UNIVERSIDAD DE DEUSTO

PROGRAMA:
CIENCIA DE LA COMPUTACION E INTELIGENCIA ARTIFICIAL

DESARROLLO DE UN METODO DE ESTIMACION DE ESFUERZO BASADO EN MEDIDAS DE TAMAÑO Y COMPLEJIDAD EN EL PARADIGMA ORIENTADO A OBJETOS

Tesis doctoral presentada por Dña. Rebeca Cortazar Golcoechea
Dirigida por la Dra. Asunción Barredo Fuentes

La Directora

La Doctoranda

Bilbao, Noviembre de 1998

Resumen

La presente tesis doctoral aporta un novedoso modelo de estimación del esfuerzo, basado en dos medidas, una de tamaño y otra de complejidad, centradas en los sistemas de información orientados a objetos. Para ello, hemos definido una medida de complejidad con una sólida base teórica y formal, que puede utilizarse desde la fase de análisis, puesto que en su derivación y validación se utilizan características independientes de la implementación. A continuación, hemos desarrollado un nuevo modelo de estimación analógico parametrizable, implementación del meta-modelo de Cowderoy y Jenkins, que se centra en la similitud que un proyecto posee con respecto a los casos almacenados en un repositorio de proyectos pasados. La validación empírica de dicho método se ha realizado en el entorno académico tanto en lo relativo al impacto de la medida de complejidad en la precisión de las estimaciones, como al propio modelo de estimación. La validez de la hipótesis de partida ha quedado demostrada en dicho estudio, obteniéndose unos resultados adecuados según Conte *et al.* y excepcionales considerando que nuestro modelo se puede utilizar en las fases iniciales del ciclo de vida de un proyecto.

Abstract

This doctoral dissertation provides a new resource estimation model, based on two measures, one related to size and the other to complexity, and focused on object oriented information systems. In order to do it, we defined a complexity measure, based on formal and solid theoretical support, that can be used from the analysis phase, since the features used in its derivation and validation are free from implementation details. Following, we developed a new analogical, customizable by parameters, estimation model, which is an implementation of the meta-model by Cowderoy and Jenkins, and centred on the similarity a project holds regarding the cases stored in a project repository. The empirical validation of this method was carried out in an academic environment, considering the impact of the complexity measure in the accuracy of the estimations, as well as the estimation model itself. The correctness of the initial assuption was proved in this work, as we obtained adequate results according to Conte *et al.* and outstanding ones, if we take into consideration that we can make use of our model in the early steps of the life cycle of a project.

AGRADECIMIENTOS

- A la Dra. Asunción Barredo, mi directora de tesis y amiga, por su apoyo, sus consejos y orientaciones y sin cuya ayuda me hubiera sido imposible llevar a cabo la misma.

- A todos mis compañeros de la Universidad (y especialmente a Begoña, compañera de fatigas), que con su ánimo han hecho posible que haya conseguido terminar esta tesis.

- Y por último, a José Luis, que me ha escuchado durante todos estos años, que me ha soportado en los momentos más difíciles y que ha estado a mi lado cuando le he necesitado.

A Luis, Itziar y Mikel

INDICE

INDICE DE FIGURAS

INDICE DE TABLAS

1. INTRODUCCION

«Information is of no value until a number is associated with it.»

Benjamin Franklin

1.1 MOTIVACION E INTERES DEL TRABAJO

Mejorar la calidad y el rendimiento de los productos software, así como la productividad de los equipos de desarrollo se ha convertido en un objetivo prioritario para cualquier organización cuya actividad se soporte en sistemas informáticos. Así como las prestaciones del hardware se han ido duplicando cada tres años aproximadamente, las mejoras en la productividad del software se han ido incrementando a un ritmo de un modesto 4% anual [Putn91]. A medida que los ordenadores son más sofisticados y potentes, los usuarios exigen un software acorde, es decir, sofisticado y potente. La forma en que se ha realizado el proceso de desarrollar software y de mantener sistemas ya en explotación, provoca grandes costes para la organización, y lo que es peor, la pérdida de oportunidades para nuevos desarrollos. Los problemas del software son muy serios y hoy en día afectan a muchas compañías y gobiernos de todo el mundo. Estudios recientes han mostrado que menos de un 1% de los grandes proyectos de sistemas de software terminados se entregan dentro de los plazos temporales previstos, ajustándose al presupuesto y cumpliendo todos los requisitos de usuario [Heem92, Lede93]. Es más, estadísticamente se ha demostrado que un proyecto grande de software termina como media con un año de retraso y cuesta el doble del coste inicial estimado. A principios de los 80, la *Government Accounting Office* de los EEUU realizó una investigación que arrojó los siguientes resultados:

- Software entregado pero nunca utilizado: 47%
- Software pagado pero nunca entregado: 29,7%
- Software utilizado pero con serias reformas o abandonado más tarde: 19%
- Software que entró en explotación tras sufrir modificaciones: 3%
- Software que pudo ser utilizado tal y como se entregó: ≈ 2%

Las publicaciones financieras proporcionan continuamente datos sobre fracasos atribuibles a problemas relacionados con el desarrollo del software. Las cuestiones que aparecen siempre son las siguientes:

- ¿ Por qué el desarrollo de software es tan costoso?
- ¿ Por qué se necesita tanto tiempo?

Consideremos por ejemplo Lotus 1-2-3 V.3, que llegó al mercado con un año de retraso sobre lo planificado. En esta aplicación, que se estima contaba con 400.000 líneas de código, se utilizaron 263 personas-año y costó 22 millones de dólares, de los cuales 15 se invirtieron en actividades de control de calidad. Esta es una inversión modesta si se compara con la que se hizo para desarrollar el *Space Shuttle* de la NASA. Se codificaron 25.6 millones de líneas de código, con un esfuerzo de 22.096 personas-año, y costó 1.200 millones de dólares [Schl89].

La realidad es que se están realizando inversiones fabulosas en la producción de nuevo software y en el mantenimiento y modificación del existente. Si además se añade la baja productividad de los equipos de desarrollo y la pobre calidad de los productos que se distribuyen, las empresas de desarrollo de software pueden ver peligrar su negocio y sus ganancias.

Desde el punto de vista estratégico, uno de los aspectos más preocupantes del problema del desarrollo de software hoy en día es *el gran número de proyectos que se entregan fuera de los plazos planificados*. Esto conduce a una pérdida de posibilidades de ventas y una situación de descontento del cliente. Además, debe tenerse en cuenta que en determinados sectores de alta tecnología, la organización que llega al mercado en primer lugar con un nuevo producto, es la que se queda con un segmento mayor que el de sus competidores. Por tanto, es de vital importancia tener la capacidad de realizar unas estimaciones precisas del tiempo de desarrollo tratándose de los productos de software. También es importante reducir el tiempo que transcurre desde que se concibe el producto hasta que éste alcanza al cliente.

Por otra parte, los problemas derivados de la calidad y rendimiento del software pueden afectar de forma adversa a *las relaciones entre una organización y sus clientes*. El software que se distribuye a los clientes hoy en día rara vez es perfecto. Es muy habitual que los clientes descubran errores ocultos después de la adquisición del producto. La probabilidad de que esto ocurra puede minimizarse implementando técnicas de control de calidad durante el proceso de desarrollo, de modo que se cometan menos errores y se identifiquen y corrijan antes de su distribución al cliente un alto porcentaje de estos fallos. El problema de la baja calidad en el software es especialmente grave en los sistemas utilizados en entornos en los que la seguridad es un aspecto de vital importancia. En un sistema de control industrial, de instrumentación médica, de energía nuclear, de control de tráfico aéreo, etc. un error latente en el código puede provocar incluso pérdidas de vidas humanas.

Muchas organizaciones están descubriendo que tanto los problemas de planificación de proyectos como de calidad en el software son más problemas relacionados con la *gestión* que meramente *técnicos*. IBM lleva realizando un estudio desde principios de 1996, con aproximadamente 500 organizaciones europeas de desarrollo de software, con el fin de conocer la situación de las mismas en Europa, de forma que ellas puedan establecer unos criterios de comparación a nivel de negocio, a nivel de país y a nivel de la muestra completa, dentro del contexto de la operativa utilizada en el desarrollo de software.

Los resultados han sido sorprendentes. En este estudio, se ha identificado la existencia de una correlación entre los distintos procedimientos operativos de desarrollo empleados y los niveles de rendimiento alcanzados. Los resultados muestran los procedimientos que tienen más éxito, así como también los aspectos en los que las inversiones no parecen tener ninguna correlación con los niveles de rendimiento alcanzados.

Entre los aspectos que más impacto tienen sobre la calidad y el rendimiento desde el punto de vista de la entrega de producto terminado, están la *Gestión de Proyectos*, el *enfoque de Estimación*, las *Métricas* y otros aspectos que pertenecen al *Peopleware* (implicación y moral de los empleados, relaciones con los clientes,...). Los aspectos que menos impacto tienen son los de tipo tecnológico: uso de herramientas CASE, nivel de lenguaje de programación utilizado, esfuerzo dedicado a la prueba, etc. Las organizaciones con procedimientos operativos estables obtienen una productividad en el desarrollo cinco veces superior a las que no las tienen, en el mantenimiento es diez veces superior y la precisión de sus estimaciones se mueve dentro de! 10%. En cambio, en el resto ésta es superior al 40% (Kunt97).

1.1.1 Gestión y Medidas en el Proceso de Desarrollo del Software

La gestión de un proyecto software cubre todo el proceso de desarrollo del mismo desde el principio al fin. Para conseguir un proyecto software fructífero, que minimice los problemas planteados en la sección anterior, la medición es fundamental.

Las *medidas o métodos cuantitativos* ayudan a entender tanto el proceso técnico que se utiliza para desarrollar un producto, como el propio producto. El proceso se mide para intentar mejorarlo. El producto se mide para intentar aumentar su calidad. Básicamente, la idea que subyace es el hecho de que *no se puede controlar aquello que no se puede medir* [DeMa82].

Las medidas del software proporcionan beneficios reales y tangibles [Gile95]. Aunque esto pueda parecer irreal con respecto a la experiencia de algunas organizaciones, para aquellas que han disfrutado de los resultados de un uso efectivo de la medición, los beneficios son patentes. Desde el punto de vista de la Ingeniería del Software, las medidas:

- proporcionan un control sobre el proceso y el producto,
- sirven para demostrar la existencia de factores como la efectividad, la productividad y la calidad del producto, y
- ayudan a identificar áreas de complejidad en el software en desarrollo.

Desde el punto de vista de la dirección, las medidas:

- fomentan el respeto de los clientes y aumenta la credibilidad de la dirección, incrementándose también la satisfacción de dichos clientes debido a la mayor calidad del producto,
- dan soporte a la estimación de costes, tiempos, recursos y calidad, de forma más precisa, debido a la existencia de datos históricos,
- proporcionan información sobre el estado y progreso real de un proyecto, de manera que se pueda hacer frente a problemas potenciales,
- permiten la corrección de los problemas antes de que las consecuencias sean evidentes,

- sirven de guía en la toma de decisiones con respecto a los recursos, los ajustes por prioridades y la estabilidad de la planificación; así, dichas decisiones se fundamentan en datos, no en meras conjeturas,
- proporcionan datos sólidos sobre los que se pueden establecer mecanismos de mejora de procesos,
- favorecen la disminución del costo de desarrollo, como resultado de un incremento de la productividad, y
- provocan mejoras en la capacidad de la organización para planificar nuevos proyectos.

De todas maneras, la implementación de un sistema de métricas depende en gran medida del apoyo que la dirección de la organización proporcione. Es necesario un cambio de actitud en los equipos de desarrollo, que debe ir orientado hacia la mejora de la calidad. La dirección debe apoyar el uso de las métricas para acelerar esos cambios culturales a nivel corporativo.

Actualmente, la aplicación de métricas está directamente relacionada con los Programas de Mejora Continua, relacionados con el *Modelo de Madurez del Proceso del Software* creado por el *Software Engineering Institute* (SEI). El desarrollo en cada nivel se supone más organizado y mejor gestionado que en los niveles inferiores. Esto se basa en la premisa que se producirá software de mejor calidad con menos recursos, a medida que el proceso de desarrollo vaya madurando. Así, el papel que juega el uso de las métricas en una organización llega a ser clave según va aumentando su nivel de madurez.

1.1.2 La Tecnología Orientada a Objetos

A las dificultades asociadas a la ya de por sí compleja tarea de la medición y la estimación, debemos añadir la del *vértigo tecnológico* en el que nos encontramos sumidos. Así, cuando parecía que se había alcanzado cierto dominio sobre el paradigma clásico de desarrollo, comienzan a aparecer en el mercado nuevos lenguajes de programación, nuevas filosofías de desarrollo, conceptos como la reusabilidad, cambios en las arquitecturas tradicionales del software, etc.

Obviamente todo ello afecta de manera drástica a los métodos de estimación de costes y tiempos, así como a la aplicabilidad de las medidas que se han ido definiendo a lo largo de los últimos veinte años.

En las secciones anteriores hemos comentado lo extraordinariamente complejo que puede resultar diseñar software y gestionar su proceso de desarrollo. También hemos hecho hincapié en que utilizar medidas es un método sistemático de desarrollar una disciplina en este campo del conocimiento.

Hemos resaltado que al convertirse la mejora continua del proceso de desarrollo en un objetivo de muchas organizaciones, ha aumentado la demanda de métricas con las que gestionar el proceso de desarrollo del software. La necesidad de éstas se hace particularmente patente en el caso de una organización que adopta una nueva tecnología como la *Tecnología Orientada a Objetos*, para la cual todavía no se han establecido procedimientos estándares. Además, como Booch afirma, al adoptar una nueva tecnología, se corre el riesgo de sufrir una regresión en la escala de niveles del SEI con lo cual se debe re-elaborar el proceso que se sigue para el desarrollo de sistemas [Booc94].

El enfoque de la Orientación a Objetos (OO) se centra en modelizar el mundo real en términos de sus objetos. Esto contrasta con enfoques más antiguos y tradicionales que enfatizan la visión orientada a la función que separa los datos de los procedimientos. Algunos autores incluso han especulado con la idea de que los enfoques OO inducen a un comportamiento diferente en la resolución de problemas y en el proceso cognitivo del diseño [Booc94].

A la vista de los conceptos tan diferentes en los que se basan estas dos técnicas de desarrollo (Orientada a Objetos y Tradicional), no es sorprendente descubrir que las medidas elaboradas para los métodos tradicionales no se ajustan a los conceptos de la OO como clases, herencia, encapsulamiento y comunicación por mensajes. Por ejemplo, las clases pueden contener varios métodos, cada uno de los cuales tiene unas pocas líneas de código y con un flujo de control pobre, por lo general. Mediante el uso de una métrica tradicional, se obtendría un valor de complejidad bajo para cualquiera de estas clases y sin embargo, el número de métodos y el número de variables utilizadas, a menudo implican dificultades en la verificación y en el rendimiento. Las métricas tradicionales no captan este aspecto de la complejidad. Otro ejemplo de la insuficiencia del alcance de las métricas tradicionales es la incapacidad de considerar las relaciones de herencia entre clases. Diversos investigadores han estudiado sobre las limitaciones de las métricas existentes y establecido la necesidad de nuevas métricas especialmente diseñadas para la Orientación a Objetos. Se ha sugerido que las métricas tradicionales, como los Puntos de Función de Albretch o la Complejidad de McCabe, pueden ser inadecuadas para el diseño y el desarrollo OO, a menos que sufran considerables modificaciones para soportar las características intrínsecas de este nuevo paradigma.

Por tanto, dado que las métricas actuales están sujetas a críticas y que no soportan conceptos claves del paradigma orientado a objetos, se hace necesario el desarrollo de un conjunto de métricas especialmente diseñadas para medir aspectos propios del Paradigma Orientado a Objetos, de manera que sirvan de base a la estimación de costes y a la planificación temporal.

1.2 OBJETIVOS

El motivo de esta investigación surge del reconocimiento de que el software en general, y en particular el software Orientado a Objetos, es un elemento clave y estratégico en el desarrollo global de cualquier organización y por tanto, la gestión de su proceso de producción es de vital importancia. De este modo, proporcionando un conjunto de medidas estables del proceso de producción del software, puede esperarse que deje de ser considerado 'artesanía'.

Aunque en la literatura reciente se han publicado diferentes propuestas con respecto a las medidas del software OO, estas resultan insuficientes debido a que incumplen requisitos indispensables: no se sustentan en una base teórica sólida, no cuantifican de manera precisa la característica que pretenden medir, no soportan una validación formal y empírica, y no se pueden calcular por medios automatizados. Además, hasta el momento han sido escasos los intentos de definir medidas específicas orientadas hacia la estimación de esfuerzos, traduciéndose dichos intentos en propuestas insuficientemente validadas [Morr89, Lore94].

En consecuencia, el primer objetivo de esta tesis es desarrollar un conjunto de medidas con una sólida base teórica y formal que sirvan como base para la estimación del esfuerzo de desarrollo de un producto de software. Para que estas medidas puedan utilizarse desde la fase de análisis, se pretende utilizar en su derivación y validación características independientes de la implementación; nos centraremos únicamente en elementos pertenecientes al Dominio del Problema.

El segundo objetivo es obtener un modelo de estimación de esfuerzo, adaptado a las características del software que las medidas aportadas proporcionan. Para ello, utilizaremos el meta-modelo de Cowderoy y Jenkins [Cowd88] y los estudios realizados por Boehm en el desarrollo del modelo COCOMO [Boeh95, Boeh95a]. Nuestra elección de una solución basada en *Estimación por Analogía* se debe a que nos permite trabajar con información insuficiente y subjetiva, manipular cualquier tipo de característica y se adapta a la evolución del software y a los cambios en los conductores de coste. Además, estas estrategias han demostrado tener éxito en dominios donde las normas están insuficientemente definidas y no hay un modelo

teórico sólido, situación que encaja en la realidad de la estimación de esfuerzos en el desarrollo de software.

Una vez desarrollado el método de estimación, el siguiente paso será su automatización. Para ello, se construirá un herramienta que, ante la introducción de los datos relacionados con las medidas del software especificado, realice una estimación del esfuerzo necesario para su desarrollo.

1.3 ORGANIZACION DE LA MEMORIA

La presente memoria se compone de tres grandes partes que configuran las fases en las que se ha dividido el trabajo de investigación realizado: estudio de las diferentes áreas de conocimiento implicadas en el problema y definición del trabajo, desarrollo del mismo, y obtención y presentación de resultados. Estas partes se estructuran a su vez en los capítulos que se exponen a continuación.

El capítulo 2 presenta de manera detallada los antecedentes y un análisis de las soluciones existentes en el ámbito de las distintas áreas implicadas en el trabajo. En primer lugar, se describen los conceptos básicos de la Tecnología de Objetos, señalando las diferencias conceptuales con el desarrollo tradicional. Seguidamente, se comenta de manera sucinta la evolución histórica de las medidas del software, examinando de forma más exhaustiva las distintas medidas OO propuestas en la literatura reciente. A continuación, se analizan los diferentes métodos actuales de validación de medidas. Para ello, se describen los conceptos fundamentales de la *Teoría de la Medición*, que forman la base de las propiedades que una medida sólida y formal debe cumplir. Por último, se describen los distintos métodos de estimación desarrollados hasta el momento, haciendo especial hincapié en los Métodos No Paramétricos, los Analógicos y los Orientados a Objetos.

El capítulo 3 incluye tres secciones diferenciadas. En primer lugar, se describe y utiliza el modelo *GQM* de Briand, Morasca y Basili [Bria94] con el fin de definir con precisión los objetivos de las medidas a desarrollar y las propiedades formales que éstas deben cumplir. Así, se describe el marco global de definición y validación de las medidas, los pasos a dar en dicha definición y los supuestos de partida. También se describen las fases del proceso de desarrollo genérico a seguir y los ciclos de vida posibles, además del proceso básico de estimación.

En segundo lugar, se documenta un modelo matemático y por tanto formal, basado en la *Teoría de Categorías* [Barr95] que captura las propiedades de un sistema de información orientado a objetos. El objetivo de este formalismo es proporcionar

rigor y precisión en la medición de las propiedades estáticas de un modelo de objetos y por tanto, permitir medir características como el tamaño y la complejidad.

En tercer lugar, se describen dos medidas: una medida de tamaño, validada teóricamente a la luz de las propiedades de Whitmire [Whit97], y una medida de complejidad, denominada ID_{medu}, que captura el grado de interdependencia entre los elementos de un modelo, proporcionando una medida de la dificultad de su desarrollo. También se detallan las propiedades formales que cumple esta medida, en base a las propuestas de Whitmire [Whit97], Weyuker [Weyu88] y Zuse [Zuse98].

El capítulo 4 está dedicado a la descripción del método de estimación de esfuerzos ESTIMAN aportado; método que se centra en el Razonamiento Basado en Casos y más en concreto, en el Razonamiento Analógico. Se detallan los cinco pasos que se siguen en el proceso de estimación, siendo de especial interés el cálculo del *índice de similitud*. Tomando como hipótesis que proyectos similares tendrán esfuerzos de desarrollo parecidos, este índice permite valorar la similitud de los proyectos en función distintas características, con vistas a poder extrapolar el esfuerzo de desarrollo de los proyectos similares ya realizados. Evidentemente, esto convierte a ESTIMAN en un método genérico, adaptable a las distintas características que se quieran analizar con fines estimadores. En el caso de este trabajo, nos hemos centrado en las dos medidas descritas en el capítulo 3.

Por otra parte, un proyecto no se puede caracterizar únicamente en función de su tamaño y complejidad; existen otros factores que afectan al esfuerzo de desarrollo que también deben ser tomados en consideración. Boehm describe un posible conjunto de factores en su modelo COCOMO [Boeh95, Boeh95a] y los hemos utilizado en el proceso de ajuste de las características que no se corresponden entre el proyecto objetivo y los proyectos base.

El entorno operativo y la validación empírica se detallan en el capítulo 5. En primer lugar, se describen de forma exhaustiva los distintos módulos que forman el entorno integrado de medición y estimación, y sus componentes, así como las estructuras de datos internas necesarias para su correcto funcionamiento.

Seguidamente, se detallan los métodos estadísticos utilizados en la prueba de campo y los resultados de dicho estudio, que demuestran que el método ESTIMAN, parametrizado con las medidas de tamaño y complejidad propuestas en el capítulo 3, es un método útil, al ser estadísticamente preciso y sobre todo, considerando que se puede utilizar en fases iniciales del proceso de desarrollo, cuando se dispone de escasa información sobre el proyecto.

La memoria se completa en el capítulo 6 con las conclusiones finales del trabajo y las líneas de investigación que quedan abiertas, la bibliografía utilizada y de relevancia para la investigación y cuatro apéndices. El apéndice A proporciona una especificación formal de un modelo orientado a objetos, además de la descripción algebraica de las medidas. El apéndice B ilustra las tablas de valoración numérica de los factores de escala y conductores de coste del *Modelo Post-Arquitectural de COCOMO II.1997*. En el apéndice C, se documentan los *Formularios de Recogida de Datos* y los de *Cálculo de Medidas*, así como un ejemplo debidamente cumplimentado de cada uno de ellos. En este apéndice, también se detalla el *Cuestionario de Definición de Contextos*, que permite capturar informacion relacionada con los factores que caracterizan a un proyecto y que afectan a su desarrollo. Por último, en el apéndice D, se aporta información adicional sobre las pruebas estadísticas realizadas.

2. ANTECEDENTES Y ANALISIS DE LAS SOLUCIONES EXISTENTES

«A major difference between a 'well-developed' science such as Physics and some of the less 'well-developed' sciences such as Psychology or Sociology is the degree to which things are measured.»

Fred S. Roberts, 1979

2.1 INTRODUCCION

En este capítulo presentamos un resumen de los antecedentes y estado del arte de las áreas de conocimiento implicadas en este trabajo de investigación y base del mismo. Las cuatro disciplinas fundamentales son la *Filosofía Orientada a Objetos*, las *Medidas del Software*, los *Métodos de Validación de las Medidas* y los *Modelos de Estimación de Recursos*.

2.2 LA TECNOLOGIA ORIENTADA A OBJETOS

La *Tecnología Orientada a Objetos*, que incluye el *Análisis Orientado a Objetos*, el *Diseño Orientado a Objetos*, la *Programación Orientada a Objetos* y las *Bases de Datos Orientadas a Objetos*, además de novedosa, es la filosofía de desarrollo más prometedora hoy en día y se espera que alivie todos los problemas de coste, planificación y calidad que se dan en los sistemas de software. Los defensores de la Orientación a Objetos afirman que este enfoque transformará el desarrollo de aplicaciones en una actividad de ensamblaje de objetos estándar que estarán disponibles en extensas bibliotecas públicas, privadas y comerciales. No es sorprendente que las palabras 'Orientado a Objetos' u 'OO' se hayan convertido en palabras mágicas dentro de la Industria del software de los 90.

2.2.1 Conceptos Básicos

El principio central de la Orientación a Objetos es que el software debería ser una representación del espacio del problema, en vez de ser simplemente una secuencia de instrucciones que un ordenador debe ejecutar para resolver el mismo. Como un modelo del dominio del problema rara vez separa los datos de las operaciones que se realizan sobre ellos, un requisito fundamental de la OO es el hecho de que los datos y las instrucciones deben estar integrados en una única abstracción llamada *objeto*.

Un *objeto* es un elemento que representa un aspecto del espacio del problema, contiene información (*atributos* o variables) e instrucciones (*métodos*) que permiten al objeto responder a peticiones (*mensajes*) de otros objetos. Esta es la principal diferencia con los paradigmas de desarrollo tradicionales que separan los programas de los datos y los gestionan de modo independiente.

Los objetos individuales son únicos puesto que tienen nombres que los identifican y valores propios. Sin embargo, varios objetos pueden pertenecer a una misma *clase* si comparten los mismos tipos de variables y métodos. Así, se puede considerar una clase como una plantilla o prototipo de una colección de objetos.

Las clases y objetos son los bloques constitutivos básicos del software OO. El poder del paradigma OO es la facilidad con la que una aplicación puede ser modelizada utilizando objetos y la capacidad de ir añadiendo abstracciones incrementalmente. Con la acumulación de clases y objetos a largo plazo, el desarrollo de nuevas aplicaciones puede llegar a convertirse en un ensamblaje de objetos ya existentes, con lo que se fomentará la reusabilidad al máximo. Para referirse a una clase, se utilizan distintos términos dependiendo de la situación en la que se utiliza: superclase, subclase, clase padre, hijo, ascendente, descendente, ancestro,... Todos ellos hacen referencia a la situación de una clase con respecto a otras.

Los atributos o variables son los datos que un objeto posee y representan el estado de un objeto. Los métodos son operaciones o funciones; es decir, un conjunto de actividades que deben realizarse y definen el rol o papel de un objeto. Proporcionan un mecanismo para asignar, recuperar y modificar el estado actual de un objeto y llevan a cabo actividades de comunicación de los servicios solicitados y ofrecidos por los distintos objetos del sistema.

Otros conceptos básicos y diferenciadores de la OO son el *encapsulamiento*, el *ocultamiento de información*, la *herencia*, la *comunicación de mensajes*, la *asociación* y el *polimorfismo* (sinónimo de enlace dinámico).

El encapsulamiento hace referencia al hecho de agrupar los atributos de un objeto y todas las operaciones válidas que actúan sobre esos atributos en un paquete, protegiendo, gracias al ocultamiento de información, el estado de un objeto de las actividades de otros objetos [Bera97]. La manipulación del estado (los valores de los atributos) debe ser privada y de alcance restringido. La única manera de acceder a los valores de datos es a través de la invocación de métodos especiales que son parte del mismo objeto. La decisión de cuánta información sobre un objeto debe estar a disposición de otro depende del dominio del problema y del diseño. Un método público invocado desde un objeto externo puede depender de un método privado para poder realizar su función. Estos detalles internos son independientes del objeto externo que invoca o necesita al método público. Un uso adecuado de la encapsulación y del ocultamiento de información reducen la complejidad, mejoran la modularidad y facilitan el mantenimiento.

La herencia es posiblemente la característica más diferenciadora de la OO. Permite diseñar clases de objetos que son especializaciones de otras clases de objetos. Permite la definición de una jerarquía de clases, donde las clases inferiores heredan los atributos y métodos de la clase o clases superiores (herencia simple y herencia múltiple), además de incorporar sus propios atributos y métodos. Desde la perspectiva de la Ingeniería del Software, la herencia proporciona un mecanismo para la compártición de código y la reusabilidad.

A diferencia del software diseñado con métodos tradicionales, los objetos no realizan llamadas a funciones; se comunican mediante el paso de mensajes. El receptor de un mensaje invoca a un método propio para que lo responda. Este mecanismo de comunicación por mensajes es otra característica diferenciadora de la OO.

La asociación es el mecanismo por el que se establecen relaciones entre objetos. Existen dos tipos de relaciones entre objetos: un objeto puede estar *contenido* en otro objeto o puede estar *ligado* a otro objeto, asociado con él. Este último tipo de asociación implica el envío y recepción de mensajes entre ambos objetos y por tanto, es una relación cliente/servidor entre ambos. El objeto que juega el papel de cliente realiza peticiones al objeto servidor. Una petición accede al protocolo que invoca a un método público que pertenece al objeto servidor. Otro tipo de asociación ya tratada previamente es la relación de herencia entre objetos.

Por último, el polimorfismo proporciona un mecanismo para caracterizar el tipo de una variable, método u objeto. Se refiere a la capacidad de añadir nuevos objetos y clases en la jerarquía de herencia, sin tener que recompilar el código existente. Con esta característica, es posible invocar operaciones en un objeto sin considerar el tipo del objeto hasta el momento de la ejecución.

2.3 ANTECEDENTES DE LA MEDICION DEL SOFTWARE

La Medición del software es una de las líneas que más investigación ha generado desde mediados de los 70. Hoy en día, en cualquier foro que trate de Ingeniería del Software, siempre sale a relucir el tema de la medición. Realmente, la necesidad de medición que existe en esta disciplina deriva directamente de la propia definición de Ingeniería del Software:

> «La aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software; es decir, la aplicación de la ingeniería al software.» [IEEE90]

El término *cuantificable* en la definición anterior es la indicación de que, al menos el IEEE, considera la medición como parte integral oe una filosofía de software bien concebida y no meramente un aspecto adicional.

2.3.1 ¿Qué es la Medición?

La Medición en las ciencias cuenta con una larga tradición. Así, la medida de la longitud, el *metro*, data de 1889, y sólo hay que considerar que Fahrenheit presentó la graduación de la distancia entre puntos fijos en 1714 y Celsius en 1742. En la Ingeniería del Software, la magnitud de los costes implicados en el desarrollo y mantenimiento acrecenta la necesidad de unas bases científicas de medición en las que apoyar los estándares de programación y la toma de decisiones relacionadas con la gestión. Ya en 1980, Curtis afirmaba:

> «Se deben aplicar procedimientos científicos rigurosos al estudio del desarrollo de sistemas de software si queremos transformar la programación en una disciplina de ingeniería. En el núcleo de estos procedimientos se encuentra el desarrollo de técnicas de medición y la determinación de relaciones causa-efecto.» [Curt80]

Como hemos señalado anteriormente, el establecimiento de estructuras de medición en la Física llevó mucho tiempo y la realidad está demostrando que el proceso será similar y muy largo con la medición del software. Podemos definir la *Medición* como *el proceso mediante el que se asignan números o símbolos a los atributos de las entidades del mundo real de manera que dichos atributos sean descritos de acuerdo a unas reglas previamente definidas* [Fent96].

En la Figura 2.1 se ilustra el proceso de Medición de Kriz, modificado por Zuse (Zuse98).

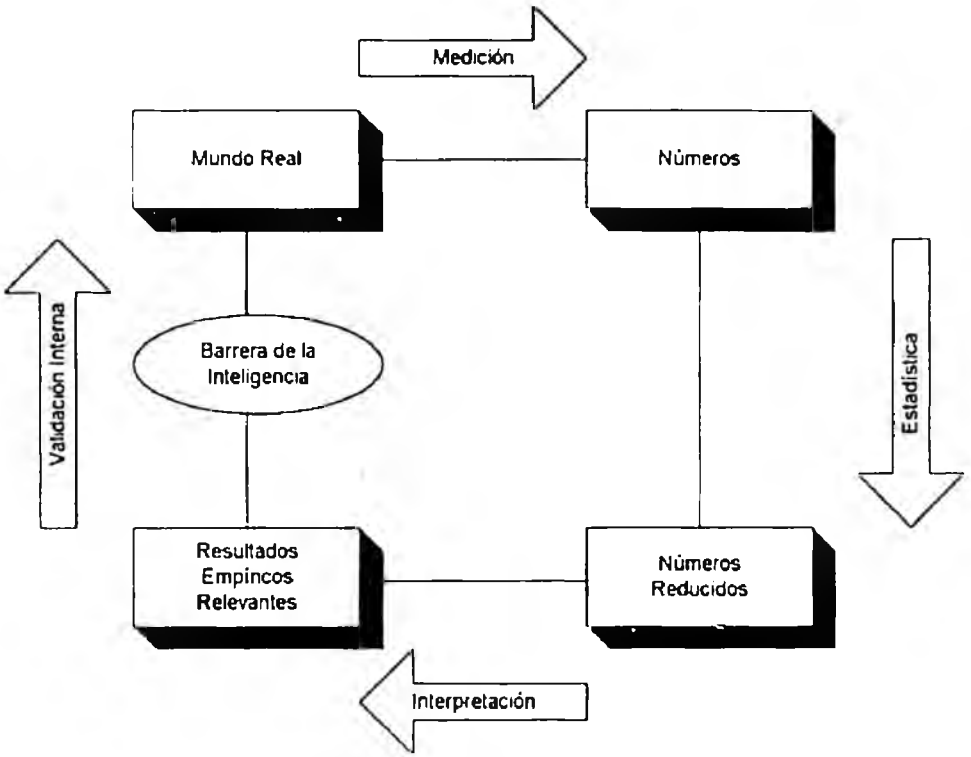

Figura 2.1 Proceso de Medición modificado de Zuse.

El proceso de medición comienza en el *mundo real*, donde se encuentran los objetos a medir y sobre los cuales queremos realizar afirmaciones empíricas relevantes, como por ejemplo, *A* es más fácil de mantener que *B*. Sin embargo, en ocasiones nuestro cerebro no puede producir afirmaciones empíricas, no podemos reducir la información sin ayuda; así, no podemos valorar directamente la facilidad de mantenimiento de un módulo. Es lo que el modelo denomina *barrera de la inteligencia*. Una manera de superar la barrera de la inteligencia es mediante el uso de modelos matemáticos y estadísticos. Pero una condición previa es que la información empírica de los objetos y sus relaciones se hayan transformado de manera adecuada en objetos numéricos y sus relaciones: es lo que llamamos *medición*. La estadística se utiliza para poder procesar la información y obtener mediante este proceso unos *números reducidos*. Un paso importante es la *interpretación* o asignación de un significado empírico a estos números; sin esta interpretación no podemos hacer afirmaciones empíricas. Por último, se debe comparar la interpretación empírica de los números con las afirmaciones del mundo real; así, sabemos si la medida cumple con los requisitos inicialmente planteados.

A este proceso se le denomina *validación interna*. Es un paso difícil en el que se comprueba que ambas visiones encajan: la del mundo real y la de los números.

En cuanto al uso de los términos, tradicionalmente en el mundo del software se ha utilizado el término *métrica* para denotar la asignación de un número a un atributo del software, aunque en los últimos años, varios autores se han centrado en la distinción entre lo que es una *métrica* y una *medida*.

Así, una *métrica* es estrictamente una función con dos argumentos (por ejemplo, la distancia entre dos puntos), mientras que una *medida* es un valor numérico para un atributo cuya magnitud se desea valorar en función de una escala concreta. Aunque esto es estrictamente correcto y por lo tanto, el término adecuado es *medida*, la gran mayoría de los autores continúan utilizando ambos indistintamente [Hend96]. Nosotros también usaremos ambos términos a lo largo de la disertación.

2.3.2 Categorización según Fenton

La taxonomía de las medidas de Fenton [Fent96] se basa en las entidades del mundo real del software y sus atributos.

Así, distingue tres tipos de entidades:

- *Procesos*: conjuntos de actividades relacionadas con el software.
- *Productos*: cualquier elemento de configuración resultante de una actividad.
- *Recursos*: entidades que una actividad necesita.

Para cada tipo de entidad, se distinguen dos tipos de atributos:

- Atributos *Internos* de un proceso, producto o recurso: aquellos que se pueden medir directamente, en función del proceso, producto o recurso en sí.
- Atributos *Externos* de un proceso, producto o recurso: aquellos que se pueden medir en función de cómo se relaciona el proceso, producto o recurso con su entorno.

Por lo tanto, podemos obtener una medida directa e interna de un producto, como por ejemplo, el número de líneas de código, con vistas a estimar una medida externa de proceso como puede ser el esfuerzo de desarrollo.

2.3.3 Necesidad de las Medidas del Software

Aunque sabemos que el uso de las medidas del software no es la panacea para todos los males del proceso de desarrollo, su utilización en unión de otras actividades, tales como la captura precisa de información sobre los equipos de desarrollo y de control de calidad y una formación adecuada, ha demostrado ser un factor catalizador de la calidad y la productividad. Las medidas de software permiten dar respuesta a las siguientes preguntas:

1. ¿Cómo podemos valorar las mejoras y la migración a nuevos entornos?

2. ¿Podemos reducir la impredecibilidad asociada a los proyectos de desarrollo de software? ¿Cómo podemos detectar los problemas ocultos a tiempo?

3. ¿Cuánto cuesta cada proceso? ¿Cuál es la productividad del equipo de desarrollo? ¿Cómo podemos cuantificar la extensión y los beneficios aportados por la reusabilidad?

4. ¿Cuál es la calidad de los productos que se están desarrollando? ¿Estará el usuario satisfecho con el producto? ¿Cómo podemos mejorar la calidad?

5. ¿Se pueden probar los requisitos? ¿Hemos encontrado todos los errores? ¿Cuál será el esfuerzo de mantenimiento?

Poder dar respuestas precisas y fiables a las cuestiones anteriores constituye una innegable ventaja competitiva para las organizaciones de desarrollo que lanzan programas de establecimiento de métricas. Además, no hay que olvidar que estos programas constituyen el requisito inicial de cualquier esfuerzo de certificación [Brit94, Fent96].

2.3.4 Métricas del Software: Pasado y Presente

Los cimientos de las Métricas del Software y del proceso de Medición se establecieron en los años 60 y sobre todo en los 70; a partir de estos primeros trabajos, se han ido proponiendo nuevos enfoques durante los 80 y 90 [Zuse97].

El motivo que subyace a la *creación* o *invención* de las métricas del software es el conocimiento de que la estructura de un programa y la modularidad son aspectos fundamentales del desarrollo de software fiable.

La mayoría de los expertos en software están de acuerdo en que las cotas más altas de fiabilidad se alcanzan cuando los sistemas de software están altamente modularizados y la estructura de los módulos es simple [Schn77]. La modularidad es una de las características a las que más pronto se le prestó atención. Parnas [Parn75] sugirió que los módulos debían estructurarse de forma que un módulo no tuviera conocimiento de la estructura interna de otros módulos.

Stevens, Myers y Constantine [Stev74] describieron el concepto de la fortaleza de un módulo, introduciendo los términos *acoplamiento* y *cohesión*; Yourdon analizó la modularidad en 1975 [Your75]. Estos factores afectan tanto al coste como a la calidad del software como indica, por ejemplo, Porter [Port90].

La primera medida directa del *Tamaño del Software* que se empezó a utilizar fue la Línea de Código (LOC - Line of Code); es una medida muy polémica pero que todavía se usa hoy en día [Park92].

En cuanto a la *Complejidad del Software*, es probable que la primera publicación sobre el tema sea la de Rubey *et al.* en 1968 [Rube68]. En 1979, Belady [Bela79] menciona la existencia de una tesis doctoral sobre la complejidad de software fechada en 1971. Belady se plantea definir la complejidad de los programas, la cuantificación de la complejidad... y buscando en la literatura, localiza la tesis de Van Emden: 'Un Análisis de la Complejidad' [Emde71]. Este trabajo se centraba en el estudio de la complejidad desde el punto de vista del Teoría de la Información y hacía factible la construcción de un modelo de complejidad de sistemas interconectados (programas compuestos de módulos).

En 1975, Kolence acuñó el término *Física del Software* [Kole75] y en 1977, Halstead introdujo el de *Ciencia del Software* [Hals77]. La idea que subyacía en estos términos era la de aplicar métodos científicos a las propiedades y estructuras de los programas de ordenador. La teoría de Kolence hace converger a las medidas de rendimiento clásicas como el tiempo de respuesta y la disponibilidad hacia las medidas de gestión tradicionales como la productividad, el coste por unidad de servicio y los presupuestos. Sin embargo, la Física del Software se encuentra entre las primeras teorías que tratan exclusivamente con la medición del tamaño y carga de los programas de computador. Además, fue la base para una gran cantidad de estudios adicionales [Wals77a, Fitz78, Bank79].

Una de las medidas más famosas sobre las que hoy en día todavía se investiga, fue creada a mediados de los 70: la medida de McCabe [McCa76], que se centra en los caminos de control de un programa. Esta es una de las métricas que más variaciones, extensiones y estudios adicionales ha tenido [Hans78, Schn79].

En 1977, Hecht [Hech77] propuso dos medidas de complejidad del software: la *longitud de secuencia derivada del intervalo* y la *conectividad* del bucle. Se basan en la posibilidad de dividir un grafo de flujo en intervalos. Las propuestas de Rubey, Van Emden y Hecht no han tenido trascendencia (aunque la medida de Van Emden se utilizó como base de la medida de complejidad de Khoshgoftaar *et al.* [Khos92]).

En esta época, Gilb [Gilb77] publicó un libro que es un clásico en el área de Medidas del Software. En 1978, Jones [Jone78] publicó un artículo en el que consideraba los distintos métodos para medir la calidad del software y la productividad; aparecen sus inquietudes sobre las unidades de medida. También McClure propone sus medidas de complejidad [McCl78].

En 1979, Belady propone su medida BAND que es sensible al anidamiento [Bela79] y en 1980, Oviedo [Ovie80] desarrolla un Modelo de Calidad de Programas. Este modelo trata simultáneamente la complejidad del flujo de control del programa y la del flujo de datos. Así, la complejidad del programa es una medida única compuesta de dos parámetros. Durante este año, Curtis publica un artículo de importancia que se centra en el rigor de las mediciones [Curt80, Jone78].

En 1981, Ruston propone una medida que describe el flujo de un programa mediante un polinomio. Esta medida considera tanto los elementos del diagrama de flujo como su estructura. Este método parece ser correcto pero no se ha extendido como el de McCabe [Rust81]. Harrison *et al.* desarrollan una medida de complejidad basada en la descomposición de grafos de flujo en rangos. Así, se puede determinar el nivel de anidamiento de los nodos en diagramas estructurados y, especialmente, en los no estructurados [Harr81].

En 1982, Piwowarski sugirió una modificación de las medidas de Harrison porque presentaban algunos problemas, como por ejemplo que un diagrama no estructurado era menos complejo que su versión estructurada [Piwo82]. Troy *et al.* [Troy81] proponen un conjunto de 24 medidas para analizar la modularidad, tamaño, complejidad, cohesión y acoplamiento de un sistema de software. En concreto, el acoplamiento y la cohesión son criterios fundamentales para medir la facilidad de compresión de sistema, entre otros aspectos. La división básica de las medida de complejidad en componentes inter-modulares e intra-modulares y los conceptos específicos de acoplamiento y cohesión se basan en los trabajos de Constantine [Stev74]. Basándose en ellos, se han realizado innumerables intentos de crear medidas de complejidad de la interconectividad de grandes sistemas. En 1981, Henry y Kafura [Henr81] propusieron su métrica de interconectividad: se basa en la multiplicación del grado de entrada y de salida (*fan in* y *fan out*) de los módulos de un sistema.

A finales de los 80, propusieron una modificación: una métrica híbrida formada por una medida intra-modular, como las medidas de Halstead y McCabe y una medida inter-modular, la medida de interconectividad [Henr88].

Hasta esta época, la mayor parte de las investigaciones se dirigían hacia la medida de la complejidad basada en el análisis del flujo de control del programa. En 1982, Weiser presentó el concepto de *slice* o 'parte de un programa' [Weis82]. Sus investigaciones demostraron la importancia de las dependencias entre los datos, pudiendo ser una medida de este tipo una herramienta útil para el desarrollo de software fiable. Basándose en Weiser, en 1986 Longworth presentó su medida de cohesión [Long86], y a partir de 1991, las propuestas de Ott se consideran fundamentales como medidas de la cohesión funcional de un sistema [Ott93, Biem94].

Durante los 80, surgieron algunas investigaciones que aplicaban la Teoría de la Información a las métricas del software [Berl80, Coul87]. Entre ellas, algunas adaptaban la noción de *entropía* para ser usada como medida de complejidad [Moha81, Davi88, Robi89, Harr92]. Estas propuestas han tenido poca expansión.

En la década de los 80, se realizaron diferentes investigaciones en el Centro de Desarrollo del Aire de Rome (EEUU), perteneciente a las Fuerzas Aéreas. En este instituto, se intentó encontrar una relación entre las medidas del software y los atributos de calidad del mismo (facilidad de uso, de mantenimiento, de prueba,...). El objetivo de estas investigaciones era desarrollar un marco de calidad del software que cuantifique las técnicas de usuario y de gestión que afectan a la calidad del software como producto [RADC84].

Otra organización que comenzó hace varios años la investigación relacionada con la Medición del software es la NASA, a través de su Laboratorio de Ingeniería del Software [NASA84, NASA90]. También se deben mencionar los esfuerzos de IEEE para proponer medidas estándarizadas [IEEE89, IEEE89a, IEEE92, IEEE92a] y los programas de medición que se están llevando a cabo bajo los auspicios del SEI (Software Engineering Institute) de la Universidad de Carnegie Mellon [Mill88, Park95, SEI95, Rozu96, Rozu94, Goet92, CMU96, McAn93, Rifk91, Arch95, Park94, Carl92], además de la labor de ANSI e ISO al generar estándares para la calidad del software que exigen la existencia de un proceso de medición en las organizaciones [ANSI91].

En cuanto a la situación de la Medición del Software en Europa, en 1986 comenzó un proyecto en el Reino Unido denominado *Medición del Software Basada en la Estructura* (Proyecto Alvey SE/069). Su investigación se centró en la modelización formal, el análisis y la medición de la estructura del software [Elli86];

parte de los resultados se exponen en [Fent91]. Desde 1989 hasta 1992, se llevó a cabo el proyecto METKIT, promovido por la Comunidad Europea, bajo el programa ESPRIT. El objetivo era concienciar e incrementar el uso de las métricas en la industria europea, generando material educativo tanto para el entorno industrial como para el académico [Fent91].

Otros proyectos ESPRIT relacionados con las medidas del software son: AMI (Applications of Metrics in Industry) de 1990 a 1992 [Pulf96], MUSIC (Metrics for Usability Standards in Computing) de 1990 a 1993, MUSE (Software Quality and Reliability Metrics for Selected Domains: Safety Management and Clerical Systems) de 1987 a 1990, PYRAMID (Promotion of Metrics) de 1990 a 1992, cuyo objetivo era mejorar la calidad y la productividad del desarrollo y mantenimiento de sistemas de software europeos usando métodos cuantitativos, COSMOS (Cost Management with Metrics of Specification) de 1989 a 1994 y MERMAID (Metrications and Resource Modelling Aid) de 1988 a 1992.

Por último, queda mencionar que hasta mediados de los 80 la tendencia era aplicar métricas intra-modulares en la fase de codificación del software. Sin embargo, desde mediados de los 80, se está prestando más atención a la medición en las primeras fases del ciclo de vida, como la fase de diseño preliminar y sobre todo, la fase de especificación de requisitos. Un modelo sencillo del ciclo de vida del software podría ser el de la Figura 2.2.

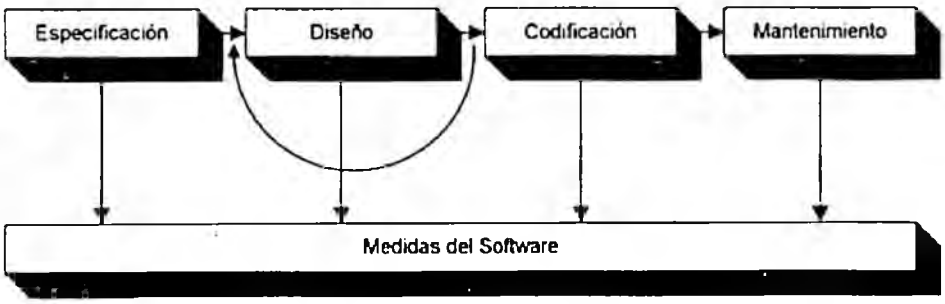

Figura 2.2 Fases de Ciclo de Vida del Software y la Medición en cada fase.

En la Figura 2.2, se ilustra que las métricas del software deberían aplicarse en todas las fases del ciclo de vida. El objetivo de aplicar medidas en las fases iniciales del ciclo de vida es poder mejorar el desarrollo del software en la fase de diseño mediante un proceso de realimentación controlado por dichas medidas, obtener una mejor implementación del software en la fase de codificación y conseguir un mantenimiento barato y menos complicado.

A continuación, se describen las métricas del software más conocidas y que más extensión han tenido en la industria: las *Líneas de Código*, la *Complejidad Ciclomática* de McCabe y la *Ciencia del Software* de Halstead.

2.3.4.1 Las Líneas de Código

Se considera la primera medida del software. Es una medida muy polémica pero que todavía se usa hoy en día [Park92]. En 1974, Wolverton [Wolv74] llevó a cabo uno de los primeros intentos de medir de un modo formal la productividad de los programadores utilizando líneas de código. Propuso como medida de productividad la instrucción-objeto/persona-mes y sugirió lo que él consideraba debían ser unos índices de productividad típicos.

La base de las LOC es que la longitud de un programa se puede utilizar como una medida predictiva de determinadas características del mismo como la fiabilidad y la facilidad de mantenimiento [Shep93]. Sobre todo debido a su simplicidad y carencias, es una medida sujeta a severas críticas. Las líneas de código reales y las estimadas se utilizan para realizar estimaciones de productividad, tal y como se describen en Walston y Felix [Wals77] en las fases de definición y análisis de rendimiento de los desarrollos de software.

Es una métrica muy fácil de capturar, pero muy limitada: sólo proporciona una medida del tamaño de un programa. Como no se centra en absoluto en distinguir los distintos componentes de una línea de código, no proporciona ninguna noción de complejidad intrínseca de cada línea y es obvio que existen unas más complejas que otras [Cont86].

Uno de los inconvenientes principales de esta métrica es la falta de acuerdo sobre lo que es una línea de código. Las diferencias se centran en si se consideran línea de código a los comentarios, las líneas en blanco, las sentencias no ejecutables, las múltiples sentencias en una línea, las múltiples líneas por sentencia,... así como la cuestión de cómo contabilizar las líneas de código reutilizadas [Mill88]. Una definición muy extendida es la de Conte *et al.* [Cont86] en la que no se consideran líneas de código los comentarios y las líneas de blanco, y sin embargo, se incluyen las cabeceras de los programas, las declaraciones, las sentencias ejecutables y no ejecutables. Se les suele denominar *Líneas de Código Efectivas* (ELOC - Effective Lines of Code) y hay otra denominación: *Instrucciones Fuente Entregadas* (DSI - Delivered Source Instructions) [Fent91]. Conte considera que este tipo de líneas no suponen esfuerzo y podrían condicionar a los programadores, ya que provocarían desviaciones si se utilizaran como medidas de productividad.

Este es otro inconveniente de las LOC: si se valora la productividad de una persona por el número de líneas que entrega, utilizará gran número de comentarios o elaborará código muy extenso, cuando la misma funcionalidad se puede codificar con menos líneas.

Otro problema más que se presenta a la hora de utilizar las LOC como métrica predictiva de estimación de esfuerzo de desarrollo es su disponibilidad en el tiempo: las LOC se obtienen en las fase de codificación, cuando ya no hace falta realizar una estimación de esfuerzo. Se pueden utilizar como métricas de mantenimiento o de facilidad de prueba.

Por último, también hay que considerar que esta métrica es dependiente del lenguaje de programación utilizado y que no es fácil aplicarla con los lenguajes de programación modernos, generadores de aplicaciones y 4GL.

2.3.4.2 La Complejidad Ciclomática de McCabe

McCabe [McCa76] derivó una medida de la complejidad del software basándose en la Teoría de Grafos y utilizando la definición del número ciclomático. Interpretó el número ciclomático como el mínimo número de caminos independientes en un grafo de flujo (representación simplificada de un diagrama de flujo), considerando este mínimo número de caminos como la complejidad de un programa (*Complejidad Ciclomática*). La estrategia consiste en visualizar un programa como un grafo dirigido. La Complejidad Ciclomática se define como:

$$V(G) = a - n + 2$$

donde *V* es la Complejidad Ciclomática, *G* es el grafo dirigido del programa, *a* es el número de aristas del grafo y *n* es el número de nodos del mismo.

McCabe propuso medir la complejidad de un programa usando *V(G)*, controlar el 'tamaño' de los programas utilizando un límite superior para *V(G)* (de hecho, sugirió $V(G) = 10$ como límite superior, recomendando la descomposición de aquellos programas que lo superaran) y utilizar *V(G)* como la base para elaborar un método de prueba.

También propuso una variación: la *Complejidad Esencial*. Esta medida sirve para medir el grado de estructuración de un programa. Para ello, en el grafo de flujo todas las construcciones debe quedar reducidas a secuencias, selecciones e iteraciones.

2.3.4.3 La Ciencia del Software de Halstead

Se centra en el código fuente de los programas. Halstead demostró que el esfuerzo estimado o el tiempo de programador se pueden expresar como una función del número de operadores, de operandos y de usos [Hals76]. Este método se ha utilizado en numerosas organizaciones incluyendo IBM, General Electric y General Motors.

El modelo de Halstead se fundamenta en que un programador hace uso de dos conjuntos al construir un programa: operadores y operandos. Los componentes de la Ciencia del Software son las *propiedades medibles de los algoritmos*:

η_1 = número de operadores únicos o distintos.

η_2 = número de operandos únicos o distintos.

N_1 = uso total de todos los operadores.

N_2 = uso total de todos los operandos.

A partir de estos componentes, Halstead derivó las siguientes propiedades:

Tamaño del Vocabulario → $\eta = \eta_1 + \eta_2$

Longitud del Programa → $N = N_1 + N_2$

Volumen de Programa → $V = N * \log_2 \eta$

Dificultad del Programa → $D = (\eta_1 * N_2) / (2 * \eta_2)$

Esfuerzo → $E = V * D$

Aunque su uso está muy extendido, la Ciencia del Software ha sido dejada de lado en los últimos años debido a que su enfoque es puramente empírico y carece de las rigurosas propiedades que toda métrica de complejidad debería poseer [Card87].

Se han encontrado errores en los datos que Halstead utilizó para validar sus métricas y toda su investigación está siendo cuestionada por no poseer los controles adecuados en la experimentación (Halstead murió a finales de los 70 y es una lastima que no pueda defender sus medidas hoy en dia).

2.4 LAS METRICAS EN EL PARADIGMA ORIENTADO A OBJETOS

2.4.1 Las Métricas Tradicionales y la Filosofía Orientada a Objetos

Como hemos expuesto previamente, las métricas del software han evolucionado a lo largo de los últimos veinte años. Algunos creen que estas métricas tradicionales se pueden aplicar en el Paradigma Orientado a Objetos [Tega92i. Esta idea viene avalada por tres puntos:

1. Estas métricas ya existen.
2. Existen evidencias empíricas que dan soporte a estas métricas utilizadas en los sistemas estructurados.
3. Tanto los investigadores como los profesionales están familiarizados con ellas.

Estos argumentos parecen meras excusas para evitar el cambio. Además, debe tenerse en cuenta que cada métrica se desarrolló en una época concreta con una tecnología determinada: la Ciencia de Software se centra en las unidades básicas de lenguajes como el Ensamblador y el Fortran; la Complejidad Ciclomática y derivados se ajustan a las necesidades de la Ingeniería del Software de su época la Programación Estructurada, las medidas de Cohesión, como las de Ott, asumen que un módulo es un procedimiento o función, etc. [Melt96].

Cuando los conceptos de la Orientación a Objetos aparecieron por primera vez como una de las claves para terminar con la llamada *Crisis del Software*, la actitud de los ingenieros de software fue de escepticismo. A medida que pasa el tiempo, los conceptos, lenguajes de programación, herramientas CASE para la Tecnología Orientada a Objetos y los beneficios que se obtienen de su uso se van extendiendo. Pero ¿qué diferencia a un Sistema Orientado a Objetos de un Sistema Tradicional? Las diferencias se pueden resumir en varios puntos [Hend96]:

1. Un Ciclo de Vida Recursivo/Iterativo.
2. La Reusabilidad.
3. La Estructuración del Sistema: subsistemas, clases, métodos.

4. La Comunicación de Mensajes entre iguales (a diferencia del grado de entrada y salida).

5. Más Niveles de Abstracción.

6. Estructuras de Herencia.

7. Características Internas (con respecto a McCabe y el grado de entrada y salida).

Aunque se han hecho intentos de aplicar las métricas tradicionales al Paradigma Orientado a Objetos, existen dos aspectos importantes a considerar.

En primer lugar, estas métricas tradicionales no hacen referencia directa a las propiedades o características de un sistema Orientado a Objetos. Estas propiedades incluyen al propio concepto de objeto, clases, atributos, métodos, la encapsulacion, el paso de mensajes, el polimorfismo y la herencia. Estas propiedades tienen influencia sobre la complejidad, la facilidad de mantenimiento, la reusabilidad y la facilidad de extensión de un producto de software Orientado a Objetos. El enfoque Orientado a Objetos combina datos y operaciones en una única entidad, mientras que el enfoque tradicional separa los datos de las operaciones. El enfoque tradicional está claramente orientado a la función. Por lo tanto, si se desea utilizar las medidas tradicionales, existen tres posibilidades:

- Adaptación para que contemplen alguna de las propiedades anteriormente mencionadas [Grah95, Thom94, Bind94]

- Aplicación únicamente en aquellos ámbitos comunes con el desarrollo tradicional, por ejemplo, para medir la complejidad de los métodos [Morr89, Lore94], aunque existen motivos para pensar que estas medidas son poco significativas puesto que si se usan LOC como medida de tamaño, los métodos OO suelen ser pequeños y si se usa la Complejidad Ciclomática de McCabe, generalmente tendrán un V(G) entre 1 y 7, siendo la media de la complejidad un valor cercano a 0, lo cual obviamente, es absurdo [Kole93].

- Utilización de métricas generales que han sido diseñadas para su uso tanto en sistemas tradicionales como OO y que han sido probadas en ambos entornos [Cant95, Pant95].

En segundo lugar, las medidas tradicionales del software carecen de una base teórica firme y de las propiedades que toda métrica debería poseer. Estas métricas han sido evaluadas gracias a datos históricos empíricos. Este método fundamenta las métricas en la experiencia obtenida en distintos proyectos, en la experiencia de gestores, diseñadores, en los conocimientos de los programadores, lenguajes de

implementación, entornos de desarrollo, etc. Ninguno de los elementos anteriores es suficientemente sólido como para establecer una base teórica firme. Para poder comparar, contrastar y utilizar métricas Orientadas a Objetos es necesario un marco formal. Este marco formal está formado por un conjunto de propiedades que identifican los puntos que una medida debe cumplir [Weyu88].

2.4.2 Métricas del Software Orientado a Objetos: Pasado y Presente

El estudio de las Métricas del Software en el Paradigma Orientado a Objetos se puede considerar una línea de investigación relativamente nueva, que todavía carece de métodos empíricos y cuantitativos bien definidos y de un enfoque teórico sólido. Dentro de esta área de estudio, se han propuesto distintas métricas de tamaño y complejidad, que van desde la complejidad de los métodos y atributos hasta el acoplamiento entre los mismos, pasando por la cohesión de métodos, clases, la profundidad de la jerarquía, el número de descendientes, de ancestros, etc.

A continuación, se exponen los distintos enfoques, resultados e inconvenientes de las investigaciones que se han realizado a lo largo de los últimos años.

2.4.2.1 La Ley de Demeter de Lieberherr

Lieberherr *et al.* realizaron un intento de formalizar las reglas que se deben cumplir para poseer un estilo de programación orientada a objetos correcto [Lieb88]. Se basaron en los conceptos de acoplamiento y cohesión que se utilizan en el desarrollo tradicional. Existen dos *leyes* relacionadas con el estilo de programación correcto llamadas *Ley de Demeter Débil* y *Ley de Demeter Fuerte*. Ambas están relacionadas con la definición de las variables de instancia de una clase concreta. Si se cumple la ley fuerte, cualquier cambio que se realice sobre las estructuras de datos afectará únicamente a los métodos declarados en las clases que definen dichas estructuras. Un cumplimiento riguroso de la ley fuerte supone una penalización en términos de rendimiento y una tendencia al aumento del tamaño del sistema. La ley débil supone un compromiso entre un ocultamiento de información estricto y el rendimiento. Los autores proporcionan ejemplos de buenas y malas prácticas de programación, pero no se proponen métricas concretas que puedan servir de forma predictiva para la gestión de un proyecto. Aparentemente no se pueden automatizar, no presentan una base teórica ni hay datos empíricos de su validez.

2.4.2.2 Las Métricas de K. Morris

Uno de los primeros investigadores en materia de medición de software Orientado a Objetos fue Kenneth Morris (Morr89), quien propuso un conjunto de métricas basadas en un estudio empírico de la literatura, observaciones y entrevistas con miembros de un equipo de desarrollo de un proyecto OO. Morris no proporcionó ni unos fundamentos teóricos ni trató de implementar sus métricas para demostrar su factibilidad o facilidad de uso. Se supone que estas métricas son adecuadas para medir varias características de la OO, a las que Morris denominó *variables de impacto en la productividad*. Las métricas de Morris son las siguientes:

- Métodos por clase

  Se cuenta el número total de métodos en una clase. Es una medida tanto de complejidad como de tamaño. Según Morris, la facilidad para extender el sistema disminuye si el número de métodos por clase es excesivo. Además esto probablemente complica las pruebas debido al tamaño y complejidad crecientes del objeto.

- Dependencias relativas a la Herencia

  Es la máxima profundidad en el árbol de herencia de una clase. Es equivalente a la métrica *Profundidad del Arbol de Herencia* del conjunto de Chidamber (ver sección 2.4.2.3). La profundidad del árbol de herencia probablemente sea más favorable que la anchura en términos de Reusabilidad. Los árboles profundos promueven la compartición de métodos mejor que los anchos. Sin embargo, un árbol profundo puede ser más difícil de probar que uno ancho. Además, el grado de comprensión también puede disminuir si el número de niveles de herencia es creciente.

- Grado de Acoplamiento entre Objetos

  A mayor acoplamiento entre objetos, mayor posibilidad de complicar el mantenimiento debido a las interconexiones entre objetos y mayor posibilidad de generar errores en el desarrollo. Cuanto más independientes sean los objetos, mayor será la probabilidad de reutilizarlos en la misma u otra aplicación. Además, los objetos no acoplados deberían ser más fáciles de extender que otros con un alto grado de dependencias de uso, debido a su bajo grado de interacción.

  Se puede medir de dos formas diferentes:

  a) la media de dependencias de uso por objeto:

$$Media\_dependencia\_usos\_por\_objeto = \frac{N^{o}\_total\_arcos}{N^{o}\_total\_objetos}$$

donde los *arcos* son flechas en un grafo de usos de objetos.

b) el máximo número de dependencias

$$Maximo\_dependencias\_usos = Max\{N^{o}\_arcos\_uso\}$$

donde *arcos_uso* son arcos ligados a un objeto en un grafo de usos.

- Grado de Cohesión de Objetos

Mide el grado de relación existente entre los atributos y los métodos. Se define como el ratio del grado de entrada total para todos los objetos y el número total de objetos.

$$Grado\_Cohesión\_Objetos = \frac{Grado\_entrada\_total}{N^{o}\_total\_objetos}$$

Los objetos con menores dependencias sobre los datos de otros objetos es posible que se puedan reutilizar mejor. Sin embargo, una baja cohesión puede producir niveles de errores crecientes en el desarrollo y añade complejidad que se traduce en una reducción de la fiabilidad del producto producido. Considerando el propósito de la métrica, la fórmula definida por Morris no es sólida: la comunicación entre clases refleja el grado de acoplamiento, no la cohesión.

- Efectividad de la Librería de Objetos

Si los objetos están siendo diseñados para ser reutilizados, los efectos deberían aparecer en las estadísticas de uso de la librería de objetos. Existen tres implementaciones para esta métrica:

a) la existencia o no de un objeto de librería.

b) el porcentaje de objetos de librería en una aplicación, que es el número de objetos de librería dividido entre el número total de objetos en la aplicación y multiplicado por 100.

$$Porcentaje\_objetos\_libreria = \frac{N^{o}\_total\_objetos\_libreria}{N^{o}\_total\_objetos} \times 100$$

c) el número medio de veces que un objeto de librería es reutilizado, siendo el ratio del número total de reutilizaciones de objetos y el número total de objetos de librería.

$$Media\_reusabilidad\_objeto = \frac{N^{o}\_total\_reutilizaciones\_objetos}{N^{o}\_total\_objetos\_libreria}$$

- Efectividad de la Factorización

Morris la define como el ratio del número de métodos únicos y el número total de métodos.

$$Efectividad\_factorización = \frac{N^{o}\_métodos\_unicos}{N^{o}\_total\_objetos}$$

El valor más alto posible es 1, que implica que cada método en la aplicación es único. El objetivo es reducir el número de implementaciones para un método en una aplicación. Así, disminuye la probabilidad de que se comentan errores durante la codificación.

- Grado de Reusabilidad de los Métodos Heredados

El simple hecho de definir métodos que se puedan reutilizar vía herencia no garantiza que esos métodos se reutilicen en realidad. Son dos las métricas que se definen:

a) el porcentaje de usos de métodos potenciales realmente reutilizados (PP): es el ratio del número total de usos de métodos reales y el número total de usos de métodos potenciales, multiplicado por 100.

$$PP = \frac{N^{o}\_total\_usos\_métodos\_reales}{N^{o}\_total\_usos\_métodos\_potenciales} \times 100$$

b) el porcentaje de usos de métodos potenciales redefinidos (PM): es el ratio del número total de métodos redefinidos y el número total de usos de métodos potenciales, multiplicado por 100.

$$PM = \frac{N^{o}\_total\_métodos\_anulados}{N^{o}\_total\_usos\_métodos\_potenciales} \times 100$$

Esta métrica es similar a la Efectividad de la Factorización.

- Complejidad Media de los Métodos

  Se calcula mediante la Complejidad Ciclomática de McCabe dividido entre el número total de métodos de la aplicación.

$$Complejidad\ \ media = \frac{\sum_{i=1}^{i=total\_metodos} V(G)_i}{N^o\ \ total\ \ metodos\ \ aplicación}$$

  donde $V(G) = a - n + 2$.

  Cuanto más complejos sean los métodos, más difícil será de mantener la aplicación, llevará a grados de comprensión menores y dificultará la prueba.

- Granularidad de la Aplicación

  La implementación de esta métrica se define como la proporción entre el número total de objetos y el número total de puntos de función.

$$Granularidad\_aplicación = \frac{N^o\_total\_objetos}{N^o\_total\_puntos\_función}$$

  El concepto que subyace en esta métrica es que una aplicación construida con objetos de una granularidad más fina (por ejemplo, un número de funciones por objeto menor) es más fácil de mantener porque los objetos serán más pequeños, menos complejos, más fáciles de reutilizar, comprender y analizar..

2.4.2.3 La Colección de Métricas de Chidamber & Kemerer

Chidamber y Kemerer han propuesto un conjunto completo de métricas para gestionar proyectos de desarrollo de software OO [Chid91, Chid94]. Sus métricas se derivan de los principios ontológicos de Bunge [Bung77]. Bunge veía el mundo como un conjunto de individuos substanciales, cosas y conceptos que se definen mediante las propiedades que poseen. Basándose en estos conceptos, los autores construyeron definiciones que se adaptan a la Filosofía Orientada a Objetos y a continuación, definieron las propiedades que pretendían medir en términos ontológicos. El último paso fue crear las métricas que miden estas propiedades. Los autores se centraron en tres propiedades del diseño orientado a objetos:

1. la definición de los objetos y su relación con otros objetos.

2. los atributos y las propiedades de los objetos.

3. la comunicación entre objetos.

Proporcionaron unos fundamentos teóricos utilizando las definiciones ontológicas para cada métrica que especificaban. Evaluaron cada una de las métricas contra las propiedades de Weyuker [Weyu88]. A continuación, las validaron empíricamente.

Los autores definieron seis métricas que son las siguientes:

- Métodos Ponderados por Clase (WMC)

  Mide la complejidad de un objeto individual, perteneciente a la clase C, con métodos M_1, ... M_n, que están definidos en C, siendo c_1, ... c_n, la complejidad de los métodos. Consideran que la complejidad de todos los métodos es igual a la unidad y por tanto, el valor de la métrica es igual a *n*, siendo este el número de métodos. Así se determina la complejidad total de un objeto.

- Profundidad del Arbol de Herencia (DIT)

  Mide el nivel de profundidad de una clase en el árbol de herencia. Para una clase con herencia múltiple será el camino del árbol más largo. Es razonable asumir la existencia de alguna relación de herencia entre las clases de un sistema, debido a que la herencia es el método por el que las clases existentes se reutilizan. A mayor profundidad de una clase en la jerarquía de herencia, mayor nivel de complejidad. Esta métrica puede usarse como un indicador de la reusabilidad a través de la herencia.

- Número de Hijos (NOC)

  Proporciona el número de subclases inmediatas en la jerarquía de clases. También es un indicador de reusabilidad, ya que indica cuántas subclases pueden heredar las propiedades de una superclase. Esta métrica asume la existencia del mecanismo de herencia en la estructura de clases del sistema.

- Acoplamiento entre objetos (CBO)

  Esta métrica se basa en la relación entre dos objetos. Si están relacionados, actúan uno sobre el otro. El número de clases con la que una dada está acoplada es una medida del grado de interdependencia entre clases. Si un objeto de una clase, puede afectar a la historia de un objeto de otra clase, se dice que esas clases están acopladas. Se determina analizando los accesos de los métodos o variables de instancia de otro objeto. Un grado bajo de interdependencia sugiere una mayor reusabilidad, mientras que un alto grado

de interdependencia indica una mayor necesidad de prueba, al haber más métodos que llaman y más variables implicadas.

- Respuesta de una Clase (RFC)

  Representa el número de elementos en el conjunto de respuestas de una clase. Este conjunto de respuestas se puede considerar como el conjunto de abarca a los métodos disponibles en una clase. Incluye tanto los métodos de la propia clase, como los métodos de otras clases (incluyendo la herencia y las relaciones de uso). Esta métrica se puede considerar como un método de medir la comunicación entre clases.

- Escasez de Cohesión en los Métodos (LCOM)

  Esta métrica se centra en el grado de similitud entre métodos. Las clases cohesivas contendrán métodos que comparten variables de instancia. La cohesión es una característica deseable ya que fomenta la propiedad de encapsulamiento. Si todos los métodos de la clase no tienen una variable en común, el valor de la métrica es 0 - no presenta cohesión. Si algunos métodos en una clase comparten algunos atributos comunes, existe cohesión en esa clase. La falta o escasez de cohesión entre los métodos de una clase, indica que la clase debería ser descompuesta en subclases o ser rediseñada.

  Las métricas anteriores se pueden clasificar en tres categorías:

1. Definición de Objeto: Profundidad del Arbol de Herencia, Número de Hijos y Métodos Ponderados por Clase.

2. Atributos: Escasez de Cohesión en los Métodos y Respuesta de una Clase.

3. Comunicación: Acoplamiento entre Objetos y Respuesta de una Clase.

Chidamber y Kemerer llegaron a la conclusión que todas las metodologías OO comparten la misma base ontológica para los objetos que describen. Por lo tanto, son independientes del método utilizado (y también del lenguaje de programación). Se pueden utilizar una vez se dispone de un diseño más o menos preliminar. Sin embargo, no cubren todos los conceptos de la Filosofía Orientada a Objetos.

Esta colección de medidas, siendo un punto de referencia para todos los investigadores del área, ha sido duramente criticada. CBO ha sido criticada por considerar que las distintas relaciones de acoplamiento tiene la misma importancia [Bink96, Hitz96], LCOM presenta serias anomalías [Hitz96, Hend96], RFC no es independiente de CBO [Hitz95], las definiciones en las que basarse para calcular las

medidas son imprecisas [Chur95, Chur95a] y ninguna de ellas se puede utilizar de forma predictiva, puesto que no son de escala de ratio, según Zuse.

Incluso se ha llegado a criticar el hecho de que la base formal de las medidas sea la Ontología de Bunge. Segun Graham [Grah96], la Ontología sufre de graves defectos filosóficos: es atómica. Concibe los elementos como reducibles a componentes irreducibles; sin embargo, la identidad de un objeto es independiente de sus propiedades, todas ellas pueden cambiar sin alterar la identidad intrínseca del objeto.

2.4.2.4 Las Métricas de Lorenz y Kidd

Mark Lorenz y Jeff Kidd [Lore94] utilizaron un enfoque puramente práctico para obtener sus métricas (a diferencia del uso de pruebas y teorías matemáticas de Chidamber). Utilizan un enfoque un tanto *precipitado* en la búsqueda de soluciones rápidas para mejorar el diseño, la reusabilidad y la precisión de las estimaciones. Sus objetivos son promocionar el uso de las métricas entre los profesionales del software, mejorar factores de calidad como la fiabilidad, la facilidad de mantenimiento, de expansión y la reusabilidad y mejorar la gestión de proyectos.

Estas métricas se clasifican en dos categorías: de proyecto y de diseño. Definieron ocho métricas de proyecto, que cubren el tamaño de la aplicación, del equipo de desarrollo y la planificacion. Las métricas de diseño se dividen en varios grupos: el tamaño de los métodos, los aspectos internos de los métodos, el tamaño de las clases, la herencia de clase y de método, y los aspectos internos y externos de las clases.

Esta colección de métricas se aplicó a cinco proyectos, implementados en C++ y Smalltalk y con un tamaño variable de 60 a 700 o más clases. Se desarrollaron en periodos de entre 6 meses y 2,5 años, involucrando desde 2 a 25 personas. No se tomaron todas las métricas en todos los proyectos.

Los autores analizaron las siguientes características para cada métrica: resultados estadísticos con los datos de los proyectos, influencia de factores como la disponibilidad del usuario final, interfaz de usuario, las métricas relacionadas, umbrales y valores heurísticos que marcan un rango válido o identifican anomalías. También sugirieron las acciones a seguir ante la detección de un problema por desviaciones del umbral.

La principal crítica a estas medidas es que no hay un modelo formal bajo ellas. En realidad, son un conjunto de medidas empíricas que cuantifican propiedades

estáticas y por lo tanto, se pueden utilizar con fines de *valoración* o de comparación de características e incluso para *diagnóstico*, pero *no como medidas predictivas* [Hend96].

2.4.2.5 La Colección de Métricas de Chen y Lu

Chen y Lu [Chen93] publicaron un nuevo conjunto de métricas para el método de desarrollo de Grady Booch [Booc94, Booc96], presentando los resultados de un experimento que demostraba que dichas métricas eran factibles. El método de la captura de datos para la selección de medidas se basaba en GQM de Basili *et al.* [Basi88, Basi94]. Estas métricas se desarrollaron para medir la complejidad con el objetivo de servir de ayuda en la fase de diseño. El conjunto de métricas propuesto es el siguiente:

- Métrica de Complejidad de las Operaciones

  Mide la complejidad de los métodos u operaciones de una clase. Es la suma de los valores de complejidad de cada método de una clase. Los valores se derivan de una tabla de rangos que varían de 'Nulo' a 'Muy baja', 'Baja', 'Nominal', 'Alta', 'Muy Alta' hasta 'Exta Alta'. Estas categorías se corresponde a rangos de valores de 0, 1-10, 11-20, 21-40, 41-60 , 61-80 y 81-100, respectivamente.

- Métrica de Complejidad de los Argumentos de las Operaciones

  Consta de la suma de los valores de las complejidades de los argumentos o de los atributos de cada operación en una clase. Los valores se obtienen de una tabla de tipos de datos de argumentos y sus correspondiente valores. Las entradas de la tabla van desde 'Booleano o Entero' con un valor de 0, hasta 'Fichero' con un valor de 10. Los valores asignados no son inflexibles: una organización que decida utilizar la tabla, deberá calibrarla para sus necesidades y requisitos particulares.

- Métrica de Complejidad de los Atributos

  Es la suma de los valores de complejidad de los atributos para cada atributo de una clase. Los valores que se utilizan son los mismos que en la Métrica de Complejidad de los Argumentos de las Operaciones.

- Métrica del Acoplamiento de Operaciones

  Es la suma del número de operaciones que acceden a otras clases, más el número de operaciones que se acceden desde otras clases, más el número de operaciones que cooperan con otras clases.

- Métrica de Acoplamiento de Clase

  Mide el acoplamiento entre clases. Es la suma del número de accesos a otras clases, por otras clases y el número de clases que cooperan. Esta métrica es similar a la anterior, con la salvedad de que hace referencia a la clase, en vez de a las operaciones.

- Métrica de Cohesión

  Mide el grado de cohesión entre las operaciones de una clase. Es el ratio del número de conjuntos disjuntos de argumentos y el número de operaciones de una clase. Cuanto más pequeño sea el número, más cohesión existirá entre operaciones.

- Métrica de Jerarquía de Clases

  Es la suma de la profundidad de una clase en el árbol de herencia, más el número de subclases, más el número de superclases directas más el número de operaciones locales o heredadas disponibles.

- Métrica de Reusabilidad

  El valor de esta métrica es 1 si se reutiliza la clase desde otro proyecto y 0, en caso contrario. La reusabilidad puede ser parcial o total.

Basándose en las métricas definidas, los autores realizaron un experimento con el fin de recoger datos y validar las métricas. Se utilizaron seis temas distintos y dos proyectos de software experimentales. Seleccionaron la metodología de Booch y se tomaron los datos a partir del diseño manualmente; los participantes utilizaron las métricas de los autores basándose en las tablas de rangos. Se contó con expertos para la evaluación de los diseños, con independencia de las métricas definidas. Se puntuó cada clase sobre nueve aspectos de evaluación, con valores desde 0, representando a 'Muy Pobre' hasta 10, para 'Muy Bien'. Se aplicó un modelo de regresión estadística a los resultados, siendo la suma de los puntos de cada clase el objetivo del modelo.

Usando las puntuaciones de los expertos como el valor de complejidad definitiva, se analizaron estadísticamente las métricas para cada clase mediante el análisis de regresión multivariable. La validación empírica tuvo los mismos problemas que suelen tener las investigaciones sobre métricas en general: un ámbito de datos limitado y un número de participantes pequeño. El tamaño de los dos proyectos no parecía ser muy representativo del tamaño de los proyectos que habitualmente se encuentran en la industria. Además, basarse en evaluaciones de expertos para valorar la complejidad es una actividad cuestionable, ya que no existen estándares

para juzgar o valorar su conocimiento experto. Por otra parte, utilizaron valoraciones de pesos subjetivas que son escalas Likert, que deberían evitarse porque no tienen validez científica [Hend96].

2.4.2.6 Las Métricas de Binder

Binder investigó en este área con el objetivo de formular un métrica de la facilidad de prueba de un sistema OO [Bind94]. En su modelo de medición de la facilidad de prueba, incluyó las seis métricas de Chidamber [Chid94] y la Ley de Demeter de Lieberherr [Lieb88]. Además, añadió varias métricas que no llegó a desarrollar basadas en la Teoría de la Medición y para las que no ofreció ninguna validación empírica. Son las siguientes:

- Porcentaje de Elementos Públicos y Protegidos

  Mide el porcentaje de todos los miembros de datos (como se utilizan en C++) que son públicos y protegidos. Los miembros de datos protegidos y privados son los mecanismos para la implementación del Ocultamiento de Información que permiten el Encapsulamiento. Un valor alto de esta métrica aumenta el riesgo de sufrir efectos laterales entre las clases.

- Accesos Públicos a Miembros de Datos

  Representa el número de accesos externos a los miembros de datos públicos o protegidos de una clase. De manera similar a la métrica anterior, un valor alto de esta métrica incrementará el espacio de estado de un programa, así como también los requisitos de prueba.

- Número de Clases Raíz

  Es el número de jerarquías de clases distintas entre el conjunto de clases de un programa.

- Grado de Entrada (Fan In)

  Es el número de clases de las que una clase concreta hereda. Sólo es mayor que 1 cuando hay Herencia Múltiple.

- Porcentaje de Llamadas No Sobrecargadas

  Es el porcentaje de llamadas realizadas a métodos no sobrecargados. Los métodos sobrecargados se caracterizan por tener múltiples implementaciones, que se distinguen por tener signaturas únicas. Dichas signaturas únicas

permiten eliminar la ambigüedad en la llamada al método en tiempo de ejecución.

- Porcentaje de Llamadas Dinámicas

  Es el porcentaje de receptores de mensajes que se determinan en tiempo de ejecución.

- Rebotes

  Es el número de caminos de tipo *yo-yo*. Un camino *yo-yo* es el que rastrea la jerarquía de una clase hacia arriba y hacia abajo debido a los enlaces dinámicos. Esta métrica se puede aplicar tanto a la jerarquía de clases como a un sistema en ejecución.

- Complejidad de Clase

  Es la Complejidad Ciclomática del grafo de control formado por la unión de todos los grafos de control de todos los métodos con un diagrama de transición de estados de la clase.

- Número Nominal de Métodos

  Es el número de métodos por clase. Se sugiere un límite superior de 20 métodos por clase. Está métrica también puede expresarse como la suma de otras dos métricas: Número Nominal de Funciones y Número Nominal de Procedimientos. La diferencia entre ambos es que un procedimiento cambia el estado de un objeto o sistema, mientras que una función meramente informa sobre un valor, sin ningún efecto lateral.

- Número Total de Métodos por Clase

  Es similar al Número Nominal de Métodos, con la inclusión de todos los métodos heredados. Al igual que el Número Nominal de Métodos, también se puede descomponer en la suma de dos métricas: Número Total de Funciones y Número Total de Procedimientos.

A partir de estas métricas, Binder construye un modelo de implementación para la facilidad de prueba en un sistema OO, que es parte de un modelo más general.

2.4.2.7 Las Métricas de Sharble y Cohen

Curiosamente, el objetivo de la investigación de Sharble y Cohen [Shar93] no fue proponer un conjunto de métricas orientadas a objetos.

Seleccionaron las métricas de Chidamber y Kemerer [Chid94], ya que estas se basan en la Teoría de la Medición y no están condicionadas por ninguna metodología concreta para llevar a cabo un desarrollo orientado a objetos. Utilizaron dichas métricas para comparar la eficacia de dos metodologías distintas de desarrollo de software orientado a objetos.

Las métricas que se utilizaron fueron, por lo tanto, los Métodos Ponderados por Clase (WMC), la Profundidad del Arbol de Herencia (DIT), el Número de Hijos (NOC), el Acoplamiento entre objetos (CBO), la Respuesta de una Clase (RFC) y la Escasez de Cohesión en los Métodos (LOC). Además, completaron este conjunto de métricas añadiendo tres más:

- Atributos Ponderados por Clase

  Es similar a la métrica Métodos Ponderados por Clase en el hecho de que se valoran los atributos de una clase en función de sus tamaños dependiendo de que sean enteros, de tipo caracter, vectores, etc...

- Número de *Tramps*

  Un *tramp* es un parámetro que forma parte de la signatura de un método, pero al que no se hace referencia dentro de dicho método (este término está relacionado con el Acoplamiento entre módulos en un sistema estructurado tradicional [Page88]). Dado un método en particular, esta métrica representa al número de argumentos que no se utilizan en el cuerpo del método.

- Violaciones de la Ley de Demeter

  A partir de la Ley de Demeter, esta métrica reduce la complejidad atribuida al acoplamiento en el software orientado a objetos. Esta ley especifica que una clase concreta sólo puede invocar a los métodos de un conjunto limitado de clases. Este conjunto limitado comprende a las clases de los atributos de un objeto de la clase concreta, las clases de los parámetros de los métodos de dicha clase concreta y las clases de las variables de instancia de un objeto de la clase.

Sharble y Cohen utilizaron estas métricas para evaluar dos filosofías de diseño: los métodos orientados a los datos [Coad91] y los métodos orientados a la responsabilidad [Wirf90]. El análisis realizado con este objetivo no sirve en realidad para validar las métricas, aunque se demuesta que sirve a efectos de valorar comparaciones de métodos. (El método orientado a la responsabilidad produce valores de acoplamiento más bajos y de cohesión mayores que el método orientado a los datos).

2.4.2.8 Las Métricas de Li y Henry

El objetivo de Wei Li y Sallie Henry, al igual que Sharble y Cohen, no fue proponer un conjunto de métricas, sino centrarse en la relación existente entre las métricas tomadas en la fase de diseño de software y la facilidad de mantenimiento [Li93, Li95]. Para ello, se basaron también en las métricas propuestas por Chidamber y Kemerer [Chid94], aunque propusieron algunas adicionales debido a que consideraban que aquellas no captaban con precisión algunas propiedades de la OO, debido a la vaguedad de las definiciones. Identificaron tres tipos de acoplamiento entre clases diferentes:

- Acoplamiento por Paso de Mensajes (MPC)

  Es muy común entre clases, ya que las clases se comunican entre sí mediante el envío y recepción de mensajes. Esta métrica mide el número de mensajes que una clase emite.

- Acoplamiento por Herencia

  Se mide mediante las métricas Profundidad del Arbol de Herencia (DIT) y Número de Hijos (NOC), propuestas por Chidamber *et al.*

- Acoplamiento de Abstracciones de Datos (DAC)

  Lo causan las variables de los tipos abstractos de datos. Un tipo abstracto de datos es la definición de otra clase [Hend90]

Además, consideraron que entre las métricas propuestas no había ninguna que se centrara en medir el tamaño de la interfaz de una clase. La interfaz de una clase es el conjunto de operaciones (métodos) definidos en la misma. Así, aunque la métrica Métodos Ponderados por Clase (WMC) considera a los métodos para medir la complejidad de una clase, no se centra en los aspectos de interfaz. Por tanto, propusieron una métrica adicional: Número de Métodos (NOM).

También propusieron dos métricas de tamaño, con unos nombres un tanto peculiares:

- Tamaño 1 (SIZE1)

  Es el número de líneas de código fuente, excluyendo las líneas de comentario.

- Tamaño 2 (SIZE2)

  Es el número total de atributos de datos más el número de métodos locales de una clase.

De todos modos, estas dos últimas métricas no se utilizaron en los análisis estadísticos por sus altas correlaciones con las demás métricas.

El análisis estadístico pretendía demostrar la validez predictiva de estas métricas tomadas durante el diseño con respecto al esfuerzo de mantenimiento, ya que en [Li93] se demostró su validez habiendo sido tomadas las métricas del código. De los documentos generados durante el diseño, sólo se pueden recopilar con efectividad DIT, RFC, LCOM, DAC y NOM. Se estudiaron únicamente dos sistemas comerciales y se utilizó un modelo de regresión múltiple.

Las conclusiones fueron que se podía medir cuantitativamente un diseño y que, a partir de las métricas, se podía predecir el esfuerzo de mantenimiento.

La crítica principal a este trabajo es el hecho de que la existencia de una correlación entre dos variables no hace que esta se pueda utilizar como un modelo predictivo. Hitz advierte que la variable dependiente, el esfuerzo de mantenimiento, depende de un sinfín de factores que no se mantienen constantes entre proyectos y que por tanto, las predicciones pueden ser muy poco fiables [Hitz95a]. Por otra parte, Binkley afirma que tanto DAC como MPC consideran que todas las relaciones de acoplamiento son iguales y que por tanto, no son medidas muy precisas [Bink96].

2.4.2.9 Las Métricas de Kolewe

Kolewe [Kole93] también basa su propuesta, como muchos, en las métricas de Chidamber y Kemerer [Chid91, Chid94].

Sin embargo, se centra en el hecho de que un sistema orientado a objetos necesita dos niveles de métricas, debido a la presencia de dos estructuras en este tipo de sistemas (descarta el uso de métricas tradicionales de tipo lingüístico): las *métricas para el nivel de clase*, que miden la complejidad de las clases que componen el diseño, y las *métricas para el nivel de sistema*, que miden la complejidad de las interacciones de los objetos en el diseño.

- Métricas de Nivel de Clase

  Son cinco de las métricas propuestas por Chidamber: los Métodos Ponderados por Clase (WMC), la Profundidad del Arbol de Herencia (DIT), el Acoplamiento entre objetos (CBO), la Respuesta de una Clase (RFC) y la Escasez de Cohesión en los Métodos (LOC). Todos ellos están relacionados con la complejidad y tamaño de una clase.

- Métricas de Nivel de Sistema

  La utilidad de esta categoría de métricas no se ha analizado demasiado. Una de las métricas de Chidamber pertenece a este grupo: Número de Hijos (NOC). Pero, además, Kolewe propone tres métricas adicionales:

  Número de Jerarquías de Clases: esta sencilla medida del tamaño del sistema intenta captar el número de clusters o agrupaciones de conceptos fundamentales que el sistema contempla.

  Número de Clusters de Clases: esta métrica mide el número de interconexiones entre clases de un sistema, así como el número de clusters o agrupaciones de conceptos fundamentales que el sistema contempla. De manera formal, dado un sistema con un número de clases, esta métrica se define como el número de conjuntos disjuntos formados por la intersección de los conjuntos de las clases asociadas a una clase dada.

  Complejidad de Asociación: esta métrica es una medida de la complejidad de la estructura de asociación de un sistema. Se define de forma análoga a la métrica de McCabe: $AC = A - C + 2P$, donde A es el número de asociaciones en el diagrama de clases, C es el número de clases y P es el número de grupos desconectados en el diagrama de clases.

Kolewe concluyó que todas estas métricas intentan medir el acoplamiento y la cohesión de una clase de un modo u otro, pero que se centran en distintos aspectos. También concluyó que el Acoplamiento de Clase y la Respuesta de una Clase eran las métricas con mayor capacidad predictiva con respecto a los índices de errores.

2.4.2.10 Las Medidas de Brito e Abreu

Fernando Brito e Abreu propuso su primera colección de medidas, denominada MOOD, en 1993 [Brit94a], modificándolas y ofreciendo una segunda versión más refinada en [Brit96]. Además, como estas medidas están definidas de manera genérica, sin vinculación a ningún lenguaje de programación en concreto, el autor también ha publicado la forma de interpretar las variables de las expresiones matemáticas, dependiendo del lenguaje de programación objetivo, ofreciendo correspondencias con C++ [Brit95, Brit96] y con Eiffel [Brit96, Brit96a]. El conjunto de medidas propuestas se expresan mediante cocientes, son independientes del tamaño y adimensionales, se pueden calcular en la fase de diseño preliminar y cubren todas las características de la orientación a objetos. Son las siguientes:

- Factor de Ocultamiento de Métodos (MHF)

  Cociente en el que el numerador es la suma de *invisibilidades* de todos los métodos definidos en todas las clases. La *invisibilidad* de un método es el porcentaje de clases desde las cuales un método no es visible. El denominador es el número total de métodos. La expresión es la siguiente:

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

  donde $(M_{mi}) = \frac{\sum_{j=1}^{TC} esvisible(M_{mi}, C_j)}{TC - 1}$ y

$$esvisible(M_{mi}, C_j) = \begin{cases} 1 & \Leftrightarrow \begin{cases} j \neq i \\ C_j \text{ puede llamar a } M_{mi} \end{cases} \\ 0 & \text{en caso contrario} \end{cases}$$

  siendo *TC* el número total de clases y $M_d(C_i)$ los métodos definidos, no heredados.

- Factor de Ocultamiento de Atributos (AHF)

  Esta medida es idéntica a la anterior, pero considerando los atributos en vez de los métodos. La expresión es la siguiente:

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

  donde $(A_{mi}) = \frac{\sum_{j=1}^{TC} esvisible(A_{mi}, C_j)}{TC - 1}$ y

$$esvisible(A_{mi}, C_j) = \begin{cases} 1 & \Leftrightarrow \begin{cases} j \neq i \\ C_j \text{ puede referenciar a } A_{mi} \end{cases} \\ 0 & \text{en caso contrario} \end{cases}$$

  donde $A_d(C_i)$ son los atributos definidos, no heredados.

- Factor de Herencia de Métodos (MIF)

  Es la proporción de los métodos heredados de todas las clases y todos los métodos del sistema.

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

donde $M_a(C_i) = M_d(C_i) + M_i(C_i)$, siendo $M_a(C_i)$ los métodos disponibles, $M_d(C_i)$ los métodos definidos y $M_i(C_i)$, los métodos heredados.

- Factor de Herencia de Atributos (AIF)

  Esta medida es idéntica a la anterior pero con respecto a los atributos. Su expresión es:

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

  donde $A_a(C_i) = A_d(C_i) + A_i(C_i)$ donde $A_a(C_i)$ son los atributos disponibles, $A_d(C_i)$ los atributos definidos y $A_i(C_i)$, los heredados.

- Factor de Polimorfismo (POF)

  Representa el número real de situaciones polimórficas sobre el número máximo posible de las mismas. La expresión es:

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

  donde $M_d(C_i) = M_n(C_i) + M_o(C_i)$, siendo $M_n(C_i)$ el número de métodos nuevos, $M_o(C_i)$ el de métodos redefinidos y $DC(C_i)$ el número de clases descendentes de C_i.

- Factor de Acoplamiento (COF)

  Representa el número real de relaciones cliente-servidor no imputables a la herencia sobre el número máximo de los mismos. La expresión es la siguiente:

$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} escliente(C_c, C_s)\right]}{TC^2 - TC}$$

  Y $escliente(C_c, C_s) = \begin{cases} 1 & \Leftrightarrow C_c \to C_s \wedge C_c \neq C_s \\ 0 & \text{en caso contrario} \end{cases}$, donde de C_c es la clase cliente, C_s es la servidora y $C_c \to C_s$ implica una relación cliente-servidor no de herencia.

2.4.2.11 Resumen de otras Medidas Orientadas a Objetos

A continuación, en la Tabla 2.1, ofrecemos un resumen de otras aportaciones al área de las medidas orientadas a objetos, que consideramos de interés.

Autor	*Tipo de Medida*	*Nombre de la Medida*	*Validación*	
			Teo.	*Emp.*
Karunanithi *et al.* [Karu93]	Reusabilidad	Medidas Clasificadas con dos criterios: - a nivel de cliente, de servidor y de sistema. - de uso directo, modificado y genérico	-	-
Briand *et al.* [Bria94a]	Acoplamiento	Acoplamiento de Exportación e Importación	Si	Si
	Cohesión	Ratio Optimista, Pesimista y Neutro de Interacciones Cohesivas		
Cant *et al.* [Cant94]	Complejidad Cognitiva	Medida de Complejidad de Particiones	-	Si
Martin [Mart95]	Acoplamiento	Acoplamiento Aferente y Eferente, Inestabilidad, Grado de Abstracción, Distancia.	-	-
Lewis [Lewi95]	Encapsulación	Nº Variables de Instancia Públicos y Nº Referencias a Variables de Instancia Externas	-	-
	Acoplamiento	Nº Mensajes enviados, Acoplamiento entre objetos, Mensajes reconocidos y Cierre de Invocaciones		
	Cohesión	Falta de Cohesión		
	Herencia	Nº Antecesores, Progenie de Herencia y Métodos Heredados		
	Sobrecarga	Nº Métodos Sobrecargados y Grado de Sobrecarga de Métodos, Nº Operadores Sobrecargados y Grado de Sobrecarga de Operadores		
	Redefinición	Nº Métodos Redefinidos y Grado de Redefinición de Métodos		
	Polimorfismo	Nº Referencias Polimórficas, Grado de Polimorfismo y Grado de Polimorfismo Oculto		
Ott *et al.* [Ott95]	Cohesión	Cohesión de Datos Fuerte y Débil, Adhesión de Datos, Cohesión de Clase Rígida y Flexible	-	-

Autor	*Tipo de Medida*	*Nombre de la Medida*	*Validación*	
			Teo.	*Emp.*
Hitz *et al.* [Hitz95]	Acoplamiento	Dependencia de Cambios entre Clases	-	-
	Cohesión	Localidad de los Datos		
Lee *et al.* [Lee95]	Acoplamiento	Acoplamiento de Herencia basado en el Flujo de Información	Si	Si?
		Acoplamiento no de Herencia basado en el Flujo de Información		
	Cohesión	Cohesión basada en el Flujo de Información		
Page-Jones [Page95]	Acoplamiento y Cohesión	*Connascense*	-	-
Binkley *et al.* [Bink96]	Acoplamiento OO y Clásico	Medida de Dependencia de Acoplamiento	-	-
Pant *et al.* [Pant96]	Tamaño/Complejidad	S/C	-	Si
Ebert *et al.* [Eber97]	Volumen	N° Variables y Métodos de Instancia y de Clase	-	Si
	Estructura de Métodos	N° Parámetros/método		
		N° Variables Temporales/método		
		N° Paso de mensajes/método		
	Cohesión	Accesos a variables fuera de protocolos		
	Acoplamiento	N° Clases invocadas sin herencia		
	Arbol de Herencia	Variables y Métodos heredados usados		
	Organización de Clase	Nomenclatura, Comentarios y uso de Protocolos predefinidos		
Bansiya *et al.* [Bans97]	de Sistema Simple	N° Clases, Jerarquías, Clases independientes, Clases con Herencia Simple, con Herencia Múltiple, Clases Internas, Clases Abstractas y Clases Terminales	-	Si
	de Sistema Derivadas	Medidas de Abstracción Funcional, de Atributos, de Accesos a Datos, de Accesos a Operaciones, Profundidad y Anchura media de Herencia		
	Externas a la Clase	N° de hijos, de Ancestros, Profundidad de Herencia, Acoplamiento Directo de Clase		
	Internas a la clase	N° de Métodos, Métodos Triviales, Polimórficos, N° de Operadores, Tamaño de Interfaz de Clase, Complejidad de Entropía de Clase		

Autor	*Tipo de Medida*	*Nombre de la Medida*	*Validación* Teo.	Emp.
Briand *et al.* [Bria97a]	Acoplamiento	Medidas de Interacción de Clase con Atributo, Clase con Métodos y Métodos con Métodos	SI	SI
Poulin [Poul97]	Reusabilidad	%Reusabilidad ROI-Proyecto	-	-
Shih *et al.* [Shih97]	Complejidad del Grafo de Herencia	Unidad de Herencia Repetida (URI) y Nivel de Herencia Medio	-	-
Gillibrand *et al.* [Gill98]	Complejidad de Clase Fiabilidad	Indice de Especialización	-	-

Tabla 2.1 Resumen de Medidas Orientadas a Objetos.

Y por último, no hay exposición sobre medición que no termine haciendo referencia a las palabras de Lord Kelvin:

> «When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge of it is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely in your thoughts, advanced to the stage of science»
>
> William Thomson, Lord Kelvin, 1889

2.5 LOS METODOS DE VALIDACION DE MEDIDAS

En los últimos años se han definido cientos de medidas para cuantificar toda clase de atributos del software. Pero para que todas estas medidas sean aceptadas y comparables entre sí, es necesario saber en qué condiciones se han definido y en qué situaciones se pueden utilizar, establecer de forma sistemática qué atributo es el que pretenden medir, determinar exactamente cuál es el uso de la medida, etc.

Se puede decir que lo que se necesita no sólo es un método de validación de las medidas, sino un estándar: unas características o propiedades deseables que todas las medidas deberían cumplir con vistas a uniformizar su significado y uso.

Como Briand, El Emam y Morasca señalan [Bria96], el prerrequisito obvio para que una medida sea aceptada en la academia e industria es que sea teóricamente sólida; es decir, que la medida realmente cuantifique la característica del software

que se supone mide. Por tanto, el mero proceso exploratorio de buscar correlaciones no es un proceso de validación científico aceptable, a menos que venga acompañado de una sólida teoría que lo soporte.

En consecuencia, la validación se debe enfocar desde dos puntos de vista distintos y complementarios [Bria95, Kitc95]:

1. la validación *teórica*, y
2. la validación *empírica*.

La validación teórica se centra en garantizar que una medida cumple ciertas propiedades que son parte de un criterio predefinido. Esta validación pretende demostrar que una medida está realmente cuantificando el atributo propuesto y que es útil. Se puede realizar siguiendo dos tendencias marcadas en la literatura: la Teoría de la Medición y el enfoque axiomático (aunque se puede demostrar que por debajo de muchos axiomas propuestos en la literatura subyace la Teoría de la Medición).

Son muchos los autores que han propuesto propiedades para las medidas del software, como Conte [Cont86], Weyuker [Weyu88], Lakshmanan [Laks91], Kitchenham y Fenton [Kitc95, Fent94, Fent97], Shepperd [Shep93], Tian y Zelkowitz [Tian95], Briand *et al.* [Brian96], Whitmire [Whit97] y Zuse [Zuse98], entre otros.

La validación empírica pretende corroborar que las medidas de los atributos son consistentes con los valores predichos por los modelos que se basan en dichos atributos. Es decir, se intenta relacionar la medida de un atributo interno con la medida de uno externo. Esto nos lleva a distintos tipos de análisis de datos cuya elección dependerá de las medidas de las que dispongamos. La validación empírica es la propuesta por el IEEE en su estándar sobre medición [IEEE92a], por Schneiderwind [Schn92, Schn94], Briand [Bria95] y MacDonell [MacD94], entre otros.

De todos modos, hasta el momento no hay un enfoque único para llevar adelante la validación, ni desde el punto de vista teórico, donde los estudiosos no se ponen de acuerdo sobre las propiedades que una medida debe cumplir (sobre todo si es una medida de complejidad), ni desde el punto de vista de la validación empírica, donde varía el grado de rigor del uso de los métodos estadísticos

A continuación, se explican brevemente algunos conceptos de la Teoría de la Medición, a la luz de los cuales se pueden entender las propiedades teóricas propuestas por distintos autores y sus críticas, así como los condicionantes de la validación empírica.

2.5.1 Conceptos Fundamentales de la Teoría de la Medición

Uno de los primeros autores en aplicar la Teoría de la Medición a la medición del software fue Horst Zuse, a mediados de los 80, usando como base los trabajos de Roberts [Robe79] y Krantz *et al.* [Kran71]. Este autor, junto con otros, se centró en el uso de esta teoría porque proporciona una interpretación empírica de los números de las medidas del software y condiciones para las operaciones de concatenación y descomposición, que son estrategias típicas en la Ingeniería del Software. Así, mediante el uso de estos sistemas axiomáticos, se pueden caracterizar los modelos cualitativos sobre los que se soportan las medidas. La Teoría de la Medición implica una descripción matemática de las escalas, medidas y métodos de medición [Zuse98]. La noción que subyace es que si existe en nuestro universo del discurso una cierta comprensión intuitiva o empírica de las relaciones que se dan entre los distintos objetos, entonces esas relaciones se pueden formalizar en un sistema matemático. (Precisamente, al proceso mediante el que se asignan símbolos o números a los atributos de una entidad del mundo real se le denomina *medición*, como mencionabamos en la sección 2.3.1).

A continuación se detalla un conjunto básico de conceptos de la Teoría de la Medición.

Sistema Relacional Empírico

Sea $\mathbf{A} = (A, \bullet\geq, \circ)$ un sistema relacional empírico, donde A es un conjunto no vacío de objetos, $\bullet\geq$ es una relación empírica sobre A, y $\circ$ es una operación binaria cerrada sobre A (por supuesto, hay más de una operación y relación posibles).

Sistema Relacional Numérico

Sea $\mathbf{B} = (\Re, \geq, \odot)$ un sistema relacional numérico, donde $\Re$ son los número reales, $\geq$ es una relación en $\Re$, y $\odot$ es una operación binaria cerrada sobre $\Re$ (por ejemplo, una operación binaria puede ser la adición).

Definición formal de una medida

Una medida es una correspondencia $\mu: A \rightarrow \Re$, tal que para todo a, b $\in$ A, se cumple:

$$a \bullet\geq b \Leftrightarrow \mu(a) \geq \mu(b)$$

La medida será *aditiva* si se cumple además:

$$a \circ b \Leftrightarrow \mu(a) + \mu(b)$$

Escala

Se denomina escala a la terna formada por (**A**, **B**, μ), siendo una escala aditiva

$$((A, \bullet\succeq, \circ), (\Re, \geq, +), \mu).$$

Transformación Admisible

Sea (**A**, **B**, μ) una escala. Una correspondencia $g:A \rightarrow \Re$ es una transformacion admisible si y sólo si (**A**, **B**, g) es también una escala.

Tipos de Escalas

Existen distintos tipos de escalas, debido a las distintas transformaciones admisibles, que se resumen en la Tabla 2.2.

Tipos de Escalas	*Transformacion Admisible*
Nominal	g correspondencia 1 1
Ordinal	g funcion monotónica estrictamente creciente
Intervalo	$g(x) = ax + b$, $a > 0$
Ratio	$g(x) = ax$, $a > 0$
Absoluta	$g(x) = x$

Tabla 2.2 Tipos de Escala y sus transformaciones.

Orden Débil

Sea $\bullet\succeq$ una relación binaria sobre objetos de **P** Esta relación será un orden débil si es *transitiva* y *completa*.

$$P1 \bullet\succeq P2 \wedge P2 \bullet\succeq P3 \Rightarrow P1 \bullet\succeq P3 \quad \text{Transitiva}$$

$$P1 \bullet\succeq P2 \vee P2 \bullet\succeq P1 \qquad \text{Completa, para todo P1, P2, P3} \in \mathbf{P}.$$

El orden débil es un prerrequisito para la medición que permita la ordenación. Por tanto, si existe una medida μ que cumple lo anterior, entonces $((P, \bullet\geq), (\Re, \geq), \mu)$ es una escala ordinal.

Estructura Extensiva

El sistema relacional $(P, \bullet\geq, \circ)$ es una estructura extensiva si y sólo si se cumplen los siguientes axiomas para $P1,..., P4 \in P$:

A1: $(P, \bullet\geq)$ es un Orden Débil

A2: $P1 \circ (P2 \circ P3) \sim (P1 \circ P2) \circ P3$ (Asociatividad Débil)

A3: $P1 \bullet\geq P2 \Leftrightarrow P1 \circ P3 \bullet\geq P2 \circ P3 \Leftrightarrow P3 \circ P1 \bullet\geq P3 \circ P2$ (Monotonicidad)

A4: Si $P3 \bullet> P4$, $\forall P1, P2\ \exists n \in \aleph$, tal que $P1 \circ nP3 \bullet\geq P2 \circ nP4$ (Axioma de Arquímedes).

Teorema de la Estructura Extensiva

Sea P un conjunto no vacío, $\bullet\geq$ una relación binaria sobre P y $\circ$ una operación binaria cerrada sobre P. Entonces, $(P, \bullet\geq, \circ)$ es una estructura extensiva cerrada si y sólo si existe una función μ que asigna números reales, tal que para todo $P1, P2 \in P$ se cumple:

$P1 \bullet\geq P2 \Leftrightarrow \mu(P1) \geq \mu(P2)$ (se preserva el orden)

$\mu(P1 \circ P2) = \mu(P1) + \mu(P2)$ (aditiva en la concatenación)

Este teorema implica que una medida aditiva asume directamente una estructura extensiva y además la transformación admisible de la escala de ratio es cierta.

Tipos de Escalas y Métodos Estadísticos

El punto en el que convergen la Teoría de la Medición y la validación empírica es en el uso que se puede hacer de los distintos métodos de análisis de datos disponibles [Fent96]. En la Tabla 2.3, se resumen los métodos estadísticos apropiados a cada tipo de escala:

Tipos de Escalas	*Método Estadisticos Adecuados*	*Tipo de Método*
Nominal	Moda	No Paramétrico
	Frecuencia	
Ordinal	Mediana	No Parametrico
	τ de Kendall	
	r de Spearman	
Intervalo	Media	Parametrico y no Parametrico
	Correlacion de Pearson	
Ratio	Media geometrica	Parametrico y no Parametrico
	Coeficiente de Variacion	
Absoluta	Todos los anteriores	Parametrico y no Parametrico

Tabla 2.3 Métodos estadisticos y tipos de escala.

2.5.2 Las Propiedades de Zuse

Este autor se ha dedicado a aplicar la Teoría de la Medición a las medidas de complejidad con verdadero rigor. Se centra en las condiciones que los sistemas relacionales empiricos deben satisfacer, de forma que se puedan utilizar medidas aditivas en la escala de ratio [Zuse94, Zuse98].

Centra toda su investigacion *en la propuesta de la estructura extensiva modificada como forma de llegar a una escala de ratio*, considerando esta estructura como el modelo cualitativo subyacente Las modificaciones a la estructura clasica son la adición del axioma de conmutatividad débil, la sustitución del axioma de monotonicidad por el de monotonicidad débil y se hace más estricto el axioma de Arquimedes. Así, un sistema relacional (P, $\bullet\succeq$, °) es una estructura extensiva modificada si y sólo si se cumplen los siguientes axiomas para P1,..., P4 ∈ P:

A1: (P, $\bullet\succeq$) es un Orden Débil

A2: P1° (P2 ° P3) ≈ (P1 ° P2) ° P3 (Asociatividad Débil)

A3: P1 ° P2 ≈ P2 ° P1 (Conmutatividad Débil)

A3: P1 $\bullet\succeq$ P2 ⇒ P1 ° P3 $\bullet\succeq$ P2 ° P3 ⇔ P3 ° P1 $\bullet\succeq$ P3 ° P2 (Monotonicidad Debil)

A4: Si P3 $\bullet\succ$ P4, ∀ P1, P2 ∃ n ∈ ℕ, tal que P1 ° nP3 $\bullet\succ$ P2 ° nP4 (Axioma estricto de Arquimedes).

También indica que una medida en una escala ordinal, si es aditiva, entonces es una medida de ratio a través del Teorema de la Estructura Extensiva.

El autor también describe cuatro Condiciones de Independencia, en orden creciente de rigor. Estas condiciones describen el grado de independencia entre los componentes de una operación de concatenación. Son las siguientes:

C1: $a \approx b \Rightarrow a \circ c \approx b \circ c \wedge a \approx b \Rightarrow c \circ a \approx c \circ b$

C2: $a \approx b \Leftrightarrow a \circ c \approx b \circ c \wedge a \approx b \Leftrightarrow c \circ a \approx c \circ b$

C3: $a \bullet\geq b \Rightarrow a \circ c \bullet\geq b \circ c \wedge a \bullet\geq b \Rightarrow c \circ a \bullet\geq c \circ b$

C4: $a \bullet\geq b \Leftrightarrow a \circ c \bullet\geq b \circ c \wedge a \bullet\geq b \Leftrightarrow c \circ a \bullet\geq c \circ b$

Si la condición C1 no se cumple, no se puede cumplir ninguna de la sucesivas; es decir, cada una de ellas es el prerrequisito de las siguientes. La consecuencia del cumplimiento de estas condiciones es importante: si la condición C1 no se cumple, no sólo no se cumple ninguna de las demás, sino que implica que no puede existir una regla de concatenación para la medida; es decir, no existe ninguna función *f* tal que: $\mu(P1 \circ P2) = f(\mu(P1), \mu(P2))$. Si no existe una función de concatenación, entonces nunca se van a poder determinar el valor de la medida de un sistema completo en función de los valores de sus componentes y además, se cierra el camino hacia la estructura extensiva, ya que son el puente entre el orden débil y esta última.

En cuanto a la Orientación a Objetos, según Zuse, las medidas orientadas a objetos rara vez van a poder asumir una estructura extensiva; se necesitan sistemas de axiomas más débiles. Así, Zuse y Fetcke [Fetk95] proponen las operaciones de concatenación a nivel de clase que se pueden dar en la orientación a objetos que son CUNI, la unión de clases y CINT, la intersección de clases. La primera consecuencia es que CUNI es idempotente [CUNI(A,A) = A], con lo que se cierra el camino hacia la estructura extensiva porque no se cumple el axioma de Arquímedes. También proponen las operaciones de concatenación HAGG (unión por agregación) y HGEN (unión por generalización). Estos mismos autores señalan que la regla de combinación típica de la orientación a objetos es $\mu(A \circ B) = \mu(A) + \mu(B) - \mu(A \cap B)$, es decir, $\mu(CUNI(A, B)) = \mu(A) + \mu(B) - \mu(CINT(A, B))$, donde A y B son clases (también puede darse la relación $\geq$). Esta regla tienen sentido ya que las operaciones de concatenación se definen en base a la teoría de conjuntos. El término '$- \mu(CINT(A, B))$' implica el rechazo de la condición de independencia C1, por lo que también se rechaza la estructura extensiva.

Por ello, Zuse propone una estrategia de comprobación de los axiomas que cumple una medida, de manera que si se supera el orden débil, su escala esté por encima de la ordinal. Los axiomas que se investigan son una mezcla de la Función de Confianza modificada, la Relación de Confianza modificada y los axiomas de DeFinetti.

El orden de comprobación propuesto es el siguiente, de forma que si no cumple un punto, se pasa a examinar el siguiente:

1. Comprobación de los axiomas de la Estructura Extensiva
2. Comprobación de la Función de Confianza modificada
3. Comprobación de los axiomas de la Relación de Confianza modificada, que son el orden débil, el axioma de predominio, la monotonicidad parcial de DeFinetti y la positividad.

Los axiomas que se deben comprobar se resumen en la Tabla 2.4 que se ilustra a continuación.

Nombre del Axioma	*Descripción del Axioma*
Estructura Extensiva	Orden Débil, Asociatividad Debil, Conmutatividad Débil, Monotonicidad Débil, Axioma de Arquimedes
Funcion de Confianza Modificada	$\mu(A \cup B) = \mu(A) + \mu(B) - \mu(A \cap B)$.
Axioma de Predominio	$\forall A \supseteq B \Rightarrow A \succeq B$
Monotonicidad Parcial de DeFinetti	$\forall (A \supseteq B, A \cap C = \emptyset) \Rightarrow (A \succeq B \Leftrightarrow A \cup C \succeq B \cup C)$
Positividad	$A \succeq 0$

Tabla 2.4 Axiomas a comprobar en una medida orientada a objetos.

Los principales detractores de Zuse son Lionel Briand, Sandro Morasca y Khaled El Emam [Bria96, Bria96a] que argumentan que Zuse obliga a que las medidas de complejidad sean aditivas, cuando esto no es nada intuitivo. Critican que Zuse, con esta visión tan limitada, está acercando las medidas de complejidad a las de tamaño, no captando plenamente los aspectos reales de la complejidad (interdependencias entre componentes).

Otro punto en el que ambas escuelas tienen opiniones completamente contrapuestas es en el tema del uso apropiado de los métodos estadísticos dependientes del tipo de escala de la medida a validar.

Fenton y Zuse argumentan que el tipo de escala de una medida debe determinar el tipo de prueba estadística a realizar; es decir, desde su punto de vista, dada una medida, en primer lugar se debería determinar su escala y luego, usar solamente los métodos apropiados.

Por otra parte, Briand, El Emam y Morasca argumentan que este enfoque es excesivamente riguroso para el conocimiento actual del que disponemos en el mundo del software. Su principal argumento es que al limitar el tipo de pruebas estadísticas a realizar, podemos perder la oportunidad de llegar a conclusiones que obtendríamos con pruebas 'inválidas'. Así, las pruebas estadísticas se deberán elegir dependiendo de las cuestiones a las que se desee dar respuesta con el uso de la medida, sin tener en cuenta el tipo de escala de la misma, considerando además, que muchas veces es realmente difícil determinar el tipo de escala de una medida.

Su visión es más pragmática y se apoyan en disciplinas como la Sociología, en las cuales el progreso no hubiera sido posible si se hubiera prohibido este uso 'aproximado' de los tipos de escala [Bria95, Bria96a].

Existe una tercera visión, muy reciente, propuesta por Scott Whitmire [Whit97] que probablemente sea la más purista de todas desde el punto de vista de la Teoría de la Medición. Este autor hace un estudio realmente exhaustivo de las diferentes estructuras empíricas que llevan a los distintos tipos de escalas. Su enfoque se diferencia con respecto a los anteriores en que aquellos comienzan con una medida dada y prosiguen. Whitmire propone comenzar con las preguntas sobre las que se desea obtener respuestas, que llevarán a exigir el uso de estadística paramétrica o no paramétrica. Esto condiciona el tipo de escala que necesitamos; conocida la escala, se busca la estructura más básica que lleve a ella, inventando una medida, en caso de que no exista alguna que cumpla los requisitos o analizando estructuras más complejas. Así, no se centra en buscar una medida específica, sino que un tipo de escala.

2.5.3 Las Propiedades de Weyuker

El trabajo de Elaine Weyuker [Weyu88] es uno de los primeros intentos para formalizar el difuso concepto de la complejidad de los programas. Las propiedades propuestas por esta autora han sido muy discutidas y comentadas por numerosos investigadores y hoy en día siguen siendo un punto de referencia y comparación para cualquiera que investigue y proponga medidas de complejidad del software.

Weyuker se centra en las características sintácticas de los programas para enunciar sus propiedades, siendo la notación la siguiente: P, Q y R son programas, |P| es un número no negativo que representa la complejidad de P con respecto a una medida hipotética, P≡Q representa que P es funcionalmente equivalente a Q y P;Q representa la concatenación de los programas P y Q.

Las propiedades de Weyuker son las siguientes:

- W1: *No Generalidad* o *Axioma Básico de una Medida*

  $$\exists P \exists Q \quad |P| \neq |Q|$$

  Implica que una medida que asocia la misma complejidad a todos los programas, no es realmente una medida.

- W2: *Granularidad* o *Identificadores Finitos*

  Sea *c* un número no negativo. Existe un número finito de programas con complejidad igual a *c*.

  Implica que una medida no debe ser tan poco 'sensitiva' como para clasificar todos los programas en unas pocas clases de complejidad equivalente.

- W3: *No Unicidad* (Noción de *Equivalencia*)

  $$\exists P \exists Q \quad |P| = |Q|$$

  Implica que una medida no debe ser tan restrictiva que asigne a cada programa una complejidad única.

- W4: *Detalles de Diseño*

  $$\exists P \exists Q \quad P \neq Q \wedge |P| \neq |Q|$$

  Implica que aunque dos programas tengan la misma funcionalidad, son los detalles de diseño los que afectan a la complejidad; es decir, se intenta medir la complejidad del programa, no de la función que se está realizando.

- W5: *Monotonicidad*

  $$\forall P \forall Q \quad |P| \leq |P;Q| \wedge |Q| \leq |P;Q|$$

  Implica que los componentes de un programa no son más complejos que el programa en sí. Además, según la autora, esta es una propiedad fundamental y señala que es difícil imaginar en qué sentido una medida que no cumpla esta propiedad puede medir la complejidad de un programa.

- W6: *No Equivalencia de Interacción*

  $$\exists P \exists Q \exists R \quad |P| = |Q| \wedge |P;R| \neq |Q;R|$$

  $$\exists P \exists Q \exists R \quad |P| = |Q| \wedge |R;P| \neq |R;Q|$$

  Implica que la concatenación de dos programas no afecta a la complejidad del programa resultante de una manera uniforme. Así, aunque R tiene una complejidad fija si se considera de forma aislada, después de la concatenación

puede que no tenga ninguna interacción con P; sin embargo, puede tener interacciones importantes que afectan a la complejidad resultante después de la concatenación con Q.

- W7: *Permutación*

  Existen programas P y Q, tal que Q está formado por la permutación de las sentencias de P, y $|P| \neq |Q|$.

  Implica que una medida de complejidad debería responder al orden de las sentencias y por tanto, a las interacciones potenciales entre las mismas.

- W8: *Cambio de Nombre*

  Si P es el resultado de un cambio de nombre de Q, entonces $|P| = |Q|$.

  Implica que el cambio de nombre de las variables no afecta a la complejidad, aunque la propia autora reconoce que puede haber circunstancias en las cuales la complejidad debería ser diferente, siendo estas circunstancias aquellas en las que se intenta valorar la dificultad de comprensión (complejidad psicológica).

- W9: *Supra-aditividad*

  $$\exists P \exists Q \quad |P| + |Q| < |P;Q|$$

  Implica que al menos en algunos casos, la complejidad de un programa formado por la concatenación de dos programas es mayor que la suma de sus complejidades individuales. Refleja el hecho de que puede haber interacciones entre los programas concatenados.

  A continuación, Weyuker añade una variante más rigurosa de esta propiedad, a la que denominamos Supra-aditividad Débil:

  $$\forall P \forall Q \quad |P| + |Q| \leq |P;Q|$$

  que implica que la complejidad de un programa debería ser no menor que la suma de la complejidad de sus componentes. Como esta propiedad es de las más polémicas, la propia autora la deja como una cuestión abierta a la investigación.

Muchos han sido los autores que han criticado las propiedades de Weyuker; entre ellos podemos mencionar a Chernlavsky y Smith [Cher91] que propusieron una medida que cumplía con todas las propiedades anteriores y que era absurda, pues intuitivamente no medía ningún atributo de software. De todas maneras, otros autores han dado la replica a Chernlavsky *et al.*, señalando que el hecho de que una

medida sin interés cumpla las propiedades no es motivo para rechazar dichas propiedades; es más, debería valorarse la utilidad de una medida después de tener la garantía de que cumple las propiedades [Bria96, Fent94].

Las críticas más severas a las propiedades de Weyuker han venido en los últimos años de la mano de Horst Zuse, cuyos trabajos sobre las propiedades de las medidas se centran en la Teoría de la Medición y se han comentado en la sección anterior. Zuse está de acuerdo con las cinco primeras propiedades y con la octava. Sin embargo, le parece que las demás son contradictorias entre sí, debido a que exigen objetivos completamente opuestos. Así, la propiedad W6 rechaza la primera condición de independencia C1, cuya consecuencia es que no se puede llegar a una medida en una escala de ratio a través de una estructura extensiva. Además, impide la existencia de una regla de combinación con lo cual cierra la posibilidad de determinar el valor de una medida para un sistema completo en función de los valores de sus componentes.

La propiedad W7 tiene consecuencias similares: se rechaza la conmutatividad débil, rechazando la escala de ratio a través de la estructura extensiva.

En cuanto a la propiedad W9, Zuse indica que esta propiedad es razonable e intuitiva ya que la autora asume interacciones entre los programas P y Q, aunque no se definen dichas interacciones. Según Zuse, la definición formal de las interacciones se puede realizar mediante la definición de las condiciones de independencia C1-C4, pero la condición C1 se rechaza completamente en la propiedad W6. Además, la propiedad W9 requiere una escala de ratio, que queda excluida por la vía de la estructura extensiva.

Ante estas críticas, las reacciones no se han hecho esperar. Así, Briand, El Emam y Morasca [Bria96a, Mora97] argumentan que las propiedades W6, W7 y W9 son significativas para la escala de ratio puesto que se les puede aplicar las transformaciones admisibles para dicha escala.

Además, señalan que las propiedades de Weyuker no son compatibles con que el sistema empírico de la medida sea una estructura extensiva y que esta es una condición suficiente para obtener una escala de ratio, pero no necesaria. Es decir, señalan que sólo se excluye la medida de ratio *vía* la estructura extensiva, pero no a través de otras estructuras. La respuesta de Zuse ha sido la siguiente: con respecto a las propiedades W6 y W7, admiten transformaciones también de la escala de intervalo, ordinal e incluso nominal. El hecho de que las propiedades sean significativas para una cierta escala no implica que una medida se pueda utilizar con dicha escala. En cuanto al requisito de obtener una medida de ratio sin la estructura extensiva, Zuse reta a Briand *et al.* a definir una medida que sea de escala de ratio,

sin asumir una estructura extensiva, que cumpla con el axioma de positividad débil (propiedad W5), rechace la conmutatividad débil, las condiciones de independencia C1-C4, el axioma de monotonicidad débil y de monotonicidad, indicando sarcásticamente que si una medida así se pudiese definir, sería todo un progreso científico.

Otros autores que ha analizado estas propiedades son Henderson-Sellers y Fenton [Hend96, Fent94, Fent96], que señalan el aspecto contradictorio de las mismas desde el punto de vista de los objetivos en los que se centran y que resaltan que son propiedades para medidas sintácticas, más centradas en la implementación. Así, la propiedad W1 se viola con las medidas de no estructuración que asignan el mismo valor a todos los programas no estructurados. Las propiedades W5 y W6 no se pueden cumplir simultáneamente porque W5 se centra en el tamaño, mientras que W6 está relacionada con la comprensión. Las conclusiones de Henderson-Sellers se resumen en la Tabla 2.5, en la que se valora para qué medidas son apropiadas las propiedades de Weyuker [Hend96].

En cuanto a la aplicabilidad de estas propiedades a los sistemas orientados a objetos, parece que no debería ser inmediata, ya que estas propiedades se desarrollaron considerando como elementos primitivos los programas y como operación básica la concatenación secuencial. Sin embargo, en la orientación a objetos el elemento básico es la *clase*, en el que el orden de los atributos y las operaciones no importa (propiedad W7) y en cuanto a la concatenación, Fetcke, Hitz y Zuse [Fetc95, Hitz96, Zuse98] señala que existen cuatro formas diferentes de concatenar dos clases por lo que todas las propiedades deberían examinarse bajo esta perspectiva.

	Apropiadas	**No Apropiadas**
W1	Programas Estructurados	Medidas de No Estructuración
W2	Todos	-
W3	Relacionadas con Tamaño	Complejidad del Flujo de Control
W4	Relacionadas con Comprensión OO → Basadas en Interfaz	-
W5	Relacionadas con Tamaño	Relacionadas con Comprensión
W6	Relacionadas con Comprensión	Relacionadas con Tamaño Incompatible Ratio Vía E. Extensa.
W7	Relacionada con Complejidad Psicológica	Incompatible Ratio Vía E. Extensa.
W8	Relacionadas con Tamaño	Relacionadas con Comprensión
W9	Escala de Ratio (no aditiva)	Escala Ordinal

Tabla 2.5 Aplicabilidad de la propiedades de Weyuker.

2.5.4 Las Propiedades de Briand, Morasca y Basili

Briand, Morasca y Basili [Bria96] argumentan que cuando se proponen propiedades para las medidas del software, dichas propiedades suelen ser *únicas* independientemente del concepto que se pretende medir. Así, señalan que las medidas suelen medir el tamaño o la complejidad, y bajo este nombre, se incluyen todo tipo de características, que en realidad son diferentes. Como consecuencia, las distintas medidas generalmente no pueden cumplir todo el conjunto de propiedades propuestas. Por este motivo, proponen un conjunto de propiedades para las medidas del software, pero propiedades distintas dependientes del concepto que se pretende medir: propiedades para el tamaño, longitud, complejidad, cohesión y acoplamiento.

Para detallar su conjunto de propiedades, los autores definen *S* como un sistema formado por elementos que se relacionan. También utilizan el término *módulo* para señalar subconjuntos de elementos de un sistema *S* (así, un módulo puede ser un subprograma, un subsistema, una clase o incluso un grupo de clases), teniendo en cuenta que los elementos internos de un módulo se conectan al resto del sistema mediante relaciones de entrada o de salida. En realidad, se puede considerar que se basan en la teoría de conjuntos. A continuación, comentamos únicamente las propiedades propuestas para el tamaño y la complejidad.

Propiedades para una medida de Tamaño

- T1: *No Negatividad*

  El tamaño de un sistema *S* es no negativo:

  $$\text{Tamaño}(S) \geq 0$$

  donde *S* es un conjunto de elementos (que podemos llamar subsistemas o módulos).

- T2: *Valor Nulo*

  El tamaño puede ser nulo:

  $$S = \emptyset \Rightarrow \text{Tamaño}(S) = 0$$

- T3: *Aditividad para Conjuntos Disjuntos*

  El tamaño es aditivo para sistemas o conjuntos disjuntos:

  $$A1 \cap A2 = \emptyset \Rightarrow \text{Tamaño}(A1 \cup A2) = \text{Tamaño}(A1) + \text{Tamaño}(A2)$$

Sigue el siguiente principio:

Tamaño (A1 ∪ A2 ∪ ... ∪ An) ⇒ $\alpha_1 - \alpha_2 + \alpha_3 - ... + (-1)^{n-1}\alpha_n$, donde α_i es la suma de los tamaños de las intersecciones tomando *i* componentes cada vez.

- T4: *Monotonicidad Creciente*

  Añadir elementos a un sistema no puede hacer que su tamaño disminuya:

  A1 ⊆ A2 ⇒ A1 ≤ A2

- T5: *No Sinergia*

  El tamaño de la fusión de dos sistemas no puede exceder la suma de ambos:

  A = A1 ∪ A2 ⇒ Tamaño (A) ≤ Tamaño (A1) + Tamaño (A2)

Propiedades para una medida de Complejidad

- C1: *No negatividad*

  La complejidad de un sistema S es no negativa:

  Complejidad(S) ≥ 0

- C2: *Valor Nulo*

  La complejidad de un sistema *S* puede ser nula si no existen relaciones entre sus elementos:

  R = ∅ ⇒ Complejidad (S) = 0 , donde *R* es el conjunto de relaciones entre los elementos de *S*.

- C3: *Simetría*

  La complejidad de un sistema *S* no depende del convenio elegido para representar las relaciones entre sus elementos.

  S = (E, R) ∧ S^{-1} = (E, R^{-1}) ⇒ Complejidad (S) = Complejidad (S^{-1}), donde *E* son elementos y *R* son relaciones.

- C4: *Monotonicidad de Módulos*

  La complejidad de un sistema *S* es no menor que la suma de las complejidades de cualquiera dos de sus módulos que no tienen relaciones entre sí.

  m1 ∪ m2 ⊂ S ∧ $R_{m1} \cap R_{m2} = \emptyset$ ⇒

  Complejidad(S) ≥ Complejidad(m1) + Complejidad(m2)

- C5: *Aditividad de Módulos Disjuntos*

  La complejidad de un sistema *S* compuesto de dos módulos disjuntos es igual a la suma de las complejidades de los dos módulos.

  $$m1 \cup m2 = S \wedge m1 \cap m2 = \varnothing \Rightarrow$$

  $$\text{Complejidad}(S) = \text{Complejidad}(m1) + \text{Complejidad}(m2)$$

- C6: *Monotonicidad no Decreciente*

  Esta propiedad se deriva de las cinco anteriores e indica que la adición de relaciones entre los elementos de un sistema *S* no puede hacer decrecer su complejidad.

  $$S' = (E, R') \wedge S = (E, R) \wedge R' \subset R'' \Rightarrow \text{Complejidad}(S') \leq \text{Complejidad}(S'')$$

Por último, los autores indican que las cinco propiedades C1-C5 se cumplen cuando se aplica la transformación admisible de la escala de ratio y por lo tanto, no hay contradicción entre su concepto de complejidad y la definición de una medida de complejidad en la escala de ratio.

La visión de la complejidad de estos autores se centra en que es una propiedad de un sistema que depende de las interrelaciones entre sus elementos; es decir, no es una propiedad de un elemento aislado. Por este motivo, la complejidad puede ser nula: basta que tengamos elementos no relacionados (hay autores que argumentarían que los elementos también tienen su complejidad). También señalan que la propiedad C4 es la que mejor diferencia a la complejidad de otros conceptos, haciendo énfasis en que existen detractores que discuten que la adición de relaciones a un sistema no siempre hace que este sea más complejo. Los autores se defienden indicando que ellos se centran en la *complejidad* y no en la *comprensión*, que suele ser el motivo por el que se hace la argumentacion anterior y que se debe considerar seriamente que la complejidad es sólo uno de los factores que contribuyen a la comprension.

2.5.5 Las Propiedades de Whitmire

Scott Whitmire es, por el momento, el único investigador que ha estudiado la Teoria de la Medición al completo para su aplicación exclusiva a la Orientación a Objetos [Whit94, Whit97]. En su libro [Whit97], este autor propone diferentes estructuras que un sistema empírico puede cumplir y el tipo de escala al que lleva cada estructura (así que no se centra sólo y exclusivamente en la estructura extensiva como Zuse) y lo aplica a la definición de medidas de tamaño, complejidad,

acoplamiento, suficiencia, completitud, cohesión, primitivismo, similitud y volatilidad de productos orientados a objetos. En cuanto al uso de propiedades, Whitmire defiende que dejando a un lado los requisitos impuestos por la Teoría de la Medición, una medida debe satisfacer ciertas propiedades empíricas que garantizan que la medida se comporta tal y como intuitivamente se espera de ella. Además, estas propiedades ayudan a seleccionar estructuras relacionales empíricas, discriminando a aquellas que llevarían a una medida inapropiada.

A continuación, describimos las propiedades propuestas por Whitmire.

Propiedades para una medida de Tamaño

Este autor distingue diferentes puntos de vista dentro de lo que es el concepto de *tamaño*: *población*, *longitud*, *funcionalidad*, y *volumen*. La *población* es lo que la mayoría consideramos que es el tamaño de un producto: el conteo estático de varios tipos de elementos, por lo tanto, está relacionado con la noción de pertenencia a un conjunto. Así, la cardinalidad es una medida potencial de población.

Las propiedades propuestas para la población son las cinco propiedades de Briand, Morasca y Basili [Brian96], expuestas en la sección anterior: la No Negatividad, el Valor Nulo, la Aditividad para Conjuntos Disjuntos, la Monotonicidad Creciente y la No Sinergia. Además, el autor añade una propiedad adicional impuesta por la Teoría de la Medición, común a todas las estructuras relacionales empíricas: el orden débil.

- T6: *Orden Débil*

  Sea $\bullet\succeq$ una relación binaria que denota 'población mayor o igual'. Es un orden débil:

  $$A1 \bullet\succeq A2 \ \vee \ A2 \bullet\succeq A1$$

  $$A1 \bullet\succeq A2 \Leftrightarrow \text{Población}(A1) \geq \text{Población}(A2)$$

Propiedades para una medida de Complejidad

Se centra en la complejidad estructural de los productos del diseño orientado a objetos y propone dos niveles de granularidad: el nivel de clase, en el que sus componentes se consideran atómicos y el nivel de diseño, en el que las clases se consideran atómicas y se valora la complejidad de un diseño en base a las relaciones entre las mismas y no en base a la complejidad interna de cada clase.

Las propiedades propuestas son: No negatividad, Valor Nulo y Simetría, como en Briand, Morasca y Basili [Brian96] y expuestas en la sección anterior. Añade algunas propiedades originales:

- C4: *No existe una relación fija entre la complejidad de una composición y la combinación de complejidades de sus componentes.* De hecho, la complejidad del conjunto de componentes no contribuye a la complejidad de la composición.

  Esta propiedad es absolutamente contraria a todas las propiedades de otros autores: no hay condiciones en las relaciones entre componentes y composición. El autor se basa en su experiencia, señalando que la complejidad de la composición puede ser superior o inferior a la complejidad combinada de sus partes. Además, considera que la aditividad está completamente alejada de la sensación intuitiva sobre la complejidad.

- C5: *Monotonicidad no Decreciente*

  La adición de una clase o una relación a un diseño o la adición de un atributo o un método a una clase, no puede hacer decrecer su complejidad.

  $$(a \notin C1 \wedge C2 = C1 \cup a) \vee (aRb \notin C1 \wedge C2 = C1 \cup aRb) \Rightarrow$$

  $$\text{Complejidad}(C1) \leq \text{Complejidad}(C2)$$

  donde *Ci* son clases, *a* y *b* son elementos de una clase y *R* es una relacion binaria entre elementos de una clase.

  $$(a \notin D1 \wedge D2 = D1 \cup a) \vee (aRb \notin D1 \wedge D2 = D1 \cup aRb) \Rightarrow$$

  $$\text{Complejidad}(D1) \leq \text{Complejidad}(D2)$$

  donde *Di* son grupos de clases (un diseño), *a* y *b* son clases y *R* es una relación binaria entre clases.

- C6: *Estructura Similar, Idéntica Complejidad*

  Dos objetos, ya sean clases o diseños, que tienen estructuras similares, tienen la misma complejidad:

  $$\text{Estructura}(A) \approx \text{Estructura}(B) \Rightarrow \text{Complejidad}(A) = \text{Complejidad}(B)$$

  donde *Estructura (A)* es una descripción formal de la estructura de un objeto, en términos de sus elementos y relaciones.

- C7: *La Complejidad es Independiente del Tamaño*

  El tamaño de los elementos no debe tener influencia sobre la complejidad: la complejidad debe sustentarse por sí misma.

Debemos ser capaces de distinguir estructuras más complejas en relación a otras menos complejas. Intuitivamente, podemos tener estructuras pequeñas pero complejas, estructuras grandes y simples o cualquier combinación de tamaño y complejidad. Conclusión, una medida de complejidad debe ser ortogonal al tamaño.

- C8: *Orden Débil*

  La complejidad forma un orden débil; podemos ordenar clases o diseños en términos de su complejidad relativa.

2.6 LOS METODOS DE ESTIMACION

2.6.1 Necesidad y Problemas

¿Cuánto costará? ¿Cuánto durará el proyecto? Son cuestiones notoriamente difíciles de contestar con precisión y más en las fases preliminares del proceso de desarrollo del software [Mose96a]; sin embargo, la respuesta a estas cuestiones es un requisito crítico hoy en día, considerando que el desarrollo del software es el principal factor de coste en los sistemas informáticos [Jens91].

Obviamente, una parte fundamental de la gestión de estos costes es la capacidad de las organizaciones de anticipar, tan pronto como sea posible, los requisitos de esfuerzo que el desarrollo de software demandará [Mukh92a]. Asimismo, los gestores de proyectos necesitan estimaciones precisas y fiables para poder realizar las asignaciones más adecuadas y el control de recursos. También necesitan determinar si para un tamaño de sistema concreto, el coste de un proyecto potencial es excesivamente alto, con vistas a redefinir el alcance del mismo o establecer los planes de contingencia más apropiados (por ejemplo, un análisis de riesgos).

Por otra parte, una estimación imprecisa puede tener consecuencias demoledoras en aquellas organizaciones que desarrollan software bajo contrato. En estos casos, la situación típica es aquella en la que los clientes proporcionan la especificación de las características del producto a desarrollar al personal comercial y comienza la fase inicial de negociación del contrato. En este punto, es necesario realizar una estimación de costes basada en la limitada información de la que se dispone. Una estimación imprecisa llevará a errores en la elaboración de la oferta y en consecuencia, se perderá negocio si el precio ofertado es excesivo y la organización de desarrollo tendrá pérdidas económicas si es demasiado bajo.

Es patente que los problemas de la estimación de costes y la planificación temporal siguen sin solución. Sigue habiendo proyectos de desarrollo que cuestan el doble de lo que se planificó inicialmente y cuya duración excede seriamente de la fecha estipulada de entrega.

Existen varias razones que nos llevan a realizar estimaciones imprecisas. Abdel-Hamid afirma que aunque se han desarrollado una serie de modelos de estimación, su utilidad ha demostrado ser pequeña [Abde93]. En consecuencia, muy pocas organizaciones confían en estos modelos; la mayoría elabora las estimaciones manualmente y luego, comprueba dichas estimaciones con los resultados propuestos por los modelos. También señala que los motivos por los que la estimación es problemática no es ningún misterio, mencionando un único motivo:

> «Las estimaciones de esfuerzo y coste deben realizarse en un punto en el tiempo en el que los valores de gran número de factores que afectan al desarrollo del software son desconocidos. Esto es aplicable tanto a factores de producto (como el tamaño y la complejidad), como a factores organizativos (como la capacidad del personal).» [Abde93]

Stutzke proporciona una visión algo distinta del problema [Stut96]; según él, el proceso de estimación es difícil por tres motivos:

1. Los proyectos deben cubrir objetivos conflictivos. Por ejemplo, el desarrollo de software que debe proporcionar una funcionalidad concreta con unos límites de rendimiento precisos a un precio determinado y con una fecha de entrega fija.

2. Las estimaciones son necesarias antes de que el producto final esté bien definido.

3. Los procesos de desarrollo de software están cambiando. A medida que surgen nuevos procesos de desarrollo, se necesitan nuevos métodos de estimación de costes y de planificación.

Asimismo, Reifer proporciona una lista muy completa de problemas [Reif94]:

1. Falta de acuerdo en la terminología. Las estimaciones son muy sensibles a la definición de las medidas en las que se basan, de forma que pequeñas variaciones de estos parámetros, llevan a enormes variaciones en el resultado de la estimación.

2. No se comprende bien la naturaleza de las tareas a realizar: los requisitos son vagos y volátiles. Es difícil acotar la funcionalidad a desarrollar.

3. No se considera el alcance completo de las tareas Los gestores realizan supuestos inadecuados sobre el comienzo y terminacion de las tareas, y no consideran todos los costes en sus estimaciones, bajo la creencia que otros los habran cubierto

4. Optimismo Se asume que el desarrollo tendra exito, realizando estimaciones poco realistas

5. No se utilizan datos fiables de proyectos pasados Se utilizan reglas heuristicas que no han sido validadas

6. Se carece de experiencia A veces las estimaciones las realiza una persona que carece de formacion y de experiencia en el proceso de estimacion

Todo esto nos lleva a una conclusion

«Software cost estimation is more of an art than a science » [Pilla97]

2.6.2 Clasificación de los Métodos de Estimación

Existen seis metodos principales de estimacion de recursos del software segun Reifer [Reif94]. Cada uno de ellos tiene sus propias ventajas e inconvenientes, ilustrados en la Tabla 2 6 y la seleccion de uno de ellos en concreto depende de las preferencias del gestor, de la disponibilidad de datos para las calibraciones, del nivel de conocimiento de los estimadores y de la madurez del proceso A continuacion, se consideran brevemente cada uno de los tipos o modelos de estimacion

- Metodo de Estimacion por Analogia

  Se desarrollan estimaciones de recursos basadas en experiencias del pasado con sistemas similares, utilizando un *catalogo* o el concepto de *refinamiento por parejas* Los catalogos capturan datos historicos sobre los que se pueden realizar estimaciones. Bajo el refinamiento por parejas, las estimaciones se preparan mediante comparaciones de las predicciones contra unos modelos de referencia de los que se conoce el tamaño dificultad y coste Finalmente, las estimaciones de los componentes se ordenan y suman para generar la estimación total

- Metodo de Estimación Ascendente o Basado en Actividades

  Se desarrollan estimaciones de recursos a nivel de actividad o tarea (diseño, código, prueba, etc) o a nivel de componente (programa, versión, producto, etc) utilizando Analogias, PERT,... cualquiera de las demás técnicas, ya que no

son mutuamente excluyentes, se solapan y se pueden utilizar simultáneamente. Se acumulan las estimaciones por actividad o por elemento de producto de acuerdo con la Estructura de Descomposicion de Actividades para elaborar la estimacion final. Generalmente, la duracion suele ser un parámetro impuesto. Se suelen ajustar las estimaciones de esfuerzo con el fin de determinar qué se necesita para concluir el proyecto y compensar los riesgos detectados y planificados.

- Método de Estimación Paramétrico u Holístico

Se elaboran las estimaciones de recursos utilizando modelos predictivos que relacionan matemáticamente el esfuerzo y la duración con el conjunto de parámetros que tienen influencia sobre ellos (por ejemplo, el tamaño). También se les denomina *holísticos* porque estiman considerando el sistema como un todo, sin detenerse en las estimaciones de las actividades individuales [Dean89, SPC94]. El uso de estos modelos está muy extendido y además, existen herramientas que los soportan. Entre ellos se encuentran el modelo COCOMO, SLIM, PRICE-S y SOFTCOST. Cada uno de ellos tiene su propia filosofía y matemática de estimación. En la Figura 2.3 se ilustra una clasificación de los modelos paramétricos.

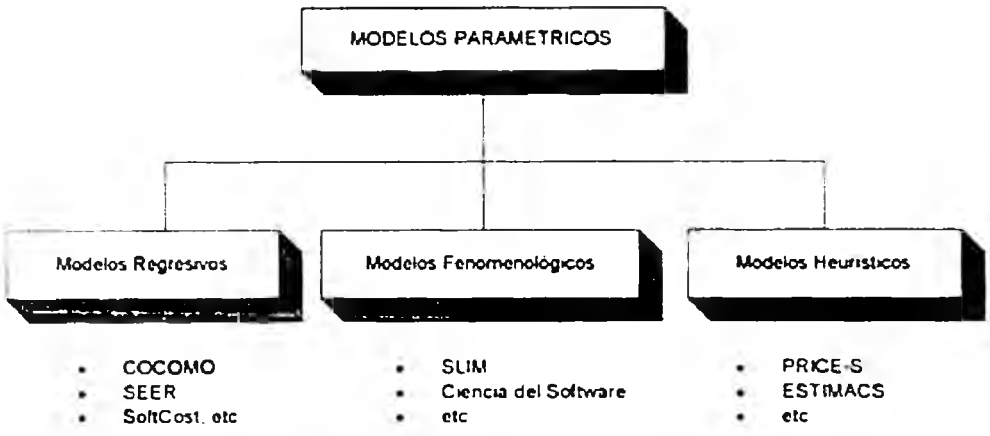

Figura 2.3 Clasificación de Modelos de Estimación Paramétricos.

- Método PERT

Las estimaciones de esfuerzo se desarrollan suponiendo una distribución de probabilidades normal o cualquier otra (p.e. una distribución beta) para las estimaciones del peor caso posible (*l*), el mejor caso posible (*h*) y el caso más probable (*m*) de esfuerzo y duración, usando la siguiente expresión:

$$\text{Esfuerzo} = l + 4m + h/6$$

Este enfoque permite a los estimadores compensar el riesgo de elaborar una estimación ponderada. Los valores de las tres variables se suelen determinar utilizando la Estimación por Analogía o el método Delphi. El inconveniente de este método es que la expresión presupone que no hay correlación entre las estimaciones de cada elemento.

- Modelo Descendente o de Estimación del Sistema

  Se asignan estimaciones de recursos a actividades específicas de la Estructura de Descomposición de Actividades, basadas en experiencias del pasado. Se pide a los estimadores que propongan estimaciones de coste; los conflictos que surgen se resuelven de forma iterativa, ya que los desarrolladores determinan cómo cubrir los requisitos con recursos limitados. En este caso también, la duración de proyecto suele ser un parámetro impuesto, que se suele ajustar para compensar un incremento de coste desmedido.

- Modelo Delphi

  Se desarrolla la estimación de recursos utilizando a un conjunto de expertos. Cada experto estima los recursos necesarios bajo supuestos similares, pero usando métodos distintos. A continuación, el equipo se reúne e intentan llegar a un consenso sobre el valor de la estimación total. Las disputas se resuelven mediante análisis. La filosofía que subyace en este método es la búsqueda de inconsistencias en la estimación y el intento de resolver dichas inconsistencias.

Walkerden y Jeffery [Walk96] completan la clasificación de Reifer (resumida en la Tabla 2.6), aportando tres categorías adicionales:

- Modelos No-paramétricos

  Son modelos que carecen de un mecanismo funcional específico. Se centran en el hecho de que el dominio de la estimación de recursos del software no dispone de un modelo causal sólido basado en fundamentos rigurosos y que se sitúa en un entorno de procesos de alta volatilidad y enormemente dependientes del contexto [Mukh92]. Esto justifica el desarrollo y uso de modelos centrados en técnicas de aprendizaje automático, redes neuronales, etc., que ofrecen mecanismos adaptativos adecuados para abordar este problema.

- Modelos Analógicos Automatizados

  Son similares a los anteriores en el hecho de que carecen de un mecanismo funcional específico. Su base es el uso del razonamiento analógico; se trata de

intentar automatizar los procesos que los expertos llevan a cabo y realizar las estimaciones mediante Razonamiento Basado en Casos.

- Modelos Teóricos

Son modelos creados mediante lo que se podría denominar la aplicación de la Dinámica de Sistemas a la Ingeniería del Software. Se fundamentan en la aplicación de los principios de los sistemas de control realimentados al modelado, análisis y comprensión del comportamiento dinámico de sistemas complejos, es decir, su evolución temporal. Así, los proyectos de desarrollo de software se consideran sistemas dinámicos socio-tecnológicos complejos, cuya evolución temporal vendrá dada tanto por su estructura, como por las políticas de dirección empleadas y las relaciones establecidas entre el personal técnico.

Metodo	*Ventajas*	*Inconvenientes*
Analogia	• Basado en la experiencia • Valoración de las similitudes y las diferencias	• Posibilidad de que la experiencia pasada no sea aplicable o este obsoleta • Posibilidad de confusion en las similitudes y diferencias
Ascendente	• Análisis detallado de las actividades implicadas • Estimaciones relacionadas con aspectos especificos	• Posibilidad de pasar por alto factores de nivel de sistema • Necesidad de disponer de mucho tiempo y esfuerzo
Parametrico	• Objetivo y repetible • Consideracion de múltiples factores • Consideracion de factores de escala	• Calibracion difícil • Subjetividad de determinados factores • No aplicable a todos los dominios
PERT	• Consideración de riesgos • Necesidad de valorar el problema	• Suposicion de que las estimaciones no estan condicionadas • Dificultad para obtener entradas adecuadas
Descendente	• Se centra en el nivel del sistema • Permite la asignación por cada actividad del Diagrama de Descomposicion de Actividades	• Posibilidad de pasar por alto detalles • No admite contingencias
Delphi	• Objetivo utiliza más de una persona o método	• La precisión depende de los conocimientos de los expertos involucrados

Tabla 2.6 Ventajas e Inconvenientes de los Métodos de Estimación.

Dependiendo de las circunstancias, cada método tiene sus puntos fuertes y débiles. Para proyectos pequeños, la Analogía parece funcionar bien. En los grandes, se utilizan los Paramétricos porque consideran la experiencia pasada. En proyectos de riesgo, se prefiere el método Delphi porque implica a un conjunto de expertos que intentarán acotar el riesgo. En cualquier tipo de proyecto, el método Basado en Actividades es adecuado porque proporciona una lista de tareas que recuerda que se debe incluir el esfuerzo para actividades que se podrían olvidar.

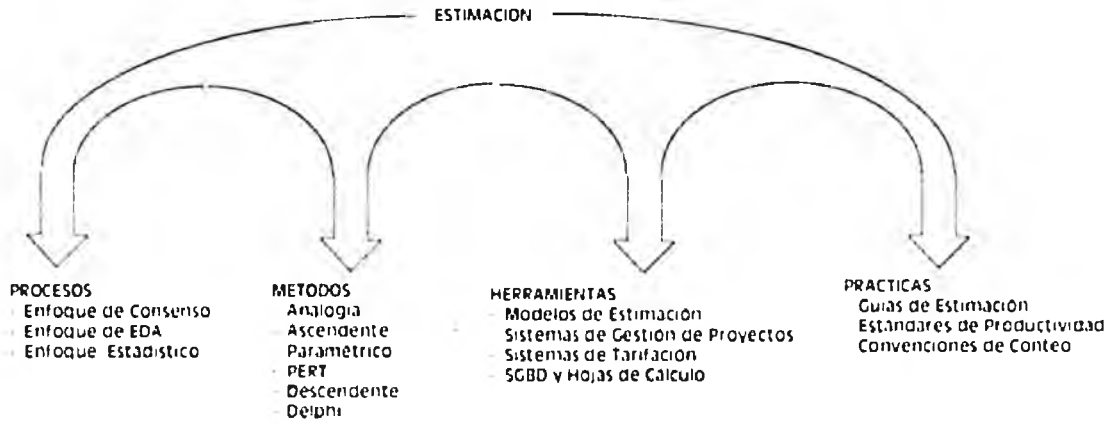

Figura 2.4 Estado del Arte de la Estimación según Relfer.

Además, en cierta medida estos métodos mantienen una correspondencia con la progresion en los niveles de madurez de procesos de SEI [SPC94]. Si una organización de desarrollo de software se encuentra en el nivel de madurez 1, el *inicial*, probablemente no disponga de datos de proyectos pasados en una base de datos En este caso, la estimación deberá realizarse con un modelo paramétrico con las constantes sin calibrar

Para cuando una organización alcanza el nivel de madurez 2, el nivel de *repetición de procesos*, ya dispondrá de una base de datos con información de experiencias pasadas como para poder calcular los costes de las actividades principales de desarrollo de software (pe. definición de requisitos, diseño, codificación y prueba, integración y prueba) y utilizar un modelo Basado en Actividad o Ascendente.

Si la organización alcanza el nivel de madurez 3, nivel de *definición*, se dispondrá de suficiente experiencia y datos como para realizar estimaciones más precisas y para estimar el coste de la documentación, prueba y soporte al software.

Si se alcanza en nivel 4, nivel de *gestión*, se podrán utilizar los métodos Descendentes o de Estimación del Sistema al disponer de suficiente experiencia.

Finalmente, en el nivel 5, el nivel *óptimo*, la organización habrá alcanzado la suficiente sofisticación como para poder gestionar los riesgos y costes de mantenimiento dentro de la rutina habitual de gestión.

A medida que se va incrementando el nivel de madurez, se irán acumulando datos y experiencia que se utilizarán para ajustar los valores de los parámetros de las ecuaciones de los modelos.

2.6.3 Métodos de Estimación: Pasado y Presente

En esta sección, se describe la evolución de las técnicas de estimación de recursos en el contexto del desarrollo del software [Stut96]. A continuación, se ofrece una descripción detallada de los métodos más interesantes y que más expansión han tenido en la industria. En la Tabla 2.7 se muestra un resumen de dichos métodos [Zuse97].

Método	*Organización*	*Año*
Delphi	Rand Corp	1966
SDC de Nelson	SDC	1966
Wolverton	TRW	1974
PRICE-S	RCA	1976
Halstead		1977
Walston & Felix	IBM	1977
Método de Punto de Función		1979
Método COCOMO	TRW	1981
SOFTCOST	JPL	1981
Bailey & Basili	NASA	1981
Metodo BANG		1982
ESTIMACS		1983
SPQR	SPR	1986
Puntos de Función MARK II		1988
Metodo de Pfleeger		1989

Tabla 2.7 Principales métodos de Estimación.

El estudio formal de las técnicas de estimación del software no comenzó hasta la década de los 60, aunque las primeras investigaciones datan de 1958 y son las de Peter Norden [Nord58].

Durante los 60, Freiman desarrolló el concepto de *estimación paramétrica* que llevó al desarrollo del modelo PRICE para el hardware. Fue el primer método automatizado de disposición general. Fue extendido para que se ajustara al software

en los 70. A mediados de los 60, se introdujo el modelo de estimación de coste Delphi [Helm66] y el SDC de Nelson [Nels66].

La década de los 70 fue un periodo muy activo. Fue creciendo la necesidad de predecir con precisión los costes y la planificación del desarrollo del software y se le empezó a prestar más atención. Cada vez se estaban desarrollando sistemas de mayor envergadura y muchos de los proyectos del pasado habían sido verdaderos desastres financieros. Fred Brooks, estando en IBM, describió muchos de estos problemas en su libro *The Mithical Man-Month* [Broo74, Broo75, Broo95]. Es un clásico que proporciona una idea muy real de los problemas de la época.

Durante los 70, se comenzaron a utilizar de forma masiva los lenguajes de programación de alto nivel como el FORTRAN, ALGOL, Pascal,... que no soportaban la reusabilidad. Por este motivo, los sistemas realmente se desarrollaban partiendo de la nada. Así, los modelos de estimación de este periodo contemplan únicamente los aspectos relacionados con el desarrollo de productos nuevos. Muchos de los autores de esta época, se centraron en los modelos paramétricos, basados en el uso de la estadística para identificar los factores significativos que influyen en el coste (mediante técnicas de correlación) y la creación de modelos (mediante técnicas de regresion), cuyos coeficientes se elegían entre los que mejor se ajustaban a los datos del pasado. De esta epoca datan el modelo COCOMO de Barry Boehm [Boeh81] y PRICE-S de Freiman y Park [Frei79, Park88]. El inconveniente principal de estos modelos es que se basan en estimaciones del tamaño de un producto, por lo tanto, estan supeditados al exito de estas estimaciones.

A finales de los 70, Allan Albrecht y John Gaffney de IBM desarrollaron el *Analisis de los Puntos de Función* para la estimación del tamaño y esfuerzo de desarrollo de los sistemas de información de gestión [Albr79, Albr83]. En esta época, se realizó la tentativa de definir un modelo basado en aspectos formales: Putnam basó su *Modelo del Ciclo de Vida del Software* (SLIM - Software LifeCicle Model) en la curva de Raleigh-Norden y en datos de proyectos del ejército [Putn78, Putn79].

Durante los 80, la tendencia fue a consolidar y mejorar los mejores modelos. Durante estos años, aparecieron los PC que facilitaron la automatización de dichos modelos. Surgió el lenguaje ADA, cuyas características tuvieron impacto en el desarrollo y mantenimiento. Así, en 1987 se publicó la revisión del modelo COCOMO para Ada, denominado ADA-COCOMO [Boeh87]. Este modelo también considera que el ciclo de vida puede ser incremental, ajustándose a los inevitables cambios en los requisitos, considera la gestión del riesgo, la generación rápida de prototipos y el uso de algunas prácticas de desarrollo modernas.

Tausworthe extendio el trabajo de Boehm, Putman, Walston y Felix [Wals77] y Wolverton [Wolv74] para desarrollar un modelo de estimación de coste para la NASA [Taus81]. Este modelo fue posteriormente extendido por Donald Reifer, creando el modelo automatizado para PC SOFTCOST. En 1981, Bailey y Basili sugirieron que la estimación del esfuerzo debería depender del entorno y personal concreto de una instalación. Propusieron su método de estimación utilizando datos de proyectos desarrollados para la NASA, por lo tanto, datos muy homogéneos; este metodo rechaza la idea de un estimador universal. En 1982, Tom DeMarco propuso su método BANG [DeMa82] que se centra en medir las funciones que el sistema debe entregar tal y como las percibe el usuario.

Jensen [Jens83, Jens84] amplio las investigaciones de Putnam eliminando algunos de los inconvenientes. Hoy en dia, este modelo está también automatizado: la herramienta se llama SEER y la versión 4.0 salió en 1994. Rubin desarrolló el modelo ESTIMACS en 1983 [Rubi83]. Es un modelo heurístico basado en el tamaño, cuyo producto incluye módulos para estimar el esfuerzo, coste y personal de desarrollo de un sistema, para estimar la configuración hardware necesaria para el sistema y para estimar el riesgo.

A mediados de los 80, Caper Jones amplió el Análisis de los Puntos de Funcion para incluir el efecto de los sistemas con algoritmos computacionalmente complejos, llamándolo *Análisis de los Puntos de Característica* [Jone86, Jone87]. Tambien propuso un modelo de estimación llamado SPQR (Software Productivity, Quality and Reliability) parecido a COCOMO y con 45 factores que afectan al coste y a la productividad. Es un producto comercial que requiere la respuesta del usuario a 100 cuestiones para poder realizar las estimaciones [Jone86b]. Symons [Symo88, Symo91] propuso otra revisión de los Puntos de Función, llamada Mark II. En 1984, Thebaut *et al* propusieron su modelo COPMO (*Cooperative Programming Model*) [Theb84, Cont86]. En este modelo, la ecuación del esfuerzo total incluye dos términos: uno que corresponde al esfuerzo consumido en actividades relacionadas con la programación y otro al consumido en la coordinación de tareas entre los miembros del equipo de desarrollo.

Durante los 90, debido a la gran diversidad de procesos de desarrollo de software, se está prestando mucha atención a la mejora de los modelos de estimación. El SEI comenzó un proyecto sobre el tema en el 94, pero es más interesante el esfuerzo de Boehm por revisar y extender COCOMO. La nueva versión denominada COCOMO II [Boeh95, Boeh95a, USC97] contempla la reusabilidad y algunos factores como el nivel de madurez del proceso del SEI, la dispersión geográfica del equipo de desarrollo, etc. También se han revisado algunos coeficientes.

Por otra parte, como Reifer afirma:

> «Se necesita innovación en los métodos de estimación para que contemplen los sistemas orientados a objetos, la reusabilidad, los lenguajes de cuarta generación y los generadores de aplicaciones. Además, hay que considerar que la mayor parte de los métodos tradicionales necesitan una medida de tamaño del sistema, generalmente LOC, con los inconvenientes que conlleva. Se debe investigar en métodos alternativos de estimación que no se centren este tipo de medidas» [Reif94]

Así, en los últimos años se han publicado métodos basados en diferentes medidas como el de Verner y Tate [Vern92] para los 4GL en el que se identifican y cuentan objetos como pantallas e informes y a los que se les asigna un peso dependiendo del número de elementos de los que están compuestos. Estos mismos autores también sugieren un método para calcular el tamaño basado en contabilizar las entradas de un diccionario de datos en un entorno CASE [Tate91]. También se han desarrollado métodos como el de Lo *et al.* [Lo95] para la estimación del esfuerzo de desarrollo y prueba de un sistema GUI, realizado usando Programación Visual. En su método, se relaciona el esfuerzo con el número de los distintos tipos de controles (*widgets*) que se utilizan en las pantallas.

Otro modelo que resulta interesante es el modelo teórico de Abdel-Hamid y Madnick [Abde86, Abde91, Abde93] que simula dinamicamente escenarios de gestión para ver los efectos que determinadas políticas tienen sobre la duración, coste y esfuerzo. Modeliza las complejas relaciones entre la gestión de recursos humanos y la produccion del software. Curiosamente, la conclusión que se deriva del uso de su modelo es que la Ley de Brooks [Broo74] no siempre es cierta: añadir personal a un proyecto retrasado, hace que el proyectos sea más caro, pero no tiene por qué retrasarlo mas.

Por último, cabe decir que todos los modelos de estimación de coste siguen enfrentándose al mismo obstáculo: no existen modelos teóricos válidos del desarrollo del software. Por lo tanto, no hay unas 'leyes' de la *Física* del Software que definan las relaciones lógicas y las restricciones entre las distintas variables independientes presentes en un entorno de proyecto. El desarrollo del software sigue siendo una actividad altamente intelectual y por tanto, depende del pensamiento humano. Hasta que la Psicología pueda construir modelos cuantitativos sobre la resolución de problemas por parte de los seres humanos, hay pocas esperanzas de desarrollar una *Física* del Software. En consecuencia, la estimación de recursos seguirá siendo una ciencia experimental. A los que realizan las estimaciones no les queda más remedio que confiar en su juicio e intuición para definir reglas heurísticas y validarlas con datos de proyectos reales [Stut96, Hend96].

2.6.3.1 Los Puntos de Función de Albrecht

Allan J. Albrecht, estando en IBM, presentó en 1979 la primera versión del *Análisis de los Puntos de Función* [Albr79], un nuevo tipo de métrica orientada a realizar estimaciones de coste y esfuerzo, sin depender de las líneas de código. Básicamente, se diferencia de todas las demás métricas en que no está dirigida a la implementación, sino que a la fase de análisis y especificación de requisitos. Este método ha sufrido diversas variaciones y extensiones, e incluso se ha constituido un *Grupo Internacional de Usuarios de los Puntos de Función* (IFPUG) que ha publicado unas directrices de uso de la métrica.

La métrica de los puntos de función pretende medir la funcionalidad de un producto de software con unas unidades estándar, independientes del lenguaje de programación [SPC94]. Un punto de función es una medida de funcionalidad basada en el número estimado de elementos externos (entradas, salidas, consultas e interfaces) de un sistema, más el número estimado de archivos lógicos internos. La idea básica es que una especificación de requisitos de software con cierta frecuencia describe el comportamiento externo y visible del producto potencial. Esta medida tiene una relación directa con esta visión.

El procedimiento de cálculo es el siguiente: se estima la complejidad de cada elemento de las cinco categorías en 'baja', 'media' o 'alta' y se multiplica el sumatorio de cada categoría por un peso. El sumatorio total proporciona el *número de funciones* o la *cuenta de función*. Esta es una medida del tamaño del sistema sin ajustar.

El siguiente paso es el calculo del *factor de ajuste de la complejidad*, mediante la valoración del impacto de 14 factores que afectan al tamaño funcional del sistema. Algunos factores son los siguientes: las comunicaciones de datos, la captura de datos interactiva, la reusabilidad del código, la facilidad de instalación, la facilidad de operación, la facilidad de cambio, etc. Cada uno de los factores anteriores se valora en una escala desde 0, que significa *factor no presente* o *sin influencia*, hasta 5, que indica *influencia esencial*.

Una vez los 14 factores han sido valorados, se suman y se ajustan para formar un *multiplicador de ajuste de complejidad*. Por último, se halla el producto del multiplicador anterior por el número de puntos de función sin ajustar.

Esta medida sólo se puede calcular a partir de documentos de especificación de software detallados y completos, es independiente del lenguaje utilizado pero parece que no del método de análisis y diseño utilizado [Vern89]; además, sólo tiene correlación con el esfuerzo en las aplicaciones de proceso de datos. Incluso se ha

buscado la correspondencia entre los puntos de función y las líneas de código, para distintos lenguajes de programación y aplicaciones. Para su uso, una organización debe conocer la productividad de sus desarrolladores en puntos de función/persona-mes, para así, poder averiguar el esfuerzo de cualquier proyecto de desarrollo.

También se debe considerar que su cálculo implica un grado bastante alto de subjetividad; por tanto, su recopilación no se puede automatizar de forma completa. Debido a esto, hacen falta unas normas de contabilidad detalladas para que haya un cierto nivel de consistencia. Es necesario distinguir entre entradas, salidas y consultas y determinar qué es un archivo lógico [Fent91]. Los pesos de complejidad, por otra parte, provienen de la experiencia de IBM y pueden no servir en otras organizaciones.

Algunos de los inconvenientes anteriores han sido solucionados por varios estudios entre los que destacan los de Jones para adaptar los puntos de función a los sistemas de control y Symons, que se centra en reducir la subjetividad al tratar ficheros y hacer el tamaño independiente del hecho de que se implemente el sistema como un todo o como un conjunto de subsistemas relacionados. Su método, denominado Puntos de Función Mark II, basa el cálculo del tamaño (o funcionalidad) en las *transacciones lógicas*. Cada actividad de procesamiento realizada por el sistema se analiza en términos del número de elementos de datos de entrada, de datos referenciados y de datos de salida. Se contabilizan y ponderan para hallar el *tamaño* en puntos de función.

Tambien ha habido algunas propuestas con respecto a los sistemas Orientados a Objetos:

- Los *Puntos de Objetos* de Sneed [Snee94] o *Puntos de Datos* son una simplificación de los puntos de función que dejan de lado la parte funcional. Es un método útil para una aplicación basada en una base de datos, ya que simplifica la tarea de modelización y medida.

- Los *Puntos de Tarea* de Graham [Grah95] son una alternativa a los puntos de función. Es el número de tareas atómicas en un sistema y considera la complejidad total. Las estimaciones de esfuerzo se derivan de la expresión: $E = a + pT^k$, donde E representa el esfuerzo (personas-hora), T es el número de puntos de tarea, p es la inversa de la productividad en puntos de tarea/personas-hora (una constante empírica) y por último, a y k son constantes. Graham ofrece un completo conjunto de medidas como son el número de objetos externos en el modelo de contexto, la complejidad ponderada de cada tarea, los grados de entrada y de salida, la profundidad de

las estructuras y el número de tareas atómicas (nodos terminales en un árbol de tareas) [Hend96]. Este método no se ha probado de forma completa y se ha creado un club cuya función es precisamente recoger datos y fomentar su uso: el *IOMeC, International Object Oriented Metrics Club* [Grah96].

2.6.3.2 COCOMO de Boehm

Probablemente el modelo de Boehm [Boeh81, Boeh95, Boeh95a, USC97] sea el de uso más extendido hoy en día. Si no se disponen de datos de productividad propios de la organización, se pueden utilizar las propias constantes del modelo. Si se dispone de datos de experiencias pasadas, se deberán usar los parámetros derivados de la propia experiencia. El modelo básico de predicción de esfuerzo tiene la siguiente expresión:

$$\text{Esfuerzo} = a\,(\text{Tamaño})^{b}$$

Existen tres modelos básicos o tipos de aplicación: el organico, el semi-acoplado y el embebido. También existen tres niveles: el basico, el intermedio y el detallado.

El modo orgánico se aplica a desarrollos considerados en el rango de pequeños a medianos en un entorno de desarrollo conocido. El modo embebido representa una situacion de desarrollo con restricciones rigidas, donde el producto opera en un sistema interactivo, con un software/hardware complejo. Los sistemas militares, el software de control de tiempo real, el software de control de tráfico aéreo, ... son algunos ejemplos de este tipo de productos. El modo semi-acoplado es el intermedio entre el organico y el embebido.

Boehm también proporcionó la distribucion de esfuerzo y tiempo para las actividades principales del desarrollo de software, teniendo en consideración los distintos modos y tamaños de sistemas.

El nivel de COCOMO intermedio es una extensión del nivel básico que incluye unos *factores multiplicadores de coste* que modifican las estimaciones de esfuerzo (personas-mes) elaboradas con el modelo básico. Estos factores se pueden clasificar en cuatro grupos:

1. Atributos de Producto: Fiabilidad requerida del software, Tamaño de la Base de Datos y Complejidad del producto.

2. Atributos del Ordenador: Restricciones de Tiempo de Ejecución, Restricciones de almacenamiento de memoria y Volatilidad del Almacenamiento Virtual

3. Atributos de las Personas: Capacidad de los analistas, Experiencia en la aplicación, Capacidad de los programadores, Experiencia con la máquina virtual y Experiencia con el lenguaje de programación.

4. Atributos del proyecto: Uso de Prácticas de programación modernas, Uso de herramientas de software y Existencia de una planificación de desarrollo.

A cada uno de los atributos anteriores se le asigna un valor de forma subjetiva, ponderando su influencia en la estimación de esfuerzo entre 'Muy Baja' hasta 'Extra Alta'. A cada valoración le corresponde un número.

La expresión genérica de la estimación de esfuerzo en el modelo es la siguiente:

$$ESF = a(KDSI)^b \cdot \prod_{i=1}^{n} M_i$$

donde M_i son los valores asignados a los factores multiplicadores de coste. Estos multiplicadores no son completamente independientes los unos de los otros, aunque el modelo los trata como tal. Así, si se usa un número elevado de multiplicadores, el efecto acumulativo de todos ellos afectarán seriamente a la precisión de las estimaciones, que pueden ser o excesivas o insuficientes.

El nivel detallado proporciona dos funciones de las que carece el modelo intermedio que son el uso de multiplicadores de esfuerzo sensibles a la fase en la que se aplican y sensibles a tres niveles de granularidad (módulo, subsistema y sistema).

Este modelo ha sido extendido para adaptarse a un lenguaje como ADA. También se han propuesto extensiones o modelos basados en COCOMO para considerar la reusabilidad e incluso para estimar el coste de un desarrollo utilizando como paradigma el desarrollo evolutivo en espiral [Bald90].

2.6.3.3 Métodos de Estimación No Paramétricos

Como ya sabemos, el proceso de desarrollo del software es una actividad compleja y la información que se puede obtener no se caracteriza precisamente por presentar los atributos necesarios para la construcción de modelos: no se puede asumir que esta información siga una distribución normal, puede tener desviaciones y valores atípicos, las variables explanatorias pueden no ser independientes, etc. Esto supone un problema para el uso de modelos paramétricos (estadísticos) [Bria92, MacD96, Gray97]. Además, parece también poco probable que un simple modelo estadístico represente de manera adecuada el mundo real.

Este es el motivo por el que en los últimos años han emergido otros enfoques dirigidos al desarrollo de modelos que provienen del campo de la Inteligencia Artificial. Estos enfoques se centran en el uso de la Lógica Difusa, las Redes Neuronales, los Algoritmos Genéticos, los Arboles de Regresión y el Razonamiento basado en Casos, enfoque este último que se comenta en la siguiente sección.

Así, por ejemplo, Selby y Porter [Selb88] describen uno de los primeros trabajos en la generación de árboles de decisión o clasificación para la estimación del esfuerzo. La construcción de los árboles se realiza utilizando datos de proyectos pasados, de los que van seleccionando iterativamente los atributos que mejor dividen los datos en poblaciones disjuntas. La predicción de esfuerzo para un caso nuevo se realiza descendiendo por el árbol de decisión siguiendo el camino adecuado, hasta llegar a la hoja del árbol, donde se encuentra el valor del esfuerzo estimado.

Srinivasan y Fisher [Srin95] también describen un enfoque de aprendizaje automatico para estimar el esfuerzo de desarrollo utilizando un algoritmo para la construcción de *árboles de regresión* (son árboles de decisión denominados así porque el objetivo de la categorización es generar una predicción en una dimensión continua).

Estos mismos autores describen un metodo de estimacion basado en una red neuronal, llamado BACKPROPAGATION, mediante el que obtienen unos resultados con un error relativo medio menor que en el caso de los arboles de decision, aunque el coste computacional de entrenar la red neuronal es mayor que el de derivar un árbol de decision. El enfoque de las redes neuronales se ha aplicado también en otras areas de modelizacion de métricas, como la fiabilidad, el riesgo de mantenimiento, etc. y los resultados, en general, han sido favorables a esta técnica. Sin embargo, este metodo tambien tiene inconvenientes [MacD96, Gray97]. En primer lugar, su uso requiere un conocimiento profundo del tema, no es sencillo, y en segundo lugar, está su naturaleza encapsulada, donde conocemos las entradas y las salidas pero el proceso que convierte las unas en las otras queda oculto

Otro método a mencionar es el propuesto por Briand, Basili y Thomas [Bria92] que describen un método de reconocimiento de patrones llamado *Reducción de Conjuntos Optimizada* (OSR - Optimized Set Reduction) que se basa en parte en el uso de árboles de decisión. Esta técnica elige un subconjunto de proyectos en los que basar la predicción de la productividad de un nuevo caso; estos proyectos comparten características comunes con el nuevo proyecto, características que se eligen en base a un criterio estadístico. Se deriva una distribución de probabilidad de la distribución de frecuencias de los proyectos seleccionados en un rango de

intervalos de productividad. Así, la productividad de un nuevo proyecto se estima calculando su valor según la distribución de probabilidad. La validación de este método demostró que se comportaba mejor que los métodos paramétricos con los que se comparó. Su ventaja principal es que se puede aplicar utilizando datos de entrada incompletos y se pueden usar datos de la escala nominal y ordinal sin necesidad de establecer una correspondencia numérica con ellos.

2.6.3.4 Métodos de Estimación Analógicos

Curiosamente, si se le pregunta a una persona que se dedica a la gestión de proyectos a cerca del método utilizado para hacer estimaciones, contestará que las realiza de forma intuitiva, considerando su experiencia pasada y las diferencias que el proyecto en estudio tiene con respecto a otros proyectos similares. Es lo que en la sección 2.6.2, Reifer denomina 'Estimación por Analogía'.

Este método ha sido denostado en la literatura y considerado como 'un ejercicio de adivinación'. Sin embargo, existen datos que indican que es el método más ampliamente utilizado en el mundo del desarrollo del software y que siendo un método sujeto a la intuición personal, a la experiencia y a la subjetividad en general, los resultados que se obtiene no son nada desalentadores si se comparan con los que se obtienen con métodos supuestamente más precisos [Keme87, Vicl91, Höst97, Lede98]. Así, Hughes sugiere que las investigaciones dirigidas hacia la estimación de proyectos deberían desarrollar modelos y sistemas de información cuyo objetivo sea dar soporte a la opinión de los expertos en vez de intentar reemplazarlos [Bisi95, Hugh96]. Briand *et al.* señalan que el método de estimación más extendido es *la comparación con proyectos pasados similares basados en la memoria personal* [Bria97].

La situación anterior ha llevado a explorar otros enfoques en la estimación de recursos: los métodos analógico que pretenden automatizar los proceso que los expertos realizan al hacer una estimación; es decir, métodos que incorporan el Razonamiento basado en Casos.

Cowderoy y Jenkins [Cowd88], como parte de un proyecto ESPRIT de la CEE, desarrollaron una herramienta de estimación que incorpora una base de datos con información de proyectos pasados, así como conocimiento sobre prácticas de ingeniería del software en general y sobre la organización en particular, además de integrar otras herramientas de gestión. El núcleo de su trabajo fue la definición de un meta-modelo de estimación por analogía, que ha sentado las bases de las investigaciones posteriores. Este meta-modelo se puede implementar con sistemas complejos como un sistema experto, pero es suficientemente simple como para

implementarlo en una hoja de cálculo o mediante un procedimiento manual que permita la captura de datos y la transición posterior a una herramienta sofisticada. El meta-modelo comprende las siguientes actividades

- Selección de analogías: esta labor la debe realizar el gestor del proyecto, ayudado de una herramienta que facilite el acceso a la información. Puede clasificar la información, basándose en atributos sobre preferencias o en inferencias que un pequeño sistema experto puede realizar

- Valoracion de similitudes si las diferencias entre el entorno del proyecto actual y el de la analogia son sustanciales, debera rechazarse la analogia Este paso es automatizable

- Valoración de la propia calidad de la analogía es difícil hacer un buen seguimiento a un proyecto terminado retrospectivamente es decir, capturar información precisa sobre el desarrollo a posteriori Las analogías basadas en este tipo de información pueden resultar arriesgadas Si se dispone de analogías potencialmente mas fiables, es lógico descartar aquellas que pueden ser de peor calidad

Una vez se ha elegido una analogía se puede utilizar un procedimiento estadístico o de otro tipo para valorar la similitud y finalmente, se obtiene una solucion la estimacion para el proyecto nuevo

Mukhopadhyay, Vicinanza y Prietula [Vici90, Mukh92] propusieron un metodo basado en razonamiento basado en casos denominado ESTOR Este metodo utiliza cinco procesos básicos

- Construccion de una representacion del problema objetivo

- Recuperación de un caso adecuado para que actúe como analogía base

- Transferencia de la solucion de la base al objetivo

- Correspondencia de las diferencias entre la base y el objetivo

- Ajuste de la solucion inicial para tomar en consideracion las diferencias

En ESTOR, los casos son proyectos de software y se representan mediante los valores de un conjunto de medidas: en concreto los componentes de los Puntos de Funcion y las entradas al modelo intermedio de COCOMO ESTOR recupera un solo caso para que actúe como fuente, basándose en los valores de los componentes de los puntos de funcion del proyecto objetivo

Para hallar el vecino más cercano utiliza el cálculo de la distancia vectorial. La solución inicial, el esfuerzo del proyecto nuevo, es el valor del esfuerzo del proyecto análogo Las diferencias entre el análogo y el nuevo se establecen comparando los valores de sus medidas. El valor del esfuerzo del análogo se ajusta mediante el uso de reglas de producción para tomar en consideracion dichas diferencias. Las reglas que ESTOR utiliza se derivaron de los protocolos verbales de un experto cuyas estimaciones fueron precisas para el conjunto de datos usado. Estas reglas ajustan el valor del esfuerzo en forma de un multiplicador, si se cumplen determinadas precondiciones entre los valores del proyecto nuevo y el base

La validacion realizada por los autores indica que aplicando ESTOR, se obtuvo un error relativo menor que con los metodos con los que fue comparado los Puntos de Funcion y COCOMO, es decir, actua con una precision y una consistencia bastante fiables.

Otro enfoque muy similar al de ESTOR y centrado en el meta-modelo de Cowderoy y Jenkins es el implementado en la herramienta ANGEL de Schofield y Shepperd [Scho95, Shep96 Shep96a Shep97] En este metodo, los proyectos analogos a uno dado tambien se localizan mediante la distancia vectorial y el esfuerzo del proyecto nuevo es una media ponderada del esfuerzo de sus vecinos Con esta herramienta, el usuario puede especificar con que medidas desea buscar las analogias, ademas, la herramienta puede sugerir cual es el subconjunto de medidas optimo para un juego de datos, basandose en la minimizacion de la media de la magnitud del error relativo Las diferencias principales con ESTOR son dos ESTOR unicamente recupera una analogia y en ANGEL, podemos elegir entre una dos tres o cuatro analogias y la diferencia mas importante ESTOR ajusta el esfuerzo de la unica analogia mediante reglas de produccion, mientras que en ANGEL, si se recupera una analogia, se utilizara su esfuerzo directamente y si se recuperan varias, se calculara una media

La validacion de ANGEL tambien demostro que los resultados eran mejores que los proporcionados por metodos estadisticos tradicionales Los autores tambien demostraron que se pueden realizar estimaciones fiables con juegos de datos pequeños, siempre que sean del mismo entorno Asi, en [Shep96a] se ilustra un ejemplo en el que la muestra de casos era pequeña, sólo 8 proyectos, y ANGEL tuvo una media de la magnitud del error relativo del 60 %, mientras que el modelo de regresion lineal tuvo un 226 %, usando el mismo juego de datos

Bisio y Malabocchia [Bisi95] proponen un método similar a los anteriores centrandose en la siguiente hipótesis *los proyectos similares tienen costes similares* Estos autores desarrollaron una herramienta denominada FACE, que automatiza el

proceso de recuperacion de proyectos similares a uno dado y proporciona una estimación en base a la información de dichos proyectos. Su metodo implica la captura de unas treinta caracteristicas de los proyectos por medio de un cuestionario y el posterior almacenamiento de esta información en una base de datos. Estos atributos se distinguen por su uso: unos se utilizan para la seleccion de proyectos similares, otros para la adaptación y otros para ambas actividades. Esta categorización es necesaria para que en la fase de selección, el algoritmo de busqueda se concentre en un espacio con un número menor de dimensiones A cada caracteristica se le asigna un peso representativo de su contribución a la función de similitud. La asignación de los pesos, se hizo en una primera version en base a los comentarios de un experto, pero en una segunda version, entrenaron una red neuronal que al final del proceso, proporciono el conjunto de pesos. El proceso de estimación consiste en extraer los casos similares de la base de datos; solo aquellos que son similares utilizando un determinado umbral son seleccionados; si ningun proyecto es suficientemente similar al proyecto objetivo, se concluye que no se puede realizar una estimacion fiable y el proceso termina Dado un conjunto de proyectos similares, se calcula es esfuerzo que cada uno de ellos sugiere para el proyecto objetivo, ajustando dicha solución en base a las diferencias que ambos proyectos tienen con respecto a los atributos de adaptacion Por último, se calcula un resultado integrado, utilizando las distintas estimaciones obtenidas de los proyectos seleccionados

En resumen la ventaja de estos métodos es que parece que han demostrado tener éxito en la estimación usando datos donde no se localizaban relaciones estadisticas significativas y con muestras pequeñas. Pero tambien hay que considerar los inconvenientes, que están directamente ligados a la incertidumbre de las estimaciones Para que puedan usarse los proyectos de una coleccion deben ser representativos del entorno de desarrollo en el que se ha llevado adelante el proyecto nuevo a estimar Así, si se deben realizar estimaciones de una amplia tipología de proyectos, se debera disponer de un gran número de muestras o puntos de datos ya que el espacio de las medidas de entrada y los rangos de los valores pueden ser muy extensos.

Por otra parte, si se desea obtener un intervalo de estimación, como por ejemplo el valor mínimo y el máximo, basado en un rango de los valores de las entradas, estos modelos no ofrecen valores seguros ya que es muy posible que muestren variaciones discontinuas, debido a las distintas analogías que se han seleccionado. También puede darse la situación de que el caso o casos seleccionados tengan los mismos valores de las medidas que el proyecto objetivo, pero no ser representativos de dicho proyecto debido a factores que no se han considerado en la representación de los proyectos [Walk96].

2.6.3.5 Métodos de Estimación en Proyectos Orientados a Objetos

En esta seccion, se describen los escasos métodos de estimacion que se han publicado centrados exclusivamente en los proyectos Orientados a Objetos.

Uno de los primeros intentos para proporcionar un mecanismo de estimacion en orientación a objetos data de 1990 y es el propuesto por Luiz Laranjeira [Lara90]. Se fundamenta en que para realizar una adecuada estimación de costes es necesaria una precisa estimación del tamaño del producto a desarrollar, considerando ademas, que esta estimacion es necesaria, obviamente, en fases iniciales del ciclo de vida. Propone realizar una especificacion del problema usando objetos de entidad del dominio y sus relaciones, para seguidamente descomponerlos en otros objetos de niveles de abstraccion mas bajos. Para ello, se consideran los atributos, métodos, interacciones y los aspectos no funcionales de los objetos (ya que estos influyen en el tamaño de los mismos). Así, se estima dicho tamaño de bajo nivel y se asciende en la jerarquia, calculando el tamaño de los objetos de niveles superiores. El valor de la estimacion del tamaño del producto sera el sumatorio del tamaño de los objetos superiores de la jerarquia mas el tamaño de los objetos *de utilidad* necesarios. El modelo estadistico utilizado parte de las curvas de precision de la estimacion de costes basada en la fase del ciclo de vida de un producto presentada por Boehm [Boeh81], que según Laranjeira se pueden extender a aspectos relacionados con la estimación del tamaño. Así, las expresiones adaptadas de las curvas exponenciales son

$x(n) = A \cdot (1 - e^{-B \cdot n})$ para la curva exponencial superior

$x(n) = \frac{A}{1 + e^{-B \cdot n}}$ para la curva exponencial inferior

donde *x(n)* es la estimacion del tamaño del producto, siguiendo los pasos anteriores, *A* es tamaño real del producto y *n* hace referencia al nivel de descomposición de los objetos en la especificación. El valor de *B* es un coeficiente que determina la velocidad de convergencia de la estimación hacia el valor real, según se pasa de un nivel de descomposición a otro. La hipótesis es que el modelo de especificación de objetos captura suficiente conocimiento del sistema como para hacer que las estimaciones converjan hacia el tamaño real. El valor del coeficiente *B* se deberá obtener analizando un número representativo de proyectos terminados cuyos

tamaños se calcularon usando el método propuesto. Sugiere para ello usar una tecnica estadística de ajuste de curvas.

Operando con las expresiones de las curvas y calculando los intervalos de confianza para las estimaciones en base a las desviaciones, finalmente obtiene un valor de estimación del tamaño y un intervalo: el valor maximo y el mínimo, habiendo una alta probabilidad de que el tamaño real se encuentre dentro del intervalo. La confianza de este intervalo depende del valor de *B* y de *n*.

El autor valida su método con datos de dos proyectos usando como medida de tamaño las lineas de código y sin indicar claramente cómo llega a estimarlas a partir de la información que indica se debe recoger en la especificacion orientada a objetos Además, nótese que este método estima el tamaño de un producto, con lo que el autor sugiere que se use COCOMO como método de estimacion de recursos

Curiosamente, varios años despues Henderson-Sellers publico una correccion al metodo de Laranjeira [Hend96a, Hend97] Expone serios errores matematicos relacionados con la Estadistica, las funciones exponenciales y la naturaleza de los datos discretos y continuos Así recalca que las ideas basicas son interesantes pero ademas de los errores tipograficos y una exposicion matematica descuidada también hay que revisar las expresiones de las curvas exponenciales que no son correctas porque nunca van a llegar a converger También señala la indefinicion en el caso de la indicacion de que el tamaño de producto se encuentre dentro del intervalo con una *alta* probabilidad Segun, Henderson-Sellers todo intervalo de confianza estadístico debe llevar asociado un nivel de probabilidad.

Por otra parte, Jenson y Bartley sugieren un metodo parametrico para la estimacion del esfuerzo en proyectos orientados a objetos [Jens91] Su metodo utiliza como base la información que se puede obtener de una descripcion abstracta orientada a objetos del software, informacion que se puede obtener en fases preliminares del proyecto. Así, extraen de una descripcion escrita en ingles el número de objetos, de operaciones y de interfaces entre objetos y su hipotesis es que el esfuerzo es una funcion de estos tres parámetros

$$\text{Esfuerzo} = f(\text{Objetos, Operaciones, Interfaces})$$

Para validarla, utilizaron 16 proyectos de software pequeños, de un entorno de desarrollo estable. Probaron distintos modelos de regresion y llegaron a la conclusión de que la única variable dependiente con capacidad predictiva era el número de objetos y que además tenía una relación no lineal con el esfuerzo También encontraron multicolinealidad entre el número de objetos y el de interfaces dado lo pequeños que eran los proyectos de muestra, pero indicaron la

posibilidad de que esto no se diera en productos de software más complejos, es decir, que las tres variables fueran significativas. El modelo estadístico que mejores resultados obtuvo fue el modelo exponencial con el número de objetos como única variable dependiente.

Haynes, Menzies y Phipps [Hayn95] describen su experiencia basada en observaciones empíricas que indican que el uso del número de clases puede ser una buena base para la estimación de esfuerzo. En este artículo, describen un conjunto de observaciones:

1. El esfuerzo tiene una alta correlación con las líneas de código.

2. El paso de mensajes, como medida de tamaño, tiene una alta correlación con las líneas de código.

3. Hay menos variación en la correlación entre esfuerzo y tamaño si se usa el paso de mensajes como medida de tamaño en vez de las líneas de código

4. Los métodos poseen distribuciones de frecuencia sobre el tamaño que indican que son pequeños (una media de 3,1 líneas de código/método) e independiente del lenguaje de programación y del dominio

5 El tamaño de las clases sigue una distribución normal (una media de entre 44 y 69 líneas de código/clase) que es independiente del lenguaje, por lo que se pueden utilizar como base de la estimación del tamaño

6 Se han medido diferencias de productividad de hasta 7 veces, por lo que se debe controlar la productividad del equipo de desarrollo: según esta experiencia, la mayor causa de variación en el valor de la estimación es la capacidad individual de los desarrolladores.

7 Se han obtenido las siguientes cifras de productividad:

   - 2 clases / persona-mes
   - 2,5 clases / persona-mes → Desarrollo en C + + grande
   - 4 clases / persona-mes → Desarrollo en Smalltalk grande
   - 11 clases / persona-mes → Desarrollo en Smalltalk pequeño

En un trabajo posterior, Haynes y Henderson-Sellers proponen un proceso de estimación de costes básico y unas cifras de equivalencias útiles, todo ello basado en sus observaciones empíricas [Hayn96].

También hacen especial hincapié en el hecho de que existe una precondicion para la realización de estimaciones fiables: el estado de madurez de los procesos de la organizacion de desarrollo; en su opinión, para que el proceso de estimacion tenga exito, la organización necesita estar, como mínimo, en un nivel 2 del CMM. El proceso propuesto es el siguiente:

1 Determinar el tamaño del producto a desarrollar

2 Determinar el índice de productividad de los desarrolladores

3 Dividir el tamaño del producto entre el índice de productividad para determinar la cantidad de esfuerzo que se requiere

4 Multiplicar el esfuerzo por el coste de la hora de trabajo obteniedose el coste total del desarrollo

El objetivo clave de la primera fase es crear una estimacion de número total de clases que se van a requerir. Para ello, un experto debe realizar un estudio de los requisitos y llegar a representar las clases del dominio, tambien llamadas *objetos de negocio*, que cubren dichos requisitos.

La experiencia de estos autores indica que si se analiza un sistema en desarrollo, una parte importante del mismo no esta relacionada con la aplicacion concreta a ser desarrollada, sino que esta vinculada a la tecnologia (bases de datos, telecomunicaciones, GUI, ...); es decir, los componentes tecnologicos pueden consumir entre el 60 y 70 % del esfuerzo de desarrollo. Ademas, dado que la infraestructura tecnologica de una aplicacion es relativamente constante, la mayor variabilidad cuando se diseña un sistema se va a encontrar en el modelo del dominio de la aplicacion. Cuando se conoce el numero de clases del dominio en un sistema, es posible estimar el número correspondiente de clases tecnológicas asociadas y por tanto, el tamaño total. Así, las observaciones de los autores llevan a la conclusión de que dados los requisitos y el modelo de objetos del dominio, cada clase identificada tendrá las siguientes clases asociadas:

- 0,5 clases tecnológicas
- 1,0 clase de base de datos
- 1,0 clase de interfaz de usuario

Sumando a esta cantidad la propia clase del dominio, se dispone de un factor de 3,5 que se aplica al número total de clases del dominio para obtener una estimación del tamaño total. Hay que ser cautos en su uso, debido a una restriccion

parece ajustarse bien a aplicaciones de gestión cliente-servidor tradicionales de tres niveles, que tienen una base de datos y una interfaz de usuario grafica. (Así, se probo el modelo con tres proyectos que utilizaban un marco de almacenamiento de objetos en una base de datos relacional; se obtuvo un ratio de 2,6 al no haber una capa de base de datos. Este valor se observó de forma consistente en los tres proyectos).

A continuacion, para estimar el coste y la duracion del proyecto, se necesita un indice de productividad. Los valores tipicos en la industria para los proyectos orientados a objetos parecen rondar el intervalo entre 2 y 4 clases por persona-mes, independientemente del lenguaje de programacion. Así que dividiendo la estimacion del número de clases entre 2,5, se obtiene una estimación del esfuerzo que se requiere. Por supuesto, hay otros factores que deben tenerse en cuenta la capacidad de los desarrolladores, el tipo del producto, la fiabilidad,etc

Los autores han llegado a estas conclusiones empiricamente por lo que su propuesta ha sido validada Proporcionan cifras de 12 proyectos, en los que las estimaciones tuvieron una precision de entre el 5 y 15 %. La validacion estadistica está descrita en un articulo posterior (Hayn97)

Para finalizar el capitulo queremos hacer enfasis en la dificultad que entraña tomar medidas precisas, fiables, validas de un producto con el objetivo de utilizarlas en un modelo de estimacion de esfuerzo y mas en una disciplina tan cambiante como la Ingeniería del Software y en un area tan concreta como la Tecnología Orientada a Objetos, cuyo uso en la industria está en su infancia Nos enfrentamos a algo intangible como el software y ademas, a algo mucho mas *peligroso* y variable las personas que lo desarrollan Por ultimo, unas palabras de Albert Einstein

> «Not everything that counts can be counted and not everything that can be counted counts»

3. MEDIDAS DE TAMAÑO Y COMPLEJIDAD PARA LA FASE DE ESPECIFICACION

«You can t control what you can t measure If you think the cost of measurement is high consider the cost of being out of control !!! »

Tom DeMarco, 1982

3.1 INTRODUCION

3.1.1 Planteamiento del Problema

Al igual que en cualquier disciplina de ingeniería, el desarrollo del software requiere un mecanismo de medicion que se pueda utilizar para la realimentacion y evaluacion La medicion nos permite crear una memoria corporativa y es una ayuda en la busqueda de respuestas asociadas con la mejora de los procesos del software Nos permite realizar la planificacion de un proyecto, determinar los puntos debiles y fuertes de los procesos y productos en uso, proporciona una base racional para la adopcion o refinamiento de técnicas de desarrollo y permite evaluar la calidad de los procesos y los productos Ademas, tambien ayuda a valorar el progreso durante el transcurso de un proyecto, asi como a tomar medidas correctivas basadas en dicha valoracion y a evaluar el impacto de las medidas utilizadas

Sin embargo, los enfoques pasados que se han usado para el diseño de nuevas medidas del software rara vez han considerado explícitamente los objetivos especificos que se pretendían alcanzar; es más, generalmente no se han basado en los supuestos e información a tener en cuenta sobre las características del entorno de desarrollo objeto de estudio Esto incluye descripciones de la estructura

organizativa y los procedimientos de trabajo, guías, estándares, etc. A menudo esto ha llevado a un cierto grado de ambigüedad en las definiciones de las medidas, en sus propiedades y en los fundamentos sobre los que se sustentan, haciendo que su uso sea complicado, su interpretación peligrosa y los resultados de las validaciones contradictorios.

Como consecuencia de lo anterior, el número de medidas disponible en la literatura es realmente amplio, pero el número de medidas utilizadas y útiles en la industria es pequeño

Por lo tanto, y de acuerdo con distintos estudios realizados en la industria sobre la aplicacion de métricas y modelos, para que la medición sea efectiva debe ser:

1. Dirigida al cumplimiento de objetivos específicos.

2 Aplicada a todos los productos, procesos y recursos del ciclo de vida

3 Interpretada en función de la caracterización y estudio del contexto, entorno y objetivos de la organización.

Esto implica que la medicion debe definirse siguiendo un enfoque *descendente* debe basarse en objetivos y en modelos. Un enfoque ascendente no funcionaria ya que existen demasiadas características observables en el software (por ejemplo, el tiempo, número de defectos, complejidad, líneas de codigo, severidad de los errores, esfuerzo, productividad, densidad de los fallos,...), y no es un aspecto obvio determinar que medidas utilizar y como interpretarlas sin el uso de unos modelos y objetivos apropiados que nos definan el contexto Es decir, la definicion de medidas exige el uso de un enfoque riguroso y disciplinado, ya que las medidas utiles dependen en gran escala de los objetivos y supuestos que se establezcan en el proceso de software en uso. Este enfoque parece particularmente importante para las medidas de producto, ya que son más complejas que las de proceso y hacen referencia a fenomenos que a penas comprendemos

Desafortunadamente y a diferencia de en otros campos científicos, parece difícil elaborar un conjunto de medidas *universal* en la Ingeniería del Software y que se pueda usar en distintos entornos de aplicacion.

Por todo, a continuacion describimos brevemente el marco de referencia *Goal Question Metric* (GQM) que se centra en el supuesto de que para que una organizacion pueda tomar medidas que sean significativas, en primer lugar debe especificar sus propios objetivos, a continuación debe rastrear esos objetivos hasta los datos que se

supone que definen dichos objetivos de una forma operativa y finalmente, debe proporcionar un marco para la interpretación de los datos con respecto a los objetivos formulados [Basi88, Basi94].

El marco se centra en tres niveles:

1. Nivel Conceptual (OBJETIVO): Un objetivo se define para un objeto, debido a una variedad de razones, con respecto a unos modelos de calidad, desde distintos puntos de vista y en relación a un entorno particular. Los objetos de la medición son:

   - Productos: entregables y documentos que se producen durante el ciclo de vida de un sistema.

   - Procesos: actividades relacionadas con el software y asociadas generalmente al tiempo.

   - Recursos: elementos que los procesos utilizan para producir sus salidas.

2. Nivel Operacional (PREGUNTA): Se elaboran un conjunto de preguntas para caracterizar el modo en que se va a realizar la valoración o el grado de cumplimiento de un objetivo específico. Las preguntas tratan de caracterizar al objeto de la medición con respecto a un aspecto de calidad concreto y tratan de determinar la calidad de dichos objetos desde el punto de vista seleccionado.

3. Nivel Cuantitativo (MEDIDAS): Se asocia un conjunto de datos a cada pregunta, con el fin de proporcionar una respuesta de manera cuantitativa. Los datos pueden ser:

   - Objetivos: si dependen únicamente del objeto que se esta midiendo y no del punto de vista desde el que se captan (por ejemplo, el número de versiones de un documento).

   - Subjetivos: si dependen tanto del objeto que se está midiendo como del punto de vista desde el que se captan (por ejemplo, el nivel de satisfaccion del usuario).

Como el proceso de definición de los objetivos es crítico para que la utilización del marco GQM tenga éxito, este proceso se soporta en unos pasos metodológicos. Así, un objetivo posee cuatro factores: el propósito, el aspecto de calidad, el objeto y el punto de vista.

En cuanto a las preguntas, estas se pueden agrupar en tres categorías:

1. ¿Cómo se puede caracterizar el objeto con respecto al objetivo global del modelo GQM específico?

2. ¿Como se pueden caracterizar los atributos del objeto que son relevantes con respecto al aspecto de calidad definido en el modelo GQM específico?

3. ¿Como se deben evaluar las características del objeto que son relevantes con respecto al aspecto de calidad definido en el modelo GQM específico?

Una vez se han elaborado las preguntas, se procede a asociarles unas medidas adecuadas. Los factores a considerar para realizar esta actividad son muy variados; entre ellos:

- La cantidad y calidad de los datos existentes: se tratará de maximizar el uso de las fuentes de datos existentes, siempre que estén disponibles y sean fiables

- La madurez de los objetos de medición: se aplicarán medidas objetivas a los objetos de medicion más maduros, y se utilizarán evaluaciones subjetivas cuando se trate con objetos informales o poco estables.

- El proceso de aprendizaje: los modelos GQM siempre necesitan refinamientos, por tanto, las medidas que se definan deben ayudar a evaluar no sólo al objeto de medición, sino también la fiabilidad del modelo utilizado para evaluarlo

Como se ha descrito previamente, el marco GQM proporciona un proceso muy detallado para la definición de los objetivos, ya que estos condicionan a los elementos del resto de los pasos. Sin embargo, este marco únicamente ofrece una descripción de los métodos y modelos para seleccionar una medida [Walk96], es decir, usa modelos descriptivos para alcanzar los objetivos, pero no especifica como generarlos.

Por este motivo, vamos a utilizar como marco para la definición de nuestras medidas una expansión del modelo GQM propuesta por Briand, Morasca y Basili [Bria94]. Estos autores describen un enfoque para la definición de medidas, orientada a los objetivos y basada en GQM, pero especialmente centrado en las medidas de producto y en la construccion de sistemas predictivos, que son una aplicacion crucial de la medición. Además, para cubrir el aspecto de la selección de las medidas, incorporan en su modelo la definición de las propiedades matemáticas que las medidas deben cumplir.

En la Figura 3.1, se ilustran los pasos de los que se compone esta estrategia de definición de medidas.

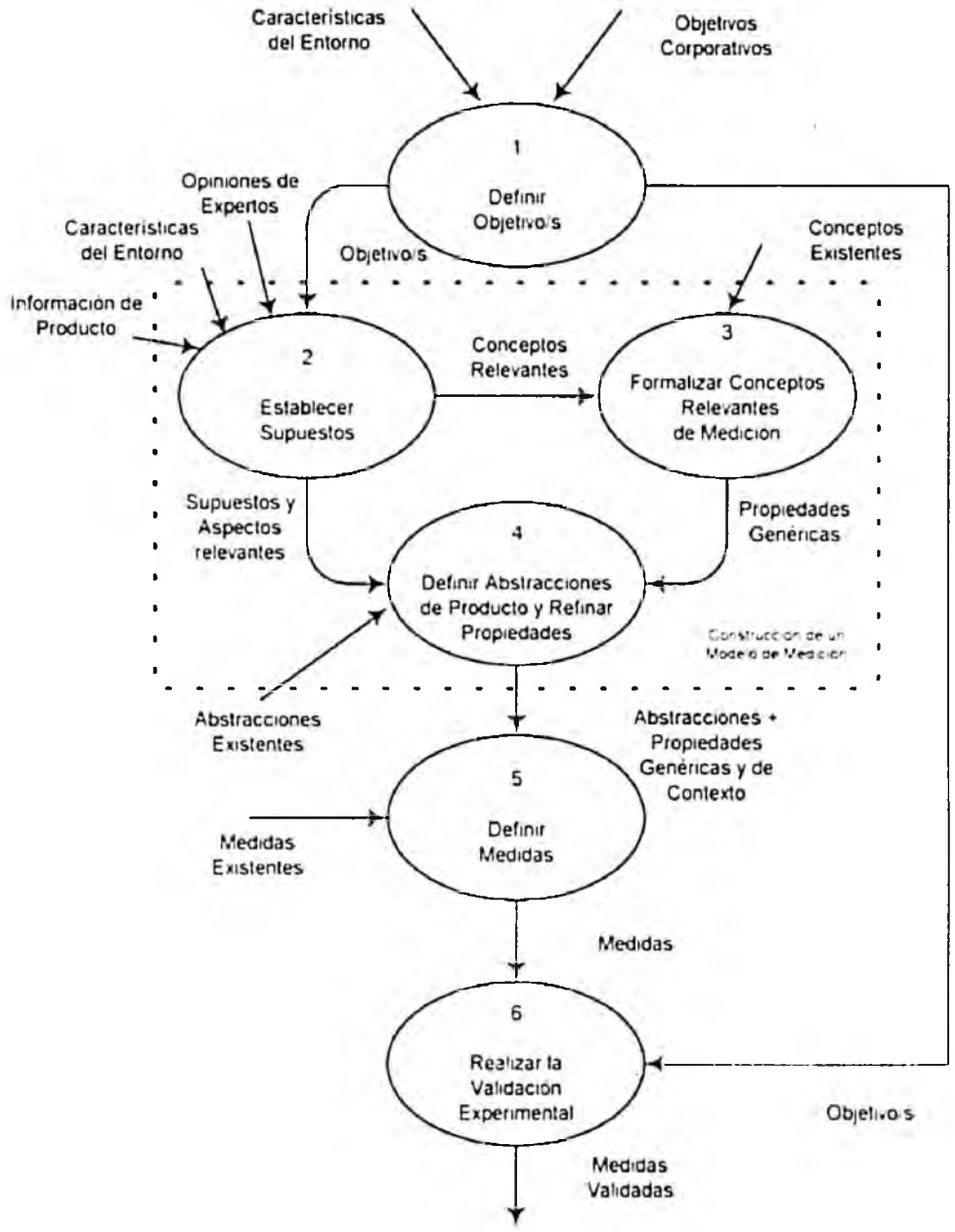

Figura 3.1 Enfoque de definición de Medidas basado en GQM y en Propiedades.

El modelo representado en la Figura 3.1 no utiliza la segunda parte de GQM, la relacionada con la formulación de las preguntas que se derivan de los objetivos, como línea principal del proceso. Dichas preguntas han sido sustituidas por los resultados que se obtiene de los pasos 2, 3 y 4.

Por tanto, en este marco no se utilizan las medidas como respuesta a las preguntas; se utilizan para validar los supuestos que se realizan en el paso 2. Sin embargo, puede haber aspectos relevantes de las características del entorno que no se pueden modelizar explícitamente; en estos casos, se deberán formular las preguntas para dar soporte a una interpretacion completa de las medidas.

A continuación vamos a examinar con detenimiento cada fase del proceso, aplicando el marco a nuestra situación.

Paso 1: Definir los Objetivos

En esta sección, se aplica el primer paso de GQM. Debemos definir los objetivos considerando el objeto de estudio, el propósito, el aspecto de calidad de interes y el punto de vista.

El *objeto de estudio* determina los productos de software que deberán ser modelizados mediante abstracciones matemáticas para poder ser analizables y los supuestos que pueden ser relevantes.

El *proposito* indica cuál es la intención de uso de las medidas a definir y por tanto, condiciona el tipo y la cantidad de información a recopilar.

El *aspecto de calidad* ayuda a determinar cuál es la variable dependiente contra la que se van validar las medidas de producto que se definan y los supuestos que unen al objeto de estudio con dicho aspecto de interes.

El *punto de vista* indica el momento en el tiempo en el que deberían realizarse las caracterizaciones, estimaciones y evaluaciones, y por tanto, ayuda a determinar cuál será la información del producto que estará disponible para definir las abstracciones del mismo y las medidas. También ayuda a determinar la información que puede resultar difícil adquirir y ayuda a definir un modelo descriptivo del aspecto de calidad de interés.

Aplicando el marco a nuestras necesidades, obtenemos la siguiente información:

- Proposito: realizar de forma precisa una estimación inicial del esfuerzo de desarrollo de un proyecto de gestion (nivel de interfaz, dominio y gestion de datos) siguiendo el Paradigma Orientado a Objetos en una fase temprana del ciclo de vida.

- Objeto de estudio: vamos a centrarnos en el producto que se obtiene en la fase de Análisis del Dominio siguiendo un método y técnicas Orientadas a Objetos, según un proceso de desarrollo genérico (sección 3.1.2). El producto, por tanto, será parte del documento de especificación de requisitos, en concreto, el Modelo del Dominio.

- Aspecto de Calidad: la variable dependiente contra la que validaremos las medidas que se definan a raíz de este proceso será el esfuerzo de desarrollo, medido en personas-semana. A partir de esta medida de proceso, podremos estimar el coste de desarrollo y hacer una planificación temporal y de recursos.

- Punto de Vista: el punto de vista utilizado es el del Gestor del Proyecto y considerando el tiempo en que se puede realizar la estimación, el punto de vista es el de la fase de Análisis del Dominio, que condiciona la información de la que dispondremos[1].

En resumen, nuestro objetivo es *realizar de forma precisa una estimación inicial del esfuerzo en personas-semana, analizando el Modelo del Dominio que se obtiene como resultado de la fase de Análisis del Dominio usando el Paradigma Orientado a Objetos como estrategia de desarrollo.*

Paso 2: Establecer los Supuestos

En base al objeto de estudio y al aspecto de calidad, tal y como los hemos definido en el paso anterior, debemos establecer un conjunto de supuestos relevantes que engloben nuestro conocimiento intuitivo sobre el entorno de desarrollo y el objeto de estudio. Estos supuestos definen implícitamente un orden en el conjunto de objetos de estudio con respecto al aspecto de calidad de interés. Por ejemplo, podemos ordenar componentes con respecto al esfuerzo necesario para desarrollarlos. En este paso, se necesita disponer de información relacionada con el entorno de desarrollo (por ejemplo, un modelo de procesos descriptivo), información sobre el producto y opiniones de expertos que proporcionan la base intuitiva de los supuestos. Los resultados de este paso engloban no sólo los supuestos, sino también un conjunto de conceptos de medición relevantes y una definición más precisa del objeto de estudio.

---

[1] Nótese que al restringir la fase del ciclo de vida en la que vamos a realizar la medición al Análisis del Dominio, no vamos a poder considerar un aspecto sumamente importante en la Orientación a Objetos y en la Estimación de Recursos como es la Reusabilidad. En este punto en el tiempo no disponemos de información como para valorar el índice de reusabilidad en un proyecto.

En nuestro caso concreto, nos podemos centrar en el supuesto clásico que todos los autores dedicados a la estimación de esfuerzo han establecido: *el esfuerzo de desarrollo está relacionado con el tamaño del producto*

Este supuesto se ha mantenido independientemente del entorno de desarrollo, del modelo de estimación utilizado o de la manera de medir el tamaño Nuestro objeto es el Modelo del Dominio en un proyecto orientado a objetos, por lo que el supuesto será:

SUPUESTO 1 El esfuerzo de desarrollo está relacionado con el tamaño del modelo del dominio, de manera que cuanto mayor es dicho dominio, mayor es el esfuerzo necesario para desarrollarlo.

Esto nos lleva a considerar el supuesto 1 con un matiz ligeramente distinto dos dominios con tamaños iguales, ¿consumirán el mismo esfuerzo? La respuesta es obviamente que no Intuitivamente percibimos que el tamaño es un atributo insuficiente para poder estimar el esfuerzo. Dos productos de tamaños iguales pueden llevar a esfuerzos completamente distintos debido a la *complejidad* inherente a cada uno de ellos

Así, podemos tener un producto considerado como de tamaño pequeño pero con un determinado nivel de complejidad que consume los mismos recursos que un producto de tamaño grande, con un nivel de complejidad reducido.

SUPUESTO 2 El esfuerzo de desarrollo está relacionado con el tamaño y la complejidad del modelo del dominio, de manera que cuanto mayor es dicho dominio o mayor es la complejidad, mayor es el esfuerzo necesario para desarrollarlo

Además, no podemos perder de vista que en la estimacion del esfuerzo participan otros atributos a los que podemos denominar *conductores de coste* y que captan las relaciones intuitivas de los expertos con respecto a la estimacion Estos atributos generalmente están vinculados a procesos de comparacion entre las caracteristicas del contexto de proyectos pasados y las caracteristicas del proyecto actual.

SUPUESTO 3 El esfuerzo de desarrollo de un producto está condicionado por un conjunto de caracteristicas del contexto del proyecto entre las que podemos citar los factores del producto, de plataforma, del personal y del proyecto [Boeh81, USC97]

Paso 3: Formalizar los Conceptos de Medición Relevantes

En este paso, se definen formalmente los conceptos de medicion relevantes usando determinadas propiedades matematicas. Así, podemos definir un *concepto de medicion* como una clase de medidas caracterizadas por un conjunto de propiedades matematicas y asociado a un atributo de producto intuitivo (por ejemplo, el tamaño) De esta manera, la eleccion de las medidas esta condicionada al cumplimiento de estas propiedades

Por lo tanto, la búsqueda no es tan exploratoria y disponemos de unos criterios matematicos precisos para valorar la solidez de las medidas que vayamos a proponer Las propiedades matemáticas que caracterizan a los conceptos de medicion son independientes de cualquier instancia que se pueda crear de dichos conceptos en una medida en concreto y por lo tanto, se denominan *propiedades de conceptos genericos*.

En nuestro caso particular, necesitamos un conjunto de propiedades matematicas genericas para los conceptos de medicion *tamaño* y *complejidad*. Las distintas propiedades que se han propuesto en la literatura ya se han comentado en la seccion 2 5 Las propiedades que vamos a considerar son las propuestas por Weyuker [Weyu88], Briand *et al* [Bria96] y Whitmire [Whit96, Whit97]

Paso 4: Definir las Abstracciones de los Productos y Refinar las Propiedades

Necesitamos definir una abstraccion que nos permita captar toda la informacion (por ejemplo, clases, atributos, relaciones, ..) necesaria para definir los conceptos involucrados en los supuestos del punto 2 Una abstraccion es una representación matemática de un producto, generalmente en forma de grafo Es importante establecer correspondencias entre los productos y sus abstracciones, ya de este modo dichas abstracciones son analizables y algunos de sus atributos se vuelven cuantificables. La elección deberia estar por completo condicionada por el objetivo marcado (el objeto de estudio y el aspecto de calidad de interes) y el conjunto de supuestos del paso 2; es decir, las abstracciones deben captar todos los conceptos involucrados en el conjunto de supuestos asociados al objeto de estudio

Se debe comprobar la completitud de la correspondencia establecida entre el producto y su abstraccion. Una forma de valorar lo conveniente que resulta una abstracción consiste en estudiar el efecto que ciertas modificaciones de relevancia tienen en el producto y valorar el impacto sobre la abstracción Por ejemplo, se cambia la topologia del grafo o se añaden y eliminar nodos o arcos.

Además, se debe ampliar el conjunto de propiedades asociado a cada concepto de medición. Estas nuevas propiedades son específicas de un determinado contexto y por tanto, se denominan *propiedades dependientes del contexto*. En la mayor parte de los casos, captarán el efecto que las modificaciones sobre las abstracciones anteriormente descritas tienen sobre la ordenación de las mismas.

En nuestro caso, el objeto de estudio es el Modelo de Análisis del Dominio de un proyecto orientado a objetos, especificado en forma de un diagrama de clases que no deja de ser un grafo cuyos elementos son cuantificables. Sin embargo, para llevar a cabo una formalización más sistemática y rigurosa de las características de un modelo orientado a objetos, vamos a utilizar la Teoría de Categorías.

La abstracción de nuestro producto así obtenida nos permitirá representarlo con fidelidad, cuantificar sus atributos y analizar el impacto de las modificaciones de elementos sobre la abstracción. La formalización matemática de las propiedades de un sistema orientado a objetos, es decir, la abstracción de nuestro producto y sus propiedades dependientes del contexto, se describen en la sección 3.2.

Por otra parte, tenemos necesidad de formalizar la captura de la información del entorno, que afecta a la adaptación de las estimaciones de esfuerzo. Esta parte no está directamente relacionada con el producto ni con su abstracción, sino que depende en gran medida de la intuición de los expertos. Por ello, es necesario contar con su colaboración para captar la información subjetiva sobre conductores de coste en forma de cuestionario.

Paso 5: Definir las Medidas

Debemos definir las medidas en base a las abstracciones del producto, a los conceptos de medición y a sus propiedades, tanto genéricas como de contexto. Se pueden utilizar medidas ya existentes, si cumplen las propiedades establecidas.

En las secciones 3.3 y 3.4, se describen respectivamente una medida de tamaño y una de complejidad, basadas en la formalización del producto descrita en la sección 3.2, y se especifican tanto sus propiedades genéricas como las que dependen del contexto.

Además, con respecto a la definición de los conductores de coste, los que hemos elegidos son los documentados en el modelo COCOMO II, que captan las opiniones de los expertos con respecto a los factores que provocan variaciones en el esfuerzo. Estos conductores se comentan en el Apéndice B.

Paso 6: Realizar la Validación Experimental

Despues de la definición de las medidas, se debe recopilar información sobre los productos y los procesos con vistas a validar experimentalmente dichas medidas y los supuestos que se establecieron en su definición. El procedimiento a seguir para la validación experimental varía significativamente dependiendo del proposito de la medicion. Con respecto a la predicción, que es nuestro objetivo, necesitamos validar las medidas del producto considerando su relación estadística con el aspecto de calidad de interes. Es decir, la validacion experimental puede considerarse como la busqueda de las relaciones estadísticas entre las medidas del objeto de estudio y los modelos descriptivos del aspecto de calidad. Estas relaciones se pueden verificar utilizando distintas técnicas de análisis, ya sea estadístico o de aprendizaje automatico.

En nuestro caso, como el objetivo de la definición de las medidas es obtener estimaciones fiables del esfuerzo, nuestra validacion consistirá en probar que dichas medidas estan correlacionadas con el esfuerzo. Para ello, propondremos un modelo de estimacion de esfuerzo, basado en las medidas y resultados de proyectos pasados, que nos sirva para obtener una primera estimacion de un proyecto que va a comenzar. Al disponer de una coleccion de proyectos pasados, completos, donde se conocen los valores de las medidas de tamaño y complejidad y las valoraciones de los expertos con respecto a los factores de coste, y donde también se conoce el esfuerzo que se consumió para realizarlos, vamos a usar una técnica de valoracion de la *precision* de las estimaciones que consiste en realizar estimaciones de prueba con cada proyecto pasado. Para ello, se elimina la información del proyecto pasado de la coleccion y se utiliza como proyecto en curso. La estimación obtenida debe ser cercana al valor real de esfuerzo que el proyecto tuvo. Calcularemos la magnitud del error relativo y repitiendo el proceso para toda la colección de proyectos pasados, podremos calcular la media de la magnitud del error relativo y la fracción de proyectos que se han podido estimar con un error relativo inferior a un umbral determinado.

3.1.2 Un Proceso de Desarrollo Orientado a Objetos Genérico

Como se ha señalado en la sección anterior, nuestro objetivo es poder realizar una estimación del esfuerzo de desarrollo lo más precisa posible en la fase más temprana del proyecto en la que se posea un mínimo de información sobre el desarrollo. Para ello, necesitamos definir métricas que se puedan calcular analizando los datos que están disponibles en los productos generados en dicha fase.

Ademas, para que la estimacion se pueda realizar mediante la comparacion de atributos con proyectos pasados, es necesario que se hayan realizado siguiendo un proceso de desarrollo, si no idéntico, por lo menos, similar.

Por todo ello, a continuación vamos a describir un proceso genérico para el desarrollo de sistemas orientados a objetos. Al ser genérico, es posible establecer correspondencias entre las actividades de este proceso y otros definidos en la literatura como OOSE [Jaco92]. Además este modelo se ajusta al modelo de referencia propuesto por OMG [OMG92]. El proceso contempla un conjunto de actividades que aunque las describamos de forma serializada, no implica que su uso vaya a ser así. En realidad, esta estrategia de desarrollo puede usarse tanto en un ciclo de vida en cascada como en un ciclo de vida iterativo (incremental, en espiral en fuente, etc).

El conjunto de actividades es el siguiente:

1 Conceptualización

2. Análisis del Dominio

3 Diseño Lógico

4 Diseño Físico

5 Construccion

6. Integración

A estas actividades de desarrollo hay que añadirles dos mas que se extienden a lo largo de todo el proceso y que también consumen recursos: la gestion de la configuracion y la planificacion y seguimiento del proyecto, como se ilustra en la Figura 3 2.

Conceptualización

A esta fase también se le denomina *Analisis Preliminar* o *Captura de Requisitos* y su objetivo es la identificación de los responsabilidades y funciones del sistema a desarrollar Se deben identificar las necesidades, requisitos, preferencias, expectaciones y restricciones desde el punto de vista de los usuarios y llegar a un enfoque general que cumpla con los requisitos, tomando en consideracion la tecnologia disponible, la situación del mercado, los recursos, el tiempo del que se dispone y otros enfoques alternativos existentes [Rumb95]

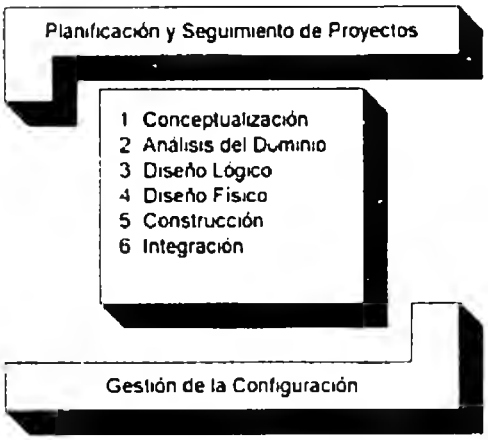

Figura 3.2 Un proceso de desarrollo genérico.

Es importante que la especificación de los requisitos sea clara y no ambigua, aunque se debe contar con el hecho de que puede que el usuario no consiga expresar claramente sus necesidades y expectativas.

En esta fase, la técnica que UML propone para la especificación de requisitos son los *casos de uso* [Booc97, Mull97], que captan gráfica y textualmente la funcionalidad de negocio que la aplicación a desarrollar debe soportar desde el punto de vista del usuario.

Sin embargo, cuando los proyectos que una organización desarrolla son para terceros, es decir, se desarrolla bajo contrato o en el caso de un proyecto para un concurso público, la especificación de requisitos puede venir redactada generalmente en forma de una especificación funcional en lenguaje natural. Esta lista puede resultar densa, confusa, contener requisitos contradictorios y no reflejar correctamente las necesidades de los usuarios. En estos casos, en esta primera fase se deberá realizar la transformación de la especificación funcional a la especificación orientada a objetos, en la forma de los casos de uso [Hayn96].

Análisis del Dominio

En esta fase es en la que realiza un estudio detallado de los requisitos del sistema con la intención de comprenderlos, explorar sus implicaciones y eliminar inconsistencias y omisiones. El modelo que se obtiene en esta fase es la verdadera especificación de los requisitos del sistema. El objetivo es analizar el dominio de la aplicación; en otras palabras, se deben identificar las clases del dominio que pertenecen al entorno de la aplicación y las interacciones entre ellas.

Las subactividades a realizar (en cualquier orden o en paralelo) son las siguientes [Whit97]:

1 Identificar las clases del dominio del problema

Se deben elegir cuáles son las clases de entidad o de alto nivel del dominio del problema. Para ello habrá que analizar los casos de uso, para garantizar que las clases elegidas dan soporte a las funciones de negocio.

2. Identificar las estructuras

Se identificarán estructuras de generalización y de agregación, ya que hay que considerar que la clases no existen de forma independiente; es decir, colaboran a veces de forma dinámica y otras veces participando en relaciones estaticas con otras clases.

3. Identificar los patrones de comunicación

Existe una necesidad de comunicación entre clases para poder cumplir con las funciones de negocio. Mediante estos enlaces de comunicación se establecen relaciones dinámicas entre los objetos de las clases. Estos enlaces son los que definen la complejidad de la aplicación, más que cualquier otro tipo de relacion.

4. Identificar las restricciones temporales y de concurrencia

Algunos dominios poseen restricciones temporales y concurrentes de forma inherente, que a menudo llevan a descubrir nuevos atributos y operaciones. Todo ello debe ser añadido al modelo de análisis. El número de restricciones temporales proporciona una idea de la naturaleza critica de la aplicacion y el número de restricciones sobre la concurrencia afecta a la forma de la aplicación en términos de linealidad u operacion en paralelo. Ambos aspectos tendrán impacto sobre el esfuerzo dedicado a la fase de diseño.

5 Identificar y describir los atributos

Se deben identificar las caracteristicas descriptivas de las clases del dominio. A medida que va avanzando el proceso, se iran descubriendo mas atributos que deberan ir añadiéndose al modelo.

6. Identificar conexiones de instancia

Se deben localizar grupos de clases que comparten una clave o atributo identificador. Este atributo debe ser el identificador primario en una clase y parte del identificador primario o un atributo común en las otras clases. Estos atributos comunes forman las relaciones asociativas en el modelo. Se deben

localizar asociaciones 1:1, 1:N y M:N, considerando además que si la relacion posee atributos propios además de los compartidos, se debe crear una clase para contenerlos. Además, se debe considerar que la mayoría de las relaciones M:N exigen la adición de una clase asociativa al modelo del dominio. Del mismo modo que las relaciones estructurales, las asociaciones contribuyen directamente en la complejidad de la aplicación. Cuanto más densamente relacionadas estén las clases, mayor será la complejidad de la aplicación

7. Identificar los estados de las clases

   No todas las clases tienen un comportamiento dinámico, pero para aquellas que lo tienen habrá que analizar cuales son los estados que pueden tener, basándonos en los valores de los atributos. Además, los estados se consideran periodos entre eventos, siendo estos últimos los mensajes a los que una clase debe dar respuesta.

8. Identificar los servicios

   Para cada función de negocio o responsabilidad del sistema documentada en los casos de uso, cada clase que participe en su cumplimiento debe proporcionar un conjunto de servicios orientados a alcanzar dicha responsabilidad. Estos servicios son el punto de partida del conjunto de métodos que las clases proporcionarán en fases posteriores.

9. Identificar las conexiones de mensajes

   Este tipo de conexión se produce entre dos clases cuando un servicio de una de ellas invoca o hace uso de un servicio de otra clase. Este tipo de conexión se identifica cuando se localizan los servicios y junto con los demás tipos de relaciones, contribuyen a la complejidad de la aplicación.

10. Revisar el modelo obtenido

    En este punto se debe valorar el modelo obtenido como un todo. Para reducir la complejidad del modelo, se pueden combinar clases y mensajes. Para reducir la complejidad de una clase, se debe valorar la adición de estructuras de agregación y de generalización.

Las técnicas utilizadas en esta fase serán los diagramas de clases, los diagramas de estados y los diagramas de colaboración o de secuencia. Nótese que estos diagramas se pueden ir completando y refinando iterativamente durante el periodo de tiempo que dure la fase de análisis.

Asi, en las primeras iteraciones y consumiendo poco esfuerzo, es posible disponer de suficiente información como para poder realizar estimaciones medianamente fiables Desafortunadamente, parece que no es posible crear estimaciones significativas previas, usando directamente la informacion de los requisitos [Haye96].

Diseño Lógico

Esta fase comienza considerando aspectos relacionados con la arquitectura y termina con aspectos dirigidos a la implementacion.

Se debera determinar el tipo de arquitectura que es apropiada para la aplicacion. Esto implica seleccionar el grado de distribución y la independencia de los distintos componentes arquitectonicos, que básicamente son el dominio de la aplicacion, la interfaz de usuario, la gestion de datos, la gestion de procesos y tareas la interfaz con la red, las comunicaciones interprocesos y las funciones de utilidad Se deberan asignar las responsabilidades del sistema a los distintos componentes

Se deberan refinar las clases del dominio. Esto puede llevar a dividir una clase en varias o a integrar varias clases en una, teniendo presente en todo momento el movimiento y adicion de atributos y operaciones y refinando los parametros de entrada y salida de las operaciones. También se incluiran las clases de soporte o de utilidad (para acceso a bases de datos, servicios de comunicaciones, gestion de excepciones, gestión de objetos, etc.). Algunas se deberan desarrollar, otras se podran comprar y otras se podran reutilizar de proyectos pasados. Esta fase es en la que el número de clases se incrementa, por lo que no hay que perder de vista que se deben considerar los atributos y servicios de estas nuevas clases

En esta fase también se realiza un diseño detallado de los pasos de procesamiento de las operaciones, es decir se diseñan los algoritmos y se documentan de forma exhaustiva dichas operaciones.

Diseño Fisico

Se deben empaquetar las clases en modulos fisicos con el objetivo de realizar una construccion eficiente, que minimice el impacto de los cambios que los componente del diseño pueden sufrir, no solo desde el punto de vista de tener que modificar tambien sus respectivos diseños, sino que tambien desde el punto de vista de tener que recompilarlos

Para llegar a un empaquetamiento eficaz, se deberán analizar las dependencias entre clases y el objetivo será minimizar las interdependencias entre módulos.

Construcción

En esta fase se lleva a cabo la codificacion del producto y las pruebas de unidad.

Integración

Esta última fase, podría denominarse *Entrega* si se utiliza un ciclo de vida en cascada. Las actividades a realizar son las pruebas de integración y aceptación, la instalación y completar la documentación administrativa, de instalación y de usuario.

Sin embargo, usando un ciclo de vida iterativo, las actividades serían las mismas, sólo que sobre una fracción del sistema, la de la iteración en curso.

3.1.2.1 Ciclos de vida Iterativos

Las actividades del modelo de proceso genérico anterior se pueden organizar en distintos ciclos de vida.

Desde el punto de vista de nuestro interés, la estimación del esfuerzo en una etapa inicial del proyecto, esta actividad debe realizarse en la primera etapa del ciclo de vida, disponiendo de una informacion muy preliminar y dedicando poco esfuerzo.

Por lo tanto, se pueden aplicar distintos ciclos de vida, incluido el modelo clásico en cascada. Sin embargo, por tratarse de proyectos orientados a objetos, resulta más interesante coordinar el proyecto a través de un ciclo de vida iterativo. La variante que mejor se ajusta a un proyecto concreto dependera del tamano del proyecto, de la complejidad del dominio y de la elección de la arquitectura [Mull97].

La primera variante se adapta a aplicaciones de tamaño modesto o a aquellas que están bien definidas y que se van a implementar en un entorno arquitectónico probado y que no va a generar sorpresas. Los ciclos son cortos y la integración es continua. Las iteraciones se centran en la construcción (Figura 3.3).

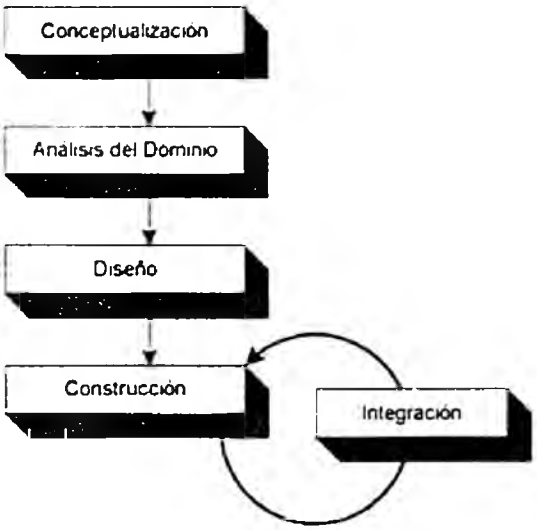

Figura 3.3 Ciclo de vida iterativo centrado en la construcción.

En el caso en que los requisitos estén incompletos al comienzo, o si se van a tener que adaptar cambios arquitectónicos, se pueden añadir bucles de modificación o revisión hacia las etapas iniciales. En esta variante, la iteración principal sigue concentrada en la construcción incremental, pero se consideran flujos secundarios que se centran en refinar el análisis o la arquitectura (Figura 3.4).

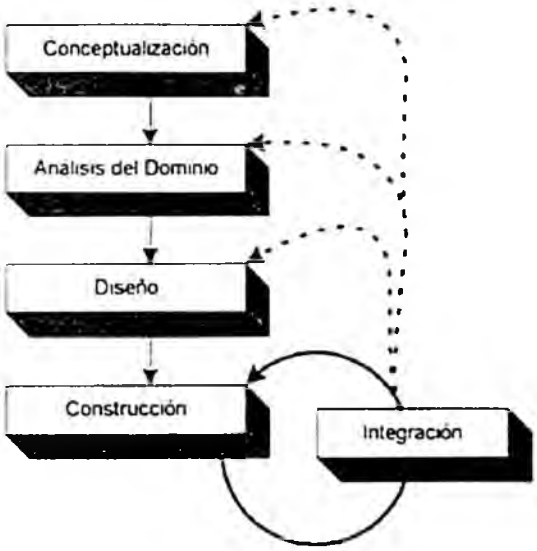

Figura 3.4 Ciclo de vida iterativo con realimentación a fases anteriores.

Por último, se deben considerar aquellos casos en los que los bucles secundarios comienzan a comportarse como primarios. Esta situación es típica de los proyectos en los que hay una gran incertidumbre con respecto a los requisitos, la arquitectura se ha definido parcialmente o donde el proyecto va a ser desarrollado por múltiples equipos en paralelo. Para estos casos, se puede aplicar una tercera variante que se deriva del ciclo de vida en espiral de Boehm [Boeh88]. Las iteraciones del ciclo de vida comprenden las actividades de análisis y diseño como se ilustra en la Figura 3.5.

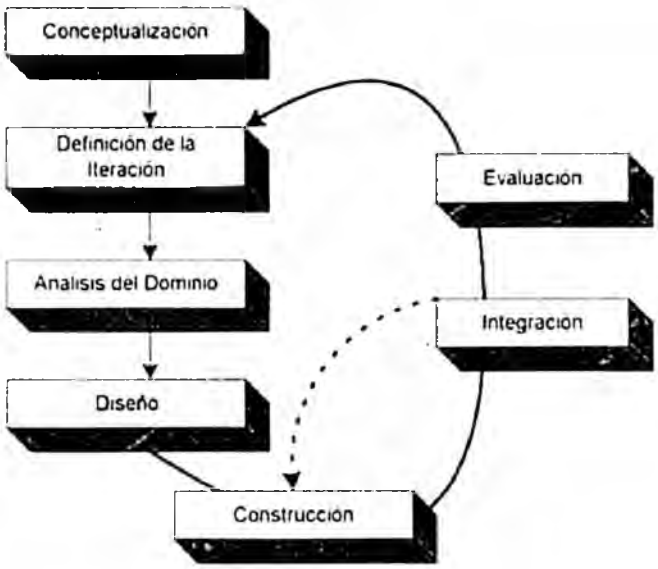

Figura 3.5 Ciclo de vida iterativo basado en el Modelo en Espiral.

Independientemente de la variante de ciclo de vida iterativo que se utilice, es importante avanzar en el análisis tanto como se pueda pero sin caer en la parálisis provocada por la búsqueda del *análisis perfecto*. Un ciclo de vida iterativo, al exigir la entrega periódica de programas ejecutables, favorece la producción de resultados concretos.

3.1.3 El Proceso de Estimación

En esta sección describimos el proceso de estimación que proponemos como parte de esta investigación. En los últimos quince años, distintos autores han propuesto diferentes procesos; la mayoría de ellos describen cómo aplicar un único método de predicción, basado en uno o varios modelos paramétricos [Bail81, DeMa82, Heem92].

Nuestro proceso es una adaptación de los procesos propuestos por Heemstra [Heem92] y Humphrey en su método de estimación PROBE [Hump95, Hump96]. Implica la realización de los siguientes pasos:

- Crear la Base de Datos de proyectos terminados.

  El paso inicial consiste en realizar una recopilacion de los datos de los proyectos terminados. La información a recoger sera la necesaria para poder comparar los proyectos nuevos con los pasados y aplicar el método de estimación propuesto.

- Calibrar el modelo de estimación.

  El entorno en el que el modelo se ha desarrollado y los proyectos almacenados en los que se basa, pueden ser distintos del entorno en el que se va a utilizar. En estos casos, este paso se hace imprescindible.

- Llevar a cabo la fase de Conceptualización y elaborar el Modelo del Dominio.

  En este paso, se realizan las fases iniciales del proceso genérico de desarrollo para el proyecto cuya estimación inicial de esfuerzo queremos realizar. Estas fases se describen en la sección 3.1.2.

- Calcular los valores del tamaño y complejidad del modelo.

  Se calcularán los valores de ambas medidas, para el modelo del dominio dado, siguiendo los procedimientos descritos en las secciones 3.3 y 3.4.

- Captar las características del contexto del proyecto.

  El personal de desarrollo deberá proporcionar la informacion sobre el contexto del proyecto a estimar. Esta información se captará rellenando un cuestionario.

- Estimar el esfuerzo de desarrollo

  La estimación del esfuerzo para un proyecto determinado se realizará utilizando la información de los proyectos pasados, siguiendo el método ESTIMAN, descrito en el capítulo 4.

El alcance de nuestro trabajo termina con los puntos anteriormente comentados, puesto que nuestro objetivo es poder proporcionar una estimación de esfuerzo preliminar y global, con vistas a calcular el coste de un proyecto. Sin embargo, desde el punto de vista del diseño del proceso de estimacion, dicho proceso quedaría incompleto. Por este motivo, a continuación comentamos los pasos que añadimos para completarlo:

- Elaborar la planificación temporal.

  Se deberá distribuir el esfuerzo del paso anterior a lo largo de las distintas fases del ciclo de vida que se va a seguir (sección 3.1.2.1), conociendo la duración del proyecto. Para ello, se necesita saber con qué recursos humanos podemos contar y conocer la productividad típica de dicho equipo.

- Desarrollar el producto

- Analizar el proceso de desarrollo.

  Este paso es realmente interesante, puesto que tienen dos usos distintos: por una parte es un proceso continuo de seguimiento a lo largo de todo el desarrollo, de manera que se puedan hacer ajustes si hay desviaciones. Pero por otra parte, es un proceso *post-mortem*, en el que se extraen las características del proyecto terminado y se realimenta la base de datos, añadiendo un proyecto más cuya información se utilizará para futuras estimaciones.

  En la Figura 3.6 se ilustra el proceso de estimación descrito.

3.2 FORMALIZACION DE LAS PROPIEDADES DE UN SISTEMA ORIENTADO A OBJETOS

Los campos de la Informática y de los Sistemas de Información están repletos de conceptos fundamentales que apenas se definen. Por ejemplo, los términos sistema, subsistema, módulo, objeto, interfaz, acoplamiento, cohesión, jerarquía, entrada, salida, entorno, descomposición, ... son el punto central de un gran número de investigaciones y teorías, y sin embargo, no se definen con precisión y rigor. En ausencia de una base correctamente formulada, es poco probable que se progrese en estos campos [Wand90]. Como dice Parnas [Parn90], el uso de términos confusos evita realizar análisis sistemáticos, que únicamente se consiguen con definiciones precisas.

Por este motivo, es necesario definir un modelo formal que capture las propiedades de un sistema de información orientado a objetos.

Además, en el área de conocimiento en que nos estamos moviendo, debemos considerar que la medición caracteriza un atributo o propiedad de una entidad en términos de un número o un símbolo matemático. Una representación de medida hace corresponder una relación empírica entre dos entidades con una relación

numérica entre dos números o símbolos. Para que la medición sea posible, desde el punto de vista del proceso, se debe disponer de tres requisitos fundamentales: un modelo de la relación empírica en forma de una estructura relacional empírica, un modelo de la relación numérica en forma de una estructura relacional numérica y una correspondencia que preserve la estructura entre ambos. Esta correspondencia debe reflejar fielmente las propiedades del sistema empírico en el sistema numérico

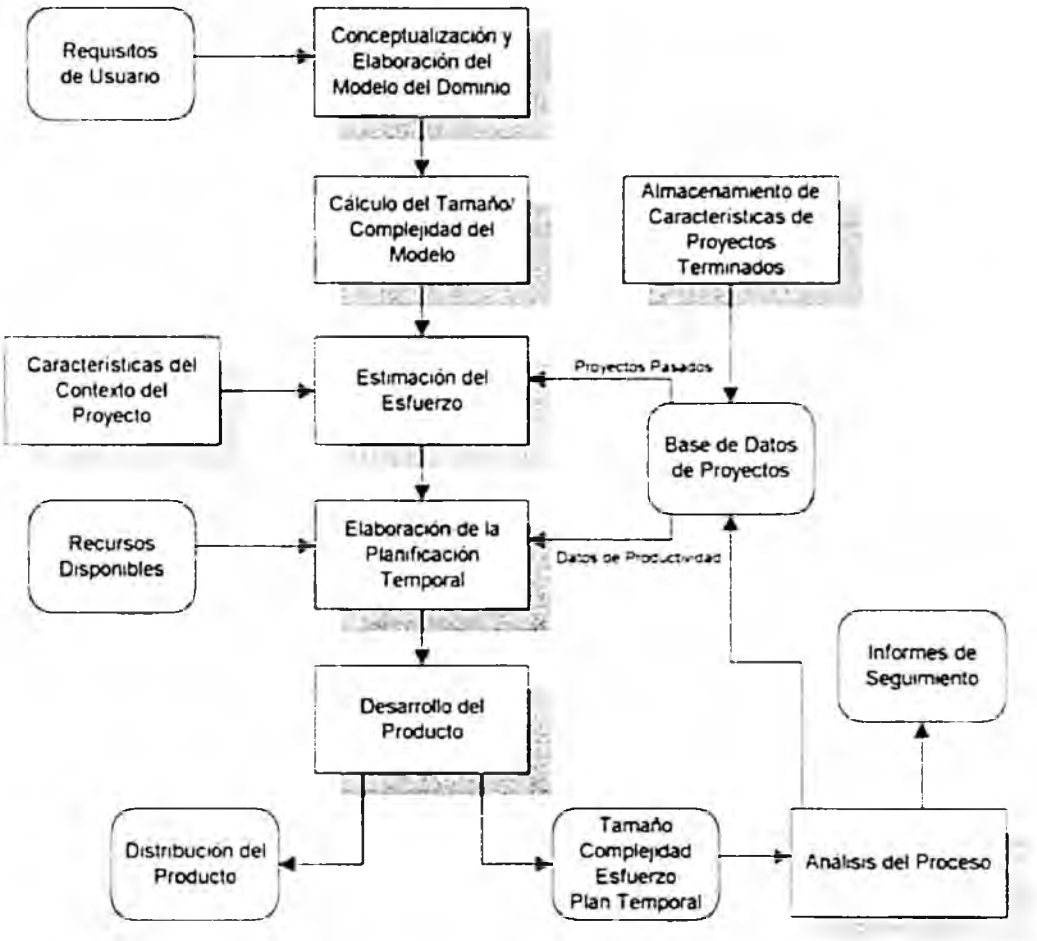

Figura 3.6 El proceso de Estimación.

Así que no podemos medir un objeto sin disponer de un modelo de dicho objeto. El modelo debe soportar la construcción de representaciones de medida, debe representar con fidelidad y explicar nuestra comprensión empírica sobre la naturaleza y la estructura de los objetos individuales y el modo en que se interrelacionan. Un único modelo deberia soportar la medición de múltiples características, tanto estáticas como dinámicas. Debe ser formal, matemático, para que pueda servir de base de medición fiable.

Vamos a definir un modelo formal de objetos, basado en la *Teoría de Categorías* [Barr95] y en el modelo de Whitmire [Whit96, Whit97], que permita la medición de las propiedades estáticas de un diseño y de conceptos empíricos como el tamaño, la complejidad, el acoplamiento, la cohesión, la suficiencia, la completitud, la similitud y la volatilidad. Dicho modelo es una versión abreviada que se ajusta a las necesidades de definición y cálculo de nuestras medidas.

Las categorías son una estructura matemática que generalizan y unifican muchas de las estructuras algebraicas de la Matemática Discreta. Nos ayudan a movernos entre estructuras algebraicas, preservando la estructura en sí, a medida que nos movemos de lo simple a lo complejo. Las categorías nos proporcionan varios constructores universales que sirven para explicar muchas de las propiedades del software orientado a objetos.

El modelo describe la estructura de una clase en términos de sus atributos y métodos. Describe el espacio de estado de una clase como el producto cartesiano de los dominios de los atributos y en consecuencia, describe el estado de un objeto individual como el resultado de una selección de este espacio de estado. El modelo describe las operaciones o servicios de una clase como transiciones entre estados[2]. Permite la definición de la comunicación por mensajes y la modelización de relaciones estáticas y dinámicas entre clases y objetos.

El modelo consta de cinco categorías: *Clase*, *EstadoDiseño*, *Diseño*, *Mensaje* y *Método*. La medición de las propiedades estáticas de un diseño se realiza utilizando las categorías *Clase* y *EstadoDiseño*. Una forma simple de representar las categorías es mediante grafos que muestren los objetos y flechas que representen a las funciones.

Los objetos de la categoría *Clase* son los dominios de atributos y los estados individuales en el espacio de estados. Las flechas en esta categoría serán funciones de proyección que representan a los atributos individuales, funciones entre estados que representan a las operaciones o métodos de una clase y funciones de selección de estados individuales a partir del espacio de estados.

Los objetos de la categoría *EstadoDiseño* son las clases del diseño. Las flechas son las relaciones entre clases: la generalización, la asociación, la agregación y la comunicación de mensajes.

---

[2] Aunque no es esta exactamente la visión que nos interesa de las operaciones, como se verá más adelante, es la que prevalece en la mayoría de los modelos basados en la Ontología

Las categorías *Clase* y *EstadoDiseño* son las que permiten la medición directa de conceptos empíricos como el tamaño, la complejidad, etc.

La tercera categoría, *Diseño*, se utiliza para poder realizar simulaciones algebraicas de un diseño determinado y manipulaciones algebraicas de la estructura del diseño en sí. Los objetos de esta categoría son estados del diseño. Las flechas son las operaciones que un diseñador puede utilizar para alterar un diseño (por ejemplo, añadir una clase). Combinando las tres categorías, podemos medir las características estáticas de un diseño o determinar cual será la respuesta del diseño ante un cambio o una secuencia de cambios. Así, no solo se puede medir la complejidad de un diseño, sino que también se puede medir su complejidad después de modificarlo, sin haber hecho realmente ningún cambio. En resumen, permite la toma de decisiones ante la valoración de lo que supone alterar un diseño, simulando las alteraciones, sin realizarlas.

Las categorías *Mensaje* y *Método* se utilizan para definir los mecanismos universales de paso y el enlace dinámico de mensajes. Los objetos de ambas categorías son los elementos atómicos de los contenidos de los mensajes, interpretados como los parámetros de los métodos. Las flechas son funciones de selección.

3.2.1 La categoría Clase

El mundo real está compuesto de *cosas*, a las que vamos a llamar objetos [Bung77, Wand90]. Existen dos clases de objetos: *concretos* (llamados *entidades* o *individuos substanciales*) y *conceptuales*. La Ontología considera a los objetos concretos y a los conceptuales de forma diferente.

En el software, todos los objetos son conceptuales ya que únicamente son *representaciones*, *abstracciones* o *modelos* de los objetos conceptuales y concretos de la realidad. Trataremos a las representaciones de ambos tipos de objetos de la misma forma.

Clases de Objetos

La unidad fundamental del software orientado a objetos es el *objeto*. Definimos un objeto como un conjunto de propiedades (atributos y métodos) y un conjunto de funciones de estado que devuelven los valores en curso de cada atributo. Si se ignoran los valores de los atributos, podemos describir grupos de objetos por medio de su conjunto de atributos y métodos. De hecho, podemos

utilizar el conjunto de atributos y métodos para particionar el conjunto de objetos en un conjunto más pequeño de clases de equivalencia, a los que vamos a llamar, simplemente, *clases*.

El particionar un conjunto de objetos en clases de equivalencia es una operación estándar de la teoría de conjuntos. Además, es exactamente la operación que se usa para agrupar objetos de software en clases. Según Booch [Booc94, Booc96], la estructura y el comportamiento de objetos similares se puede definir en términos de su clase común. Define una clase como un conjunto de objetos que comparten una estructura común (atributos) y un comportamiento común (métodos). Podemos describir la relación entre una clase y sus objetos desde dos puntos de vista, el de la clase y el de los objetos:

1. Clase: conjunto de propiedades compartidas por un conjunto de objetos en una clase de equivalencia.

2. Objeto: instancia de una clase.

El concepto de *clase* se puede ver como:

1. Una clase es un modelo teórico de un objeto.

2. Una clase define un conjunto de atributos y operaciones genérico que se aplican a todos los objetos que pertenecen a la clase.

3. Una clase es una plantilla de un objeto que se utiliza para construir objetos específicos.

4. Una clase es un esquema funcional de un objeto [Wand90].

Atributos de una clase

Vamos a formalizar las nociones de atributo, miembro de una clase y espacio de estados de una clase, en términos de clases en vez de objetos, porque estos conceptos son aplicables a todos los objetos de una clase.

Formalmente, un *atributo* es el nombre de un conjunto de valores potenciales, llamado *dominio del atributo*, o simplemente *dominio*. Una clase puede imponer limitaciones directas sobre los posibles valores de un atributo. Estas limitaciones consideradas de forma colectiva se denominan *invariante de clase*. En la Ontología de Wand y Weber se les llama *afirmaciones de ley* [Wand90]. Limitan los valores potenciales de un atributo a un subconjunto de su dominio. También pueden restringir el valor de un atributo en función de los valores de otros atributos. Por

tanto, es un conjunto de condiciones que cada objeto de una clase debe satisfacer en cualquier *estado estable* de la clase. Al igual que en la Ontología, un estado es *inestable* si se viola cualquier condición de la invariante de clase. Un objeto puede entrar en un estado inestable en respuesta a una petición de servicio por parte de otro objeto o de una entidad externa a la aplicación. Una definición de clase debe contemplar una transición de cualquier estado inestable a uno estable. Estas transiciones son parte del comportamiento que se haya definido para la clase. En conjunto, el dominio de un atributo y la invariante de clase definen el universo efectivo del atributo.

Una clase es un álgebra $<A, f_i : i \in I>$ que consta de un conjunto A de atributos y una familia $(f_i : i \in I)$ de operaciones. Cada elemento del conjunto A es un conjunto de valores, no un valor en particular. El *espacio de estado* de una clase es un subconjunto del producto cartesiano de los conjuntos de dominio, por lo tanto $E_c \subseteq A^n$. $A^n = a_1 \times a_2 \times \ldots \times a_n$, $a_i \in A$. El espacio de estado consiste en n tuplas donde n es la cardinalidad del conjunto A. Es posible que la invariante de clase no permita la existencia de algunas tuplas; por tanto, se excluyen del espacio de estado y es el motivo por el que hemos definido el espacio de estado como un subconjunto de A^n. El tamaño del espacio de estado es una función del número de atributos y del número de valores potenciales que cada atributo puede tener. La cardinalidad de A^n nos proporciona el límite superior del tamaño del espacio de estado.

El dominio de un atributo es independiente de la clase o clases a las que pertenezca el atributo. Como un dominio es un conjunto de valores, cualquier atributo, en cualquier clase, cuyo dominio contenga el mismo conjunto de valores tiene el mismo dominio debido a la definición de la igualdad de conjuntos. Esta igualdad, no sólo equivalencia, es independiente del nombre del atributo. De aquí se deduce que dos atributos, independientemente de sus nombre en sus respectivas clases, son el *mismo atributo*, si y solo si comparten el mismo dominio. Podemos definir el *ámbito* de un atributo a como el conjunto de clases a los que el atributo pertenece. Formalmente:

$$Am_a = \{c: \text{clase} \mid b \in c \wedge \text{dom } a = \text{dom } b\}$$

Valor en curso de un atributo

Las clases tienen un espacio de estado y los objetos tiene un estado. El estado de un objeto contiene uno y sólo un valor para cada atributo de la clase a la que pertenece el objeto. Por el modo en que hemos definido el espacio de estado, sabemos que el valor de un atributo es un miembro de su dominio.

Podemos seleccionar un estado *j* del espacio de estados utilizando una función de selección σ_j. Mediante una función de proyección π_{ij}, podemos obtener el valor del atributo *i* del dominio en el estado *j*. Esta correspondencia se puede realizar directamente utilizando una función de selección/proyección σ_{ij}.

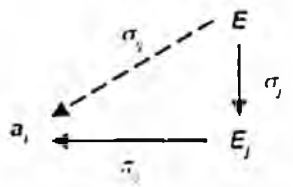

Figura 3.7 Espacio de estados y valor de un atributo.

El estado de un objeto se determina combinando diagramas de valores de atributos para todos los atributos de una clase. En realidad, se pueden representar dos diagramas con dos visiones distintas. El primero se usa a nivel de clase y se basa en el espacio de estado y representa el cambio de los valores de los atributos basándose en el cambio de estado de la clase (σ_{ij}). El segundo se usa a nivel de objeto y se centra en un estado dado al que le corresponden unos valores para los atributos (π_{ij}).

Operaciones de una clase

En la Ontología, un *evento* provoca que un objeto transite de un estado a otro, o al mismo estado, según lo determine su comportamiento. Vamos a modelizar formalmente un evento como un par ordenado $P = <E_1, E_2>$, donde E_1 es el estado de inicio y E_2 es el estado final (pueden ser el mismo estado) [Wand90]. El estado E_2 lo determina el comportamiento del objeto y el estado E_1 depende del modelo de objetos de la clase.

Vamos a modelizar el comportamiento de un objeto como el conjunto de operaciones definidas para la clase. Estas operaciones de clase se aplican a todos los objetos de la misma.

En la Figura 3.8, las flechas *op* representan a las operaciones de la clase. Así, en esta clase, la operación o evento op_1 provoca que un objeto en el estado E_1 transite al estado E_2. La operación op_3 no tiene ningún efecto si el objeto se encuentra en otro estado que no sea E_3, al no haber ninguna otra flecha con esta etiqueta en el diagrama. El objeto ⊙ es el estado nulo. La flecha *Crear* representa la creación del objeto y la *Destruir*, su destrucción.

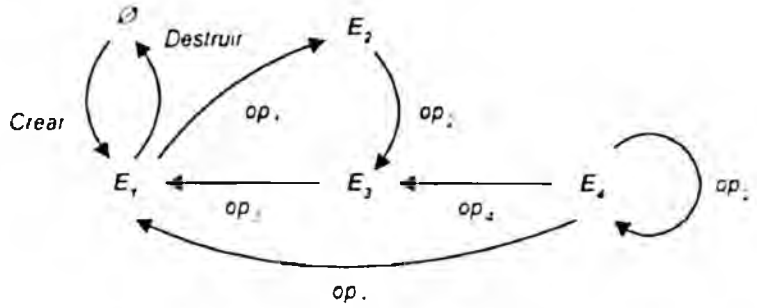

Figura 3.8 Representación de las transiciones de estado de una clase.

Las operaciones de *Crear* y *Destruir* se definen implícitamente para todas las clases. La operación de *Crear* asigna un identificador al nuevo objeto, lo convierte en miembro del conjunto de objetos activos y lo sitúa en el estado inicial definido para la clase de la que es instancia. Por tanto, se compone de tres funciones:

1. *Asignar identificador*: un identificador tiene la forma *nombre n* donde *nombre* es el nombre de la clase a la que pertenece el objeto y *n* es un número ordinal que denota el número de instancias de la clase que han sido creadas.

2. *Activar objeto*: añade el identificador al conjunto de Objetos Activos.

3. *Inicializar*: asigna a cada atributo el valor definido en el estado inicial de la clase.

3.2.2 La categoría EstadoDiseño

Las relaciones que se dan entre clases definen las relaciones posibles o permisibles entre los objetos de dichas clases. Los objetos no se pueden relacionar de formas que no estén incluidas en el conjunto de relaciones de clases. Una relación o asociación entre clases indica que en ciertos momentos, un número variable de objetos de una de las clases puede participar en una relación con uno o más objetos de otra clase. Booch, Rumbaugh y Jacobson [Booc97] han definido en UML dos tipos de relaciones entre objetos: la generalización y la asociación. La agregación se considera un tipo de asociación y la comunicación de mensajes, que creemos es otro tipo de relación, se considera de forma implícita.

Así, disponemos de las estructuras necesarias para modelizar los componentes de un diseño concreto, con una configuración estática. Denominaremos *estado del diseño* a esta configuración estática, ya que los cambios que un diseñador pueda

realizar, van a provocar un cambio de estado del diseño. Vamos a modelizar el estado en curso de un diseño en forma de categoría, llamada *EstadoDiseño*, en la que los objetos son categorías *Clase* y las flechas son relaciones entre clases. Para completar las definiciones, debemos definir la flecha identidad y la composición de parejas de flechas. Así, para cada objeto (de la categoría *Clase*), existe una correspondencia *sí-mismo*:$a \to a$ que sirve como flecha identidad y que representaremos con una flecha etiquetada con el nombre de la clase (Figura 3.9).

Figura 3.9 Representación de la flecha Identidad.

Las únicas parejas de flechas que se pueden componer en la categoría *EstadoDiseño* son las secuencias de generalizaciones.

Para mostrar que *EstadoDiseño* es realmente una categoría debemos demostrar que las flechas, en este caso relaciones entre clases, son efectivamente morfismos. Es más, debemos demostrar que dichos morfismos son de hecho funtores entre instancias de la categoría *Clase*.

Podemos definir formalmente la generalización, asociación y agregación como relaciones de la teoría de conjuntos. Para ello, recuérdese que una clase es un conjunto de propiedades, por lo tanto, no es inconsistente hacer referencia a una clase con el nombre de un conjunto. Utilizaremos f, g,... para representar las operaciones, a, b,... para representar objetos y x, y,... para representar atributos de clase u objeto. Para mostrar que a es un miembro de la clase A, lo expresaremos como $a \quad A$. Si x es un atributo de A, se representará como $x \subset A$ y si una operación f es un método de la clase A, será $f \subset A$. Sin embargo, para hacer referencia al valor de un atributo x en un objeto a, la representación será $a.x$. Para solicitar el servicio f del objeto a, la expresión será $\lambda a.f()$.

La Generalización

Una generalización es la relación que se establece entre una superclase y sus subclases. Sean A y B clases distintas. En términos ontológicos, A es una generalización de B si todas las propiedades de un objeto de A están en un objeto de B. Es decir:

1 $A \subseteq B$, y

2. $A \cap B = A$

Esto no quiere decir que la clase *A* pierda su identidad dentro de la clase *B*, ya que son clases distintas. Las propiedades de *A* no son miembros de *B* y esto simplemente se representa utilizando la relación de generalización. La combinacion de propiedades ocurre en el momento de la instanciación. Por tanto, un objeto de la clase *B* tendrá todas las propiedades de la clase *B* más todas las de la clase *A*, mientras que la estructura de clases mantiene a *A* y *B* como clases separadas.

Vamos a representar la relación de generalización entre las clases *A* y *B*, donde *A* es la generalizacion de *B*, con el diagrama expuesto en la Figura 3.10.

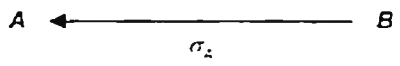

Figura 3.10 Relación de Generalización.

donde σ_2 es la función de selección que extrae los atributos y operaciones de *A* hacia *B*. La dirección de la flecha indica la dependencia de la clase *B* en *A*, y por tanto, representa el hecho de que las definiciones de *A* deben ser visibles para *B*.

La Asociación

Una asociación es una relación entre clases que implica la existencia de un conjunto de conexiones entre objetos de dichas clases [Booc97]. Una asociacion tiene un nombre y dos o más roles, uno por cada clase que participa en la relación. Una clase puede cumplir con múltiples roles en una asociación; cada uno de ellos se considera distinto. Sean *A* y *B* dos clases distintas, y sea *X* un subconjunto no vacio de los atributos de *A*. Un objeto $a \in A$ esta asociado con uno o más objetos $b \in B$ si existe un conjunto *Y* de atributos de *B*, tal que *Y* es una permutación de *X* en términos de dominios: un atributo de *Y* tiene el mismo dominio que un atributo de *X*, pero pueden tener distintos nombres. Es decir, existe una correspondencia $f: X \rightarrow Y$, que es biyectiva y preserva los dominios, pero no necesariamente los nombres. Como los atributos son flechas cuyo origen es el espacio de estado de una clase y cuyo destino es un conjunto dominio, la correspondencia $f: X \rightarrow Y$ es realmente una biyeccion entre conjuntos de flechas. Por tanto, las flechas en *X* en la representacion de *A*, compartirán codominio con las flechas de *Y* en la representación de *B*, pero pueden tener distintas etiquetas.

Dado un par de objetos $a \in A$ y $b \in B$, los atributos deben tener los mismos valores.

Podemos expresar estas condiciones formalmente de la siguiente forma:

$$\forall x \in X, y \in Y \bullet \exists a \in A, b \in B \; X \sqsubset A \wedge Y \sqsubset B \wedge \text{cod } x = \text{cod } y \wedge a.x = b.y$$

A los conjuntos X e Y se les denomina los *conjuntos anclaje* de la relación, puesto que son los que afianzan la relación. Como estos dos conjuntos son equivalentes en términos de dominio, se puede hacer referencia a cualquiera de ellos como *el* conjunto anclaje.

Se puede representar una asociación entre las clases A y B, añadiendo una flecha f que muestra la asociación entre clases.

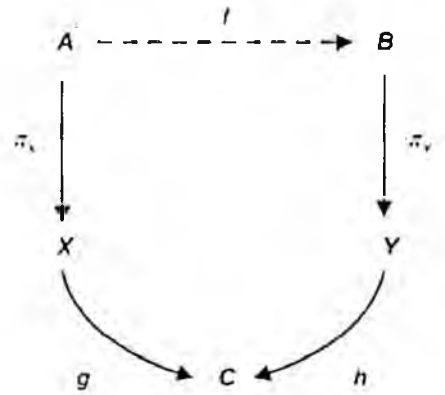

Figura 3.11 Relación de Asociación.

En el diagrama representado en la Figura 3.11, los objetos (de categoría) X e Y son los conjuntos anclaje previamente definidos. Las flechas π_X y π_Y son funciones de proyección de sus respectivos conjuntos anclaje desde las clases A y B. El objeto (de categoría) C es un conjunto de dominios de atributos e indica que los conjuntos anclaje X e Y comparten el mismo conjunto de dominios de atributos. C es un subconjunto de la intersección de los codominios de las flechas de atributos de A y B

La Agregación

La agregación es un caso especial de asociación con la connotación de una relación *Todo-Parte* [Booc97]. En una agregación, un objeto sirve de atributo de otro objeto. Sean A y B clases diferentes. La clase A es una clase *agregada* o *compuesta* y la clase B es un *componente* si A tiene como atributo a un objeto de la clase B: $B \in A$.

Las clases *A* y *B* se mantienen como clases distintas. La agregación y la asociación difieren en que en la asociación dos objetos *comparten* un conjunto de atributos, mientras que en la agregación un objeto *es* un atributo de otro. Podemos representar la relacion de agregación de la clase *B* como parte de la clase *A* con el siguiente diagrama (Figura 3.12):

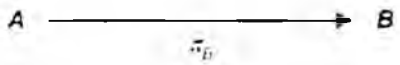

Figura 3.12 Relación de Agregación.

En el diagrama de la Figura 3.12, la flecha π_B representa la funcion de proyección del atributo, un objeto *b* de *B*, hacia su conjunto dominio, la clase *B*. Esto indica que cualquier objeto de la clase *B* puede ser el valor del atributo *b* en un objeto de la clase *A*, sujeto a las restricciones de la invariante de clase de *A*. Esta definición nos permite considerar a cualquier dominio de atributo como una clase en sí, y a cualquier miembro de un dominio como un objeto. La direccion de la flecha también es un indicador de los requisitos de dependencia y visibilidad de la relacion.

La Comunicación por Mensajes

Un método de un objeto puede invocar a una operación de otro objeto de la misma o distinta clase, mediante el envio de un mensaje. La relacion es *Emisor.Metodo* → *Receptor.Mensaje*. Un enlace de mensaje entre dos clases, indica la existencia de situaciones en las que un objeto de una clase puede enviar un mensaje a un objeto de otra clase.

Un enlace de mensaje se representa con el siguiente diagrama (Figura 3.13):

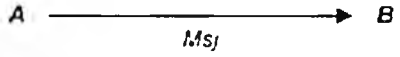

Figura 3.13 Comunicación por Mensajes.

que es un indicador de que un objeto de la clase *A* puede enviar un mensaje a un objeto de la clase *B*. La flecha se etiqueta con el nombre del mensaje y puede comenzar y terminar en la misma clase. En este caso, la flecha no se debe interpretar como una flecha identidad. La dirección de la flecha refleja la relacion de dependencia y los requisitos de visibilidad que deben cumplirse entre las clases *A* y *B*.

3.2.3 La categoría Diseño

Mediante la modelización del estado de un diseño como una categoría, en vez de modelos separados, podemos crear objetos en los que las flechas son operaciones de cambio del diseño. Esta nueva categoría, llamada *Diseño*, nos proporciona los medios para:

1. Modelizar los efectos de los cambios de diseño sobre el estado del mismo.
2. Disponer de una traza de la historia del diseño por medio de todos los estados previos.
3. Valorar el impacto de un cambio potencial antes de que el cambio se implante en la realidad.

Como esta categoría puede llegar a ser muy extensa (es la única que puede llegar a tener un tamaño infinito), sólo vamos a incluir los estados en los que el diseño ha estado en el pasado y las operaciones que conectan dichos estado, más algunos estados potenciales que serían el resultado de los cambios que se estén considerando.

Operaciones sobre las Clases

A medida que se trabaja con un diseño, se va modificando el conjunto de clases, el conjunto de relaciones entre ellas o la estructura de una o más de las mismas. En ello, se utilizan un número sorprendentemente reducido de operaciones atómicas. Cada operación provoca un cambio de estado en el diseño. Vamos a definir estas operaciones como funciones que hacen corresponder un estado del diseño con otro.

A continuación, vamos a definir formalmente las operaciones que actúan sobre las clases. Todas estas operaciones son flechas entre objetos de la categoría *Diseño* y funtores entre las categorías *EstadoDiseño*. Como funtores, cada operación tiene dos componentes: uno que opera sobre clases y otro que opera sobre las flechas entre clases. Como también hemos modelizado las clases como categorías, el componente que actúa sobre las clases, debe ser un funtor entre categorías *Clase*, con sus dos componentes propios. Uno de los componentes, establece una correspondencia entre los dominios de atributos de una clase y los de otra y los estados de una en los de otra. El otro componente establece una correspondencia entre las flechas entre estados y los dominios de atributos con sus parejas correspondientes en la nueva categoría. Algunas de estas correspondencias pueden

causar la eliminación de objetos o de flechas, mientras que otras pueden provocar la adición de los mismos.

Adición de una Clase

Esta operación añade una nueva clase al diseño, lo que implica añadir un nuevo objeto a la categoría *EstadoDiseño*. Vamos a denotar esta operación como $ED_2 = ED_1 + C$, donde ED_1 es el estado inicial del diseño y ED_2 es el estado después de añadir la clase *C*. La operación añade un nuevo espacio de estado, un nuevo conjunto de estados e incluso podría añadir un nuevo conjunto de dominios de atributos, junto con las flechas apropiadas, que definen a la nueva clase. El modelo básico de esta operación es la unión de conjuntos.

Por otra parte, la adición de una clase no es una operación trivial, puesto que dicha clase puede necesitar establecer todo tipo de relaciones con otras clases del diseño. Así, puede compartir uno o varios dominios con otras clases y lógicamente, establecer asociaciones de cualquier tipo con un número variable de clases del diseño

Eliminación de una Clase

Esta operación borra una clase en su rol de objeto de la categoría *EstadoDiseño*. Vamos a representar esta operación con la siguiente expresión $ED_2 = ED_1 - C$. Esta operación elimina un espacio de estados, un conjunto de estados, un conjunto de flechas de operaciones, de atributos y potencialmente un conjunto de dominios de atributos. El modelo básico de esta operación es la sustracción de conjuntos

No es una operación de sustracción simple puesto que debe considerarse, por ejemplo, el hecho de que la clase a eliminar comparta un dominio con otras clases, en cuyo caso el dominio no puede desaparecer.

Combinación de dos Clases

El propósito de esta operación es tomar dos clases distintas y fusionar sus respectivos conjuntos de propiedades, atributos y métodos, en una única y nueva clase, a la que se le puede denominar con el nombre de una de las clases de partida Por ejemplo, podemos combinar dos clases *A* y *C*, en una nueva clase a la que también llamaremos *C*.

Formalmente, esta operación, representada por la expresión $C_{ED2} = C_{ED1} \mapsto A_{ED1}$, es un funtor con cuatro funciones componentes. Una función de flecha de atributo, que es una operación de junción común del álgebra relacional, representada con $C \mapsto A$, establece la correspondencia entre las flechas en ED_1 con dominio = A y las flechas en ED_2 con dominio = C, reetiquetando las flechas como corresponda y eliminando las duplicadas (aquellas que después del reetiquetado, son iguales a flechas ya existentes).

Nótese que el nuevo espacio de estados es un subconjunto del producto cartesiano de la combinación de los conjuntos de atributos de las clases. Será igual al producto cartesiano si ninguno de los dominios de atributos se comparte.

Una función de estado elimina los objetos de estado de ambas clases, combina las invariantes de clase utilizando la conjunción y crea un nuevo conjunto de estados que satisfacen la invariante de clase combinada. Lo definimos como una suryección que lleva de los estados de las dos clases en ED_1 a los estados de la nueva clase combinada en ED_2 de forma que en cada estado de ED_2 se satisface la invariante de clase combinada.

Una función de flecha de operación hace corresponder las flechas de los estados de ambas clases en ED_1 con los estados de la nueva clase en ED_2. Esta correspondencia lleva el dominio de cada flecha para cada clase de ED_1 a todos los objetos de estado para la nueva clase en ED_2 que satisfacen las precondiciones de la operación. Puede ser una correspondencia de muchos a muchos. Los codominios de cada flecha para cada clase en ED_1 se hacen corresponder con todos los objetos de estado de la nueva clase en ED_2 que satisfacen las poscondiciones de la operación. Esta también puede ser una correspondencia de muchos a muchos. Se pueden crear nuevas flechas en ED_2 que serán etiquetadas con el nombre de la operación. Se descartan las flechas duplicadas, aquellas con el mismo dominio, codominio y etiqueta.

Una función de flecha de relación entre clases establece una correspondencia entre las flechas para las que una de las clases individuales es el dominio o codominio de la nueva clase. Esta suryección preserva las relaciones entre clases.

Nótese que después de combinar las clases de este modo, la extracción de una de las clases originales no tiene por qué ser necesariamente la operación inversa. Puede no ser posible reconstruir las dos clases originales con la información contenida en ED_2. Esto ocurrirá en el caso de que las intersecciones de los conjuntos de propiedades de las dos clases originales no sea el conjunto vacío.

Extracción de Propiedades de una Clase

En esta operación, se añade una nueva clase al diseño, pero se toman todos los atributos y operaciones de una clase ya existente. En principio, esta operacion es mas simple que la adición de una nueva clase, porque no tenemos que añadir nuevos dominios de atributos o flechas de atributos. Sin embargo, puede resultar bastante complicada con respecto a los estados. Se debe añadir un nuevo espacio de estados

Adición de una Relación de Generalización

Es una operacion bastante simple: no hay precondiciones La significatividad de esta nueva flecha no se pone de manifiesto hasta que la especializacion recibe un mensaje que invoca a una operacion de la generalización. El efecto de esta operacion es que las propiedades de la clase generalizacion (superclase) son posteriormente tratadas como un subconjunto de la clase especialización (subclase)

Es poco probable que la adicion de una relacion de generalizacion cree flechas de operacion duplicadas, pero es posible Ademas, no podemos eliminar sin mayor consideracion las flechas de operación duplicadas (aquellas en las que coinciden los dominios, codominios y las etiquetas), a menos que sean realmente duplicadas, incluyendo los efectos secundarios que puedan provocar (que se especifican en las poscondiciones). Es posible redefinir una operación heredada en una subclase y enviar un mensaje a la superclase para hacer que la operación heredada realice lo que viene a ser el comportamiento por defecto. Por consiguiente, dejamos en manos del diseñador el decidir qué flechas de operacion estan duplicadas y pueden ser eliminadas

Adición de una Relación de Asociación

La adicion de una asociación está sujeta a unas precondiciones bastante estrictas Debe recordarse que hemos definido una asociacion en terminos de conjuntos anclaje que representan a un subconjunto de flechas de atributos cuyos codominios participan en una intersección. Gráficamente, la situacion es la expuesta en la Figura 3.14.

Esta situación debe existir antes de que se añada la asociación *f* representada por la flecha $f\ A \rightarrow B$, si el objeto de la clase *A* depende del objeto de la clase *B*, o $f\ B \rightarrow A$, si la dependencia es a la inversa, es decir, el objeto de la clase *B* depende del objeto de la clase *A*.

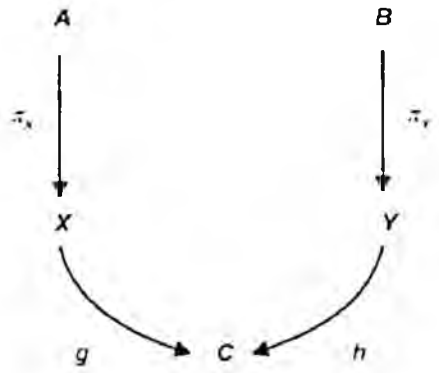

Figura 3.14 Situación antes de añadir una asociación.

La semántica de una nueva asociación depende de las necesidades de la aplicación. A diferencia de la adición de una generalización, la adicion de una asociacion es un tema puramente estructural o sintáctico.

Adición de una Relación de Agregación

Hemos definido una agregación como una relación en la que un objeto de una clase es un atributo de otra clase. La primera clase mencionada es el componente y la segunda es la clase compuesta. En términos semánticos, una relación de agregación entre clases no es diferente a una relación entre una clase y un dominio de atributos.

De hecho, en lo que concierne a la clase compuesta, la clase componente *es* un dominio de atributos. Por tanto, podemos extender esta operación para que admita la adición de flechas de agregación entre una clase y un dominio de atributos, así como también entre dos clases. Esta situación se representará con la adición de una flecha $\pi_\theta : A \rightarrow B$.

Adición de una Conexión de Mensajes

Esta operación virtualmente no tiene ningún efecto en la estructura del diseño. Esta operación se representa como *msj*: $A \rightarrow B$ y añade una flecha que llevará como etiqueta el nombre del mensaje.

Eliminación de una Relación

Esta operación, independientemente del tipo de relación que se elimine, suprime una flecha de la categoría *EstadoDiseño*, y no tiene ningún otro efecto. La estructura de una clase, el conjunto de clases y el conjunto de dominios de atributos no se ven afectados. Sólo existe una excepción: la eliminación de una relación de agregación es lo mismo que la eliminación de una flecha de atributo, ya que en ambos casos se elimina un atributo del espacio de estados de la clase. Esto puede tener efectos significativos en los estados de la clase y en la invariante de clase. Puede incluso afectar a las precondiciones y poscondiciones de una o más operaciones de clase.

La supresión de una relación puede también tener efectos semánticos si se desea simular el comportamiento del diseño. Por ejemplo, eliminar una relacion de generalización puede tener como consecuencia que un conjunto de atributos y métodos que se asume que son parte de la clase, dejan de serlo, dejen de estar presentes Un objeto que pertenece a una subclase puede dejar de reconocer ciertos mensajes que anteriormente era capaz de interpretar.

Todo el formalismo expuesto en las secciones anteriores tiene como objetivo la medición de conceptos como el tamaño, el acoplamiento, la cohesion, la complejidad, etc. En lo que concierne a esta investigación, es un modelo excesivamente amplio, que admite una serie de simplificaciones considerando la información de la que disponemos en la fase del ciclo de vida en la que nos interesa definir y calcular las medidas. Por ello, en la siguiente sección describimos en qué va a consistir exactamente el modelo sobre el que tomaremos nuestras medidas

3.2.4 Descripción del Modelo del Dominio

Como hemos expuesto en la sección 3.1.2, el Análisis del Dominio es una fase crucial en el desarrollo orientado a objetos, debido a que es en esta fase en la que se analizan las áreas de conocimiento en las que un problema se localiza y se producen representaciones abstractas y una red de relaciones que forman los cimientos del sistema de software y que se utilizarán durante el resto del proceso de producción

Existen distintos enfoques para documentar el resultado de la fase de análisis del dominio y que se pueden usar como fuente para el cálculo de las métricas que se van a proponer. Los tres métodos más populares son los de Booch [Booc94, Booc96], OMT de Rumbaugh [Rumb91] y el de Ivar Jacobson [Jaco92]. Estos tres autores se han unido en un esfuerzo común con el objetivo de crear un único método que reúna las

ventajas de los suyos propios y que se convierta en un estándar en la industria. Han propuesto una notación unificada que se denomina *UML (Unified Modeling Language)* y es la que hemos elegido para las representaciones a lo largo de esta exposición [Booc97]. Otro método muy popular para la fase de análisis del dominio es el de las *tarjetas CRC*, cuya nomenclatura se ajusta muy bien a las necesidades de información en esta fase inicial [Beck89, Wirf90] y el método de Coad y Yourdon que utiliza unas representaciones bastante extendidas por su sencillez [Coad91a].

Como el modelo matemático de Whitmire resulta difícil de automatizar y además, las herramientas CASE habituales implementan modelos como los propuestos en UML, que son suficientes para nuestro propósito, hemos elegido esta última opción.

En la sección 3.1.2 se describen los pasos que se deben llevar a cabo durante el Análisis del Dominio y se indica las técnicas que se deben utilizar. En los comienzos de esta fase, la técnica más importante es el llamado *diagrama de clases*, donde se documentan las *clases del dominio*; este diagrama se va completando con la información que surge a partir de las demás técnicas. Cada clase se define de forma parcial mediante la *lista de servicios o responsabilidades* que ofrece, representando estos servicios el comportamiento o funcionalidad pública de dichas clases del dominio. Además, para poder cumplir con sus responsabilidades, si una clase necesita interactuar con otras clases del sistema, a estas interacciones de les denomina *colaboraciones* o *interdependencias*. Las clases con las que colabora una dada se denominan *colaboradores*, dándose estas colaboraciones a través de la generalización, la agregación, la asociación y la comunicación de mensajes. Nótese que estamos reflejando la categoría *EstadoDiseño* (formada por objetos de la categoría *Clase*, siendo las flechas las relaciones entre clases) y una visión simplificada de la categoría *Clase*. Como hemos expuesto en la sección 3.2.1, la categoría *Clase* tiene como objetos los dominios de los atributos y los estados de la clase, estando las operaciones representadas mediante flechas entre dichos objetos. Como esta visión, en la que las operaciones están vinculadas a los cambios de los atributos y estados de la clase, no nos interesa para las definiciones de nuestras medidas, proponemos simplemente considerar las clases, desde el punto de vista del álgebra relacional, como conjuntos de operaciones, a los que vamos a llamar *servicios* o *responsabilidades*.

En resumen, utilizaremos un modelo del dominio abreviado, que muestra las responsabilidades de las distintas clases del área de conocimiento. El modelo debe mostrar la lista de servicios o responsabilidades de una clase o entidad y sus enlaces o dependencias con los colaboradores, si los hay.

En la Figura 3.15 se muestra la clase *TransConsultaSaldo*, que colabora con las clases *CuentaCorriente*, *CuentaAhorro* y *RegTrans* y posee dos responsabilidades o servicios

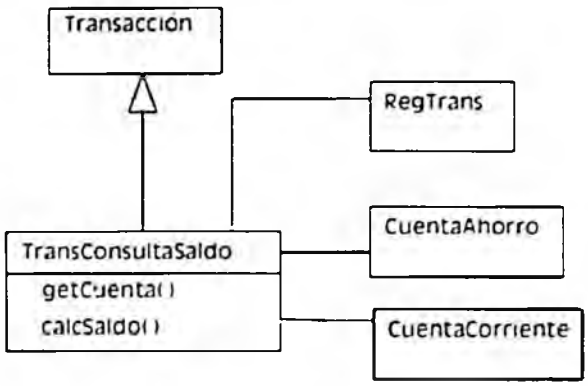

Figura 3.15 Ejemplo de Entidades, Responsabilidades y Colaboraciones.

Notese que en el modelo ilustrado en la Figura 3.15, no se puede determinar exactamente qué interdependencias se dan como resultado de cada responsabilidad o servicio propuesto. Por lo tanto, un servicio puede necesitar cero o mas colaboradores, pero en conjunto hacen uso de todos ellos.

Así, *TransConsultaSaldo* es una clase hijo que hereda de una clase ascendente o padre llamada *Transacción*. Una clase puede heredar responsabilidades de sus clases ascendentes. Estas responsabilidades se denominan *responsabilidades heredadas* y seran parte de la funcionalidad de la clase hijo, es decir, de su *interfaz*. Los *mecanismos*[5] de las responsabilidades heredadas se habran creado ya en la clase ascendente y la clase descendente únicamente las reutiliza a través del mecanismo de herencia.

El resto de las responsabilidades se denominan *no heredadas* y se crean de forma especifica para la clase hijo. Hay veces en que las responsabilidades heredadas se deben redefinir o reimplementar en una clase descendiente; en estos casos se consideran como no heredadas y se ilustran de nuevo en la representación de la clase hijo.

La distinción que realizamos entre las responsabilidades heredadas y no heredadas es muy importante desde el punto de vista de la complejidad de una clase

[5] Mecanismo. Forma flexible de hacer referencia a las estructuras de datos y algoritmos que dan soporte a una responsabilidad en concreto

La *TransConsultaSaldo* tiene dos responsabilidades no heredadas tal y como se muestran en la Figura 3.15. Se usará el término *servicios o responsabilidad de una clase* para hacer referencia a las no heredadas. El término *servicios o responsabilidad total* se utilizará para denotar tanto a las heredadas como no heredadas, que de forma conjunta forman la interfaz externa de una clase.

3.2.5 El Modelo de Medición

El proceso genérico de medición del software es el representado en la Figura 3.16. La información de entrada al proceso de medición es el documento en el que se describe el modelo del software y por tanto, en el que se centran las medidas. La salida del proceso está formada por las medidas de los atributos del software. El documento de software utilizado depende directamente del proceso de desarrollo y del paradigma utilizado. En nuestro caso, el documento es un modelo de clases que representa al dominio del problema analizado.

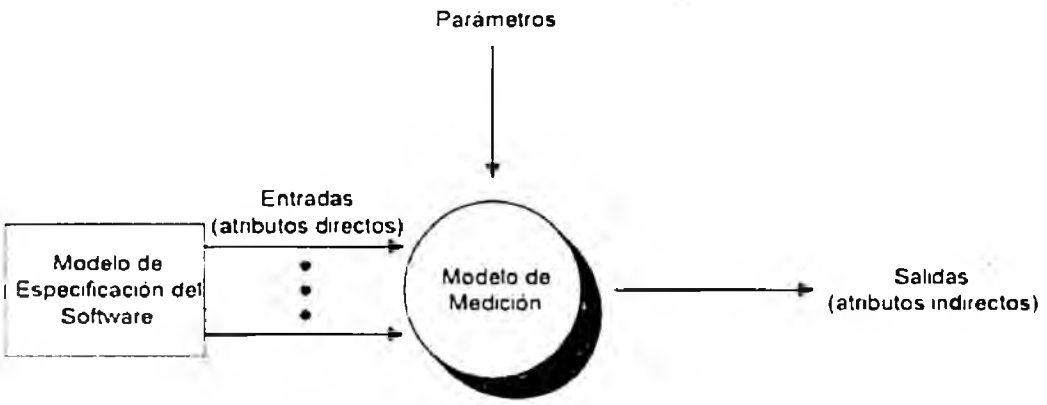

Figura 3.16 Modelización de la Medición.

El modelo del software que se utiliza como entrada al modelo de medición generalmente suele tener varias propiedades o atributos. Por ejemplo, el código orientado a objetos tiene operandos, operadores, clases, objetos, clases antecedentes, etc. En nuestro modelo de software vamos a disponer de clases, responsabilidades en forma de servicios y colaboradores en forma de clases con las que se debe interactuar. Un atributo que se puede medir directamente en un modelo de software es un *atributo directo*. Un atributo cuyo valor se basa en atributos directos se denomina *atributo indirecto*. Es muy común que las medidas del software se centren en un atributo indirecto, cuya medición implica a otros atributos.

Dependiendo de lo completo que sea el modelo del software, el número de atributos directos a utilizar puede ser variable. Cuantos más atributos se utilicen mas detallado será el modelo; así, una medida se expresa como una función matemática de los atributos. Si A_1, A_2,... A_n representan a los atributos y el atributo de salida es S, una medida es una función matemática f sobre los atributos:

$$S = f(A_1, A_2, \ldots A_n)$$

La precisión de la medición depende de si se han elegido los atributos correctos y de lo adecuada que sea la formulación de f. Por otra parte la eleccion del modelo más apropiado es más fácil en las fases iniciales del proceso de software ya que existen menos dimensiones de información para investigar. Por lo tanto, la parte más complicada es la obtención de la función f más adecuada.

3.3 UNA MEDIDA DE TAMAÑO

El *tamaño* es un atributo interno, clasificado como atributo de producto, según Fenton [Fent96], y que contribuye a una gran variedad de medidas de producto, de proceso y de proyecto. El concepto de tamaño tiene especial relevancia sobre nuestros sistemas predictivos, incluidos los que tienen como objetivo la estimación de esfuerzo y la planificación de proyectos. También permite normalizar otras medidas de producto que tienen tendencia a incrementarse a medida que va creciendo una aplicacion, de manera que esta normalización permite realizar comparaciones significativas entre productos, procesos u organizaciones de desarrollo [Whit97]

De manera superficial, el tamaño del software parece un concepto relativamente intuitivo. Sin embargo, un análisis más detallado lleva a la conclusion de que es un concepto bastante impreciso: tan impreciso que nunca se ha definido con detalle y rigor en el contexto del software. Uno de los problemas principales es que nuestra noción intuitiva de tamaño varia con los objetos que estan bajo nuestra consideración y con el punto de vista con el que los investigamos. Podemos medir el número de lineas de código fuente, el número de páginas de código, el número de clases de un modelo, el número de registros en una base de datos, etc.

El tamaño de un producto de software tiene varios orígenes, siendo el más importante el dominio del problema. El tamaño del producto medido a este nivel nos proporciona el tamaño mínimo de la solución. El proceso de diseño, que traduce esta descripción del dominio a una solución de software, únicamente puede

hacer que se incremente el tamaño. En otras palabras, el tamaño de la solución siempre es al menos igual al tamaño del problema.

Como hemos descrito en la sección 2.6.3.5, Haynes y Henderson-Sellers están de acuerdo con la afirmación anterior. Sus estudios empíricos les llevan a afirman que existe una relación entre el tamaño del dominio, medido en número de clases, y el tamaño de la solución [Hayn96].

McGregor también apoya esta misma hipótesis indicando que el número de clases del dominio localizadas durante la fase de análisis, se puede multiplicar por una constante que represente al número medio de clases de soporte que se deben desarrollar para cada clase del dominio, obteniéndose una estimación preliminar del número real de clases que se deberán implementar [McGr95]. Según McGregor, dada una organización y un contexto de desarrollo concreto, basta con examinar la información de dos o tres proyectos para obtener el valor de la constante que mejor se ajusta al proceso de desarrollo local. Lorenz y Kidd van más allá sugiriendo un valor para esta constante, basado en sus datos empíricos: el número de clases de soporte se estima multiplicando el número de clases del dominio por un factor comprendido entre 2 y 3. El número total de clases será el sumatorio de las clases del dominio más las de soporte [Lore94].

Este factor tambien ha sido propuesto a menos otras dos veces en la literatura: una vez por Burgett y Ohman [Burg95] que asignan a la constante un valor total de 3,6 y otra, por Haynes y Henderson-Sellers [Hayn96] que le dan un valor total de 3,5.

Humphrey también está de acuerdo con que el concepto de clase es un factor adecuado para la estimación [Hump95]. Establece una serie de criterios para que un factor pueda utilizarse en un modelo de estimación: debe tener una relación demostrada con el esfuerzo de desarrollo, se debe poder captar de forma automatizada y en las fases iniciales del proyecto, debe ser adaptable a las necesidades de una organización y adaptable a las necesidades de implementación. Afirma que los propios principios de la orientación a objetos sugieren que las clases son un buen factor en el que basar las estimaciones.

Por las razones expuestas anteriormente, vamos a utilizar como medida de tamaño una estimación del número de clases a desarrollar en un proyecto. Nuestros elementos empíricos de medición son los objetos de categoría *Clase* en la categoría *EstadoDiseño*, haciendo referencia al modelo planteado en la sección 3.2. Así, tomaremos *el número de clases del dominio de nuestro modelo de análisis como medida básica, y utilizaremos una constante con valor 3,5 como factor representativo del número medio de clases de soporte por cada clase del dominio.*

3.3.1 Validación Teórica: las Propiedades de Whitmire

A continuación, vamos a analizar con detalle el comportamiento de esta medida de tamaño, considerando las propiedades propuestas por Whitmire en la sección 2.5.5 y la Teoría de la Medición, para poder averiguar el tipo de escala de esta medida.

Nótese que estas propiedades fueron originalmente propuestas por Briand *et al.* [Bria96] y que Whitmire únicamente añade una propiedad más: el orden débil [Whit97].

Considerando la taxonomía que Whitmire realiza de las medidas de tamaño, la que nosotros consideramos es una medida de *población*, expresada en forma de cardinalidad de un conjunto, donde el conjunto está formado por las clases del dominio documentadas en un modelo preliminar de análisis.

Seguidamente, vamos a analizar con detalle el comportamiento de nuestra medida con respecto a las propiedades que una medida de tamaño debe cumplir. Denominamos *M* al modelo del dominio al que hacemos referencia y *NCls* representa la función de cálculo de la medida, siendo el resultado obtenido el número de clases del modelo.

- Propiedad T1: No Negatividad

  Dado un modelo M, el número de clases nunca podrá ser negativo.

- Propiedad T2: Valor Nulo

  El tamaño puede ser nulo, si estamos tratando con un modelo vacío, sin clases.

- Propiedad T3: Aditividad para Conjuntos Disjuntos

  El tamaño es aditivo para modelos que no tienen clases en común:

  $M1 \cap M2 = \emptyset \Rightarrow NCls(M1 \cup M2) = NCls(M1) + NCls(M2)$

- Propiedad T4: Monotonicidad Creciente

  Añadir clases a un modelo no puede hacer que su tamaño disminuya:

  $M1 \subseteq M2 \Rightarrow M1 \leq M2$

- Propiedad T5: No Sinergía

  El tamaño de la fusión de dos modelos no puede exceder la suma de ambos

  $M = M1 \cup M2 \Rightarrow NCls(M) \leq NCls(M1) + NCls(M2)$

- Propiedad T6: Orden Débil

  Sea $\bullet\geq$ una relación binaria que denota 'número de clases mayor o igual'. Es un orden débil:

  $M1 \bullet\geq M2 \vee M2 \bullet\geq M1$

  $M1 \bullet\geq M2 \Leftrightarrow NCls(M1) \geq NCls(M2)$

Desde el punto de vista de la Teoría de la Medición, nos puede resultar de interés considerar el tipo de escala de la medida. Para ello, y siguiendo a Zuse [Zuse98], podemos hacer una comprobación con respecto a los axiomas de la estructura extensiva que se enumeran en la sección 2.5.2.

El requisito de orden débil lo hemos demostrado anteriormente. Para poder analizar los demás axiomas, definimos la operación de concatenación como la operación de *adición de una clase* a un modelo, que básicamente se comporta como la unión de conjuntos clásica. Desde el punto de vista del modelo formal definido en la sección 3.2, esta operación añade un objeto a la categoría *EstadoDiseño*.

El axioma de asociatividad no presenta ninguna pega puesto que la unión de conjuntos es asociativa. El problema se plantea con la monotonicidad planteada con la siguiente expresión:

$M1 \bullet\cdot M2 \Leftrightarrow M1 \cup M3 \bullet\cdot M2 \cup M3 \Leftrightarrow M3 \cup M1 \bullet\geq M3 \cup M2$

Es obvio que esta expresión será falsa en el momento en que intentemos combinar modelos que tienen clases en común. Así, se puede dar el caso de que M3 tenga clases en común con M1 ($M1 \cap M3 \neq \emptyset$) y no las tenga con M2 ($M2 \cap M3 = \emptyset$). Al unir M1 con M3, las clases repetidas se contabilizan una única vez y por lo tanto, la expresión anterior es falsa.

La conclusión es que nuestra *representación no es una estructura extensiva* y por lo tanto, la *medida no es de escala de ratio*, será de *intervalo* (a menos que cumplamos los requisitos de otra estructura que lleve a ratio).

La situación anterior surge del hecho de considerar que dos modelos pueden contener clases comunes. Si podemos garantizar que los modelos a combinar son disjuntos, es decir, que no comparten ninguna clase, podemos contar con la *estructura extensiva* y con *una medida en la escala de ratio*.

3.4 UNA MEDIDA DE COMPLEJIDAD

3.4.1 Modelos de Complejidad

La complejidad es probablemente la característica del software mas estudiada y menos comprendida Fenton [Fent96] lo achaca a la falta de una comprension empírica unificada de la complejidad como fenómeno. Los muchos intentos de definir la complejidad se agrupan principalmente en dos tendencias: aquellos que la definen en términos del esfuerzo o recursos que un sistema externo consume para interactuar con un componente de software, o aquellos que la definen en terminos de la estructura del componente [Whit97].

La complejidad se ha definido de diferentes maneras a lo largo de los años siendo algunas de dichas definiciones las siguientes:

- "... una característica de la interfaz del software que tiene influencia sobre los recursos que otro sistema consumira durante la interaccion con dicho software".
- "... una medida de los recursos consumidos por un sistema durante la interacción con un elemento de software, con el objetivo de realizar una tarea"
- "una medida del esfuerzo mental que se requiere para comprender un objeto"
- "La complejidad cognitiva del software hace referencia a esas características del software que afectan al nivel de recursos que una persona utiliza para poder realizar una tarea sobre el"
- "La complejidad de un objeto es una funcion de las relaciones entre los componentes de dicho objeto"
- "La complejidad denota el grado de esfuerzo mental necesario para la comprensión".

Todas estas definiciones ilustran la variedad de puntos de vista que prevalecen sobre la complejidad. Dada toda esta variedad, parece que la complejidad cubre distintos aspectos independientes. En consecuencia, nunca se podra definir una única medida de complejidad que intente representarlos a todos ellos [Fent96]

Nuestro interés se centra en la complejidad como un factor que afecta al esfuerzo necesario para desarrollar software. La tendencia tradicional en la estimación de recursos ha sido considerar siempre como variable independiente una medida de tamaño, dejando la complejidad únicamente como un factor adicional, algo secundario. Nosotros proponemos llevar a cabo la estimación de recursos, basandonos por igual en ambos conceptos: tamaño y complejidad.

Esta idea viene avalada tambien por otras disciplinas relacionadas con la fabricacion de productos [Houl94]. Así, como Hoult describe:

> «Considerese el siguiente problema su firma dispone de dos diseños de un componente, A y B. Los dos diseños tienen aproximadamente el mismo tamaño y el material y el proceso de produccion es el mismo. Sin embargo, B es mas complejo que A. Se debe determinar cuanto mas costara B que A.» [Houl94]

Hoult sugiere que existe una medida de complejidad subyacente y que el tiempo de desarrollo de un componente es proporcional a su complejidad.

Simon Moser señala que el termino *tamaño* para las medidas que se van a utilizar con vistas a la estimacion del esfuerzo no es muy adecuado. Según el, deberian llamarse *propiedades que inducen esfuerzo*, porque el termino *tamaño* en dominios ajenos al software, no esta necesariamente relacionado con el esfuerzo (requiere mas esfuerzo construir un pequeño reloj que clavar varias tablas enormes para hacer una gran caja de madera; sugiere una relacion entre esfuerzo y complejidad) [Mose96].

La vision de Henderson-Sellers también se ajusta a esta idea, como exponemos a continuacion:

> «Para estimar el esfuerzo (medida de proceso) tradicionalmente se ha establecido una conexion con el *tamaño* total (medida de producto) del sistema a desarrollar mediante una valor de la productividad del equipo de desarrollo especifica de la organizacion. El uso del tamaño de entre todas las medidas de producto disponibles, como dato de entrada a la ecuacion del modelo de proceso, no deberia sugerir que el tamaño es la mejor medida de producto, es la mas obvia y facil de medir, aunque desafortunadamente solo esta disponible al final del ciclo de vida. En consecuencia, un estudio mas riguroso del dominio de las medidas de producto sugieren que la *complejidad*, y no el tamaño, puede ser mas relevante en los sistemas modernos. Podemos definir la *complejidad* de forma flexible como esa propiedad del software que hace que se requiera esfuerzo (recursos) para diseñarlo, comprenderlo y codificarlo. Probablemente esta definicion tan superficial sea la mejor que se puede hacer para el termino generico *complejidad*.» [Hend96 50]

Otra tendencia es la señalada por MacDonell [MacD94a]. Así, indica que aunque la complejidad rara vez se define mediante términos medibles, la mayor parte de los estudios sobre esta propiedad aceptan la asociación intuitiva entre los aspectos de tamaño de un sistema y la interconectividad, con la complejidad global de un producto. Por lo tanto, a su modo de ver, la complejidad es una función del tamaño de un producto y la interconectividad. De aquí se deduce que a medida que el tamaño y la interconectividad se incrementan, la complejidad del sistema global se incrementa y consecuentemente, el esfuerzo de desarrollo es mayor.

Whitmire es contrario a esta idea, en lo que respecta a la medición. Como ya explicamos en la sección 2.5.5, Whitmire considera que una medida de complejidad debe ser independiente del tamaño [Whit96].

En cuanto a la noción de interconectividad, en relación a la complejidad, este supuesto ya fue utilizado por Henry y Kafura [Henr81] para la definición de su medida de complejidad basada en el número de flujos de información de entrada y en el número de flujos de información de salida de un procedimiento. Según ellos, la reducción de costes y el incremento de la calidad son objetivos compatibles que se pueden alcanzar cuando la complejidad de la estructura del software sea controlada adecuadamente.

Su teoría del flujo de información ha tenido un número importante de seguidores que han propuesto modificaciones que mejoraban el planteamiento clásico. Recientemente, un estudio empírico ha demostrado que el flujo de información de salida es más importante que el de entrada y el que tiene correlación con el tiempo de desarrollo [Fern97].

En relación a la orientación a objetos, los resultados de estudios recientes indican que los métodos tienden a ser pequeños, tanto en términos del número de sentencias como en complejidad lógica, sugiriendo que la estructura de conectividad del sistema puede ser más importante que el contenido de los métodos individuales [Chur95]. Así, siguiendo la teoría del flujo de información, Lee, Liang, Wu y Wang [Lee93, Lee94, Lee95], definen sus medidas de acoplamiento y cohesión.

Kolewe nos proporciona su noción de complejidad sugiriendo que la medición de este concepto es especialmente importante en el diseño y en la programación orientada a objetos, donde la complejidad se manifiesta de nuevas maneras [Kole93]. Señala que las medidas tradicionales pueden ser lingüísticas o estructurales y que sin embargo, las orientadas a objetos tienen tendencia a ser estructurales. Además afirma que estos sistemas requieren dos niveles de medidas que se corresponden con los dos tipos de estructuras que se encuentran en ellos: en primer lugar, medidas

a nivel de clase, que midan la complejidad de las clases que componen el diseño y en segundo lugar, como un sistema orientado a objetos está formado por múltiples objetos que interactúan, medidas a nivel de sistema, que midan la complejidad de las interacciones de los objetos del sistema.

Kolewe también sugiere que estas interacciones a lo mejor no siempre llevan a un incremento de la complejidad. (De hecho, un sistema puede simplificarse cuando se introduce una nueva clase, como por ejemplo, cuando se factorizan los comportamientos comunes a dos clase sobre una nueva; esto esta relacionado con la generalización que hay que favorecer, porque simplifica; también muestra que la adición de una interdependencia nueva puede hacer que la complejidad disminuya).

Henderson-Sellers proporciona una clasificación realmente exhaustiva del concepto de complejidad del software en su libro [Hend96]. Según él, existen varios tipos de complejidad que son características externas (o atributos) del software. La más discutida es la complejidad estructural, que se cuantifica mediante medidas de complejidad estructural. Estas medidas se pueden centrar en el nivel de módulo (medidas intramódulo) o a nivel intermódulo y medir los atributos internos del codigo o del diseño.

Así, en la Figura 3.17, se ilustra esta clasificación, donde llama la atención la consideración que se hace de las medidas de tamaño como un tipo de medida de complejidad.

Por último, una de las cuestiones que es inevitable plantear es si la medición de la complejidad de una representación tan inicial como la del análisis del dominio es válida. Es una cuestión más profunda de lo que aparentemente parece: puede hacer que nos plateemos los propios fundamentos de la Ingeniería del Software.

Debemos convencernos de que la información disponible en las primeras fases del desarrollo son un reflejo de lo que va a obtenerse a continuación y del producto final. Por supuesto, esto también ocurre para los productos de la fase de análisis de un proceso de producción orientado a objetos, pero con una ventaja: el proceso orientado a objetos lleva a una modelización lo más fiel posible de la realidad, con lo que se puede establecer una correspondencia directa entre los componentes de esta fase y los componentes del código. Como Churcher y Shepperd afirman:

> « es posible desarrollar medidas predictivas efectivas basadas en el supuesto de que la estructura de la implementación refleja la estructura del diseño de forma más precisa que en el caso del desarrollo de software siguiendo un método tradicional » [Chur95]

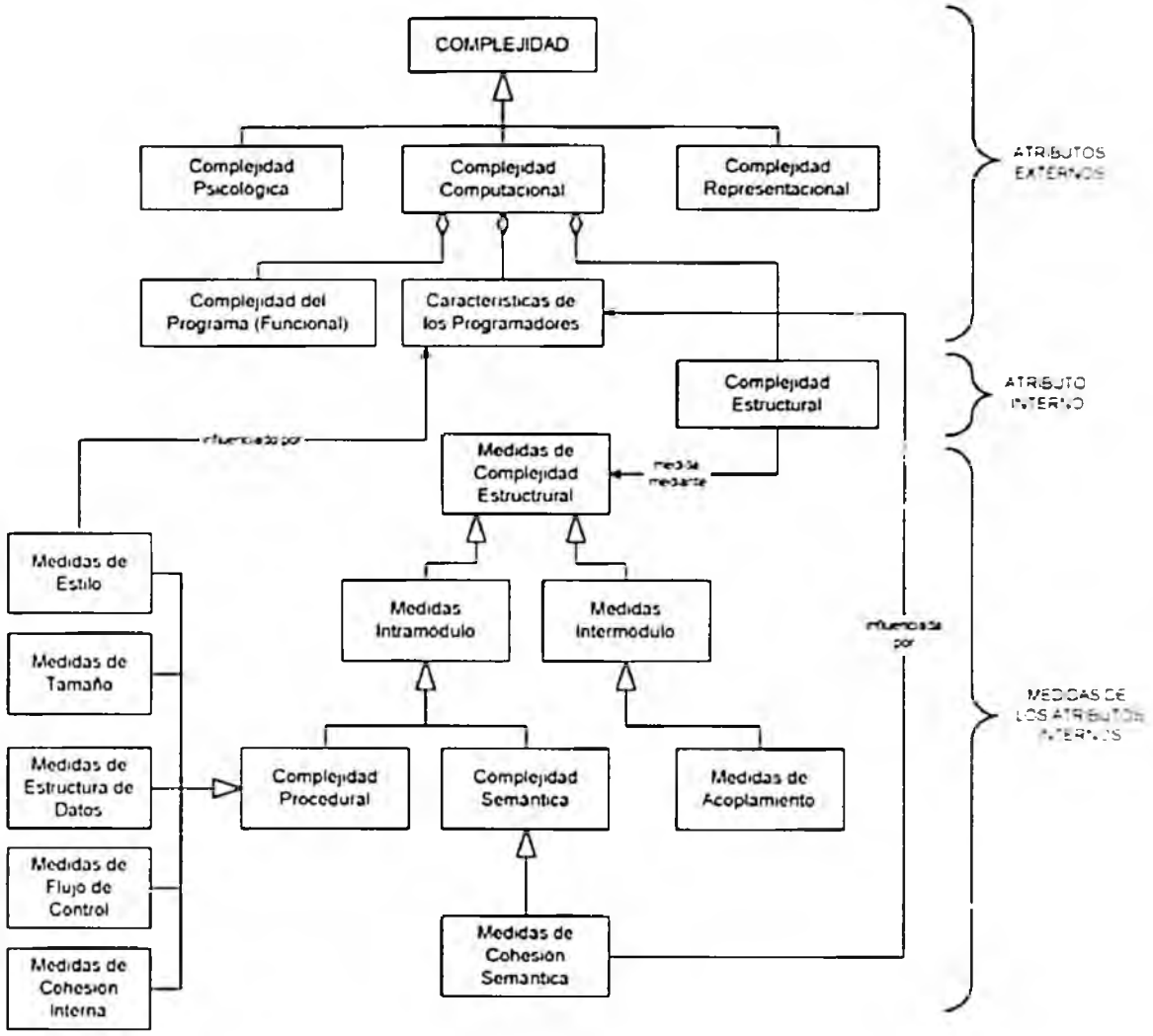

Figura 3.17 Clasificación de la Complejidad del Software.

3.4.2 Modelo de Interconectividad propuesto

Nuestro objetivo es obtener una medida de la complejidad cognitiva o psicológica [Cont86] de la representación de un sistema orientado a objetos en la fase de análisis del dominio, con el objetivo de estimar el esfuerzo de desarrollo, entendiendo por *complejidad cognitiva* la medida de la dificultad que una persona tendría para comprender e implementar la estructura de un diseño concreto [McGr95]. Una definición más precisa es que es una medida en la escala ordinal mediante la que se puede clasificar un conjunto de clases o un conjunto de modelos del dominio. El supuesto básico en el que nos centramos es que la complejidad cognitiva tiene un impacto directo en el tiempo, esfuerzo y por tanto, coste de desarrollo.

El atributo de nuestra medida tiene una manifestación directa: el esfuerzo de desarrollo; por lo tanto, no se genera confusión en la comparación de dos clases, ni de modelos de uno o varios sistemas. La medida se denomina *Interdependencia (ID)* puesto que modeliza la complejidad debido al número y grado de las interdependencias o interconexiones que una clase posee con otras clases del sistema. La *complejidad global* del sistema se calculará como la suma de las complejidades de las clases individuales. Como nuestro uso de la medida de complejidad va a ser a efectos de estimación de esfuerzo y en combinacion con la medida de tamaño anteriormente expuesta, necesitaremos *neutralizar* el efecto del tamaño en nuestra medida de complejidad. Lo haremos dividiendo el valor de la medida obtenido por el número de clases implicadas en el modelo [Whit97].

Nuestro modelo de complejidad se basa en la *interconectividad*, es decir, en las colaboraciones o dependencias mediante las cuales se interconectan varias clases en un sistema. Este modelo capta tanto las propiedades de complejidad *intra-clase* como las *inter-clase* (véase la Figura 3.17 que ilustra la Clasificación de la Complejidad del Software) ya que la complejidad cognitiva de un conjunto de clases es una función de la complejidad cognitiva de las clases individuales y de la complejidad cognitiva de sus interconexiones. En nuestro modelo de medición, *el número de servicios de alto nivel* o *responsabilidades* y el *número de colaboradores* se consideran atributos directos.

Las clases se relacionan entre sí a través de tres tipos de asociaciones: la *agregación*, la *asociación* y la *generalización* [Booc97], al que añadimos la *comunicación por mensajes*. Un modelo preliminar de clases puede mostrar estos tres tipos de relaciones bajo dos tipos de categorías: las *colaboraciones*, que realmente representan relaciones cliente/servidor y la *generalización*. Tanto las relaciones de agregación como las asociaciones se consideran indistintamente *colaboraciones*. En la Figura 3.18 se ilustra una clase con sus relaciones.

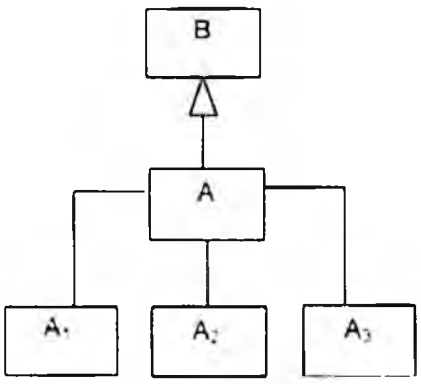

Figura 3.18 Una clase con sus colaboradores.

La clase *A* de la Figura 3.18 colabora con las clases A_1, A_2 y A_3, y las colaboraciones se representan mediante lineas continuas. Por otra parte, la clase *A* hereda características de la clase *B* y se muestra mediante una flecha con origen en la subclase (clase hijo) y destino en la superclase (clase padre).

Este modelo puede proporcionar dos tipos de información sobre las clases que aparecen representadas:

1. Información *externa* a la clase
2. Información *interna* a la clase

La *información externa* proporciona la lista de responsabilidades y la *interna*, la lista de colaboradores. Desde el punto de vista del desarrollador (diseñador y programador) de una clase, tanto la lista de responsabilidades como la de colaboradores son imprescindibles. Sin embargo, si se va a utilizar una clase como un colaborador, sólo es necesario conocer su lista de responsabilidades. Por lo tanto, el desarrollador de la clase A no tiene ningún interés o necesidad de conocer la lista de colaboradores de las clases A_1, A_2 y A_3.

A la vista de lo anteriormente expuesto, en esencia, percibimos dos tipos de complejidad. Un tipo es la *complejidad interna* que el desarrollador intuye y esta complejidad es precisamente el objetivo de nuestra medida. El otro tipo de complejidad es aquella a la que se enfrenta el cliente o el usuario de una clase Denominaremos a este último tipo *complejidad de interfaz*.

Ambos tipos participan en la definición de *ID*, ya que la complejidad cognitiva de una clase toma en consideracion a las complejidades de interfaz de sus colaboradores.

Notación:

En las secciones siguientes, *A*, *B* y *C* representan a distintas clases de un dominio El número de responsabilidades o servicios no heredados de *A* se representa como s_A y el número total de responsabilidades de *A* como S_A. El número total de colaboradores de una clase *A* será c_A. Cuando se considere una única clase, se pueden omitir los subíndices (*S* en vez de S_A). Si se desea representar un conjunto de clases, se utilizará una representación con subíndices. Así, un conjunto de *n* clases se representa mediante la expresión $(F_1, F_2, \ldots, F_n)$. Si una clase *A* sufre algún tipo de cambio, la clase modificada se representa como *A'*.

3.4.2.1 Supuestos

A continuación describimos los supuestos sobre el comportamiento del modelo con respecto a los atributos directos de nuestro modelo de clases, en los que nos basamos: el número de servicios de alto nivel y el de colaboradores o clases asociadas. Estos supuestos se fundamentan en la experiencia y resultan razonables, desde un punto de vista intuitivo. También vamos a proporcionar una justificacion racional para cada supuesto.

- Supuesto 1: El *desarrollo* de una clase que posee un número elevado de servicios *no heredados* y de colaboradores es *más complejo* que una con un número reducido de servicios y colaboradores.

- Supuesto 2: Es *más complejo* utilizar una clase con un número *total* elevado de servicios, heredados o propios, que una clase con pocos servicios.

- Supuesto 3: La complejidad del desarrollo crece de forma *no lineal* a medida que aumenta el número de servicios *no heredados* y de colaboradores.

- Supuesto 4: La complejidad del uso aumenta de forma *no lineal* a medida que se incrementa el número total de servicios, heredados o propios.

- Supuesto 5: Una clase *siempre* puede colaborar *consigo misma*.

A continuación, exponemos las razones en las que se fundamentan los supuestos anteriores. En primer lugar, suponemos que la dificultad de desarrollo de una clase está relacionada con el número de responsabilidades y de colaboradores; es decir, la complejidad de una clase se incrementa a medida que el número de servicios e interconexiones aumenta. El desarrollador de una clase debe implementar los servicios que se enumeran y es fácil ver que la complejidad aumenta a medida que aumenta la lista de servicios. Además, para que una clase interactúe con otras, el desarrollador debe dedicar un esfuerzo adicional a la comprensión de la interfaz de cada uno de esos colaboradores. Así, la complejidad debe aumentar a medida que aumenta el número de colaboradores.

Por otra parte, no hay que implementar todas las responsabilidades de una clase. Algunas, con sus respectivas implementaciones, se heredan de las superclases, donde realmente se proporciona la implementación. Como estas responsabilidades se heredan, el desarrollador no debe proporcionar mecanismos para las mismas. Sin embargo, si no se proporciona implementación alguna desde las superclases, como puede ocurrir con las clases abstractas, estos servicios se consideran dentro de la categoría de no heredados.

En resumen, únicamente debemos considerar aquellas responsabilidades no heredadas, como se indica en el supuesto 1.

Una cuestión fundamental que debemos plantearnos es si una colaboracion realmente incrementa la dificultad del desarrollo. A fin de cuentas, el fundamento de las colaboraciones es que aumentan la reusabilidad y se pueden evitar reimplementaciones. Podríamos llegar a situaciones en las que el supuesto 1 no es cierto. Por ejemplo, supongamos que un desarrollador implementa una clase *A* sin colaboradores y que dispone de todos los mecanismos que necesita Podríamos comparar la clase *A* con otra clase *B*, que cumple con el mismo conjunto de responsabilidades pero que además, solicita los servicios de algunos colaboradores Se podría concluir que el supuesto 1 no se cumple en este caso, ya que *A*, aun teniendo menos colaboradores que *B*, es probablemente una clase más compleja de desarrollar.

Pero no hay que olvidar el otro aspecto de la situacion anterior: aunque la clase *A* tiene un número reducido de colaboradores, proporciona una elevada funcionalidad que se reflejará en la existencia de una gran colección de servicios no heredados. Por lo tanto, la clase *A* puede que no tenga el mismo número de responsabilidades que *B*: es posible que tenga un número mucho mayor Como el supuesto 1 hace recaer la dificultad del desarrollo tanto en las responsabilidades como en los colaboradores, este supuesto se cumple.

Sin embargo, no es un supuesto muy concreto en cuanto a la cantidad real de incremento de complejidad Supongamos que extendemos una clase *A* con un número de servicios no heredados *s* y colaboradores *c*, para incluir más servicios no heredados y colaboradores. No está claro si el incremento de complejidad es dependiente o independiente de los valores de *s* y *c*. El supuesto 3 intenta clarificar este aspecto.

Este supuesto afirma que la relacion entre la variable dependiente, la complejidad, y las variables independientes, el número de responsabilidades y colaboradores, es no lineal. Una responsabilidad es una descripcion abstracta de una parte de la interfaz de una clase que en la fase de diseño puede tener correspondencia con un grupo de métodos Por lo tanto, es poco probable que un incremento de la complejidad tenga una relacion lineal con respecto al numero de responsabilidades Además, un conjunto numeroso de colaboradores generalmente es indicador de que la clase con la que colaboran necesita de un gran numero de asociaciones e interacciones para cumplir sus objetivos Todo ello contribuye a la complejidad de una manera no lineal.

Los supuestos 2 y 4 están dedicados a la dificultad de las colaboraciones entre clases debido a la complejidad de la interfaz de cada colaborador. Como ya se ha mencionado previamente, el desarrollador de una clase debe comprender y utilizar las interfaces de cada uno de los colaboradores. Toda clase y por tanto, todo colaborador, posee un modelo de estados conceptual asociado con su comportamiento, el cual debe ser perfectamente comprendido por parte de todos sus clientes.

El supuesto 2 implica que es más fácil comprender la interfaz de una clase con un número total pequeño de responsabilidades que una clase con un número elevado. El supuesto 4 aporta más detalles al afirmar que es más fácil comprender la interfaz de varias clases pequeñas que la interfaz de una única clase cuya interfaz sea la combinación de las pequeñas. El supuesto 4 fomenta las colaboraciones con clases más pequeñas porque creemos que existe una relación no lineal de incremento de la complejidad de la intefaz a medida que se aumenta el número total de responsabilidades de una clase.

El último supuesto, el 5, afirma que una clase puede colaborar consigo misma. Es bastante obvio debido al hecho de que cualquier método de una clase puede llamarse a sí mismo de forma recursiva o puede llamar a cualquier otro método. También creemos que la complejidad cognitiva de una clase *A* debe incluir a la complejidad externa de *A*; el desarrollador de *A* necesita comprender la propia interfaz de *A*.

3.4.3 Medida de Interdependencia

En esta sección exponemos la formulación completa de la medida de complejidad cognitiva. La medida se denomina *Interdependencia (ID)* para denotar que mide la interconectividad que una clase alcanza con sus colaboradores. La formulación se basa en los supuestos anteriormente expuestos y considera dos términos: el primer término representa el *número de colaboraciones* y el segundo representa el *grado* de las mismas. La expresion de cálculo de *ID* es la suma de ambos términos como se indica a continuación:

ID = Número de Colaboraciones + Grado de Colaboraciones

Para facilitar la exposición, vamos a llamar *T1* al número de colaboraciones y *T2* al grado de las mismas. Para una clase dada *A*, la medida será:

$ID(A) = T1(A) + T2(A)$

Para que la medida sea efectiva, se debe calcular cada uno de los términos Entendemos que el número de colaboraciones es el número maximo de veces que se cumple una colaboración con cada uno de los colaboradores, tal y como se indica en la representación. Este número máximo proporciona una indicacion clara de la complejidad potencial que una clase puede alcanzar. Pero como en estas fases iniciales no se dispone de esta información, asumimos que cada colaborador se utiliza en los mecanismos que existen bajo cada responsabilidad que debe ser implementada. Por tanto, *T1* se calcula como el producto del número de colaboradores *(1 + C)* y el número de responsabilidades no heredadas *s*.

$$T1(A) = s\,(1 + C)$$

Considerando el supuesto 3, éste indica una dependencia no lineal entre la complejidad y los valores de *s* y *C*. La expresión de *T1* satisface la no linealidad de dicha relación. El número de colaboradores de la expresión es *(1 + C)* en vez de únicamente *C* porque, aplicando el supuesto 5, una clase puede colaborar consigo misma.

El segundo término es el grado de las colaboraciones que representa la dificultad total asociada a la compresión de las interfaces de cada colaborador. Tal y como se ha definido previamente, se utiliza el término *complejidad de la interfaz* para representar la dificultad de cada uno de los colaboradores. Se calcula el grado de las colaboraciones como la suma de las complejidades de interfaz de los colaboradores individuales.

Podríamos calcular la complejidad de la interfaz como *S*, el número total de responsabilidades. Sin embargo, esta forma de hacerlo contradice el supuesto 4, que asume una relación no lineal entre *S* y la complejidad de la interfaz. Por este motivo, vamos a medir la complejidad de la interfaz como *S* elevado a una constante *m*. El problema que se plantea es elegir un valor para *m*. Vamos a utilizar $m = 2$ de manera que la complejidad de la interfaz de una clase viene dada por la expresion S^2. Esta eleccion viene apoyada por la literatura, ya que existen varias ocasiones en las que un término ha sido elevado al cuadrado para preservar el tipo de relación que existe aquí entre la complejidad de la interfaz y *S* (Henry *et al* [Henr81] lo hicieron con su metrica basandose en los estudios de Brooks [Broo75] y Belady [Bela79a]; Card *et al* tambien han utilizado este recurso en sus medidas de complejidad [Card88]). Vamos a llamar a la complejidad de la interfaz calculada de este modo *tamaño de interfaz*.

Por lo tanto, el grado de las colaboraciones se calcula como la suma de los tamaños de interfaz de los colaboradores individuales y el tamaño de la interfaz de la clase concreta en estudio (debido al supuesto 5 en el que una clase puede colaborar

consigo misma). Si una clase *A* con un número total de responsabilidades *S*, colabora con *c* clases distintas $A_1, A_2, \ldots, A_c$, el término *T2* tendrá la expresión:

$$T2(A) = S^2 + \sum_{i=1}^{c} TI(A_i)$$

donde *TI(A)* representa al tamaño de la interfaz de la clase *A*, que se calcula a su vez mediante la expresión:

$$TI(A_i) = S^2_{A_i}$$

Por tanto, la expresión completa de *ID* será:

$$ID(A) = s(1+c) + S^2 + \sum_{i=1}^{c} S^2_{A_i}$$

Para que el sumatorio sea válido se debe demostrar que el número y tipo de colaboraciones afecta a la complejidad cognitiva. Los dos términos de la expresion estan diseñados para cubrir situaciones en las que un diseñador debe elegir decrementar el número de colaboraciones con vistas a reducir la complejidad de la medida. Se puede realizar en teoría utilizando una colaboración con un *colaborador universal*, que se define como una clase que contiene todos los servicios que cualquier otra clase pueda necesitar. Sin embargo, dicho colaborador universal haciendo mérito a su nombre, deberá tener una interfaz inmensa y en consecuencia, el segundo término de la expresión aumentará, incrementando la complejidad y reflejando esta situación. Conclusión: el sumatorio consigue un equilibrio en la complejidad.

Cumpliendo con el supuesto 1, esta medida asocia valores de complejidad mayores a las clases que tiene un número elevado de responsabilidades y colaboradores. Esta conclusión está expuesta de una forma explícita en la expresión de la medida. Sin embargo, existen otras propiedades implícitas que se investigan en la siguiente sección.

3.4.3.1 Cálculo de ID para un conjunto de clases

La medida *ID* se puede extender para calcular la complejidad total de un conjunto de clases que representan un sistema completo. El valor total se obtiene sumando los valores de *ID* de las clases individuales.

Por tanto, si el modelo de un sistema *M* está compuesto de las clases A_1, A_2,..., A_n, la complejidad de *M* será la siguiente:

$$ID(M) = \sum_{i=1}^{n} ID(A_i)$$

Si se desea calcular la complejidad de una parte o fragmento de un sistema (por ejemplo, un subsistema) para poder decidir si es adecuado o no en función de su complejidad, es fundamental considerar que la complejidad total del sistema es igual a la suma de la complejidad de la parte en estudio y de la complejidad del resto del sistema; este último sumando depende de cómo interactúa el fragmento con el resto del sistema.

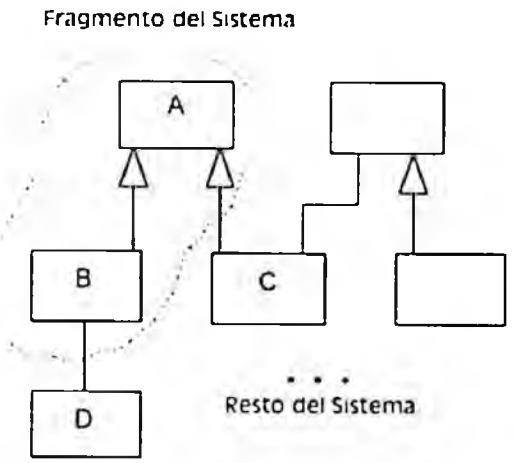

Figura 3.19 Complejidad de un fragmento del sistema.

Como se ilustra en la Figura 3.19, las clases *A* y *B* forman parte de un fragmento del sistema. Las demás clases, como *C*, *D*, etc. interactúan con las clases *A* y *B*. La clase *C* hereda propiedades de la clase *A* y la clase *D* colabora con la clase *B*. Obviamente, si se considera un fragmento diferente del sistema, las interacciones con el resto del sistema serán distintas también. Por lo tanto, siempre que se evalúe la complejidad de una parte o subsistema, se debe tener conciencia de la influencia del resto del sistema sobre dicha parte. Los valores de *ID* para las clases del fragmento a considerar se pueden utilizar como una medida predictiva de complejidad, con el fin de comparar soluciones distintas para un subsistema y poder elegir aquel que posea un complejidad menor.

Por último, nos queda señalar que el uso que vamos a hacer de esta medida es la version en la que se neutraliza el número de clases que participan en el cálculo.

obteniendo de esta manera la *medida de interdependencia media* de las clases del sistema objeto de estudio.

Por tanto, si el modelo de un sistema *M* está compuesto de las clases A_1, A_2,..., A_n, la complejidad media de las clases de *M* será la siguiente:

$$ID_{media}(M) = \frac{\sum_{i=1}^{n} ID(A_i)}{n}$$

3.4.3.2 Ejemplo de cálculo de ID

Para clarificar el cálculo de la medida *ID*, a continuación se muestra el ejemplo relacionado con la Figura 3.20, resultado de ampliar los detalles de Figura 3.19.

En la Figura 3.20 los símbolos s_n representan las responsabilidades o servicios de cada clase. Se ignora el contenido preciso de cada servicio, debido a que *ID* no toma en consideración el texto. Nótese que las clases *A* y B_1 comparten una misma responsabilidad: la s_1. Aparece duplicada en el modelo para representar que el servicio definido en la clase padre B_1 ha sido redefinido en la clase *A* y por tanto, se debe reimplementar.

El número total de responsabilidades (tanto las no heredadas, *s*, como las totales, *S*) de las clases A_1, A_2 y A_3 son 2, 2 y 3 respectivamente, y para las clases padre B_1 y B_2 son 2 y 1 respectivamente. Sin embargo, para la clase *A*, las no heredadas son 2, mientras que el número total de responsabilidades es 4: una que se hereda de B_1 y otra que se hereda de B_2. El número de colaboradores de la clase *A* es de 3.

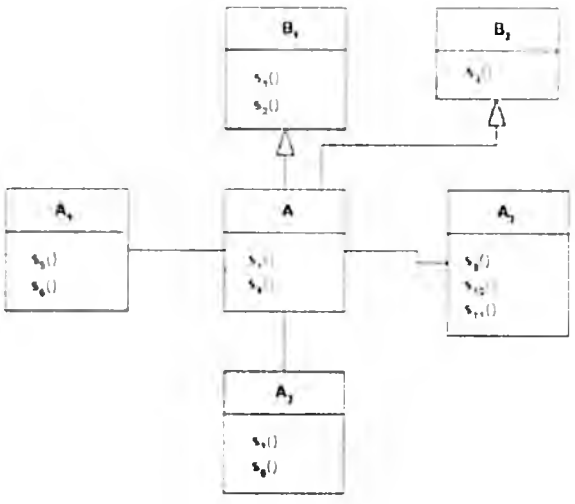

Figura 3.20 Ejemplo de Cálculo de la medida ID.

Una vez se han obtenido los valores de los atributos directos relevantes, el cálculo de la medida es el que sigue:

$$T1(A) = s(1 + c) = 2 \cdot (4) = 8$$

$$T2(A) = TI(A) + TI(A_1) + TI(A_2) + TI(A_3)$$

$$= 4^2 + 2^2 + 2^2 + 3^2 = 30$$

$$ID(A) = T1(A) + T2(A) = 8 + 30 = 38$$

Por lo tanto, la clase *A* tiene una complejidad de 38 según se ha medido con *ID*. Como puede observarse, *ID* mide la complejidad en forma de números enteros. Es una medida ordinal debido a que, para un rango dado, existe un número finito de valores de complejidad.

3.4.3.3 Especificación Formal de la Medida

Muchas de las medidas que se encuentran en la literatura no se han especificado formalmente y la consecuencia principal es el hecho de que existen constantes ambigüedades sobre su uso, lo que lleva a los usuarios de estas medidas a escoger a veces opciones poco acertadas. El ejemplo más típico de esta situación es el de las líneas de código, como ya se ha mencionado anteriormente. No existe una manera estándar de tomar esta medida, por lo que personas distintas la calculan de formas diferentes. Como consecuencia de esta situación, se puede decir que no existen resultados de validaciones concluyentes sobre la efectividad de las LOC. Nuestra medida se ha descrito con una ecuación matemática, pero podrían quedar cuestiones pendientes que sólo se pueden resolver mediante una rigurosa especificación del documento de software que se utiliza como modelo del dominio y la definición asociada al modelo de la medida.

La siguiente especificación algebraica está basada en el modelo formal de propiedades de un sistema orientado a objetos que se ha descrito en la sección 3.2. Pretende ser una referencia para cuando se planteen dudas en el cálculo de la medida. En primer lugar, vamos a especificar el documento de software, el modelo del dominio, definiendo las categorías (entidades) *EstadoDiseño* (sinónimo de *sistema*) y *Clase*. Las definiciones son las siguientes:

EstadoDiseño = conjunto de *Clase*

Clase = <*nombre, super_clases, colaboradores, servicios*>

super_clases = conjunto de *nombre*

colaboradores = conjunto de *nombre*

La entidad *EstadoDiseño* que representa a un conjunto de clases, se define como un conjunto de elementos de la categoría *Clase*. La clase en sí, es una tupla de *nombre*, *superclases*, *colaboradores* y *servicios*. La representación se ajusta a la información de la que disponemos en un modelo inicial del dominio. El elemento *nombre* está formado por un conjunto de caracteres y representa al nombre de la clase (no nos interesa el formato de identificación completo, *nombre:n*, puesto que no manipulamos instancias de clase, objetos). Los elementos *superclases* y *colaboradores* representan a un conjunto de nombres de clases, mientras que el elemento *servicio* representa a un conjunto de elementos formados por cadenas de caracteres.

Nótese que bajo el nombre *super_clases* aparecen aquellas clases que tiene una relación de generalización con una clase dada y que, a efectos de nuestra medida, no necesitamos distinguir entre los diferentes tipos de relación que las clases tienen entre sí; es decir, consideramos de la misma forma una relación de agregación, una asociación o una comunicación de mensajes, es decir, cualquier relacion de dependencia cliente/servidor. Todas las clases que tienen relaciones del tipo anterior con otra dada, se consideran dentro del conjunto de *colaboradores*.[4]

A continuación se definen las operaciones que permiten construir una clase o el modelo de un sistema completo, basándose en operaciones de construcción básicas. Las operaciones se especifican proporcionando las categorías a las que pertenecen los operandos y el resultado de la operación. Algunas operaciones se consideran *operaciones externas* y detectan errores debido al uso de parámetros erróneos. Las operaciones de construcción externas son las siguientes:

añadir_clase: *estadodiseño* x *clase* ⇒ *estadodiseño* ∪ {Error}

añadir_super_clase: *estadodiseño* x *clase* x *clase* ⇒ *clase* ∪ {Error}

añadir_colaborador: *estadodiseño* x *clase* x *clase* ⇒ *clase* ∪ {Error}

añadir_servicio: *clase* x *servicio* ⇒ *clase* ∪ {Error}

eliminar_clase: *estadodiseño* x *clase* ⇒ *estadodiseño* ∪ {Error}

---

[4] Estas puntualizaciones afectan a las operaciones definidas en la sección 3.2

eliminar_super_clase: *estadodiseño* x *clase* x *clase* ⇒ *clase* ∪ {Error}

eliminar_colaborador: *estadodiseño* x *clase* x *clase* ⇒ *clase* ∪ {Error}

eliminar_servicio: *clase* x *servicio* ⇒ *clase* ∪ {Error}

La operación *añadir_clase* sirve para añadir una clase al modelo de un sistema, y por tanto, es una operación de construcción del mismo. Las operaciones de construcción de una clase seran *añadir_super_clase*, *añadir_colaborador* y *anadir servicio*. El significado de las operaciones es el que el nombre de las mismas sugiere. La definición formal de las operaciones se describe en el Apéndice A.

3.4.4 Comportamiento de la Medida

Con cierta frecuencia, durante el desarrollo de un sistema utilizando el paradigma orientado a objetos, los ingenieros de software deben encarar el siguiente dilema: elegir entre un conjunto de opciones de partición de un subsistema en sus componentes, siendo estas opciones alternativas y conflictivas. En esta sección, vamos a analizar el comportamiento de la medida en estas circunstancias. Vamos a investigar una serie de situaciones en las que se consideran dos descomposiciones alternativas y en conflicto, y donde el ingeniero es el que debe decidir. Las situaciones son suficientemente genéricas como para poder extraer conclusiones de ellas. El propósito de este análisis es validar la medida de forma teórica.

3.4.4.1 Comportamiento con respecto al Tamaño

Un aspecto de suma importancia que debemos investigar en cualquier medida basada en el concepto de clase es la respuesta de dicha medida con respecto al tamaño. Los expertos en orientación a objetos no recomiendan en absoluto, el uso de clases grandes; las que tienen un número excepcionalmente grande de responsabilidades se consideran resultado de un mal diseño. Es más, generalmente las clases grandes se descomponen en aquellos componentes que puedan ser potencialmente reutilizables

También es necesario indicar que un tamaño de clase excesivamente pequeño tampoco es recomendable. A largo plazo, puede llegar a ser pernicioso; considérese, por ejemplo, el coste de mantenimiento de un gran número de clases muy pequeñas. De todos modos, este problema de clases pequeñas no se suele dar con frecuencia. Es más, el problema real reside en una mala tendencia o hábito de los

diseñadores que no descomponen un concepto lo suficiente, siendo el resultado final un conjunto de clases de tamaño respetable que no son reutilizables.

Por tanto, la medida de la complejidad debe servir para juzgar si una clase tiene el tamaño óptimo. A continuación, se exponen dos situaciones en las que se proponen opciones que afectan al tamaño de las clases resultantes. En la primera situación, partimos una clase en dos clases más pequeñas e investigamos lo que la medida tiene que decir al respecto. En la segunda, se combinan dos clases que tienen elementos comunes y se observa el comportamiento de la medida.

Vamos a representar una clase A que posee un conjunto de responsabilidades S y un conjunto de colaboradores C. Definimos la partición como el proceso de dividir A en un conjunto de n clases $A_1, ...A_n$, con conjuntos de responsabilidades $S_1,...S_n$ y conjuntos de colaboradores $C_1,...C_n$, tal que $S_1 \cup ... \cup S_n = S$ y $S_1 \cap ... \cap S_n = \emptyset$ y $C_1 \cup ... \cup C_n = C$ y $C_1 \cap ... \cap C_n = \emptyset$.

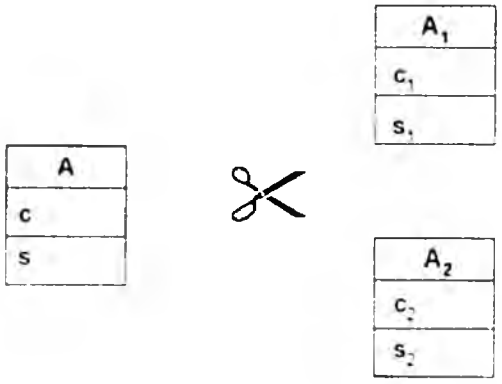

Figura 3.21 Partición de una clase.

Partición de clases

Una clase con una cohesión débil generalmente se puede fragmentar en clases más pequeñas y cohesivas. Vamos a probar que *ID* proporciona una complejidad menor para la combinación de las clases resultantes de la partición. Una clase A con s servicios o responsabilidades[5] y c colaboradores (Figura 3.21) se divide en dos clases A_1 y A_2 con sus responsabilidades s_1 y s_2 y colaboradores, c_1 y c_2. Como las clases son resultado de una partición, se cumple que $s = s_1 + s_2$ y $c = c_1 + c_2$.

[5] En esta sección, la clase A no tiene superclases, por lo tanto, no hereda y $s = S$

Para determinar los valores de *ID* en ambos casos, vamos a considerar los valores de *T1* y *T2* de forma individual, tanto en el caso dividido como en el compuesto.

$$T1(A) = (1 + C)\,s$$

$$= (C_1 + C_2 + 1)(s_1 + s_2)$$

$$= (C_1 + 1)\,s_1 + (C_2 + 1)\,s_2 + C_2\,s_1 + C_2\,s_2$$

$$> T1(A_1) + T1(A_2)$$

Vamos a considerar que $\{B_1,\ldots, B_c\}$ es el conjunto de colaboradores de A, que se divide entre las clases A_1 y A_2, formándose los conjuntos $\{B_1,\ldots, B_{c1}\}$ y $\{B_{c1+1},\ldots, B_c\}$, respectivamente. Se darán las siguientes relaciones entre los términos *T2* del caso dividido y del caso compuesto:

$$T2(A) = \sum_{i=1}^{c} TI(B_i) + s^2$$

$$= \sum_{i=1}^{c_1} TI(B_i) + \sum_{k=c_1+1}^{c_2} TI(B_k) + (s_1 + s_2)^2$$

$$= T2(A_1) + T2(A_2) + 2s_1s_2$$

$$> T2(A_1) + T2(A_2)$$

Por tanto, $ID(A) > ID(A_1) + ID(A_2)$ y la medida proporciona una complejidad combinada menor para las clases divididas. Generalmente, los expertos suelen fomentar la partición, siempre que haya alguna abstracción conceptual reutilizable y útil asociada a las clases A_1 y A_2 que resultan de tal partición.

Como ya se ha mencionado anteriormente, desde el punto de vista de esta medida, no se puede considerar la complejidad de una parte de un sistema de manera aislada al resto. Es decir, si un fragmento tiene una complejidad pequeña, las colaboraciones que induce con el resto del sistema pueden ser numerosas. En la situación anterior se ha visto que la complejidad de las clases A_1 y A_2 es menor pero queda la duda de saber cómo colaboran estas clases con otras en el sistema.

Para completar la exposición, supongamos la existencia de una clase B que colabora con la clase A. Debido a esta colaboración, la complejidad de B aumenta en $\Delta = s_B + s_A^2$. En cambio, si B colabora con A_1 y A_2, la complejidad aumenta en $\Delta' = 2s_B + s_1^2 + s_2^2$. Así que dependiendo de la diferencia relativa entre s_B y $2s_1\,s_2$, la

complejidad de B será menor o mayor en el caso dividido. En resumen, el incremento relativo de la complejidad depende de si el aumento de complejidad, debido a un colaborador adicional, es mayor o menor que la reducción en la complejidad de la colaboración debido a colaboradores menores. Es probable que $s_B < 2S_1$, S_2 y por tanto, $\Delta' < \Delta$ con lo que se puede prever una complejidad más favorable para el caso dividido.

Combinación directa de clases

A continuación, vamos a investigar qué efecto tiene sobre la complejidad la combinación en una sola clase de dos que tienen elementos comunes en términos de colaboradores. En la Figura 3.22 se muestran dos clases A_1 y A_2 con responsabilidades s_1 y s_2, y un conjunto común de colaboradores $C = \{B_1, \ldots, B_c\}$, donde $c = |C|$.

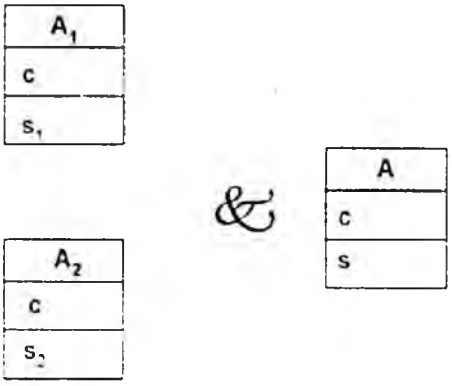

Figura 3.22 Combinación de dos clases.

Supongamos que la complejidad de la colaboración con los c colaboradores viene representada por la siguiente expresión:

$$\sum_{i=1}^{c} TI(B_i)$$

que es una constante para todas las clases de este ejemplo, constante representada con el símbolo δ.

Supongamos que combinamos ambas clases en una nueva clase A, tal que el conjunto de responsabilidades de A sea la unión del conjunto de responsabilidades de A_1 y A_2, y que no existen responsabilidades comunes entre A_1 y A_2. En estas circunstancias, el número de responsabilidades de A es $s = s_1 + s_2$. Para analizar la complejidad relativa de la situación original y del caso combinado, vamos a

considerar los términos *T1* y *T2*. La expresión de *T1* que relaciona a las tres clases *A*, A_1 y A_2 es la siguiente:

$$T1(A) = (C + 1)\, S$$

$$= (C + 1)(S_1 + S_2)$$

$$= (C + 1)\, S_1 + (C + 1)\, S_2$$

$$= T1(A_1) + T1(A_2)$$

Como se puede observar, el término *T1* se mantiene constante para el caso original (la clase *A*), así como para el caso combinado (clases A_1 y A_2). Por tanto, la complejidad relativa se debe juzgar únicamente en términos de *T2*, tal y como se expone a continuación.

$$T2(A) = \delta + S^2$$

$$= \delta + (S_1 + S_2)^2$$

$$T2(A_1) + T2(A_2) = 2\delta + S_1^2 + S_2^2$$

Si $\delta > 2S_1 S_2$, entonces $T2(A) < T2(A_1) + T2(A_2)$ y por tanto, la complejidad del caso combinado es menor. Para comprender esto mejor, reexaminemos la situación. Si las clases A_1 y A_2 deben colaborar con los *C* colaboradores, la complejidad de la colaboración con el mismo conjunto de colaboradores se da en dos lugares distintos. Por tanto, el hecho de combinar las clases reduce la complejidad en un factor δ. Sin embargo, al combinar las responsabilidades de la clase *A*, también hemos provocado un aumento de la complejidad en un factor $2S_1S_2$. Por tanto, dependiendo de cuál de los factores es mayor, la medida fomenta o desaprueba la combinación de clases. De todos modos, como hemos mencionado previamente, es necesario evaluar tanto la situación combinada como la dividida con respecto al diseño completo antes de asumir cuál es la mejor.

Adición de una clase

Supongamos que disponemos de un modelo de un sistema *M* y que nuestra intención es añadir una clase *A*, sin ninguna colaboración. El modelo *M* tendrá un valor de la medida que será *ID(M)* y la clase *A* tendrá su respectivo valor *ID(A)*. La adición de *A* al modelo *M* generará como modelo resultante *M'*, y como nuestra medida se construye de forma acumulativa para los modelos, la expresión de la medida para *M'* será la siguiente:

$ID(M') = ID(M) + ID(A)$

Por lo tanto, la adición de una clase hace que aumente la complejidad final del modelo. Si la clase *A* se añade para que sea una colaboradora de alguna clase de *M*, la complejidad del modelo final *M'* se incrementará al aumentar el número de colaboradores, como comentamos a continuación.

Adición de un colaborador

Supongamos una clase *A*, cuyo número de servicios propios y colaboradores es s_A y c_A, respectivamente, y el número de servicios total es S_A. Vamos a analizar que ocurre con la medida si añadimos un colaborador *B*, cuyo tamaño de interfaz es *TI(B)*.

Como consecuencia de la adición de una colaboración, podemos prever un aumento de la complejidad y además, también se entiende el incremento de la complejidad desde el punto de vista de un desarrollador, puesto que para llevar a cabo la colaboración, el desarrollador deberá comprender la interfaz del nuevo colaborador *B* y puede que tenga que implementar nuevas funcionalidades necesarias para la colaboración. La expresión de la complejidad para *A*, será la siguiente:

$$ID(A) = s_A(1 + c_A) + TI(A) + \sum_{i=1}^{c} TI(A_i)$$

Al añadir un colaborador, la complejidad de *A* cambiará, siendo la expresión:

$$ID(A) = s_A(1 + 1 + c_A) + TI(A) + \sum_{i=1}^{c} TI(A_i) + TI(B)$$

Por lo tanto, nuestra medida valora el incremento de la complejidad de la adición de una colaboración entre *A* y *B* en un factor s_A + *TI(B)*.

Adición de un servicio

La adición de un servicio a una clase *A* hará que aumente la complejidad, ya que hay que implementar dicho servicio. La complejidad de la clase *A* será la siguiente:

$$ID(A) = s_A(1 + c_A) + S_A + \sum_{i=1}^{c} TI(A_i)$$

Al añadir un servicio, la expresión de la complejidad será la siguiente:

$$IID(.1) = (s_{.1} + 1)(1 + c_{.1}) + (S_1^{\cdot} + 1) + \sum_{i=1}^{c} IID(.1_i)$$

Como podemos observar, la complejidad se ha incrementado en $(C_{.1} + 1) + 1 + 2S_{.1}$.

3.4.4.2 Comportamiento con respecto a la Generalización

En los sistemas orientados a objetos, la relación de generalización se utiliza principalmente para modelizar relaciones de supertipo/subtipo, utilizándose el mecanismo de herencia[6], en las que las clases hijo heredan todas las responsabilidades de las clases padres. Puede haber necesidad de redefinir y por tanto, reimplementar algunas de las responsabilidades de los padres en las clases hijos. El mecanismo de herencia es de gran importancia para reducir la complejidad de los sistemas orientados a objetos, haciéndolos menos susceptibles a los cambios y en consecuencia, más faciles de mantener. Por este motivo, es de suma importancia comprender el comportamiento de la medida en las situaciones en las que se utiliza la generalización entre clases.

Basándonos en la Figura 3.23, vamos a analizar la complejidad de un modelo que posee dos clases interrelacionadas. Se puede cambiar la organización de estas dos clases, de manera que hereden de una clase padre abstracta que encapsule las características comunes de ambas.

En general, una clase padre de esta categoría suele proporcionar una visión restrictiva y a menudo de interés, de un conjunto de dos o más clases especializadas. Vamos a suponer que la clase abstracta no dispone de ningún tipo de implementación.

Para poder examinar la influencia de la relación de generalizacion en la medida vamos a considerar el valor que nos proporciona antes y después de añadir dicha relación.

---

[6] Algunos desarrolladores utilizan el mecanismo de herencia aunque no exista una relacion de supertipo subtipo con el fin de reutilizar mecanismos implementados en otras clases (herencia de implementacion). Existe un gran número de detractores dentro de la comunidad de profesionales de la orientacion a objetos con respecto a este tipo de herencia, por este motivo no la consideramos en este trabajo.

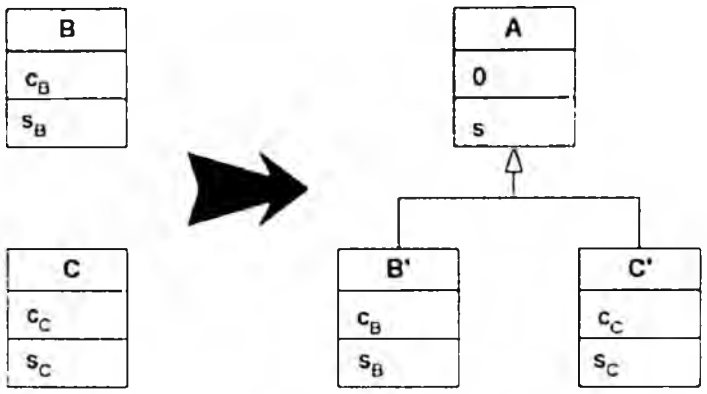

Figura 3.23 Dos clases, antes y después de la adición de la relación de generalización.

Combinación con Generalización

Consideremos dos clases *B* y *C* con sus correspondientes servicios s_B y s_C y colaboradores, c_B y c_C. Supongamos que estas dos clases se agrupan en una pequeña jerarquía de generalización, tal y como se muestra en la Figura 3.23, donde *B'* y *C'* heredan de una clase abstracta *A*. Dicha clase *A* incorpora una serie de servicios que son un subconjunto de los servicios de las clases *B* y *C*. Supongamos que *A* tiene un número *s* de servicios ($s < s_B$ y $s < s_C$), que no tienen ninguna implementación y que el número de colaboradores es 0. Como *A* no proporciona ninguna implementación, los elementos internos de *B'* y *C'* son idénticos a los de *B* y *C*.

Nuestro objetivo en este ejemplo es determinar si la introducción de la clase abstracta beneficia al modelo o no. El valor de *T1* no cambia aunque se aplique la generalización.

$$T1_{BC} = T1(B) + T1(C)$$

$$T1_{ABC} = T1(B') + T1(C') + T1(A)$$

$$= T1(B) + T1(C) + s \cdot 0$$

$$= T1_{BC}$$

Por tanto, $T1_{ABC} = T1_{BC}$. Sin embargo, el término *T2* en el caso de la generalización es mayor, como se muestra a continuación:

$$T2_{BC} = T2(B) + T2(C)$$

$$T2_{ABC} = T2(B') + T2(C') + T2(A)$$

$$T2_{A,B,C} = T2(B) + T2(C) + s^2$$

$$= T2_{B,C} + s^2$$

Por tanto, la complejidad del caso donde se aplica el mecanismo de herencia es mayor que en el que no se usa. Pero de todos modos, debe considerarse que las clases del ejemplo son meros fragmentos de un diseño y no el diseño completo. Por tanto, es necesario considerar la complejidad de las colaboraciones con estas clases.

Consideremos una clase *D* que es parte del resto del sistema y que necesita únicamente el conjunto de servicios de *A*. La colaboración con *A* incrementa la complejidad de *D* en un factor de $\Delta' = s_D + s^2$. Sin embargo, en el caso en el que no hay relación de generalización, *D* tendría que colaborar tanto con *B* como con *C*, incrementándose la complejidad de *D* en un factor $\Delta = 2s_D + s_B^2 + s_C^2$.

De aquí se deduce que la cuestión de la complejidad relativa de los dos casos depende de los valores relativos de s^2 y $\Delta - \Delta'$, que parece tener como valor al menos $s_D + s_B^2$. Como $s_B > s$, la medida *ID* nos lleva a juzgar que el caso con generalización tiene una complejidad global menor que el caso sin ella.

Por último, cabe mencionar que el hecho de que se elija el caso con generalización depende en la existencia de al menos un uso de *A*. Esta idea es bastante intuitiva, ya que la existencia de una clase abstracta que no se utiliza en ningún momento es absurda.

3.4.5 Validación Teórica: Propiedades de la Medida

En esta sección, vamos a validar la medida de forma teórica siguiendo dos fases. En primer lugar, vamos a analizar la respuesta de la medida a las características específicas de todo sistema orientado a objetos. En segundo lugar, definiremos algunas propiedades axiomáticas que cualquier medida de complejidad cognitiva debe satisfacer. Por último, validaremos *ID* contra dichas propiedades.

3.4.5.1 Características Orientadas a Objetos

Los diseños orientados a objetos poseen determinadas propiedades que han demostrado reducir la complejidad del software [Lieb88, Li93]. A continuación, se enumeran dichas características y se explica brevemente el papel que cumple la medida en distintas situaciones.

1. *Herencia*: En el caso de que se deba desarrollar una clase, disponemos de dos alternativas: que herede de una clase ya existente mediante una relación de generalización o crearla partiendo de cero. La medida siempre favorecera la relación de generalización sobre la creación, indicando que esta medida es sensible a este tipo de reusabilidad.

2. *Polimorfismo*: En los programas orientados a objetos una referencia a un tipo abstracto en realidad puede asignarse a cualquier objeto que pertenezca a su subtipo. Esta característica permite que una clase colabore con una sola clase que encapsule todas las propiedades comunes de todo un conjunto de clases, reduciendo el número de colaboradores Debido a que la medida asigna valores bajos de complejidad cuantas menos colaboraciones se den, se fomenta el polimorfismo

3. *Encapsulacion* Mediante la encapsulacion de atributos, una clase consigue restringir su interfaz externa a un mínimo estrictamente necesario Así, una clase tiene absoluta libertad para elegir los mecanismos internos que va a utilizar, siempre que mantenga su interfaz externa Como la encapsulación reduce la interfaz externa, potencialmente puede reducir la lista de responsabilidades. Esta medida siempre favorece la disminución de la lista de responsabilidades.

4. *Agregación*: Cuando un mecanismo interno de una clase utiliza los servicios de otra clase que ya existe a través de la instanciación, a esta situación se le denomina agregación. Este es el segundo tipo de reutilización típico de los programas orientados a objetos (el primero es la reutilización mediante herencia). La reutilización por agregación se suele favorecer porque reduce el número de responsabilidades en la clase colaboradora. Por lo tanto, una clase mejora (desde el punto de vista de la complejidad) si ciertas responsabilidades pertenecen a otra clase.

5. *Acoplamiento Débil*: Se supone que las clases son tan independientes las unas de las otras como el dominio al que representan lo permite. Una interacción innecesaria con otras clases se considera una mala práctica de diseño. Siguiendo esta pauta, *ID* es contraria a los números elevados de colaboraciones. Es cierto que no se puede medir un acoplamiento fuerte únicamente mediante el número de colaboraciones, sino que también debe considerarse la intensidad de la colaboración que se da con cada colaborador. Por ejemplo, si se utilizan todos los métodos de la interfaz de un colaborador, obviamente el acoplamiento es intenso. En la fase de análisis del dominio, no disponemos de información del nivel de los métodos de interfaz. De cualquier manera, un

colaborador con una lista amplia de responsabilidades posee una predisposición al acoplamiento fuerte y la medida es sensible a estas situaciones.

6 *Cohesion Fuerte*: En la sección anterior se ha ilustrado un caso en el que dividir una clase con una cohesión pobre ha tenido como resultado una disminucion en el valor de la complejidad total.

3.4.5.2 Propiedades Axiomáticas

En esta seccion, vamos a definir el conjunto de propiedades que una medida de complejidad valida debe satisfacer obligatoriamente Shepperd *et al* [Shep93a] propusieron que las operaciones de concatenacion (operaciones utilizadas de forma recursiva para construir un sistema) utilizadas en las propiedades deben especificarse formalmente antes de especificar las propiedades en si Nosotros ya hemos proporcionado las especificaciones algebraicas de las operaciones de concatenacion ' y + que se encuentran detalladas en el Apendice A

En las propiedades que se exponen a continuacion, *A*, *B* y *C* representan clases y la complejidad cognitiva de *A* medida a través de una medida ideal se representa como |*A*|. Las clases pueden concatenarse por medio de una relacion de generalizacion o por medio de una colaboración. Cuando una clase *B* hereda de otra clase *A*, se representa a la subclase resultante como $B \uparrow A$ (Figura 3.23). Cuando una clase *A* colabora con otra clase *B*, la clase que colabora se representa como $A \rightarrow B$. Dos clases se dice que son equivalentes si tienen el mismo conjunto de responsabilidades. El número total de responsabilidades de una clase *A* se representa como *S(A)* y el número de responsabilidades no heredadas como *s(A)*. El tamaño de la interfaz de *A* viene dado por *TI(A)* que es igual a $S^2(A)$.

Las propiedades de Weyuker

A continuacion, detallamos el cumplimiento que nuestra medida hace con respecto a las propiedades propuestas por Elaine Weyuker, que se encuentran detalladas en la sección 2.5.3.

- Propiedad W1: (∃ A) (∃ B) (|A| ≠ |B|).

  Existen dos clases A y B tal que la complejidad de A no es igual a la complejidad de B. Esta propiedad requiere que la medida no suministre el mismo valor de complejidad a todas las clases. Con nuestra medida siempre vamos a tener dos

clases con dos valores distintos de complejidad. De hecho, dada una clase *A* siempre le podemos añadir una nueva responsabilidad y crear una nueva clase *B*.

- Propiedad W2: Sea *c* un número no negativo. Existe un número finito de clases con complejidad de valor *c*.

  Una medida debe ser suficientemente sensible y esta propiedad implica que debe disponer de un amplio rango de valores que pueda asignar. En la formulación de *ID*, los terminos *T1* y *T2* son números enteros por lo tanto solo existen $c+1$ posibilidades de asignarles valores. Ademas teniendo en cuenta que el valor de los terminos es constante, existe un numero finito de combinaciones de valores de *c*, *s* y *S* para una clase dada y de los valores de *S* de sus colaboradores. Por lo tanto, la propiedad se cumple. Aunque es cierto que potencialmente se pueden representar infinitos modelos con los mismos valores para *s*, *c*, *S*, *S*, , tambien es cierto que cualquier herramienta de calculo de métricas impone límites.

- Propiedad W3: Existen dos clases distintas *A* y *B*, tal que A = B

  Una medida que asigna un único valor distinto a cada clase no es una medida. Iría contra el principio de la Teoría de la Medicion que requiere que el numero de objetos que se pueden medir sea mayor que el rango de valores que la medida asigna. Nuestra medida, obviamente, cumple esta propiedad. Dada una clase *A*, si cambiamos el texto de una de sus responsabilidades obtenemos una nueva clase *B* que tiene el mismo valor de complejidad[7]

- Propiedad W4 (A) (B) (A B A B)

  Existen clases A y B, tal que A es equivalente a B, pero la complejidad de A no es igual que la complejidad de B. Nuestra medida cumple esta propiedad porque podemos diseñar dos clases equivalentes que proporcionan exactamente la misma interfaz pero con mecanismos internos distintos (por ejemplo, usando una jerarquía de herencia distinta o un conjunto distinto de colaboradores).

- Propiedad W5a (A) (B) (A < A B)

  Para toda clase *A* y *B*, si *A* colabora con *B*, entonces la complejidad de *A* + *B* es mayor que la complejidad original de *A*. Como consecuencia de la adicion de

[7] El texto de las responsabilidades y el nombre de los colaboradores no afectan al cálculo del valor de la complejidad.

una colaboración, podemos prever un aumento de la complejidad La propiedad original utiliza el operador $\leq$; en nuestro caso, el operador es mas estricto. Tambien se entiende el incremento de la complejidad desde el punto de vista de un desarrollador. Para llevar a cabo la colaboracion, el desarrollador deberá comprender la interfaz del colaborador y puede que tenga que implementar nuevas funcionalidades necesarias para la colaboracion Nuestra medida valora el incremento de la complejidad de la colaboracion $A \rightarrow B$ en un factor $S(A) + TI(B)$

- Propiedad W5b: $(\forall A)(\forall B)(|B \leq |B \circ A|)$.

  Para toda clase A y B, si B hereda de A, entonces la complejidad de $B \circ A$ *es mayor o igual que la complejidad original de B* Debido al mecanismo de herencia, la clase *B* puede haber heredado algunas responsabilidades o servicios de la clase padre *A*, en cuyo caso la complejidad se incrementa Sin embargo, si la clase *B* realiza una redefinicion de todas las responsabilidades heredadas de la clase *A*, la complejidad no se incrementara Con esta medida si las responsabilidades de la clase padre no se redefinen, el valor de la complejidad de $B \circ A$ es mayor que el valor de *B* en un factor $TI(A) + 2S(A) - S(B)$

- Propiedad W6: $(\exists A)(\exists B)(\exists C)(|A| = |B| \cdot |A:C| \neq |B:C|)$

  Existen clases *A*, *B* y *C* tal que la complejidad de *A* es igual a la de *B* pero la complejidad de la concatenación de *A* y *C* no es igual a la complejidad de la concatenacion de *B* y *C*. Aunque *A* y *B* tengan la misma complejidad los atributos individuales de cada clase como el numero de responsabilidades y el numero de colaboradores, pueden ser diferentes Nuestra medida cumple esta propiedad Un ejemplo muy simple es el caso en que la concatenacion es por colaboracion y si $S(A) \neq S(B)$, entonces $|A:C| \neq |B:C|$

- Propiedad W7 Existen clases *A* y *B* tal que si *B* se forma permutando el orden de las responsabilidades o servicios de *A*, entonces $|A| = |B|$

  El orden de las responsabilidades en la especificacion de una clase no tiene ninguna consecuencia y no deberia, bajo ningun concepto modificar la complejidad de una clase. Esta propiedad es perfectamente aplicable a nuestra medida, puesto que *ID* toma en consideracion el *numero* y no la disposicion de los servicios en un modelo.

- Propiedad W8: Si *A* es el resultado de renombrar a *B*, entonces $|A| = |B|$

  El nombre de una clase se utiliza únicamente como medio de identificacion y no deberia tener ninguna relación con el calculo de la complejidad Una medida de complejidad, por tanto, no deberia variar el valor asignado a una

clase, basándose en el nombre de la misma. Como nuestra medida no incluye el nombre de la clase, esta propiedad se cumple.

- Propiedad W9a: (∃ A) (∃ B) (|A| + |B| > |B ↑ A|).

- Propiedad W9b: (∃ A) (∃ B) (| A | + | B | < | B ↑ A |).

  Para todas las clases *A* y *B*, si *B* hereda de *A*, entonces la complejidad de *B* ↑ *A* es unas veces menor y otras mayor que la suma de las complejidades de *A* y *B*. Es decir, no garantizamos que la medida cumpla en todos los casos la propiedad W9 original. Supongamos que el valor de *S(B* ↑ *A)* es S. Su valor variará desde *S(B)* hasta *S(B)* + *S(A)*. Si S = *S(B)*, caso en que todas las responsabilidades heredadas de la clase padre se redefinen, entonces |*B* ↑ *A*| = |*B*|, cumpliéndose la propiedad W9a y W5b.

Las propiedades de Whitmire

Seguidamente, vamos a investigar cuáles son la propiedades propuestas por Whitmire y comentadas en la sección 2.5.5, que nuestra medida cumple.

- C1: No negatividad

  Nuestra medida de la complejidad nunca asigna números negativos.

- C2: Valor nulo

  Esta propiedad señala que la complejidad de un sistema puede ser nula si no hay relaciones entre sus componentes. Nuestra medida no cumple esta propiedad porque no es únicamente una medida inter-clase; también pretende captar la complejidad intra-clase, basada en los servicios. Por ese motivo, aunque tengamos dos fragmentos de modelo que no colaboran entre sí, siempre consideraremos la complejidad interna de cada una de ellos.

- C3: Simetría

  La complejidad de un sistema no depende del convenio elegido para representar las relaciones entre sus elementos. Nuestra medida cumple esta propiedad puesto que es indiferente a la forma en la que se consideren las relaciones.

- C4: No existe una relación fija entre la complejidad de una composición y la combinación de las complejidades de sus componentes.

Nuestra medida se ajusta a esta propiedad, como hemos comentado previamente No tenemos la certeza de la relación existente entre la suma del valor de la medida de los componentes y el valor de la composicion

- C5 Monotonicidad no Decreciente

  Efectivamente, ya hemos probado que la adicion de una clase o un colaborador a un modelo o la adicion de un servicio a una clase no puede hacer decrecer su complejidad. Por lo tanto, nuestra medida cumple esta propiedad

- C6: Estructura Similar, Identica Complejidad

  Si dos clases o dos modelos tienen una estructura similar, basada en su numero de servicios, número de colaboradores y tamaño de interfaz de los colaboradores, la medida les asignará el mismo valor de complejidad

- C7: La complejidad es Independiente del Tamaño

  Nuestra medida es en cierta forma independiente del tamaño, en el sentido que puede dar valores pequeños de complejidad a estructuras con un numero elevado de clases y valores altos a estructuras con un numero pequeño de clases Depende de la combinacion de servicios y colaboradores Pero por otra parte, esta vinculada al tamaño, puesto que para calcular el valor de la medida para un modelo, la expresion es el sumatorio de las complejidades de las clases y cuantas mas clases hay, mayor es la complejidad Por este motivo desde un punto de vista practico, utilizaremos la complejidad media de las clases del modelo.

- C8 Orden Debil

  Nuestra medida es transitiva y completa, es decir podemos utilizarla para clasificar un conjunto de modelos en orden de complejidad

Los Axiomas de la Función y Relación de Confianza Modificada

Como exponiamos en la seccion 2 5 2, Zuse proporciona un metodo diferente de caracterizar una medida orientada a objetos que no cumple con los axiomas de la estructura extensiva, de forma que su tipo de escala sea superior a la ordinal Hemos investigado el comportamiento de nuestra medida con respecto a la Funcion de Confianza modificada $\mu(CUNI(A, B)) \geq \mu(A) + \mu(B) - \mu(CINT(A, B))$, y nuestra medida la cumple Esto es debido a que esta función se basa en la teoria de conjuntos y obviamente, el comportamiento de los servicios y colaboradores de las clases se

ajusta a dicha teoría. Así, la unión de dos clases, siempre nos llevará a combinar sus componentes individuales y a eliminar aquellos servicios y colaboradores que son comunes y que se han contabilizado por duplicado.

Asimismo, hemos investigado el comportamiento de la medida con respecto a los axiomas de la Relación de Confianza Modificada y hemos llegado a la conclusión que todos ellos se cumplen. Esto implica que nuestra medida no es de escala ordinal únicamente, sino que está por encima de esta. Por lo tanto, vamos a poder utilizar métodos estadísticos paramétricos para la validación empírica.

4. ESTIMAN: UN METODO DE ESTIMACION BASADO EN ANALOGIA

«Debido a que el instrumento de medida ha sido construido por el observador ... hemos de recordar que lo que observamos no es la Naturaleza en sí, sino la Naturaleza expuesta a nuestro metodo de observacion »

Heisenberg, 1958

4.1 INTRODUCCION

Como hemos comentado en secciones anteriores, existen diferentes metodos de estimacion y la estrategia que hemos elegido es la de la *Estimacion por Analogia*

Nuestra elección ha estado condicionada a las características que hacen que una solucion basada en sistemas de Inteligencia Artificial sea adecuada, en comparacion con métodos más tradicionales. Dichas caracteristicas son las siguientes [Bisi95]:

1 *Distribución de los Datos*

- *Pocos Datos*: Los proyectos almacenados son proyectos del pasado. Dada la gran variedad de proyectos, nos encontramos con un espacio de los mismos *n*-dimensional (muchas características) y escasamente poblado. Esto hace que la construcción de un modelo de estimación único y general sea poco adecuado.

- *No Uniformidad*: El espacio de proyectos no está cubierto de forma uniforme. Algunas regiones muestran una alta densidad de población mientras que otras están completamente descubiertas. Esto a veces ocurre

debido a las correlaciones entre las características que describen a los proyectos (pe. la característica 'muchos programadores' está relacionada con 'muchas líneas de código'). Otras veces se da porque ciertos tipos de proyectos son poco comunes (pe. los proyectos de gran envergadura son menos frecuentes que los proyectos pequeños).

- *Valores Atípicos*: Existen proyectos que tienen costes completamente alejados de la media. No son proyectos poco comunes y su importancia no debe subestimarse. Los métodos basados en regresiones representan bastante bien el caso 'medio' pero pueden ser poco fiables para los proyectos que no sean demasiado estándar.

2 *Entorno dinámico*

- *Evolución del Software*: Como el proceso de desarrollo del software todavía no ha alcanzado su madurez, está sujeto a cambios como lo ha estado en el pasado. Un buen modelo de estimación debería ser capaz de seguir esta evolución, de la cual son testigos los datos contenidos en el repositorio de proyectos (La capacidad natural de un sistema de IA para aprender encaja muy bien con este requisito).

- *Cambios en los Conductores de Estimación*: A medida que el proceso de desarrollo cambia, algunas características pueden ver incrementada su importancia mientras que la de otras disminuye. Las características analizadas deben revisarse y actualizarse periódicamente (los sistemas de IA mantienen los costes de los cambios a un nivel razonable).

3 *Fiabilidad de los Datos*

- *Datos Subjetivos o Estimados*: Mientras que algunas características se pueden obtener y medir sin error alguno (pe. el lenguaje de programación) otros dependen de la opinión de un experto y están ligados a la subjetividad (pe. la experiencia del analista) y otros se estiman de forma cualitativa (pe. el tamaño estimado en fases iniciales del proyecto). Como este fenómeno es inevitable, se debe controlar explícitamente y el modelo de estimación debería tener la capacidad para soportarlo.

- *Nivel de Detalle*: El nivel de detalle de los atributos disponibles no es homogéneo, unos pueden ser indicadores muy sintéticos, mientras que otros hacen referencia a detalles técnicos. El modelo de estimación debería ser capaz de dar más importancia a aquellas características que engloban más información.

- *Incompletitud Descriptiva*: Como los datos se recogen a través de cuestionarios, algunos de los campos pueden estar en blanco. La existencia de valores omitidos se produce tanto en el proyecto en evaluación como en los proyectos almacenados. El modelo debe ser capaz de trabajar incluso con descripciones parciales.

- *Relevancia de Características*: Cada característica considerada contribuye de forma diferente a la estimación.

- *Valores Simbólicos y Numéricos*: Algunas de las características toman valores de un conjunto predefinido, que puede estar total o parcialmente ordenado. Los valores numéricos pueden ser enteros, reales o pueden tomar valores de rangos completamente arbitrarios.

En general, considerando todos los problemas expuestos anteriormente, este tipo de estrategia de resolución de problemas se presenta como el más adecuado para cumplir los requisitos del contexto de la estimación de recursos.

Además debemos también considerar que estas estrategias han demostrado tener exito en dominios donde las normas están insuficientemente definidas y no hay un modelo teorico solido que encamine la resolucion de problemas. Precisamente, estas dos caracteristicas encajan perfectamente en la realidad del software. Esto es, se manipulan un número elevadísimo de factores (en la literatura aparecen referencias al uso de hasta 74 factores [Wrin87]); estos factores se definen de manera subjetiva y aquellos que parecen estar correctamente definidos, no son lo que parecen e incluso, estan sujetos a colinealidad, complicando todavia mas la elaboración de modelos cuantitativos.

En consecuencia y la vista de todos estos obstáculos, nos hemos decidido por un enfoque basado en el Razonamiento Basado en Casos.

4.2 LA ANALOGIA Y EL RAZONAMIENTO BASADO EN CASOS

El Razonamiento Analógico es una de las herramientas fundamentales de la Resolución de Problemas. Aunque los expertos reconocen que es un método muy util para su aplicación a la estimación de recursos en los proyectos de desarrollo [Vici91, Höst97], la realidad es que existen pocas publicaciones sobre investigaciones empiricas o teóricas que apoyen el uso de esta técnica en nuestro dominio [Mukh92].

La Analogía es el principal método que utilizan, por ejemplo, en la NASA para estimar el tamaño y tiempo de ejecución de los sistemas terrestres de control de satelites y para el desarrollo de bases de conocimiento para los sistemas de simulación de trayectorias orbitales. Sin embargo, la literatura empírica sobre razonamiento analógico para la estimación de recursos en el mundo del software es escasa

En el nivel más general, la resolución de problemas basada en Analogía implica asociar un problema resuelto previamente o una experiencia pasada al problema en curso, de forma que se facilita la búsqueda de una solución aceptable. Al problema a resolver se le denomina el *objetivo* de la analogía y al resuelto se le denomina la *base* o *fuente*. La analogía se forma cuando se percibe una similitud entre la base y el objetivo, similitud que depende del contexto del dominio.

Así, se establece una correspondencia entre la base y el objetivo y se *transfieren* los elementos similares del uno al otro. Esta correspondencia se utiliza para establecer inferencias analógicas que generan información que facilita la resolucion del problema.

Las teorías generales sobre la resolución por Analogía describen marcos para la comprension de los procesos que un experto, usando este tipo de razonamiento, realiza al desarrollar estimaciones. Sin embargo, estas teorías, amplias en alcance y que cubren una gran gama de situaciones, son demasiado generales para nuestros propositos Aún así y todo, necesitamos un marco especifico que nos permita analizar el razonamiento analogico pero con vistas a la automatizacion del modelo

Además, aunque la teoría general de resolución por Analogía debe acomodar la correspondencia de una base cuyos elementos son conceptualmente distintos de los del objetivo (por ejemplo, entre dominios), en general las investigaciones del Razonamiento Basado en Casos se han centrado en situaciones en las que la base y el objetivo pertenecen al mismo dominio y tienen la misma estructura representativa

Así, como consecuencia de la restricción de las bases, de forma que sean casos que se han encontrado previamente de la misma clase de problemas, el marco proporciona una definición más explícita de los procesos cognitivos que podemos esperar encontrar en la estimación de recursos del software En este dominio la representación del caso objetivo, un proyecto de software, es similar a la de los casos previos, que son los proyectos de software pasados.

En el Razonamiento Basado en Casos, el experto que va a resolver el problema, después de crear una representación mental del problema objetivo, recupera de su memoria a largo plazo uno o más episodios que tienen caracteristicas similares

A continuación, estos casos se evalúan y se elige el más apropiado para que sea el caso base. La correspondencia de las características base-hacia-objetivo a menudo se realiza directamente, ya que el grupo de características es común a todos los casos.

La solución que resolvió el problema en el caso base se transfiere al objetivo y seguidamente, se modifica para compensar la incidencia de los elementos análogos cuyos elementos paralelos no se corresponden.

Basándonos en las teorías del Razonamiento Basado en Casos y en la Analogía para la resolución de problemas, en las siguientes secciones proponemos un método que emula el razonamiento y procesos mentales que los expertos realizan para llevar a cabo una estimación de recursos.

4.3 ESTIMAN: ESTIMACION POR ANALOGÍA

Para desarrollar un método de estimación que sea automatizable nos hemos basado en el marco propuesto por Mukhopadhyay *et al.* [Mukh92, Vici90] y por Bisio *et al.* [Bisi95].

Así, usando como modelo teórico los mecanismos del Razonamiento basado en Casos de la resolucion de problemas por Analogía, los procesos computacionales a realizar se clasifican en cinco categorías:

1. Construcción: desarrollo de una representación del problema objetivo.
2. Recuperación: selección y recuperación de uno o varios casos base.
3. Transferencia: uso de la solución del caso o casos base como punto de partida de la solución del problema objetivo.
4. Correspondencia: localización de las diferencias entre el caso base y el problema objetivo.
5. Ajuste: ajuste de la solución inicial sobre la base de las diferencias encontradas.

La utilidad de usar esta estructura teórica como guía es que permite la discriminación de un mecanismo general cognitivo (aunque sea uno adaptado al funcionamiento analógico) desde el conocimiento necesario para aplicar dicho mecanismo a un dominio específico.

El método ESTIMAN implementa los cinco procesos de manera independiente del dominio, estando el conocimiento específico contenido en una base de conocimiento separada. Esta base de conocimiento engloba tres tipos de conocimiento del dominio: representaciones de los *casos* (los proyectos de software), conocimiento para *seleccionar* analogías adecuadas para cada caso (o proyecto) objetivo, y conocimiento para ajustar la estimación, basado en la interacción entre las representaciones de los casos base y el objetivo.

A continuación, describimos los procesos genéricos implementados en nuestro método, en combinación con la estructura de la base de conocimiento.

4.3.1 La implementación de la estrategia de razonamiento en ESTIMAN y el conocimiento del dominio de estimación del software

ESTIMAN utiliza los cinco procesos básicos anteriormente comentados: construcción, recuperación, transferencia, correspondencia y ajuste. Además, se consideran tres clases distintas de información del dominio para estimar el esfuerzo, usando su mecanismo de inferencia: conocimiento sobre los casos, conocimiento para la seleccion de casos y conocimiento para el ajuste de las no-correspondencias

En la Figura 4.1 se ilustra la arquitectura logica de ESTIMAN.

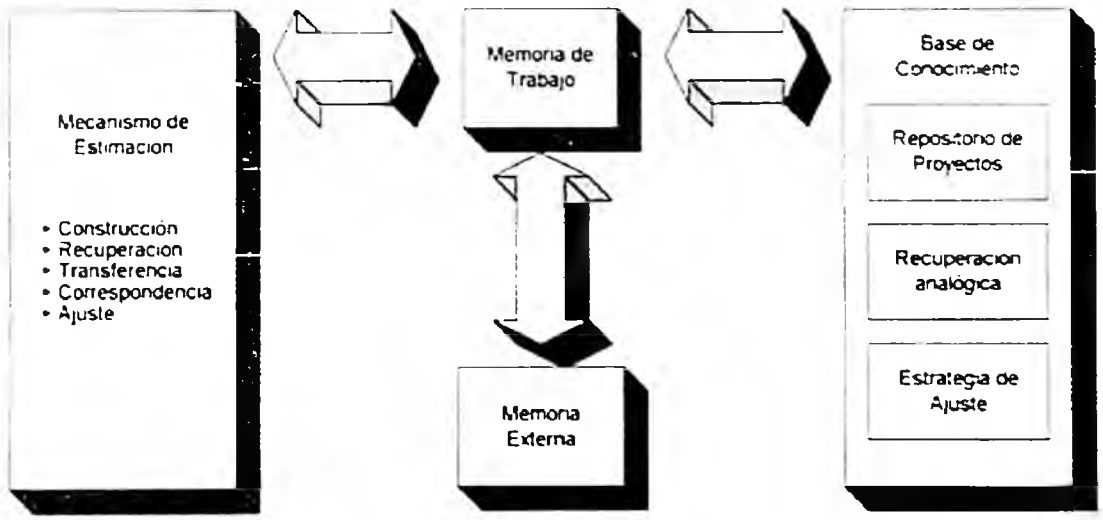

Figura 4.1 Arquitectura lógica de ESTIMAN.

4.3.2 El conocimiento sobre los Casos y el Proceso de Construcción

El *conocimiento sobre los casos* refleja la memoria episódica de proyectos de software pasados, incluyendo la cantidad de esfuerzo real que se necesitó para llevarlos a cabo. Por tanto, la abstracción de proyecto creada para este trabajo engloba la siguiente información:

- Número de clases del software: esta es una información que es real en caso de los proyectos base y se debe disponer de una estimación, en caso del proyecto objetivo, como se indica en la sección 3.3.

- Complejidad media de las clases del dominio: información que se puede obtener en fases iniciales del desarrollo, tanto en los proyectos base como en el objetivo, como se indica en la sección 3.4.

- Características del Contexto del proyecto: consideramos que una forma suficientemente extendida y mencionada en la literatura para representar el contexto de un proyecto es el uso de los cinco factores de escala y los diecisiete conductores de coste del modelo COCOMO [Boeh81, Boeh87, Boeh95, Boeh95a, USC97], enumerados en el Apéndice B.

- Esfuerzo real de desarrollo

El esquema de representación automatizable elegido para el repositorio de casos es una tabla en la que las columnas se corresponden con la información previamente mencionada y cada fila representa un proyecto pasado. Los valores de las veinticuatro características o atributos de los proyectos base, así como del proyecto objetivo, se infieren a través de un formulario y un cuestionario que los expertos del dominio deben cumplimentar. El cuestionario es una modificación del que las organizaciones que participan en el proyecto de revisión continua del método COCOMO rellenan cada vez que aportan información sobre un proyecto terminado [USC97a]. Ambos documentos se describen en el Apéndice C.

Así, el proceso de construcción se utiliza en dos vertientes distintas:

- La creación inicial de la representación interna de los proyectos base, introduciendo todos los atributos en las filas/columnas correspondientes

- La creación de la representación interna del proyecto objetivo, cuyos atributos se introducirán manualmente en una fila específica.

4.3.2 El conocimiento sobre los Casos y el Proceso de Construcción

El *conocimiento sobre los casos* refleja la memoria episódica de proyectos de software pasados, incluyendo la cantidad de esfuerzo real que se necesitó para llevarlos a cabo. Por tanto, la abstracción de proyecto creada para este trabajo engloba la siguiente información:

- Número de clases del software: esta es una información que es real en caso de los proyectos base y se debe disponer de una estimación, en caso del proyecto objetivo, como se indica en la sección 3.3.

- Complejidad media de las clases del dominio: información que se puede obtener en fases iniciales del desarrollo, tanto en los proyectos base como en el objetivo, como se indica en la sección 3.4.

- Características del Contexto del proyecto: consideramos que una forma suficientemente extendida y mencionada en la literatura para representar el contexto de un proyecto es el uso de los cinco factores de escala y los diecisiete conductores de coste del modelo COCOMO [Boeh81, Boeh87, Boeh95, Boeh95a, USC97], enumerados en el Apéndice B.

- Esfuerzo real de desarrollo

El esquema de representación automatizable elegido para el repositorio de casos es una tabla en la que las columnas se corresponden con la información previamente mencionada y cada fila representa un proyecto pasado. Los valores de las veinticuatro características o atributos de los proyectos base, así como del proyecto objetivo, se infieren a través de un formulario y un cuestionario que los expertos del dominio deben cumplimentar. El cuestionario es una modificación del que las organizaciones que participan en el proyecto de revisión continua del método COCOMO rellenan cada vez que aportan información sobre un proyecto terminado [USC97a]. Ambos documentos se describen en el Apéndice C.

Así, el proceso de construcción se utiliza en dos vertientes distintas:

- La creación inicial de la representación interna de los proyectos base introduciendo todos los atributos en las filas/columnas correspondientes.

- La creación de la representación interna del proyecto objetivo, cuyos atributos se introducirán manualmente en una fila específica.

4.3.3 El conocimiento para la Selección de Casos y el Proceso de Recuperación

El conocimiento para la seleccion de casos permite la seleccion de una o varias analogías adecuadas del conjunto de casos pasados. El análisis del proceso de estimacion de los expertos indica que estos usan una combinación de metricas de tamaño que estén disponibles para realizar la selección, pero la verbalizacion de este proceso no proporciona unas indicaciones adecuadas para que pueda ser reconstruido Como la selección y recuperación de casos están intimamente ligados a la organización de la memoria episodica y emplean procesos cognitivos inconscientes, seria dificil, por no decir imposible, que el sujeto-experto proporcione directamente una descripcion precisa del procesamiento cognitivo invocado en esta tarea Así, para recuperar unos casos base adecuados, ESTIMAN utiliza una heuristica de seleccion de casos. Dado un proyecto objetivo, esta heuristica examina cada proyecto del repositorio de casos y selecciona un conjunto de los mismos, basándose en un *indice de similitud* y en un *umbral* η predefinido.

Para realizar estos cálculos, solamente se utilizan dos de las caracteristicas del repositorio de casos: el número de clases y la complejidad media; es decir, el proceso de seleccion se centra en un subconjunto de los atributos de la abstracción de un proyecto, en vez de en la colección completa. (Nosotros hemos elegido estas dos características en el dominio específico que estamos estudiando, basándonos en los resultados obtenidos por Bisio *et al* [Bisi95] que demostraron que la seleccion basada en un subconjunto de caracteristicas es mas precisa que la basada en el conjunto completo De todos modos, nótese que el método es generico, por lo que debe admitir modificaciones tanto en el número de características como en el uso que se haga de las mismas).

El procedimiento es el siguiente

Dado un proyecto objetivo, se debe extraer el conjunto de casos más similares del repositorio de casos. Para ello, se debe realizar un paso previo: se deben normalizar en el intervalo [0..1] los valores de las dos características a considerar, tanto para todos los casos del repositorio como para el proyecto objetivo. En consecuencia, se consigue que cada característica tenga el mismo grado de influencia y el metodo es inmune a las unidades elegidas. La normalización se lleva a cabo hallando para cada caracteristica el valor más alto del conjunto de los proyectos base mas el objetivo y dividiendo todos los valores de la característica por dicho valor maximo De este modo, todos los valores varían entre 0 y 1 [Zuse98].

Una vez realizada la normalización, se debe calcular un *índice de similitud* para cada proyecto base. El *índice de similitud* se define en base a la proximidad en un espacio *n*-dimensional, donde cada dimensión corresponde a una característica.

Este enfoque es muy intuitivo ya que calculamos el *índice de similitud* como la distancia euclídea en dicho espacio *n*-dimensional, así los más cercanos, los más similares, son los que están a una distancia menor, como se ilustra en la Figura 4.2.

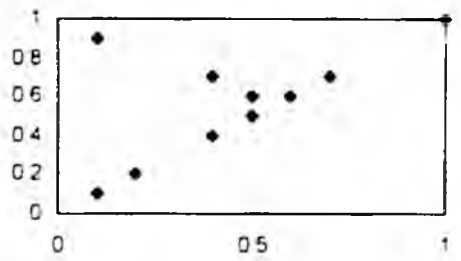

Figura 4.2 Representación de puntos similares, basados en la distancia.

La expresión que nos permite calcular el índice de similitud es:

$$ISO(P_B, P_O) = \sqrt{\sum_{i=1}^{n} (C_{P_{B_i}} - C_{P_{O_i}})^2}$$

donde P_B es el proyecto base, P_O es el proyecto objetivo, C_{PB} es una característica del proyecto base, C_{PO} es una característica del proyecto objetivo y n es el número de características a considerar.

A continuación, los índices de similitud se normalizan en el intervalo [0..100] y se establece una correspondencia de manera que el número 100 implica un emparejamiento perfecto. La función de normalización y es la siguiente:

$$y = -\frac{100}{\sqrt{2}} x + 100$$

donde x es el índice de similitud. Así, $x = 0$ implica distancia mínima y por tanto $y = 100$, mientras que $x = \sqrt{2}$ implica distancia máxima y por tanto, $y = 0$.

La puntuación obtenida en el intervalo [0..100] nos permite clasificar los proyectos base del repositorio, basándonos en este criterio. Estas puntuaciones sugieren cómo de poblada está la región que rodea al proyecto objetivo.

Consideraremos proyectos similares a aquellos que poseen una puntuación superior a un umbral (), al que haremos valer inicialmente 70, siendo este umbral calibrable dependiendo de las características de los casos almacenados en el repositorio (Figura 4.3).

Así solamente se utilizarán los proyectos cuya puntuación supera el umbral para el calculo del esfuerzo. Se denominan los *proyectos()*.

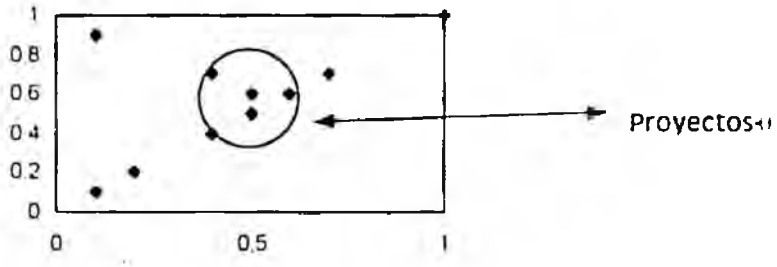

Figura 4.3 Proyectos(), cuya distancia al objetivo supera un umbral ().

Si no se encuentra ningún caso que cumpla este requisito, implica que los proyectos base son muy diferentes del proyecto objetivo y que por tanto, no se puede realizar una estimación medianamente fiable.

4.3.4 El Proceso de Transferencia y Correspondencia

El siguiente paso es la transferencia de la solución de los casos base al caso objetivo. Esta transferencia se realiza tomando el atributo esfuerzo de los proyectos base seleccionados y transfiriéndolos a la memoria de trabajo, para su posterior ajuste.

La correspondencia entre las bases y el objetivo se realiza llevando las características de todas las bases seleccionadas como similares y las características del objetivo a la memoria de trabajo; se comparan de una en una y se añaden las que no se corresponden a una lista.

4.3.5 El conocimiento para el Ajuste de las No Correspondencias y el Proceso de Ajuste

Por ultimo, debido a la existencia de características que no se corresponden entre los proyectos base y el objetivo, se debe llevar a cabo una estrategia de ajuste.

El conocimiento para el ajuste de las no correspondencias se puede clasificar en dos categorias los cinco factores de escala y los diecisiete conductores de coste del modelo COCOMO, que de forma conjunta forman el *contexto* de un proyecto es decir, expresan las caracteristicas del desarrollo del proyecto

El proceso de ajuste esta formado por dos pasos: en primer lugar, se calcula el esfuerzo estimado para el proyecto objetivo, considerando de forma individual cada proyecto-i Asi, se toma como esfuerzo inicial el que el proyecto-i posee siendo este esfuerzo modificado en base a los factores de escala y conductores de coste La funcion general de adaptacion sera la siguiente:

$$Esfuerzo_i(P_O) = Esfuerzo(P_{\theta_i}) \cdot \left(\frac{\sum_{j=1}^{f} Factor-escala_j(P_O)}{\sum_{j=1}^{f} Factor-escala_j(P_{\theta_i})} \right)^{\alpha} \cdot \left(\frac{\sum_{k=1}^{k} Conductor-coste_k(P_O)}{\sum_{k=1}^{k} Conductor-coste_k(P_{\theta_i})} \right)^{\beta}$$

donde P_O es el proyecto objetivo, P_{θ_i} es uno de los proyectos-i, i varia entre 1 y la cardinalidad de conjunto formado por los proyectos-i ($C\theta$) *Factor-escala* y *Conductor-coste* son los valores para los correspondientes factores y conductores de los proyectos, siendo f el número de factores y k el número de conductores y donde α y β son dos constantes empiricas que se deben calibrar para ajustar mejor el modelo. Nosotros consideraremos $\alpha = \beta = 1$, inicialmente. En este punto del proceso, disponemos de tantos valores estimados para el esfuerzo del proyecto objetivo, como proyectos-i hemos obtenido en la fase de recuperacion analogica Por lo tanto, se hace necesaria una función de integración que calcule un resultado único equilibrando las distintas estimaciones generadas por los proyectos-i Esta función es sensible al indice de similitud *IS* de cada proyecto-i, de manera que el esfuerzo de aquellos proyectos más cercanos al objetivo debe tener una influencia mayor sobre el resultado final. Para que esta influencia sea más patente, elevamos al cuadrado la inversa del indice de similitud, quedando la expresion de la funcion de integracion como sigue

$$Esfuerzo-estimado(P_O) = \frac{\sum_{i=1}^{|C\theta|} \frac{Esfuerzo_i(P_O)}{IS^2(P_{\theta_i}, P_O)}}{\sum_{i=1}^{|C\theta|} \frac{1}{IS^2(P_{\theta_i}, P_O)}}$$

De este modo, obtenemos un valor estimado para el esfuerzo de desarrollo de un proyecto objetivo. Obviamente, cuando el proyecto objetivo finalice, se convertirá en un proyecto base que formará parte del repositorio de casos.

En resumen, en este capítulo hemos expuesto un modelo de estimación basado en razonamiento analógico, especificando el conocimiento del dominio necesario para completar el proceso. Este modelo, al estar basado en un método que es automatizable, posee un conjunto de ventajas innegables, relativas al apoyo que puede procurar a un estimador humano.

La memoria humana es falible: podemos olvidar la información de los proyectos terminados, confundir características, omitir atributos que no tienen correspondencia y de manera inadvertida, pasar por alto factores de importancia. Sin embargo, la memoria de nuestro modelo, una vez automatizado en una herramienta, es inmune a los olvidos y distorsiones. Además, podemos realizar búsquedas exhaustivas en la base de proyectos hasta localizar la mejor analogía.

Por último, señalar que la ventaja más significativa del uso de ESTIMAN es que al ser una derivación del modelo de razonamiento humano, es intuitivo y consistente, desde el punto de vista cognitivo, con el método genérico de deliberación. Esto es, es un modelo relativamente fácil de comprender, utilizar y si es necesario, modificar.

Y como observación final sobre la estimación, consideremos las palabras de Aristóteles (330 a.C.):

> «... es característico de una mente instruida descansar satisfecha con el grado de precisión permitido por la naturaleza de cada asunto y no buscar la exactitud cuando solo es posible una aproximación de la verdad ...»

5. LA VALIDACION EXPERIMENTAL

«The software field can not hope to have its Kepler or its Newton until it has had its army of Tycho Brahes carefully preparing the well-defined observational data from which a deeper set of scientific insights may be derived.»

Barry Boehm, 1984

5.1 EL ENTORNO OPERATIVO

En el mundo de la Informática hemos tomado prestado el término *herramienta* del lenguaje artesanal. En general, una herramienta es un instrumento que se utiliza para facilitar una operación mecánica y para llevar a cabo tareas de lo más variadas. Si se utilizan de forma adecuada, las herramientas ayudan a realizar actividades de una forma más rápida y mejor. Sin embargo, como indican Giles & Daich [Gile95], si se utilizan mal, ¡podemos literalmente perder un dedo!. Se sabe que para aquel que maneja bien el martillo, todo le parece un clavo, y debemos recordar que los martillos y las sierras no construyen casas; lo hacen las personas. Evidentemente, todo lo que hemos expuesto anteriormente también se aplica a las herramientas automatizadas de medición y estimación.

Las tareas fundamentales que una herramienta automática relacionada con la medición debe realizar son tres:

- la captura de los datos, siendo esta actividad de suma importancia. Debe considerarse que al ser en esta etapa en la que se realiza la recolección de los datos, si se realiza de forma automatizada, se garantiza que dicha recolección es sistemática y repetible y que por lo tanto, los datos en los que se van a fundamentar las demás actividades van a ser uniformes, objetivos y comparables. Esto es así porque se consigue eliminar la subjetividad que introduce el componente humano al realizar una recopilación manual

Además, también debemos considerar que esta actividad es tremendamente repetitiva, aburrida, propensa a errores y que consume esfuerzo; esto es, cumple todos los requisitos para ser una actividad automatizada.

- el analisis de los datos. Esta es una tarea que puede englobar dos actividades diferentes: el cálculo de las medidas y el análisis (estadístico o de otro tipo) que se desee realizar sobre la información. Esta tarea puede resultar computacionalmente costosa, por lo que es implanteable realizarla de una manera manual (aunque una simple hoja de cálculo puede cubrir holgadamente estos requisitos).

- la generación de informes de resultados, en los que basar otros informes de seguimiento, de progreso del proyecto, de productividad, etc.

Con respecto a la estimación, cabe decir que en realidad esta actividad no la realizan directamente las herramientas, sino los expertos, ya que se necesitan tomar decisiones razonadas y cumplir con los objetivos de la organización de un modo que no puede ser simplemente delegado a un proceso automático. Las herramientas deben

- liberar al experto de las actividades de cálculo pesado. Debemos considerar que las estimaciones se basan en el *proceso de comparación*. Para realizar dichas estimaciones, los expertos evalúan las similitudes y diferencias con respecto a situaciones que se han experimentado anteriormente.

- ser la memoria del experto, considerando que antes de poder realizar las estimaciones, se debe adquirir el conocimiento necesario para realizarlas. Se debe recopilar y cuantificar toda la información de interés de otros proyectos, de manera que los expertos puedan realizar sus evaluaciones comparativas sobre bases firmes y demostrables [Park94].

Por todo lo anteriormente expuesto, es obvio que para completar este trabajo es imprescindible el desarrollo de una herramienta que mecanice los distintos aspectos relacionados tanto con el proceso de medición como con el método de estimación que proponemos.

En la Figura 5.1, se ilustra la arquitectura modular del conjunto de herramientas que de forma integrada implementan el conjunto completo de actividades de nuestra investigación.

La arquitectura propuesta está formada por cuatro módulos distintos que giran alrededor de un repositorio general de proyectos orientados a objetos.

En este repositorio se almacenan las características estructurales de dichos proyectos, en base a los cuales se realizan todos las actividades posteriores. El modelo de datos del repositorio general se describe de forma más detallada en la sección 5.1.4.

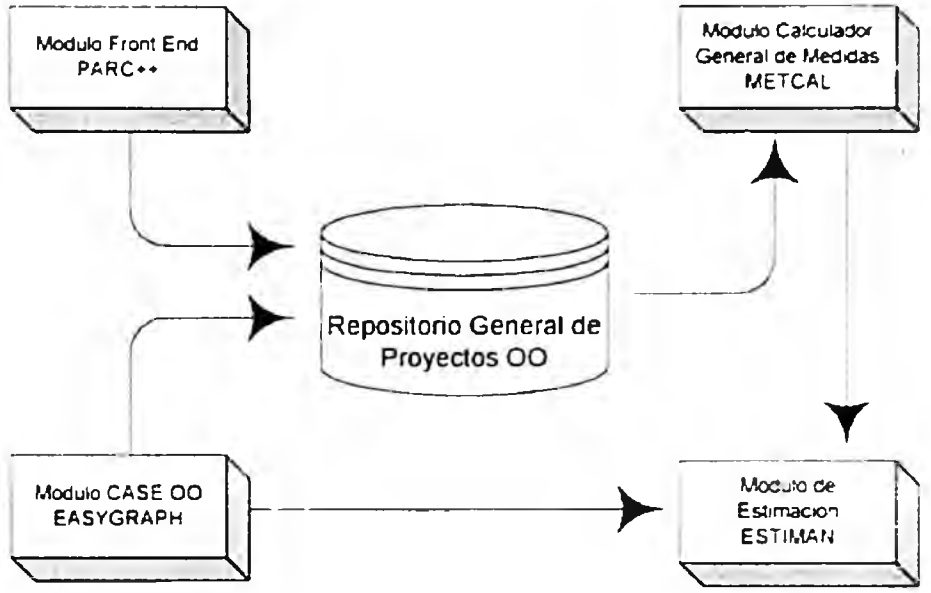

Figura 5.1 Arquitectura general del entorno operativo de Medición y Estimación.

El módulo *EASYGRAPH* es una sencilla herramienta CASE que admite como única diagramación posible el modelo de clases siguiendo la notación UML. Este módulo cumple la función de captura de datos, datos que se depositan en el repositorio. Además, posee una conexión con el módulo *ESTIMAN* a través de la cual le transmite las medidas necesarias para el proceso de estimación.

El módulo *PARC++* también está dirigido a la captura de datos, pero desde una perspectiva temporal diferente: sirve para la extracción de características de proyectos a partir de código fuente C++. Así, podemos poblar el repositorio con información de proyectos pasados de los que no se disponga documentación de la fase de análisis.

El módulo *METCAL* no está directamente relacionado con el objetivo de esta investigación. Es un módulo que implementa el cálculo de dos colecciones de medidas propuestas en la literatura, además de nuestra medida de interdependencia. Además, dispone de un componente que automáticamente comunica a *ESTIMAN* el valor de la medida de tamaño y complejidad de los proyectos del repositorio general, con vistas a poblar inicialmente el repositorio interno de *ESTIMAN*.

Y este es el cuarto y último módulo, *ESTIMAN*, cuyo objetivo es proporcionar una estimación del esfuerzo lo más precisa posible, basándose para ello en la información de proyectos pasados

A continuación vamos a describir con mayor detalle la composición interna de cada uno de los módulos y de los repositorios necesarios para su correcto funcionamiento.

5.1.1 El módulo CASE EASYGRAPH

Este módulo implementa una sencilla herramienta CASE cuyo objetivo primario es captar la información de la que se dispone en las fases iniciales de un proyecto orientado a objetos. Por ello, está especialmente diseñada para diagramar el modelo de clases del dominio, a partir del cual podemos extraer las características estructurales que consideremos de interés.

En la Figura 5.2, se ilustra la descomposición interna de este módulo.

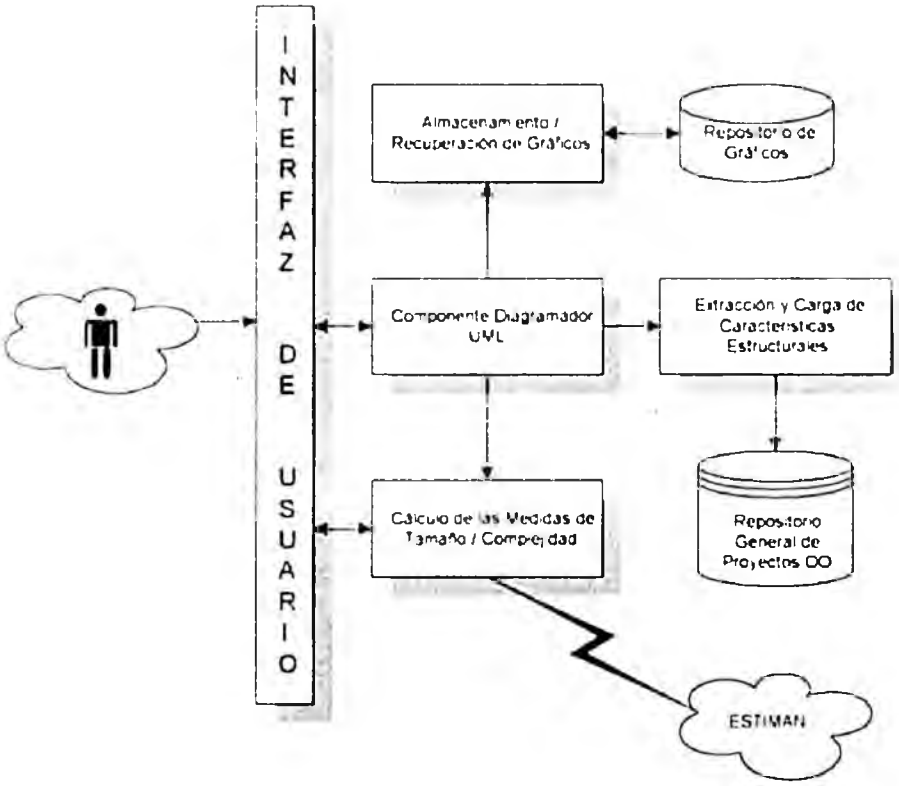

Figura 5.2 Componentes del módulo EASYGRAPH.

Este modulo posee una *Interfaz Grafica* de usuario puesto que su función principal es la introducción de un modelo de clases pero a traves del componente *Diagramador UML*. Los elementos graficos que se pueden diagramar son aquellos que representan la información de la que se dispone en las fases tempranas de un proyecto como clases del dominio, atributos, servicios de alto nivel relaciones de generalizacion, agregacion y asociaciones y relaciones de uso (mensajes)

Tambien disponemos de un componente de *Almacenamiento Recuperacion de Graficos*, de modo que nuestros diagramas sean persistentes gracias a un *Repositorio de Graficos*

A partir de la informacion que se ha introducido de forma grafica el componente de *Extracción/Carga de Caracteristicas Estructurales* capta todas los atributos de interés para depositarlos en el *Repositorio General de Proyectos OO*. De este modo disponemos de toda esa información de forma persistente y para que pueda ser utilizada por otros modulos.

Además, la información gráfica también sirve para calcular directamente distintas medidas extraibles del modelo del dominio, mediante el componente *Calculo de las Medidas de Tamaño/Complejidad*. Así, por ejemplo, podemos calcular el número de clases, atributos y operaciones del modelo, el numero medio de atributos y operaciones por clase, el número de clases abstractas el numero de clases independientes, asi como nuestra medida *ID* para cada clase global para todo el modelo e ID_{media}. Tambien podemos extraer estas medidas para una seleccion limitada del dominio. Consideramos que la posibilidad de realizar estos calculos de medidas *on-line*, mientras se está diagramando el dominio, es de sumo interes, puesto que nos permite tener una base sobre la que valorar nuestras propias decisiones. Esto es, podemos probar modelizaciones alternativas y valorar su complejidad, seleccionando aquella que resulte mas simple, tambien podemos tomar nota de aquellas clases que superan el numero medio de operaciones por clase o la ID_{media}, puesto que es probable que dichas clases sean mas dificiles de implementar e incluso de probar y mantener.

Por último, este módulo mantiene una conexión con el modulo *ESTIMAN*. Dado un proyecto en una fase inicial de su ciclo de vida podemos modelizar su dominio graficamente y calcular el número de clases contempladas y su ID_{media}. Mediante una conexion que envia estos dos parametros a *ESTIMAN*, podemos proseguir la actividad en este último modulo y despues de introducir algunos datos adicionales, estimar el esfuerzo en personas-semana, para este proyecto determinado

5.1.2 El módulo PARC++

Este módulo tiene como objetivo cargar el *Repositorio General de Proyectos OO* con características estructurales extraídas de código fuente C++. La razón de la existencia de este módulo viene de la necesidad de contar con una muestra lo más amplia posible de proyectos terminados con éxito en el pasado. Y una característica que no debemos obviar de estos proyectos es el hecho de que mayormente no se han documentado; es decir, no disponemos de ninguna documentación útil de la que extraer las medidas del modelo del dominio, y si existe alguna documentación, generalmente sigue una diagramación clásica de tipo estructurado.

Este módulo es en cierta forma un módulo de Ingeniería Inversa, puesto que su utilidad para nosotros se centra en obtener la información del dominio a partir del código fuente. Los componentes internos de este módulo están representados en la Figura 5.3.

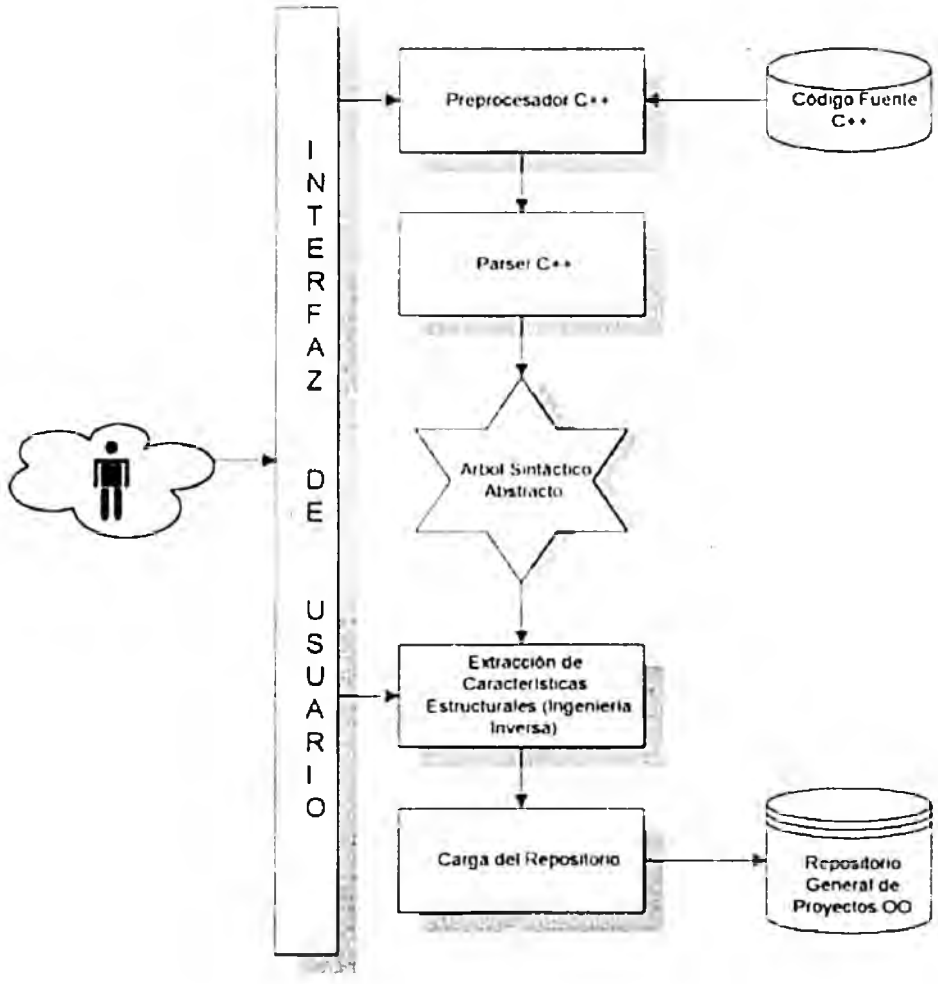

Figura 5.3 Componentes del módulo PARC++.

El funcionamiento de este módulo se divide en tres fases En primer lugar se toma el codigo fuente C - - que entra a un *Preprocesador C - -* encargado de resolver el uso de las definiciones distribuidas entre los distintos archivos La salida proporcionada por el preprocesador alimenta al *Parser C - -* que origina un *Arbol Sintactico Abstracto*, con todas las caracteristicas extraidas del codigo

En la segunda fase, se hace indispensable la colaboracion del usuario En el *Arbol Sintactico Abstracto* se encuentran representadas absolutamente todas las clases del código, con todos sus atributos y metodos Entre estas clases se encuentran las del dominio, pero tambien todas las demas, las de soporte de interfaz, de comunicaciones, de base de datos, de utilidad, etc El usuario tiene la responsabilidad de filtrar las clases del dominio, con el fin de poder utilizar posteriormente sólo dichas clases

En la tercera fase, se utiliza el componente de *Extraccion de Caracteristicas Estructurales*, que utilizando únicamente las clases del dominio, pasa la información correspondiente al componente de *Carga del Repositorio*, de manera que las propiedades relacionadas con el dominio del proyecto terminado se almacenan en el *Repositorio General de Proyectos OO*, mediante un intento de emular las condiciones de las fases iniciales.

Es de importancia señalar que la calidad de los datos que se vuelcan en el repositorio después de este proceso es variable: dependerá principalmente de lo riguroso que haya sido el usuario al hacer la seleccion de las clases del dominio

5.1.3 El módulo METCAL

En el modulo *METCAL* podemos distinguir dos funciones bien diferenciadas Una de ellas, tiene como objetivo el calculo de dos colecciones de medidas propuestas en la literatura *MOOSE* de Chidamber & Kemerer [Chid94] y *MOOD* de Brito e Abreu [Brit96], comentadas en las secciones 2 4 2 3 y 2 4 2 10 respectivamente Para realizar estos cálculos se debe seleccionar previamente el proyecto objetivo dado el conjunto de proyectos almacenados en el *Repositorio General de Proyectos OO*

La segunda función, implementada en el componente *Cálculo de Medidas Tipicas y de Tamaño Complejidad*, a su vez, también tiene una doble vertiente Por un lado, nos permite calcular medidas generales relacionadas con el tamaño, ademas de nuestra medida de complejidad global *ID* e ID_{metJ}, dado un proyecto objetivo Pero por otra parte, nos permite realizar el calculo masivo de estas últimas medidas para todos los proyectos del repositorio. El objetivo último de esta función es la

comunicación con el módulo *ESTIMAN* y el envío masivo en forma de parámetros de las medidas de los proyectos, de manera que se cargue el repositorio interno de *ESTIMAN*.

Es decir, esta función realmente realiza la segunda parte de la obtención de muestras de proyectos pasado, siendo la primera la realizada por el módulo *PARC* , y realizandose el secuenciamiento de ambas actividades a través del *Repositorio General de Proyectos OO*

Los componentes internos del modulo METCAL se ilustran en la Figura 5.4

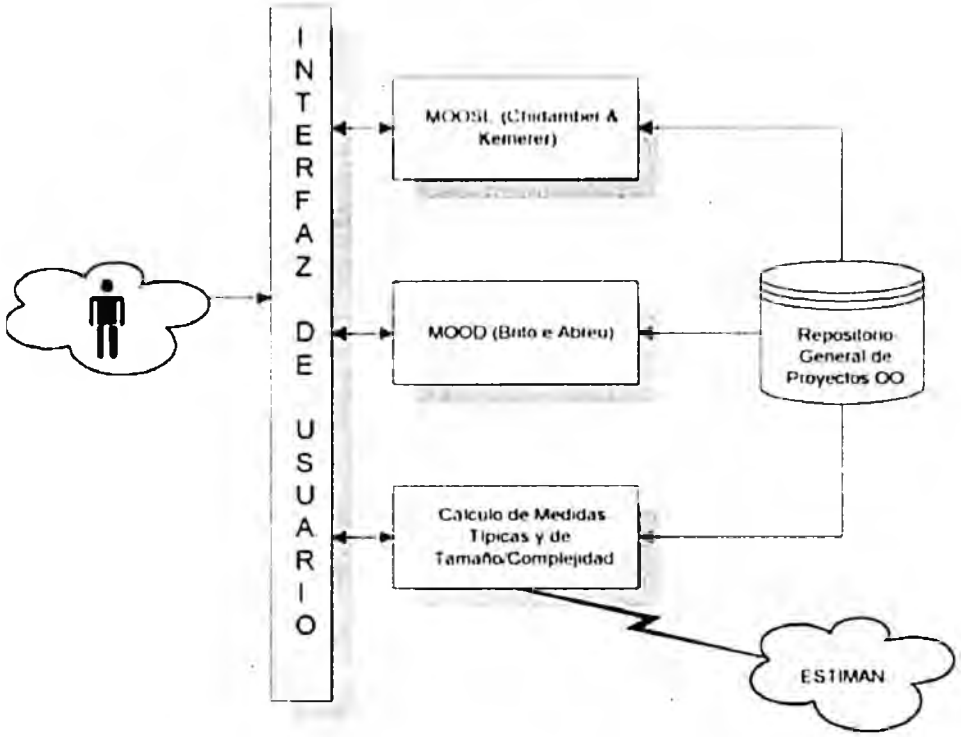

Figura 5.4 Componentes del módulo METCAL.

Por último, con respecto a este módulo, nos queda mencionar su utilidad en el cálculo de distintos conjuntos de medidas con fines comparativos, ya que nos permite calcular medidas cuyo fundamento es similar pero que están planteadas de modos diferentes y nos puede ayudar a localizar relaciones entre ellas.

5.1.4 El Repositorio General de Proyectos OO

Los tres módulos previamente descritos están vinculados al *Repositorio General de Proyectos OO*. *PARC++* y *EASYGRAPH* son los modulos responsables de la carga del repositorio y *METCAL* se encargan realiza calculos relacionados tanto con nuestras medidas de tamano y complejidad como con las medidas de los conjuntos *MOOD* y *MOOSE*.

Por lo tanto, es obvio que el punto de confluencia de los tres modulos es el repositorio. Por ello, cuando nos planteamos el diseno de dicho repositorio tuvimos que realizar un analisis exhaustivo de las necesidades de informacion de *METCAL* puesto que es el modulo que hace un uso real de los datos almacenados.

Con respecto a *MOOD*, no tuvimos problemas, puesto que todas las medidas de esta coleccion se definen con expresiones matematicas claras y precisas, por lo que la tarea se centro unicamente en elaborar un relacion de las variables que participaban en los cálculos.

Sin embargo, *MOOSE* nos llevó algo más de trabajo, ya que en el articulo que proporciona la definición de las medidas, la descripción de estas se realiza de forma textual, por lo que tuvimos que analizar con detalle cada definicion para no cometer errores relacionados con una mala interpretación de las variables. En la siguiente figura, se ilustra el modelo de datos del *Repositorio General de Proyectos OO*.

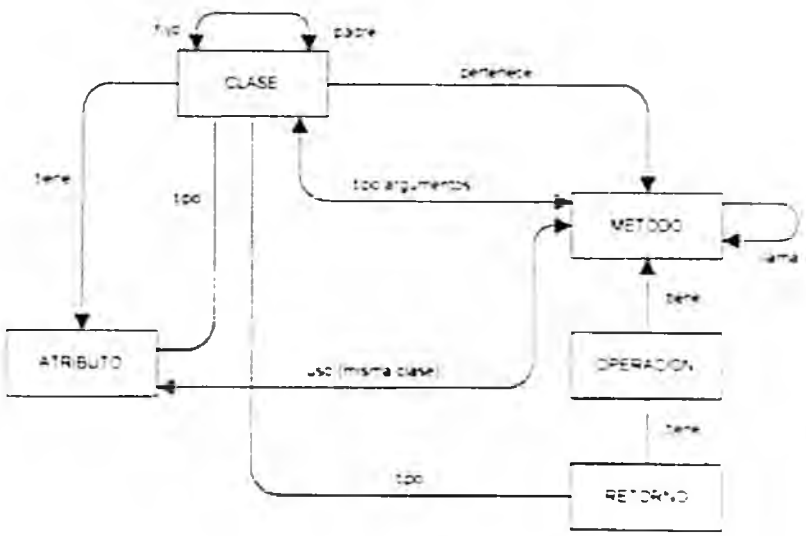

Figura 5.5 Modelo de Datos del Repositorio General de Proyectos OO.

El diccionario de datos asociado al modelo anterior se expone a continuación.

```
CLASE = nombre-clase + nivel + lineas-código
    NIVEL = [raiz  otro]
CLASE-HIJO = nombre-clase-padre + nombre-clase-hijo
ATRIBUTO = nombre-atributo + visibilidad + tipo
    VISIBILIDAD = [publica  privada  protegida]
    TIPO = [primitivo  nombre-clase]
CLASE-ATRIBUTO = nombre-clase + nombre-atributo
OPERACION = nombre-operacion + visibilidad + tipo-retorno
    TIPO-RETORNO = [primitivo  nombre]
METODO = nombre-clase + nombre-operacion + id-argumentos + lineas-código
ARGUMENTO-CLASE = id-argumentos + nombre-clase
METODO-ATRIBUTO = nombre-clase + nombre-operacion + nombre-atributo
LLAMA = nombre-clase-e + nombre-operacion-e + nombre-clase-r + nombre-operacion-r
```

Figura 5.6 Diccionario de Datos del Repositorio General de Proyectos OO.

Tambien debemos señalar que las variables necesarias para calcular medidas de tipo general, así como nuestra medida de complejidad, ya se encuentran incluidas entre las variables necesarias para implementar el cálculo de *MOOD* y *MOOSE*.

Por ultimo, nos queda indicar que los modulos *PARC++* y *EASYGRAPH* también estan condicionados por la estructura anterior, ya que únicamente captarán la informacion anteriormente descrita (que es la que se puede almacenar).

5.1.5 El módulo ESTIMAN

Este módulo es el mas interesante de los cuatro debido a que en realidad es un modulo genérico, una *meta-herramienta* que implementa una estrategia concreta pero que es altamente *parametrizable*, por lo que su uso no queda restringido a este trabajo, sino que da pie a iniciar múltiples lineas futuras de investigación.

En la Figura 5.7, se representa la estructura interna del módulo ESTIMAN.

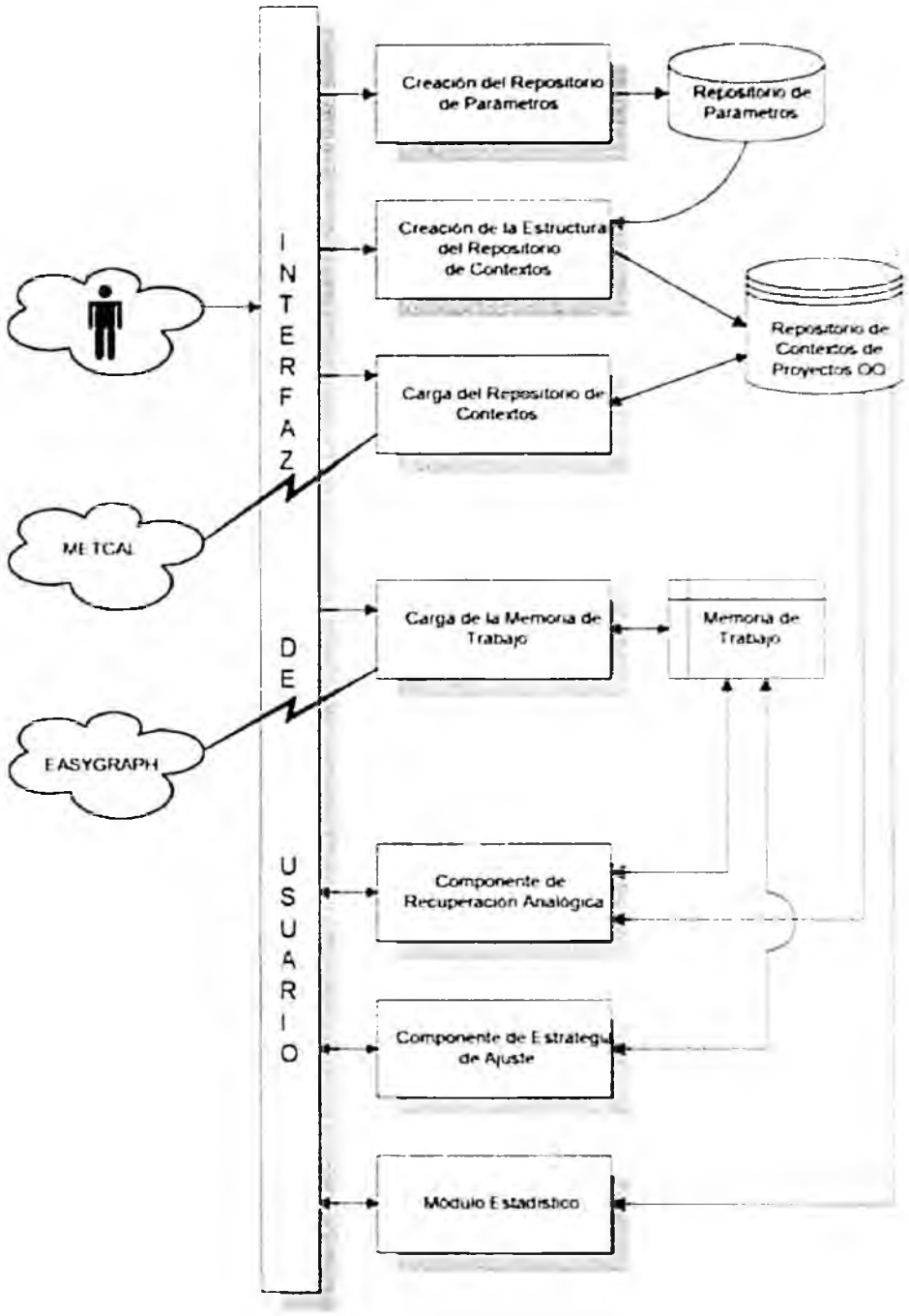

Figura 5.7 Componentes del módulo ESTIMAN.

El modulo *ESTIMAN* centra todas sus actividades alrededor de un *Repositorio de Contextos de Proyectos OO* y podemos dividir su funcionamiento en tres fases bien diferenciadas:

1. Creación de un *Repositorio de Contextos de Proyectos OO*.
2. Carga de dicho repositorio.
3. Proceso de Estimación por Analogía.

La primera fase implica crear una estructura de repositorio, pero como hemos mencionado previamente, nuestra intención es implementar una meta-herramienta, no condicionada ni ligada a unos datos concretos. Por este motivo, la estructura del repositorio no esta predefinida, no es estática; es una estructura que debemos generar dinamicamente y podemos generar tantas como consideremos necesarias. Para ello, disponemos del componente de *Creación del Repositorio de Parámetros*, que se encarga de captar y almacenar los parámetros concretos en los que vamos a basar la estructura del *Repositorio de Contextos de Proyectos OO* particular que vayamos a utilizar. En la siguiente figura se enumeran los meta-datos del *Repositorio de Parametros*.

ESTATICOS

IDENTIFICADOR DE PROYECTO
ESTADO DEL PROYECTO = (terminado activo)

VARIABLES

ATRIBUTO DE BUSQUEDA
ESTIMADOR
FACTOR DE ESCALA
CONDUCTOR DE COSTE
TEXTO

Figura 5.8 Meta-datos del Repositorio de Parámetros.

Los elementos estáticos *Identificador de Proyecto* y *Estado* son obligatorios e inamovibles. Sin embargo, los datos variables son los que podemos parametrizar en función de nuestras necesidades; además no son obligatorios, por lo que podemos prescindir de ellos. Por ejemplo, podemos implementar un método simplificado que no utilice *Factores de Escala* ni *Conductores de Coste* e incluso podemos prescindir

de elementos de tipo *Texto* si no nos interesa almacenar informacion adicional sobre los proyectos. Obviamente de lo que no podemos prescindir es de utilizar al menos un atributo de búsqueda y un estimador. En cuanto al número de los elementos, este es completamente variable y solo tenemos como límite superior la capacidad del ordenador (pero no debemos perder de vista que los cálculos que se realizan son computacionalmente muy costosos y que cuanto mayor sea el numero de elementos, obtendremos tiempos de respuesta peores).

Una vez creado el *Repositorio de Parametros* concretos utilizando los metadatos anteriores, entra en funcionamiento el componente de *Creacion de la Estructura del Repositorio de Contextos*, que crea una estructura de base de datos en función de los parámetros señalados

Para esta investigación, hemos creado un repositorio utilizando los parametros que se enumeran a continuación

ESTATICOS

IDENTIFICADOR DE PROYECTO
ESTADO DEL PROYECTO = [terminado activo]

VARIABLES

ATRIBUTOS DE BUSQUEDA
- Número de Clases Estimadas
- Interdependencia Media

ESTIMADOR
- Esfuerzo real en personas-semana

FACTORES DE ESCALA
- Precedencia
- Flexibilidad en el Desarrollo
- Arquitectura / Resolucion de Riesgos
- Cohesion del Equipo
- Madurez del Proceso

CONDUCTORES DE COSTE
- RELY, DATA, CPLX, RUSE,DOCU, TIME,
- STOR PVOL, ACAP, PCAP PCON AEXP
- PEXP, LTEX, TOOL, SITE SCED

TEXTOS
- Esfuerzo Real
- Fecha de Comienzo y Fin
- Tamaño medio del Equipo

Figura 5.9 Nuestra parametrización concreta en el Repositorio de Parámetros.

En nuestra parametrización, identificamos los proyectos mediante un nombre y consideramos dos estados:

- *terminado*, que es un proyecto pasado que servirá como base de la estimación,
- *activo*, que es un proyecto potencial a estimar

Logicamente, podemos tener varios proyectos terminados, pero también varios proyectos en curso sobre los que deseemos realizar estimaciones. Utilizamos dos atributos de búsqueda que son:

- una medida de tamaño (el número de clases estimadas) y
- una de complejidad (la complejidad media de las clases en forma de ID_{media}).

Además, como factores de escala y atributos de coste, usamos los propuestos por COCOMO II, en su versión post-arquitectural. Como elementos de texto que documentan nuestros proyectos hemos elegido:

- el esfuerzo estimado para los proyectos pasados, ya que esta información nos va a permitir valorar la precisión del método. Basta con comparar el esfuerzo real que se almacena en el repositorio con el que se estimó en su día y dispondremos de una indicación de la precisión de la estrategia.
- Fecha de Comienzo y Fin, de forma que podamos eliminar del repositorio aquellos proyectos que sean excesivamente antiguos.
- Tamaño medio del Equipo, información que nos proporciona una idea de los recursos humanos que se dedicaron al proyecto.

Para finalizar con esta explicación sobre los repositorios, en la Figura 5.10, se ilustran los meta-datos de *ESTIMAN*, utilizando la modelización UML.

Una vez creada la estructura del *Repositorio de Contexto de Proyectos OO*, la segunda fase del conjunto de componentes de *ESTIMAN* se centra en la carga de dicho repositorio. Para ello, disponemos de dos posibilidades:

- la carga masiva del repositorio, opción en la que colabora el módulo *METCAL*, que proporciona la información estructural de los proyectos almacenados en el *Repositorio General de Proyectos OO*. No debemos perder de vista que la información así obtenida es parcial y estará relacionada principalmente con los elementos estáticos y con los atributos de búsqueda. Por ello, se hace imprescindible la segunda posibilidad.

- la carga manual, a través de la *Interfaz de Usuario*. Así, podemos completar la información del repositorio que *METCAL* no proporciona, como el esfuerzo de desarrollo, los factores de escala, los conductores de coste o los elementos de texto. Además, esta opción también es recomendable cuando se vayan a introducir los datos de un número reducido de proyectos, aquellos que van terminando a medida que transcurre el tiempo.

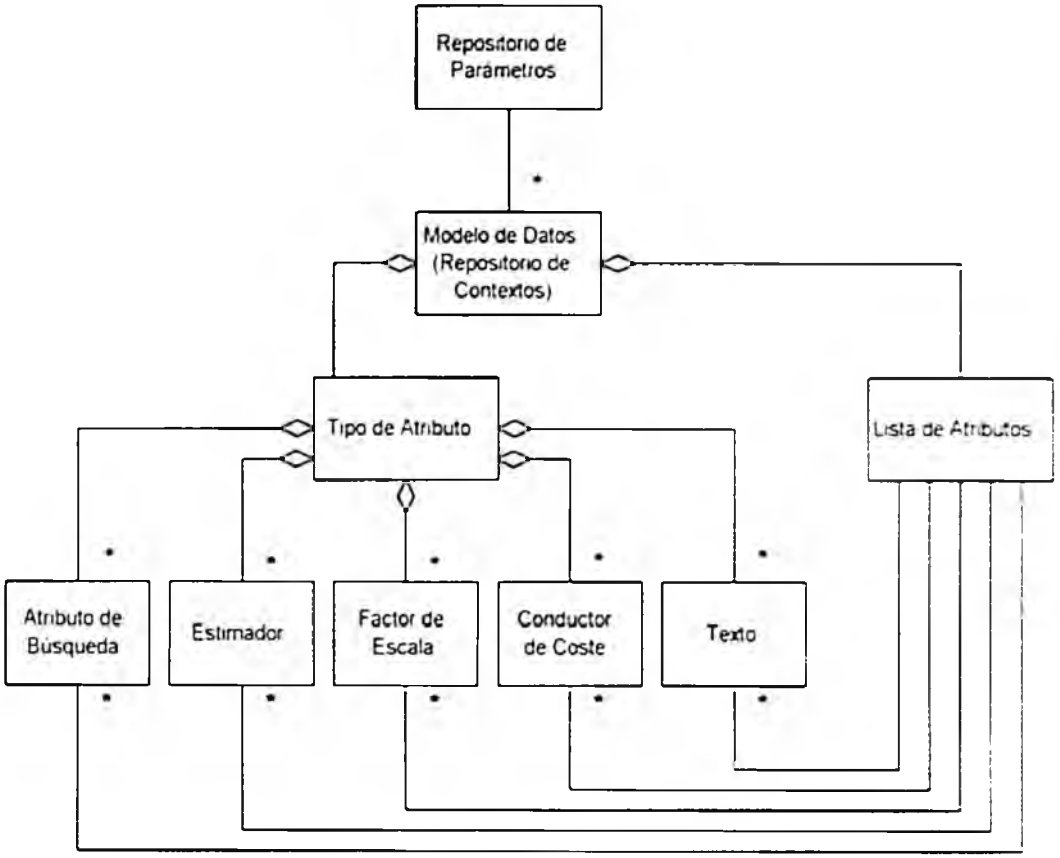

Figura 5.10 Modelización UML de los Meta-datos de ESTIMAN.

La tercera fase es la que está relacionada con el proceso de Estimacion por Analogía. El primer componente que se activa es el que se encarga de la *Carga de la Memoria de Trabajo*. A través de este componente se debe introducir la informacion que disponemos sobre el proyecto objetivo. Se admiten dos posibilidades

- la recepción de la información a través de *EASYGRAPH* Nótese que del módulo CASE únicamente podemos recibir los valores de los atributos de búsqueda por lo que necesitaremos añadir la información sobre el contexto, para poder usar la estrategia completa Por ello, también se implementa la segunda opción.

- la introducción de la información directamente a través de la *Interfaz de Usuario*. Esta opción se puede utilizar tanto para completar la información proporcionada por *EASYGRAPH*, como de manera independiente con el fin de suministrar todos los datos necesarios para el proceso de estimación.

La *Memoria de Trabajo* es, como su nombre indica, la memoria volátil donde se van a almacenar todas las informaciones intermedias que se produzcan como parte del procesamiento

Una vez se ha cargado esta memoria con los datos del proyecto objetivo, se activa el *Componente de Recuperación Analógica*, cuyo objetivo es extraer del *Repositorio de Contextos de Proyectos OO*, aquellos proyectos que sean similares al objetivo en base a los *Atributos de Búsqueda*. La información de los proyectos similares se deposita en la *Memoria de Trabajo*. Seguidamente, se activa el *Componente de Estrategia de Ajuste* que interactuando con la información de la *Memoria de Trabajo*, se encarga de realizar el proceso de ajuste del *Estimador* en base a los *Factores de Escala* y *Conductores de Coste*, ofreciendo finalmente un valor para el elemento *Estimador*.

El ultimo componente que hemos añadido al módulo *ESTIMAN* es el *Módulo Estadístico*, cuyo objetivo es valorar la precisión de las estimaciones que se pueden realizar con los proyectos almacenados en un repositorio. Este componente calcula MMRE, que es la media del error relativo absoluto y PRED, que es un indicador de la capacidad de prediccion del método en base a la muestra considerada [Cont86].

5.2 EL ESTUDIO EMPIRICO

Para llevar a cabo validación experimental de nuestro método de estimación ESTIMAN, y demostrar la validez empírica de la medida *ID*, diseñamos una estrategia basada en dos etapas:

1 Validación *parcial* del método utilizando proyectos del entorno *académico*.

2 Validación *completa* utilizando proyectos del ámbito *industrial*.

El motivo que nos llevó a dividir la validación en dos fases es la dificultad que entraña recoger una muestra industrial amplia de proyectos terminados en la actualidad. Esto es debido a que, aunque innumerables organizaciones están utilizando lenguajes de programación orientados a objetos, pocas estan utilizando un método de modelización que no sea clásico (desde DFDs hasta listas de requisitos funcionales). A esto hemos de añadir que hay organizaciones que no recogen

información sobre sus proyectos terminados, con lo que no guardan registro de las características de dichos proyectos, las incidencias en el desarrollo, el numero de personas que se involucraron en el mismo, la agenda o algún mecanismo para calcular el esfuerzo. Por último, también cabe mencionar el hecho de que aquellas organizaciones que acumulan estos datos los consideran absolutamente confidenciales, puesto que hoy en día pueden ser utilizados como una ventaja estratégica frente a la competencia. Es por esto que apenas se encuentran en la literatura especializada experiencias reales que aporten los datos de las muestras en base a los cuales poder replicar, por ejemplo, un experimento o probar un nuevo método [West96]

Por todo lo anteriormente expuesto, decidimos abordar en primer lugar una validación utilizando muestras del mundo académico, siendo conscientes que las conclusiones a las que llegáramos no podrían ser generalizadas sin un análisis posterior mas exhaustivo.

Los inconvenientes que habitualmente se achacan a los proyectos académicos es que los desarrolladores son estudiantes y por tanto, desarrolladores atípicos Generalmente son inexpertos y su capacidad de trabajo y habilidad muy variables de un individuo a otro. Además, estos proyectos suelen estar restringidos a ejercicios de programación de poca envergadura y desde luego, no son indicadores de un entorno de desarrollo y mantenimiento normal La situación ideal sería emplear un importante número de profesionales con una capacidad similar Obviamente, la probabilidad de disponer de esta muestra es realmente pequeña, comparada con la relativa disponibilidad de una población de estudiantes Así, es importante acercar las características de ambas poblaciones: intentar tener lo mas parecido a un profesional dentro del entorno académico MacDonell [MacD91] sugiere que se emplee como muestra a los estudiantes de último año de carrera y que sean de una capacidad similar, intentando tener una muestra amplia para intentar disminuir los efectos de las observaciones individuales En consecuencia, extrajimos nuestra muestra de la colección de proyectos fin de carrera de los alumnos de quinto año desarrollados durante el curso académico 97/98, ya que llegamos a la conclusión de que cumplían las características previamente comentadas eran proyectos de estudiantes cercanos a la vida profesional (algunos ya se habían incorporado al mundo laboral) y lo que es más importante, sus desarrollos no eran triviales, no eran meros ejercicios de programación.

Sin embargo, nuestra decisión nos imponía una restricción importante, ya que como consecuencia de utilizar proyectos académicos, nuestra validación sería *parcial* Esto es, no podíamos realizar una prueba del método ESTIMAN completo debido a que los proyectos provenían de desarrollos similares desde el punto de vista del

contexto de desarrollo, con lo que no tenía sentido aplicar la estrategia de ajuste en el proceso de estimación. Decidimos dejar la prueba de este componente para la validacion industrial.

Una vez determinada cuál iba a ser nuestra muestra y el alcance de la validación, nos planteamos la selección de las técnicas de recolección de datos Zelkowitz señala que estas técnicas se agrupan en tres categorías generales [Zelk98]:

1. Métodos *Observacionales*,
2. Métodos *Históricos*, y
3. Metodos *Controlados*.

Los métodos *observacionales* recogen la información relevante de un proyecto a medida que transcurre el proceso de desarrollo. A su vez se consideran cuatro tipos: la monitorización de proyectos, el estudio de casos, los asertos y los estudios de campo.

Los métodos *históricos* recogen los datos a partir de proyectos que ya han terminado, utilizando los datos disponibles después de su finalización. Existen cuatro métodos de esta categoría: la búsqueda en la literatura especializada, el estudio de datos 'heredados', las 'lecciones aprendidas' y el análisis estático.

Los métodos *controlados* proporcionan múltiples instancias de una observacion con el fin de proporcionar unos resultados con validez estadística. Es el clasico método de experimentación utilizado en otras disciplinas científicas. Existen cuatro tipos: el experimento replicado, el del entorno sintético, el análisis dinámico y la simulación.

De todos estos métodos, los que mejor se ajustaban a nuestras necesidades eran el *análisis estático*, junto con el *estudio de datos 'heredados'*. No nos interesaba utilizar ningún método observacional, puesto que no queríamos interferir en el proceso de desarrollo natural de los proyectos de nuestros alumnos. Tampoco podíamos utilizar métodos controlados porque no disponíamos de recursos para paralelizar varios desarrollos. Por lo tanto, de los métodos históricos decidimos utilizar el *análisis estático*, que se centra en estudiar un producto terminado con el fin de obtener sus características, y el *estudio de datos 'heredados'*, que incluye el análisis de toda la información disponible sobre un proyecto, desde el código fuente hasta la documentación de análisis, diseño, planificación, pruebas, información proporcionada por el director del proyecto, etc.

5.2.1 La selección de los casos

Se pusieron a nuestra disposicion aproximadamente unos 100 proyectos, de los cuales éramos conscientes que muchos no nos servirían. Para poder realizar la seleccion de los más adecuados para nuestra validacion, decidimos establecer unos criterios a cumplir:

1. La informacion sobre el dominio de los proyectos debia estar documentada siguiendo la modelizacion OMT o UML.

2. Los proyectos debían considerar una arquitectura típica, con una interfaz de usuario gráfica y un servidor de base de datos.

3. El lenguaje de programación debía ser Java, C++ o 4GL orientados a objetos (Visual Basic V5 o Delphi).

4. Los proyectos debían haber sido desarrollados siguiendo un ciclo de vida completo y uniforme.

5. Debía haber información sobre la planificación de recursos, con el fin de poder calcular el esfuerzo de desarrollo

Después de examinar la documentacion correspondiente, limitamos nuestra muestra a 13 proyectos, de tamaño entre pequeño y mediano. Todos ellos estaban documentados siguiendo la modelizacion OMT o UML. Desde el punto de vista del lenguaje de programacion, 4 proyectos se desarrollaron en Java, 3 en C++, 4 en Visual Basic V5 y 2 en Delphi. Todos habían seguido un ciclo de vida completo y similar y los esfuerzos de desarrollo variaron entre 10 y 56 personas-semana. Con respecto al dominio de las aplicaciones en la seleccion había sistemas de gestion típicos, aplicaciones de comunicaciones y sistemas sobre Internet/Intranet entre otros.

5.2.2 Procedimiento de Recolección de Datos

Como la informacion a extraer para nuestro banco de datos estaba distribuida en una documentación en papel, nos vimos obligados a realizar la ingrata tarea de la captura de los datos de manera manual, sin poder utilizar nuestro entorno integrado. Para poder realizar esta actividad de una manera cómoda y uniforme diseñamos tres formularios:

1 Un *Formulario de Recogida de Datos*, cuya función es recopilar la información necesaria para el cálculo posterior de las medidas.

2 Un *Formulario de Cálculo de Medidas*, basado en la información del formulario anterior, automatizable en una hoja de cálculo. En este formulario se encuentran los datos necesario para llevar a cabo el proceso inicial de estimacion.

3 Un *Cuestionario de Definición de Contextos*, basado en el de las organizaciones que colaboran en la evolución del método COCOMO II. En este documento se encuentran las cuestiones que ayudan a definir la situación de un proyecto con respecto a los factores relacionados con el desarrollo del mismo que afectan al esfuerzo.

En el Apéndice C, adjuntamos el patrón del *Formulario de Recogida de Datos* y el *Formulario de Cálculo de Medidas*, además de dos ejemplos ya cumplimentados. También en este mismo apéndice, adjuntamos nuestro *Cuestionario de Definición de Contextos*.

5.2.3 Selección de los Métodos Estadísticos

Para elegir los metodos de análisis de datos que mejor se ajustaran a nuestras necesidades de validacion, decidimos seguir las directrices propuestas por Schneiderwind [Schn92, Schn94] y por IEEE en su estándar sobre medición [IEEE92a]. Estos criterios de validacion han sido ampliamente utilizados en la literatura. De todos ellos, los que se centraban en los puntos de nuestro interés son los siguientes:

1 *Capacidad Predictiva*. Es una medida de la precisión de nuestro modelo para realizar estimaciones de esfuerzo. Este criterio es especialmente importante porque el exito de un proyecto de desarrollo puede estar completamente condicionado por el resultado de la estimación. Una estimación imprecisa puede llevarnos a la selección de proyectos poco adecuados y tener un serio impacto en las decisiones que se deban tomar sobre la planificación temporal y de recursos humanos.

2 *Consistencia*. Debemos considerar el hecho de que un modelo puede ser impreciso y ser consistente, si los errores de estimación son uniformes dado un conjunto de proyectos. Por ejemplo, es preferible un modelo que sobreestima en un 50% de manera consistente, que uno que estima al azar el 50% de las veces hacia arriba y el otro 50% restante hacia abajo. Si un modelo

no es consistente, las estimaciones que produzca serán poco fiables, con lo que no tendra ningún uso práctico.

Una vez seleccionados los criterios de validación, consideramos la seleccion de los métodos estadísticos a realizar sobre nuestra muestra.

5.2.3.1 La Capacidad Predictiva

La precisión de las estimaciones de un modelo se puede medir mediante el cálculo de la *Media del Error Relativo Absoluto (MMRE - Mean Magnitude of Relative Error)*, que es una medida normalizada de la discrepancia entre valores reales y valores estimados. Su cálculo se realiza siguiendo la siguiente expresión:

$$MMRE = \frac{\sum_{i=1}^{n} \frac{|Esfuerzo\_real - Esfuerzo\_estimado|}{Esfuerzo\_real}}{n}$$

Esta medida fue propuesta por Conte *et al.* [Cont86] y es una de las más utilizadas para la validación de métodos de estimación [Lo95, Mars97, Mukh92, Mukh92a, Srni95, Vici90, Vici91].

Sirve para calcular el error relativo de cada proyecto estimado, eliminando cualquier problema relacionado con el tamaño del mismo. Además, otro problema que se debe considerar es el hecho de que el modelo sobreestime, es decir, proporcione estimaciones por encima del valor real, y también proporcione estimaciones por debajo de dicho valor. En estos casos, el error relativo tendría signo negativo y positivo, respectivamente, y al calcular la media, unos errores compensarían el efecto de los otros. Este es el motivo por el que se utiliza el valor absoluto del error relativo.

Otra medida ligeramente distinta y que considera el tamaño de los proyectos implicados en la muestra es la *Media del Error Relativo Ponderado (WMMRE - Weighed Mean Magnitude of Relative Error)*, cuya expresión es la siguiente:

$$WMMRE = \frac{\sum_{i=1}^{n} \frac{|Esfuerzo\_real - Esfuerzo\_estimado|}{\sum_{i=1}^{n} Esfuerzo\_real} \Big/ n}{n}$$

Esta medida penaliza con más rigor los errores relativos de los proyectos cuyo esfuerzo es superior a la media.

Además, también decidimos utilizar una segunda medida, propuesta también por Conte *et al.*, que cuantifica la *calidad de las predicciones*. Esta medida es un indicador del ajuste general del conjunto de puntos de una muestra y se basa en el MRE de cada punto; es decir, examina la frecuencia de MRE dado un nivel de error específico. Así, sea *k* el número de proyectos de un conjunto total de *n* cuyo $MRE < l$. Esta medida se define como:

$$Pred(l) = \frac{k}{n}$$

Considerando que MMRE y Pred son dos medidas que van ligadas, Conte *et al.* proponen unos umbrales de cumplimiento simultáneos para ambas:

- Capacidad Predictiva BUENA → MMRE ≤ 0,25 y Pred(0,25) ≥ 0,75
- Capacidad Predictiva ACEPTABLE → MMRE ≤ 0,30 y Pred(0,30) ≥ 0,70
- Capacidad Predictiva MALA → MMRE > 0,30 y Pred(0,30) < 0,70

Estos umbrales son ciertamente arbitrarios y aunque su uso está ampliamente extendido en la literatura especializada, existen autores que justifican una utilización algo más flexible de dichos umbrales. Así, por ejemplo, MacDonell [MacD96] apunta que las expectativas actuales más realistas para un modelo *bueno* son MMRE ≤ 0,30 y Pred(0,30) ≥ 0,70. Además, se admite una cierta flexibilidad considerando la fase del ciclo de vida en la que se pretenda utilizar el modelo.

5.2.3.2 La Consistencia

Un modelo que es sensible a la influencia de varios factores de productividad puede estimar el esfuerzo tanto hacia arriba como hacia abajo de manera consistente si el índice de productividad estándar asumido es significativamente distinto al del entorno en el que se desarrolla el proyecto objetivo. Una medida de la sensibilidad de un modelo a los factores relacionados con el desarrollo, que no es dependiente del índice de productividad base, es la *correlación* entre los valores de las estimaciones y los valores reales.

Si se detecta una alta correlación (cercana a 1), las estimaciones de los proyectos grandes indicarán un esfuerzo mayor que en el caso de proyectos pequeños. Una correlación con valor 0 implica la ausencia de cualquier relación lineal entre los valores reales y los estimados. Obviamente, un modelo de estimación consistente debería exhibir una alta correlación.

Para calcular un coeficiente de correlación, la primera disyuntiva que nos planteamos fue si utilizar un *método paramétrico* o uno *no paramétrico*. Como hemos indicado en la sección 2.5.2, siguiendo la postura de Fenton [Fent96] y Zuse [Zuse98], al no poder hacer supuestos sobre la distribución de nuestra muestra y disponer de una medida de escala aparentemente ordinal (nótese que en la seccion 3.4.5.2 demostramos que ID cumple axiomas que la caracterizan como una medida por encima de la escala ordinal, pero no de intervalo), debíamos decantarnos por los métodos no paramétricos, como el *Coeficiente de Correlación de Spearman*. Sin embargo, por otro lado nos encontramos con la postura de Briand, El Emam y Morasca [Bria95, Bria96a] que defienden un uso más pragmático de los metodos paramétricos. Se apoyan en otras disciplinas en las que no se han puesto limitaciones excesivamente rigurosas a los métodos estadísticos. Así, señalan que aunque los paramétricos son métodos más robustos que los no paramétricos, su uso también es más arriesgado si no se conoce el tipo de escala de la medida que se vaya a utilizar. Pero por otra parte, también señalan que porque una medida no sea exactamente de escala intervalo, tampoco tiene por qué ser meramente ordinal, como es nuestro caso, y que tratándola como ordinal, podríamos no considerar información relevante en el análisis de los datos. La conclusión a la que llegan estos autores es que muchos métodos paramétricos (como el *Coeficiente de Correlacion de Pearson*) son suficientemente robustos como para ser utilizados con medidas que no son exactamente de escala intervalo, conclusión derivada de las palabras de Labovitz:

> «Although some small error may accompany the treatment of ordinal variables as interval, this is offset by the use of more powerful, more sensitive, better developed and more clearly interpretable statistics with known sampling errors.» [Labo70]

Por todo ello, decidimos que obtendríamos tanto el *Coeficiente de Correlacion de Pearson* como el de *Spearman*, con vistas a comparar resultados [Cont86].

También decidimos que calcularíamos R^2, el *Coeficiente de Determinación*, que es una medida de la proporción de la variación en la variable dependiente que se explica por la variación de las variables independientes; es decir, es una medida de la bondad del ajuste, de lo bien que la regresión estimada se ajusta a los datos. R^2 debe superar el umbral de 0,5. (El uso de este coeficiente también tiene sus detractores, como [Loka96], que afirma que su uso no es correcto para valorar un modelo de estimación porque únicamente depende de los datos de la muestra; por lo tanto, aunque es un indicador de la adecuación de los datos a su uso en un análisis de regresión lineal, no tiene que ver con el modelo particular que se utilice para ajustar los datos).

5.2.4 Evaluación de los resultados

Una vez recogidos los datos de la muestra y seleccionados los métodos estadísticos a utilizar, procedimos a realizar el proceso de estimación siguiendo el metodo ESTIMAN, con el fin de evaluar los resultados.

Generalmente, cuando se debe evaluar un modelo que consta de una base de datos de muestra, el procedimiento típico consiste en la división de dicha muestra en dos partes: la principal que sirve de base de datos y la que sirve de banco de pruebas independiente. El inconveniente de esta técnica es que resulta poco significativa si el tamaño de la muestra es pequeño. En nuestro caso, con una muestra de 13 elementos, vimos que necesitábamos algún procedimiento distinto.

Por ello, nos decidimos a aplicar el método *Jackknife* [Mill74], que consiste en realizar el proceso de estimación por cada elemento de la muestra considerada, eliminando dicho elemento y utilizando el resto de la muestra como banco de datos. La ventaja de este procedimiento radica en que se elimina la influencia del elemento sobre el que se está realizando la estimación, pero sin necesidad de asignar un número elevado de elementos al conjunto que se utiliza para la prueba.

Así, se puede obtener una idea muy aproximada del comportamiento que el modelo tendría si se aplicara a un conjunto de datos independiente, pero sin la necesidad de sacrificar los elementos de la muestra que se deben usar como banco de pruebas.

En primer lugar, llevamos a cabo el proceso de estimación completo utilizando en la selección de casos similares un umbral de similitud $\theta = 70$, tal y como se describe en la sección 4.3.3. En el Apéndice D adjuntamos dos ejemplos de estimación completos, en los que se puede observar gráficamente una nube de puntos (proyectos similares en un 70%) alrededor del origen, que es el proyecto objetivo.

Los resultados que obtuvimos en este proceso se muestran en la Tabla 5.1.

Como puede observarse, los resultados no fueron muy alentadores. El modelo tiene una capacidad predictiva algo pobre (MMRE = 0,48) pero es capaz de estimar el 69 % de los proyectos por debajo de este error (Pred(0,48) = 0,69). En cuanto a las medidas ponderadas, WMMRE es un poco mejor, pero baja Pred-W.

El Coeficiente de Correlación de Pearson es $r = 0{,}6641$ con un nivel de significación de $\alpha = 0{,}02$ y R^2 es inferior a 0,5. Los resultados utilizando r_s de Spearman son muy malos.

Nombre	Esf. Real	Esf. Estim.	MRE	WMRE
Auzonet	44	25,3391	0,42	0,64
Microsoft	41	29,0456	0,29	0,41
Corba	19	18,0462	0,05	0,03
Pediatra	56	44,0000	0,21	0,41
Tcpdump	52	31,4400	0,40	0,70
Curplan	13	19,5119	0,50	0,22
Curedi	52	26,0911	0,50	0,89
Poeg	19	26,1438	0,38	0,24
Red	24	17,4884	0,27	0,22
Mapinfo	10	23,6760	1,37	0,47
Udcomp	13	28,9167	1,22	0,54
Mapa	24	27,0135	0,13	0,10
Interbanc	13	18,6934	0,44	0,19
MEDIA	29,2308	25,8004	0,48	0,39

MMRE	PRED
0,48	0,69

WMMRE	PRED-W
0,39	0,46

Pearson	R^2
0,6641	0,4410
α = 0,02	

Spearman	R^2
0,2295	0,0527

Tabla 5.1 Resultados estadísticos para un umbral de similitud $\theta = 70$.

A la vista de los resultados, nos decidimos a replicar el proceso de estimación variando los umbrales de similitud θ, ya que como se indica en la sección 4.3.3 intuíamos que las características de los proyectos podían llevar a una mejora en los resultados variando dicho umbral. Además, esta idea venía avalada por el hecho de que en las representaciones gráficas del proceso de estimación anterior habíamos descubierto una especie de *cluster*, es decir, un grupo de proyectos muy similares que se apoyaban los unos a los otros durante el proceso.

Los resultados para un umbral $\theta = 80$ son los que mostramos en la Tabla 5.2.

En este caso, los resultados tampoco fueron muy alentadores. El modelo tiene una capacidad predictiva algo pobre (MMRE = 0,50) pero es capaz de estimar más proyectos por debajo del error que el anterior, el 85% (Pred(0,50) = 0,85). En cuanto a las medidas ponderadas, WMMRE se mantiene con el mismo valor 0,39, y Pred-W mejora con un valor de 0,54.

El Coeficiente de Correlación de Pearson es $r = 0{,}6398$ con un nivel de significación de $\alpha = 0{,}02$ y R^2 sigue siendo inferior a 0,5. Los resultados utilizando r_s de Spearman también son muy malos.

Replicamos el experimento para un umbral de similitud del 85%, siendo los resultados los expuestos en la Tabla 5.3.

Estos resultado nos resultaron bastante sorprendentes por la mejora tan drástica que suponían con respecto al caso anterior.

Nombre	Esf. Real	Esf. Estim.	MRE	WMRE
Auzonet	44	25,5605	0,42	0,63
Microsoft	41	29,7935	0,27	0,38
Corba	19	18,1247	0,05	0,03
Pediatra	56	44,0000	0,21	0,41
Tcpdump	52	31,4400	0,40	0,70
Curplan	13	19,1234	0,47	0,21
Curedi	52	31,6750	0,39	0,70
Poeg	19	13,0000	0,32	0,21
Red	24	13,7192	0,43	0,35
Mapinfo	10	23,1411	1,31	0,45
Udcomp	13	35,2676	1,71	0,76
Mapa	24	26,5499	0,11	0,09
Interbanc	13	18,2677	0,41	0,18
MEDIA	29,2308	25,3587	0,50	0,39

MMRE	PRED
0,50	0,85

WMMRE	PRED-W
0,39	0,54

Pearson	R^2
0,6398	0,4093
α = 0,02	

Spearman	R^2
0,1841	0,0339

Tabla 5.2 Resultados estadísticos para un umbral de similitud $\theta = 80$.

Como se puede ver en la Tabla 5.3, obtuvimos un MMRE = 0,40, que es bastante aceptable considerando que nuestros datos provienen de una fase muy inicial en el ciclo de vida del desarrollo, además de poder producir una estimación por debajo del error para el 73% de los proyectos. Con respecto a las medidas ponderadas, obtuvimos WMMRE = 0,33 y Pred-W(0,33) = 0,64.

Nombre	Esf. Real	Esf. Estim.	MRE	WMRE
Auzonet	44	24,5102	0,44	0,62
Microsoft	41	29,7935	0,27	0,35
Corba	19	17,3856	0,08	0,05
Pediatra	56	44,0000	0,21	0,38
Tcpdump	52	31,4400	0,40	0,65
Curplan	13	19,1234	0,47	0,19
Curedi	52	41,0000	0,21	0,35
Red	24	13,7192	0,43	0,32
Mapinfo	10	23,1411	1,31	0,42
Mapa	24	26,5499	0,11	0,08
Interbanc	13	18,2677	0,41	0,17
MEDIA	31,6364	26,2664	0,40	0,33

MMRE	PRED
0,40	0,73

WMMRE	PRED-W
0,33	0,64

Pearson	R^2
0,8246	0,6799
α = 0,01	

Spearman	R^2
0,7705	0,5936
α = 0,002	

Tabla 5.3 Resultados estadísticos para un umbral de similitud $\theta = 85$.

El Coeficiente de Correlación de Pearson fue r = 0,8246 con un nivel de significación de $\alpha = 0,01$ y $R^2 = 0,6799$ que es aceptable. Los resultados utilizando r_s de Spearman por primera vez fueron buenos. Obtuvimos un coeficiente $r_s = 0,7705$ con un nivel de significación $\alpha = 0,002$ y $R^2 = 0,5936$.

Sin embargo, aumentar el umbral de similitud al 85% también tuvo una consecuencia: no se pudieron realizar estimaciones para dos proyectos por no encontrar ninguno que superara dicho umbral (los más cercanos a cada uno de ellos estaban al 83,83%). Por lo tanto, sólo se pudieron estimar 11 proyectos de un total de 13.

Para comprobar los resultados refinando más el umbral, aplicamos el proceso utilizando un umbral de similitud del 88%, cuyos resultados se muestran en la Tabla 5.4. Obviamente, en este caso tampoco se pudieron realizar estimaciones para todos los proyectos, pero únicamente fallaron los dos casos del experimento anterior

Los resultados fueron algo mejores con respecto a MMRE (MMRE = 0,34), pero empeoraron ligeramente con respecto a Pred(0,34), bajando el número de proyectos a estimar por debajo del error al 64%.

Con respecto a las medidas ponderadas, obtuvimos WMMRE = 0,29 y Pred-W(0,29) = 0,55, con lo que se intuye que la tendencia de MMRE y WMMRE con respecto a Pred es similar.

Sin embargo, tanto el Coeficiente de Correlación de Pearson como el de Spearman mejoraron, siendo r = 0,8446 con un nivel de significacion de $\alpha = 0,01$ y R^2 = 0,7133 y $r_s = 0,7750$ con un nivel de significación $\alpha = 0,002$ y R^2 = 0,6006

Nombre	*Esf. Real*	*Esf. Estim.*	*MRE*	*WMRE*
Auzonet	44	26,1379	0,41	0,56
Microsoft	41	38,1897	0,07	0,09
Corba	19	17,1040	0,10	0,06
Pediatra	56	44,0000	0,21	0,38
Tcpdump	52	31,4400	0,40	0,65
Curplan	13	17,3289	0,33	0,14
Curedi	52	41,0000	0,21	0,35
Red	24	14,1680	0,41	0,31
Mapinfo	10	22,5250	1,25	0,40
Mapa	24	28,2239	0,18	0,13
Interbanc	13	15,5808	0,20	0,08
MEDIA	31,6364	26,8817	0,34	0,29

MMRE	*PRED*
0,34	*0,64*

WMMRE	*PRED-W*
0,29	*0,55*

Pearson	R^2
0,8446	*0,7133*
$\alpha = 0,01$	

Spearman	R^2
0,7750	*0,6006*
$\alpha = 0,002$	

Tabla 5.4 Resultados estadísticos para un umbral de similitud $\theta = 88$.

En la Tabla 5.5, mostramos un resumen estadístico de los resultados obtenidos variando los distintos umbrales.

UMBRAL	MMRE	PRED-M	PEARSON	R^2	SPEARMAN	R^2	MUESTRA
70,00%	0,48	0,69	0,6641	0,4410	0,2295	0,0527	N = 13
80,00%	0,50	0,85	0,6398	0,4093	0,1841	0,0339	N = 13
85,00%	*0,40*	*0,73*	*0,8246*	*0,6799*	*0,7705*	*0,5936*	*N = 11*
88,00%	*0,34*	*0,64*	*0,8446*	*0,7133*	*0,7750*	*0,6006*	*N = 11*
Mejor	0,35	0,69	0,8415	0,7082	0,7898	0,6238	N = 13

Tabla 5.5 Resumen estadístico según los umbrales de similitud.

Como puede observarse en la Tabla 5.5, obtuvimos los mejores resultados considerando como umbrales de similitud $\theta = 85$ y $\theta = 88$, como suponíamos que iba a ocurrir al haber detectado la existencia de un *cluster* de proyectos similares. Con respecto a la elección de un umbral concreto, nos decidimos por $\theta = 85$, considerando que la capacidad predictiva es suficiente y permite estimar más proyectos por debajo del error que $\theta = 88$. Además, el nivel de consistencia está asegurado y la reducción en el tamaño de la muestra es la misma (11 elementos).

Además, nótese que en esta misma tabla adjuntamos un umbral adicional, el que denominamos el *mejor*. Replicamos el proceso de estimación completo pero sin marcar un umbral único para toda la muestra y tomando el mejor caso posible en la estimación de cada proyecto. Éramos perfectamente conscientes de que este tipo de proceso no era de mucha utilidad práctica, pero nos interesaba conocer los resultados de capacidad predictiva y consistencia, con el fin de poder establecer comparaciones con los resultados de los umbrales fijos.

Como puede verse en la Tabla 5.5, los resultados no fueron significativamente mejores que utilizando un umbral como $\theta = 85$ o $\theta = 88$, por lo que nuestra conclusión es que fijar un umbral puede proporcionar resultados suficientemente buenos.

De manera adicional y para completar la validación empírica, nos planteamos repetir todo el experimento, pero esta vez considerando en el proceso de estimación el uso de únicamente *una característica* en el cálculo del índice de similitud (véase la sección 4.3.3). Nuestra hipótesis era que la estimación es más precisa cuanto más similares sean los proyectos en función de unas características, siempre y cuando, obviamente, estas características estén correctamente seleccionadas y sean significativas con respecto al uso que se vaya a hacer de ellas. Nuestras características en el experimento anterior habían sido el número de clases del software y la complejidad media de las clases del dominio, ID_{media}.

Por lo tanto, decidimos probar si ID_{media} ayudaba significativamente a la precisión de las estimaciones, replicando todos los cálculos pero utilizando como característica en el índice de similitud sólo el número de clases del software.

En el Apéndice D, adjuntamos las tablas de resultados para la capacidad predictiva y la consistencia de los procesos de estimación utilizando como umbrales de similitud θ = 70, 80, 85, y 88. En la Tabla 5.6, se muestra el resumen de dichos resultados.

UMBRAL	MMRE	PRED-M	WMMRE	PRED-W	PEARSON	R^2
70,00%	0,56	0,77	0,43	0,46	0,4981	0,2967
80,00%	0,54	0,69	0,41	0,54	0,5447	0,2967
85,00%	0,57	0,75	0,45	0,58	0,4273	0,1826
88,00%	0,55	0,75	0,44	0,62	0,4201	0,1765

Tabla 5.6 Resumen estadístico según los umbrales de similitud para una variable.

En la Tabla 5.6 se observa que los resultados son peores con respecto a la capacidad predictiva, ya que MMRE proporciona valores más altos. En cuanto al número de proyectos que se estiman por debajo de MMRE, los resultados son aceptables, ya que rondan y superan el 0,70. Sin embargo, con respecto a la consistencia, los resultados son definitivamente malos.

En resumen, podemos concluir que nuestra validación parcial en el entorno académico fue satisfactoria en sus dos vertientes:

1. Validar el método de estimación ESTIMAN (a excepción del proceso de ajuste de no correspondencias) y probar que su capacidad predictiva y su consistencia son aceptables, considerando que se puede utilizar en fases iniciales del ciclo de vida de un proyecto.

2. Validar empíricamente nuestra medida de complejidad ID_{media}, ya que comprobamos que mejoraba significativamente los resultados del proceso de estimación.

Y para terminar, unas breves palabras atribuidas a Isaac D'Israeli:

«There are three kind of lies: lies, damned lies and statistics»

6. CONCLUSIONES

«As new software architectures reusable components, development processes and support tools are developed, new estimation methods will continue to appear... These actions will enable the organization to use the best known software estimation practices to accurately cost and intelligently manage software projects during the remainder of the 90s and into the next century»

Richard D. Stutzke, 1996

6.1 APORTACIONES

A lo largo de la presente memoria, hemos ido indicando las conclusiones a las que hemos llegado en cada paso del desarrollo de nuestro modelo de estimación. A continuación, pasamos a resumirlas y destacar las aportaciones novedosas del mismo.

Tras revisar de manera exhaustiva los distintos métodos de validación de medidas propuestos en la literatura y analizar sus distintas vertientes, así como la valoración de las mismas: aciertos, omisiones e inconvenientes de cada una de ellas, hemos llegado a las siguientes conclusiones:

- Se han descrito gran cantidad de medidas pero sin fijar un objetivo específico en su definición como puede ser la predicción, la estimación, la valoración de indicadores de calidad, etc. En consecuencia, abundan las medidas cuya utilidad es discutible.

- Se han definido gran cantidad de medidas de manera imprecisa; es decir, el vocabulario utilizado en su definición admite varias interpretaciones. Esto hace imposible poder calcular las medidas de manera unificada.

- Hay una gran cantidad de medidas sin un modelo formal de soporte, por tanto, no se puede tener la certeza de que los números obtenidos sean representativos de las características que se pretenden cuantificar.

- Las validaciones empíricas son insuficientes para tener la certeza de que una medida determinada es útil.

Para evitar los defectos mencionados y como base para diseñar la estrategia de validación que aportamos, hemos realizado un estudio riguroso de las tendencias actuales en la validación de medidas, tanto en su vertiente teórica como en la empírica, considerando la Teoría de la Medición y las propiedades de Weyuker, Zuse, Briand *et al.* y Whitmire.

La necesidad de definir de manera rigurosa y precisa el planteamiento del problema abordado en esta tesis doctoral, nos ha llevado a utilizar una expansión del marco de referencia GQM que permite definir las medidas propuestas de manera significativa con respecto a unos objetivos, unos supuestos, unas propiedades axiomáticas, una representación abstracta del producto y una validación experimental; es decir, establecer una estrategia completa para la consecución del proyecto.

Nuestro objetivo ha sido realizar de forma precisa una estimación inicial de esfuerzo en personas-semana, analizando el Modelo del Dominio que se obtiene como resultado de la fase de Análisis del Dominio usando el Paradigma Orientado a Objetos como estrategia de desarrollo.

El supuesto principal, hilo conductor de toda la investigación, es la idea de que el esfuerzo de desarrollo está relacionado con el tamaño del modelo del dominio, pero no únicamente con este parámetro. En consecuencia, hemos partido de la hipótesis de que dos productos de software de tamaños iguales pueden llevar a esfuerzos de desarrollo completamente distintos debido a la complejidad inherente de cada uno de ellos. Es decir, que el esfuerzo de desarrollo está relacionado con el tamaño y la complejidad del modelo del dominio, de manera que cuanto mayor es dicho dominio o mayor es la complejidad, mayor es el esfuerzo de desarrollo. En consecuencia, el modelo de estimación propuesto debe basarse en una medida de tamaño y otra de complejidad. A esto hay que añadir que el esfuerzo está condicionado por un conjunto de factores de contexto del proyecto y que por tanto, también deben ser considerados en el modelo.

Para medir las propiedades estáticas de un diseño orientado a objetos de forma precisa, se ha utilizado una formalización para sistemas orientados a objetos, basada en el modelo de Whitmire y en la Teoría de las Categorías. Dado que el modelo de Whitmire utiliza como representación gráfica un grafo sencillo, el modelo en sí es difícil de automatizar y considerando que para nuestro propósito es suficiente un modelo del dominio en forma de diagrama de clases, hemos

seleccionado la notación UML para nuestras abstracciones, previa comprobación de que no hay pérdida de significado al utilizarla.

La medida de tamaño seleccionada se basa en el número de clases a desarrollar y se fundamenta en el número de clases del dominio. Esta medida, validada teóricamente a la luz de las propiedades axiomáticas propuestas por Briand *et al* y Whitmire, se comporta de forma adecuada, cumpliendo todas las propiedades propuestas. Sin embargo, desde el punto de vista del tipo de escala y siguiendo la Teoría de la Medición y los requisitos de Zuse, no podemos garantizar que la medida sea de escala ratio. Siempre será de intervalo y únicamente será de escala ratio si tenemos la certeza absoluta de que en el hipotético caso de tener que fusionar dos modelos del dominio, estos van a ser disjuntos; es decir, no van a compartir ninguna clase.

La medida de complejidad propuesta es una de las aportaciones de esta tesis doctoral. Nuestra medida, denominada *ID*, está orientada a captar la complejidad cognitiva o psicológica, como un factor que afecta al esfuerzo de desarrollo. Para ello se centra principalmente en la *Interconectividad*, en las conexiones que las distintas clases de un dominio deben establecer para poder cumplir con sus responsabilidades. Con fines predictivos y en un intento de neutralizar el efecto que el tamaño puede tener sobre la medida, hemos propuesto su normalización, utilizando de esta manera para la estimación del esfuerzo el resultado de calcular la media de *ID*. Por otro lado, la definición y utilización de esta medida carecerá de ambiguedades, debido a la especificación formal realizada.

Con el fin de comprobar si la medida muestra la tendencia de las llamadas *buenas prácticas* de diseño, hemos documentado su comportamiento en función de las distintas operaciones que se pueden dar en un modelo, como la partición y combinación directa de clases, la adición de colaboradores, etc., siendo los resultados obtenidos positivos. Además, también hemos realizado la validacion teórica de *ID*, a través de las propiedades de Weyuker y Whitmire.

Por otra parte, la valoración del tipo de escala, siguiendo las directrices propuestas por Zuse para las medidas orientadas a objetos, indica que nuestra medida cumple con la Función de Confianza Modificada y con los axiomas de la Relación de Confianza Modificada. En consecuencia, concluimos que *ID* no es meramente de escala ordinal, sino que está por encima de ésta, pero sin ser de escala de ratio, ya que no cumple con los axiomas de la Estructura Extensiva.

La aportación principal de la presente investigación es el método de estimación por analogía ESTIMAN. Este método es una implementación del meta-

modelo de Cowderoy y Jenkins, que utiliza cinco pasos para el proceso de estimación: construcción y almacenamiento de la representación de los casos, la selección y recuperación de casos análogos, la transferencia de la solución, el análisis de no correspondencias y el proceso de ajuste de las mismas.

Nuestra representación de los casos consiste en la captura de las medidas de tamaño y complejidad, así como de los factores de escala y conductores de coste del modelo COCOMO II y del valor del esfuerzo de desarrollo en el caso de los proyectos terminados que van a poblar el repositorio, siendo ésta la información necesaria para caracterizar a un proyecto de desarrollo con fines predictivos.

Para realizar el proceso de selección de casos análogos, hemos propuesto una heurística en base a un *índice de similitud* y un *umbral de selección* θ predefinido. El índice de similitud se calcula como la distancia euclídea en un espacio bidimensional, donde las dos dimensiones representadas son el tamaño y la complejidad de los proyectos; en otras palabras, consideramos la similitud de dos proyectos en función de la distancia a la que se encuentran, tomando en consideración dos variables, el tamaño y la complejidad. Así, en el proceso de recuperación se valora la distancia entre el proyecto objetivo y los proyectos del repositorio y se seleccionan únicamente aquellos proyectos cuya similitud supera un determinado umbral θ.

El proceso de transferencia transvasa las características de los proyectos a la memoria de trabajo con el fin de analizar aquellas que no se corresponden entre sí. Así, se activa el proceso de ajuste de no correspondencias que toma en consideración el contexto de los proyectos como factores de escala y conductores de coste. El resultado de este proceso es el cálculo de la estimación del esfuerzo de desarrollo para el proyecto objetivo, en función del esfuerzo de desarrollo aportado por los casos similares seleccionados del repositorio, ponderando cada valor de esfuerzo en base al índice de similitud; de esta forma, los proyectos más parecidos tendrán más influencia en el cálculo.

La característica fundamental del método ESTIMAN es su carácter genérico, ya que los cinco pasos comentados forman un núcleo estable que es aplicable a un número variable de parámetros, tanto de características de selección y adaptación, como de estimadores.

La validación de este método, que conlleva la de la medida de complejidad *ID*, se ha realizado en un entorno académico siguiendo las directrices de Schneiderwind y el IEEE, que indican cómo valorar la capacidad predictiva y la consistencia, mediante la Media del Error Relativo Absoluto (MMRE y Pred) y el Coeficiente de Correlación de Pearson y de Spearman y el Coeficiente de Determinación (R^2), respectivamente.

Los resultados obtenidos son alentadores. El método tiene una capacidad predictiva y una consistencia aceptables según los criterios de Conte *et al* y MacDonell [Cont86, MacD96], sobre todo teniendo en cuenta que se puede utilizar en las fases iniciales de un proyecto de desarrollo, cuando la información de la que se dispone es generalmente insuficiente, imprecisa y ambigua. Además, los resultados también han señalado nuestra medida de complejidad como favorecedora de la precisión de las estimaciones.

6.2 FUTURAS LINEAS DE INVESTIGACION

La continuación del trabajo expuesto en esta memoria podría orientarse hacia la validación empírica de las medidas y el método de estimación ESTIMAN en su vertiente industrial, lo que sería de gran importancia, puesto que permitiría demostrar si dicho método mantiene una capacidad predictiva adecuada y una precisión aceptable utilizando las medidas propuestas, en un entorno tan distinto.

Además, durante la etapa de validación industrial cabría la posibilidad de utilizar el componente de estrategia de ajuste de no correspondencias, no utilizado en la validación realizada debido a la uniformidad de la muestra. Nuestra propuesta considera el uso del conjunto completo de factores de escala y conductores de coste del Modelo Post-Arquitectural de COCOMO II. Uno de los inconvenientes de esta opción es el elevado número de elementos que se deben tomar en cuenta, al ser cinco los factores de escala y diecisiete los conductores de coste. Una posible solución, aunque parcial, a este problema es el uso de los conductores de coste del Modelo de Diseño Inicial de COCOMO II; aunque los factores de escala siguen siendo cinco, los conductores quedan reducidos a siete. De todos modos, consideramos que dicha solución es parcial porque en realidad, no ahorra el esfuerzo de evaluación de factores, debido a que los cinco conductores del Modelo de Diseño Inicial son una combinación estática de los diecisiete conductores del Modelo Post-Arquitectural.

Por lo anteriormente expuesto, sería muy recomendable la futura investigación de una solución alternativa. El objetivo es aislar los factores que afectan al esfuerzo de desarrollo de un proyecto pero consiguiendo reducir el número de dichos factores, de manera que se puedan seleccionar aquellos que tengan un mayor impacto sobre el esfuerzo. Subramanian y Breslawski [Subr93] describen el procedimiento completo para realizar el estudio estadístico que permite agrupar las distintas variables en *macro-factores*, que en realidad son áreas genéricas de impacto sobre el esfuerzo de desarrollo; de este modo, se puede obtener la variable concreta que posea una mayor correlación con el macro-factor en estudio, consiguiendo así, un espacio de factores realmente reducido, al

discriminar todas aquellas variables cuyo impacto sea de escasa importancia. También cabe tomar en consideración el procedimiento propuesto por Kitchenham [Kitc98], cuyo objetivo es el mismo que el de Subramanian *et al.* pero en base a métodos estadísticos diferentes, ya que utiliza el análisis de varianzas. De este modo, además de permitir simplificar el modelo de estimación, se favorecería la portabilidad del método, es decir, su utilización fiable en otras instalaciones, ya que en realidad, los procedimientos mencionados anteriormente son una forma de calibración del modelo de estimación.

Otra posible línea de investigación consistiría en estudiar el máximo aprovechamiento del carácter genérico del método ESTIMAN. Como se ha señalado, el método de estimación propuesto es *parametrizable*, en el sentido de que posee un mecanismo general de actuación, basado en el Razonamiento Analógico, pero que para su correcto funcionamiento, necesita que las características en base a las que se seleccionan los casos similares, capten propiedades esenciales vinculadas al objetivo de la estimación. Utilizando el núcleo básico que almacena el conocimiento para la recuperación analógica, podemos estimar cualquier propiedad en función de cualquier característica.

Esto permite establecer distintas líneas de acción, como por ejemplo:

- El análisis del impacto de distintos conjuntos de medidas, propuestos en la literatura especializada, sobre el esfuerzo de desarrollo

- La definición de un conjunto de medidas adicionales que capten distintas propiedades de un sistema orientado a objetos y el estudio de su capacidad predictiva con respecto a características como el tamaño del producto, el esfuerzo, el tiempo o el coste de desarrollo, la complejidad o el número de defectos del producto.

- La captura de las características fundamentales de cualquier paradigma de desarrollo con fines de estimación de propiedades. Así por ejemplo, se puede considerar el número de controles en una ventana como una característica que afecta a la complejidad de un producto en el paradigma de Programación Visual.

Esta última vía de investigación es especialmente relevante, puesto que deja una puerta abierta a cualquier cambio de paradigma al que tengamos que enfrentarnos en años venideros.

Al disponer de un método genérico parametrizable, la labor ante una nueva filosofía de desarrollo consistirá en captar, desarrollar y elaborar las medidas que

engloben las propiedades que caractericen al producto a desarrollar con el objetivo de estimar el concepto deseado: tamaño, esfuerzo, complejidad o lo que se considere oportuno.

Para terminar dando apoyo a esta última idea, unas palabras de Donald J Reifer, extraídas de la *Encyclopedia of Software Engineering*

> «We have introduced you to the topic of software estimation. We have discussed why cost estimating is important, what is involved, when it is done, who does it and how it is done in practice. The central theme is that the software community has made significant progress in the field of estimating over the past decade. The underlying message is that the industry needs to continue with its research as *it undertakes yet another paradigm change*.» [Reif94 219]

BIBLIOGRAFIA

[Abbo94] Abbot, D.H. & Korson, T.D. & McGregor, J.D., *A Proposed Design Complexity Metric for Object Oriented Development*. Technical Report 94-104, Universidad de Clemson, Junio 1994.

[Abde86] Abdel-Hamid, T.K. & Madnick, S.E., 'Impact of Schedule Estimation on Software Project Behaviour'. *IEEE Software*, Julio 1986, pg: 70-75.

[Abde91] Abdel-Hamid, T.K. & Madnick, S.E., *Software Project Dynamics: an Integrated Approach*. Prentice Hall, Eaglewood Cliffs, NJ, 1991.

[Abde91] Abdel-Hamid, T.K. & Madnick, S.E., 'Adapting, Correcting and Perfecting Software Estimates: a Maintenance Metaphore'. *IEEE Computer*, Marzo 1993, pg: 20-29.

[Albr79] Albrecht, A.J., 'Measuring Applications Development Productivity'. *Proc. of IBM Applications Development Joint SHARE/GUIDE Symposium*, Monterey, California, 1979, pg:83-92.

[Albr83] Albrecht, A.J. & Gaffney,, S.H., 'Software Function, Source Lines of Code and Development Effort Prediction: a Software Science Validation'. *IEEE Transactions on Software Engineering*, Vol. 9, N° 6, Junio 1983, pg: 639-648.

[ANSI91] ANSI/ISO, *Quality Management and Quality Assurance Standards - Guidelines for the Application of ANSI/ISO/ASQC 9001 to the Development, Supply and Maintenance of Software*. ANSI/ISO/ASQC Q9000-3-1991, American National Standards Institute / International Organization for Standardization, 1991.

[Arch95] Archer, C. & Stinson, M., *Object-Oriented Software Measures*. CMU/SEI-95-TR-002, Software Engineering Institute, Pittsburg, PA, 1995.

[Bail81] Bailey, J.W. & Basili, V.R. 'A Meta-Model for Software Development Resource Expenditures'. *Proc. Of the 5th International Conference on Software Engineering*, 1981, pg: 107-116.

[Bald90] Balda, D.M. & Gustafson, D.A., 'Cost Estimation Models for the Reuse and Prototype Software Development Life-Cycles'. *ACM SIGSOFT Software Engineering Notes*, Vol. 15, N° 3, 1990, pg: 42-50.

[Bank79] Banker, A.L. & Zweben, S.H., 'The use of Software Science in Evaluating Modularity Concepts'. *IEEE Transactions on Software Engineering*, Vol. 5, N° 2, 1979, pg: 110-120.

[Bans97] Bansiya, J. & Davis, C., 'Automated Metrics and Object Oriented Development'. *Dr. Dobb's Journal*, Diciembre 1997, pg: 42-48.

[Barn93] Barnes, G.M. & Swim, B.R., 'Inheriting Software Metrics'. *Journal of Object Oriented Programming*, Vol. 6, N° 7, Noviembre 1993, pg: 27-34.

[Barr95] Barr, M. & Wells, C., *Category Theory for Computing Science*. Prentice Hall, Eaglewood Cliffs, NJ, 1995, ISBN: 0-13-323809-1.

[Basi83] Basili, V. & Hutchens, D., 'An Empirical Study of a Complexity Family'. *IEEE Transactions on Software Engineering*, Vol. 9, N° 6, Noviembre 1983, pg: 664-672.

[Basi88] Basili, V. & Rombach, H.D., 'The TAME Project: Towards Improvement-Oriented Software Environments'. *IEEE Transactions on Software Engineering*, Vol.14, N° 6, 1988, pg-758-771.

[Basi94] Basili, V. & Caldiera, G. & Rombach, H.D., 'The Goal Question Metric Approach'. *Encyclopedia of Software Engineering*. Ed. J.J. Marciniack, John Wiley & Sons, 1994, pg: 528-532.

[Basi96] Basili, V.R. & Briand, L. & Melo, W., 'A Validation of Object Oriented Design Metrics as Quality Indicators'. *IEEE Transactions on Software Engineering*, Vol. 22, N° 10, 1996, pg:751-761.

[Beck89] Beck, K. & Cunningham, W. 'A Laboratory for Teaching Object-Oriented Thinking'. *Proc. Of OOPSLA'89 Conference. ACM SIGPLAN Notices*, Vol. 24, N° 10, Octubre 1989.

[Bela79] Belady, L.A., 'On Software Complexity'. *Workshop on Quatitative Software Models of Reliability*. IEEE N° TH0067-9, Octubre 1979, pg: 90-94.

[Bela79a] Belady, L.A. & Evangelisti, C.J., *System Partitioning and its Measure*. IBM Res. Rep. RC7560, 1979.

[Bera93] Berard, E.V., *Essays on Object Oriented Software Engineering: Volumen 1*. Prentice Hall, Englewook Cliffs, NJ, 1993.

[Bera97] Berard, E.V., 'Metrics for Object Oriented Software Engineering'. *The Object Agency, Inc.* Documento Personal, dirección web http://www.toa.com.

[Berl80] Berilnger, E., 'An Information Theory Based Complexity Measure'. *Proc. of the National Computer Conference*, 1980, pg: 773-779.

[Biem94] Bieman, J. & Ott, L.M., 'Measuring Functional Cohesion'. *IEEE Transactions on Software Engineering*, Vol. 20, N° 8, Agosto 1994, pg: 644-658.

[Bind94] Binder, R.V., 'Design for Testability in Object Oriented Systems'. *Communications of the ACM*, Vol. 37, N° 9, Septiembre 1994, pg: 87-101.

[Bink96] Binkley, A.B. & Schach, S.R., 'Impediments to the Effective Use of Metrics within the Object Oriented Paradigm'. *Proc. of the OOPSLA'96 Conference*, Special Issue of *SIGPLAN Notices*, Septiembre 1996.

[Bisi95] Bisio, R. & Malabocchia, F., 'Cost Estimation of Software Projects through Case Base Reasoning'. *Proc. of the International Conference on Case-Based Reasoning*. Portugal, 1995, pg: 11-22.

[Bode98] Bode, J., 'Decision Support with Neural Networks in the Management of Research and Development: Concepts and Application to Cost Estimation'. *Information & Management*, Vol. 34, N°1, Agosto 1998, pg 34-40.

[Boeh81] Boehm, B.W. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[Boeh87] Boehm, B.W. & Royce, W. 'ADA COCOMO and the ADA Process Model'. *Proc. of the Third International COCOMO Users Meeting*, SEI, Pittsburg, PA, Noviembre 1987.

[Boeh88] Boehm, B.W., 'A Spiral Model for Software Development and Enhacement'. *IEEE Computer*, Vol. 21, N° 5, Mayo 1988, pg: 61-72.

[Boeh95] Boehm, B.W. & Clark, B. & Horowitz, E. & Westland, C. & Madachy, R. & Selby, R., 'Cost Models for Future Software Life Cycle Processes: COCOMO V2.0'. *Annals of Sofware Engineering Special Volumen on Software Process and Product Measurement*. Baltzer AG, Science Publishers, Amsterdam, Holanda, 1995.

[Boeh95a] Boehm, B.W. & Clark, B. & Horowitz, E. & Westland, C. & Madachy, R. & Selby, R., 'The COCOMO V2.0 Software Cost Estimation Model'. *Proc. of the International Society of Parametric Analysis*, 1995.

[Booc94] Booch, G., *Object Oriented Analysis and Design with Applications*. 2ª edición, Benjamin-Cummings, 1994.

[Booc96] Booch, G., *Análisis y Diseño Orientado a Objetos*. Addison-Wesley/Diaz de Santos, 1996, ISBN: 0-201-60122-2.

[Booc97] Booch, G. & Rumbaugh, J. & Jacobson, I. *The Unified Modeling Language for Object Oriented Software Development, V1.1*. Rational Software Corporation, Septiembre 1997. Dirección Web: http://www.rational.com.

[Bria92] Briand, L. & Basili, V. & Thomas, W., 'A Pattern Recognition Approach for Software Engineering Data Analysis'. *IEEE Transactions on Software Engineering*, Vol. 18, N° 11, Noviembre 1992, pg: 931-942.

[Bria94] Briand, L. & Morasca, S. & Basili, V.R., *Goal Driven Definition of Product Metrics Based on Properties*. Technical Report CS-TR-3346-1, Universidad de Maryland, Septiembre 1994.

[Bria94a] Briand, L. & Morasca, S. & Basili, V.R., *Defining and Validating High Level Design Metrics*. Technical Report CS-TR-3301-1, Universidad de Maryland, Junio 1994.

[Bria95] Briand, L. & El Emam, K. & Morasca, S., *Theoretical and Empirical Validation of Software Product Measures*. International Software Engineering Research Network Technical Report ISERN-95-03, 1995.

[Bria96] Briand, L. & Morasca, S. & Basili, V.R., 'Property-based Software Engineering Measurement'. *IEEE Transactions on Software Engineering*, Vol. 22, N° 1, Enero 1996, pg: 68-85.

[Bria96a] Briand, L. & El Emam, K. & Morasca, S., 'On the Application of Measurement Theory in Software Engineering'. *Empirical Software Engineering: a International Journal*, Vol. 1, N° 1, 1996.

[Bria96b] Briand, L. & Daly, J.W. & Wüst, J., *A Unified Framework for Coupling Measurement in Object Oriented Systems*. International Software Engineering Research Network Technical Report ISERN96-14, 1996.

[Bria97] Briand, L. & El Emam K. & Bomarius, F., *A Hybrid Method for Software Cost Estimation and Risk Assesment*. International Software Engineering Research Network Technical Report ISERN97-24, 1997.

[Bria97a] Briand, L. & Devanbu, P. & Melo, W., 'An investigation into Coupling Measures for C++'. *Proc. of the 19th International Conference on Software Engineering*, Boston MA, 1997.

[Bria97b] Briand, L. & Daly, J.W. & Wüst, J., 'A Unified Framework for Cohesion Measurement in Object Oriented Systems'. *Proc of the 4th International Software Metrics Symposium*, Alburquerque, IEEE Computer Society Press, 1997. Publicado en *Empirical Software Engineering*, Vol 3, N° 1, 1998, pg: 65.

[Bria98] Briand, L. & Devanbu, P. & Melo, W., *Quality Modeling based on Coupling Measures in a Commercial Object-Oriented System* International Software Engineering Research Network Technical Report ISERN98-01, 1998.

[Brit94] Brito e Abreu, F. & Carapuça, R., 'Candidate Metrics for Object Oriented Software Within a Taxonomy Framework'. *The Journal of Systems and Software*, Vol. 26, N° 1, Julio 1994.

[Brit94a] Brito e Abreu, F. & Carapuça, R., 'Object Oriented Software Engineering Measuring and Controlling the Development Process'. *Proc Of the 4th International Conference On Software Quality*, McLean, VA, Octubre 1994.

[Brit95] Brito e Abreu, F. & Goulão, M. & Esteves, R., 'Towards the Design Quality Evaluation of Object Oriented Software Systems'. *Proc. Of the 5th International Conference On Software Quality*, Austin, TX, Octubre 1995.

[Brit96] Brito e Abreu, F. & Melo, W., 'Evaluating the Impact of Object Oriented Design on Software Quality'. *Proc. Of the 3th International Software Metrics Symposium*, METRICS'96 Berlin, Alemania, Marzo 1996.

[Brit96a] Brito e Abreu, F. & Esteves, R. & Goulão, M., 'The Design of Eiffel Programs: Quantitative Evaluation using the MOOD Metrics' *Proc Of TOOLS'96 USA*, Santa Barbara, CA, Julio 1996.

[Broo74] Brooks, F.P., 'The Mithical Man-Month'. *Datamation*, Diciembre 1974, pg:44-52.

[Broo75] Brooks, F.P., *The Mithical Man-Month*. Addison Wesley, 1975.

[Broo95] Brooks, F.P., *The Mithical Man-Month: Essays on Software Engineering* Addison Wesley, 1995.

[Bung77] Bunge, M. , *Treatise on Basic Philosophy, Vol. 3: Ontology I The Furniture of the World*. Boston, MA: Reidel, 1977.

[Burg95] Burgett, J. & Ohman, B., 'Experiences with Object Based Software Development Effort Estimates and Associated Metrics'. *OOPSLA'95 Workshop on Object Oriented Process & Metrics for Effort Estimation*, Austin, TX, Octubre 1995, Addendeum a las actas, pg: 138-142.

[Byar94] Byard, C., 'Software Beans: Class metrics and the Mismeasure of Software'. *Journal of Object Oriented Programming*, Vol. 7, N° 5, Septiembre 1994, pg: 32-35.

[Cant94] Cant, S.N. & Henderson-Sellers, B. & Jeffery, D.R., 'Application of Cognitive Complexity Metrics to Object Oriented Programs'. *Journal of Object Oriented Programming*, Vol.7, N° 4, 1994, pg: 52-63.

[Card87] Card, D.N. & Agresti, W.W., 'Resolving the Software Science Anomaly'. *The Journal of Systems and Software*, Vol. 7, N° 1, Marzo 1987, pg: 29-35.

[Card88] Card, D.N. & Agresti, W.W., 'Measuring Software Design Complexity'. *The Journal of Systems and Software*, Vol. 8, N° 1, 1988, pg: 185-197.

[Carl92] Carleton, A. & Park, R. & Goethert, B. & Florac, A. & Bailey, E. & Pfleeger, S. , *Software Measurement for DOD Systems: Recommendations for Initial Core Measures*. CMU/SEI-92-TR-019, Software Engineering Institute, Pittsburg, PA, 1992.

[Chen93] Chen, J.Y. & Lu, J.F., 'A New Metric for Object-Oriented Design'. *Information & Software Technology*, Vol. 35, N° 4, 1993, pg: 232-240.

[Cher91] Cherniavsky, J.C. & Smith, C. H., 'On Weyuker's Axioms for Software Complexity Measures'. *IEEE Transactions on Software Engineering*, Vol. 17, N° 6, Junio 1991, pag: 636-638.

[Chid91] Chidamber, S.R. & Kemerer, C.F., 'Towards a Metric Suite for Object Oriented Design'. *Proc. of the OOPSLA'91 Conference*, Special Issue of *ACM SIGPLAN Notices*, Vol. 26, N° 10, Noviembre 1991, pg: 197-211.

[Chid94] Chidamber, S.R. & Kemerer, C.F., 'A Metric Suite for Object Oriented Design'. *IEEE Transactions on Software Engineering*, Vol. 20, N° 6, Junio 1994, pg: 476-493.

[Chid95] Chidamber, S.R. & Kemerer, C.F., 'Author's Reply'. *IEEE Transactions on Software Engineering*, Vol. 21, N° 3, Marzo 1995, pg: 265.

[Chid98] Chidamber, S.R. & Darcy, D.P. & Kemerer, C.F., 'Managerial Use of Metrics for Object Oriented Software: An Exploratory Analysis'. *IEEE Transactions on Software Engineering*, Vol. 24, N° 8, Agosto 1998, pg 629-639.

[Chur95] Churcher, N.I. & Shepperd, M.J., 'Towards a Conceptual Framework for Object Oriented Software Metrics'. *ACM Software Engineering Notes*, Vol. 20, N° 2, Abril 1995, pg: 69-76.

[Chur95a] Churcher, N.I. & Shepperd, M.J., 'Comments on «A Metrics Suite for Object Oriented Design»'. *IEEE Transactions on Software Engineering*, Vol. 21, N° 3, Marzo 1995, pg: 263-265.

[CMU96] Carnegie Mellon University, *Defining Software Measures Conference Handout V1.0*. Software Engineering Measurement and Analysis Project, CMU, 1996.

[Coad91] Coad, P. & Yourdon, E., *Object Oriented Design*. Prentice-Hall/Yourdon Press, Englewood Cliffs, NJ, 1991.

[Coad91a] Coad, P. & Yourdon, E., *Object Oriented Analysis*. 2ª Edición. Prentice-Hall/Yourdon Press, Englewood Cliffs, NJ, 1991.

[Cont86] Conte, S.D. & Dunsmore, H.E. & Shen, V.Y., *Software Engineering Metrics and Models*. Benjamin-Cummings Publishing Co., 1986.

[Cort98] Cortazar, R. & Barredo, M.A. & Zaballa, G. & Del Val, J.L., 'Ingeniería del Software III: Una asignatura de Ingeniería del Software Orientada a Objetos'. *Actas de las IV Jornadas sobre Enseñanza Universitaria de la Informática*, Andorra, Julio 98, pg: 139-146.

[Coul87] Coulter, N.S. & Cooper, R.B. & Solomon, M.K., 'Information-Theoretic Complexity of Program Specifications'. *The Computer Journal*, Vol. 30, N° 3, 1987, pg: 223-227.

[Cowd88] Cowderoy, A.J.C. & Jenkins, J.O., 'Cost-Estimation by Analogy as a Good Management Practice'. *Proc. Of Software Engineering 1988*, Liverpool IEE/BCS, Reino Unido.

[Curt80] Curtis, B., 'Measurement and Experimentation in Software Engineering'. *Proc of the IEEE*, Vol. 68, N° 9, Septiembre 1980, pg 1144-1157.

[Daic95] Daich, G.T. & Price, G. & Dawood, M., 'Metrics Tools: Size'. *CrossTalk the Defence Software Engineering Report*, Vol. 8, N° 4, Abril 1995.

[Davi88] Davis, J.S. & LeBlanc, R.J., 'A Study of the Applicability of Complexity Measures'. *IEEE Transactions on Software Engineering*, Vol. 14, N° 9, Septiembre 1988, pg: 1366-1371.

[Dean89] Dean, E.B., 'Parametric Cost Analysis: a Design Function', *Transactions of the American Association of Cost Engineers, 33ª Annual Meeting*, San Diego, CA, Junio 1989.

[DeMa82] DeMarco, T. *Controlling Software Projects. Management, Measurement and Estimation*. Yourdon Press, NY, 1982.

[DePa97] De Panfilis, S. & Kitchenham, B. & Morfuni, N., 'Experiences introducing a Measurement Program'. *Information & Software Technology*, Vol. 39, N° 11, 1997, pg: 745-754.

[Dumk95] Dumke, R.R., 'A Measurement Framework for Object Oriented Software Development'. *Annals on Software Engineering Journal*, Vol. 1, 1995.

[Duto98] Dutoit, A. H. & Bruegge, B., 'Communication Metrics for Software Development'. *IEEE Transactions on Software Engineering*, Vol. 24, N° 8, Agosto 1998, pg: 615-628.

[Eber97] Ebert, C. & Morschel, I., 'Metrics for Quality Analysis and Improvement of Object Oriented Software'. *Information & Software Technology*, Vol. 39, 1997, pg: 497-509.

[Emde71] Van Emden, M.H. *An Analysis of Complexity*. Mathematisches Zentrum, Amsterdam, 1971.

[Eric95] Erickson, D.R. & Steadman, A.T., 'Metrics Tools: Effort and Schedule'. *CrossTalk: the Defence Software Engineering Report*, Vol. 8, N° 3, Marzo 1995.

[Etzk98] Etzkorn, L. & Davis, C. & Li, W., 'A Practical Look at the Lack of Cohesion in Methods Metric' . *Journal of Object Oriented Programming*, Vol. 11, N° 5, Septiembre 1998, pg: 27-34.

[Fent91] Fenton, N.E., *Software Metrics: a Rigorous Approach*. International Thompson Computer Press, 1991, SBN: 1-85032-242-2.

[Fent94] Fenton, N.E., 'Software Measurement: a Necessary Scientific Basis'. *IEEE Transactions on Software Engineering*, Vol. 20, N° 3, Marzo 1994, pg: 199-206.

[Fent96] Fenton, N.E. & Pflegeer, S.L., *Software Metrics: a Rigorous & Practical Approach* International Thompson Computer Press, 1996. ISBN 1-85032-275-9.

[Fern97] Ferneley, E., 'An Empirical Study of Coupling and Control Flow Metrics' *Information & Software Technology*, Vol. 39, N° 13, 1997, pg 879-887

[Fetc95] Fetcke, T., 'Investigations on the Properties of Object-Oriented Software Metrics'. *Workshop on Quantitative Methods, European Conference on OO Programming, ECOOP'95*, Dinamarca, 1995

[Fitz78] Fitzsimmons, A. & Love, T., 'A Review and Evaluation of Software Science'. *Computing Surveys*, Vol. 10, N° 1, Marzo 1978, pg: 3-18.

[Frei79] Freiman, F.R. & Park, R.E., *PRICE-S Software Model Overview* Internal Paper, RCA, Cherry Hill, 1979.

[Gilb77] Gilb, T., *Software Metrics*. Winthrop Publishers, Cambridge, Massachusetts, 1977.

[Gile95] Giles, A.E. & Daich, G.T., 'Metric Tools'. *CrossTalk: the Defence Software Engineering Report*, Vol. 8, N° 2, Febrero 1995.

[Gile95a] Giles, A.E. & Barney, D., 'Metric Tools: Software Cost Estimation' *CrossTalk: the Defence Software Engineering Report*, Vol. 8, N° 6, Junio 1995.

[Gill98] Gillibrand, D. & Liu, K., 'Quality Metrics for Object Oriented Design' *Journal of Object Oriented Programming*, Vol. 10, N° 8, Enero 1998, pg 56-59.

[Goet92] Goethert, W.; Bailey, E.; Busby, M., *Software Effort and Schedule Measurement. A framework for Counting Staff-Hours and Reporting Schedule Information*. CMU/SEI-92-TR-20, Software Engineering Institute, Pittsburg, PA, 1992.

[Grah95] Graham, I., *Migrating to Object Technology*. Addison-Wesley, 1995, R U

[Grah96] Graham, I., 'Making Progress in Metrics'. *Object Magazine*, Vol. 6, N° 8, Octubre 1996, pg: 68-73.

[Gray97] Gray, A.R. & MacDonell, S.G., 'A Comparison of Techniques for Developing Predictive Models of Software Metrics'. *Information & Software Technology*, Vol. 39, 1997, pg: 425-437.

[Hals77] Halstead, M.H., *Elements of Software Science*. Elsevier North-Holland, Nueva York, 1977.

[Hans78] Hansen, W.J., 'Measurement of Program Complexity by the pair (ciclomatic number, operator count)'. *ACM SIGPLAN Notices*, 1978, pg: 29-33.

[Harr81] Harrison, W. & Magel, K., 'A Complexity Measure Based on Nesting Level'. *ACM SIGPLAN Notices*, Vol. 16, N° 3, 1981, pg: 63-74.

[Harr92] Harrison, W., 'An Entropy-Based Measure of Software Complexity'. *IEEE Transactions on Software Engineering*, Vol. 18, N° 11, Noviembre 1992, pg: 1025-1029.

[Harr98] Harrison, R. & Counsell, S.J. & Nithi, R.V., 'An Evaluation of the MOOD Set of Object Oriented Software Metrics'. *IEEE Transactions on Software Engineering*, Vol. 24, N° 6, Junio 1998, pg: 491-496.

[Hayn95] Haynes, P. & Menzies, T. & Phipps, G., 'Using the Size of Classes and Methods as the Basis for Early Effort Estimation: Empirical Observations, Initial Applications; A practitioner's Experience Report'. *OOPSLA'95 Workshop on Object Oriented Process & Metrics for Effort Estimation*, Austin, TX. Octubre 1995, Addendeum a las actas, pg: 138-142.

[Hayn96] Haynes, P. & Henderson-Sellers, B., 'Cost Estimation of OO Projects: Empirical Observations, Practical Applications'. *American Programmer*, Vol. 9, N° 4, pg: 35-41.

[Hayn97] Haynes, P. & Avotins, J. & Henderson-Sellers, B., 'Classes as a Size Unit for Cost Estimation and Productivity Measurement'. *Proc. of OOPSLA'97*, Atlanta, GE, Octubre 1997.

[Hech77] Hecht, M.S., *Flow Analysis of Computer Programs*. Elsevier, Nueva York, 1977.

[Heem92] Heemstra, F.J., 'Software Cost Estimation'. *Information & Software Technology*, Vol. 34, N° 10, pag: 627-639.

[Helm66] Helmer-Heidelberg, O., *Social Technology*. Basic Books, Nueva York, 1966.

[Hend90] Henderson-Sellers, B., 'The Object Oriented Systems Life Cicle'. *Communications of the ACM*, Vol. 33. N° 9 , Septiembre 1990, pg: 142-159.

[Hend95] Henderson-Sellers, B., 'OO Metrics Programme'. *Object Magazine*, Vol 5, N° 6, Octubre 1995, pg: 73-95.

[Hend96] Henderson-Sellers, B., *Object Oriented Metrics: Measures of Complexity*. Prentice Hall, Eaglewood Cliffs, NJ, 1996, ISBN: 0-13-239872-9.

[Hend96a] Henderson-Sellers, B., 'The Mathematical Validity of Software Metrics' *ACM SIGSOFT Software Engineering Notes*, Vol. 21, N° 5, Septiembre 1996, pg: 89-94.

[Hend97] Henderson-Sellers, B., 'Corrigenda: Software Size Estimation of Object Oriented Systems'. *IEEE Transactions on Software Engineering*, Vol. 23, N° 4, Abril 1997, pg: 260-261.

[Henr81] Henry, S. & Kafura, D., 'Software Metrics based on Information Flow'. *IEEE Transactions on Software Engineering*, Vol. 7, N° 5, Septiembre 1981, pg: 510-518.

[Henr88] Henry, S. & Wake, S., *Predicting Maintainability with Software Quality Metrics*. TR88-46, Department of Computer Science, Virginia Polytechnic, Blacksburg, VA, 1988.

[Hitz95] Hitz, M. & Montazeri, B., 'Measuring Coupling and Cohesion in Object Oriented Systems'. *Proc. of the International Symposium on Applied Corporate Computing*, Monterey Mexico, Octubre 1995

[Hitz95a] Hitz, M. & Montazeri, B., 'Measuring Product Attributes of Object Oriented Systems'. *Proc of the 5th European Software Engineering Conference*, Sitges, Barcelona, Septiembre 1995

[Hitz96] Hitz, M. & Montazeri, B., 'Chidamber & Kemerer's Metrics Suite: a Measurement Theory Perspective' *IEEE Transactions on Software Engineering*, Vol. 22, N° 4, Abril 1996, pg: 267-270

[Hitz96a] Hitz, M. & Montazeri, B., 'Measuring Coupling in Object Oriented Systems'. *Object Currents*, Vol. 1, N° 4, Abril 1996, Direccion web: http://www.sigs.com/objectcurrents/.

[Holt97] Holt, J., 'Software Metrics - Real World Experiences' *Software Process - Improvement & Practice*, Vol. 3, N° 3, Septiembre 1997, pg: 155-163

[Höst97] Höst, M. & Wohlin, C., 'A Subjetive Effort Estimation Experiment' *Information & Software Technology*, Vol. 39, N° 11, 1997, pg: 755-762

[Houl94] Hoult, D.P. & Meador, C.L. & Eisner, M., *Predicting Product Manufacturing Costs from Design Attributes: A Complexity Theory Approach*. Management Support Technology Corporation, Internal Working Paper, 1994.

[Hugh96] Hughes, R.T., 'Expert Judgment as an Estimating Method'. *Information & Software Technology*, Vol. 38, 1996, pg: 67-75.

[Hump95] Humphrey, W.S., *A Discipline for Software Engineering*. Addison Wesley, ISBN: 0-201-54610-8, 1995.

[Hump96] Humphrey, W.S., 'Estimating with Objects, Part: 1'. *Object Currents*, Vol. 1, N° 7, Julio, 1996.

[IEEE89] IEEE Computer Society: *IEEE Standard Dictionary of Measures to Produce Reliable Software*. IEEE Standard 982.1-1989.

[IEEE89a] IEEE Computer Society: *IEEE Guide for the Use of Standard Dictionary of Measures to Produce Reliable Software*. IEEE Standard 982.2-1989.

[IEEE90] IEEE Computer Society: *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Standard 610.12-1990.

[IEEE92] IEEE Computer Society: *IEEE Standard for Software Productivity Metrics*. IEEE Standard 1045-1992.

[IEEE92a] IEEE Computer Society: *IEEE Standard for a Software Quality Metrics Methodology*. IEEE Standard 1061-1992.

[Jaco92] Jacobson, I. & Christerson, M. & Jonsson, P. & Övergaard, G. *Object Oriented Software Engineering: a Use Case Driven Approach*. Addison-Wesley, 1992. ISBN: 0-201-54435-0.

[Jens83] Jensen, R.W. 'An Improved Macro-Level Software Development Resorce Estimation Model'. *Proc. of the 5th International Society of Parametric Analysis*, 1983.

[Jens84] Jensen, R.W. 'A comparison of the Jensen and COCOMO Estimation Models'. *Proc. of the 6th International Society of Parametric Analysis*, 1984, pg: 96-106.

[Jens91] Jenson, R.L. & Bartley, J.W., 'Parametric Estimation of Programming Effort: an Object-Oriented Model'. *The Journal of Systems and Software*, Vol. 15, 1991, pg: 107-114.

[Jone78] Jones, C., 'Measuring Programming Quality and Productivity' *IBM Systems Journal*, Vol. 17, N°1, 1978.

[Jone86] Jones, C., *The SPR Feature Points Method* Software Productivity Research, Inc., 1986

[Jone86a] Jones, C., *Programming Productivity*. McGraw-Hill, Nueva York, 1986

[Jone87] Jones, C., *A short history of Function Points and Feature Points* Software Productivity Research, Inc., 1987.

[Jone96] Jones, C., *Patterns of Software Systems Failures and Success* International Thomson Computer Press, ISBN: 1-850-32804-8, 1996

[Kama95] Kamath, S. & McGregor, J.D., *Psychological Complexity Measure at the Domain Analysis Phase of an Object Oriented System* Technical Report 95-110, Universidad de Clemson, 1995.

[Kan95] Kan, S.H., *Metrics & Models in software Quality Engineering* Addison Wesley, ISBN: 0-201-63339-6, 1995.

[Karu93] Karunanithi, S. & Bieman, J.M., 'Candidate Reuse Metrics for Object Oriented and ADA Software'. *Proc IEEE-CS 1st International Metrics Symposium*, 1993, pg: 120-128.

[Keme87] Kemerer, C.F., 'An Empirical Validation of Software Cost Estimation Models'. *Communications of the ACM*, Vol 30, N° 5, Mayo 1987, pg 416-429.

[Khos92] Khoshgoftaar, T.M. & Munson, J.C., 'An Aggregate Measure of Program Module Complexity'. *Annual Oregon Workshop on Software Metrics* Silver Falls, OR, Marzo 1992.

[Kitc92] Kitchenham, B., 'Empirical Studies of Assumptions that underlie Software Cost-Estimation Models'. *Information & Software Technology* Vol. 34, N° 4, Abril 1992, pg: 211-218.

[Kitc95] Kitchenham, B. & Pfleeger, S.L. & Fenton, N., 'Towards a Framework for Software Measurement Validation' *IEEE Transactions on Software Engineering*, Vol. 21, N° 12, Diciembre 1995, pg: 929-944

[Kitc98] Kitchenham, B., 'A Procedure for Analyzing Unbalanced Datasets'. *IEEE Transactions on Software Engineering*, Vol. 24, N° 4, Abril 1998, pg: 278-301.

[Kole75] Kolence, K.W., 'Software Physics'. *Datamation*, Junio 1975, pg: 48-51.

[Kole93] Kolewe, R., 'Metrics in Object-Oriented Design and Programming'. *Software Development*, Vol.1, N°4, Octubre 1993, pg: 53-62.

[Kran71] Krantz, D.H. & Luce, R.D. & Suppes, P. & Tversky, A., *Foundations of Measurement - Additive and Polynomial Representations*. Academic Press, Volumen 1, 1971.

[Kunt97] Kuntz, J.J., 'Is your organization one of Europe's Leaders or Laggers?'. *IBM Survey Update, Reunión SPIN-Spain*, 24 de Abril, 1997.

[Labo70] Labovitz, S., 'The Assignment of Numbers to Rank Order Categories'. *American Sociological Review*, Vol. 35, 1970, pg: 515-524.

[Laks91] Lakshmanan, K.B. & Jayaprakash, S. & Sinha, P.K., 'Properties of Control-Flow Complexity Measures'. *IEEE Transactions on Software Engineering*, Vol. 17, N° 12, Diciembre 1991, pg: 1289-1295.

[Lara90] Laranjeira, L. A., 'Software Size Estimation of Object Oriented Systems'. *IEEE Transactions on Software Engineering*, Vol. 16, N° 5, Mayo 1990, pg: 510-522.

[Lede93] Lederen, A.L. & Prasad, J., 'Information systems software cost estimating: a current assesment'. *Journal of Information Technology*, Vol. 8, pg: 22-23.

[Lede98] Lederen, A.L. & Prasad, J., 'A Causal Model for Software Cost Estimating Error'. *IEEE Transactions on Software Engineering*, Vol. 24, N° 2, Febrero 1998, pg: 137-148.

[Lee93] Lee, Y. & Liang, B. & Wang, F., 'Some Complexity Metrics for Object Oriented Programas Based on Information Flow'. *Proc. of the International Conference on Computer Systems & Software Engineering COMPEURO'93*, Paris, 1993.

[Lee94] Lee, Y. & Liang, B. & Wang, F., 'Some Complexity Metrics for Object Oriented Programs Based on Information Flow: A Study of C++ Programs'. *Journal of Information & Software Engineering*, N° 10, 1994, pg: 21-50.

[Lee95] Lee, Y. & Liang, B. & Wu, S. & Wang, F., 'Measuring the Coupling and Cohesion of an Object Oriented Program Based on Information Flow'. *Proceedings of the International Conference on Software Quality ISCO'95*, Maribor, Slovenia, 1995.

[Lee98] Lee, A. & Cheng, C. H. & Balakrishnan, J., 'Software Development Cost Estimation: Integrating Neural Network with Cluster Analysis' *Information & Management*, Vol. 34, N°1, Agosto 1998, pg 1-9.

[Lewi95] Lewis, J.A., 'Quatified Object-Oriented Development: Conflict and Resolution'. *Proceedings of the 8th Software Quality Conference*, Dundee 1995.

[Li93] Li, W. & Henry, S., 'Object Oriented Metrics which predict Maintainability'. *The Journal of Systems and Software*, Vol 23, N°2, 1993, pg: 117-122.

[Li95] Li, W. & Henry, S. & Kafura, D. & Schulman, R., 'Measuring Object Oriented Design'. *Journal of Object Oriented Programming*, Vol. 8, N° 4, Julio 1995, pg: 48-55.

[Lieb88] Lieberherr, K. & Holland, I. & Riel, A., 'Object Oriented Programming: an Objective Sense of Style'. *3^{a} Annual ACM Conference on Object Oriented Programming Systems, Languages and Applications, OOPSLA'88*, ACM Press, pg: 323-334.

[Lint96] Linthicum, D.S., 'In Search of the Elusive OO Metric'. *Object Magazine*, Vol. 6, N° 6, Agosto 1996, pg: 64-66.

[Lo95] Lo, R. & Webby, R. & Jeffery, R., 'Sizing and Estimating the Coding and Unit Testing Effort for GUI Systems'. *Proc. of the Australian Conference on Software Metrics*, Sidney, Australia, 1995.

[Loka96] Lokan, C.J., 'Early Size Prediction for C and Pascal Programs'. *The Journal of Systems and Software*, Vol. 32, N° 1, 1996, pg: 65-72.

[Long86] Longworth, H.D. & Ottstein, L.M. & Smith, M.R., 'The Relationship between Program Complexity and Slice Complexity During Debugging Tasks'. *IEEE COMPSAC*, Octubre 1986, pg:383-389.

[Lore94] Lorenz, M. & Kidd, J., *Object Oriented Software Metrics*. Prentice-Hall Object Oriented Series. Prentice-Hall, Eaglewood Cliffs, NJ, 1994, ISBN 0-13-179292-X.

[MacD91] MacDonell, S.G., 'Rigour in Software Complexity Measurement Experimentation'. *The Journal of Systems and Software*, Vol. 16, 1991, pg: 141-150.

[MacD93] MacDonell, S.G., 'Deriving relevant functional measures for automated development projects'. *Information & Software Technology*, Vol. 35, N° 9, 1993, pg: 499-512.

[MacD94] MacDonell, S.G., 'Statistical Analysis Procedures for Software Complexity Assesment Data'. *New Zealand Journal of Computing*, Vol. 5, N° 1, 1994, pg: 67-75.

[MacD94a] MacDonell, S.G., 'Comparative Review of Functional complexity Assesment Methods for Effort Estimation'. *Software Engineering Journal*, Vol. 9, N° 3, 1994, pg: 107-116.

[MacD96] MacDonell, S.G. & Gray, A. R., 'Alternatives to Regression Models for Estimating Software Projects'. *Proc. of the IFPUG Fall Conference*, Dallas, Texas, 1996, pg: 279.1-279.15.

[MacD96a] MacDonell, S.G., 'Effort Estimation for the Development of Spatial Information Systems'. *Proc. of the 8th Annual Colloquium of the Spatial Information Research Centre*, Dunedin, Nueva Zelanda, 1996, pg: 149-155.

[Mada97] Madachy, R.J., 'Heuristic Risk Assessment using Cost Factors'. *IEEE Software*, Vol. 14, N° 3, Mayo 1997, pg: 51-59.

[Mars97] Marshall, I.M. & Price, S. & Dugard, P.I. & Hobbs, P. & Samson, W.B., 'Code-based Analysis of the Development Effort of a Large Scale Courseware Project'. *Information & Software Technology*, Vol. 39, N° 8, 1997, pg: 541-549.

[Mart95] Martin, R., 'OO Design Quality Metrics: an Analysis of Dependecies'. *Report on Object Oriented Analysis & Design*, Vol. 2, N° 3, Septiembre 1995.

[McAn93] McAndrews, D., *Establishing a Software Measurement Process*. CMU/SEI-93-TR-016, Software Engineering Institute, Pittsburg, PA, 1993.

[Mcca76] McCabe, T., 'A Complexity Measure'. *IEEE Transactions on Software Engineering*, Vol. SE-2, N° 4, Diciembre 1976, pg: 308-320.

[McCl78] McClure, C.L., 'A Model of Program Complexity Analysis'. *Third International Conference on Software Engineering*, Mayo 1978, pg: 149-157.

[McGr95] McGregor, J.D., 'Managing Metrics in an Iterative Environment' *Object Magazine*, Vol. 5, N° 6, Octubre 1995, pg: 65-71.

[Melt96] Melton, A. *Software Measurement*. International Thomson Computer Press, ISBN: 1-85032-178-7, 1996.

[Mill74] Miller, R.G., 'The Jackknife - A Review'. *Biometrika*, Vol. 61, N° 1, 1974, pg: 1-15.

[Mill88] Mills, E.E., *Software Metrics*. SEI Curriculum Module SEI-CM-12-1 1, Software Engineering Institute, Pittsburg, PA, Diciembre 1988.

[Mišì98] Mišic, V.B. & Tešic, D.N., 'Estimation of Effort and Complexity: an Object Oriented Case Study'. *The Journal of Systems and Software*, Vol. 41, N° 2, 1998, pg: 133-143.

[Moha81] Mohanty, S.N., 'Entropy Metrics for Software Design Evaluation'. *The Journal of Systems and Software*, Vol. 2, 1981, pg: 39-46.

[Mora97] Morasca, S. & Briand, L. & Weyuker, E. J. & Zelkowitz, M.V., 'Comments on «Towards a Framework for Software Measurement Validation»'. *IEEE Transactions on Software Engineering*, Vol. 23, N° 3, Marzo 1997, pg: 187-188.

[Morr89] Morris, K.L., *Metrics for Object Oriented Software Development Environments*. M.S. Thesis, 1989. Massachusetts Institute of Technology, Sloan School of Management, Cambridge, MA.

[Mose96] Moser, S., 'Estimating the Modern Software Process'. *Proceedings of ECOOP'96*, Linz, Austria.

[Mose96a] Moser, S. & Nierstrasz, O., 'The Effects of Object Oriented Frameworks on the Developer Productivity'. *IEEE Computer*. Vol 29, N° 9, Septiembre 1996, pg:45-51.

[Mull97] Muller, P.A., *Instant UML*. Wrox Press, 1997, ISBN: 1-861000-87-1.

[Mukh92] Mukhopadhyay, T. & Vicinanza, S. S. & Prietula, M.J., 'Examining the Feasibility of a Case-Based Reasoning Model for Software Effort Estimation'. *MIS Quarterly*, Junio 1992, pg: 155-171.

[Mukh92a] Mukhopadhyay, T. & Kekre, S., 'Software Effort Models for Early Estimation of Process Control Applications'. *IEEE Transactions on Software Engineering*, Vol. 18, N° 10, Octubre 1992, pg: 915-923.

[NASA84] National Aeronautics and Space Administration, *Measures and Metrics for Software Development*. Software Engineering Laboratory Series, SEL-83-002, Marzo 1984.

[NASA90] National Aeronautics and Space Administration, *Proc. of the 15th Annual Software Engineering Workshop*. Software Engineering Laboratory Series, SEL-90-006, Noviembre 1990.

[Neis66] Nelson, E.A., *Management Handbook for the estimation of Computer Programming Costs*. AD-A648750, Systems Development Corp. 1966.

[Nesi98] Nesi, P., 'Managing OO Projects Better'. *IEEE Software*, Vol. 15, N° 4, Julio/Agosto 1998, pg: 50-60.

[Nesi98a] Nesi, P. & Quercy, T., 'Effort Estimation and Prediction of Object Oriented Systems'. *The Journal of Systems and Software*, Vol. 42, N° 1, Julio 1998, pg: 89-102.

[Nord58] Norden, P. V., 'Curve Fitting for a Model of Applied Research and Development Scheduling'. *IBM Journal of Research and Development*, Vol. 2, N° 3, Julio 1958.

[OMG92] OMG Object Analysis and Design SIG, *Object Analysis & Design, Reference Model, Draft 7.0*, Object Management Group, 1992.

[Ott93] Ott, L.M & Thuss, J.J., 'Slice Based Metrics for Estimating Cohesion'. *Proc. IEEE Computer Society International Software Metrics Symposium*, 1993, pg:71-81.

[Ott95] Ott, L.M. & Bieman, J.M. & Kang, B. & Mehra, B., 'Developing Measures of Class Cohesion for Object Oriented Software'. *Proc. of the Annual Oregon Workshop on Software Metrics (AOWSM'95)*, Junio 1995.

[Ovie80] Oviedo, E., 'Control Flow, Data Flow and Programmers Complexity'. *Proc. of COMPSAC'80*, Chicago, IL, 1980, pg: 146-152.

[Page88] Page-Jones, M., *The Practical Guide to Structured Systems Design*. Prentice-Hall, Eaglewood Cliffs, NJ, 1988, ISBN: 0-13-690777-6.

[Page95] Page-Jones, M., *What every programmer should know about object-oriented design*. Dorset House, New York, 1995, ISBN: 0-93-263331-5.

[Pant95] Pant, Y.R. & Verner, J.M. & Henderson-Sellers, B., 'S/C a Software Size/Complexity Measure', Capítulo 50 en *Software Quality and Productivity. Theory, Practice, Education and Training*. Ed. Lee & Barta & Juliff, Chapman & Hall, Londres, 1995, pg: 320-327.

[Pant96] Pant, Y.R. & Henderson-Sellers, B. & Verner, J.M., 'Generalization of Object Oriented Components for Reuse: Measurement of Effort and Size Change'. *Journal of Object Oriented Programming*, Vol. 9, N° 2, Mayo 1996, pg: 19-31.

[Park88] Park, Robert E., 'PRICE-S The Calculations Within and Why'. *Proc of International Society of Parametric Analysis, 10th Annual Conference*, Brighton, R.U. Julio 1988.

[Park92] Park, Robert E., *Software Size Measurement: A Framework for Counting Source Statements*. Software Engineering Institute. Pittburg. SEI-92-TR-020, Mayo 1992.

[Park94] Park, R. & Goethert, W. & Webb, J., *Software Cost and Schedule Estimating: A Process Improvement Initiative*. CMU/SEI-94-SR-003, Software Engineering Institute, Pittsburg, PA, Mayo 1994.

[Park95] Park, R., *A Manager's Checklist for Validating Software Cost and Schedule Estimates*. CMU/SEI-95-SR-004, Software Engineering Institute, Pittsburg, PA, 1995.

[Parn75] Parnas, D.L., 'The influence of Software Structure on Reliability'. *Proc of International Conference on Reliable Systems*, 21-23 Abril 1975, pg:358-362.

[Parn90] Parnas, D.L., 'Education for Computing Professionals'. *IEEE Computer*, Vol. 23, N° 1, Enero 1990, pg: 17-22.

[Parr80] Parr, F.N., 'An Alternative to the Rayleigh Curve Model for Software Development Effort'. *IEEE Transactions on Software Engineering* Vol 6, N° 3, 1980.

[Pete98] Peters, K., 'Software Project Estimation'. *Application Development Strategies (Newsletter)*, Vol. X, N° 7, Julio 1998, pg: 1-16.

[Pflee91] Pfleeger, S.L. & Fitzgerald, J.C., 'Software Metrics Tool Kit: Support for Selection, Collection and Analysis'. *Information & Software Technology*, Vol. 33, N° 7, Septiembre 1991, pg: 477-482.

[Pilla97] Pillai, K. & Sukumaran, N., 'A Model of Software Development Effort and Cost Estimation'. *IEEE Transactions on Software Engineering*, Vol 23, N° 8, Agosto 1997, pg: 485-497.

[Piwo82] Piwowarski, P., 'A Nesting Complexity Measure'. *ACM SIGPLAN Notices*, Vol. 17, N° 9, 1982, pg: 44-50.

[Poul97] Poulin, J. S., 'Fueling Software Reuse with Metrics'. *Object Magazine*, Vol. 7, N° 7, Septiembre 1997, pg: 68-73.

[Port90] Porter, A.A. & Selby, R.W., 'Empirically guided Software Development using Metric-Based Classification Trees'. *IEEE Software*, Vol. 7, N° 2, 1990, pg: 46-54.

[Pulf96] Pulford, K. & Kuntzmann-Combelles, A. & Shirlaw, S., *A Quantitative Approach to Software Management: The AMI Handbook*. AMI-ESPRIT, Addison-Wesley, 1996. ISBN: 0-201-87746-5.

[Putn78] Putnam, L.H., 'A General Empirical Solution to the Macro Software Sizing and Estimating Problem'. *IEEE Transactions on Software Engineering*, Vol. 4, N° 4, Julio 1978, pg: 345-361.

[Putn79] Putnam, L.H. & Fitzsimmons, A., 'Estimating Software Costs'. *Datamation*, Septiembre 1979, pg: 189-198.

[Putn91] Putnam, L.H., 'Trends in Measurement, Estimation and Control'. *IEEE Software*, Marzo 1991, pg: 105-107.

[RADC84] RADC (Rome Air Development Center), *Automated Software Design Metrics*, RADC-TR-84-27, 1984, Air Force System Command, Base de las Fuerzas Aereas de Griffies, NY.

[Reif94] Reifer, D.J., 'Cost Estimations'. *Encyclopedia of Software Engineering*. Ed. J.J. Marciniack, John Wiley & Sons, 1994, pg: 209-220.

[Rifk91] Rifkin, S. & Cox, C., *Measurement in Practice*. CMU/SEI-91-TR-016, Software Engineering Institute, Pittsburg, PA, 1991.

[Robe79] Roberts, F.S., 'Measurement Theory with Applications to Decision Making, Utility and the Social Sciences'. *Encyclopedia of Mathematics and its Applications*. Addison-Wesley, 1997.

[Robi89] Robillard, P.N. & Boloix, G., 'The Interconnectivity Metrics: a New Metric Showing How a Program is Organized'. *The Journal of Systems and Software*, Vol. 10, 1989, pg: 29-39.

[Rose97] Rosenberg, L.H. & Hyatt, L.E., 'Software Quality Metrics for Object Oriented Environments'. *CrossTalk: the Defence Software Engineering Report*, Vol. 10, N° 4, Abril 1997.

[Rozu94] Rozum, J.A., 'Defining and Understanding Software Measurement Data'. *Proc. of the 5th International Applications of Software Measurement Conference*, 1994.

[Rozu96] Rozum, J. A. & Iyer S., *A Data Definition Framework for Defining Software Measurements*. SEI Technical Report, Software Engineering Institute, Pittsburg, PA, 1996.

[Rube68] Rubey, R.J. & Hartwick, R.D., 'Quantitative Measurement Program Quality'. *ACM National Computer Conference*, 1968, pg: 671-677.

[Rubi83] Rubin, H.A., 'Macro-Estimation of Software Development Parameters: the ESTIMACS System'. *Proc. SOFTFAIR: A Conference on Software Development Tools, Techniques and Alternatives*. IEEE, Julio 1983, pg: 109-118.

[Rumb91] Rumbaugh, J. & Blaha, M. & Premerlani, W. & Eddy, F. & Lorensen, W., *Object Oriented Modeling and Design*. Prentice Hall, Eaglewood Cliffs, NJ, 1991.

[Rumb95] Rumbaugh, J., 'OMT: The Development Process'. *Journal of Object Oriented Programming*, Vol.8, N° 2, Mayo 1995, pg: 8-16.

[Rust81] Ruston, H., *Software Modelling Studies: The Polynomical Measure of Complexity*. RADC-TR-81-183, Rome Air Development Centre, Air Force Systems Command, Roma, NY, Julio 1981.

[Schl89] Schlender, B.R, 'How to break the software logjam'. *Fortune*, Septiembre 1989, pg: 108-112.

[Schn77] Schneiderwind, N.F., 'Modularity considerations in Real Time Operating Structures'. *COMPSAC'77*, pg: 397-403.

[Schn79] Schneiderwind, N.F. & Hoffman, H., 'An Experiment on Software Error Data Collection and Analysis'. *IEEE Transactions on Software Engineering*, Vol. 5, N° 3, 1979, pg: 105-117.

[Schn92] Schneiderwind, N.F., 'Methodology for Validating Software Metrics' *IEEE Transactions on Software Engineering*, Vol. 18, N° 5, Mayo 1992, pg: 410-421.

[Schn94] Schneiderwind, N.F. 'Methodology for Validating Software Metrics' *Encyclopedia of Software Engineering*. Ed. J.J. Marciniack, John Wiley & Sons, 1994, pg: 666-676.

[Scho95] Schofield, C. & Shepperd, M., 'Software Support for Cost Estimation by Analogy'. *Proceedings of ESCOM 6*, Rolduc, 1995.

[SEI95] Software Engineering Institute, *Checklists and Criteria for Evaluating the Cost and Schedule Estimating Capabilities of Software Organizations*, CMU/SEI-95-SR-005. Software Engineering Institute, Pittsburg, PA.

[Selb88] Selby, R.W. & Porter, A.A., 'Learning from Examples: Generation and Evaluación of Decision Trees for Software Resource Analysis'. *IEEE Transactions on Software Engineering*, Vol. 14, N° 12, Diciembre 1988, pg: 1743-1756.

[Shar93] Sharble, R.C. & Cohen, S.S., 'The Object Oriented Brewery: A Comparison of Two Object Oriented Development Methods'. *ACM SIGSOFT Software Engineering Notes*, Vol. 18, N° 2, Abril 1993, pg: 60-73.

[Shep93] Shepperd, M. *Software Engineering Metrics - Vol 1: Measures and Validations*. McGraw-Hill, International Series on Software Engineering, 1993.

[Shep93a] Shepperd, M. & Ince, D., *Derivation and Validation of Software Metrics*. Claredon Press, 1993, Oxford.

[Shep95] Shepperd, M. *Foundations of Software Measurement*. Prentice Hall, ISBN: 0-13-336199-3, 1995.

[Shep96] Shepperd, M. & Schofield, C., 'Effort Estimation by Analogy: a Case Study'. *Proceedings of the 7th European Control and Metrics Conference*, Wilmslow, Reino Unido, Mayo 1996.

[Shep96a] Shepperd, M. & Schofield, C. & Kitchenham, B., 'Effort Estimation using Analogy'. *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Alemania, 1996, pg: 170-178.

[Shep97] Shepperd, M. & Schofield, C., 'Estimating Software Project Effort Using Analogies'. *IEEE Transactions on Software Engineering*, Vol. 23, N° 12, Noviembre 1997, pg: 736-743.

[Shih97] Shih, T.K. & Wang, C. & Chung, C., 'Using Z to specify Object Oriented Software Complexity Measures'. *Information & Software Technology*, Vol. 39, N° 8, 1997, pg: 515-529.

[Snee94] Sneed, H. 'Calculating Software Costs using Data Points'. *SES*. Ottobrunn/Munich, Alemania.

[SPC94] Software Productivity Consortium. *Software Measurement Guidebook, Versión 2.01*. SPC-91060-CMC. Software Productivity Consortium Services Corporation, Herndon, Virginia, 1994.

[Srin95] Srinivasan, K. & Fisher, D., 'Machine Learning Approaches to Estimating Software Development Effort'. *IEEE Transactions on Software Engineering*, Vol. 21, N° 2, Febrero 1995, pg: 126-136.

[Stev74] Stevens, W.P. & Myers, G.J. & Constantine, L.L., 'Structured Design' *IBM Systems Journal*, N°2, 1974, pg: 115-139.

[Stut96] Stutzke, R.D., 'Software Estimating Technology: a Survey'. *CrossTalk The Defense Software Engineering Report*, Vol. 9, N° 5, Mayo 1996 .

[Subr93] Subramanian, G.H. & Brelawski, S., 'Dimensionality Reduction in Software Development Effort Estimation'. *The Journal of Systems and Software*, Vol. 21, 1993, pg: 187-196.

[Symo88] Symons, C.R., 'Function Point Analysis: Difficulties and Improvements' *IEEE Transactions on Software Engineering*, Vol. 14, N° 1, Enero 1988 pg: 2-11.

[Symo91] Symons, C.R., *Software Sizing and Estimating MARKII FPA* John Wiley & Sons, 1991.

[Tate91] Tate, G. & Verner, J., 'Approaches to measuring Size of Application Products with CASE Tools'. *Information & Software Technology*, Vol 33, N° 9, Noviembre 1991, pg: 622-628.

[Taus81] Tausworthe, R.C., *Deep Space Network Software Cost Estimation Model* Technical Report N° 81-7, Jet Propulsion Laboratory, 1981

[Tega92] Tegarden, D. & Sheetz, S. & Monarchi, D., Effectiveness of Traditional Software Metrics for Object Oriented Systems *Proc of the 25th Hawaii International Conference on System Sciences*, Vol 4, IEEE Computer Society Press, 1992, pg: 359-368.

[Theb84] Thebaut, S.M. & Shen, V.Y., 'An Analytic Resource Model for Large-Scale Software Development'. *Information Processing and Management*, Vol. 20, N° 1-2, 1984, pg: 293-315.

[Thom94] Thomson, N. & Johnson, R. & McLeod, R. & Miller, G. & Hanse, T., 'Project Estimation Using an Adaption of Function Points and Use Cases for OO Projects'. *Workshop on Pragmatic and Theoretical Directions in Object Oriented Software Metrics, OOPSLA'94*, Portland, OR, Octubre 1994.

[Tian95] Tian, J. & Zelkowitz, M.V., 'Complexity Measure Evaluation and Selection'. *IEEE Transactions on Software Engineering*, Vol 21, N° 8 Agosto 1995, pag: 641-650.

[Troy81] Troy, D. & Zweben, S., 'Measuring the Quality of Structured Design'. *The Journal of Systems and Software*, Vol. 2, 1981, pg: 113-120.

[Tsau98] Tsaur, W. & Horng, S., 'A New Generalized Software Complexity Metric for Distributed Programs'. *Information & Software Technology*, Vol. 40, N° 5-6, Julio 1998, pg: 259-269.

[USC97] COCOMO II.1997 *Model Definition Manual. Version 1.4.* University of Southern California, 1997.

[USC97a] COCOMO II *Cost Estimation Questionnaire. Version 1.6.* University of Southern California, 1997.

[Vern89] Verner, J.M. & Tate, G. & Jackson, B. & Haywood, R.G., 'Technology dependence in Function Point Analysis: a Case Study and Critical Review'. *Proc. of the International Conference on Software Engineering*, 1989, pg: 375-382.

[Vern92] Verner, J.M. & Tate, G., 'A Software Size Model'. *IEEE Transactions on Software Engineering*, Vol. 18, N° 4, Abril 1992, pag: 265-278.

[Vici90] Vicinanza, S. & Prietula, M.J. & Mukhopadhyay, T., 'Case-Based- Reasoning in Software Effort Estimation'. *Proc. of the 11th International Conference on Information Systems*, 1990, pg: 149-158.

[Vici91] Vicinanza, S. & Mukhopadhyay, T. & Prietula, M.J., 'Software Effort Estimation: an Exploratory Study of Expert Performance'. *Information Systems Research*, Vol. 2, N° 4, Diciembre 1991, pg: 243-262.

[Walk96] Walkerden, F. & Jeffery, R., *Software Cost Estimation: a Review of Models, Process and Practice.* CAESAR Technical Report #96/1; publicado en *Advances in Computers*, editor M.V. Zelkowitz, Academic Press, 1997.

[Wals77] Walston, C.E. & Felix, C.P., 'A Method of Programming Measurement and Estimation'. *IBM Systems Journal*, Vol. 16, N° 1, 1977, pg: 54-73.

[Wals77a] Walston, C.E. & Felix, C.P., 'On Lines of Code and Programming Productivity'. *IBM Systems Journal*, Vol. 16, N° 4, 1977, pg: 422-423.

[Wand90] Wand, Y. & Weber, R., 'An Ontological Model of an Information System'. *IEEE Transactions on Software Engineering*, Vol. 16, N° 11, Noviembre 1990, pg: 1282-1292.

[Weis82] Weiser, M.D., 'Programmers Use Slices When Debugging'. *Communications of the ACM*, Vol. 25, N° 7, Julio 1982, pg: 446-452.

[West96] West, M., 'Taking the Measure of Metrics'. *Object Expert*, Vol. 1, N° 2, Enero 1996, pg: 43-45.

[Weyu88] Weyuker, E.J., 'Evaluating Software Complexity Measures'. *IEEE Transactions on Software Engineering*, Vol. 14, N° 9, Septiembre 1988, pg: 1357-1365.

[Wilk98] Wilkie, F.G. & Hylands, B., 'Measuring Complexity in C++ Application Software'. *Software - Practice & Experience*, Vol. 28, N° 5, 25 de Abril 1998, pg: 513-546.

[Wirf90] Wirfs-Brock, R.J. & Wilkerson, B. & Wiener, L., *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[Whit94] Whitmire, S.A., 'Object Oriented Measurement of Software'. *Encyclopedia of Software Engineering*. Ed. J.J. Marciniack, John Wiley & Sons, 1994, pg: 737-739.

[Whit96] Whitmire, S.A., 'A Formal Object Model for Measurement'. *Proc of the OOPSLA'96 Conference*, Special Issue of *SIGPLAN Notices*, Septiembre 1996.

[Whit97] Whitmire, S. A., *Object Oriented Design Measurement*. John Wiley & Sons, 1997, ISBN: 0-471-13417-1.

[Whit96] Whitty, R., 'Object Oriented Metrics: a Status Report'. *Object Expert*, Vol. 1, N° 2, Enero 1996, pg: 35-40.

[Wolv74] Wolverton, R.W., 'The Cost of Developing Large-Scale Software'. *IEEE Transactions on Computers*, Vol. C-23, N° 6, Junio 1974, pg: 615-636.

[Wrig87] Wrigley, C.D. & Dexter, A.S., 'Software Development Estimation Models: a Review and Critique'. *Proc of the ASAC Conference*, Toronto, Junio 1987, pg: 125-138.

[Wrig91] Wrigley, C.D. & Dexter, A.S., 'A Model for Measuring Information System Size'. *MIS Quarterly*, Vol. 15, N° 2, 1991.

[Your75] Yourdon, E., *Techniques of Program Structure and Design*. Prentice-Hall, Eaglewood Cliffs, NJ, 1975, ISBN: 0-13-901702-X.

[Zelk98] Zelkowitz, M.V. & Wallace, D.R., 'Experimental Models for Validating Technology'. *IEEE Computer*, Vol. 31, N° 5, Mayo 1998, pg: 23-31.

[Zuse94] Zuse, H., 'Analysis of Complexity Metrics'. *Encyclopedia of Software Engineering*. Ed. J.J. Marciniack, John Wiley & Sons, 1994, pg: 131-164.

[Zuse97] Zuse, H., *History of Software Measurement*. Documento Personal, dirección web http://irb.cs.tu-berlin.de/~zuse.

[Zuse98] Zuse, H., *A Framework of Software Measurement*. De Gruyter, 1998, ISBN: 3-11-015587-7.