UNIVERSIDAD NACIONAL DE LA PLATA

# Mantenibilidad y Evolutividad en el Software Libre y de Código Abierto

JORGE RAMIREZ MORALES

SETIEMBRE DE 2010

# Mantenibilidad y Evolutividad en el Software Libre y de Código Abierto

Trabajo Final Integrador para la obtención del título de Especialista en Ingeniería de Software

**Autor**: Jorge Ramirez Morales

**Director**: M.Sc. Gustavo Gll

**Co-director**: Dr. Gustavo Rossi

Universidad Nacional de La Plata

Septiembre de 2010

# Índice de contenido

# Introducción

El presente trabajo parte del objetivo general de elaborar criterios para evaluar aspectos técnicos de productos F/OSS (Software Libre y de Código Abierto) como posibles bases o componentes para el desarrollo de aplicaciones.

Una de las características más importantes de este tipo de software es la disponibilidad pública del código; en muchos casos, además, también se dispone de amplia información sobre el desarrollo del producto. Las licencias bajo las cuales se distribuyen habitualmente los productos F/OSS no sólo habilitan sino que alientan a la reutilización y/o modificación de las aplicaciones.

Al momento de plantearse el desarrollo con F/OSS, sin embargo, es frecuente encontrar que la información pública de la cual se dispone es incompleta, deficitaria o desactualizada[36][65]; además, nada sabemos *a priori* respecto de la facilidad o dificultad que supondrá realizar cambios en un código desarrollado por una comunidad[87][79].

Estos aspectos representan limitaciones de peso para la reutilización del F/OSS. Resulta necesario, entonces, desarrollar modelos, criterios y herramientas que permitan evaluar las características de productos y proyectos F/OSS, y al mismo tiempo proveer a la comunidad de desarrollo de elementos que posibiliten atender la demanda de mantenibilidad[73].

Por otra parte, los productos F/OSS más extendidos se caracterizan por la continua adición de nuevas funcionalidades, adaptación a nuevos entornos y corrección de errores[26]; estas características implican una constante evolución del F/OSS, entendida como necesidad de que el software sufra modificaciones para seguir utilizándose en el tiempo[49].

La posibilidad de adaptar un producto de F/OSS para una necesidad particular o de integrarlo a un nuevo desarrollo supone considerar las dificultades que la evolución del producto pudiera plantear.

Desde el punto de vista del reuso de aplicaciones para desarrollo de software, las características que se vinculan con la mantenibilidad y la evolutividad constituyen elementos de valoración esenciales para la selección de programas y componentes.

Antes de indagar sobre las propiedades de los productos de software, es preciso definir

con claridad los conceptos de manteniminento y evolución y, derivados de ellos, los de mantenibilidad y evolutividad.

Estos atributos también son de carácter complejo, por lo que diversos autores han propuesto diferentes desagregaciones en atributos menores susceptibles de ser medidos. Estos conjuntos de atributos y sus relaciones con los conceptos que nos interesan constituyen modelos de mantenibilidad y evolutividad.

## Trabajos Previos

El presente trabajo se vincula con el Proyecto de Investigación 1690 "Desarrollo de un entorno virtual de enseñanza-aprendizaje para la Universidad Nacional de Salta", del Consejo de Investigación de la Universidad Nacional de Salta, bajo la dirección del M. Sc. Gustavo Gil.

Ya en el año 2007, integrantes del proyecto estimaron el esfuerzo que insumiría el desarrollo de una plataforma educativa como Moodle[67], partiendo del código fuente de esa aplicación.

Posteriormente, el estudio de la información disponible en repositorios F/OSS llevó a buscar indicadores de la utilización del seguimiento de errores en proyectos de ese tipo[66], con miras a evaluar el desempeño de proyectos de ese tipo respecto de la información y corrección de errores.

## Estructura del informe

En la primera parte relevaremos críticamente diversas definiciones de mantenimiento y evolución, enfatizando los aspectos problemáticos. Comenzaremos reviendo bibliografía de carácter general sobre Ingeniería de Software y sobre temas vinculados al mantenimiento y la evolución.

En la segunda sección revisaremos modelos de evolutividad y mantenibilidad, destacando el punto de vista desde el cual se proponen y su posible relevancia respecto de la evaluación de estos atributos en productos F/OSS.

En tercer lugar presentamos un recorrido de propuestas de medición de los atributos mencionados, tanto desde perspectivas teóricas como empíricas, enfatizando las caracte-

rísticas que a nuestro entender les otorgan importancia o validez para nuestro objetivo.

En la cuarta sección discutiremos las posibilidades y limitaciones de la utilización de la información públicamente accesible en proyectos F/OSS y analizaremos la validez o la utilidad de las propuestas relevadas en la sección anterior que consideremos más adecuadas para nuestros objetivos, en virtud de datos recogidos empíricamente de la información pública de proyectos F/OSS.

Finalmente, presentaremos nuestras conclusiones y los trabajos a realizar en el futuro.

# 1.- Mantenimiento y Evolución del Software

## 1.1 Mantenimiento y evolución del software

Los productos de software suelen sufrir modificaciones aún después de terminados o entregados. Estas variaciones pueden deberse a cambios en los requerimientos, modificaciones en la propia organización para la cual está destinado, necesidades de adaptación o fallas de funcionamiento.

Cualquiera sea la causa, el costo de producir las modificaciones en cuestión es generalmente elevado; no existe un consenso general respecto de la incidencia del mantenimiento en los costos de desarrollo, pero diversas estimaciones y estudios[27] lo ubican entre el 60% y el 90% del total invertido a lo largo de la vida de un producto de software.

Estos porcentajes pueden resultar sorprendentes desde una perspectiva intuitiva de la noción del mantenimiento, al que suele considerarse limitado a la corrección de errores; en el software las tareas de mantenimiento comprenden la adecuación del producto a las modificaciones en el entorno en el que debe desempeñarse (por ejemplo, adopción de una nueva plataforma de software o de hardware), cambios o agregados respecto de los requerimientos originales, etc.[49].

En definitiva, los productos de software debe modificarse necesariamente para poder seguir satisfaciendo la necesidad que les da origen en entornos organizacionales y tecnológicos que varían. Esta característica del software se conoce como evolutividad (evolvability) y constituye un área de investigación de interés creciente.[49][55]

No existe acuerdo general respecto del conjunto de atributos que inciden en las dificultades de mantenimiento, ni sobre la forma de evaluarlos; no obstante, existen numerosos trabajos que aportan perspectivas diferentes, a veces parciales o acotadas, y que es necesario cotejar para conformar una mirada abarcativa de las posibilidades y dificultades actuales.

### 1.1.1 Definiciones de Mantenimiento y Evolución del Software

En la guía para el cuerpo de conocimiento de la Ingeniería de Software [1] se define al Mantenimiento como un área de conocimiento de la Ingeniería de Software que com-

prende las actividades a realizar a lo largo de todo el ciclo de vida del software; si bien la fase de mantenimiento comienza una vez entregado el producto de software, las tareas de mantenimiento se inician mucho antes.

La guía mencionada retoma el estándar IEEE 1219 (estándar para el mantenimiento de software), donde se define al mantenimiento como "la modificación de un producto de software posterior a su entrega, con el fin de mejorar su performance u otros atributos, o para adaptarlo a un nuevo entorno".

En el glosario de terminología de la Ingeniería de Software de la IEEE[37] se define al mantenimiento como "el proceso de modificar después de su entrega un sistema de software o un componente para corregir errores, mejorar su performance u otros atributos o adaptarlo a un entorno modificado".

La visión del mantenimiento sustentada en estas definiciones se orienta principalmente a las actividades posteriores a la entrega del software, pese a las advertencias mencionadas en la guía SWEBOK[1].

Weiderman y otros[84], en un informe técnico del CMU/SEI, enfocan estos términos no sólo respecto del software sino en relación al sistema en el que opera. Estos autores diferencian Mantenimiento del Software (entendido como intervenciones de grano fino, a corto plazo y cambios localizados) frente a Evolución del Sistema (de grano más grueso, de alto nivel, una forma estructural de cambio que facilita el mantenimiento del software).

Sommerville[77] destaca que el software sufre modificaciones en sus requerimientos una vez que ya está en funcionamiento; el cliente o usuario puede advertir con el uso la necesidad de que el software incorpore nuevas prestaciones, o los cambios en los negocios, en el dominio o en el entorno pueden obligar a que el software sufra cambios para continuar prestando servicios. Esos cambios constituyen la evolución del software.

Para Sommerville, cuando la transición entre el desarrollo y las modificaciones no es continua, se aplica el término *mantenimiento*.

El mismo autor pone de relieve otras connotaciones que diferencian los dos términos en cuestión cuando se refieren al software. Denomina mantenimiento al proceso general de

cambiar el software una vez entregado, resaltando que la expresión se aplica especialmente cuando los equipos de desarrollo encargados de la modificación son diferentes a los desarrolladores originales.

Endres y Rombach[19] adoptan definiciones sencillas, aunque acotadas, tanto de mantenimiento como de evolución. Sostienen que *evolución* es el término que designa la adaptación de sistemas en funcionamiento a nuevos requerimientos; en tanto, *mantenimiento* se refiere a la actividad relativa a la eliminación de fallas.

Mens[55] observa que las nociones reflejadas en los estándares mencionados se asocia al ciclo de vida en cascada propuesto por Royce, y se orienta según una visión afín a la de las prácticas industriales; así, la idea de mantenimiento se refiere principalmente a la corrección de errores y pequeños ajustes, a realizarse luego de la entrega del software. Sin embargo, los productos de software se desarrollan para satisfacer una determinada necesidad y cumplir con una serie de requerimientos que corresponden a un momento determinado de una organización; con el tiempo, el ambiente en que trabaja el software sufrirá cambios que significarán demandas de modificación en el propio producto para no caer en la obsolescencia.

Grubb y Takang[27] revisan las definiciones de mantenimiento y de evolución de software, remarcando que muchas de ellas se centran en aspectos específicos mientras que otras son de carácter demasiado general. Las propuestas van desde la visión del mantenimiento como corrección de errores hasta "todo lo que se haga con el software después de entregado o lanzado".

Estos autores diferencian fundamentalmente visiones centradas en la corrección de errores, la adaptación a nuevos entornos y el soporte al usuario.

Finalmente, adoptan la definición propuesta por la IEEE Std. 1219 mencionada anteriormente.

Kemerer y Slaughter[43] sostienen que las actividades de mantenimiento se realizan en cualquier momento desde la implementación del desarrollo de un producto, en tanto que la evolución del software se define en función del comportamiento dinámico de los sistemas y cómo cambian a lo largo del tiempo.

Bennett y Rajlich[10] adoptan la definición de IEEE 1219 para el mantenimiento, pero utilizan esa expresión para las actividades generales post-lanzamiento, en tanto que evolución se reserva a una etapa de una fase dentro de un modelo de ciclo de vida en el que el desarrollo pasas por sucesivas secuencias de lanzamientos, evolución y revisión, a partir del lanzamiento inicial.

### *Cambios en el software después de la entrega*

Los dos términos, mantenimiento y evolución, se refieren a los cambios que se realizan en el software una vez entregado o publicado. Para definir con mayor precisión la incumbencia de cada uno de esos conceptos, conviene revisar las características de las modificaciones en cuestión y las actividades que se relacionan con ellas.

El estándar IEEE 1219[38] propone en la primera fase del proceso de mantenimiento la identificación, clasificación y priorización de las modificaciones o problemas a los que se referirá el mantenimiento. La clasificación se basa en la propuesta de Swanson y Lientz[51], y define los siguientes tipos de mantenimiento:

- correctivo
- adaptativo
- perfectivo
- emergencia

Las tres primeras fueron definidas originalmente por Swanson y Lientz, redefinidas en el estándar que estamos considerando.

En el Glosario Estándar de Terminología de la Ingeniería del Software[37] llama *mantenimiento correctivo* al que se realiza para corregir fallas; el *adaptativo* es el que se realiza para volver usable a un programa de computadora en un entorno modificado; el mantenimiento perfectivo, en tanto, es el que se lleva a cabo con el fin de mejorar el funcionamiento, la mantenibilidad u otros atributos del software.

La última categoría no figuraba en la clasificación de Swanson y se refiere a las actividades de mantenimiento no planificadas y que se realizan para mantener operativo el software. El Glosario Estándar para la Terminología de la Ingeniería de Software

incluye, en cambio, el tipo de mantenimiento *preventivo* que se refiere a las tareas que se realizan con miras a evitar futuros problemas.

En gran parte de la literatura sobre mantenimiento y evolución se retoma esta clasificación u otras similares basadas en ella; estas tipologías podrían permitir una mirada más precisa sobre las diferentes actividades de mantenimiento, con miras a obtener un conocimiento más profundo del proceso.

Bennet y Rajlich, en el trabajo citado anteriormente proponen un modelo de ciclo de vida basado en "evidencias empíricas" donde se vincula la evolución y el mantenimiento a etapas diferentes de la vida de un software.

La ilustración I muestra el modelo simplificado

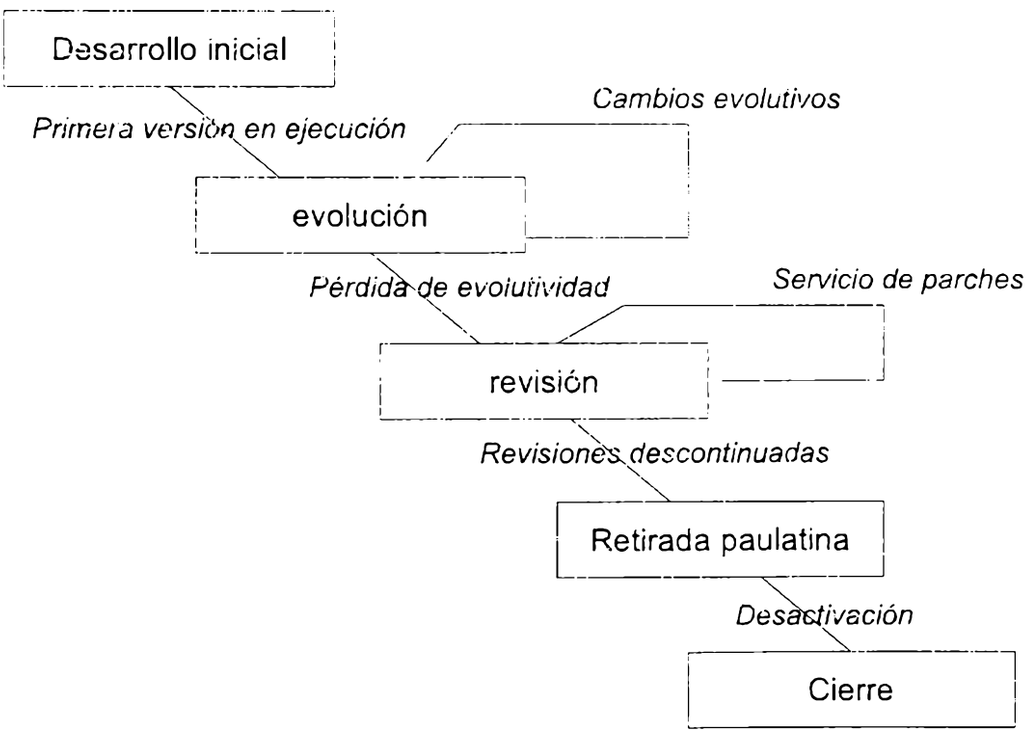


*Ilustración I: Ciclo de vida en etapas propuesto por Bennet y Rajlich*

En ese modelo, los cambios evolutivos responden a nuevos requerimientos y necesidades de adaptar el sistema a nuevos entornos; el servicio de parches se refiere a modificaciones tácticas para corregir errores o para incorporar pequeñas mejoras.

## *Similitudes y diferencias entre evolución y mantenimiento*

Evolución y mantenimiento de software se usan a veces como sinónimos,a veces con diferencias menores y otras como dos perspectivas diferentes sobre el cambio de los programas informáticos que se realizan posteriormente al lanzamiento o entrega del producto.

Pese a estas similitudes, existen diferencias semánticas que conviene explicitar.

Mens[55] cuestiona las connotaciones negativas del vocablo mantenimiento, que parece sugerir que el software sufre deterioros que deben ser corregidos.

Godfrey[25] retoma la observación de Parnas y otros según la cual el término mantenimiento connota mantener en funcionamiento un sistema sin producir modificaciones de diseño.

Jarzabek[39] propone diferenciar mantenimiento de evolución. El primero se refiere a las actividades día a día, en tanto que el segundo término alude a los cambios a largo plazo.

Cuando discutíamos las definiciones, mencionamos la perspectiva de Weiderman y otros respecto de la diferente granularidad a la que se aplican los términos, y el carácter sistémico y estructural de la evolución. En el mismo sentido, es interesante recordar la posición de Perry [62], quien advierte que las visiones de la evolución limitadas a las correcciones y mejoras deja de lado aspectos esenciales del trabajo de ingeniería de software, tales como el dominio en el que opera y los procesos que se llevan adelante y que también están involucrados en la evolución.

Cook, Ji y Harrison[14] presentan una perspectiva que intenta englobar las diferencias y similitudes entre los dos conceptos, considerando a la evolución como término más general que mantenimiento, pero sin establecer una frontera entre ambos.

Bennet y Rajlich[10] adoptan una perspectiva opuesta a la anterior, reservando el término "mantenimiento" a las modificaciones "tácticas", es decir, que no comprometen a la arquitectura. Para estos autores, los cambios más generales constituyen la evolución. Esos mismos cambios irían degradando la estructura o comprometiendo la arquitectura,

de modo de tornar cada vez más difícil la evolución hasta limitar los cambios a ajustes menores (mantenimiento), los que también se abandonarán cerca del final de la vida útil del sistema de software.

La falta de uniformidad y consenso en las definiciones dificulta el relevamiento de información sobre estos temas.

Resumiendo; los dos conceptos en cuestión se refieren a las modificaciones en el software, fundamentalmente después del lanzamiento o entrega del mismo; la noción de evolución enfatiza la adaptación a nuevos entornos o requerimientos, en tanto que el mantenimiento destaca la corrección de errores. Por otra parte, el mantenimiento se asocia frecuentemente a la actividad de corto plazo, con poca incidencia estructural sobre el software y descuidando el contexto en el que opera efectivamente.

A los fines del presente trabajo, resulta relevante conocer los aspectos que diferencian a los dos conceptos para determinar la vinculación de los mismos con el objetivo marco, es decir, la posibilidad de reutilizar productos F/OSS.

El desempeño de la comunidad que sostiene un producto F/OSS respecto de la corrección de errores representa un aspecto a considerar en la eventual selección para el reuso.

Por otra parte, la mayor o menor dificultad que represente la modificación del software, tanto para adaptarlo a necesidades particulares como para adicionar nuevas características, constituye un aspecto de gran importancia desde esa misma perspectiva.

Muchos de los productos F/OSS se desarrollan a lo largo de muchos años, sufriendo cambios y modificaciones de diversa profundidad. El contexto, el equipo de desarrollo, la comunidad, los métodos que se siguen y los requerimientos que enfrenta un producto F/OSS a lo largo de su vida útil siguen caminos diferentes a los tradicionales en el mundo empresarial; sin embargo, estos aspectos o dimensiones -en los términos de Perry- también intervienen en esa evolución[16]. Las visiones de evolutividad deberán tomar en cuenta estas características.

En definitiva, nos interesa conocer alternativas para evaluar la mantenibilidad y la evolutividad del software. Las propuestas que analizaremos, no obstante, no siempre se ajustan a la misma terminología, por lo que tendremos presente esa heterogeneidad.

### 1.1.2 Definiciones y Dificultades de la Mantenibilidad

Se denomina "mantenibilidad" a la facilidad con que puede modificarse un sistema o un componente de software para corregir fallas, mejorar su funcionamiento o adaptarlo a un nuevo ambiente[37].

Esta definición de carácter general no permite establecer cómo evaluar esta característica, ni qué atributos se vinculan con esta cualidad.

Broy, Deißenboeck y Pizka[12] hablan sobre los "mitos" acerca de la mantenibilidad. En un estudio realizado en 2003 sobre actividades de mantenimiento en organizaciones de Software de Alemania, observaron que sólo el 20% de los desarrolladores entrevistados llevaban un control de la mantenibilidad como parte del aseguramiento de la calidad. Más aún, sobre ese porcentaje que lleva adelante esa verificación, los criterios seguidos son muy diferentes: algunos verifican la orientación a objetos, otros se limitan a valorar la complejidad ciclomática o la cantidad de líneas de código por método, entre otras medidas.

Kajko-Mattson, Lehman[40] y otros elaboraron una serie de sugerencias para la elaboración de un modelo de mantenibilidad considerando que aún no se dispone de uno que tenga el suficiente consenso y que no se limite a definiciones generales. Esos autores afirman que no existe un modelo de mantenibilidad universalmente aceptado y que eso convierte a la evaluación de la mantenibilidad en un problema cercano a lo imposible; en cambio, la comunidad de software tiende a justificar que cada organización adopte su propia definición de mantenibilidad, en base a la complejidad del concepto; los modelos en uso se orientan a los procesos de software, antes que a los productos.

Entre las objeciones que presentan los autores mencionados, se destaca la insuficiencia del único estándar establecido (el ya mencionado ISO 9126) y la falta de bases comunes que orienten la investigación sobre aspectos de relevancia estratégica, dificultando la creación de modelos y métodos que proporcionen orientaciones para construir, preservar y evaluar la mantenibilidad.

En el trabajo en cuestión, los autores proponen un marco para desarrollar un modelo de mantenibilidad del producto.

### 1.1.3 Evolución y Evolutividad

El término Mantenimiento, tomado de otras disciplinas ingenieriles, suele percibirse como referido principalmente a la corrección de errores que se realiza a posteriori de la entrega o lanzamiento de un producto de software. Sin embargo, Pigoski señala que muchos estudios concluyen que la mayor parte del esfuerzo de mantenimiento se destina a actividades no correctivas.

En los años '70. Lehman y otros estudiaron los cambios en los productos de software basándose en la necesidad comprobada de que esos productos sufran modificaciones para seguir siendo útiles a lo largo del tiempo. Hablan, entonces, de *evolución* del software[49].

En los años '70 Lehman analizó los lanzamientos de las diferentes versiones del sistema operativo OS/360. A partir de los datos empíricos de ese estudio, postuló un conjunto de leyes de la evolución del software.

En el año '74 Lehman propuso las 3 primeras leyes; en 1980, incorporó otras dos y en 1996 postuló dos más, completando 8 enunciados[48].

Las leyes de Lehman parten de una clasificación de los sistemas de software en tres categorías: S, P y E.

Los programas de tipo S son aquellos correctos en sentido matemático, en relación con una especificación formal previa. En el tipo P se clasifican programas que en principio pueden especificarse formalmente, pero donde los usuarios tienen un interés particular en la corrección de los resultados. El tipo E, finalmente, se refiere a aquéllos que mecanizan una actividad social o humana, que modelan e interactúan de alguna manera con el mundo real[52]. Las leyes en cuestión se aplican únicamente a los programas de tipo E, más sujetos a cambios debidos a las demandas de los usuarios.

A continuación presentamos un repaso de las leyes de la evolución del software, a fin de poner de relieve las características de este proceso

#### *Leyes de la Evolución del Software*

Ley I: un programa de tipo E que está en utilización debe adaptarse continuamente o de

lo contrario será cada vez menos satisfactorio.

Ley II: La complejidad de los programas se incrementa a medida que evolucionan, a menos que se realice un trabajo específico para mantener la complejidad o reducirla.

Ley III: El proceso de evolución de los sistemas tipo E es autorregulado, con las medidas de procesos y productos siguiendo una distribución próxima a la normal.

Ley IV:Conservación de la estabilidad organizacional (tasa de trabajo constante). La tasa promedio de actividad global efectiva en un sistema de tipo E es invariante a lo largo de su vida.

Ley V: Conservación de la familiaridad. Mientras un sistema de tipo E evoluciona, las personas vinculadas con él (por ejemplo, desarrolladores, usuarios, vendedores, etc.) necesitan mantener un dominio sobre su contenido y comportamiento. Un cambio demasiado grande dificultaría el manejo de nuevas versiones del software, por lo que el crecimiento incremental promedio tiende a mantenerse constante durante la evolución.

Ley VI: Crecimiento continuo. La cantidad de funciones que ofrezca un software de tipo E debe aumentar con el tiempo para seguir satisfaciendo a los usuarios

Ley VII: Declinación de la calidad. La calidad del software de tipo E tenderá a decrecer, a menos que sea mantenido y adaptado a nuevos entornos de manera rigurosa.

Ley VIII: Sistema de retroalimentación. El proceso de evolución de los sistemas tipo E constituye sistemas de retroalimentación de nivel múltiple, multi-agente y multi-bucle y deben tratarse como tales para alcanzar mejoras significativas sobre una base razonable.

### *Evolutividad*

En su trabajo mencionado anteriormente, Cook y otros[14] definieron a la evolutividad como *la capacidad del software de evolucionar para continuar sirviendo a sus usuarios de una manera económicamente ventajosa.*

Esta definición es de carácter muy general, pero permite agrupar diversos estudios relacionados con la mantenibilidad y la evolutividad en función de la información que ofrecen respecto de la posibilidad de evolucionar de un producto de software.

En definitiva, la búsqueda de producciones científicas referidas a la evolutividad se

relacionan con la facilidad o dificultad que presenta un producto de software para ser modificado con el fin de continuar satisfaciendo la necesidad que le dio origen. Desde el punto de vista evolutivo, las modificaciones en cuestión pueden abarcar aspectos estructurales o arquitectónicos de la aplicación en cuestión.

# 2.- Medición de la mantenibilidad y la evolutividad

## 2.1 Métricas de Software

*"Toda calidad se manifiesta por una cantidad determinada, y sin cantidad no puede haber calidad"* Mao Ze Dong, Métodos de Trabajo de los Comités del Partido. Marzo de 1949

Para evaluar un producto respecto de una característica, necesitamos alguna forma de medirla; tenemos que asignarle algún valor (numérico o no) que nos permita comparar con la misma característica en otros productos.

Fenton y Pfleeger[23] definen a la medición como el proceso por el cual se asignan números o símbolos a atributos o entidades del mundo real, de modo de describirlos siguiendo reglas claramente establecidas.

Diferencian *medición* de *cálculo*, explicando que la primera se refiere a la obtención directa del valor de un atributo mientras que el segundo es indirecto y combina diversas mediciones en un ítem cuantificado que reflejan cierto atributo cuyo valor se intenta entender.

En su libro Object Oriented Metrics in Practice, Lanza y Marinescu[47] Adoptan una definición operacional de métricas, entendidas como el mapeo de una característica de una entidad sobre una escala numérica.

El enfoque del libro de Lanza es eminentemente práctico. Advierten que las métricas no son una panacea. Reconocen que es muy difícil la medición del diseño y la calidad del software; no obstante, sostienen que las métricas pueden brindar información importante a los ingenieros de software, pudiendo servir como punto de partida para un análisis más profundo.

Las métricas podrían ayudar como señales que llamen la atención sobre las entidades que presentan valores atípicos de una cierta métrica (excesivamente alto o excesivamente bajo, por ejemplo).

Para este trabajo, la búsqueda de métricas referidas a la mantenibilidad siguen la línea propuesta por Lanza y Marinescu: no pretendemos dilucidar un conjunto de mediciones definitivas, sino destacar aquellas que pueden aportarnos información relevante o indi-

cios valederos sobre las características de un proyecto o un producto de F/OSS.

### 2.1.2 Medición de la mantenibilidad

El administrador de proyecto de software precisa de elementos que permitan planificar y evaluar la mantenibilidad del software que se desarrolla; por lo tanto, es necesario contar con métricas que den cuenta de este aspecto, o -por lo menos- de un conjunto de atributos que el administrador considere relevantes.

Las mediciones deberían servir para indicarnos durante el desarrollo la probabilidad de que nuestro producto de software sea fácil de comprender, mejorar o corregir. Fenton y Pfleeger también advierten que la mantenibilidad no se relaciona sólo con el código; se vincula con la documentación de especificación y de diseño e incluso con el plan de pruebas[23].

Las visiones diferentes respecto de la mantenibilidad implican conjuntos diversos de atributos o factores que inciden en esa características y, por ende, distintos enfoques respecto de qué medir y cómo evaluar esta propiedad.

Estos conjuntos de propiedades y las relaciones entre ellas conforman modelos de mantenibilidad.

## 2.2 Mantenibilidad y Evolutividad en la bibliografía general sobre métricas

A continuación, presentamos un repaso de la cobertura de los temas de mantenibilidad y evolutividad como se presenta en la bibliografía general sobre métricas, a fin de tener un marco de referencia que permita comparar y contextualizar las propuestas que revisaremos después.

### 2.2.1 Mantenibilidad en Fenton y Pfleeger

En el clásico "Software Metrics: a Rigorous and Practical Approach"[23], Fenton y Pfleeger comienzan presentando las diferentes categorías de mantenimiento: correctivo, adaptativo, preventivo y perfectivo. Nótese que la última de estas categorías no coincide con las que se exponen con el estándar IEEE 1219-1998 mencionado anteriormente.

Sostienen que la mantenibilidad es un atributo externo del software, ya que depende no sólo del producto sino también de los responsables del mantenimiento. Consideran que hay dos grandes enfoques respecto del tema:

1) Enfoque externo: Medición de la efectividad del proceso de mantenimiento. Si el proceso es efectivo, entonces el producto es mantenible
2) Enfoque interno: identificar atributos internos del producto buscando predecir las características del proceso de mantenimiento.

El segundo enfoque es más apropiado para el administrador; sin embargo, los autores advierten la imposibilidad de definir la mantenibilidad exclusivamente en base a los atributos internos.

#### *Vista externa de la mantenibilidad*

Cualquiera sea la categoría de mantenimiento realizado, los autores proponen la medición del Tiempo Medio de Reparación (MTTR, Mean Time To Repair), como el tiempo promedio que lleva al equipo de mantenimiento la implementación de un determinado cambio.

A continuación sugieren las siguientes medidas, dependientes del entorno:

- Proporción del tiempo de implementación del cambio total respecto del número de cambios implementados
- Número de problemas sin resolver
- Tiempo gastado en problemas sin resolver
- porcentaje de cambios que introducen nuevos fallos
- número de módulos modificados para implementar un cambio

Este conjunto de medidas constituyen indicadores de aspectos de la actividad de mantenimiento; la selección de los instrumentos adecuados debe ajustarse a las metas y necesidades de la organización.

### *Atributos internos que afectan la mantenibilidad*

Fenton y Pfleeger enfatizan que la medición de atributos internos puede relacionarse con la mantenibilidad, pero no constituyen una medida de la misma; en todo caso, tales indicadores podrían servir para la evasión de riesgos respecto de la mantenibilidad.

Para seleccionar los atributos internos según su incidencia en la mantenibilidad, es necesario vincularlos a la evaluación externa.

Medidas simples, como la complejidad ciclomática, resultan insuficientes como indicadores de mantenibilidad, ya que capturan una visión muy limitada de la estructura del software.

Una serie de medidas que permitiría detectar problemas estructurales que se relacionan con la mantenibilidad es la familia VINAP, propuesta por Bache. Las mismas preservan los axiomas que él mismo postula para caracterizar la complejidad del flujo de control. Fenton y Pfleeger afirman que estudios empíricos sostienen que VINAP es un indicador de mantenibilidad más confiable que otras métricas de complejidad.

## *2.2.2 Kan: Métricas del proceso de mantenimiento*

En su libro "Metrics and Models in Software Quality Engineering"[41], Stephen Kan incluye un apartado sobre métricas de mantenimiento de software en su capítulo cuarto (Software Quality Overview).

El autor considera la fase de mantenimiento, entendida como una instancia posterior a la entrega del producto. En ese marco, las métricas no apuntan tanto a la calidad del producto sino más bien a la calidad del proceso de mantenimiento.

El proceso de mantenimiento será mejor cuanto antes y mejor sea la corrección de fallas desde el momento en que las mismas se detectan. De allí surgen medidas básicas, como la duración promedio del intervalo entre el informes de error y el tiempo promedio que demoran en cerrarse.

Una lista de errores pendientes al final de cada mes brinda información significativa respecto del proceso de mantenimiento.

Una métrica que recoge esta información es el BMI (Backlog Management Index), definido de la siguiente manera:

$$BMI = \frac{problemas\ cerrados\ en\ el\ mes}{problemas\ llegados\ en\ el\ mes} * 100$$

### *Tiempo de respuesta de corrección y reactividad de corrección*

Un indicador de la eficiencia del proceso de mantenimiento estará dado por el tiempo promedio que transcurre entre la comunicación de una falla y el momento en que se corrige.

En general, resulta más útil diferenciar las fallas por su gravedad, de modo que la organización encargada del desarrollo evalúe con mayor rigor la corrección de los defectos que pueden traer consecuencias más severas en el desempeño del software.

Es posible que las duraciones de los intervalos mencionados sean muy variables, por lo que podría ser más adecuado utilizar como medida a la mediana antes que a la media.

### *Porcentaje de correcciones demoradas*

La métrica "promedio (o mediana) del tiempo de respuesta" es una medida de tendencia central; una medida más sensible es el porcentaje de correcciones demoradas:

$$pcd = \frac{núm\ de\ correcciones\ cuyo\ tiempo\ excedido\ según\ su\ gravedad}{núm\ de\ correcciones\ producidos\ en\ el\ tiempo\ especificado}$$

### *Calidad de corrección*

La calidad de corrección o la cantidad de correcciones defectuosas constituye otra métrica de importancia respecto del proceso de mantenimiento. Una corrección es defectuosa si no resuelve el problema informado o genera nuevos problemas o errores.

### *Efectividad de la remoción de defectos*

En el capítulo 6, Kan[41] retoma el tema de la efectividad en la remoción de defectos. Realiza una revisión de la literatura.

En los 60 el desarrollo consistía básicamente en codificación y prueba, siendo la prueba de software la única actividad de remoción.

En los '70 adquirieron importancia la inspección y la revisión formal como maneras de favorecer la calidad del producto de software. Kan retoma [21] el trabajo de Fagan sobre inspección de software, de donde surge la noción de eficiencia de detección:

$$eficiencia\ de\ detección = \frac{errores\ encontrados\ en\ la\ inspección}{total\ de\ errores\ antes\ de\ la\ inspección} x\ 100$$

Pese a su importancia, esta medida no fue muy discutida hasta mediados de los '80, cuando Casper Jones propone la siguiente definición:

$$eficiencia\ de\ remoción = \frac{defectos\ encontrados}{defectos\ encontrados + defectos\ no\ encontrados} x\ 100$$

Se entiende por "defectos no encontrados" a los que se detectan con posterioridad al proceso analizado.

Kan también presenta las medidas propuestas por Kolkhorst y Macina[46], y que fueran elaboradas en el ámbito de los procesos de calidad definidos por IBM Houston en sus desarrollos para la NASA en los años '80:

$$Porcentaje\ de\ detección\ temprana = \frac{núm\ de\ errores\ mayores\ de\ inspección}{núm\ total\ de\ errores} x\ 100$$

Los errores mayores de inspección se refieren a los errores encontrados por la inspección de código y de diseño que podrían resultar en un Informe de Discrepancia válido (DR, Discrepancy Report) en caso de que esos errores se incorporen al software. Estos

informes de discrepancia registran una diferencia entre el software y la letra, los objetivos o los propósitos operativos expresados en la especificación de requerimientos.

El número total de errores es la suma de los errores mayores de inspección y los informes de discrepancia válidos.

En 1987 H. R. Dunn propuso una medida de eficiencia de remoción un poco diferente:

$$E = \frac{N}{N+S} * 100$$

Aquí E es la efectividad de una actividad, N el número de fallas detectados por la actividad y S las fallas encontradas con posterioridad. Esta métrica puede ajustarse limitando los errores considerados a aquéllos que estaban presentes durante el desarrollo de la actividad evaluada y que fueran susceptibles de evidenciarse mediante la actividad en cuestión.

En 1992 Daskalantonakis[15] presentó las métricas usadas por Motorola, entre las cuales figura la Efectividad Total de Contención de Defectos (TDCE) y la Efectividad de Contención de la Fase (PCEi)

$$TDCE = \frac{N^{o}\ de\ defectos\ antes\ del\ lanzamiento}{N^{o}\ de\ def.\ prelanzamiento + N^{o}\ de\ def.\ postlanzamiento}$$

$$PCE_i = \frac{n^{o}\ de\ errores\ de\ la\ fase\ i}{n^{o}\ de\ errores\ de\ la\ fase\ i + n^{o}\ de\ defectos\ de\ la\ fase\ i}$$

Kan observa que las métricas relevadas son muy similares, pudiendo llevar a confusiones; tales confusiones serían insignificantes cuando se trate de mediciones referidas al proceso global de remoción de defectos o a alguna fase específica puntual. Sin embargo, durante el desarrollo los defectos "posteriores" a la etapa aún no se manifestaron.

### *La efectividad en la remoción de defectos vista de cerca*

En las métricas relevadas hasta aquí no se consideró que no sólo se producen al comienzo del desarrollo sino que cada etapa puede incorporar nuevos errores.

Kan[41] esquematiza la relación de los defectos nuevos, detectados, y corregidos durante una etapa, así como los derivados de la tarea de corrección. El esquema se pre-

senta en la Ilustración II.

La eficiencia de la remoción para una etapa se definiría de la siguiente manera:

$$E = \frac{errores\ corregidos}{errores\ al\ comienzo\ de\ la\ fase + errores\ introducidos\ durante\ la\ fase} x\,100$$

Si se considera muy bajo el porcentaje de errores corregidos incorrectamente, el cómputo de la eficiencia puede limitarse a los defectos detectados.

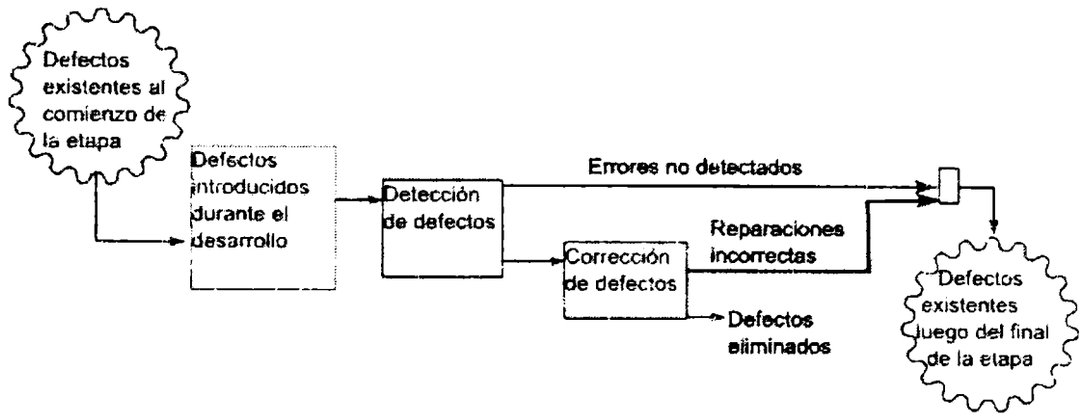


*Ilustración II: Los defectos durante una etapa del desarrollo, Según Kan*

Kan propone componer una matriz de defectos donde se deje constancia de la etapa en la que se encontró el error. De esta forma, se puede determinar la efectividad correspondiente a cada fase de acuerdo con la fórmula que figura más arriba.

### 2.2.3 Tian: Prevención de defectos y mejora de procesos

Jeff Tian incluye un capítulo referido a la prevención de defectos y mejora de procesos en su libro "Software Quality Engineering. Testing, Quality Assurance and quantifiable Improvement"[83].

Tian diferencia la prevención de defectos de otras actividades de aseguramiento de la calidad, las cuales tratan con defectos que están presentes una vez concluido el desarrollo.

Existen dos problemas en los enfoques de QA (Aseguramiento de la Calidad) en el software:

- la corrección de errores ya inyectados (incorporados al desarrollo) es enormemente costosa

- la ineficacia y las limitaciones de las técnicas de QA actuales para detectar errores en etapas tempranas.

Se pueden automatizar algunas actividades propensas a introducir errores; ese enfoque se denomina "bloqueo de errores"

Por otra parte, pueden analizarse las causas por las que se introducen actividades incorrectas; ese enfoque se denomina "remoción de fuentes de error".

Ambos enfoques integran las técnicas de prevención de efectos.

### *Análisis de las causas raíz para la prevención de defectos*

Al comienzo del desarrollo no se dispone de datos respecto de las fallas del software a desarrollar; por lo tanto, es necesario basarse en la experiencia previa, especialmente la que se refiere a productos o procesos similares a los que se prevé llevar adelante.

El análisis de causas puede realizarse de dos formas:

- Análisis lógico: se busca establecer las relaciones causales que vinculan errores y defectos. Esta aproximación exige especialistas con amplios conocimientos sobre los procesos de desarrollo, el dominio del sistema en cuestión, y el entorno general.
- Análisis estadístico: intenta establecer relaciones predictivas mediante correlaciones estadísticas en desarrollos previos.

A partir de la identificación de las causas, se pueden definir actividades de bloqueo de errores o remoción de fuentes de error.

## *Clasificación y análisis de errores*

### *Tipos generales de análisis de defectos*

Cuando se descubre un defecto en particular, se pueden realizar análisis individuales referidos a ese defecto; a medida de que se dispone de registros más amplios -una mayor cantidad de registros-, también pueden plantearse análisis de carácter más general.

En ambos casos, las preguntas que orientan el análisis son similares, por ejemplo:

- ¿Qué?: de qué se trata el defecto, en qué categorías podría incluirse
- ¿Dónde?: a fin de brindar información de retroalimentación mediante el análisis de la distribución de los defectos
- ¿Cuándo?: en qué fase o etapa del desarrollo se detectó o se produjo el error
- ¿Antes o después del lanzamiento?: es una extensión de la pregunta anterior, aunque de carácter más general.
- ¿Cómo y por qué?: determinar de qué manera se introdujo el defecto y por qué causas.

### *Análisis de distribución de defectos*

Este tipo de análisis buscar responder el "qué" el "dónde" de las preguntas anteriores.

Por lo general, estos análisis se centran en la corrección de defectos (encontrados durante la revisión de cualquiera de las instancias del desarrollo) y en las fallas que aún estén pendientes de resolución.

Los datos relevados pueden llamara la atención sobre módulos o procesos, para los cuales se podrán definir criterios y métricas con miras a su mejora en el futuro.

El análisis de tipo de defectos se relaciona con categorías que pueden implicar diferentes consecuencias o distintas relevancias. Tian presenta ejemplos de IBM, donde los defectos se vinculan con atributos de calidad (como capacidad, usabilidad, performance, confiabilidad, etc.), así como la clasificación de errores reflejados en el log de un servidor Web referidos a un sitio en particular.

Una vez detectados los defectos, se puede analizar la ubicación de los mismos. Tian ejemplifica con un estudio de IBM que pone de relieve qué módulos presentan mayor cantidad de defectos corregidos

### *Observaciones generales*

Los análisis presentados más arriba no permiten adoptar medidas tempranas para evitar defectos. Se necesitan medios para identificar áreas potencialmente riesgosas o de alta cantidad de defectos, a partir de los datos históricos.

### *Análisis de tendencias de defectos y modelos dinámicos de defectos*

Los datos referidos a los defectos incluyen habitualmente información temporal; como mínimo, los defectos que se encuentran se clasificarán como previos o posteriores al lanzamiento del producto. Es muy probable que la información refiera la fase del desarrollo en la que se produjeron los defectos.

A partir de esa información pueden esbozarse modelos tendientes a brindar predicciones.

Tian muestra la elaboración de una matriz donde se cuantifican los defectos para cada fase del desarrollo;

Tenemos hasta aquí un panorama de la diversidad y heterogeneidad con que se aborda la medición de la mantenibilidad y la evolutividad en la bibliografía general sobre métricas, y diferentes visiones acerca de los factores que influyen y los atributos que componen estas características.

Podemos ver que hay un consenso bastante extendido sobre la relevancia de ciertos aspectos en cuanto a su incidencia en la mantenibilidad y evolutividad; no obstante, los conceptos en juego no siempre se entienden de la misma manera.

En la siguiente sección repasaremos los resultados de diversos estudios empíricos, tomando en cuenta los métodos utilizados y remarcando eventuales fortalezas y/o debilidades de cada trabajo.

## 2.3 Modelos de mantenibilidad y de Evolutividad

No existe un consenso general respecto de cómo evaluar la mantenibilidad de un producto de software; sin embargo, se han producido una gran cantidad de trabajos referidos a las características que inciden en que un software sea fácil o difícil de mantener.

En el trabajo ya mencionado de Kajko-Mattson y otros[40] se hace hincapié en la necesidad de establecer bases que permitan evaluar los productos y no sólo los procesos, como proponen los estándares actuales (CMMI, ISO 9001 y Spice).

Los autores proponen crear un modelo de mantenibilidad del software (SMM por sus siglas en inglés), para lo cual postulan un conjunto de pautas generales para desarrollarlo. El modelo que plantean se orienta a evaluar la mantenibilidad en función del producto.

Entre los aspectos que debería considerar el modelo se destacan:

- Características comunes: se trata de definir y validar aspectos que inciden en la mantenibilidad en cualquier sistema de software
- Características variables: pueden variar según los tipos de software. Ejemplifican con el atributo "comprensibilidad" (understandability), que puede incluir factores descriptivos, de modularidad y estructurales que deberían considerarse bajo diferentes metodologías.
- Características cualitativas y cuantitativas: Es necesario definir modelos cualitativos y cuantitativos para evaluar diferentes características
- Características de rastreabilidad: la posibilidad de evolución de un artefacto depende fuertemente de la posibilidad de vincularlo con artefactos anteriores y posteriores, desde los niveles más bajos a los más altos.
- Niveles de madurez. Las características de mantenibilidad deberían asignarse a niveles diferentes de madurez, que reflejen la confiabilidad de que un producto de software satisfaga ciertas pautas de mantenibilidad.
- Taxonomía de defectos de mantenibilidad.

- Ejemplos de sistemas de software mantenibles y no mantenibles, que puedan proporcionarse para compararlos con sistemas de software específicos.

Pizka y Deißenböck[63], en cambio, sostienen que la mantenibilidad es una característica compleja que no se limita a las características del producto de software; por el contrario, consideran tres dimensiones de la mantenibilidad, compuestas por las propiedades técnicas del sistema en consideración, las habilidades de la organización que se encarga de la tarea y las características de la ingeniería de requerimientos. Para los fines del presente trabajo, resulta de mayor interés la incidencia de las propiedades técnicas del sistema.

Hashim y Key[31] se basan en los procesos que involucra la fase de mantenimiento, a partir de la diferenciación entre los tipos de mantenimiento correctivo, perfectivo y adaptativo. Sostienen, entonces, que la mantenibilidad puede verse como dos cualidades separadas: evolutividad y reparabilidad.

La reparabilidad se refiere al mantenimiento correctivo. Los autores definen a un sistema como "reparable" si permite la remoción de los defectos residuales presentes luego del lanzamiento o entrega del software, así como los errores que se introdujeran durante el propio mantenimiento.

Los autores consideran que la modularidad del sistema constituye un factor principal en la reparabilidad, permitiendo acotar la incidencia de los defectos a un conjunto reducido de módulos.

Respecto de la evolutividad, los autores advierten que la modularidad favorece la introducción de modificaciones asociadas con la evolución del software; sin embargo, las propias modificaciones pueden afectar negativamente la modularización original.

La evolutividad se relaciona con el mantenimiento perfectivo y el adaptativo.

Hashim y Key[31] citan el trabajo de Briand y otros donde identifican los siguientes factores que afectan a la mantenibilidad:

- Falta de rastreabilidad
- Falta de documentación

- Mal diseño e implementación
- Lenguaje de programación, herramientas y técnicas de desarrollo inapropiados.

Los autores proponen un modelo de atributos de la mantenibilidad del software de acuerdo con el cuadro que aparece a continuación.

| | |
|---|---|
| **Modelo de Atributos de Mantenibilidad** | Modularidad |
| | Legibilidad |
| | Lenguaje de programación |
| | Estandarización |
| | Nivel de validación y prueba |
| | Complejidad |
| | Rastreabilidad |

Kemerer y Slaughter[44], basándose en observaciones empíricas y en la literatura, sostienen que hay una serie de factores asociados con la cantidad y el tipo de mantenimiento que se realiza sobre determinados módulos. En particular, destacan tres:

- Funcionalidad
- Prácticas de desarrollo
- Complejidad del software

La funcionalidad de una aplicación puede determinar diferentes necesidades y frecuencias en su modificación. Los autores observan que los módulos de software en un sistema estratégico deben mejorarse con mayor frecuencia que otro dedicado principalmente a la gestión contable.

Respecto de las prácticas de desarrollo, sostienen que determinadas técnicas (por ejemplo, los generadores de código) producen aplicaciones menos propensas a errores.

La complejidad para estos autores se refiere a características del software que lo vuelven más difícil de entender y modificar. Sostienen que existen varias dimensiones de la complejidad (citando a Banker y otros[5]), pero se concentran en lo que denominan *densidad de decisión* o complejidad ciclomática.

La densidad de decisión, entendida como la cantidad de decisiones o caminos de control por línea de código, es un factor que se asocia reiteradamente con la complejidad, entendiendo que una mayor cantidad de rutas de ejecución obscurece la relación entre entradas y salidas y dificulta la comprensión del código por parte de los programadores.

David Peercy[61] presentó en 1981 un modelo con 6 factores que favorecen la mantenibilidad:

- modularidad
- descriptividad
- consistencia
- simplicidad
- expandibilidad
- instrumentación

La modularidad se refiere al particionamiento lógico de la aplicación y a la relativa independencia de sus partes. La descriptividad se refiere a la disponibilidad de descripciones del diseño del sistema y su documentación. La consistencia se refiere al cumplimiento de reglas de denominación (nombres de variables, de métodos, de módulos, por ejemplo), preferiblemente de acuerdo con convenciones y estándares referidos a los artefactos en cuestión.

La simplicidad es un atributo opuesto a la complejidad; toma en cuenta aspectos tales como el lenguaje de programación (Peercy lo ejemplifica indicando que un programa escrito en un lenguaje de alto nivel es más sencillo de entender que uno escrito en Assembler) así como la cantidad de operandos, operadores , estructuras de control anidadas, etc.

La expandibilidad se refiere a las características que facilitan el escalamiento de la aplicación (incorporación de nuevas prestaciones, por ejemplo). La instrumentación designa la inclusión de herramientas y técnicas de prueba integradas en el desarrollo.

En esta muestra observamos que no hay consenso respecto de los atributos de la mantenibilidad y los posibles factores que inciden en ella; no obstante, una mirada a las pro-

puestas presentadas más arriba y a los enfoque más frecuentes en el ámbito de la investigación sobre el tema, revela que algunos factores cuentan con gran aceptación tanto en ámbitos académicos como industriales.

Stavrinoudis y otros[81], en tanto, apelaron al modelo de Factores-Criterios-Métricas de McCall, considerando que los factores a tener en cuenta en la mantenibilidad son la consistencia, simplicidad, concisión, autodescriptividad y modularidad.

Hordijk y Wieringa[34] plantearon el estudio de la mantenibilidad de productos de software tomando los cambios como indicadores del atributo en cuestión. En base a esa consideración y a la literatura revisada por los autores, proponen la siguiente clasificación de factores:

- Factores del producto de software:
  - Nivel de especificación, donde consideran la corrección o el tamaño funcional.
  - Nivel de diseño, donde incluyen la modularidad y el acoplamiento.
  - Nivel de código, como el tamaño del código, la complejidad, la madurez o la duplicación
- Factores del proceso de mantenimiento
  - Frecuencia de los cambios
  - Procedimientos de mantenimiento
  - Procedimientos de prueba del software
- Propiedades de recursos
  - Propiedades del equipo de desarrollo, tales como tasa de actividad, estructura de comunicación, rotación del personal
  - Propiedades de los miembros del equipo, tales como las habilidades y el conocimiento del sistema
  - Infraestructura de mantenimiento, entre los que se considera el entorno de desarrollo y las herramientas.

- Propiedades del cambio
  - Propiedades funcionales del cambio, como el tamaño funcional, los tipos de cambio, o la complejidad funcional.
  - Propiedades de la implementación del cambio, entre los que se incluyen el tamaño del cambio, el alcance del cambio o el potencial del fallo.

El trabajo en cuestión se orienta a indagar qué decisiones de diseño tienen influencia sobre el esfuerzo de mantenimiento. La mantenibilidad a la que apuntan se refiere al esfuerzo que requiere el cambio funcional.

Riaz, Mendes y Tempero[69] presentaron en 2009 una revisión sistemática sobre métricas y predicción de la mantenibilidad. Allí observaron que los aspectos que con más frecuencia intentan relacionarse con la mantenibilidad fueron la complejidad, el tamaño y el acoplamiento, fundamentalmente relevados a nivel de código.

La clasificación que proponen Hordijk y Wieringa[34] permite organizar los factores a tomar en cuenta en la evaluación de la mantenibilidad de un producto de software. Si bien los factores y los atributos asociados en las perspectivas presentadas son diferentes, hay aspectos que se repiten con frecuencia.

Esta heterogeneidad permite comprender la cantidad de estudios que intentan establecer relaciones entre la mantenibilidad o la evolutividad y diferentes características del software o de los procesos de desarrollo. Estos últimos se definen como variables independientes, en tanto que se analizan las posibles correlaciones entre los datos obtenidos mediante diferentes métodos (correlación lineal, multilineal, árboles de regresión, etc.) para ofrecer una confiabilidad empírica en las relaciones sugeridas.

A la hora de plantearse estudios empíricos, resulta relevante también la definición de mantenibilidad que se adopta y los indicadores que se consideran para evaluar las variaciones en este atributo.

La complejidad aparece como un factor relacionado con la mantenibilidad, si bien no se explicitan con claridad la manera en que ese aspecto influye. En la próxima sección repasamos definiciones y propuestas de evaluación de esta característica.

## 2.4 La Complejidad del Software

A fines de los '80 Elaine Weyuker escribió un trabajo[85] que tuvo gran influencia en la investigación en las áreas de métricas y mantenimiento de software.

Weyuker se propuso formalizar las características que debería cumplir una medida de la complejidad del software. Para ello, consideró que los programas están constituidos por programas más pequeños, los que a su vez se integran con asignaciones y predicados. Una asignación constituye un "cuerpo de programa", y los cuerpos de programa pueden ser condicionales (se ejecutan según un predicado) o compuestos (integrado por otros cuerpos de programa).

Parte de la perspectiva de que una declaración de programa tiene la forma P(variables) -es decir, es una función de las variables de entrada- y una declaración de salida tiene la forma S(Variables).

De acuerdo con esas definiciones, un programa consta de una declaración de programa, un cuerpo de programa y una declaración de salida.

La complejidad de un cuerpo de programa P se expresaría mediante un valor numérico | P| no negativo con las siguientes características:

$$\forall P,Q|P|\leqslant|Q|\vee|Q|\leqslant|P|$$

Las propiedades que debería cumplir una métrica que represente la complejidad son las siguientes:

1) No todos los programas pueden tener la misma complejidad

$$(\exists P)(\exists Q)(|P|\neq|Q|)$$

2) Dado un número no negativo **c**, existe una cantidad finita de programas cuya complejidad es igual a **c**.

3) Existen programas diferentes P y Q para los cuales el valor de la complejidad es el mismo

4) Dos programas que realizan la misma función pueden tener valores de compleji-

dad diferentes (la complejidad no está dada por la función del programa)

$(\exists P)(\exists Q)(P \equiv Q \wedge |P| \neq |Q|)$

5) La composición de dos programas tendrá un valor de complejidad igual o mayor a cada uno de los programas integrantes

$(\forall P)(\forall Q)(|P| \leqslant |P;Q|) \wedge (|Q| \leqslant |P;Q|)$

6) La composición de programas deben poder arrojar complejidades, aún cuando los programas por separado presentan igual valor de complejidad.

$\exists P, \exists Q, \exists R / (|P| = |Q|) \wedge (|P;R| \neq |Q;R|)$

7) El valor de la complejidad debe ser sensible al orden en el que se encuentran las sentencias en un programa. Podria haber dos programas integrados por las mismas instrucciones dispuestas en órdenes diferentes y -como consecuencia- devolver valores diferentes de complejidad.

8) Si P es el programa Q renombrado, |P|=|Q|

9) La complejidad para la composición de dos programas debe poder ser mayor que la suma de las complejidades por separado.

A partir de esta serie de características, Weyuker observa que ni la complejidad ciclomática, ni la medición de esfuerzo (derivada del Volumen de Halstead), ni la complejidad del flujo de datos constituyen métricas que reflejen apropiadamente la complejidad de un programa.

En el año 1986 Kearney y otros publicaron un artículo[42] que tuvo amplio impacto respecto del concepto y medición de la mantenibilidad.

Los autores plantearon la falta de consenso en cuanto a la definición de qué se entiende por complejidad y objetaron la falta de una teoría que modele los programas, los programadores, el entorno de programación y las actividades de programación. Citan la definición de Basili, que entiende a la complejidad como la medida de los recursos gastados por el sistema en la interacción con una porción de software para realizar una

tarea determinada. En este caso, si la interacción a la que se refiere la definición se produce con un programador, la complejidad corresponderá a la dificultad para la codificación, depuración, prueba y modificación del software. El trabajo sostiene que esta última es la interacción a la cual se hace referencia generalmente con la expresión en cuestión.

A continuación, repasan las medidas utilizadas más ampliamente, como las derivadas de la "ciencia del software" de Halstead[29], la complejidad ciclomática de McCabe[54] y la métrica propuesta por Henry y Kafura[32].

Los autores también proponen una serie de características que deberían cumplir las métricas de complejidad:

- Robustez. Las métricas deben ser sensibles a las modificaciones del programa, de modo que una modificación tendiente a reducir la complejidad se refleje en la variación de las métricas en cuestión
- Normatividad: las métricas deben brindar criterios contra los cuales evaluar y comparar los valores que se obtienen, con el fin de caracterizar la complejidad y adoptar decisiones al respecto.
- Prescriptividad: las métricas no sólo deben permitir evaluar el nivel de complejidad de un producto sino también sugerir métodos para reducirla.

A pesar de que las características no se definen con rigurosidad, esta serie constituye una guía de interés, junto a las propiedades presentadas anteriormente y propuestas por Weyuker.

Honglei, Wei y Yanan[33] definen a las métricas de complejidad como la medida del costo consumido en desarrollar, mantener y usar el software. Consideran que la complejidad es el principal factor que puede inducir a defectos en el software.

Los autores destacan que el concepto de complejidad incluye diversos aspectos; presentan una serie de aspectos que describen la complejidad del software:

1) la dificultad para entender el programa
2) la dificultad para corregir defectos y mantener el software

3) la dificultad de explicar el software a otras personas
4) la dificultad de actualizar el programa de acuerdo con ciertas reglas determinadas
5) la carga de trabajo de escribir programas de acuerdo con el diseño
6) la disponibilidad de los recursos necesarios cuando el programa se ejecuta.

Posteriormente, resumen los presupuestos en los que se basan los diferentes enfoques respecto de la mantenibilidad; destacan las relaciones postuladas entre la complejidad, las estructuras de datos, las estructuras de control, el acoplamiento y -en el ámbito de la Programación Orientada a Objetos- la herencia y el acoplamiento entre clases.

Ilustran el planteamiento con dos métricas que se utilizan ampliamente como indicadores de complejidad: la Complejidad Ciclomática de McCabe y el conjunto de métricas orientadas a objetos de Chidamber y Kemerer.

### 2.4.1 La Complejidad Ciclomática

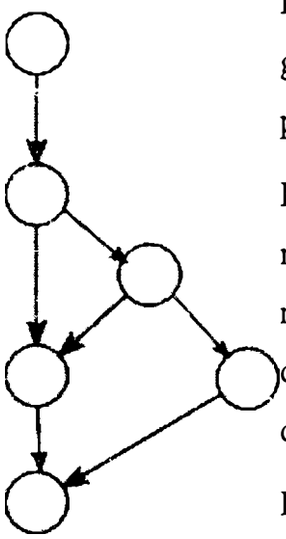

*Ilustración III: Grafo dirigido con 3 caminos independientes*

En 1976, McCabe definió la complejidad ciclomática de un programa como la cantidad de caminos independientes a través del programa.

Esta medida supone que a mayor cantidad de caminos alternativos, más complejo será el software. McCabe sostenía que que un módulo que alcanzara un valor mayor de 10 para esta métrica debería ser reescrito, ya que sería propenso a errores y muy difícil de mantener.

La cantidad de caminos independientes puede determinarse sumando uno a la cantidad de decisiones en el programa.

En la ilustración vemos un grafo en el que existen dos nodos que permiten caminos diferentes; por lo tanto, en ese ejemplo, la complejidad ciclomática alcanzará el valor de 2.

Kan[41] señala que diversos estudios comprueban una correlación fuerte entre la complejidad ciclomática y la cantidad de defectos; no obstante, observan que también se

manifiesta una correlación significativa entre esta métrica y el tamaño del software, medido en líneas de código.

Fenton y Pfleeger[23] observan que la complejidad ciclomática representa una vista parcial, ya que un programa puede tener muchos caminos independientes y sin embargo ser fácil de entender. De esa observación se desprende que esta métrica, al menos por sí sola, no constituye un indicador de complejidad.

Esta crítica se extiende a los trabajos empíricos y a las propuestas de validación de métricas que utilizan a la complejidad ciclomática como representación de la mantenibilidad o de la complejidad total del software.

### 2.4.2 Ciencia del software de Halstead

En 1970, Halstead postuló la Ciencia del Software[29][41][64] diferenciándola de la ciencia de la computación. Este enfoque se basó en el análisis del código fuente, definiendo una serie de medidas en función de los elementos que componen un programa:

$n_1$: cantidad de operadores diferentes en el programa

$n_2$: cantidad de operandos diferentes en el programa

$N_1$: cantidad de ocurrencias de operadores

$N_2$: cantidad de ocurrencias de operandos

A partir de estas definiciones, propone una serie de medidas como la "longitud" del programa, que se calcula según la siguiente fórmula:

$$N = n_1 \log_2(n_1) + n_2 \log_2(n_2)$$

Halstead considera las comparaciones mentales que serían necesarias para escribir un programa de longitud N, definiendo el *volumen del programa* con la siguiente fórmula:

$$V = N \log_2(n_1 + n_2)$$

Propone calcular el nivel del programa relacionando el Volumen con el "volumen potencial" V' , que correspondería a la implementación mínima del programa:

$$L = \frac{V'}{V}$$

La inversa del nivel se denomina "Dificultad del programa".

Estas métricas alcanzaron gran difusión, y en muchos casos se utilizaron como referencia para la validación de otras métricas. Pressman[64] afirma que "se ha llegado a un buen acuerdo entre lo previsto analíticamente y los resultados experimentales". Sin embargo, Fenton y Pfleeger[23] señalan deficiencias importantes, al punto de afirmar que constituyen "un ejemplo de medición confusa e inadecuada"; objetan la falta de consenso acerca del significado de los atributos medidos (es decir, a qué se refieren el volumen, la longitud, la dificultad, etc.) y -por ende- respecto de los atributos del mundo real que se pretende reflejar.

A pesar de esas objeciones, diversos estudios empíricos sugieren correlaciones entre los valores calculados por estas fórmulas y algunos indicadores de complejidad o mantenibilidad. Si bien no hay comprobaciones claras acerca de cómo se vinculan estas medidas, siguen siendo utilizadas.

### *2.4.3 Complejidad y Acoplamiento*

Las métricas presentadas más arriba no toman en cuenta la interacción entre diferentes partes de un programa; intuitivamente, podemos suponer que mientras mayor es la interacción entre partes, mayor será la dificultad para comprender y modificar el código.

Entre las primeras métricas que intentaron considerar el acoplamiento, alcanzaron gran difusión las de Henry y Kafura[32], quienes en 1981 se propusieron medir la complejidad del "flujo de información".

El acoplamiento se mide por la cantidad de módulos que llaman a un módulo determinado (llamado Fan-in o $f_{in}$ ) y la cantidad módulos a los que él llama (Fan-out o $f_{out}$).

Los módulos que prestan servicios a muchos otros y que -por lo tanto- tienen un valor alto de fan-in, suelen necesitar poco de los servicios de los demás módulos (lo que significa un valor bajo de fan-out). Entonces, valores altos de ambas métricas para algún módulo podrían sugerir problemas de diseño[41].

Henry y Kafura proponen una *complejidad de flujo de información* de un módulo M según la fórmula fcc(M)= longitud(M) x $(f_{in} \times f_{out})^2$.

Fenton y Pfleeger[23] señalan que estas métricas no se adecuan a la teoría de la medición, destacando que muchos módulo tendrían un valor de complejidad igual a 0 por el sólo hecho de no invocar a otro módulo o no ser invocado por ningún otro (ya que en ambas fórmulas se multiplican los valores). Otro aspecto que contradice la teoría de la medición es la "naturaleza híbrida" de la medida, que combina la longitud con las relaciones funcionales de los módulos.

Shepperd e Ince[76] analizaron la propuesta de Henry y Kafura tanto mediante la exploración de relaciones empíricas como desde la teoría de la medición; en base a las críticas a aquellas métricas postularon un refinamiento respecto de cómo contabilizar llamadas recursivas, variables compartidas, etc. Con esas especificaciones en cuanto a la forma de medir fan-in y fan-out, postulan la "Complejidad de Shepperd"=(fin x fout)2 (Ver Fenton y Pfleeger)

Las medidas relevadas hasta aquí no contemplan las características específicas del diseño y el desarrollo orientado a objetos. En la siguiente sección revisaremos brevemente el mundo de las métricas propuestas para los desarrollos bajo ese paradigma, destacando aquellas que se relacionan con la mantenibilidad y evolutividad.

## 2.5 Métricas y Orientación a Objetos

La difusión del desarrollo orientado a objetos trajo consigo la necesidad de contar con medios para medir distintos aspectos del desarrollo. En particular, cobró importancia la búsqueda de medidas que reflejen la estructura y la organización de las aplicaciones.

Resulta de interés para nuestro objetivo encontrar métricas que describan diferentes atributos estructurales de los programas orientados a objetos por dos razones:

- La hipótesis de que existen aspectos estructurales que inciden en la mantenibilidad
- La posibilidad de evaluar no sólo las características del código sino también realizar inferencias sobre el diseño de una aplicación a partir de las características

del código.

## 2.5.1 Métricas CK

En 1990 Chidamber y Kemerer[13] presentaron una serie de métricas para sistemas orientados a objetos. Las medidas propuestas se enmarcan en los conceptos ontológicos del filósofo argentino Mario Bunge, y retoman la aplicación a los sistemas orientados a objetos iniciada por Yand y Webber en 1970.

Las métricas de Chidamber y Kemerer (a menudo mencionadas como CK) alcanzaron amplia difusión; están entre las más referenciadas y existen muchas herramientas comerciales vinculadas a las actividades de aseguramiento de calidad que obtienen estas mediciones[18]. También fueron utilizadas en el ámbito específico de los estudios de calidad sobre Software Libre y de Código Abierto[79].

El conjunto de métricas propuesto por Chidamber y Kemerer es el siguiente:

### *WMC*

Métodos ponderados por clase: es la suma ponderada de los métodos de una clase. Se vincula con la noción de complejidad. Esta métrica se calcula de acuerdo con la siguiente fórmula:

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

donde $c_i$ es la "complejidad" del i-ésimo método. Esa complejidad se representa frecuentemente mediante la complejidad ciclomática.

Intentar computar la complejidad ciclomática choca con las características propias de la Orientación a Objetos, puesto que sería difícil medir ese valor cuando algunos métodos son heredados. Kan señala que en muchos estudios se asume que la complejidad de los métodos es similar, de modo que esta medida se reduce a contar la cantidad de métodos.

### *DIT*

Profundidad del árbol de herencia: es la cuenta de la cantidad de ancestros de una clase dada. Se vincula con el alcance de las propiedades de una clase y con la cantidad de clases que pueden afectar a la actual mediante herencia.

### *NOC*

Número de hijos: es la cantidad de subclases inmediatas de una clase dada.

### *CBO*

Acoplamiento entre objetos: Una clase está acoplada a otra si invoca métodos o variables de instancia de esta última. CBO de una clase es la cantidad de otras clases a las que está acoplada[41]

### *RFC*

Respuesta por clase: es el número de métodos que pueden ejecutarse en respuesta a un mensaje recibido por un objeto de esa clase[41].

### *LCOM*

Falta de cohesión: es la cantidad de métodos "disjuntos" (no comparten variables locales). Se puede calcular como la diferencia entre la cantidad de pares de métodos que comparten variables de instancia y los pares de métodos que no comparten ninguno

Cuando esta diferencia arroja un número negativo, se asigna cero.

### *Críticas a las métricas CK*

Chidamber y Kemerer desarrollaron su famoso conjunto de métricas en base a las categorías ontológicas de Bunge, de donde derivaron definiciones que fundamentaron las medidas propuestas. Desde otros puntos de vista, hubo diversas objeciones respecto de su validez.

Hitz analizó el grupo CK desde la perspectiva de la Teoría de la Medición. Objetó que los atributos que supuestamente representan algunas de las métricas no se comportan de la misma forma que éstas. Por ejemplo, el acoplamiento entre objetos -que en CBO se limita a contar las clases con las que cada una se acopla- no toma en cuenta que algunas vinculaciones suponen acoplamientos más fuertes que otros.

## *2.5.2 Las métricas MOOD*

En 1994, Brito e Abreu y Carapuça[2][30] propusieron un conjunto de métricas que intentaba capturar las características específicas del diseño Orientado a Objetos.

Estos autores definieron un conjunto de métricas que representan aspectos del comportamiento a nivel de clase.

El total de métodos definidos para una clase C se denota con $M_d(C)$ y comprende a los métodos nuevos definidos en esa clase (denotados como $M_n(C)$) y los métodos heredados pero reescritos ($M_o(C)$). El total de métodos incluye también a los que se heredaron sin modificación ($M_i(C)$). También definieron DC(C) como la cantidad total de descendientes de la clase C, CC(C) es la cantidad de hijos (descendientes directos, solamente), PC(C) cuenta los padres de la clase y AC(C) el total de ancestros.

Los atributos se cuentan de manera similar, considerando si fueron nuevos, reescritos o heredados. De allí surgen $A_a(C)$ -todos los atributos de la clase C-, $A_i(C)$ -atributos heredados- $A_o(C)$ -atributos sobreescritos y $A_d(C)$ -atributos definidos en la clase.

No todos los lenguajes permiten herencia múltiple; es importante señalar que los trabajos de Brito y Carapuça se orientaron a los lenguajes C++ y Eiffel.

Las siguientes métricas buscan reflejar el encapsulamiento:

El Factor de ocultamiento de métodos (MHF por sus siglas en inglés) busca contabilizar los métodos que no son visibles para todas las clases, calculándose como:

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

TC es el total de clases y $V(M_{mi})$ se define de acuerdo con la siguiente fórmula:

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

La función isVisible se define de la siguiente forma:

$$isVisible(M_{mi}, C_j) = \begin{cases} 1 \; si \; j \neq i \wedge C_j \; puede \; llamar \; M_{mi} \\ 0 \; en \; otro \; caso \end{cases}$$

En definitiva, $V(M_{mi})$ es el porcentaje del total de clases para las cuales el método $M_{mi}$ es visible.

De la misma manera, computa el Factor de Ocultamiento de Atributos (AIH), como la cantidad total de atributos que no son visibles para cada una de las clases

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

Estos autores intentaron también medir el polimorfismo, mediante el Factor de Polimorfismo POF:

$$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) * DC(C_i)]}$$

El numerador es la suma de todos los métodos sobreescritos

Para el acoplamiento, definen la función esCliente($C_c$, $C_s$) vale 1 cuando la clase $C_c$ es cliente de $C_s$, lo que significa (para estos autores) que $C_c$ hace al menos una referencia a un método o atributo de $C_s$ que no derive de herencia.

A partir de esa función, proponen el Factor de Acoplamiento (COF)

$$COF = \frac{\sum_{i=1}^{TC} [\sum_{j=1}^{TC} esCliente(C_i, C_j)]}{TC^2 - TC}$$

La herencia se evalúa con el cociente entre la suma de métodos heredados y el total de métodos, siempre referido a todas las clases. Este cociente se denomina Factor de Herencia de Métodos (MIF)

$$MIF = \frac{\sum_{j=1}^{TC} M_i(C_j)}{\sum_{j=1}^{TC} M_a(C_j)}$$

Siguiendo el mismo razonamiento, se define al Factor de Herencia de Atributos AIF

$$AIF = \frac{\sum_{j=1}^{TC} A_i(C_j)}{\sum_{j=1}^{TC} A_a(C_j)}$$

# 2.6 Estudios Empíricos

## 2.6.1 Métricas, Validación y Predicción

Como vimos anteriormente, muchas de las métricas relevadas fueron criticadas desde el punto de vista teórico. Otro enfoque para evaluar la validez o importancia de las métricas que se proponen se basa en investigar las posibles vinculaciones causales o relaciones funcionales entre los valores de esas métricas y los atributos que nos interesan (en nuestro caso, la mantenibilidad y la evolutividad).

La pregunta a responder es si los valores de las métricas consideradas inciden en una mayor o menor cantidad de fallas en el software desarrollado, o si afectan la facilidad o dificultad de modificación.

Los estudios en cuestión también son heterogéneos; difieren en la forma de medir los atributos de interés y las técnicas estadísticas usadas para evaluar la vinculación entre las métricas y los indicadores de atributos elegidos por los investigadores en cada caso.

## 2.6.2 Algunos estudios empíricos

En 1984 Basili y Perricone[8] analizaron la relación entre la distribución y frecuencia de errores frente a diversos factores del entorno, como la complejidad del software, la experiencia del equipo de desarrollo y la reutilización de diseño y de código.

El estudio se realizó sobre un software de unas 90.000 líneas de código, mayoritariamente escrito en FORTRAN, recabando información de los cambios durante las etapas de codificación, prueba, aceptación y mantenimiento a lo largo de 33 meses.

Los autores clasificaron los errores encontrados en 8 categorías diferentes en función del origen de los mismos, incluyendo -entre otras- a los derivados de la mala interpreta-

ción de requerimientos, la implementación incorrecta de uno o de varios módulos, y los derivados de la corrección de un error. En base a esa clasificación muestran que los diferentes tipos de errores se asocian con esfuerzos de corrección muy diversos, destacando que requieren mayor esfuerzo los problemas ocasionados en la comprensión incorrecta de especificaciones o de especificaciones equivocadas.

Basili y Perricone también elaboraron una tipología asbtracta basada en el papel que juega cada error respecto del módulo en que se encuentra. Así, distinguen errores de inicialización, de estructura de control, de interfaz, de datos y de cómputo.

Los autores también analizaron la posible vinculación entre el tamaño de los módulos y la cantidad de errores por cada 1000 líneas de código, y la complejidad ciclomática respecto del tamaño de los módulos.

Uno de los resultados de este trabajo que contradicen los de otros estudios es el de la posible correlación entre tamaño y cantidad de errores. Basili y Perricone encontraron que los módulos de menor tamaño tienen una densidad de errores menor.

En 1993, Li y Henry[50] presentaron el análisis de diversas métricas de la orientación a objetos respecto de la mantenibilidad. Estudiaron dos sistemas comerciales en relación con las métricas de Chidamber y Kemerer y otras sugeridas por los propios autores.

Los autores consideraron las siguientes métricas: Profundidad del árbol de herencia (DIT), Número de Hijos (NOC), Acoplamiento del paso de mensajes (MPC), Respuestas por clase (RFC), Falta de Cohesión en los métodos (LCOM), acoplamiento en la abstracción de datos (DAC), Complejidad ponderada de los métodos (WMC), número de métodos (NOM), cantidad de punto y coma (SIZE1) y cantidad de atributos y métodos (SIZE2).

Li y Henry concluyen que, para los casos en estudio, el conjunto de métricas consideradas constituyen un buen predictor del cambio en el software. En un segundo análisis observan que una gran parte del esfuerzo de mantenimiento puede predecirse a partir de ese solo par. Finalmente, comprueban que el par de métricas de tamaño no pueden considerarse como predictores tan buenos como el conjunto.

En 1996 Basili, Briand y Melo[7] estudiaron las métricas CK en relación con la proba-

bilidad de fallas de una clase. Para ello, analizaron mediante regresión logística los datos obtenido de una serie de desarrollos realizados por estudiantes, observando los valores de las métricas y la presencia o ausencia de fallos detectados en cada clase.

Las clases, por su parte, fueron analizadas según se tratara de desarrollos desde cero, vinculados a una librería provista para el desarrollo o vinculada a la manipulación de bases de datos.

Los resultados del estudio sostienen una relación fuerte entre las métricas DIT, NOC y RFC con la presencia de fallas en las clases; la medida WMC, en tanto, muestra una relación importante. En cambio, CBO y LCOM aparecen con una vinculación no significativa con la presencia o ausencia de fallas en las clases.

Recientemente Riaz, Mendes y Tempero presentaron una revisión sistemática[69] sobre métricas y predicción de la mantenibilidad.

Para ello, exploraron motores de búsqueda y bases de datos en línea, revistas electrónicas, y otros recursos académicos. El primer criterio para incluir en esa exploración fue que los trabajos contaran con revisión de pares.

A continuación, los autores definen criterios de evaluación de la calidad de los papers y la pertinencia respecto del tema analizado.

En definitiva, seleccionaron un conjunto de 15 trabajos sobre los cuales analizaron las técnicas de predicción y el tipo de mantenimiento al que se vinculaban. El trabajo se centró en observar las técnicas de validación utilizadas, las variables que se analizaron y los resultados obtenidos.

Subramanyan y Krishnan[82] se concentraron en analizar WMC (Métodos Ponderados por Clase), CBO (Acoplamiento Entre Objetos) y DIT (Profundidad del Árbol de Herencia). En la primera parte del trabajo revisan los estudios de diversos investigadores, destacando que en muchos de ellos no se toma en cuenta el tamaño como una variable independiente que afecta en mayor medida que las métricas que se consideran.

Olague y otros[60] estudiaron el desempeño de los conjuntos de métricas propuestos por Chidamber y Kemerer y las MOOD entre otras. Estos autores analizaron diferentes

bilidad de fallas de una clase. Para ello, analizaron mediante regresión logística los datos obtenido de una serie de desarrollos realizados por estudiantes, observando los valores de las métricas y la presencia o ausencia de fallos detectados en cada clase.

Las clases, por su parte, fueron analizadas según se tratara de desarrollos desde cero, vinculados a una librería provista para el desarrollo o vinculada a la manipulación de bases de datos.

Los resultados del estudio sostienen una relación fuerte entre las métricas DIT, NOC y RFC con la presencia de fallas en las clases; la medida WMC, en tanto, muestra una relación importante. En cambio, CBO y LCOM aparecen con una vinculación no significativa con la presencia o ausencia de fallas en las clases.

Recientemente Riaz, Mendes y Tempero presentaron una revisión sistemática[69] sobre métricas y predicción de la mantenibilidad.

Para ello, exploraron motores de búsqueda y bases de datos en línea, revistas electrónicas, y otros recursos académicos. El primer criterio para incluir en esa exploración fue que los trabajos contaran con revisión de pares.

A continuación, los autores definen criterios de evaluación de la calidad de los papers y la pertinencia respecto del tema analizado.

En definitiva, seleccionaron un conjunto de 15 trabajos sobre los cuales analizaron las técnicas de predicción y el tipo de mantenimiento al que se vinculaban. El trabajo se centró en observar las técnicas de validación utilizadas, las variables que se analizaron y los resultados obtenidos.

Subramanyan y Krishnan[82] se concentraron en analizar WMC (Métodos Ponderados por Clase), CBO (Acoplamiento Entre Objetos) y DIT (Profundidad del Árbol de Herencia). En la primera parte del trabajo revisan los estudios de diversos investigadores, destacando que en muchos de ellos no se toma en cuenta el tamaño como una variable independiente que afecta en mayor medida que las métricas que se consideran.

Olague y otros[60] estudiaron el desempeño de los conjuntos de métricas propuestos por Chidamber y Kemerer y las MOOD entre otras. Estos autores analizaron diferentes

versiones del software Rhino (un producto F/OSS) con miras a establecer relaciones entre los valores de las métricas y la tendencia a presentar defectos. Para el caso estudiado, observaron que las métricas CK y QMOOD se presentan como mejores predictores.

Selvarani y otros[74] realizaron en 2009 un estudio sobre 20 aplicaciones comerciales de tamaño pequeño a mediano. Basándose en resultados de diversas fuentes, proponen una fórmula polinómica que combina diferentes métricas con el fin de calcular un "índice de propensión a defectos" de carácter predictivo.

El trabajo en cuestión analiza la influencia de las métricas DIT, RFC y WMC con las que elabora el índice mencionado.

Los datos obtenidos por el equipo de Selvarani sugieren valores umbrales para estas métricas que constituirían advertencias sobre la posible propensión a defectos de las clases que devuelven esos valores.

Briand, Wüst, Daly y Porter[11] realizaron un experimento sobre programas desarrollados por estudiantes simulando una situación de producción de software comercial. Los autores exploraron las vinculaciones entre las fallas encontradas en el software y un conjunto de métricas a nivel de clase referidas al acoplamiento, la cohesión y la herencia.

Los autores estudiaron la distribución y la varianza de las diferentes métricas, atendiendo a los valores atípicos que se registren tanto para una métrica en particular como para una combinación entre ellas.

El total de métricas consideradas fue de 49: 28 referidas al acoplamiento, 10 a la cohesión y 11 a la herencia.

El trabajo fue de tipo experimental, analizando el desarrollo de sistemas medianos a cargo de diversos grupos de estudiantes siguiendo un método secuencial basado en el modelo de Cascada.

A continuación, construyeron una serie de modelos de predicción que fueron contrastados con los datos reales.

En el análisis univariante, que vinculaba cada métrica con la información de fallos registrados, observaron una correlación *fuerte entre muchas medidas de cohesión y acoplamiento y la probabilidad de encontrar fallos.*

Es interesante señalar que de las 49 métricas analizadas se desprende que muchas de ellas se comportan de manera similar; los autores observan que el número de dimensiones realmente capturadas por las medidas es mucho menor que la cantidad de métricas, revelando las similitudes entre las hipótesis detrás de esas métricas y los atributos que intentan reflejar. El análisis multivariante sugiere que se pueden elaborar modelos con capacidad predictiva usando algunas de las métricas de acoplamiento y herencia.

Gyimothy[28] y otros estudiaron el desempeño de las métricas CK sobre un producto F/OSS de gran tamaño (Mozilla), desarrollado en C++. Para ello, relacionaron las fallas reportadas en el sistema de seguimiento del proyecto correspondientes a cada versión liberada del programa y el valor de las métricas CK para las diferentes clases que integran Mozilla. En ese trabajo agregan una métrica más: el tamaño de la clase medido en líneas de código; además, diferencian LCOM (que no permite valores negativos) de LCOMN (permitiendo valores negativos).

Los autores comparan la distribución de los valores obtenidos para cada métrica con los reportados oportunamente por Basili.

### 2.6.3 Limitaciones de la validez de los trabajos empíricos

Los trabajos mencionados aquí de ningún modo cuentan con acuerdo general en la comunidad de investigadores en Ingeniería de Software.

El Emam y otros[17] señalan el efecto distorsivo del tamaño sobre los estudios de validación de métricas como los que resumimos anteriormente; en base a una serie de experimentos, afirman que si se toma en cuenta el tamaño de las clases, ninguna de las métricas CK mantiene una incidencia estadísticamente significativa sobre la propensión a fallas. Esta objeción de El Emam, sin embargo, tampoco estuvo exenta de críticas metodológicas: Evanco [20] observa que el análisis mencionado no invalida la relación entre métricas CK y la propensión a defectos, ya que lo se revela como incorrecto sería intentar predecir tal propensión usando simultáneamente el tamaño y las métricas en

cuestión.

## *Umbrales y Valores atípicos*

Pese a la heterogeneidad que observamos, sería útil explorar la posibilidad de determinar umbrales, valores de las métricas que -al menos- llamen la atención sobre posibles debilidades en el diseño o en el código.

Shatnawi[75] estudió las métricas CK en relación con las fallas en el software, buscando determinar umbrales, es decir, valores significativos que sugieran límites para las diferentes métricas.

En primer lugar, realizó un relevamiento de trabajos que buscaron detectar empíricamente relaciones entre los valores de las métricas y la propensión o tendencia a tener fallas. Con esta información elabora un cuadro que refleja que las métricas WMC, CBO y RFC son las que se desempeñaron mejor como indicadores de potenciales fallas en la mayoría de las experiencias.

Shatnawi analiza el código del popular entorno Eclipse en su versión 2.0 para determinar valores de las métricas por encima de las cuales la probabilidad de fallas sería mayor que una cifra determinada. Para definir los valores utilizó el método de Bender.

Mediante análisis univariante, determinó que las métricas mencionadas más arriba constituyen las de mayor significatividad como predictores de fallas.

A continuación, define los valores que supondrían umbrales para esas tres métricas bajo diferentes exigencias de confiabilidad.

Es interesante notar que las cifras obtenidas para una probabilidad estimada de fallas menor a 0.05, los umbrales se ubican en números fuera del rango de la distribución.

El autor evalúa también la efectividad de los umbrales para la siguiente versión de Eclipse, así como para otros productos F/OSS como Mozilla y Rhino.

Shatnawi considera que los umbrales resultan de utilidad, pero advierte sobre los riesgos de generalizarlos, y asegura que es necesario realizar otros experimentos y estudios empíricos que brinden un conocimiento más profundo.

Negro y Giandini[59] realizaron un estudio basado en las hipótesis de Hatton respecto de los posibles umbrales o valores límites que sugerirían una complejidad cognitiva importante. Estos autores analizaron dos aplicaciones desarrolladas en Java, evaluando las métricas de CK. Como resultado, afirman que los datos no permiten sostener la existencia de tales umbrales y que, en caso de que la relación con la tendencia a fallos exista, se trataría de una relación continua que no permitiría fijar valores límites.

El trabajo de Negro y Giandini constituye una advertencia de gran importancia, que debe tenerse en cuenta a fin de no sobrevalorar las posibilidades de evaluación mediante métricas. No obstante, cabe considerar otras experiencias que habilitan a definir umbrales, algunas de las cuales comprenden millones de líneas de código. Un trabajo anterior de Benlarbi y otros[9] llega a conclusiones similares.

Negro y Giandini observan que la posible continuidad en la relación entre métricas CK y la propensión a fallas -y, por lo tanto, la inexistencia de umbrales- no significa que se puedan adoptar valores arbitrarios que orienten la tarea de gestión.

Lanza y Marinescu[47], en tanto, sostienen la necesidad de definir puntos de referencia que permitan juzgar si un valor es demasiado alto o demasiado bajo; en ese sentido proponen también la definición de umbrales.

En su libro Object-Oriented Metrics in Practice[47], presentan una serie de posibles umbrales para las métricas que utilizan, basadas en el estudio estadístico 45 proyectos en Java y 37 en C++. De los resultados se desprende que los umbrales establecidos estadísticamente dependen del lenguaje de programación en cuestión.

### *Métricas y "Bad Smells"*

Las métricas de software no sólo pueden usarse como indicadores de un atributo externo; también pueden servir para detectar ciertos problemas de diseño que la experiencia considera como origen de diversas dificultades, aunque no causen un problema concreto en un desarrollo particular.

En un libro que alcanzó gran influencia, Fowler y Beck[24] acuñaron la expresión "malos olores" del software para referirse a ciertas estructuras que a veces se encuen-

tran en el código fuente y que sugieren (y a veces gritan, según los autores) la necesidad de refactorización.

Estos autores definen con nombres pintorescos 22 tipos de estos "malos olores", entre los que se destacan la "duplicación de código", el "método largo", la "intimidad inapropiada" (referida al uso por parte de una clase de métodos o atributos de otra que deberían estar encapsulados), "clase perezosa" (con poca o nula responsabilidad), "envidia de funcionalidad" (un método que parece más interesado en otra clase que aquella a la que pertenece) y "clase grande".

Fowler y Beck consideran que ningún conjunto de métricas puede rivalizar con la intuición humana para detectar estos "malos olores"; no obstante, mucho trabajo se realizó tratando de precisar las definiciones propuestas por ellos[88], estudiar la manera en que las métricas podían relevar la presencia de estas estructuras y clasificarlas según características comunes que facilitaran su detección y corrección[53].

Estructuras de este tipo también suelen llamarse "antipatrones"(empleándose la expresión "malos olores" para los síntomas), con lo que se destaca que constituyen soluciones (o intentos de solución) comunes que provocan o pueden provocar nuevos problemas.

Lanza y Marinescu[47] plantean diversas heurísticas para detectar algunos de estos problemas; hay una serie de herramientas (Moose, iPlasma, inFusion, entre otras) que permiten inspeccionar el código Java o C++ en busca de estructuras indeseables.

En su tesis de Maestría, Mäntylä[53] también explora las posibilidades de detectar algunos "malos olores" mediante métricas.

Mäntyla critica a Fowler y Beck que expongan la lista de "bad smells" sin organización ni categorización de ningún tipo; además, cuestiona la presunción de esos autores de que ningún conjunto de métricas podría reemplazar a la experiencia.

En el desarrollo de su tesis, Mäntyla propone organizar los "bad smells" en categorías y muestra cómo algunas métricas combinadas pueden indicar la presencia de alguno de estos antipatrones. También califica a los "bad smells" según la facilidad y la precisión con la que pueden detectarse mediante valores de métricas; por ejemplo, la llamada "envidia de funcionalidad" es fácil de determinar mediante métricas de acoplamiento,

mientras que un Intermediario (Middle Man, una clase que no hace nada por sí misma sino que delega todo en otras) es especialmente difícil de inferir mediante valores registrados de métricas.

## *Caracterización y Evaluación*

Lanza y Marinescu[47] proponen dos técnicas para caracterizar el diseño de una aplicación: La pirámide de panorama general y la visualización de métricas y diseño.

La pirámide de panorama general se divide en tres partes que reflejan, según los autores, los tres aspectos más relevantes del diseño: tamaño y complejidad, herencia y acoplamiento. La disposición de la información en la pirámide es la que se ve en la ilustración IV.

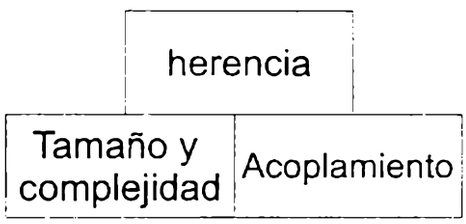


*Ilustración IV: Pirámide de Panorama General*

Los autores proponen un conjunto de métricas para cada una de estos aspectos, organizándolas desde lo general a lo particular respecto de la altura de la pirámide.

Respecto del tamaño y la complejidad, utilizan las siguientes métricas: CYCLO (Complejidad ciclomática), LOC (líneas de código), NOM (Número de métodos u operaciones), NOC (número de clases) y NOP (números de paquetes). Estas métricas se asocian de manera descendente para reflejar la estructuración de alto nivel (cantidad de clases por paqueteNOC/NOP), estructuración de clase (NOM/NOC) , estructuración de operación (LOC/NOM) y complejidad de operación intrínseca (CYCLO/LOC).

De la misma manera organiza las medidas referidas al acoplamiento, utilizan FOUT (fan-out, cantidad de clases llamadas) y CALLS (total de llamadas a métodos u operaciones diferentes).

Los autores usan la expresión "métodos y operaciones", para referirse no sólo a los servicios que ofrecen los objetos sino también a las funciones definidas por el usuario y de

carácter global.

Con esas medidas evalúan la intensidad de acoplamiento (CALLS/NOM) y la "dispersión del acoplamiento" (FOUT/CALLS).

La estructura de la jerarquía de clases se representa con ANDC (Número promedio de clases derivadas) y AHH (Altura Promedio de la jerarquía).

La "visualización de métricas" se refiere al mapeo de los valores de las métricas a características visuales que permiten apreciar fácilmente las diferencias. Los autores en cuestión adoptan una "vista polimétrica", una representación de las entidades que constituyen el diseño y las relaciones entre ellas, enriquecida con la visualización de las métricas. Por ejemplo, la cantidad de métodos y atributos se expresan mediante el ancho y el alto, brindando una idea visual del tamaño de una clase.

La evaluación del diseño, para Lanza y Marinescu, implica considerar la "armonía" del diseño, que incluye armonía de identidad -referida a que cada elemento justifique su existencia y desempeñe el rol que se espera de él-, armonía de colaboración -entendiendo que cada entidad debe colaborar con las demás para cumplir con su propósito y que que el uso de los servicios ajenos no puede ser muy escaso ni excesivo- y armonía de clasificación -que vincula las anteriores en el contexto de la herencia.

Lanza y Marinescu también muestran una serie de métodos para detectar faltas a esas armonías (antipatrones), como la "clase Dios" (similar a clase grande de Fowler y Beck), "método cerebro" (una clase que ha ido adquiriendo más y más funcionalidad hasta volverse inmanejable) y "envidia de funcionalidad" entre otras.

## 2.7 Evaluación de productos mediante métricas OO

Diversos trabajos intentan vincular la información que surge de diferentes métricas con las debilidades de diseño o con la propensión a fallos de determinadas secciones del código.

Los resultados obtenidos no son coincidentes; sin embargo, algunas métricas CK (las más estudiadas) aparecen estadísticamente vinculadas a la tendencia a presentar fallas.

Otras métricas también muestran desempeños aceptables en diversos estudios, aunque en todos los casos se trata de muestras limitadas, ya sea de desarrollos ad-hoc, productos F/OSS o -con menor frecuencia- desarrollos comerciales cerrados.

De la revisión de los trabajos presentados surge que:

- No hay un consenso general respecto de la validez de un conjunto de métricas OO en el sentido de que representen atributos específicos del diseño o del código
- No existe consenso respecto de la relación entre métricas OO y la tendencia a producirse fallas en un producto o en una parte de un producto (una clase, un paquete)
- No obstante, hay evidencia empírica que sugiere que tales relaciones existen, si bien no han podido establecerse los mecanismos por los que operarían
- No hay sustento en las diferentes investigaciones estudiadas para determinar umbrales de validez general; no obstante, pueden ser de utilidad práctica como señales a tener en cuenta por parte de los de proyectos
- Las métricas pueden alertar también sobre posibles debilidades o errores en el diseño, aunque no las determinen de manera categórica

A los fines del presente trabajo, las métricas OO tienen la ventaja de brindar información acerca del diseño de una aplicación partiendo del código fuente.

Así como Fenton[22] desalienta toda expectativa de que un solo número pueda representar la complejidad del programa, una métrica o un pequeño conjunto de métricas tampoco puede dar certezas sobre la mantenibilidad o evolutividad de un producto.

Conscientes de esa limitación, podemos utilizar diversas mediciones como un indicativo a tener en cuenta a la hora de evaluar un producto.

# 3.- Mantenibilidad, evolutividad y F/OSS

## 3.1 F/OSS y Calidad de Software

El F/OSS se desarrolla siguiendo metodologías claramente diferentes a las maneras tradicionales. Eso no significa que exista una forma única ni siquiera un puñado de metodologías que describan la heterogeneidad que caracteriza al universo F/OSS.

Este grupo de productos comparte un conjunto de características a partir de las licencias de distribución aceptadas en las principales corrientes de los movimientos de Software Libre y Código Abierto.

Suele asociarse el desarrollo F/OSS con el modelo esbozado por Eric Raymond en su famoso ensayo "La Catedral y el Bazar"[68], donde el autor compara el desarrollo cerrado o privativo con la construcción planificada y ordenada de una catedral, frente a la heterogeneidad y caos aparente en el que se desenvuelven los proyectos que nos ocupan.

Sin embargo, el desarrollo en el ámbito F/OSS tampoco es uniforme; si bien el modelo esbozado por Raymond es representativo de algunos productos (el núcleo de Linux en particular), existen muchas maneras de organizar el desarrollo, con diferencias significativas en cuanto a la forma de gestión y las características de la participación de desarrolladores y usuarios.

En su tesis doctoral[70], Gregorio Robles advierte que F/OSS y el modelo de desarrollo del bazar no son sinónimos. No obstante, destaca que buena parte de los desarrollos exitosos se relacionan con ese modelo.

Algunos productos F/OSS alcanzaron una aceptación enorme e incluso posiciones de liderazgo en diversos dominios (Apache, Linux y BIND son ejemplos ampliamente mencionados)

Dentro de la heterogeneidad, todos los productos F/OSS comparten:

- disponibilidad pública del código fuente
- una licencia que permite su modificación y redistribución

Además de estas características, en muchos casos la información del desarrollo está disponible públicamente, ya sea en repositorios de productos F/OSS (como SourceForge,

Savannah y JavaSource entre muchos otros) o en sus sitios Web.

## 3.2 Relevamiento de información de proyectos F/OSS

La información disponible públicamente en muchos proyectos F/OSS abre, en principio, posibilidades de evaluación amplias que no están disponibles en el universo de desarrollo tradicional o "cerrado".

En el trabajo de Robles[70] mencionado anteriormente se esboza una clasificación de las fuentes de información de proyectos F/OSS

- Datos obtenidos del propio producto, fundamentalmente el código fuente; a veces incluye documentación, materiales gráficos y multimedia y archivos de diseño de GUI entre otros.
- Datos obtenidos de las herramientas usadas en el desarrollo, especialmente de los sistemas de gestión de configuraciones (generalmente CVS y Subversion)
- Datos obtenidos de las herramientas de comunicación, incluyendo foros y sistemas de seguimientos de errores.

El primer tipo de fuente está siempre disponible en estos productos, si bien la completitud, claridad y pertinencia pueden ser muy variables.

### 3.2.1 Dificultades para el relevamiento de información

Pese a la disponibilidad pública, la calidad de la información puede no ser adecuada para evaluar las características de un producto F/OSS.

Actualmente, el mayor repositorio de F/OSS es SourceForge. Al momento de escribir este trabajo, el sitio informa que alberga más de 250.000 proyectos.

Cada proyecto en ese repositorio dispone de herramientas de gestión de código (Subversion), seguimiento de artefactos (entre los que puede incluirse el seguimiento de errores), herramientas de comunicación diversas, registro de actividad de desarrolladores, informes de usuarios, documentación, etc.

La utilización de estos recursos es también muy heterogénea. El "bug-tracker"de muchos de ellos presenta muchas veces información incompleta o desactualizada[66];

en algunas ocasiones, la administración de errores y de nuevas funcionalidades se mantiene fuera del ámbito de SourceForge.

La forma en que se utilizan las herramientas también puede variar significativamente entre proyectos.

Estas limitaciones deben tenerse en cuenta a la hora de evaluar un proyecto F/OSS[36]; la ausencia de la información buscada no significa necesariamente una omisión en el proyecto, ya que puede ocurrir que los datos relacionados no sean públicos o que se gestionen mediante herramientas diferentes a las provistas en los repositorios. No obstante, y con las salvaguardas pertinentes, se ha incrementado enormemente la actividad de investigación sobre los repositorios F/OSS, generándose además abundante información disponible públicamente[35].

## 3.3 Investigación cuantitativa

La posibilidad de acceso al código fuente y a diferentes aspectos del proceso de desarrollo favoreció una creciente producción de investigación sobre diferentes aspectos relacionados no solo con el F/OSS sino también con diferentes tópicos de la ingeniería de software.

Como mencionamos más arriba, el uso de los repositorios presenta limitaciones que deben atenderse. Sin embargo, y con diferentes niveles de precaución, se han producido muchísimos trabajos basados en la información obtenida directamente de los repositorios.

Michlmayr y Senyard[56] señalaban en 2006 que, pese a la abundancia de datos disponible, existían pocas investigaciones que la estudien. En su trabajo, los autores mencionados realizaron un análisis estadístico profundo del proceso de informe y corrección de errores en el sistema Debian, desde 1994 hasta fines de 2003 observando alrededor de 200.000 reportes.

Kim y Whitehead[45] trabajaron sobre información referida a la mantenibilidad; consideran que para evaluar esa característica, el tiempo de resolución de un error es una medida poco utilizada en comparación con la cantidad de errores. Los autores analizan

las estadísticas relacionadas para los proyectos F/OSS ArgoUML y PostgresSQL durante un período de tiempo considerable (1 año en el primer caso, 4 años en el segundo) concluyendo que la resolución de la mitad de los errores requiere -en los casos estudiados- entre 100 y 300 días.

Asundi y Jayand[4] estudiaron los diferentes roles de los desarrolladores en el proceso de corrección de errores; para ello analizaron las listas de correo de cuatro proyectos de características diferentes en cuanto a tamaño, destinatarios, y número de desarrolladores; en el caso del proyecto Gaim (un popular cliente libre de mensajería en Internet), recopilaron manualmente la información de la interfaz Web del bug tracker del proyecto en SourceForge.

Como mencionamos anteriormente, la recopilación de datos de SourceForge u otros repositorios no está exenta de peligros y limitaciones, según advierten Howison y Crowston[36]; estos autores analizaron 140 proyectos alojados en el repositorio mencionado, los cuales contaban 7 o más desarrolladores y 100 o más errores reportados en el el sistema de seguimiento. A partir de esa experiencia, exponen una serie de previsiones necesarias para la extracción de información mencionando, entre otras dificultades, las ocasionadas por datos truncos o erróneos, la inconsistencia en la utilización de los recursos de desarrollo en algunos proyectos y las complicaciones surgidas de la migración de la información de un proyecto desde otros sistemas.

Las pretensiones de extender conclusiones basadas en esa información a otros ámbitos, particularmente a proyectos de código cerrado, también encuentra importantes restricciones, según revelan Yu, Schach y Chen[87] en un trabajo sobre la medición de la mantenibilidad en proyectos de código abierto. Esos autores advierten que las condiciones en que se producen los datos en un proyecto de código abierto no son asimilables a los procesos de un desarrollo cerrado. Respecto de la información obtenida del seguimiento de errores, cuestionan la falta de vinculación con versiones concretas del software, el desfasaje entre la antigüedad de los datos disponibles y la fecha en que se produjeron los primeros lanzamientos y la ausencia de elementos que permitan estimar el esfuerzo que requiere la solución de errores.

Las dificultades a que aludimos aquí no son exclusivas del ámbito F/OSS; Aranda y Venolia[3] observaron omisiones y datos incorrectos analizando repositorios de seguimientos de bugs en proyectos de Microsoft. Los autores sostienen en ese trabajo que tales deficiencias no son exclusivas de la empresa mencionada.

Sowe[78] y otros resumen una serie de inconvenientes a los que debe enfrentarse un investigador que apele a los datos disponibles en un repositorio, como enumeramos a continuación:

- Falta de estandarización en cuanto a las denominaciones de los diferentes elementos que albergan los repositorios, dificultando la tarea de integrar la información obtenida de diferentes orígenes.
- El acceso a los datos a través de la interfaz Web pública puede chocar con modificaciones en esa interfaz, que obliguen a rever las herramientas con las que se colectan los datos.
- Es posible que parte de la información tenga carácter confidencial en el momento en que los investigadores necesiten acceder a ella, en particular lo que se refiere a los mensajes producidos por particulares.
- Solicitar información a los responsables de un proyecto particular o los administradores del repositorio pueden demandar de mucho tiempo para obtener respuestas.

Una alternativa más amigable para los investigadores es utilizar información de "repositorios de repositorios", donde se almacenan los datos de una gran cantidad de proyectos albergados en diferentes repositorios. Entre ellos, se destaca flossmole[35].

Estas fuentes pueden permitir al investigador o al encargado de seleccionar F/OSS disponer de vistas muy diferentes sobre cada proyecto que considere; sin embargo, no todos los proyectos están alojados en repositorios ni todos ofrecen acceso a la administración de configuraciones, seguimiento de errores, listas de correo, etc.

En el presente trabajo, entonces, nos centramos en las posibilidades que ofrece el estudio del código fuente.

## 3.4 Evaluación de F/OSS a partir del código

Varios referentes destacados de los movimientos de software libre y de código abierto sostienen que la metodología de desarrollo de F/OSS tiende a producir código de mayor calidad que de la forma tradicional; esta afirmación constituyó un aliciente para la investigación tendiente a evaluar la calidad del código.

Existen diversos trabajos[57][86] basados en "fuzzy testing" (una técnica basada en enviar entradas aleatorias a una aplicación) que sostienen las afirmaciones mencionadas, desde el punto de vista externo.

En el año 2002 se publicó un trabajo de Stamelos y otros[80] en el que analizan la calidad del código de 100 programas escritos en lenguaje C e incluidos en la distribución GNU/Linux alemana SUSe. Utilizaron para ello la herramienta Logiscope de IBM, que considera un conjunto de 10 métricas estructurales para la evaluación.

La herramienta usada por estos investigadores asocia los valores de las métricas de cada componente a diferentes niveles de facilidad de prueba, simplicidad, legibilidad y la condición de autodescriptivo, cailificándolos con la siguiente escala: aceptar, comentar, inspeccionar, probar, reescribir.

La evaluación de Stamelos y su grupo concluye que el código tiene mayor calidad que la esperada por los críticos del desarrollo de F/OSS; no obstante, de acuerdo a los estándares propuestos por la propia herramienta, la calidad de esas aplicaciones sería menor a la requerida en el ámbito industrial.

Samoladas y otros[71] usaron la misma herramienta para analizar el ERP/CRM Compiere, concluyendo que la calidad del código se podía considerar "buena", pero advirtiendo que una evaluación completa desde el punto de vista de la calidad requeriría un estudio del sistema en operación en situaciones reales. A nuestros fines, resulta relevante destacar que el estudio arrojó resultados favorables en cuanto a la mantenibilidad del producto, de acuerdo con los parámetros de la herramienta utilizada.

En su libro Code Quality: The Open Source Perspective[79], Diomidis Spinellis considera que la medición de la mantenibilidad permite evaluar la lucha contra la "entropía" del código a medida que el software evoluciona, comparar entre diferentes sistemas el

desempeño en cuanto a mantenibilidad y evaluar las partes de un sistema que pueden ser propensos a errores o difíciles de mantener.

Spinellis considera el Indice de Mantenibilidad (MI) y las métricas CK; no obstante, señala que no se puede adoptar el MI sin ejercitar el juicio propio. El autor sugiere utilizar las métricas para detectar puntos problemáticos, no como un canon para implementar sistemas mantenibles.

Spinellis ilustra sus ideas mediante el estudio de dos productos F/OSS ampliamente extendidos: HSQLDB y Eclipse.

En los últimos años ha aumentado el número de investigaciones que ponen en juego diversas métricas aplicadas a productos F/OSS. Anteriormente mencionamos el estudio de Gyimothy y otros[28] donde analizan las métricas CK sobre Mozilla.

Murgia y otros[58] analizaron dos F/OSS muy difundidos, los IDEs Eclipse y NetBeans. Evaluaron el código estático mediante las métricas CK y estudiaron el acoplamiento mediante gráficos de clase que representan las clases y las relaciones entre ellas (incluyendo herencia y colaboración).

El libro de Lanza y Marinescu[47] citado anteriormente en este trabajo ilustra su propuesta de caracterizar y evaluar el diseño de un software mediante el estudio del F/OSS ArgoUML, una herramienta de diseño UML también ampliamente difundido.

Algunos autores ya mencionados, como Samoladas y Spinellis, elaboraron una propuesta para la evaluación de F/OSS denominada Qualification and Selection of Open Source Software (QSOS)[72]. Esta metodología de selección contempla la evaluación del código mediante los valores obtenidos de diversas métricas asociadas a loa atributos de calidad estipulados en ISO 9126. Cabe señalar que en el modelo propuesto por estos autores algunas métricas se vinculan con más de un atributo.

### *3.4.1 Análisis de la evolución de Sweet Home 3D*

Para poner en juego las propuestas que relevamos hasta aquí, analizamos la aplicación Sweet Home 3D, escrita en Java y albergada en SourceForge.

Utilizamos las herramientas iPlasma e InFusion para observar la evolución de las carac-

terísticas de la aplicación a lo largo de los diferentes lanzamientos.

En la ilustración V observamos la evolución de las proporciones propuestas por Lanza y Marinescu en las diferentes versiones del programa.

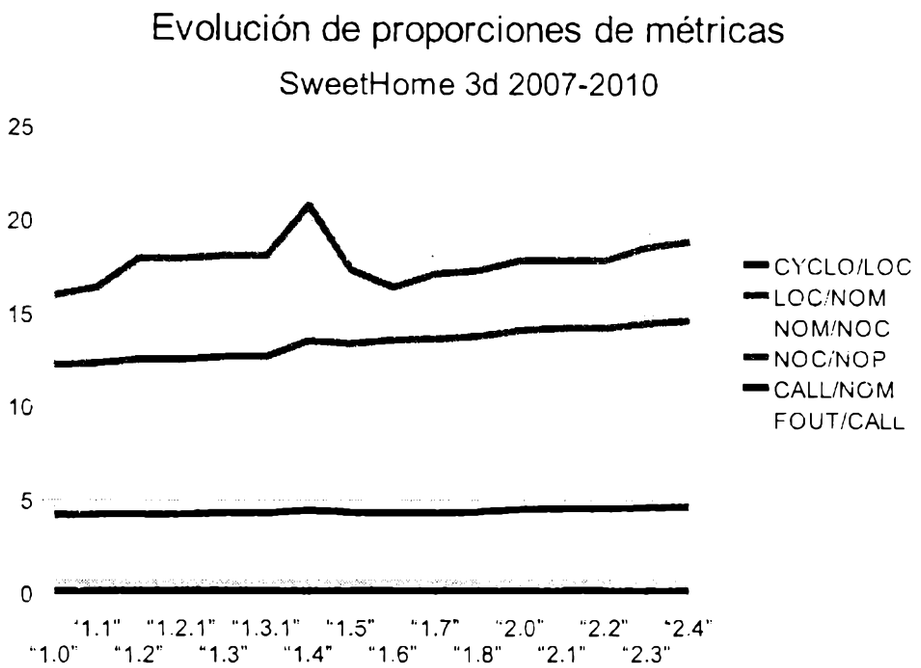


*Ilustración V: Información relevada con la herramienta iPlasma*

Como puede apreciarse, la variación de las proporciones estudiadas es pequeña, tendiendo a permanecer constantes la mayoría de ellas, con un leve tendencia creciente de la estructura de operación, la estructuración de clases y la estructura de alto nivel.

En principio, estos resultados sugieren que la evolución de la aplicación no ha producido alteraciones sustanciales en la estructura, ni se observan indicios que permitan inferir una mayor complejidad de la aplicación.

Otro aspecto que se desprende de la información es que el crecimiento del tamaño de la aplicación en el tiempo sigue una tendencia aproximadamente lineal, a diferencia de lo observado por diversos investigadores para el caso del núcleo de Linux[26].

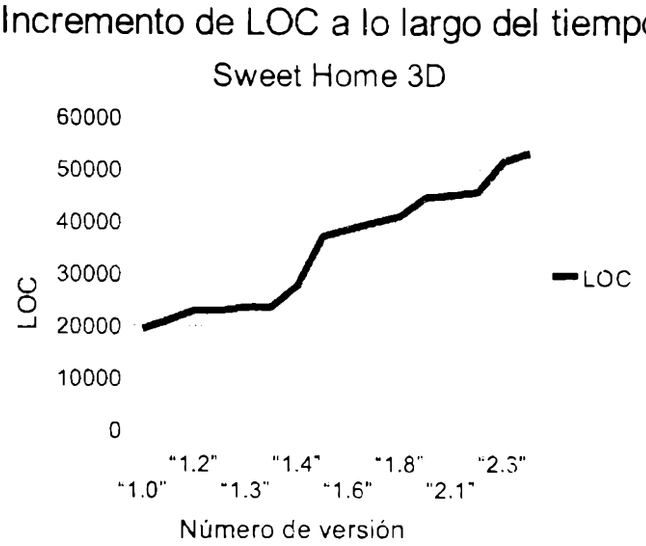


*Ilustración VI: LOC medidos con iPlasma*

Los datos relevados también indican que entre las versiones 1.3.1, (17 de agosto de 2008), y la 1.4 (7 de octubre de 2008) se produjo una modificación importante, ya que la relación entre el número de paquetes y la cantidad de clases se redujo, luego de un incremento atípico con relación a las variaciones entre las distintas versiones.

La herramienta inFusion también nos permite visualizar siete posibles casos de "Clase Dios" en el programa: las clases Home, PlanController, HomeController, PlanComponent, ImportedFurnitureWizardController, HomePane y HomeComponent3D. Analizando cada clase, la herramienta nos llama la atención sobre posibles ocurrencias de antipatrones en los métodos de estas clases, como "Envidia de Funcionalidad" o "Duplicaciones Significativas".

Dentro del período analizado, no se observa que la comunidad de desarrollo de SweetHome 3D haya tomado medidas para eliminar esos posibles antipatrones.

# Conclusiones

Los términos evolución y mantenimiento se usan a menudo como sinónimos. Sin embargo, se asocian o aluden a aspectos diferentes: la noción de mantenimiento se vincula con el término equivalente en otras ingenierías en las que refiere principalmente a la corrección de fallas; la palabra evolución, en cambio, se emparenta con conceptos de la biología y enfatiza el cambio a lo largo del tiempo.

La mantenibilidad y la evolutividad designan a la facilidad o dificultad de modificar el software, ya sea para corregir defectos, adecuarlo a nuevas circunstancias o para agregar funcionalidades.

Hemos revisado una gran cantidad de propuestas que intentan evaluar la mantenibilidad y la evolutividad a partir del código. Entre ellas, surgen algunas -principalmente en el mundo del desarrollo Orientado a Objetos- que extraen conclusiones sobre el diseño de una aplicación a partir de los valores registrados en diferentes métricas.

Los conceptos de evolución se vinculan con nuestro interés principal, ya que nuestra perspectiva se orienta a la posibilidad de reusar o adaptar software, lo que supone un tipo de modificación sobre el producto original que se asocia a los llamados "mantenimiento perfectivo" y "mantenimiento adaptativo"; estas actividades se asocian a la adecuación del software a nuevos entornos y a la incorporación de nuevas funcionalidades que le permiten al mismo continuar siendo útiles para los usuarios: estas son las características que impulsan la evolución.

Sin dudas, queda mucho trabajo por hacer en la definición de métricas que se ajusten a una teoría de la medición y que al mismo tiempo resulten útiles para la evaluación del software.

Citando a Lanza y Marinescu[47], las métricas no son una panacea sino una herramienta poderosa aunque con límites.

Una medida o un conjunto de medidas tal vez nunca puedan asegurarnos que un programa es fácil o difícil de modificar; sin embargo, pueden orientar nuestra mirada, pueden sacar a la luz problemas en el diseño, que -como mínimo- nos lleven a revisar una parte de una aplicación.

El mantenimiento puede analizarse desde una perspectiva externa o interna, tal como lo

plantean Fenton y Pfleeger[23]; buena parte de las métricas propuestas en la bibliografía sobre métricas apunta a medir la forma en que se realiza el mantenimiento; en ese sentido, la información histórica puede brindar un panorama sobre la mantenibilidad, y como tal puede servir para evaluar su relación entre los atributos internos, sin desconocer que en ese proceso intervienen muchos otros factores ajenos al código e incluso al diseño (como las competencias de las personas encargadas del mantenimiento, la documentación, etc.).

Se han propuesto muchas métricas que intentan vincular características del código con la mantenibilidad. La complejidad ciclomática y las métricas de la ciencia del software de Halstead son ampliamente utilizadas, principalmente para describir programas desarrollados desde una perspectiva estructurada. A pesar de las críticas que se le hicieron, y a que no existe un consenso en cuanto a la validez de la supuesta vinculación de estas medidas con la modificabilidad del código, continúan sirviendo de referencia e incorporándose a otros modelos.

Diversos trabajos exploraron relaciones entre métricas y mantenibilidad[69]; de entre esas investigaciones surgió el Índice de Mantenibilidad, combinando una serie de métricas y calibrando los coeficientes de acuerdo con la experiencia; sin embargo, este índice no contempla las características específicas de la Orientación a Objetos.

Otros trabajos estudiaron la validez empírica de determinadas métricas del diseño Orientado a Objetos como indicadores de mantenibilidad[7][60][6][8][28]; en general, tomaron una métrica o un conjunto de métricas como variables independientes, y algún indicador externo de mantenibilidad como variable dependiente (por ejemplo, la cantidad o la presencia de errores detectados en cada módulo analizado).

Mediante diversos métodos estadísticos se examinó la posible vinculación de las métricas consideradas. Los resultados son dispares, aunque la mayoría de los trabajos sugieren la existencia de vinculaciones estadísticamente relevantes. Al mismo tiempo, algunos trabajos revelan que diversas métricas se asocian a características similares, o presentan variaciones semejantes, de modo que la evaluación con métricas debería considerar sólo un subconjunto.

Las métricas que aparecen con mayor frecuencia en los estudios son las definidas por Chidamber y Kemerer (CK). De ese conjunto, a su vez, WMC, CBO y LCOM parecen presentar mejores desempeños como predictores de mantenibilidad.

Se han desarrollado heurísticas y algoritmos para detectar problemas de diseño o de codificación a partir de métricas; se pueden disponer de indicativos de diversos "malos olores" del código[24] o antipatrones a partir de los valores de diferentes métricas, lo que puede sugerir análisis más profundos sobre un producto F/OSS.

Una consideración esencial, a nuestro entender, es evitar la confusión ocasionada en atribuir relaciones de causa-efecto entre las métricas y la propensión de producir fallos, o a la dificultad o facilidad de modificación. Las advertencias de Fenton[22] en ese sentido deben tenerse en cuenta en cualquier modelo de evaluación de mantenibilidad o evolutividad basado en métricas.

Sin embargo, las métricas pueden darnos un panorama general, llamando la atención sobre características estructurales que podrían potencialmente acarrear dificultades en el mantenimiento.

La disponibilidad de información en el ámbito de los proyectos F/OSS puede permitirnos estudiar la evolución de esos productos desde múltiples perspectivas.

# Bibliografía:

[1] ABRAN, A.; BOURQUE,P.; DUPUIS,R.; MOORE,J.W.; TRIPP, L.L.: *Guide to the Software Engineering Body of Knowledge - SWEBOK - Trial Version*, 2004 version .IEEE Press, 2004.

[2] ABREU, F.B.E.; MELO, W.. Evaluating the Impact of Object-Oriented Design on Software Quality. En *METRICS '96: Proceedings of the 3rd International Symposium on Software Metrics*. 1996.

[3] ARANDA, J.; VENOLIA, G.. The secret life of bugs: Going past the errors and omissions in software repositories. En *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. 2009.

[4] ASUNDI, J.; JAYANT, R.: Patch Review Processes in Open Source Software Development Communities: A Comparative Case Study, *Hawaii International Conference on System Sciences*, 0, 2007.

[5] BANKER, R.; DATAR,S.; ZWEIG, D.. Software complexity and maintainability. En *ICIS '89: Proceedings of the tenth international conference on Information Systems*. 1989.

[6] BARKMANN, H.; LINCKE,R.; LOWE, W.. Quantitative Evaluation of Software Quality Metrics in Open-Source Projects. En *WAINA '09: Proceedings of the 2009 International Conference on Advanced Information Networking and Applications Workshops*. 2009.

[7] BASILI, V.; BRIAND,L.; MELO, W.: A Validation of Object Oriented Design Metrics as Quality Indicators, *IEEE Transactions on Software Engineering*, 22, 1996.

[8] BASILI, V.R.; PERRICONE, B.T.: Software errors and complexity: an empirical investigation0, *Commun. ACM*, 27, 1984.

[9] BENLARBI, S.; EMAM,K.E.; GOEL,N.; RAI, S.N.: Thresholds for Object-Oriented Measures, 2000.

[10] BENNETT, K.H.; RAJLICH, V.T.. Software maintenance and evolution: a roadmap. En *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*. 2000.

[11] BRIAND, L.; WÜST,J.; DALY,J.; PORTER, V.: Exploring the Relationships between Design Measures
and Software Quality in Object-Oriented Systems, 1998.

[12] BROY, M.B.; DEISSENBOECK,F.; PIZKA, M.. Demystifying Maintainability. En *In Proc. 4th Workshop on Software Quality (4-WoSQ)*, . 2006.

[13] CHIDAMBER, S.R.; KEMERER, C.F.: A Metrics Suite for Object Oriented Design, *IEEE Trans. Softw. Eng.*, 20, 1994.

[14] COOK, S.; JI,H.; HARRISON, R.: Software Evolution and Software Evolvability, 2000.

[15] DASKALANTONAKIS, M.K.: A Practical View of Software Measurement and Implementation Experiences Within Motorola, *IEEE Trans. Softw. Eng.*, 18, 1992.

[16] DEPREZ, J.; MONFILS,F.F.; CIOLKOWSKI,M.; SOTO, M.: Defining Software Evolvability from a Free/Open-Source Software, *Software Evolvability, IEEE International Workshop on*, 0, 2007.

[17] EL EMAM, K.; BENLARBI,S.; GOEL,N.; RAI, S.: The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics, *IEEE Trans. Softw. Eng.*, 27, 2001.

[18] EL-EMAM, K.: Object-Oriented Metrics: A Review of Theory and Practice, 2001.

[19] ENDRES, A.; ROMBACH, D.: *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories* , .Addison-Wesley, 2003.

[20] EVANCO, W.M.: Comments on The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics, *IEEE Trans. Softw. Eng.*, 29, 2003.

[21] FAGAN, M.: Design and code inspections to reduce errors in program development, *IBM Systems journal*, 15, 1976.

[22] FENTON, N.; NEIL, M.: Software metrics: successes, failures and new directions, *Journal of Systems and Software*, 47, 1999.

[23] FENTON, N.E.; PFLEEGER, S.L.: *Software Metrics: A Rigorous and Practical Approach*, .PWS Publishing Co., 1998.

[24] FOWLER, M.; BECK, K.: «*Bad Smells in Code*», en *Refactoring: Improving the design of existing code*, ,Addison-Wesley, 1999.

[25] GODFREY, M.. Past, Present and Future of Software Evolution. En *24th International Conference on Software Maintenance*. 2008.

[26] GODFREY, M.; TU, Q.. Growth, evolution, and structural change in open source software. En *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*. 2001.

[27] GRUBB, P.; TAKANG, A.A.: *Software Maintenance: Concepts and Practice*, 2nd .World Scientific, 2003.

[28] GYIMOTHY, T.; FERENC,R.; SIKET, I.: Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction, *IEEE Trans. Softw. Eng.*, 31, 2005.

[29] HALSTEAD, M.H.: *Elements of software science / Maurice H. Halstead*, .Elsevier, New York :, 1977.

[30] HARRISON, R.; COUNSELL,S.; NITHI, R.: An Evaluation of the MOOD Set of Object-Oriented Software Metrics, *IEEE Transactions on Software Engineering*, 24, 1998.

[31] HASHIM, K.; KEY, E.: A Software Maintainability Attributes Model, *Malysian Journal of Computer Science*, 9, 1996.

[32] HENRY, S.; KAFURA, D.: Software Structure Metrics Based on Information Flow, *IEEE Trans. Softw. Eng.*, 7, 1981.

[33] HONGLEI, T.; WEI,S.; YANAN, Z.: The Research on Software Metrics and Software Complexity Metrics, *Computer Science-Technology and Applications, International Forum on*, 1, 2009.

[34] HORDIJK, W.; WIERINGA, R.: Surveying the factors that influence maintainability: research design, *SIGSOFT Softw. Eng. Notes*, 30, 2005.

[35] HOWISON, J.; CONKLIN,M.; CROWSTON, K.: FLOSSmole: A Collaborative Repository for FLOSS Research Data and Analyses, *International Journal of Information Technology and Web Engineering*, 1, 2006.

[36] HOWISON, J.; CROWSTON, K.. The perils and pitfalls of mining SourceForge. En *Proc. of Mining Software Repositories Workshop at the International Conference on Software Engineering (ICSE)*. 2004.

[37] IEEE: IEEE Std 610.12-1990:IEEE Standard Glossary of Software Engineering Terminology, 1990.

[38] IEEE: IEEE Standard for Software Maintenance (1219-1998 ), 1998.

[39] JARZABEK, S.: *Effective Software Maintenance and Evolution. A Reuse-Based Approach*, .Auerbach Publications, 2007.

[40] KAJKO-MATTSSON, M.; CANFORA,G.; CHIOREAN,D.; DEURSEN,A.V.; IHME,T.; LEHMAN,M.M.; REIGER,R.; ENGEL,T.; WERNKE, J.. A Model of Maintainability - Suggestion for Future Research. En *Software Engineering Research and Practice*. 2006.

[41] KAN, S.H.: *Metrics and Models in Software Quality Engineering*, .Addison-Wesley Longman Publishing Co., Inc., 2002.

[42] KEARNEY, J.; SEDLMEYER,R.; THOMPSON,W.; GRAY,M.; ADLER, M.: Software complexity measurement, *Commun. ACM*, 29, 1986.

[43] KEMERER, C.F.; SLAUGHTER, S.: An Empirical Approach to Studying Software Evolution, *IEEE Transactions on Software Engineering*, 25, 1999.

[44] KEMERER, C.F.; SLAUGHTER, S.A.: Determinants of software maintenance profiles: an empirical investigation, *Journal of Software Maintenance*, 9, 1997.

[45] KIM, S.; WHITEHEAD JR, E.. How long did it take to fix bugs?. En *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*. 2006

[46] KOLKHORST, B.; MACINA, A.. Developing Error-Free Software,. En *Fifth International Conference on Testing Computer Software*. 1998.

[47] LANZA, M.; MARINESCU,R.; DUCASSE, S.: *Object-Oriented Metrics in Practice*, .Springer-Verlag New York, Inc., 2005.

[48] LEHMAN, M.. Laws of Software Evolution Revisited. En *EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology*. 1996.

[49] LEHMAN, M.; FERNÁNDEZ-RAMIL, J.C.: «*Software Evolution*», en *Software Evolution and Feedback*, ,, 2006.

[50] LI, W.; HENRY, S.: Object-oriented metrics that predict maintainability, *J. Syst. Softw.*, 23, 1993.

[51] LIENTZ, B.; SWANSON,E.; TOMPKINS, G.: Characteristics of application soft-

ware maintenance, *Commun. ACM*, 21, 1978.

[52] MADHAVJI, N.; RAMIL,F.; PERRY, D.: *Software evolution and feedback: theory and practice*, .Hoboken, NJ: John Wiley \& Sons, 2006.

[53] Mäntylä, M.. Bad Smells in Software - a Taxonomy and an Empirical Study. *Helsinki University ofTechnology*. 2003.

[54] MCCABE, T.. A complexity measure. En *ICSE '76: Proceedings of the 2nd international conference on Software engineering*. 1976.

[55] MENS, T.; DEMEYER, S.: *Software Evolution*, .Springer, 2008.

[56] MICHLMAYR, M.; SENYARD, A.. A Statistical Analysis of Defects in Debian and Strategies for Improving Quality in Free Software Projects. En *The Economics of Open Source Software Development*. 2006.

[57] MILLER, B.P.; KOSKI,D.; LEE,C.P.; MAGANTY,V.; MURTHY,R.; NATARAJAN,A.; STEIDL, J.: Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services, , , 1995.

[58] MURGIA, A.; CONCAS,G.; PINNA,S.; TONELLI,R.; TURNU, I.: Empirical study of software quality evolution in open source projects using agile practices, *CoRR*, abs/0905.3287, 2009.

[59] NEGRO, P.; GIANDINI, R.. Umbrales para Métricas Orientadas a Objetos. En *36 JAIIO, ASSE 2007*. 2007.

[60] OLAGUE, H.M.; ETZKORN,L.H.; GHOLSTON,S.; QUATTLEBAUM, S.: Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes, *IEEE Trans. Softw. Eng.*, 33, 2007.

[61] PEERCY, D.E.: A Software Maintainability Evaluation Methodology, *IEEE Trans. Softw. Eng.*, 7, 1981.

[62] PERRY, D.E.. Dimensions of Software Evolution. En *In Proceedings of the IEEE International Conference on Software Maintenance. IEEE Computer*. 1994.

[63] PIZKA, M.; DEISSENBOECK, F.. How to effectively define and measure maintainability. En *SMEF*. 2007.

[64] PRESSMAN, R.: *Ingeniería del Software: un enfoque práctico*, .Mc Graw-Hill,

2002.

[65] RAINER, A.; GALE, S.. Sampling Open Source Projects from Portals: Some Preliminary Investigations. En *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium*. 2005.

[66] RAMIREZ, J.; GIL,G.; ROMERO,G.; GIMSON, L.. Indicadores de la Utilización del Bug Tracker en Proyectos F/OSS.. En *XV Congreso Argentino de Ciencias de la Computación*. 2009.

[67] RAMIREZ, J.; GIMSON, L.. Partir de Moodle: estimación del esfuerzo que insumiría el desarrollo de una plataforma similar. En *36° Jornadas Argentinas de Informática*. 2007.

[68] RAYMOND, E.. The Cathedral \& the Bazaar. http://catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/.

[69] RIAZ, M.; MENDES,E.; TEMPERO, E.D.. A systematic review of software maintainability prediction and metrics.. En *ESEM*. 2009.

[70] Robles, G.. **Empirical Software Engineering Research on Libre Software: Data Sources, Methodologies and Results**. Universidad Rey Juan Carlos.2006.

[71] SAMOLADAS, I.; BIBI,S.; STAMELOS,I.; BLERIS, G.. Exploring the Quality of Free/Open Source Software:
a Case Study on an ERP/CRM System. En *9th Panhellenic Conference in Informatics*. 2003.

[72] SAMOLADAS, I.; GOUSIOS,G.; SPINELLIS,D.; STAMELOS, I.. The SQO-OSS Quality Model: Measurement Based Open Source Software Evaluation. En *OSS*. 2008.

[73] SAMOLADAS, I.; STAMELOS,I.; ANGELIS,L.; OIKONOMOU, A.: Open source software development should strive for even greater code maintainability, *Commun. ACM*, 47, 2004.

[74] SELVARANI, R.; NAIR,T.G.; PRASAD, V.K.: Estimation of Defect Proneness Using Design Complexity Measurements in Object-Oriented Software, *Signal Processing Systems, International Conference on*, 0, 2009.

[75] SHATNAWI, R.: A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems. *IEEE Transactions on Software Engineering*, 99, 2010.

[76] SHEPPERD, M.; INCE, D.C.. Martin J. Shepperd: An Empirical and Theoretical Analysis of an Information Flow-Based System Design Metric. En *ESEC*. 1989.

[77] SOMMERVILLE, I.: *Ingeniería del Software: un enfoque práctico*, .Pearson Educación, 2005.

[78] SOWE, S.K.; ANGELIS,L.; STAMELOS,I.; MANOLOPOULOS, Y.. Using repository of repositories (rors) to study the growth of f/oss projects: A meta-analysis research approach. En *In Third International Conference on Open Source Systems*. 2007.

[79] SPINELLIS, D.: *Code Quality: The Open Source Perspective (Effective Software Development Series)*, .Addison-Wesley Professional, 2006.

[80] STAMELOS, I.; ANGELIS,L.; OIKONOMOU,A.; BLERIS, G.L.: Code quality analysis in open source software development, *Information Systems Journal*, 12, 2002.

[81] STAVRINOUDIS, D.; XENOS,M.; CHRISTODOULAKIS, D.. Relations between Software Metrics and Maintainability. En *FESMA 99 International Conference*. 1999.

[82] SUBRAMANYAM, R.; KRISHNAN, M.: Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects, *IEEE Transactions on Software Engineering*, 29, 2003.

[83] TIAN, J.: *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*, 1 .Wiley-IEEE Computer Society Press, 2005.

[84] WEIDERMAN, N.; BERGEY,J.; SMITH,D.; TILLEY, S.: Approaches to Legacy System Evolution, , , 1997.

[85] WEYUKER, E.: Evaluating Software Complexity Measures, *IEEE Transactions on Software Engineering*, 14, 1988.

[86] WHEELER, D.. Why Open Source Software / Free Software (OSS/FS, FLOSS, or FOSS)? Look at the Numbers!. http://www.dwheeler.com/oss_fs_why.html.

[87] YU, L.; SCHACH,S.; CHEN, K.. Measuring the Maintainability of Open-Source Software. En *Empirical Software Engineering, 2005. 2005 International Symposium on*. 2005.

[88] ZHANG, M.; BADDOO,N.; WERNICK,P.; HALL, T.. Improving the Precision of Fowler's Definitions of Bad Smells. En *SEW '08: Proceedings of the 2008 32nd Annual IEEE Software Engineering Workshop*. 2008.