunto de entidad. El atributo toma la forma "número de ocurrencias de *x* en la entidad". Por ejemplo, el número de fallos observados durante las pruebas de integración sólo puede medirse contando el número de fallos observados. La unicidad de la medida es una diferencia importante entre la escala de ratio y la absoluta.

3. El enfoque orientado a objetos

3.1. Introducción

El paradigma de la orientación a objetos posee características específicas que lo diferencia de enfoques más tradicionales como la programación estructurada y esto implica que métricas aplicadas en programación estructurada pueden no ser válidas en orientación a objetos. Surgen así nuevas métricas que pretenden describir de manera adecuada estas nuevas características del software, tales como encapsulamiento, ocultamiento de la información, abstracción y polimorfismo.

El paradigma de la orientación a objetos se ha popularizado en los últimos años debido a los principales beneficios que comporta:

- Desarrollo más rápido.
- Reutilización del trabajo anterior.
- Arquitectura modular.
- Gestión de la complejidad en proyectos de gran tamaño.

3.2. Características del software orientado a objetos

Siguiendo a [Rumbaugh et al., 1991] y a [Henderson-Sellers, 1992] las características principales del enfoque orientado a objetos son:

1. IDENTIDAD. Los datos se organizan en entidades discretas llamadas objetos.
2. OBJETOS. Cada objeto tiene su propia identidad que lo distingue de otro aunque ambos posean los mismos valores en todos sus atributos.
3. CLASIFICACIÓN. Objetos con los mismos atributos y comportamiento se agrupan en clases.
4. CLASE. Una clase es una abstracción que describe las propiedades (atributos y comportamiento) relevantes de un conjunto de objetos para una aplicación dada.
5. POLIMORFISMO. Permite que una misma operación pueda llevarse a cabo de forma diferente en clases diferentes. Una operación es una acción o transformación que realiza

un objeto o que se realiza sobre él. La implementación específica de una operación determinada en una clase se denomina método.

6. HERENCIA. La herencia se basa en una relación jerárquica entre varias clases de tal forma que se pueden compartir atributos y operaciones en la subclase de la superclase de la que hereda ciertas propiedades y añade propiedades particulares.
7. La REUTILIZACIÓN de código y de diseño, que puede ser vista como una ventaja de la orientación a objetos. La herencia es la base de la reutilización.
8. ABSTRACCIÓN. Una clase abstracta es una clase que no tiene instancias directas pero cuyas clases descendientes (clases concretas) sí pueden tener instancias directas. Una clase abstracta agrupa características comunes a varias clases. Está relacionada con la herencia.
9. ENCAPSULAMIENTO. Inclusión dentro de un mismo objeto de los datos (información que describe su estado) y los métodos (mecanismos de transformación o de acceso al estado de dichos objetos). El uso de un objeto se realiza mediante una interfaz bien definida, y el resto está encapsulado dentro de ese objeto.
10. OCULTAMIENTO DE LA INFORMACIÓN. Consiste en no mostrar los detalles de la implementación de las rutinas a los elementos del programa exteriores al objeto. Esto permite realizar cambios en la implementación (que no impliquen cambios en la funcionalidad) sin afectar a las clases cliente.
11. ESPECIALIZACIÓN. La obtención de subclases por refinamiento de una superclase.

Otras características no sólo aplicables al enfoque orientado a objetos son:

1. COHESIÓN. Un objeto tiene un alto nivel de cohesión si ejecuta una tarea sencilla y requiere poca interacción con métodos que se ejecutan en otros objetos. Es una extensión del concepto de ocultamiento de la información. Un objeto coherente debe hacer (idealmente) una sola cosa.
2. ACOPLAMIENTO. Es una medida de la interconexión entre objetos. Indica en qué medida una clase utiliza atributos y/o métodos de otra.

En la figura 1 se representan las propiedades de forma jerárquica [Schach, 1996] donde los módulos representarían los subprogramas clásicos. Las propiedades que deben tener los módulos para ser considerados auténticos objetos: no sólo deben construirse usando encapsulamiento, abstracción de datos y ocultamiento de la información, sino que deben poseer alta cohesión (funcionalidad bien delimitada) y bajo acoplamiento (limitada interdependencia funcional).

La presencia de estas características en la proporción adecuada en un producto software determina de forma bastante significativa la calidad del mismo, que se establece mediante la evaluación de las propiedades indicadas en el apartado 2. La medición ayuda a evaluar la presencia de estas características tanto en los productos software como en los procesos software, bien entendido que la calidad del proceso no implica necesariamente que el producto obtenido vaya a ser de calidad.

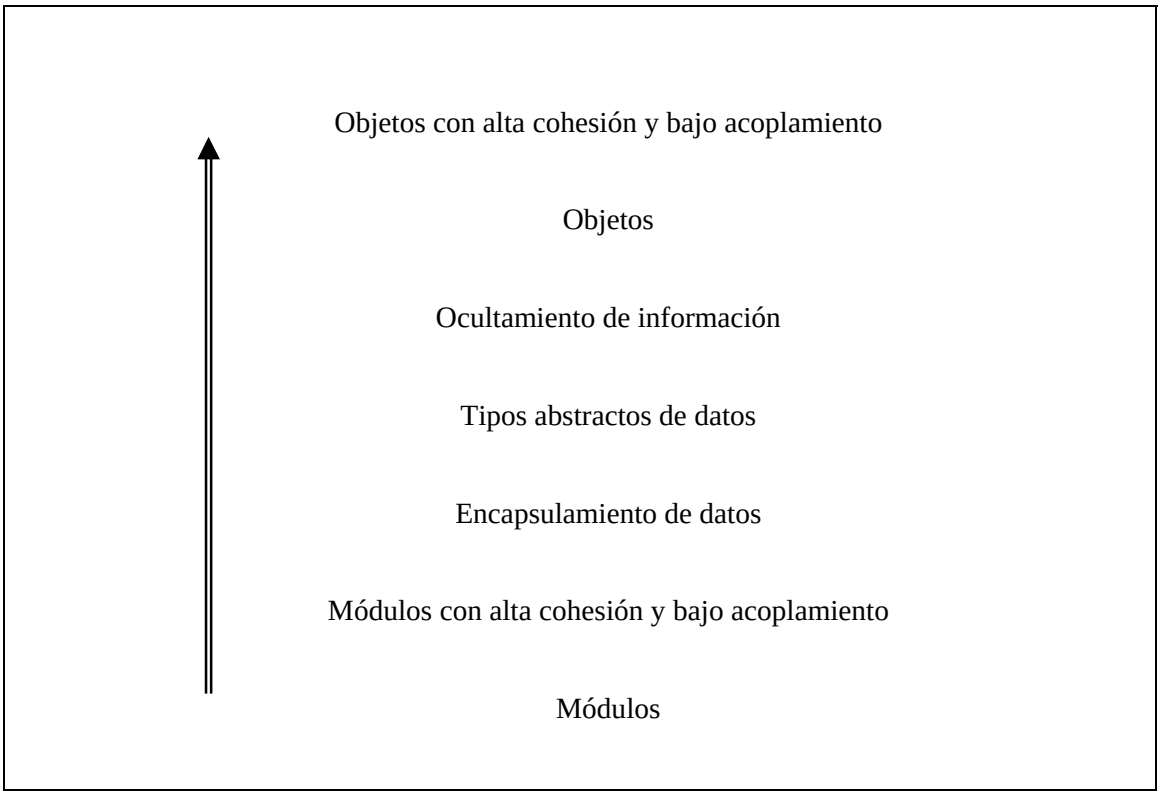

Figura 1. Propiedades de los objetos [Schach 1996].

4. Taxonomía de métricas orientadas a objetos

La tendencia de la Ingeniería del Software es introducir la medición en todas las etapas del ciclo de vida de un proyecto (sistema) software independientemente del modelo utilizado: cascada, espiral, técnicas de cuarta generación, etc.

Así, las métricas las podemos ver desde una perspectiva tridimensional, con las siguientes dimensiones:

- Característica o atributo del software a medir (por ejemplo, cohesión, complejidad, reutilización...).
- Etapa del ciclo de vida en la que se mide (análisis, diseño, implementación, pruebas...).
- Nivel de granularidad en el que se mide (por ejemplo, nivel de sistema, de programa, de clase, de método o de variable).

Las métricas no pueden ser aplicadas indiscriminadamente a cualquiera de los atributos del software que se quieren medir. El caso más típico de una asignación errónea de métricas a atributos es el de líneas de código (LOC), que siendo válida como una medida del tamaño, se usaba equivocadamente para medir la complejidad de los programas.

Por otro lado, no todas las métricas han de ser recogidas durante la fase de implementación del código fuente de los programas, aunque un gran número de ellas se obtenga de ahí. Sería deseable obtenerlas más tempranamente en la fase de diseño. [Chidamber y Kemerer, 1994] proponen un conjunto de 6 métricas para diseño orientado a objetos. Desafortunadamente, las distintas notaciones utilizadas en los documentos de diseño hace difícil la utilización de herramientas estándar de recolección de métricas. La creciente utilización del Lenguaje

Unificado de Modelado (UML) [Booch et al., 1999] puede mitigar este problema. [Genero et al., 2000] proponen un conjunto de métricas para valorar la complejidad de los diagramas de clases modelados con UML, centrándose específicamente en la complejidad de los diagramas de clases surgidos de las diferentes relaciones en UML, tales como asociaciones, agregaciones, etc., por ejemplo, definen AvsC (Atributos vs. Clases).

Por último, algunas métricas son escalables a distintos niveles de granularidad respecto de los que fueron definidos. Por ejemplo, la métrica LOC es utilizable a nivel de método, clase, programa y sistema.

A continuación se presenta una clasificación de métricas teniendo en cuenta estas tres dimensiones, y agrupadas según la característica principal del enfoque orientado a objetos que pretenden medir, indicando para cada una de ellas el nivel de granularidad y la etapa del ciclo de vida en que se puede medir.

Para dar una visión más general se ha pretendido abarcar la mayor cantidad de atributos posible, eligiendo en cada caso un conjunto de métricas que se considera representativo para dicho atributo, sin ser excesivamente exhaustivos en el número de métricas presentadas para evitar llegar a la confusión dada la cantidad de métricas aparecidas en los últimos años. Se indica para cada métrica el autor o autores de la misma, remitiendo al lector a la bibliografía para un estudio en mayor profundidad.

El punto de partida es [Chidamber y Kemerer, 1994] que establecen 6 métricas para medir cinco atributos básicos en el *diseño orientado a objetos*: acoplamiento, complejidad de una clase, reutilización, cohesión, y herencia.

[Briand et al., 1999] realizan un estudio exhaustivo de las métricas sobre *acoplamiento* existentes y proponen un marco de trabajo unificado con los siguientes criterios:

- Tipo de conexión: qué es lo que constituye acoplamiento.
- Posición o punto de impacto: *import / export*.
- Granularidad de la medida: el dominio de la medida y cómo contar las conexiones de acoplamiento.
- Estabilidad del servidor.
- Acoplamiento directo o indirecto.
- Herencia: acoplamiento basado o no en herencia.

[McCabe, 1976] se centra en las *medidas de complejidad* ciclomática de los programas. Presentadas antes de la existencia del enfoque orientado a objetos, han sido redefinidas en la herramienta *McCabe Object-Oriented Metrics Tool* [McCabe et al., 1994] para su uso en sistemas orientados a objetos.

[Abreu y Melo, 1996] estudian las *métricas a nivel de sistema* centrándose en las características básicas de la orientación a objetos: encapsulamiento, herencia, polimorfismo y paso de mensajes.

De [Lorenz y Kidd, 1994] se destacan las *métricas de tamaño*.

Las *métricas de cobertura de pruebas* en contexto de herencia son tratadas en [IPL, 1999].

4.1. Acoplamiento

El acoplamiento se ve como una medida del incremento de la complejidad, reduciendo el encapsulamiento y su posible reutilización; limita, por tanto, la facilidad de comprensión y de mantenimiento del sistema.

4.1.1. Acoplamiento entre objetos (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, sin tener con ella relaciones de herencia. Hay dependencia entre dos clases cuando una de ellas usa métodos o variables de la otra clase. Es consistente con las tradicionales definiciones de acoplamiento: "medida del grado de interdependencia entre módulos".
- Valoración: los autores sugieren que sea un indicador del esfuerzo necesario para el mantenimiento y en las pruebas. Cuanto más independiente es un objeto, más fácil es reutilizarlo en otra aplicación. Al reducir el acoplamiento se reduce la complejidad, se mejora la modularidad y se promueve el encapsulamiento. Una medida del acoplamiento es útil para determinar la complejidad de las pruebas necesarias de distintas partes de un diseño. Cuanto mayor sea el acoplamiento entre objetos más rigurosas han de ser las pruebas.
- Nivel de granularidad: clase, programa, sistema.
- Ciclo de vida: diseño.

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

- Definición: es la proporción entre el número real de acoplamientos no imputables a herencia y el máximo número posible de acoplamientos en el sistema. Es decir, indica la *comunicación entre clases*.
- Fórmula:

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

Donde:

$TC^2 - TC$ es el máximo número de acoplamientos en un sistema con *TC* clases

$$es\_cliente(C_c, C_s) = \begin{cases} 1 & si\ y\ solo\ si\ C_c \Rightarrow C_s \wedge C_c \neq C_s \\ 0 & en\ caso\ contrario \end{cases}$$

La relación cliente-servidor ($C_c \Rightarrow C_s$) significa que *Cc* (la clase *cliente*) contiene al menos una referencia no basada en la herencia a una característica (método o atributo) de la clase *Cs* (clase *proveedora*). El numerador representa el número real de acoplamientos no imputables a la herencia. El denominador representa el máximo número posible de acoplamientos en un sistema con *TC* clases. Las relaciones cliente-servidor pueden tener distintas formas:

- paso de mensajes regular.
- paso de mensajes "forzado".
- iniciación y destrucción de objetos.
- asociaciones semánticas entre clases con una cierta relación (p.e. *1:1*, *1:n* o *n:m*).

- Valoración: COF puede ser una medida indirecta de los atributos con los cuales está relacionado: complejidad, falta de encapsulamiento, carencia de reutilización, facilidad de comprensión y poca facilidad de mantenimiento. Al incrementar el acoplamiento

entre clases, se incrementa la densidad de defectos y disminuye la facilidad de mantenimiento.

- Nivel de granularidad: clase, programa, sistema.
- Ciclo de vida: diseño.
- Relacionado con: complejidad, encapsulamiento, reutilización.

4.2. Cohesión

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

- Definición: número de grupos de métodos locales que no acceden a atributos comunes.
- Valoración: indica la calidad de la abstracción hecha en la clase. Usa el concepto de grado de similitud de métodos. Si no hay atributos comunes, el grado de similitud es cero. Una baja cohesión incrementa la complejidad y por tanto la facilidad de cometer errores durante el proceso de desarrollo. Estas clases podrían probablemente ser divididas en dos o más subclases aumentando la cohesión de las clases resultantes. Es deseable una alta cohesión en los métodos dentro de una clase, ya que ésta no se puede dividir fomentando el encapsulamiento. [Henderson-Sellers, 1996] destaca dos problemas con esta métrica:
  - Dos clases pueden tener LCOM=0, mientras una tiene más variables comunes que la otra.
  - No se dan guías para la interpretación de esta métrica.
- Nivel de granularidad: clase.
- Ciclo de vida: diseño, implementación.

4.3. Complejidad

4.3.1. Respuesta para una clase (Response For a Class-RFC-) [Chidamber y Kemerer, 1994]

- Definición: RFC es el cardinal del conjunto de todos los métodos que se pueden invocar como respuesta a un mensaje a un objeto de la clase o como respuesta a algún método en la clase. Esto incluye a todos los métodos accesibles dentro de la jerarquía de la clase. Es decir, RFC es el número de métodos locales a una clase más el número de métodos llamados por los métodos locales.
- Valoración: para los autores RFC es una medida de la complejidad de una clase a través del número de métodos y de su comunicación con otras, pues incluye los métodos llamados desde fuera de la clase. Cuanto mayor es RFC, más complejidad tiene el sistema, ya que es posible invocar más métodos como respuesta a un mensaje, exigiendo mayor nivel de comprensión, lo que implica mayor tiempo y esfuerzo de prueba y depuración. Se ha señalado que la definición de esta métrica es ambigua y fuerza al usuario a interpretarla.
- Nivel de granularidad: clase, programa, sistema.
- Ciclo de vida: diseño.
- Relacionado con: acoplamiento.

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

- Definición: Dada una clase C_1 con los métodos M_1, ..., M_n y c_1, ..., c_n la complejidad de los métodos, WMC se define como el sumatorio de las complejidades de cada método de una clase. Si todos los métodos son considerados de igual complejidad, entonces c_1=1 y WMC=n (número de métodos).
- Valoración: Describe la complejidad algorítmica de una clase en términos de las complejidades de todos sus métodos. Está ligada a la calidad de la definición de complejidad de un método (c_i). Los autores la simplifican asignando 1 a cada método, convirtiéndose así en un simple contador del número de métodos dentro de una clase. En este caso habría que considerarla como una medida del tamaño de una clase y no de complejidad, ya que una clase puede tener pocos métodos pero muy complejos y otra clase puede tener muchos métodos pero muy simples. Puede servir como un indicador de que una clase determinada necesite una descomposición adicional en varias clases.
- Nivel de granularidad: clase, programa, sistema.
- Ciclo de vida: diseño.

4.3.3. Complejidad ciclomática media (Average v(G) [McCabe, 1976]

- Definición: es la complejidad ciclomática media de los métodos de una clase. La complejidad ciclomática v(G) de un método es el número de caminos independientes a lo largo del método. En métodos estructurados, v(G)=número de nodos de decisión más uno.
- Valoración: se ofrece como una medida de la complejidad estructural. Una medida que se concentra en los métodos más complejos es la complejidad ciclomática máxima (Maximum v(G)). Es un indicador útil de la dificultad de prueba y mantenimiento de un programa. Se sugiere un valor máximo de 10.
- Nivel de granularidad: clase, programa, sistema.
- Ciclo de vida: implementación.

4.4. Encapsulamiento

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

- Definición: es la proporción entre la suma de los grados de invisibilidad de los métodos en todas las clases y el número total de métodos definidos en el sistema. El grado de invisibilidad de un método es el porcentaje sobre el número total de clases desde las cuales un método no es visible. Es decir, MHF es la proporción entre los métodos definidos como protegidos o privados y el número total de métodos.
- Fórmula:

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

Donde:

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

$$es\_visible(M_{mi},C_j) = \begin{cases} 1 & si\ y\ solo\ si\ C_j\ puede\ llamar\ a\ M_{mi} \\ 0 & en\ caso\ contrario \end{cases}$$

TC es el número total de clases.
$M_d(C_i)$ es el número de métodos definidos en la clase C_i (no heredados).
$V(M_{mi})$ es la visibilidad: porcentaje del total de clases desde las cuales el método M_{mi} es visible.

- Valoración: MHF se propone como una medida de encapsulamiento, cantidad relativa de información oculta. Cuando se incrementa MHF, la densidad de defectos y el esfuerzo necesario para corregirlos debería disminuir. No se consideran los métodos heredados.
- Nivel de granularidad: clase.
- Ciclo de vida: implementación.

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

- Definición: es la proporción entre la suma de los grados de invisibilidad de los atributos en todas las clases y el número total de atributos definidos en el sistema. El grado de invisibilidad de un atributo es el porcentaje sobre el número total de clases desde las cuales un atributo no es visible. Es decir, AHF es la proporción entre los atributos definidos como protegidos o privados y el número total de atributos.
- Fórmula:

$$AHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{A_d(C_i)} (1 - V(A_{mi}))}{\sum_{i=1}^{TC} A_d(C_i)}$$

Donde:

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

$$es\_visible(A_{mi},C_j) = \begin{cases} 1 & si\ y\ solo\ si\ C_j\ puede\ llamar\ a\ A_{mi} \\ 0 & en\ caso\ contrario \end{cases}$$

TC es el número total de clases.
$A_d(C_i)$ es el número de atributos definidos en la clase Ci (no heredados).
$V(A_{mi})$ es la visibilidad: porcentaje del total de clases desde las cuales el atributo A_{mi} es visible.

- Valoración: Idealmente el valor de esta métrica debería ser siempre el 100%, intentando ocultar todos los atributos. Las pautas de diseño sugieren que no hay que emplear atributos públicos, ya que se considera que esto viola los principios de encapsulamiento 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.

- Nivel de granularidad: clase.
- Ciclo de vida: implementación.

4.5. Herencia

El uso de la herencia debe entenderse como un compromiso entre la facilidad de reutilización que proporciona y la facilidad de comprensión y de mantenimiento del sistema, que en general, están en relación inversa.

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

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

- Definición: es la proporción entre la suma de todos los métodos heredados en todas las clases y el número total de métodos (localmente definidos más los heredados) en todas las clases.
- Fórmula:

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

Donde:

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

$M_a(C_i)$ es el número de métodos disponibles.
$M_d(C_i)$ es el número de métodos definidos.
$M_i(C_i)$ es el número de métodos heredados.
TC es el número total de clases.

- Valoración: es un indicador del nivel de reutilización. También se propone como ayuda para evaluar la cantidad de recursos necesarios a la hora de probar.
- Nivel de granularidad: clase, programa, sistema.
- Ciclo de vida: diseño, implementación.
- Relacionado con: reutilización.

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

- Definición: es la proporción entre el número de atributos heredados y el número total de atributos.
- Fórmula:

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

Donde:

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

$A_a(C_i)$ es el número de atributos disponibles.
$A_d(C_i)$ es el número de atributos definidos.
$A_i(C_i)$ es el número de atributos heredados.
TC es el número total de clases.

- Valoración: es un indicador de la capacidad de reutilización en un sistema, al igual que MIF.
- Nivel de granularidad: clase, programa, sistema.
- Ciclo de vida: diseño, implementación.
- Relacionado con: reutilización.

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

- Definición: Muestra en qué medida las subclases redefinen el comportamiento de sus superclases:

$$SIX = \frac{N^o\,DeMétodosredefinidos * AnidamientoEnLaJerarquía}{N^o\,TotalDeMétodos}$$

- Valoración: Esta fórmula pondera más las redefiniciones que ocurren en niveles más profundos del árbol de herencia, ya que, cuanto más especializada es una clase, menos probabilidad existe de que su comportamiento sea reemplazado. Cuando se utilizan *frameworks* (clases de bibliotecas especializadas), algunos métodos deben ser redefinidos: estos métodos no se deben tener en cuenta al calcular esta métrica. SIX se propone como medida de la calidad en la herencia. SIX puede indicar cuando hay demasiados métodos redefinidos, de tal forma que las abstracciones pueden no ser apropiadas y sea necesario reemplazar su comportamiento. Generalmente, una subclase debería extender el comportamiento de la superclase con nuevos métodos más que reemplazar o borrar comportamiento 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, más especializada ha de ser la subclase.
- Nivel de granularidad: clase.
- Ciclo de vida: diseño, implementación.

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

- Definición: DIT mide el máximo nivel en la jerarquía de herencia. Se trata de la cuenta directa de los niveles en la jerarquía de herencia. En el nivel cero de la jerarquía de encuentra la clase raíz.
- Valoración: Chidamber y Kemerer proponen DIT como medida de la complejidad de una clase, la complejidad del diseño y la reutilización potencial, lo que se debe a que cuanto más profunda se encuentra una clase en la jerarquía, mayor es la probabilidad de heredar más métodos. Es una medida de cuántos ancestros pueden afectar a esta clase.

  En general, la herencia se maneja poco. [Cartwright y Shepperd, 1996] han encontrado una correlación positiva entre DIT y el número de problemas emitidos por el usuario. [Lorenz y Kidd, 1994] 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 se construyen 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 que surgen con esta métrica se deben a las diferentes características de la herencia, ya que DIT no queda claramente definida y no se puede ver como una medida de reutilización. Es fácil imaginar clases con gran profundidad en la jerarquía reutilizando menos métodos que una clase poco profunda, pero que es muy ancha.
- Nivel de granularidad: clase.
- Ciclo de vida: diseño.

4.6. Polimorfismo

4.6.1. Proporción de polimorfismo (Polymorphism Factor -POF-) [Abreu y Melo, 1996]

- Definición: es la proporción entre el número real de posibles situaciones polimorfas para una clase C_i y el máximo número posible de situaciones polimorfas en C_i. Es decir, es el número de métodos heredados redefinidos dividido entre el máximo número de situaciones polimorfas distintas posibles.
- Fórmula:

$$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} \left[M_d(C_i) \times DC(C_i) \right]}$$

  Donde:

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

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

El numerador representa el número de métodos heredados redefinidos.

El denominador representa el máximo número de situaciones polimorfas distintas posibles para la clase $C_{i.}$

- Valoración: POF es una medida del polimorfismo y una medida indirecta de la asociación dinámica en un sistema. El polimorfismo se debe a la herencia. En algunos casos, sobrecargando métodos se reduce la complejidad y, por tanto, se incrementa la facilidad de mantenimiento y la facilidad de comprensión del sistema. [Harrison et al., 1998] demuestran que no cumple todas las propiedades definidas en [Kitchenham et al., 1995] para ser válida, ya que en un sistema sin herencia el valor de POF resulta indefinido, exhibiendo una discontinuidad.
- Nivel de granularidad: clase.
- Ciclo de vida: diseño, implementación.
- Relacionado con: herencia.

4.7. Reutilización

4.7.1. 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, la cantidad de subclases que pertenecen a una clase.
- Valoración: Es un indicador del nivel de reutilización, la posibilidad de haber creado abstracciones erróneas, y es un indicador del nivel de pruebas requerido. Un mayor número de hijos requiere más pruebas de los métodos de esa clase y mayor dificultad para modificar una clase pues afecta a todos los hijos de dicha clase. Clases en un nivel más alto en la jerarquía deberían tener más subclases que las clases en un nivel más bajo en la jerarquía. NOC puede ser también un indicador del uso inadecuado de la herencia.

  Es un potencial indicador de la influencia que una clase puede tener sobre el diseño del sistema. Si el diseño depende mucho de la reutilización a través de la herencia, quizás sea mejor dividir la funcionalidad en varias clases.
- Nivel de granularidad: clase.
- Ciclo de vida: diseño.

4.8. Tamaño

4.8.1. 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.
- Valoración: El tamaño de un método se emplea para evaluar la facilidad de comprensión, la capacidad de reutilización y la facilidad de mantenimiento del código. Depende 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 qué métodos son candidatos a ser divididos en varios. No es una métrica recomendada para sistemas OO, pero es fácil de recoger y utilizar.

- Nivel de granularidad: método, clase, programa, sistema.
- Ciclo de vida: implementación.

4.8.2. 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.
- Valoración: NOM cuantifica el tamaño del método de una manera relativamente no sesgada. 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 sistemas no OO, y estas llamadas no se deberían contar en el número de mensajes enviados.
- Nivel de granularidad: método.
- Ciclo de vida: implementación.

5. Otras métricas

5.1. Métricas de cobertura de pruebas

Orientadas a las pruebas de caja blanca las métricas de cobertura de pruebas pretenden medir la minuciosidad de los conjuntos de prueba seleccionados, es decir, la compleción de la actividad de pruebas. Es el porcentaje de requisitos implementados (en forma de casos de pruebas definidos o capacidades funcionales) multiplicado por el porcentaje de la estructura del software probado (en unidades, segmentos, sentencias, bifurcaciones, o caminos de prueba) [Schulmeyer y MacKenzie, 2000]. Las hay de varios tipos:

- Cobertura de bloques de sentencias: es el ratio de bloques de sentencias ejecutadas por el conjunto de pruebas respecto del número total de sentencias del programa, para asegurar que cada sentencia ha sido probada al menos una vez. Un bloque de sentencias es un fragmento de código que se ejecuta en secuencia, sin bifurcaciones. Por ejemplo, en el siguiente fragmento de código escrito en lenguaje C, las sentencias 1-5, 8-10, 13 y 15 constituyen bloques de sentencias:

```
1 printf(“Teclee un número y un 0 para sumar y un 1 para
multiplicar”);
2 scanf(“%d”, &n, &k);
3 suma=0;
4 prod=1;
5 i=1;
6 while (i<=n)
```

```
7 {
8     suma+=i;
9     prod*=i;
10    i++;
11 }
12 if(k==0)
13    printf("n=%d, suma=%d\n", n, suma);
14 if(k==1)
15    printf("n=%d, producto=%d\n", n, prod);
```

Existen un total de 4 bloques de sentencias en el programa. Consideremos los siguientes casos de prueba para este programa:

- `n=0, k=0`
- `n=10, k=1`

En este caso, el ratio de cobertura de bloques de sentencias es del 100%, pues los 4 bloques de sentencias son ejecutados al menos una vez.

- Cobertura de bifurcaciones: es el ratio del número de bifurcaciones cubiertas por el conjunto de pruebas respecto del número total de bifurcaciones del programa. Para el ejemplo anterior también tenemos un 100% de cobertura, pues se ejecutan todas las ramas predicado posibles :
  - `while (i<=n)`
  - `if(k==0)`
  - `if(k==1)`
- Cobertura de caminos: cualquier secuencia de sentencias definida por el flujo de control del programa es un camino. Un camino desde la definición de una variable hasta su uso se denomina *"du-path"*. Mide el porcentaje de caminos de programa ejecutados. Debido a lo impracticable e ineficiente que resultaría probar todos los caminos, se suele reducir tratando todas las posibles iteraciones de un bucle como un camino. Para el ejemplo anterior se ejecutan todos los caminos posibles excepto los siguientes:
  - `n=>1, k=0`
  - `n=0, k=1`

  Por tanto, dado que de un total de 4 caminos posibles sólo se ejecutan 2, la cobertura de caminos alcanza sólo el 50%.
- Cobertura de flujo de datos: mide el ratio de pares *definición-uso* de variables y estructuras de datos cubiertas por el conjunto de pruebas respecto al número total de pares definición-uso del programa. Una *definición* es una sentencia, como por ejemplo `i=1`, que asigna un valor a una variable, mientras que la aparición de `i` en la sentencia `suma+=i` es un *uso* de `i`. Otro caso de uso es cuando `i` aparece dentro de un predicado, por ejemplo, en la sentencia `while (i<=n)`. Esto es lo que constituye un par *definición-uso*.

Las pruebas han de diseñarse con el objetivo de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y tiempo posible.

En orientación a objetos, la cobertura tradicional no es suficiente, ya que cuando los métodos son heredados por una clase derivada se hace necesario alguna prueba adicional. Esto es cierto tanto para aquellos métodos que han sido heredados sin modificación como para aquellos que han sido sobrecargados.

El uso de las métricas de cobertura que indican el porcentaje de código fuente ejercitado respecto del total ayuda en la definición/redefinición de casos de prueba para alcanzar todos los bloques de sentencias y bifurcaciones, mostrando aquellas partes del programa no cubiertas por

los conjuntos de pruebas. También permiten optimizar la selección de los casos de prueba para que con la mínima cantidad de casos se cubra el mayor porcentaje de sentencias del programa.

En cuanto a la jerarquía de las pruebas de cobertura, se sugiere completar la cobertura de bloques de sentencias antes de iniciar la cobertura de bifurcaciones.

El polimorfismo y el encapsulamiento del comportamiento son características principales del diseño orientado a objetos y las métricas de cobertura han de tener estas características en cuenta de cara a la prueba de software. IPL [IPL 1999] introduce una extensión a la tradicional cobertura estructural: cobertura de contexto orientada a objetos, dividida en tres clases:

5.1.1. Cobertura en contexto de herencia (Inheritance Context Coverage-ICC-) [IPL, 1999]

- Definición: tiene en cuenta la ejecución de los métodos en el contexto de la clase base y en el contexto de todas las clases derivadas. Alcanzar el 100% de cobertura en el contexto de herencia significa que el código debe ser completamente ejercitado en cada contexto.
- Valoración: ayuda a medir si las llamadas polimórficas en el sistema han sido probadas correctamente. Harrold [Harrold, 1999] propone la ejecución de pruebas de integración jerárquicas (HIT) que como primer paso ejercite todos los métodos en el contexto de una clase particular (la clase base o una clase derivada particular). Esta recomendación se aplica a todas las clases, de tal manera que las redefiniciones de un método en una clase derivada (métodos sobrecargados) son probados con el mismo nivel de detalle que la definición de la clase base original.
- Nivel de granularidad: método, clase.
- Ciclo de vida: implementación.

5.1.2. Cobertura en contexto basado en estados (State-based Context Coverage-SCC-) [IPL, 1999]

- Definición: tiene en cuenta la ejecución de los métodos en el contexto de clases que son descritas como “máquinas de estado”, cuyo comportamiento depende del estado particular en que se encuentren en un momento dado.
- Valoración: ayuda a medir si se han efectuado pruebas de cobertura para cada uno de los estados posibles de la clase. Una dificultad en la implementación suele ser la no disponibilidad del estado actual de la clase, y normalmente se debe cambiar el código para que devuelva los posibles estados.
- Nivel de granularidad: clase.
- Ciclo de vida: implementación.

5.1.3. Cobertura en contexto Multi-threaded (Multi-threaded Context Coverage-UCC-) [IPL, 1999]

- Definición: tiene en cuenta la ejecución de los métodos en el contexto de múltiples hilos de ejecución, manteniendo información de cobertura para cada hilo de ejecución.
- Valoración: permite aplicar el enfoque de cobertura de contexto a otros casos donde las métricas de cobertura tradicionales son inadecuadas, como aplicaciones con varios hilos de ejecución (*multi-threaded*).

- Nivel de granularidad: clase.
- Ciclo de vida: implementación.

6. Obtención de métricas con XML

Una de las cualidades deseables de las métricas ya expresada en el apartado 2.2 es la facilidad de recolección: el coste y esfuerzo para obtener la medida ha de ser razonable. En este apartado se propone un entorno para la obtención y análisis de métricas basado en el lenguaje de marcas XML.

El planteamiento parte de un trabajo paralelo llevado a cabo por los autores [Vázquez et al., 2001] en el que se describe la creación de un repositorio en formato XML para el código fuente de programas. Los programas convertidos a formato XML constituirán un repositorio con la información relevante de los ficheros en texto plano originales y dicha información podrá ser usada para los más diversos propósitos por herramientas de ingeniería de software creadas al efecto. En nuestro caso, se podrá utilizar para la recopilación de métricas de los programas. El objetivo del repositorio es independizarlo del lenguaje de programación utilizado, de forma que sólo sería necesario construir el correspondiente analizador sintáctico que convierta del lenguaje de programación elegido a formato XML. De esta forma sería suficiente con la creación de herramientas para el lenguaje XML, que sirve de sustrato común, que permitan la obtención de métricas para cualquier lenguaje. En la figura 2 se muestra un esquema del enfoque propuesto.

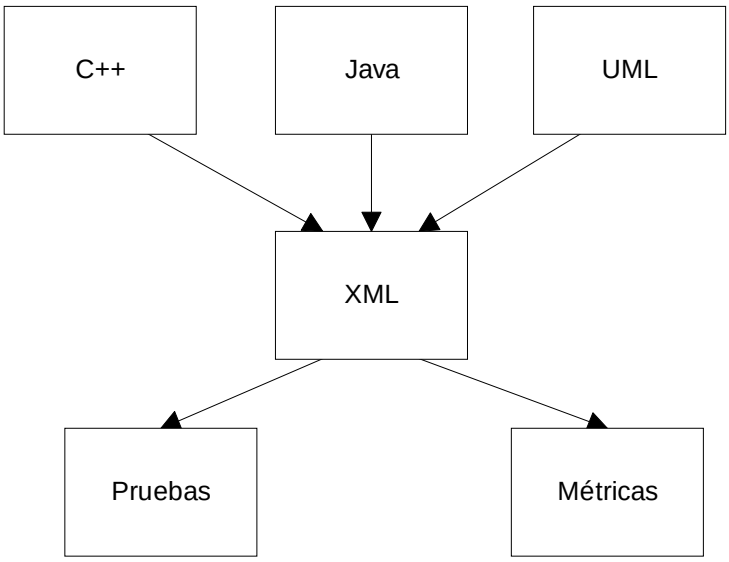

Figura 2. Obtención de métricas con XML

Se puede observar cómo este planteamiento no limita la recolección de métricas al código fuente de los programas, sino que la extiende a otras fases del ciclo de vida como el diseño.

Un ejemplo del aprovechamiento de herramientas ya existentes para el lenguaje XML sería el uso del conjunto de herramientas LT XML [UELTG, 2000] del *Language Technology Group* para hacer una inspección de un programa y obtener un inventario de elementos que utiliza con la utilidad `sgcount` para el applet `Clock2.java` (JDK1.3) y convertido a `Clock2.java.xml` con el unparser para JavaML de Greg J. Badros [Badros, 2000]:

```
/*
 * @(#)Clock2.java  1.5 98/07/09
 *
 * Copyright (c) 1997, 1998 Sun Microsystems, Inc. All Rights Reserved.
 */

import java.util.*;
import java.awt.*;
import java.applet.*;
import java.text.*;

/**
 * Time!
 *
 * @author Rachel Gollub
 */

public class Clock2 extends Applet implements Runnable {
    Thread timer;                // The thread that displays clock
    int lastxs, lastys, lastxm,
        lastym, lastxh, lastyh;  // Dimensions used to draw hands
    SimpleDateFormat formatter;  // Formats the date displayed
    String lastdate;             // String to hold date displayed
    Font clockFaceFont;          // Font for number display on clock
    Date currentDate;            // Used to get date to display
    Color handColor;             // Color of main hands and dial
    Color numberColor;           // Color of second hand and numbers

    public void init() {
        int x,y;
        lastxs = lastys = lastxm = lastym = lastxh = lastyh = 0;
        formatter = new SimpleDateFormat ("EEE MMM dd hh:mm:ss yyyy",
                                                    Locale.getDefault());
        currentDate = new Date();
        lastdate = formatter.format(currentDate);
        clockFaceFont = new Font("Serif", Font.PLAIN, 14);
        handColor = Color.blue;
        numberColor = Color.darkGray;

        try {
            setBackground(new
                    Color(Integer.parseInt(getParameter("bgcolor"),16)));
        } catch (Exception E) { }
        try {
            handColor = new
                    Color(Integer.parseInt(getParameter("fgcolor1"),16));
        } catch (Exception E) { }
        try {
            numberColor = new
                    Color(Integer.parseInt(getParameter("fgcolor2"),16));
        } catch (Exception E) { }
        resize(300,300);             // Set clock window size
    }

...

    // Circle is just Bresenham's algorithm for a scan converted circle
    public void circle(int x0, int y0, int r, Graphics g) {
        int x,y;
        float d;
        x=0;
        y=r;
        d=5/4-r;
        plotpoints(x0,y0,x,y,g);
```

```
        while (y>x){
            if (d<0) {
                d=d+2*x+3;
                x++;
            }
            else {
                d=d+2*(x-y)+5;
                x++;
                y--;
            }
            plotpoints(x0,y0,x,y,g);
        }
    }

...
```

Figura 3. Fragmento del applet Java de ejemplo: Clock2.java

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE java-source-program SYSTEM "java-ml.dtd">

<java-source-program>
<java-class-file name="C:/Proyecto/Clock2.java">
<import module="java.util.*"/>
<import module="java.awt.*"/>
<import module="java.applet.*"/>
<import module="java.text.*"/>
<class name="Clock2" visibility="public" line="42" col="0" end-line="226" end-
col="0">
  <superclass name="Applet"/>
  <implement interface="Runnable"/>
  <field name="timer" line="43" col="4" end-line="43" end-col="16"><type
name="Thread"/></field>
  <field name="lastxs" line="44" col="4" end-line="45" end-col="30"><type
name="int" primitive="true"/></field>

...

  <method name="init" visibility="public" id="Clock2:mth-60" line="53" col="4"
end-line="73" end-col="4">
    <type name="void" primitive="true"/>
    <formal-arguments/>
    <block line="53" col="23" end-line="73" end-col="4">
      <local-variable name="x" id="Clock2:var-166"><type name="int"
primitive="true"/></local-variable>
      <local-variable name="y" continued="true" id="Clock2:var-168"><type
name="int" primitive="true"/></local-variable>
      <assignment-expr op="="><lvalue><var-set
name="lastxs"/></lvalue><assignment-expr op="="><lvalue><var-set
name="lastys"/></lvalue><assignment-expr op="="><lvalue><var-set
name="lastxm"/></lvalue><assignment-expr op="="><lvalue><var-set
name="lastym"/></lvalue><assignment-expr op="="><lvalue><var-set
name="lastxh"/></lvalue><assignment-expr op="="><lvalue><var-set
name="lastyh"/></lvalue><literal-number kind="integer"
value="0"/></assignment-expr>
              </assignment-expr>
            </assignment-expr>
          </assignment-expr>
        </assignment-expr>
      </assignment-expr>
      <assignment-expr op="="><lvalue><var-set
name="formatter"/></lvalue><new><type
name="SimpleDateFormat"/><arguments><literal-string value="EEE MMM dd hh:mm:ss
yyyy"/><send message="getDefault">
              <target><var-ref name="Locale"/></target>
```

```
              <arguments/>
            </send>
          </arguments></new>
      </assignment-expr>
      <assignment-expr op="="><lvalue><var-set
name="currentDate"/></lvalue><new><type name="Date"/><arguments/></new>
      </assignment-expr>
      <assignment-expr op="="><lvalue><var-set name="lastdate"/></lvalue><send
message="format">
          <target><var-ref name="formatter"/></target>
          <arguments><var-ref name="currentDate"/></arguments>
        </send>
      </assignment-expr>
      <assignment-expr op="="><lvalue><var-set
name="clockFaceFont"/></lvalue><new><type name="Font"/><arguments><literal-
string value="Serif"/><field-access field="PLAIN"><var-ref
name="Font"/></field-access><literal-number kind="integer"
value="14"/></arguments></new>
      </assignment-expr>
      <assignment-expr op="="><lvalue><var-set
name="handColor"/></lvalue><field-access field="blue"><var-ref
name="Color"/></field-access></assignment-expr>
      <assignment-expr op="="><lvalue><var-set
name="numberColor"/></lvalue><field-access field="darkGray"><var-ref
name="Color"/></field-access></assignment-expr>
      <try>
        <block line="63" col="12" end-line="65" end-col="8">
          <send message="setBackground">
            <arguments><new><type name="Color"/><arguments><send
message="parseInt">
                    <target><var-ref name="Integer"/></target>
                    <arguments><send message="getParameter">
                        <arguments><literal-string
value="bgcolor"/></arguments>
                      </send>
                      <literal-number kind="integer" value="16"/></arguments>
                  </send>
                </arguments></new>
            </arguments>
          </send>
        </block>
        <catch><formal-argument name="E" id="Clock2:var-244"><type
name="Exception"/></formal-argument>
        </catch>
      </try>
      <try>
        <block line="66" col="12" end-line="68" end-col="8">
          <assignment-expr op="="><lvalue><var-set
name="handColor"/></lvalue><new><type name="Color"/><arguments><send
message="parseInt">
                  <target><var-ref name="Integer"/></target>
                  <arguments><send message="getParameter">
                      <arguments><literal-string
value="fgcolor1"/></arguments>
                    </send>
                    <literal-number kind="integer" value="16"/></arguments>
                </send>
              </arguments></new>
          </assignment-expr>
        </block>
        <catch><formal-argument name="E" id="Clock2:var-265"><type
name="Exception"/></formal-argument>
        </catch>
      </try>
      <try>
```

```
        <block line="69" col="12" end-line="71" end-col="8">
          <assignment-expr op="="><lvalue><var-set
name="numberColor"/></lvalue><new><type name="Color"/><arguments><send
message="parseInt">
                  <target><var-ref name="Integer"/></target>
                  <arguments><send message="getParameter">
                      <arguments><literal-string
value="fgcolor2"/></arguments>
                    </send>
                    <literal-number kind="integer" value="16"/></arguments>
                </send>
              </arguments></new>
          </assignment-expr>
        </block>
        <catch><formal-argument name="E" id="Clock2:var-286"><type
name="Exception"/></formal-argument>
        </catch>
      </try>
      <send message="resize">
        <arguments><literal-number kind="integer" value="300"/><literal-number
kind="integer" value="300"/></arguments>
      </send>
    </block>
  </method>

...

  <method name="circle" visibility="public" id="Clock2:mth-106" line="89"
col="4" end-line="109" end-col="4">
    <type name="void" primitive="true"/>
    <formal-arguments>
      <formal-argument name="x0" id="Clock2:frm-92"><type name="int"
primitive="true"/></formal-argument>
      <formal-argument name="y0" id="Clock2:frm-96"><type name="int"
primitive="true"/></formal-argument>
      <formal-argument name="r" id="Clock2:frm-100"><type name="int"
primitive="true"/></formal-argument>
      <formal-argument name="g" id="Clock2:frm-104"><type
name="Graphics"/></formal-argument>
    </formal-arguments>
    <block line="89" col="58" end-line="109" end-col="4">
      <local-variable name="x" id="Clock2:var-427"><type name="int"
primitive="true"/></local-variable>
      <local-variable name="y" continued="true" id="Clock2:var-429"><type
name="int" primitive="true"/></local-variable>
      <local-variable name="d" id="Clock2:var-433"><type name="float"
primitive="true"/></local-variable>
      <assignment-expr op="="><lvalue><var-set name="x"/></lvalue><literal-
number kind="integer" value="0"/></assignment-expr>
      <assignment-expr op="="><lvalue><var-set name="y"/></lvalue><var-ref
name="r" idref="Clock2:frm-100"/></assignment-expr>
      <assignment-expr op="="><lvalue><var-set name="d"/></lvalue><binary-expr
op="-"><binary-expr op="/"><literal-number kind="integer" value="5"/><literal-
number kind="integer" value="4"/></binary-expr><var-ref name="r"
idref="Clock2:frm-100"/></binary-expr></assignment-expr>
      <send message="plotpoints">
        <arguments><var-ref name="x0" idref="Clock2:frm-92"/><var-ref
name="y0" idref="Clock2:frm-96"/><var-ref name="x" idref="Clock2:var-
427"/><var-ref name="y" idref="Clock2:var-429"/><var-ref name="g"
idref="Clock2:frm-104"/></arguments>
      </send>
      <loop kind="while" line="97" col="8" end-line="108" end-
col="8"><test><binary-expr op=">"><var-ref name="y"/><var-ref
name="x"/></binary-expr></test><block line="97" col="19" end-line="108" end-
col="8">
```

```
          <if><test><binary-expr op="&lt;"><var-ref name="d"/><literal-number
kind="integer" value="0"/></binary-expr></test>
            <true-case>
              <block line="98" col="21" end-line="101" end-col="12">
                <assignment-expr op="="><lvalue><var-set
name="d"/></lvalue><binary-expr op="+"><binary-expr op="+"><var-ref
name="d"/><binary-expr op="*"><literal-number kind="integer" value="2"/><var-
ref name="x"/></binary-expr></binary-expr><literal-number kind="integer"
value="3"/></binary-expr></assignment-expr>
                <unary-expr op="++" post="true"><var-ref name="x"/></unary-
expr></block>
            </true-case>
            <false-case>
              <block line="102" col="17" end-line="106" end-col="12">
                <assignment-expr op="="><lvalue><var-set
name="d"/></lvalue><binary-expr op="+"><binary-expr op="+"><var-ref
name="d"/><binary-expr op="*"><literal-number kind="integer"
value="2"/><paren><binary-expr op="-"><var-ref name="x"/><var-ref
name="y"/></binary-expr></paren></binary-expr></binary-expr><literal-number
kind="integer" value="5"/></binary-expr></assignment-expr>
                <unary-expr op="++" post="true"><var-ref name="x"/></unary-
expr><unary-expr op="--" post="true"><var-ref name="y"/></unary-expr></block>
            </false-case>
          </if>
          <send message="plotpoints">
            <arguments><var-ref name="x0" idref="Clock2:frm-92"/><var-ref
name="y0" idref="Clock2:frm-96"/><var-ref name="x"/><var-ref name="y"/><var-
ref name="g" idref="Clock2:frm-104"/></arguments>
          </send>
        </block>
      </loop>
    </block>
  </method>

...

</class>
</java-class-file>
</java-source-program>
```

Figura 4. Fragmento del applet Java de ejemplo traducido a formato XML: Clock2.java.xml

```
literal-number          81
new                      9
method                  10
lvalue                  45
unary-expr               3
if                       4
superclass               1
var-set                 45
assignment-expr         45
block                   27
loop                     2
formal-argument         18
local-variable          20
literal-null             2
target                  57
cast-expr                6
arguments               77
try                      7
import                   4
send                    68
return                   2
field-access             3
this                     1
java-class-file          1
test                     6
catch                    7
formal-arguments        10
array-initializer        4
class                    1
implement                1
java-source-program      1
true-case                4
type                    76
false-case               1
var-ref                246
paren                    9
literal-string          24
binary-expr            111
field                   13
*Total*               1052
```

Figura 5. Salida de la herramienta LT XML para obtener estadísticas básicas del programa ejemplo: Clock2.java

A la vista de estos datos se puede extraer mucha información de utilidad para la obtención de métricas. Por ejemplo, el número de sentencias `if` (4), `loop` (2), servirán para calcular las métricas de complejidad ciclomática v(G) de McCabee, y con el número de bloques (27) darán una idea del número de casos de prueba necesarios para alcanzar el 100% de cobertura del programa. El número de clases (1) y el número de métodos (10), servirá para calcular la métrica de complejidad “métodos ponderados por clase WMC”, etc.

7. Conclusiones

Son bien conocidos diferentes conjuntos de métricas propuestas por distintos autores: Chidamber y Kemerer, Brito e Abreu, Lorenz y Kidd, etc. En este trabajo se han repasado las métricas consideradas más representativas de estos y otros autores, desde una perspectiva basada en las características específicas del enfoque orientado a objetos, más que en el análisis particular del conjunto de métricas propuesto por un solo autor. Esta aproximación quizá lleva a una mayor dificultad en la comprensión de las mismas, pues algunas miden las mismas propiedades pero desde diferentes perspectivas.

Se pone interés especial en las métricas de cobertura. Características específicas en OO como el polimorfismo y encapsulamiento sugieren nuevos enfoques en el diseño de casos de prueba para alcanzar el 100% de cobertura de métodos.

Por último se sugiere un marco de trabajo para la recolección automática de las métricas basado en un repositorio de programas fuente en XML que independiza del lenguaje de programación utilizado la construcción y uso de herramientas de obtención de métricas. La progresiva generalización del lenguaje de modelado unificado (UML) facilita la creación de herramientas de recolección y análisis de métricas en fases más tempranas del ciclo de vida, como la fase de diseño.

Bibliografía

[Abreu y Melo, 1996] Brito e Abreu F. y Melo, W., *Evaluating the impact of object oriented design on software quality*, Proceedings of 3rd international software metrics symp., 1996.

[Badros, 2000] Badros, Greg J., JavaML: *A Markup Language for Java Source Code*. Dept. Of Computer Sciece and Engineering, University of Washington, Seattle, WA, 2000. (http://www.cs.washington.edu/research/constraints/web/badros-javaml-www9.pdf).

[Boehm et al., 1978] Boehm, B.W., Kaspar, S.R., et al. *Characteristics of Software Quality*, TRW Series of Software Technology, 1978.

[Boehm, 1981] Boehm, B. W., *Software engineering economics*, Prentice-Hall, 1981.

[Booch et al., 1999] Booch, G., Rumbaugh, J. y Jacobson, I., *El Lenguaje Unificado de Modelado: Guía de usuario*. Addison Wesley, 1999.

[Briand et al., 1999] Briand, L.C., Daly, J.W. y Wüst, J.K., *A Unified framework for Coupling Measurement in Object-Oriented Systems*, *IEEE transactions on software engineering*, vol. 25, nº1, enero/febrero, 1999, pp. 91-121.

[Cartwright y Shepperd, 1996] Cartwright, M. Y Shepperd, M., *An empirical investigation of object software in industry*, Department of Computing, Talbot Campus, Bournemouth University, Technical Report TR96/01, 1996.

[Chidamber y Kemerer, 1994] Chidamber, S.R. y Kemerer, C.F., *A metric suite for object oriented design*, IEEE transactions on software engineering, vol. 20, nº6, junio, 1994, pp. 467-493.

[USC-CSE, 1997] University of Southern California- Center for Software Engineering, *Constructive Cost Model II (COCOMO II)*. 1997 (http://sunset.usc.edu/research/COCOMOII/index.html.)

[De Marco, 1982] De Marco, T., *Controlling software projects*, Yourdon Press Prentice-Hall, 1982.

[Fenton, 1991] Fenton, N. E., *Software measurement. A rigorous approach*, Chapman & Hall, 1991.

[Fenton y Pfleeger, 1997] Fenton, N. E. y Pfleeger, S.L., *Software metrics. A rigorous and practical approach*, PWS Pub., 1997.

[Genero et al., 2000] Genero, M., Manso, Mª.E., Piattini, M. y García, F.J., *Early metrics for object oriented information systems*. In Proceedings of the 6th International Conference on Object Oriented Information System (OOIS-2000). D. Patel, I. Chonthury, S. Patel y S.De Cesare (Eds), pp 414-425, 18-20 December 2000, London (UK).

[Harrold, 1999] Harrold, M.J. and McGregor, *Incremental Testing of Object-Oriented Class Structures*, 1999. (http://www.cs.clemson.edu/~johnmc/papers/TESTING/HIT/hit.ps).

[Henderson-Sellers, 1992] Henderson-Sellers, B., *A Book of Object-Oriented Knowledge*, Prentice Hall, NY, 1992.

[Henderson-Sellers, 1996] Henderson-Sellers, B., *Object-oriented metrics measures of complexity*, Prentice-Hall, 1996.

[IPL, 1999] IPL Information Processing Ltd. *Advanced Coverage Metrics for Object-Oriented Software*. 1999.

[Kitchenham et al., 1995] Kitchenham, B. Pfleeger, S. Y Fenton, N. *Towards a Framework for Software Measurement Validation*, IEEE Transactions on Software Engineering, 21(12), December 1995.

[Lorenz y Kidd, 1994] Lorenz, M. y Kidd, J., *Object oriented metrics*, Prentice-Hall, 1994

[McCall et al., 1977] McCall, J.A., Richards, P.K. y Walters, G.F. *Factors in software quality*. US Rome Air Development Center Reports NTIS AD/A-049 014, 015, 055 USA Air Force, 1977.

[McCabe, 1976] McCabe, T.J., *A Complexity Measure*, IEEE Transactions on Software Engineering, Vol. 5, 1976.

[McCabe et al., 1994] McCabe & Associates, *McCabe Object-Oriented Tool User's Instructions*, 1994.

[Paulk et al., 1993] Paulk, M.C., García, G.M. Chrissis, M.B. y Bush, M. *Capability Maturity model for software, version 1.1*, CMU/SEI-93-TR-24, Software Engineering Institute y Universidad Carnegie Mellon, febrero, 1993.

[Rumbaugh et al., 1991] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. y Lorensen, W., *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Schach, 1996] Schach, S.R., *Classical and Object-Oriented Software Engineering*, (3rd ed.) (Chicago: Irwin, 1996).

[Schulmeyer y MacKenzie, 2000] Schulmeyer, G.G. y MacKenzie, G.R. *Verification and Validation of Modern Software-Intensive Systems*, Prentice Hall, 2000

[SPICE, 1999] SPICE, *SPICE Document Suite, Software Process Improvement and Capability determination*, http://www.sqi.gu.edu.au/spice/, 1999.

[UELTG, 2000] University of Edinburgh Language Technology Group. *LT XML version 1.2*, Septiembre 2000. (http://www.ltg.ed.ac.uk/software/xml/xmldoc/xmldoc.html).

[Vázquez et al., 2001] Vázquez, P.J., Moreno, Mª.N. y García, F.J. *Verificación con XML*. Departamento de Informática y Automática – Universidad de Salamanca. 2001.