UNIVERSIDAD DE OVIEDO

DEPARTAMENTO DE INFORMÁTICA

TESIS DOCTORAL

DMT – METODOLOGÍA PARA EL DISEÑO DE MÉTRICAS EN TIEMPO DE EJECUCIÓN

Presentada por:

Aquilino Adolfo Juan Fuente

para la obtención del título de Doctor en Informática

Dirigida por

Dr. Juan Manuel Cueva Lovelle

Dr. Luis Joyanes Aguilar

DMT – METODOLOGÍA PARA EL DISEÑO DE MÉTRICAS EN TIEMPO DE EJECUCIÓN

UNIVERSIDAD DE OVIEDO

DEPARTAMENTO DE INFORMÁTICA

Autor:

Aquilino Adolfo Juan Fuente

Directores de Tesis:

Dr. Juan Manuel Cueva Lovelle

Dr. Luis Joyanes Aguilar

Esta tesis está dedicada...

A mi esposa Mamen y a mi hijo Héctor
por todo lo que me han apoyado durante
todo este tiempo.

A todos mis amigos y compañeros.

RESUMEN

Los sistemas de métricas de software tradicionales se han centrado fundamentalmente en las métricas de procesos, de productos y de recursos [FEN91]. Los principales objetivos de todos los sistemas de métricas desarrollados hasta el momento son el control y la estimación de los proyectos, la calidad del desarrollo o los costes y esfuerzos estimados para la fase de mantenimiento de la aplicación.

En algunas ocasiones se han utilizado las métricas en tiempo de ejecución, normalmente como estimadores de velocidad, testing de ejecución, profiling o para estimación del coste de servicios, pero en ningún caso con intención de control y monitorización de la aplicación. Con estos cometidos se han utilizado desleídas en el código y despojadas de intencionalidad.

En esta tesis se trata de dar un nuevo enfoque a las métricas. Lo primero es definir qué se entiende por métrica en tiempo de ejecución, después se separa lo que, en un diseño, atañe a proceso algorítmico puro y a las medidas de atributos insertos en la aplicación durante una ejecución de ésta.

Seguidamente, se define un framework básico para el soporte de Métricas en Tiempo de Ejecución (MTE ó RTM) en programas realizados en lenguaje Java.

También se define una ampliación a UML para soportar el diseño de sistemas de métricas y control de acuerdo a la filosofía inherente en DMT.

Por último se define un lenguaje de programación con soporte para métricas como una extensión de Java (JavaRTM) y se muestran unos casos de estudio utilizando el framework y el lenguaje JavaRTM.

PALABRAS CLAVE

Actuador, Análisis, Ciclo de vida, Clase, Clase abstracta, Codificación, Control, Controlador, Derivación, Diagrama de clases, Diagrama de estados, Diseño, Diseño de software, Escala, Esfuerzo, Formal, Framework, Ingeniería del software, Interfaz, Lenguaje de programación, Medición, Medida, Medida de software, Métrica, Métrica de software, Metrología, Modelo, Monitor, Monitorización, Objeto, Programación, Proyecto, Pruebas, Threads.

ABSTRACT

Traditional software metrics systems has been pointed mainly on process, products and resources [FEN91]. Main objectives in all the metrics systems developed up to date were control and projects estimation, development quality and costs and estimated efforts in the application maintenance phase.

Occasionally some kind of run-time metrics were used for speed measurement, execution testing, profiling and services costs. But they were never used for monitoring and control applications. For this purposes, they were used in an unclear way inside code.

This thesis tries to give a new view point to metrics. First I define what run-time metrics are, then I divide what is algorithmical and what is metrical code.

Next, I define a basic framework for supporting run-time metrics (RTM or MTE) in programs in Java language.

I also define an UML extension for supporting run-time metrics and control design, according to the philosophy of DMT.

Lastly a language is defined with support for metrics as an extension of Java –JavaRTM- and a few study cases are shown.

KEYWORDS

Abstract class, Actuator, Analysis, Class, Class diagram, Codification, Control, Controller, Derivation, Design, Effort, Formal, Framework, Interface, Lifetime, Measure, Measurement, Metric, Metrology, Model, Monitor, Monitoring, Object, Programming, Programming language, Project, Scale, Software design, Software engineering, Software measure, Software metric, States diagram, Test, Threads.

INDICE DE MATERIAS

ILUSTRACIONES

TABLAS

1. CONTENIDO

1.1. INTRODUCCIÓN

La definición del software dentro del rango de las ingenierías y de las ciencias conlleva asociada la necesidad de medir y de demostrar, mediante métodos matemáticos las hipótesis y los planteamientos que del software se hagan.

Pero la ingeniería del software se encuentra en fase de unificación entre los conceptos teóricos y los resultados empíricos que, en muchas ocasiones, no son todo lo alentadores que la disciplina necesita.

Si a todo ello sumamos que el software no es ni más ni menos que la representación de conceptos reales y la automatización de procesos (normalmente en donde el conocimiento humano tiene un gran protagonismo), nos encontramos con conceptos de difícil medida por lo complejo de su estudio y lo abstracto de sus planteamientos.

En estas condiciones, los primeros pasos dados en el sentido de medir son muchas veces frustrantes, ya que no se logran con claridad los objetivos que se persiguen.

La mayor parte de las métricas existentes están encaminadas a conseguir reducir costes, a predecir comportamientos o a controlar procesos de desarrollo de software. Al final del camino los objetivos son por tanto económicos, incluso cuando se pretende medir la dificultad del mantenimiento del producto lo que se está midiendo indirectamente es el coste de dicho mantenimiento.

En la siguiente tabla [DOL00] se muestran algunos de los objetivos habituales de la medición:

PROBLEMA	MEDIR AYUDA A
Incorrecciones	Proporcionar requerimientos verificables, expresados en términos medibles.
Toma de decisiones	Proporcionar evidencia cuantificable para apoyar las decisiones.
Falta de control	Hacer más visible el desarrollo e identificar problemas anticipadamente
Exceso de gasto	Producir predicciones de coste y plazo justificables
Costes de mantenimiento	Recomendar determinadas estrategias de prueba e identificar los módulos problemáticos
Evaluación de nuevos métodos	Valorar los efectos en la productividad y calidad

Tabla 1-1. Razones para medir

De la propia tabla se desprende que un objetivo importante de la medición es la reducción de costes.

Casi ningún trabajo abarca otro problema, que es el de medir el proceso en ejecución. Esto es, las métricas en tiempo real o métricas basadas en ejecución, que sólo han sido tocadas de forma muy superficial en el trabajo de Barnes & Swim [B&S93].

Las métricas en tiempo de ejecución abren la puerta a una nueva manera de hacer software que consiste en hacer control sobre el software en ejecución pudiendo incluso intervenir en el proceso para llevarlo a estados estables y controlados.

Es, por tanto, un enfoque absolutamente diferente del actual donde los objetivos de las métricas eran el control de costes, la calidad en diseño (de forma estática) o las previsiones de comportamiento, pero en ningún caso el control del proceso.

Haciendo un símil con los procesos industriales clásicos, esta manera de hacer software equivale al concepto industrial de control, ya que se basa en bucles de control (abiertos o cerrados) que toman medidas en el proceso y, en función de esas medidas, deciden acciones sobre el propio proceso para mantenerlo dentro de unos rangos de valores definidos en la fase de diseño.

Al desarrollar el sistema de métricas en tiempo de ejecución se han planteado varios problemas, el primero ha sido separar el control y el algoritmo de ejecución del proceso, de manera que la intervención en el proceso sea mínima y no se desvirtúen los valores medidos.

Otro problema surge a la hora de diseñar, ya que utilizar UML para el diseño haría que las clases que componen el sistema de medida y control se integrasen en los diagramas de clases quedando oscuro el concepto de la medición. Para que esto no ocurra hay que definir una simbología gráfica diferenciada que deje claramente establecida, ya en la fase de diseño, la diferencia entre ambos subsistemas.

El siguiente problema consistía en dar un soporte estándar al sistema, de manera que no hubiera que definir soluciones ad-hoc en cada proceso.

Por último esa independencia entre el sistema a controlar y el sistema de control lleva a independizar en el mayor grado posible el lenguaje con que se escribe el programa y el lenguaje con que se define el sistema de control.

El objetivo de esta tesis ha sido, por tanto diseñar todos los elementos que dieran solución a los problemas planteados y demostrar en algunos ejemplos prácticos la utilidad de lo diseñado.

1.2. CONTENIDOS DE LA TESIS

La estructura de esta tesis se muestra en los próximos párrafos, a modo de resumen se puede decir que está compuesta por cuatro partes bien diferenciadas: Una introducción teórica y clásica a los sistemas de medida, una visión del estado del arte en lo referente a los sistemas de métricas actuales, una sección crítica de dichos sistemas y una presentación de DMT.

Por último unas conclusiones sobre el trabajo realizado.

En el capítulo 2 se muestra una introducción teórica a los sistemas de métricas y mediciones, siguiendo los conceptos clásicos sobre métricas de software establecidos por Fenton.

El capítulo 3 muestra el estado actual del arte en lo que respecta a métricas, qué es lo que existe y por donde se enfocan las principales líneas de investigación actuales. En este capítulo se muestran las principales métricas clásicas y orientadas a objeto.

El capítulo 4 realiza una crítica de algunos aspectos de las métricas existentes, tanto clásicas como orientadas a objeto desde el punto de vista del trabajo desarrollado, en concreto se critica la falta de sistemas de métricas en tiempo real.

El capítulo 5 establece una serie de requerimientos que deberían tener las métricas orientadas a objeto para tiempo de ejecución (RTM), se muestra también una metodología de diseño (DMT) para métricas en tiempo de ejecución, un framework de métricas diseñado para Java que permite implementar los diseños de la metodología DMT, una extensión a Java (JavaRTM) para soportar métricas en tiempo real y también se muestran algunos resultados teóricos de la utilización de esta metodología.

El capítulo 6 muestra algunos casos prácticos en los que se utilizan las herramientas diseñadas.

Por último, el capítulo 7 establece unas conclusiones relativas a la investigación realizada y se marcan unas posibles líneas de investigación futuras para ampliación de estos resultados.

2. INTRODUCCIÓN TEÓRICA A LA MEDICIÓN

Cuando podemos medir aquello sobre lo que hablamos y expresarlo en números, entonces conocemos algo sobre ello. Pero cuando no podemos medirlo ni expresarlo en números, nuestro conocimiento es de un tipo pobre e insatisfactorio.

(Lord Kelvin)

2.1. TEORÍA DE LA MEDICIÓN

2.1.1. INTRODUCCIÓN

La informática ha evolucionado desde un enfoque puramente artesanal hasta llegar a uno más científico, donde cada vez es mayor la necesidad de justificar de una manera razonada tanto los diseños como la propia programación y la ejecución de las aplicaciones. Es, por tanto, cada vez más importante disponer de metodologías y de sistemas de métricas adecuados a nuestras necesidades.

Las ciencias y las ingenierías utilizan teorías formales generadas a partir de las observaciones empíricas y las expresan mediante notación matemática. La informática, por tanto, como ingeniería, debe utilizar también esas teorías formales.

Por todo ello y por la importancia que concedemos al hecho de medir, no podemos utilizar cualquier sistema de medida, sino que éste debe estar sujeto a una teoría: LA TEORÍA DE LA MEDICIÓN.

La ingeniería informática está aún en un estado de unificación de las observaciones empíricas y las teorías formales. Este estado, que es debido principalmente a la enorme complejidad del software, impide la penetración de los avances teóricos en la industria, reduce la calidad de los desarrollos de software y de los sistemas relacionados con el software, y crea desunión dentro de la profesión.

El objetivo de una teoría de la medida, es el de lograr que la descripción que las métricas nos aportan del sistema, sea objetiva, exacta, reproducible, segura y significativa.

En palabras de Bunge (1967) «*La historia de las ciencias ha sido, en gran parte, la historia de la cuantificación de conceptos inicialmente cualitativos*».

Por último surge la necesidad de definir de algún modo la medición, cosa que no es fácil si se desea ser suficientemente estricto para no definir con vaguedades y al mismo tiempo, suficientemente amplio para abarcar todo el campo del sujeto. La definición adoptada está extraída del *Software Engineer's Reference Book* [JON91] y se debe a Agnes A. Kaposi: «*la medición es el proceso de codificación objetiva y empí-*

rica de alguna propiedad de una clase seleccionada de entidades en un sistema formal de símbolos, de manera que la describan (a la clase)».

2.1.2. TEORÍA DE LA MEDICIÓN

La necesidad de la medición surge porque una o más propiedades son identificadas y necesitamos de alguna manera cuantificarlas para que nos aporten información. Gracias a ella, un conjunto importante de objetos (cosas, eventos, entidades,...) pueden ser descritos, identificados, catalogados y ordenados.

La medición ha permitido los avances científicos, pues permite establecer leyes generales de conocimiento a partir de las cuales podemos inferir nuevo conocimiento (nuevas teorías, hipótesis,...).

Los estándares de medición garantizan la compatibilidad, el intercambio y posibilidad de elección entre diferentes alternativas. La medición es por tanto fundamental en las actividades de la vida cotidiana y del comercio y facilita el intercambio de bienes e información.

La medición es un pilar fundamental de las ingenierías. Las medidas son necesarias en todas las fases del ciclo de vida de los productos y procesos.

El estándar BS5750 (1987) [BS5750] define los principios de aseguramiento de calidad de los productos y servicios, en él, las mediciones son necesarias como:

- Descriptores, que caracterizan una entidad ya existente. Sirven para evaluar una entidad frente a sus especificaciones y para distinguir unas entidades de otras.

- Prescriptores, que sirven para establecer características que una entidad debe tener una vez implementada.

- Predictores, que son usados durante el diseño para establecer o calcular futuras propiedades que una entidad debe tener.

2.1.2.1. METROLOGÍA, TIPOS

La metrología es la ciencia que estudia la medición, de acuerdo a la definición del estándar BS5233 [BS5233], «*La metrología es el campo del conocimiento que concierne a la medición*».

A partir de las ideas de Fiok *et al.* (1988) [FIOK88] distinguiremos dos tipos de divisiones consideradas en el original como *aspectos* y *esferas de interés* (o aspectos importantes desde los que ver la medición).

Los aspectos son una división en función de las necesidades para poder realizar una medición, implican por tanto los condicionantes teóricos, las necesidades prácticas y también las legales.

Estos aspectos son:

- La *metrología teórica*, es la teoría de base de la medición, recoge todos los conceptos filosóficos y metodológicos, fundamentos lógicos, formación de hipótesis y modelos, testabilidad, teoría general de experimentación, formación de escalas, interpretación de los datos, y teoría de errores.

- La *metodología de la metrología*, es el método de realización de la medición, comprende las estrategias de medición y la interpretación de los resultados.

- La *tecnología de la metrología*, engloba todo lo concerniente a la elección de instrumentos, procesos, almacenamiento de resultados y presentación de los mismos.

- La *metrología legal*, comprende todo aquello que concierne a la normativa, leyes, códigos deontológicos y cualquier otro tipo de restricción ajena a lo que es la teoría pura.

Las *esferas de interés* son una división jerárquica en función de la amplitud de la medida. Estas esferas son:

- La *metrología cuantitativa* o *metrología orientada a la propiedad*, que consiste en la determinación del valor de un parámetro particular de alguna entidad.

- La *metrología orientada a la caracterización del objeto*, que consiste en la caracterización de una entidad en términos de los parámetros seleccionados a partir de su modelo. (A pesar de su parecido en el nombre y de su clara utilidad en los sistemas orientados a objetos, esta esfera no es exclusiva de dichos sistemas)

- La *metrología orientada al dominio del sujeto*, la caracterización parametrizada de la clase de todas las entidades dentro de un dominio específico de ciencia, ingeniería, etc.

- La *metrología general*, que es el estudio, sin referencia a sujetos particulares, del modo en que se forman los conceptos medibles, en que se caracterizan paramétricamente las entidades y en que se desarrollan, operan y se mantienen los sistemas de medición.

2.1.2.2. EL PROCESO DE LA MEDICIÓN Y SUS CARACTERÍSTICAS

El proceso de la medición consta de tres partes claramente interrelacionadas:

- *Planificación de la medición*.- Consiste en la realización de planes para realizar la medición, la validación y la calidad de los resultados.

- *Organización de la ejecución de la medición*.- Consiste en la provisión de personal, instrumentos de medida, diseño de los procesos, entornos, soportes legales y en definitiva cualquier otra característica administrativa.

- *Control y revisión de la ejecución de la medición*.- Consiste en todos los procesos de supervisión, gestión y cualesquiera otros relacionados con el proceso durante la ejecución.

Desde el punto de vista de la metrología, está claro que el punto que más interesante resulta es el primero, quedando los otros dos fuera del ámbito de este documento.

Pero para que una **medición** sea válida se debe asegurar su calidad, lo que obliga a ésta a tener una serie de características importantes ([FEN91] y [FEN96]):

- Debe ser válida para el propósito que se pretende y debe obtener resultados significativos con respecto a ese propósito.

- El código usado en la medición no debe ser arbitrario, debe estar definido de manera que represente adecuadamente las propiedades medidas.

- No debe interferir en el proceso observado, de lo contrario los resultados obtenidos estarían falseados.

- Debe ser exacta, el error y la incertidumbre de la medición deben estar acotados.

- Debe ser reproducible, esto es, debe ser estable y objetiva, permitiendo que en las mismas condiciones, diferentes personas y en diferentes momentos obtengan el mismo resultado.

- Debe ser factible de realizar y no debe derrochar recursos ni malgastarlos en cosas irrelevantes.

- Los resultados de las medidas deben tener integridad, debe ser posible, por tanto, demostrar la imparcialidad de la medición.

2.1.3. MEDICIÓN CUANTITATIVA

Como ya se había comentado anteriormente este tipo de medición abarca la que se realiza sobre un parámetro particular de un atributo de una clase. Esta observación se realiza de modo directo, aunque es la base para la determinación del valor de una propiedad indirectamente, formando, por tanto, parte de los programas de medición orientada a objeto.

Hay que comprender que, en cualquier caso, en la práctica pocas propiedades admiten una medición directa, muchas mediciones se hacen de una forma indirecta, a través de la medición de otras propiedades que resultan más fáciles o convenientes de medir.

Para realizar esta medición cuantitativa se necesita construir dos sistemas: empírico y formal [KITC95].

2.1.3.1. SISTEMA RELACIONAL EMPÍRICO

Sea *A={a, b, ... , z}* el conjunto objeto y *K* la propiedad elegida para realizar la medición. El sistema relacional empírico, comprende el conjunto modelo elegido y todas las relaciones y operaciones que sobre dicho modelo se desee realizar, esto es, además de *A* y *K* comprenderá:

R: conjunto de relaciones n-arias en *A*.

O: conjunto de operaciones binarias en *A*.

Entonces, definiremos el sistema relacional empírico (*E*) como una 3-tupla formada por los elementos anteriormente descritos:

$$E = (A, R, O) = (\{a, b, \ldots, z\}, \{r1, r2, \ldots, rn\}, \{o1, o2, \ldots, on\})$$

que describe el sistema relacional para una propiedad concreta (*K*).

A modo de ejemplo considérese un conjunto formado por tramos de tubería en una instalación. El conjunto *A* estaría formado por todos estos tramos de tubería, *K* sería la *longitud*, el conjunto *R* podría estar formado por las relaciones de tamaño (*más largo que*, *igual a*, etc. ...) y el conjunto *O* podría estar formado por un único elemento que sería la *concatenación*.

Las relaciones *R* establecen ordenes en el conjunto *A* de acuerdo al valor del atributo *K* medido.

2.1.3.2. SISTEMA RELACIONAL FORMAL

Una vez elegido el sistema relacional empírico, definiremos el sistema relacional formal que debe ser homomorfo del anterior.

Sea *F = (A', R', O')* dicho sistema. Algunos autores llaman a éste el sistema relacional numérico, estableciendo de esta manera que las mediciones son realizadas siempre con una representación numérica. En este caso y de acuerdo a la definición que se ha hecho de medición, se ha preferido el término formal para indicar que cualquier sistema formal es válido. El sistema relacional formal *F* debe cumplir las siguientes condiciones:

- El sistema formal debe ser capaz de expresar todas las relaciones del sistema empírico *E* y debe también representar cualquier conclusión significativa prevista originalmente y obtenida a partir de los datos.
- El paso de *E* a *F* debe ser de tal manera que represente todas las observaciones, conservando todas las relaciones y operaciones del sistema empírico relacional.

Para satisfacer el primero de los requerimientos, el sistema relacional formal debe ser un objeto matemático que contenga las correspondientes relaciones y operaciones. Un ejemplo para el caso anterior de las tuberías podría ser un modelo numérico formado por:

$$F = (\{Reales\}, \{ \geq, = \}, \{ + \})$$

Por lo que respecta al segundo requerimiento, si buscamos una transformación adecuada de *E* en *F*, que mantenga todas las relaciones y operaciones definidas sobre *E*, entonces la transformación $T: E \rightarrow F$ es un homomorfismo y *E* y *F* son homomorfos.

En general una transformación es un homomorfismo si se cumple:

$$\forall a, b, ..., z \in A, rh \in R, rh' \in R' \text{ se cumple } rh(a, b, ..., z) \Leftrightarrow rh'(T(a), T(b), ...T(z))$$

$$\forall i,j \in A, og \in O, og' \in O' \text{ se cumple } T(i\ og\ j) \Leftrightarrow T(i)\ og'\ T(j)$$

Volviendo al ejemplo anterior, las dos relaciones vistas podrían ser:

- Para dos tramos de tubería *t1* y *t2*, y para la transformación *longitud del tramo*, que asocia a cada tramo de tubería con un valor real *l* equivalente a su longitud en metros *t1 es más largo que t2 si y sólo si l1 > l2*.

- En las mismas condiciones anteriores, t1 concatenado con t2 es equivalente a un tramo de tamaño l1 + l2.

donde + y > tienen su significado habitual en matemáticas.

2.1.3.3. CALIDAD DE LA MEDICIÓN

Para establecer la calidad de una medición se necesita comprender lo que es la validez de una medición y para ello se necesita comprender qué son los conceptos de escala, unicidad, representatividad y significado.

2.1.3.3.1. ESCALA

Dados los conjuntos *A* (sujeto de la medición de la propiedad *K*), *A'* (su homomorfo), los sistemas relacionales empírico y formal (*E* y *F* respectivamente) y la transformación *T* empleada, se dice que *S = (E, F,T)* es una escala de medición de la propiedad *K* respecto del conjunto *A*.

Es evidente que podemos tener muchos sistemas formales distintos y cada uno de ellos es una escala distinta de la misma medición.

Una escala es regular cuando los objetos medidos guardan unicidad en el sentido de que si dos objetos tienen el mismo valor para una propiedad concreta en la misma escala, también tienen el mismo valor en cualquier otra escala homomórfica con la primera, esto es: Sean dos homomorfismos *(E, F, T)* y *(E, F', T')*; $\forall a,b \in A$, $T(a) = T(b) \Leftrightarrow T'(a) = T'(b)$.

Una clasificación de las escalas fue hecha por Stevens ([STEV46], [STEV59]) en sendos trabajos, de acuerdo con ellos se pueden distinguir cinco tipos principales de escala:

- *Escalas nominales*.- Tales escalas se usan para denotar pertenencia a una clase (color, etiquetado,...). No hay operaciones en los conjuntos *O* ni *O'* y la única relación posible es la equivalencia. La unicidad se asegura porque la transformación *T* es de uno a uno.

- *Escalas ordinales*.- Se usan cuando las mediciones necesitan establecer juicios comparativos y ordenaciones sobre los valores de un atributo. Las relaciones son transitivas, antisimétricas e irreflexivas. La escala se preserva bajo transformaciones monótonas. Para ordenaciones débiles $x \geq y \Leftrightarrow T(x) \geq T(y)$. Para ordenaciones fuertes $x > y \Leftrightarrow T(x) > T(y)$.

- *Escalas de intervalo*.- Se utilizan para la medición de *distancias* entre pares de elementos de una clase de acuerdo a un atributo concreto. La escala se conserva bajo transformaciones lineales positivas de la forma $T(x) = \alpha m + \beta,\ \alpha > 0$.

- *Escalas de ratio*.- Son las más frecuentemente usadas en medición. Los números asignados a cada objeto del conjunto son grados en relación a un estándar usado como base de las mediciones. Las transformaciones deben conservar el origen aunque pueden modificar la pendiente de la escala: $T(x) = \alpha m,\ \alpha > 0$.

- *Escalas absolutas*.- Son escalas de ratio pero en donde el valor asignado es también un estándar, de modo que no admiten transformación posible, la única transformación de escala es la identidad: $T(x) = x$.

Esta división no es exhaustiva y es posible otro tipo de escalas regulares.

2.1.3.3.2. UNICIDAD

A pesar de que se pueden establecer escalas distintas, es importante que la medida sea única, y de hecho lo debe ser dentro de cada una de las escalas posibles.

En cualquier caso los grados de libertad en la elección del conjunto *F* tienen su origen en el concepto de escala, cada una de ellas es un homomorfismo y los homomorfismos rara vez son únicos.

A modo de ejemplo se puede medir una propiedad *coste* en distintas monedas (peseta, libra, dólar, marco) pero el significado en todas ellas es el mismo aunque el valor observable parezca distinto. En este caso la elección de uno u otro homomorfismo es una pura cuestión de conveniencia.

2.1.3.3.3. REPRESENTATIVIDAD

Define las condiciones en que debe definirse la función de transformación *T* para que el homomorfismo entre el sistema relacional empírico y formal aseguren la correspondencia entre las propiedades observadas y el sistema formal de representación elegido. Por tanto responde a la pregunta: ¿bajo qué condiciones es posible la medición?

La respuesta se debe dar de manera formal, como una serie de condiciones necesarias y suficientes que se puedan probar y que aseguren el homomorfismo. Esas condiciones serán los axiomas de representación de la medición.

El problema de la representación puede ser atacado de dos formas:

- *Constructivamente*: en este caso se trabaja hacia adelante, esto es, se busca el sistema formal a partir del empírico. En este caso se tiene mayor control sobre lo que se está realizando y los resultados y el nivel de adecuación son previsibles.

- *Retrospectivamente*: en este caso se trabaja hacia atrás, el sistema empírico y formal ya existen (a veces de forma intuitiva). No se tiene tanto control sobre los sistemas y puede que no sea posible conocer las condiciones de representación y unicidad. La calidad de la medición sólo se puede establecer *post facto*, en la etapa de validación.

Formular las condiciones de representación exige realizar pruebas en los datos tales que las condiciones de medición y los tipos de escala se puedan articular. Por otro lado, también es posible realizar pruebas de existencia para demostrar que el homomorfismo es posible.

2.1.3.3.4. SIGNIFICACIÓN

La significación es un concepto muy amplio que se aplica a diferentes conceptos dentro de los sistemas de medición. En el contexto actual se entiende por significación a la adecuación de las escalas al valor del atributo que se pretende medir, de manera que la medida realizada nos dé la información que estamos buscando.

Roberts propone una *teoría de la significación* para diferentes escalas de medición directa de propiedades simples. En el caso de escalas regulares ofrece la siguiente definición:

> *Una afirmación que incluye escalas (numéricas) es significativa si y sólo si su valor (verdadero o falso) es un invariante bajo transformaciones admisibles.*

Aceptando la siguiente definición de transformación admisible:

> *Sea f un homomorfismo de un sistema relacional A en otro B. Sea φ una función definida en todo el rango de f, esto es $f(A) = \{f(a) : a \in A\} \in B$. Entonces la composición $f \circ \varphi$ es una función de A en B. Si $f \circ \varphi$ es tam-*

bién un homomorfismo, entonces se dice que φ es una transformación de escala admisible.

Por ejemplo se puede considerar que *f* sea la longitud de la tubería en metros, φ sea la función que consiste en multiplicar por 3,281 (factor de conversión a pies) y $f \circ \varphi$ es, entonces, la longitud de la tubería en pies.

Para que una medición sea significativa, la escala debe ser adoptada en las primeras etapas de la planificación de la medición (de forma constructiva), o bien puede ser deducida en las últimas etapas de la validación de la escala (de forma retrospectiva).

A modo de ejemplos de mediciones significativas y no significativas, se puede tomar las tablas realizadas por Fenton en 1988. Algunos ejemplos están listados a continuación:

- *El punto A está al doble de distancia del punto B*.- No es significativa en una escala de ratio porque no existe un punto de referencia. Sería significativa si se tomase nuestra posición actual como punto de referencia.

- *El coste de producción del programa A es el doble que el de producir el programa B*.- Es significativa si el coste está definido en una escala de ratio.

- *Cuesta tres meses escribir el programa A*.- Es significativa para un intervalo de tiempo definido en una escala de ratio.

- *La calidad del programa A es del 97%*.- No es significativa porque no hay referencia a la escala, además no hay consenso acerca de cómo se define la calidad de un programa.

- *La complejidad de este programa es 42*.- No es significativa porque no hay referencia a la escala y además la complejidad de un programa no ha sido definida.

- *La profundidad de este árbol es 5*.- Es significativa, la escala es absoluta pues la profundidad de un árbol está perfectamente definida.

A la vista de estos resultados se comprende que la mayoría de las métricas usadas en software ni siquiera son significativas.

2.1.4. MEDICIÓN ORIENTADA A LA CARACTERIZACIÓN DEL OBJETO

La medición de una simple propiedad es un proceso raro habitualmente debido a varias razones:

- La medición directa de una propiedad es, normalmente, costosa, inconveniente y hasta incluso imposible. En estos casos, en lugar de medir directamente los resultados, es más interesante observar una o más

propiedades directamente y a partir de ellas derivar el resultado de propiedad deseada.

- La medición rara vez está enfocada a una sola propiedad, sino que habitualmente se desea una visión de varias propiedades para clasificar adecuadamente una entidad. Además sólo unas pocas de ellas se miden directamente, obteniéndose otras indirectamente, a partir de las medidas directas.

Por todo ello es más normal la combinación de mediciones directas e indirectas de varias propiedades con el fin de obtener una visión global de las entidades que se estudian. A modo de ejemplo, en software, no se puede hacer un estudio de la calidad de un programa basándose sólo en una característica (p. e. la limpieza de escritura del código), sino que es preciso estudiar más factores (reusabilidad, corrección, seguridad, compatibilidad,...) para poder emitir una opinión acertada.

La medición orientada a objeto consiste, pues, en la caracterización de una entidad mediante la elaboración de un modelo que recoja todas las particularidades y propiedades relevantes del objeto en estudio. Esto implica la construcción de perfiles de los objetos a través de la medición de sus propiedades (de forma directa o indirecta). Estos perfiles contienen entonces información que permite su clasificación o caracterización con el objeto de tomar decisiones respecto de ellos.

2.1.4.1. PREPARACIÓN DEL MODELO PARA MEDICIÓN ORIENTA A LA CARACTERIZACIÓN DEL OBJETO

Se parte de un conjunto de objetos $A= \{a, b, \ldots, z\}$ que se desea caracterizar mediante la medición de algunas de sus propiedades, una familia de propiedades a medir $K = \{k1, k2, \ldots, kn, \ldots, kq\}$ y un conjunto de medidas por cada objeto $M = \{m1, m2, \ldots, mn, \ldots, mq\}$. Todo ello puede ser representado en una matriz:

	m1	*m2*	...	*mn*	...	*mq*
a	*m1(a)*	*m2(a)*	...	*mn(a)*	...	*mq(a)*
b	*m1(b)*	*m2(b)*	...	*mn(b)*	...	*mq(b)*
...	...	...	...	...	...	...
z	*m1(z)*	*m2(z)*	...	*mn(z)*	...	*mq(z)*

Tabla 2-1. Perfil del modelo orientado a objetos.

En esta matriz las columnas *1* a *n* representan las propiedades leídas directamente y las columnas *n+1* a *q* son las propiedades medidas indirectamente o derivadas. Estas mediciones indirectas requieren que el sistema guarde no sólo las propiedades primitivas sino también sus interrelaciones para obtener las propiedades medidas indirectamente.

Para definir formalmente el sistema se partirá de los conjuntos definidos al principio de esta sección. Sea $Ai = \{ai, bi, \ldots, zi\}$ el modelo de *A* con respecto a la propiedad

i. De este modo el sistema relacional empírico *Ei = {Ai, Ri, Oi}* modela el conjunto *A* respecto a la propiedad *i*, preservando las relaciones y operaciones asociadas a esta propiedad y que pueden ser distintas para otras propiedades medidas. *Si = {Ei, Fi, Ti}* será la escala para la propiedad *i*. Así pues el conjunto *AA = {A1, A2, . . . , An}* es el modelo n-dimensional de *A*, donde cada propiedad tiene su propia escala.

Sea, ahora, *M = {m1, m2, . . . , mn}* el conjunto de mediciones directas que se pueden realizar sobre las propiedades de los objetos del conjunto *A*, si se define un función *g* en *AA*, cuyos términos pertenezcan a *M*, entonces se definirá *g* como una medida indirecta de los objetos de *A* en función de las propiedades primitivas.

2.1.4.2. SIGNIFICACIÓN

Es fácil, en general, definir medidas indirectas, a partir de la definición basta con establecer una función cualquiera que tome los valores de las variables de entre las propiedades medidas directamente en *A*. Pero para que esas nuevas medidas tengan significado, las medidas indirectas deben representar propiedades empíricas significativas, esto es, deben reflejar leyes empíricas o teóricas que sean válidas y se puedan obtener a partir de los valores medidos directamente en el conjunto *A*. Estas leyes deben ser incluidas en el modelo *AA*, de manera que este modelo describa las propiedades directas e indirectas de los objetos del conjunto *A*.

2.1.4.3. REPRESENTATIVIDAD

Las condiciones de representatividad de una medición indirecta orientada a objeto deben ser de una de estas dos clases:

- Condiciones heredadas.- que son las condiciones que cada medida primitiva debe satisfacer individualmente.

- Condiciones interrelacionadas.- que permiten deducir el significado de una medida indirecta como una función de las medidas primitivas, de acuerdo a las leyes impuestas por el modelo.

La notación adoptada por Roberts da a una condición de representación de una medida indirecta sencilla *g* la siguiente notación: *C(m1, m2, . . . , mn, g)*, donde los símbolos del argumento son los explicados en secciones anteriores y *C* es un conjunto de condiciones necesarias y suficientes que la medida indirecta *g* debe satisfacer. Estas especifican las relaciones empíricas del modelo *AA* como una operación de interrelación n-aria *g = g(m1, m2, . . . , mn)*, en adición a las condiciones de representación heredadas que cada una de las medidas directas *(m1, m2, . . . , mn)* debe satisfacer individualmente.

Si la medición orientada a objeto requiere la caracterización del objeto mediante una o más propiedades medidas indirectamente (como se ha reflejado en la Tabla 2-1), las condiciones de representación individuales para cada medida indirecta, se deben reflejar de la forma vista anteriormente.

2.1.4.4. UNICIDAD

Al igual que en el caso de la representatividad, las condiciones de unicidad de una medición indirecta orientada a objeto deben ser de una de las dos clases vistas: interrelacionadas o heredadas.

La unicidad de las medidas primitivas da las transformaciones de escala permisibles de las medidas de *M*. A partir de ellas la escala de cualquier medida indirecta se deduce mediante las condiciones de interrelación.

A modo de ejemplo, si se desea medir la potencia de un circuito eléctrico a partir de la medición del voltaje y la intensidad, obtendríamos la medida indirecta:

$$p = v \cdot i$$

si los factores de escala son respectivamente *vs* e *is*, el factor de escala para la potencia será

$$ps = vs \cdot is$$

No obstante esto no es siempre tan sencillo, en algunos casos aparecen importantes problemas. Por ejemplo:

- Para medir la situación en el espacio de un sistema de satélites, se puede usar un sistema cartesiano cuyo centro se puede localizar en la propia estación base en la tierra. Si se desea mover dicha estación y con ella todo el sistema, el nuevo sistema sigue estando medido en una escala de ratio y las posiciones responderán a la función:

  $$\varphi(x) = \alpha x + \beta, \alpha > 0$$

  Sin embargo si lo que se desea es cambiar el sistema de medida a uno en coordenadas polares o circulares, intervienen funciones trigonométricas que es más complicado de manejar es este caso.

- Otros problemas de escala surgen en las telecomunicaciones donde unos parámetros se miden en función del tiempo y otros en función de la frecuencia y para obtener conclusiones se deben relacionar ambos.

2.1.5. MEDICIÓN ORIENTADA AL DOMINIO DEL SUJETO

Cuando se habla de medición orientada al dominio del sujeto (entendiendo por sujeto la materia objeto de estudio), se deben ampliar los horizontes respecto al tipo de medición orientada al objeto, para considerar el problema de caracterizar mediante medición cualquiera de los objetos de una disciplina.

Se podría pensar en principio que se trata de una forma muy ampliada de ver la medición orientada al objeto, obteniendo en este caso una matriz como la de la Tabla 2-1 pero mucho mayor, con todos las propiedades descritas en el pasado, pero la medición orientada al dominio del sujeto debe soportar también la posibilidad de incluir las propiedades y mediciones que se vayan a describir en el futuro. Por tanto esto

exige que la implementación práctica de las mediciones se realiza de forma flexible sobre todo el dominio del sujeto. Desde el punto de vista en que las disciplinas no conocen fronteras geográficas, la metrología orientada al sujeto debe tener un ámbito global y sus conceptos y métodos deben tener reconocimiento internacional.

La metrología orientada al dominio del sujeto debe ofrecer un entorno teórico para la representación y el escalado, adaptable a cualquier problema de medición individual del dominio. Este entorno debe proveer a la disciplina de:

- Un conjunto comprensible de conceptos básicos, definidos en términos medibles.

- Un sistema de teorías coherentes que liguen esos conceptos y que de ellas se puedan derivar nuevos conceptos para la medición orientada al dominio del objeto.

Si el entorno teórico de trabajo creado para la disciplina debe soportar mediciones en la práctica, debe contener métodos y procedimientos para realizar dicho trabajo. Debe, además, contener un conjunto de normas que, si el ámbito de aplicación en internacional, deben estar internacionalmente reconocidas.

Un sistema de este tipo tiene unas posibilidades enormes:

- Utilizaría un sistema internacional para el intercambio de información en el que estarían reconocidas las unidades de medida, lo que facilitaría la intercomunicación de resultados y la validación de teorías y permitiría el uso de unas unidades de medida internacionalmente reconocidas.

- Permitiría una cultura común que facilitaría la entrada de nuevos profesionales a la disciplina.

- El sistema de medida sería utilizable en entornos no sólo científicos, sino también comerciales y legales, permitiendo una comunicación entre cliente y vendedor al estar basado en estándares y normas.

- Más importante aún, el sistema de medida sería un servicio público, siendo un instrumento de medición de la calidad de los productos y servicios derivados de esa disciplina.

Todo lo dicho hasta ahora puede parecer excesivamente voluntarioso, no obstante hay un claro ejemplo de este tipo de metrología que, aunque es imposible demostrar su uso en el futuro, ha demostrado hasta ahora adaptarse bastante bien a la evolución tecnológica. Este sistema el *Sistema Internacional de Medidas*.

Es evidente que la informática está aún lejos de obtener un sistema similar, entre otras razones por una actitud de oscurantismo llevada a cabo por muchos desarrolladores en el pasado. Esto está mermando claramente los avances en sistemas de métricas adecuados y, sobre todo, internacionalmente reconocidos.

2.2. TEORÍA DE LA MEDIDA

2.2.1. INTRODUCCIÓN

Una medición es formal si:

- Las medidas son válidas de acuerdo con la teoría de la medición o las medidas se derivan de la teoría de la medida.

y

- Las métricas satisfacen los axiomas de la teoría de la medida.

Por tanto se manejan dos teorías, la teoría de la medida: una disciplina matemática y la teoría de la medición: una disciplina filosófica.

EL concepto de métrica se define sólo en la teoría de la medida: las métricas miden distancias.

La definición de una medida depende de la teoría utilizada.

2.2.2. MÉTRICAS Y PSEUDO-MÉTRICAS

Una pseudo-métrica es una función no negativa δ de dos variables que satisface las siguientes condiciones:

Para cada x, y, z

$\delta(x, y) = 0$ *identidad*

$\delta(x, y) = \delta(y, x)$ *simetría*

$\delta(x, y) \leq \delta(x, z) + \delta(z, y)$ *desigualdad triangular*

Una métrica es una pseudo-métrica que satisface un axioma de identidad más estricto:

$$\delta(x, y) = 0 \Leftrightarrow x = y$$

2.2.3. ESPACIO DE MEDIDA Y ESPACIO MEDIBLE

Un espacio medible es un par (X, S) que consiste en un conjunto X y una σ-álgebra S de subconjuntos de X. Todos los subconjuntos de X pertenecientes a S se dice que son medibles.

S es una σ-álgebra si:

$$X \in S;$$

$$\forall A, B \in S: \quad A - B \in S \wedge A \cup B \in S;$$

$$\forall A_i \in S: \qquad \bigcup_i A_i \in S$$

Un espacio de medida es una tripleta (X, S, μ), donde (X, S) es un espacio medible y μ es una medida definida en S.

2.2.4. MEDIDAS

Una medida μ es una función de conjuntos no negativa sobre la σ-álgebra S que satisface:

$$\mu(\varnothing) = 0$$

$$\mu(A) < \mu(B) \qquad si \qquad A \subset B$$

$$\mu\left(\bigcup_i A_i\right) \leq \sum_i \mu(A_i) \qquad \text{adición}$$

La adición es completa cuando los conjuntos son disjuntos:

$$\mu\left(\bigcup_i A_i\right) = \sum_i \mu(A_i)$$

2.2.5. CLASIFICACIÓN DE LAS MEDIDAS DE SOFTWARE

Las medidas de software se pueden clasificar atendiendo a dos criterios, por un lado como entidades pueden ser procesos, productos y recursos. Por otro como atributos pueden ser internas y externas. [FEN91] y [FEN96]

Procesos son actividades que tienen una cierta duración, p. e. la codificación, integración, etc.

Productos son resultados tangibles de los procesos, p. e. tipos abstractos de datos, objetos codificados, etc.

Recursos son entradas a procesos, p. e. personal, hardware, otro software, etc.

Atributos **internos** de una entidad son atributos que pueden ser medidos en términos de la entidad misma, p. e. tamaño, funcionalidad, esfuerzo, número de errores detectados, etc.

Atributos **externos** de una entidad son atributos que sólo pueden ser medidos en términos de otras entidades, p. e. mantenibilidad, comprensibilidad, calidad, coste, etc.

2.2.6. MODELOS DE ESTIMACIÓN DE COSTE DE SOFTWARE

2.2.6.1. INTRODUCCIÓN

La gran dificultad con la que se encuentran los desarrolladores de software para estimar adecuadamente los proyectos es un hecho que cuesta anualmente mucho dinero y hace que gran parte de los proyectos software de tamaño grande acaben siendo un estrepitoso fracaso. Sólo algunas excepciones (software de la olimpiada de Barcelona 92) se salvan de este desolador panorama [NOV92].

La dificultad propia del software, la inadecuación de la mayor parte de las métricas existentes y la atomización de esfuerzos para conseguir una sistema adecuado son algunas de las razones que explican esta situación.

Mientras esta situación persista será imposible estimar con rigor y fundamento el coste o el tiempo de elaboración de un proyecto software *a priori*.

La aparición de foros internacionales de discusión de métricas de software, no obstante, hacen ser optimistas sobre el futuro de esta situación.

2.2.6.2. EL PROBLEMA DE LA GESTIÓN DE PROYECTOS

La dificultad para poder estimar adecuadamente incide negativamente en tres áreas fundamentalmente: económica, técnica y de gestión. El impacto económico se nota con mayor fuerza cuando los sistemas son subestimados provocando pérdidas económicas importantes e incluso provocando que algunos proyectos no lleguen nunca a terminarse. Jones (1986) [JON86] ha estimado que el 15% de los grandes desarrollos nunca llegan a entregarse.

Además, cuando un programa se subestima, las últimas etapas del desarrollo no se completan con la necesaria atención técnica, lo que provoca que las fases de prueba y documentación sean las más afectadas produciendo un software de baja calidad, propenso a errores y difícil de mantener.

Por último, aunque no menos importante, cuando un proyecto se subestima provoca que los plazos de entrega no se cumplan y que la tendencia sea a sumar personal al proyecto. Este personal no está normalmente integrado en el proyecto y esto aumenta los tiempos de programación con lo que el efecto se realimenta y el proyecto se enlentece aún más.

2.2.6.3. DIFICULTAD DE ESTIMACIÓN DEL COSTE DEL SOFTWARE

Ante la situación anterior está claro que todos los directores de proyectos informáticos desearían terminar con esta situación, para ello es necesario identificar, cuantificar y contrarrestar las causas que la producen.

DeMarco, en su libro *Controlling Software Projects (1982)* [DEMA82], identifica varias de estas razones:

- La estimación es un proceso complicado para realizar el cual se necesita mucha información y suficiente tiempo. Desafortunadamente las

estimaciones se deben hacer siempre con prisa y con pocos datos pues están realizadas en las primeras etapas del desarrollo. Es muy común que se deba estimar sin saber siquiera todos los requerimientos y que además el trabajo lo realice una persona que no está aún suficientemente centrada en el proyecto.

- La falta de experiencia en la estimación de proyectos grandes es otro de estos factores. No es muy normal que las empresas manejen una gran cantidad de proyectos de tamaños grandes y por tanto la experiencia de la persona que realiza la estimación no es la adecuada.

- Existe una tendencia natural a subestimar el trabajo cuando éste es grande, si a esto le sumamos que normalmente el cliente no pide tiempos de ejecución sino que marca estos tiempos por conveniencia propia tendremos otra fuente importante de desviación temporal.

- Se menosprecian o no se tienen en cuenta las no linealidades de las duraciones de las tareas en proyectos grandes, de esta manera se suman algebraicamente los tiempos calculados sin tener en cuenta los problemas de acoplamiento entre tareas y personas. Además se suelen estimar los trabajos en función de las posibilidades de los analistas *senior*, pero después son programadores *junior* los que llevan a cabo el trabajo y los tiempos son mayores.

Por tanto se puede comprobar que es difícil realizar una estimación realmente aceptable pues son muchos los factores condicionantes y sobre algunos de ellos hay poco control por parte de las personas que tienen la responsabilidad de realizar dicha estimación.

2.2.6.4. ESTIMACIÓN DE COSTE ALGORÍTMICA SOBRE MODELOS

Hay varias maneras de acercarse a la estimación de costes de proyectos software, el juicio de los expertos, la analogía, modelos algorítmicos, etc.

La mayor parte de ellos dependen de las limitaciones humanas para manejar esta gran cantidad de información o bien necesitan personal muy experimentado, por tanto se han realizado modelos de decisión cuyo objetivo es principalmente realizar estas decisiones de una manera independiente y objetiva. En cualquier caso todos los modelos desarrollados hasta ahora siguen necesitando una importante intervención humana.

Los primeros trabajos en este área aparecieron en los finales de los 60 y principios de los 70. A partir del libro de Barry Boehm (*Software Egineering Economics, 1981*) [BOEH81] aparecen dos tipos de enfoque: la corriente económica representada por los trabajos en TRW e IBM (Wolverton, 1974 [WOLV74]; Walston y Felix, 1977 [WALS77] [@WAL96]) y la corriente de Rayleigh, representada principalmente por el trabajo de Putnam (1978) [PUTN78]. Una tercera corriente, la de los puntos función apareció en 1979, pero la mayor parte del trabajo de Albertch [ALBE83] y otros apareció en la década de los 80.

A principios de los años 90 aparece un enfoque orientado a objeto que en unas ocasiones aprovecha los modelos anteriores (puntos función en MOSES o SOMA) y otras realizan enfoques nuevos que se basan en la complejidad de los diseños orientados a objetos (Taylor [TAY93], Chidamber y Kemerer [CHI91] [CHI94] [CHI95],...).

2.2.6.4.1. LA CORRIENTE ECONÓMICA

Se caracteriza por el desarrollo de sistemas de métricas orientados a estimar el coste de los sistemas de software ya realizado. Los primeros trabajos son los de Sackman *et al.* [SACK68], Gayle (1971) y Chrysler (1978). La mayor parte del énfasis de estos trabajos está en la localización de variables que influyen más en el modelo y mucho menos en las métricas de entrada-salida o en la forma funcional del modelo. Por ejemplo Walston y Felix [WALS77] investigaron los efectos de aproximadamente 29 variables.

De todos estos modelos, el de mayor difusión y aceptación fue el *COnstructive COst MOdel* (COCOMO), desarrollado por Barry Boehm de TRW [BOEH81]. Basado en el análisis de 63 proyectos de software, Boehm desarrolló un modelo fácil de entender que predecía el esfuerzo y duración de un proyecto, basándose en el tamaño de los sistemas resultantes y una serie de *drivers* de coste que Boehm pensó que afectaban a la productividad.

La popularidad de este modelo en la industria se debe a dos factores clave: el primero es que el modelo se puede encontrar documentado en los libros, al contrario que otros sistemas que son propietarios (SLIM, Estimacs, RCA's price,...) y el segundo es que Boehm ha escrito un libro detallando totalmente el desarrollo y justificación del modelo.

2.2.6.4.2. LA CORRIENTE DE RAYLEIGH

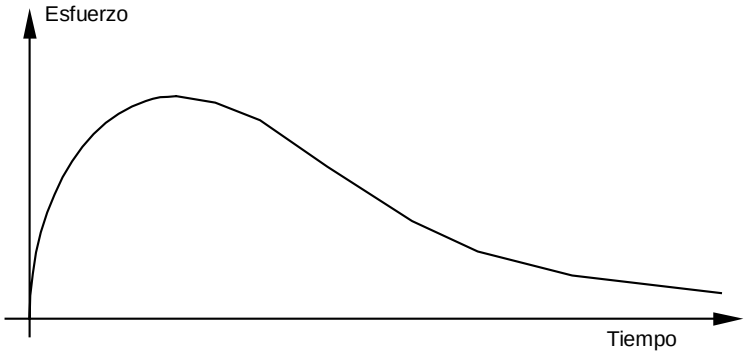


Ilustración 2-1. Curva de Rayleigh.

La curva de Rayleigh ha sido usada por diferentes desarrolladores, pero el sistema que más popularidad ha alcanzado basándose en ella ha sido SLIM (*Software Lifecycle Model*), método de estimación propietario, desarrollado a finales de los 70 (1978 y 1979) por Larry Putnam [PUTN78] de *Quantitative Software Management*.

2.2.6.4.3. LA CORRIENTE DE LOS PUNTOS FUNCIÓN

Una crítica muy severa de las dos corrientes anteriores, es que ambos modelos necesitan, para hacer estimaciones, que el usuario estime el número de líneas de código fuente (SLOC). El método de puntos función fue desarrollado por Albercht [ALBE79] en IBM y publicado por primera vez en 1979. El sistema desarrollado por Albercht está en un nivel superior de abstracción, al estimar sobre la base de requerimientos y análisis del proyecto en lugar de basándose en SLOC. Esto es más importante para alguien que desea estimar el coste o el esfuerzo de un proyecto en las primeras etapas de un desarrollo, además no necesita ser usado por personal demasiado especializado, y, además, elimina los efectos del lenguaje de programación y otras diferencias de implementación.

2.2.6.4.4. EL ENFOQUE ORIENTADO A OBJETO

En los años 90 aparece este nuevo enfoque, que si bien no se puede considerar una nueva corriente (en el sentido de que sus objetivos no son demasiado diferentes) si que se puede considerar aparte por dos razones principales: primero porque su enfoque es suficientemente nuevo y diferente de los anteriores y segundo por el número de desarrolladores implicados.

A modo de resumen se pueden citar MOOSE de Chidamber y Kemerer [CHI91] y MOOD de Brito e Abreu [BRI94]. En 1993 Li y Henry [LI93] desarrollaron una extensión de MOOSE en la que aumentaron el número de métricas y agregaron sistemas de predicción para mantenibilidad y reusabilidad. Bieman *et al.* en 1991 [BIE91] desarrollaron métricas para cohesión, reusabilidad, acoplamiento y herencia en sistemas orientados a objetos.

También existen versiones orientadas a objetos del sistema de puntos función, entre ellos Zhao-Stockman (Object Function Point Analysis) [ZHAO95], Banker *et al.* (Object Point Analysis) [BANK91], Thomson *et al.* [THOM94], Horner (MK II function points for OO) [HOR94], etc. Los dos sistemas más importantes son los de Henderson-Sellers y Edwards (MOSES) [HEN94] y Graham (SOMA) [GRA95].

Otra larga serie de desarrolladores se acercaron a las métricas orientadas a objeto, desde los primeros desarrollos en 1990 y los años 1992-1993 cuando las métricas orientadas a objeto fueron tomadas en serio hemos pasado a la situación actual con más de 200 publicaciones relacionadas con las métricas orientadas a objeto.

Una característica general de las métricas orientadas a objeto es la falta de formalidad, sólo unas pocas métricas están de acuerdo a la teoría de la medición y ninguna de acuerdo a la teoría de la medida.

2.2.6.5. EVALUACIÓN DE MODELOS

Una cuestión crítica es la decisión de usar y/o comprar un sistema para realizar las mediciones. Es importante conocer su adecuación y su fiabilidad. En principio a cualquiera de los sistemas se le supone con la fiabilidad necesaria y con la utilidad que se necesita, sin embargo no siempre es así, pero incluso los sistemas menos precisos son más interesantes que el no disponer de herramienta. En cualquier caso todos ellos

dos ellos necesitarán una calibración mediante la cual se adaptarán mejor a las necesidades requeridas.

Para evaluar adecuadamente una herramienta de medición lo que se puede usar es el grado de precisión con el cual dicha herramienta estima el esfuerzo en meses-hombre MH_e (por ejemplo), y compararlo con el esfuerzo real MH_a . Lógicamente si el sistema fuera perfecto, entonces debería ocurrir que $MH_e = MH_a$. Evidentemente esto raramente ocurre, si es que ocurre alguna vez.

Pero este tipo de estimación tiene un problema, y es que siempre hace una estimación absoluta del error y es evidente que es menos problemático que un sistema haya cometido un error de nueve meses-hombre tratando de estimar un proyecto de quince meses-hombre que si el proyecto fuera de mil meses-hombre. Por tanto es más indicativa una medida relativa del error, en concreto Boehm y otros aconsejan usar: $(MH_e - MH_a) / MH_a$

Este test elimina el impacto de la duración real del proyecto y estima mejor la desviación relativa. En cualquier caso, si se desea examinar una herramienta frente a una serie de proyectos para calcular una media de su error, la medida anterior tiene el problema de que puede tener signo (positivo si la duración estimada es mayor que la real y negativo en caso contrario), por lo cual al calcular una media se contrarrestarían los errores y eso no es interesante desde el punto de vista de que tan problemático resulta subestimar como sobrestimar las duraciones, por tanto una medida más interesante del error es la de tomar las desviaciones como valores absolutos: $MRE = |MH_e - MH_a| / MH_a$

De este modo las desviaciones de signo contrario no se compensan sino que incrementan el error producido como se deseaba.

Un problema que puede suceder es que una herramienta calibrada para un entorno más productivo sea usada en otro menos productivo o viceversa, por ello las herramientas deben poderse recalibrar para adaptarlas al entorno en el que se van a usar. Albercht y Gaffney [ALBE83] han propuesto la regresión lineal como un sistema de recalibración de estas herramientas. En este caso la variable independiente serían los meses-hombre-estimados como variable independiente y los meses-hombre-actuales como variable dependiente. En el caso concreto de los puntos-función, y puesto que no da una medida en meses-hombre, se puede usar también la regresión lineal tomando como variable independiente los puntos-función y como variable dependiente los meses-hombre.

2.2.6.6. TRABAJOS DE VALIDACIÓN

Descripción	*Golden et al.*	*Wiener-Ehrlich el al.*	*Kitchenham y Taylor*	*Mizayaki y Mori*	*Caccamese et al.*	*Keremer*	*Jeffery*
Publicación	*Database*	*IEEE Trans on Soft. Engin.*	*Journals of Systems & Software*	*Proc. of 8th Intl. Conf. on Soft. Engin.*	No publicado	*Comm. of the Assoc. for Comp. Mach.*	*IEEE Trans on Soft. Engin.*
Año	1981	1984	1985	1985	1986	1987	1987
Modelos usados	SLIM	Rayleigh (SLIM)	COCOMO, SLIM	COCOMO	COCOMO, SLIM	COCOMO, ESTIMACS, FP, SLIM	Rayleigh (SLIM)
Fuente de datos	Xeros	Bankers Trust Company	British Telecom, ICL	Fujitsu	Olivetti	Empresa consultora	Casas de software, gobierno, banca
Origen de los datos	USA	USA	UK	Japón	Italia	USA	Australia
Número de proyectos	4	4	33	33	3	15	47
Tipo de proyectos	-	Proceso de datos	Tiempo real, sistemas operativos	Aplicaciones de software	Sistemas de software	Proceso de datos	Proceso de datos
Lenguaje(s) usado(s)	-	COBOL, BASIC, FORTRAN, MACRO	S3 (ALGOL), COBOL, Ensamblador	COBOL, PL/1, Ensamblador, FORTRAN	PASCAL, PLZ, LIMO	COBOL, Natural, BLISS	COBOL, PL/1
Tamaño medio	-	46,4 KSLOC	11,9 KSLOC	-	37 KSLOC	186,6 KSLOC	27,3 KSLOC
Duración media	12,7 meses	-	11,7 meses	-	24,4 meses	14,3 meses	15,7 meses
Esfuerzo medio	100 meses de trabajo	171 meses de trabajo	46,2 meses de trabajo	-	104,3 meses de trabajo	219,3 meses de trabajo	65,3 meses de trabajo
Tipo de análisis	Sólo datos	Gráficos, regresión	Regresión	MRE, regresión	Gráficos	MRE, regresión	Regresión
Predicciones testeadas	Esfuerzo, duración	Esfuerzo por fase	Esfuerzo, duración	Esfuerzo	Esfuerzo	Esfuerzo	Productividad frente a tiempo consumido
Resultados	Media dentro del 10% de duración y 42% de esfuerzo con grandes variaciones.	La curva de Rayleigh es buena si sólo se tiene en cuenta el mantenimiento correctivo	Necesaria la calibración. Mal estimado el esfuerzo por fase.	COCOMO sobrestima el tiempo. Se eliminan 3 drivers de coste para mejorar el modelo	Esfuerzo por tiempo mal modelado a causa de que el desarrollo de los sistemas de software se comprime	Se valida el modelo FP. Los drivers de coste no resultan útiles. SLIM y COCOMO sobrestiman el tiempo.	No hay soporte para la noción de que la productividad se reduce si se reduce el tiempo.

Tabla 2-2. Proyectos de validación de modelos de estimación de coste. (*Software engineer's reference book*)

Varios son los trabajos de investigación que se han desarrollado por todo el mundo con la intención de validar los modelos de estimación de costes y/o esfuerzos.

La validación de estos modelos es interesante con el fin de demostrar hasta qué punto un modelo es de utilidad. Sin embargo la validación es un problema difícil debido a la necesidad de capturar grandes cantidades de datos de proyectos ya realizados. Estos datos pueden no estar actualizados y se necesitan por tanto mecanismos para ponerlos al día para que los resultados obtenidos sean de utilidad. Ambos procesos entrañan una gran complejidad y largos tiempos de proceso, sobre todo para los proyectos de mayor tamaño que, al final, son los más interesantes pues también son los más difíciles de estimar. En la Tabla 2-2 se puede ver un resumen de trabajos realizados hasta 1987.

2.3. METRICAS Y MEDICIÓN

2.3.1. INTRODUCCIÓN

El propósito de una métrica es el de conseguir que un producto software sea un *aparato de ingeniería*, cuya calidad pueda ser medida (por tanto asegurada) y que pueda ser producido eficientemente.

Las métricas de software se aplican a todas y cada una de las fases del desarrollo de software (especificaciones, análisis, diseño, codificación, pruebas, documentación,...), no sólo al producto final.

Pero, para usar las métricas adecuadamente, no es suficiente con medir los atributos cuantitativamente, sino que es necesario considerar un sistema de medidas cuyos valores proceden de, al menos, tres fuentes diferentes:

- *Edictos*.- Considerando como tales a las reglas y normas bajo las cuales se va a desarrollar el software, o las restricciones que el producto debe cumplir. Tales, pueden ser las decisiones comerciales, legislativas, etc.

- *Estimaciones*.- Esto ocurre cuando el valor de una métrica se necesita en un momento del desarrollo en que no es posible su lectura directa. Estas estimaciones deben estar basadas en el conocimiento del producto bajo desarrollo.

- *Mediciones*.- Estas métricas pueden ser tomadas directamente.

Los objetivos, las predicciones y los valores reales son necesarios para conocer adecuadamente el proyecto que se está desarrollando. Objetivos son las cosas que se ha planeado hacer, predicciones son las cosas que se piensa que pueden resultar y valores reales son las cosas que han resultado. Todas ellas son importantes para tener referencias con las que poder hacer estimaciones en otros proyectos futuros.

2.3.2. USO DE LAS MÉTRICAS

Hay dos clases de métricas de software importantes:

- Métricas que asisten en el *control o gestión del desarrollo* del proyecto.

- Métricas que son *predictores* (o *indicadores*) de las cualidades del producto.

En la práctica se usan las mismas métricas para ambos propósitos, pero la justificación y por tanto los criterios mediante los cuales las métricas deben ser seleccionadas y evaluadas son diferentes.

2.3.2.1. MÉTRICAS DE CONTROL

Las métricas de control no son exclusivas del software, cualquier proceso industrial podría estas controlado y gestionado con métricas similares.

Las métricas que son usadas con más frecuencia en el control de proyectos son las métricas relativas a recursos, tales como esfuerzo, desviaciones de tiempo, utilización de máquina para algunas actividades, etc.

Otras métricas que se usan para control son aquellas utilizadas para estimar la completitud de las tareas, tales como el porcentaje de módulos codificados, o el porcentaje de sentencias testadas. Estas comparan un tamaño total de una tarea prevista con el tamaño de la parte que se ha realizado hasta el momento.

El último tipo de métricas usadas en el control de proyectos son las relativas a defectos. Desde el punto de vista de la ingeniería, la detección y eliminación de defectos es el principal coste dentro del proyecto. Para comprender y controlar este coste es necesario registrar información acerca de la naturaleza y origen de los defectos y de los costes asociados con su detección y eliminación. Desde el punto de vista de la gerencia del proyecto, las actividades de detección y eliminación de defectos no se deben planear sin una estimación de los porcentajes de defectos esperados y del esfuerzo requerido para diagnosticar y eliminar los defectos. Por tanto es necesario basarse en los defectos de proyectos anteriores y monitorizar los porcentajes actuales para comprobar las posibles desviaciones de lo esperado.

2.3.2.2. MÉTRICAS DE PREDICCIÓN

Las métricas de predicción se utilizan para estimar las características finales del producto o para estimar objetivos para las métricas de control. Ejemplos de estas métricas son las métricas estructurales que se obtienen a partir de diagramas de flujo, DFD's, índices de documentos, etc.

El uso de métricas como predictores de la calidad de los productos descansa en tres aspectos:

- Las métricas miden alguna propiedad inherente del software o del documento.
- Estas mismas propiedades inherentes influencian las características del comportamiento del producto final.
- Las relaciones entre la métrica y la calidad final son conocidas (al menos de forma aproximada) y están representadas en términos de una fórmula o modelo.

En la práctica sólo los dos primeros puntos se tienen en cuenta, así, por ejemplo, en las métricas estructurales se asume, a menudo, que están relacionadas con la complejidad, donde la complejidad es una valoración subjetiva de la dificultad para crear o entender una parte de un programa. En su lugar se podría usar un término formal como la complejidad computacional. Cuanto más grande es el valor de la métrica mayor se supone la complejidad.

Se tiende a formalizar este tipo de suposiciones mediante fórmulas generales o modelos, ya que en la práctica es difícil encontrar métricas apropiadas para este tipo de problemas. En cualquier caso no se debe olvidar que la obligación de los desarrolladores de métricas es justificar y validar sus conclusiones.

2.3.3. RESTRICCIONES PRÁCTICAS

2.3.3.1. INTERPRETACIÓN DE LOS VALORES DE LAS MÉTRICAS

Uno de los mayores problemas en el uso de las métricas de software es que, debido a la complejidad innata del software, no es posible, en general tener interpretaciones de escala, por ejemplo, el valor de 10 en una métrica no tiene el mismo significado que un valor de 10ºC. Por tanto las métricas deben ser, en general interpretadas de forma relativa en una de estas tres formas:

- *Por comparación con los planes y las expectativas*.
- *Por comparación con otros proyectos similares*.
- *Por comparación con otros componentes similares dentro del mismo proyecto*.

Como se puede ver, la interpretación es siempre en función de un estándar o norma creada por anteriores proyectos similares, no es una interpretación directa del mundo real en el sentido en que puede interpretarse una medida de temperatura. Esto se debe, como se comentaba al principio, a la enorme complejidad del software, en este sentido una desviación del tiempo previsto para la codificación de un módulo puede estar causada por una baja productividad, pero también porque el diseño fue insuficiente, o porque se ha decidido usar otro algoritmo más optimizado, etc. En cualquier caso la métrica nunca dirá la causa del *no cumplimiento*.

2.3.3.2. PROCEDIMIENTOS DE RECOLECCIÓN DE DATOS

Una de las razones por las cuales las métricas no son usadas más ampliamente en la industria del software es que resulta difícil en la práctica obtener datos adecuados. Idealmente, los valores de las métricas deben ser repetibles, comparables y verificados. Repetibles implica que dos personas diferentes que hagan la misma medida, en las mismas condiciones deberían obtener los mismos valores. Comparables implica que los valores obtenidos de las mismas métricas deberían ser del mismo tipo. Verificados implica que los valores recogidos han sido chequeados para evitar errores e inconsistencias.

En la práctica compatibilidad y repetibilidad se aseguran mediante métricas formales bien definidas y procedimientos adecuados de recolección de datos.

El problema principal radica en que no existen procedimientos normalizados ni admitidos globalmente, sino que sólo se tienen sugerencias y algún intento normalizador de escaso éxito.

Estos procedimientos deberían recoger, al menos los siguientes requerimientos:

- *Definiciones de las métricas*.- Normalmente no es suficiente una simple definición de la métrica. Se debe definir también las unidades que se aplican, el tipo de software al que es aplicable y las condiciones en las cuales se deben recoger los datos.

- *Detalles organizacionales*.- Que deben identificar las personas responsables de recoger los datos, las personas responsables de verificarlos, la manera en que deben ser registrados y la forma en que deben ser verificados y analizados.

En la práctica la forma en que los datos deben ser registrados, verificados y analizados es fundamental para el éxito de cualquier recolección de datos. Kitchenham y McDermid [KITC86] sugirieron que la recolección para tener éxito debe cumplir:

- La recolección de datos está integrada dentro del proceso de desarrollo.

- Está automatizada siempre que sea posible.

- Los datos que no pueden ser recogidos automáticamente se recogen en el momento en que se producen (esto es, no se deben recoger datos de eventos pasados) y se verifican inmediatamente.

- Los tiempos entre la recogida de datos y el análisis de los mismos está minimizado.

- Los datos se tratan como recursos de la empresa y se dan facilidades para acceder a los datos históricos, no sólo a los recogidos en el proyecto actual.

- El problema de la motivación de los responsables del proyecto para recoger y registrar los datos no se subestima. El tratamiento adecuado de los datos y el uso de métricas de software junto con un rápido análisis es fundamental, pero no suficiente.

Ejemplos de esquemas de recolección de datos se pueden encontrar en Basili y Weiss [BASI82] y Kitchenham [KITC84].

2.3.3.3. ANÁLISIS DE LAS MÉTRICAS SOFTWARE

En general el análisis de los datos procedentes de las métricas se suele hacer utilizando algún tipo de técnica estadística convencional, como la media, la varianza, análisis de regresión y correlación para investigar la relación entre dos métricas, y análisis mediante tablas de contingencia para investigar cantidades y porcentajes dentro de un esquema de clasificación.

Sin embargo en la práctica, los conjuntos de datos procedentes de las métricas de software suelen tener una serie de características indeseables para este tipo de análisis

(discretos, no negativos, tienen mucho ruido,...). Además, estos datos raramente pueden ser tomados como procedentes de una población (estadística). Esto implica que el uso de técnicas estadísticas convencionales, basadas en la suposición de que los datos sean aleatorios y responden a una distribución de Gauss (esto es, simétrica, continua y respondiendo a una distribución de frecuencias) es, en general, irreal.

En general, según Hoaglin *et al.* [HOA00], parece que las técnicas *EDA* (*Exploratory Data Analysis*) desarrolladas por J. W. Tukey, parecen ser más adecuadas para el tratamiento de este tipo de datos.

2.3.3.4. EL ÁMBITO DE LAS MÉTRICAS

Las métricas actuales, en general, tienen un ámbito de uso limitado, esto es, afectan a algunas fases del ciclo de vida de una aplicación. Hay métricas para las fases de diseño, de codificación y pruebas, de especificación de requerimientos, etc. pero no hay una métrica integrada que recoja todo el ciclo de vida.

Las métricas deberían abarcar todo el ciclo de vida, ser independientes de los lenguajes (adaptarse a todos ellos), de los métodos de análisis y diseño, y ser dinámicas para adaptarse a cualquier innovación tecnológica.

3. PRINCIPALES SISTEMAS DE MÉTRICAS. ESTADO DEL ARTE

Hay muchas publicaciones, artículos y conferencias sobre métricas del software donde se abordan los aspectos técnicos de la medición; pero no existen guías claras para la implantación de programas de métricas en las empresas.

(Ignacio Pérez y Pedro L. Ferrer)

3.1. MÉTRICAS PARA CONTROL DE PROYECTOS

Las métricas para el control de proyectos se pueden agrupar en cinco tipos:

- Métricas para documentos de especificación y diseño.
- Métricas relativas a módulos.
- Métricas para actividades de chequeo y revisión (que pueden tener referencias cruzadas con las anteriores).
- Métricas para fallos y cambios (que pueden tener referencias cruzadas con las anteriores).
- Métricas relativas a los recursos.

Las métricas para documentos de especificación y diseño son similares para ambos tipos de documentos, se incluyen:

- Tamaño, medido usando palabras, frases o páginas.
- Estructuras, obtenidas de representaciones gráficas de especificaciones y diseños.
- Legibilidad, medida usando el índice Fog [GUNN62].
- Estabilidad, obtenida de los datos cambiados (número y naturaleza de las modificaciones en documentos).
- Información sobre transformaciones, tales como la relación de crecimiento del tamaño de los documentos de especificaciones y diseño.

Las métricas de módulo incluyen:

- Datos sobre el encadenamiento de los módulos, medidos mediante el fan-out (número de módulos llamados desde uno concreto) y fan-in

(número de módulos que llaman a uno concreto). También el número de accesos de lectura y escritura a items comunes.

- Características de la interfaz, tales como el número de parámetros por cada módulo y el número de diferentes estados que el módulo puede tener.

- Características internas, tales como el número de instrucciones de diseño, número de líneas de código, flujo de control (medido mediante las métricas de McCabe [MCC76]), complejidad de los datos (medida por el número de datos elementales accedidos) y características de transformación (medidas mediante la relación entre el número de sentencias de diseño y el número de líneas de código generadas).

También es importante monitorizar la conformidad entre las fases en el ciclo de desarrollo, comprobando, por ejemplo, el número de módulos identificados durante el diseño que no se implementan y viceversa: el número de módulos implementados no previstos durante el diseño.

Las métricas de prueba incluyen:

- Comprobar el número de casos de prueba seleccionados, el número de los realizados, los realizados que no estaban previstos y los previstos que no se han realizado.

- Métricas de cobertura de sentencias, condiciones y decisiones cubiertas por los tests.

Las métricas de fallos y cambios incluyen:

- Número de fallos y cambios clasificados por la fase en que fueron introducidos, la fase en la que fueron detectados y el tipo de fallo.

- Información de control indicando el número de cambios realizados y de fallos sin resolver al final de una fase concreta.

Las métricas de recursos incluyen:

- Esfuerzo por actividad.

- Tiempo empleado por actividad.

- Niveles de dirección durante el desarrollo.

Estos tipos de métrica se pueden usar para ayudar en el control del proyecto durante todo el ciclo de vida. Algunas métricas son similares en varias fases distintas (ejemplo las métricas de fallos y cambios). Las métricas dependientes de la fase son más problemáticas y necesitan procedimientos de recogida de datos particulares.

3.2. PRINCIPALES MÉTRICAS CLÁSICAS

3.2.1. INTRODUCCIÓN

Desde que se comenzó con el desarrollo de las métricas de software, han sido muchos los trabajos de investigación en este sentido, algunos de los cuales han dado lugar a métricas ampliamente usadas (dentro del escaso uso de las métricas en general).

En los siguientes apartados se presentan algunas de las métricas clásicas más conocidas. Se ha eliminado de esta lista a las métricas orientadas a objeto que serán tratadas en otro capítulo.

3.2.2. COCOMO

COCOMO (COnstructive COst MOdel) [@CYR96], fue desarrollado por Barry Boehm [BOEH81], de TRW y publicado por primera vez en 1981.

Una versión simplificada de la ecuación de esfuerzo de COCOMO para el modelo básico es de la forma:

$$WM = C(KDSI)^k$$

donde *WM* es el número de meses de trabajo, *C* es una constante que depende del modelo de desarrollo, *KDSI* son miles de instrucciones entregadas en el proyecto y *k* es otra constante que depende del modelo de desarrollo.

Boehm define *DSI* como instrucciones de código fuente entregadas, esto es, instrucciones del programa creadas por el personal del proyecto que son entregadas con el producto final. Esto incluye lenguaje de control del trabajo, sentencias formales y declaraciones de datos, y excluye comentarios y utilidades de software no modificadas.

COCOMO soporta tres modelos de desarrollo distintos: *organic*, *semi-detached* y *embedded*.

- El mejor caso es el *organic*, donde equipos de software relativamente pequeños trabajan en un entorno muy familiar y privado. Su límite está en torno a las 50 KDSI.

- El modo más complicado es el *embedded*, donde el proyecto debe trabajar bajo severas restricciones, tales como acoplamiento estricto a otro software o hardware preexistentes.

- Un tercer modo que Boehm llama *semi-detached*, tiene un grado de complejidad intermedio a los dos anteriores. Puede se intermedio por una de dos razones: 1) El proyecto tiene un nivel intermedio ó 2) es una mezcla de los dos casos extremos. El rango de este modelo va hasta los 300 KDSI.

Durante el desarrollo de COCOMO, Boehm detectó que el sistema básico calculaba el esfuerzo de forma razonable entre el 29% y el 60% de las veces dependiendo del

factor usado. Para mejorar estos resultados introdujo los efectos de 15 *drivers* de coste que son atributos del producto final, del ordenador usado, del personal y del entorno del proyecto. Él pensó que estos 15 factores afectaban a la productividad del proyecto y llamó a esta versión el modelo intermedio. La nueva ecuación es ahora:

$$MM = C(WDSI)$$

donde

$$WDSI = (KDSI)^{e_i} \prod_{j=1}^{15} EM_j$$

y *WDSI* son las instrucciones ponderadas de código fuente entregadas, e_i es el exponente para el i-ésimo modelo de desarrollo empleado y EM_j es un multiplicador de esfuerzo determinado por el j-ésimo driver de coste.

Los valores de EM_j varían entre 0,7 (para un producto de muy baja complejidad) hasta 1,66 (un proyecto desarrollado para un computador con muy severas restricciones de tiempo). En general, multiplicadores mayores de 1 indican que el equipo es menos productivo en ese proyecto y multiplicadores menores que 1 indican factores que hacen al equipo más productivo.

El modelo detallado de COCOMO es muy similar al modelo intermedio, excepto que los proyectos son divididos en cuatro fases, diseño del producto, diseño detallado, codificación/pruebas unitarias e integración/pruebas finales. Los 15 factores son estimados para cada fase en concreto en lugar de hacerlo para el proyecto en conjunto.

3.2.3. SLIM

SLIM fue desarrollado por Larry Putnam en 1978-79 [PUTN78]. Al igual que el COCOMO su estimación se basa en las líneas de código fuente (SLOC, análogas a las DSI de Boehm). Esta estimación inicial del tamaño se modifica mediante el modelo de la curva de Rayleigh para estimar el esfuerzo. El usuario puede influenciar el resultado mediante dos parámetros: la inclinación inicial de la curva (*MBI, manpower buildup index*) y un factor de productividad o constante tecnológica (*PF, productivity factor*).

La curva de Rayleigh es una curva exponencial bien conocida usada para modelar procesos de desarrollo. Tiene el esfuerzo en el eje vertical y el tiempo en el eje horizontal.

Mientras la curva de Rayleigh describe el fondo teórico del modelo, la mayor parte de la potencia de SLIM reside en su ecuación de software:

$$S = cK^{1/3}t_d^{4/3}$$

donde *S* son las sentencias fuente, *c* es la constante tecnológica (*PF*), *K* es el esfuerzo del ciclo de vida y t_d es el tiempo de pico del rendimiento personal.

Claramente un valor más alto de la constante tecnológica permite escribir más líneas de código con el mismo esfuerzo y en el mismo tiempo.

El usuario de SLIM tiene control sobre dos variables *MBI* que es K/t_d^2 y *PF* que es *c*. *MBI* ajusta la inclinación inicial de la curva de Rayleigh, por tanto cuanto mayor sea este valor, más inclinada será la curva y más rápido se hará el proyecto.

EL usuario de SLIM puede elegir estos valores de dos formas: calibrando el modelo para su compañía o bien contestando a una serie de 22 cuestiones de las cuales SLIM recomendará un valor de *MBI* y de *PF*.

3.2.4. COMPLEJIDAD CICLOMÁTICA DE MCCABE

Esta medida se deriva del grafo de flujo de control de un programa y mide el número de caminos independientes a través de un programa. Es un indicativo de la complejidad y testabilidad de un programa. También es usado como un método de evaluación de la completud de testeo de las sentencias de un programa.

El Número Ciclomático [MCC76] se determina con la siguiente fórmula:

$$V(G) = e - n + 2p$$

donde *e* es el número de nodos del grafo de control, siendo un nodo un punto de bifurcación del programa. *n* es el número de vértices del grafo de control, siendo un vértice equivalente a un grupo de sentencias secuenciales dentro del programa. Por último, *p* es el número de componentes conectados, normalmente 1.

En programas estructurados, *V(G)* es equivalente al número de predicados más 1, donde los predicados compuestos como *IF a AND b THEN* se tratan como dos. Esto es equivalente al número de decisiones del programa.

La métrica tiene una serie de dificultades y ha sido criticada por varios desarrolladores (Shepperd, [SHE88]). Los problemas generales son:

- La métrica ha sido asociada con la tasa de errores en módulos, sin embargo no ha sido demostrado que provea mejor información que otros métodos.

- Hay dificultades prácticas cuando se trata de determinar el grafo de algunos programas y es que a menudo el grafo de un programa equivalente se obtiene más rápido que el del original. Esto hace pensar sobre qué es lo que realmente se está calculando.

- La métrica resulta muy superficial tratando ciertos tipos de problemas, por ejemplo se obtiene el mismo valor con tres bucles en secuencia que con tres bucles anidados.

- Un problema fundamental con el uso de la métrica como una medida de la testabilidad del programa es que se basa exclusivamente en el flujo de control y obvia el flujo de datos. Hay muchos programas que

pueden ser escritos evitando el uso de estructuras de control, mediante tablas y arrays u otras prácticas de programación orientada a los datos (Humpphrey, [HUM86]), por tanto el valor de la métrica puede estar muy influido por el estilo de programación.

3.2.5. MÉTRICAS BANG DE DEMARCO

Las métricas Bang de DeMarco, [DEMA82] se basan en medidas tomadas a partir de los Diagramas de Flujo de Datos y de los Diagramas de Entidad-Relación. Este método se puede usar cuando se utiliza la metodología de análisis y diseño de DeMarco.

Las primitivas de las métricas Bang son:

- El número de Primitivas Funcionales (*FP*) que es el número de círculos en el DFD.
- TC_i, que es el número de tokens de datos asociados con cada *FP*. Los tokens de datos son los items de datos manipulados por una primitiva funcional. Un token de datos compuesto puede ser tratado como 1 si se manipula como un simple elemento o como 2, 3, ... si se manipula disgregado en 2, 3, ... elementos.
- *DEO*, el número de elementos de salida que cruzan la frontera hombre-máquina.
- *OB*, el número de objetos en el modelo de datos.
- *RE*, el número de relaciones entre objetos en el modelo de datos.
- RE_i, el número de relaciones asociadas con cada objeto en el modelo de datos.

Un proyecto *fuertemente funcional* tiene $RE / FP < 0{,}7$, un modelo *fuerte de datos* tiene $RE / FP > 1{,}5$. Los modelos *híbridos* tienen valores intermedios en el rango 0,7-1,5.

Una métrica Bang para un proyecto *fuertemente funcional* se deriva de las medidas del DFD. Una métrica para un modelo *fuerte de datos* se deriva de medidas del modelo E-R. Un modelo *híbrido* necesita ambos tipos de métricas Bang.

Para productos *fuertemente funcionales*, las métricas Bang se basan en el conteo de de tokens de cada primitiva funcional, TC_i y un ajuste de complejidad basado en el tipo de función (ver Tabla 3-1). Antes de la corrección de complejidad, la cuenta de tokens se corrige mediante la fórmula de Halstead [HALS77]:

$$CTC_i = TC_i \times \log_2(TC_i)$$

El conteo de tokens corregido se multiplica por el peso de su corrección de complejidad y la métrica Bang es la suma de todos los CTC_i ponderados:

$$Bang = \sum_i w_i \times CTC_i$$

donde los factores de complejidad se pueden ver en la siguiente tabla:

Clase	*Peso*	*Clase*	*Peso*
Separación	0,6	Sincronización	1,5
Amalgama	0,6	Generación de salidas	1,0
Dirección de datos	0,3	Display	1,8
Actualización simple	0,5	Análisis tabular	1,0
Gestión de almacenamiento	1,0	Aritmética	0,7
Edición	0,8	Computación	2,0
Verificación	1,0	Inicialización	1,0
Manipulación de texto	1,0	Gestión de dispositivos	2,5

Tabla 3-1. Varios factores de complejidad ponderados para diferentes clases de función. (DeMarco, 1982)

Para los productos fuertemente de datos, las métricas Bang se basan en el conteo de RE_i. El factor de correlación se muestra en la Tabla 3-2. DeMarco no especifica cómo se generó la tabla, pero parece ser que fue obtenida a partir de la fórmula del volumen de Halstead para $RE_i + 1$.

RE_i	*OB Corregido*
1	1,0
2	2,3
3	4,0
4	5,8
5	7,8
6	9,8

Tabla 3-2. Pesos para los sistemas fuertes de datos. (DeMarco, 1982)

Para los modelos *híbridos* se deben calcular ambas métricas y las estimaciones de coste deben ser realizadas usando ambas independiente y separadamente.

Las métricas Bang tienen algunos problemas. No son simples cuentas, están basadas en complejas tablas de pesos. Los pesos para los modelos *fuertemente funcionales* son subjetivos y dependientes del entorno. Los conteos básicos de las métricas Bang están basados en dudosos conceptos de las métricas científicas de software no demostrados.

3.2.6. PUNTOS FUNCIÓN

El sistema de puntos-función fue desarrollado por Allan Albercht [ALBE79] de IBM y publicado por primera vez en 1979.

El proceso se desarrolla en dos pasos: 1) contar las funciones de usuario y 2) ajustar el sistema para la complejidad del entorno del proceso. Hay cinco funciones de usuario actualmente (aunque sólo había cuatro en el original de Albercht):

- Tipos de entradas externas.
- Tipos de salidas externas.
- Tipos de ficheros lógicos internos.
- Tipos de ficheros de interfaz externos.
- Tipos de consultas externas.

Todos estos tipos se cuentan y se ponderan de acuerdo a la Tabla 3-3. El total que resulta es el número de puntos función.

	Simple	*Medio*	*Complejo*
Entradas externas	x 3	x 4	x 6
Salidas externas	x 4	x 5	x 7
Ficheros lógicos internos	x 7	x 10	x 15
Ficheros de interface externos	x 5	x 7	x 10
Consultas externas	x 3	x 4	x 6

Tabla 3-3. Ponderación del sistema de puntos función.

Por tanto:

$$FC = \sum_{i=1}^{5} \sum_{j=1}^{3} w_{ij} x_{ij}$$

donde w_{ij} son los pesos y x_{ij} son las cantidades.

IBM ha dado algunas instrucciones acerca de como evaluar la complejidad de los tipos con el fin de elegir adecuadamente los pesos (Albercht, 1984).

Albercht también encontró que la complejidad y el esfuerzo dependen del entorno de trabajo, por lo cual introdujo una lista de 14 características de complejidad del proceso que pueden influir en una escala de 0 (ninguna influencia) a 5 (mucha influencia). Estas características tienen mucho que ver con los *drivers* de coste de Boehm, que valoran la capacidad y experiencia del equipo de trabajo.

El segundo paso consiste en sumar todos los puntos asignados a la complejidad del proceso:

$$PCA = 0{,}65 + (0{,}01)\sum_{i=1}^{14} c_i$$

donde *PCA* es el ajuste de complejidad del proceso y varía entre 0,65 y 1,35, y c_i son los factores de complejidad y varían entre 0 y 5. La ecuación final es:

$$FP = FC \times PCA$$

donde *FP* son los puntos función y *FC* los puntos contados en el primer paso.

El resultado final es que los puntos función pueden variar entre ±35% de los puntos del primer paso. A través de ellos una empresa puede valorar la complejidad de sus proyectos, al principio por analogía y después, a medida que aumenta el número de datos mediante técnicas estadísticas.

3.2.7. OTRAS MÉTRICAS CLÁSICAS

Además de las anteriores se han desarrollado una serie de trabajos que dieron lugar a otras métricas menos conocidas o a modificaciones y adaptaciones de las anteriores.

A modo de ejemplo, después de desarrollados los puntos función, se desarrollaron una serie de modelos propietarios que adoptaron los puntos función como medida, entre ellos Estimacs (Rubin, [RUB83]) y SPQR (Jones, [JON86] [JON88]). Symons, en [SYM88], también publicó un trabajo de evaluación crítica de los puntos función.

Otras métricas se basan en modelos *científicos*, en el estudio de las métricas de la documentación de programas o en la estructura del diseño.

3.2.7.1. MÉTRICAS CIENTÍFICAS DE SOFTWARE

Un enfoque distinto de métrica de código son las Métricas Científicas de Software, que tratan de identificar una serie de atributos del software a partir de un pequeño número de métricas simples derivadas de la implementación en un lenguaje de programación. Estos atributos incluyen el tamaño, esfuerzo mental para crear el programa, tiempo para crear el programa y el número de errores cuando se entrega el programa.

Éstas métricas se basan en cuatro mediciones fundamentales:

- n_1, definido como el número de operadores diferentes en el programa.
- n_2, definido como en número de operandos diferentes en el programa.
- N_1, definido como el número total de operadores en el programa.
- N_2, definido como el número total de operandos en el programa.

A partir de estas cuatro métricas básicas, se obtienen otras tres:

- $n = n_1 + n_2$, llamada el vocabulario del programa.
- $N = N_1 + N_2$, llamada la longitud del programa.
- $V = N \log_2 n$, llamada el volumen del programa.

N es una medida de tamaño similar a los conteos más convencionales de sentencias ejecutables. Sin embargo, Halstead [HALS77] prefirió V como una métrica de tamaño, pues se supone que V mide el número de bits necesarios para codificar el programa en una notación con un identificador diferente para cada operador y operando.

Las métricas científicas de software pretenden caracterizar una serie de atributos del programa:

- L, llamado el nivel de abstracción, definido como la inversa de la dificultad experimentada durante el proceso de codificación.

$$L = \left(\frac{2}{n_1}\right)\left(\frac{n_2}{N_2}\right)$$

- I, llamado el contenido de inteligencia del programa, definido como una medida independiente de la implementación de las funcionalidades de un programa.

$$I = L \cdot V$$

- λ, llamado el nivel de lenguaje, definido como una medida independiente del programa del lenguaje de implementación usado.

$$\lambda = L^2 \cdot V$$

- E, llamado el esfuerzo mental, definido como la medida del número de 'discriminaciones mentales elementales' necesarias para codificar un programa.

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

- T, el tiempo en segundos para codificar un programa.

$$T = \frac{E}{18}$$

- *B*, el número de errores que el programa tiene cuando se entrega no sólo al usuario final, sino también entre las distintas fases de desarrollo.

$$B = \frac{E^{2/3}}{3000}$$

Hubo un gran número de estudios de evaluación de estas métricas realizados por Halstead [HALS77] y Fitzsimmons y Love [FITZ78]. Desafortunadamente, la mayoría de esos trabajos confunden correlación (que es la existencia de una relación) con regresión (que indica la naturaleza de una relación). Además, en muchos casos, no hubo intención de probar formalmente ninguna de las hipótesis en consideración usando métodos estadísticos apropiados.

Una reevaluación completa de los estudios de evaluación de estas métricas ha sido realizado por Harmer y Frewin [HAR82]. Su conclusión fue que el soporte experimental no lo era tal.

Otros desarrolladores han criticado los argumentos filosóficos usados para obtener la ecuación para *T* (Coulter, [COUL83] y sugieren que estas métricas no ofrecen más información sobre las características de un programa que la que ofrezcan las simples métricas de tamaño tales como el conteo de instrucciones (Kitchenham [KITC81] y Gremilion [GREM84]).

3.2.7.2. MÉTRICAS BASADAS EN DOCUMENTOS

Los documentos de software escritos en lenguaje natural pueden ser evaluados desde el punto de vista de su legibilidad de la misma manera que cualquier otro documento escrito en lenguaje natural. Uno de los más simples índices de legibilidad es el Índice Fog de Gunning [GUNN62]. Esta métrica evalúa la legibilidad de un texto escrito en inglés en términos de la longitud de las frases y del número de palabras 'duras'.

La medida se calcula sobre la base de un texto ejemplo de unas 100 líneas para cuatro páginas. Las características medidas son:

- *sen*.- número de frases.
- *wrd*.- número de palabras.
- *syl*.- número de sílabas.

El Índice Fog (*FI*) se calcula del siguiente modo:

$$FI = 0{,}4\left(\frac{wrd}{sen} + \frac{hrd}{wrd} \cdot 100\right)$$

donde *hrd* es el número de palabras de tres o más sílabas sin contar: 1) las palabras que están en mayúscula, 2) combinaciones de palabras cortas simples (como *bookkeeper*) y 3) verbos que se convierten en palabras de tres sílabas al añadirles los sufijos '-es' y '-ed'.

El índice Fog se interpreta de acuerdo a la siguiente escala:

Desde	*Hasta*	*Resultado*
-	≤ 5	Fácil
> 5	≤ 8	Estándar
> 8	≤ 11	algo difícil
>11	< 17	difícil
≥ 17	–	muy difícil

Tabla 3-4. Tabla del Índice Fog.

Esta métrica puede ser muy importante para evaluar la legibilidad y comprensibilidad de un documento, sobre todo cuando se está trabajando en grandes proyectos con muchas personas implicadas y donde muchas de ellas tienen que dejar por escrito los resultados de su trabajo.

3.2.7.3. MÉTRICAS DEL DISEÑO

En un intento de extender el ámbito de aplicación de las métricas a todas las fases del ciclo de vida de la aplicación, algunos desarrolladores han desarrollado métricas de la fase de diseño. La mayoría de tales métricas se basan en las relaciones estructurales observadas entre los módulos. La más conocida de tales métricas es la Métrica de Flujo de Información (MFI) de Henry y Kafura [HENR81].

Estas métricas se basan en el conteo de las interconexiones que un módulo tiene con otros módulos dentro del sistema, en el caso de las MFI en el flujo de información entre los módulos, que son no sólo los que se pasan por cabecera en la llamada (que Henry y Kafura llaman flujo local directo de información), sino también flujos de información pasada en los valores de retorno (que ellos llaman flujo local indirecto de información) y flujos de información que se acceden de forma global (que ellos llaman flujo global de información).

Henry y Kafura describieron una métrica compleja para los procedimientos, basada en los flujos de información. Ellos también consideran la complejidad de un módulo (que ellos definen en función de sus estructuras de datos) como compuesto por estructuras de datos y procedimientos que actualizan dichas estructuras o bien obtienen información de ellas.

El *fan-in* del flujo de información de un procedimiento es un conteo del número de flujos de datos locales que entran en el procedimiento, más el número de estructuras de datos desde los cuales el procedimiento recupera información.

El *fan-out* del flujo de información de un procedimiento es un conteo del número de flujos de datos locales que salen del procedimiento más el número de estructuras de datos que el procedimiento actualiza.

Los flujos locales de datos existen si se cumplen algunas de las siguientes condiciones:

- Un procedimiento llama a otro procedimiento.
- Un procedimiento llama a otro y hace uso del valor de retorno.
- Un procedimiento llama a otros dos *A* y *B* y la salida de *A* se pasa como entrada a *B*.

Henry y Kafura usan la siguiente fórmula para indicar la complejidad de un procedimiento:

$$longitud \times [(fan\text{-}in \times fan\text{-}out)^2]$$

donde la longitud está medida con cualquier métrica de tamaño (p. e. SLOC, Halstead, Número Ciclomático de McCabe, etc.).

Henry y Kafura sugieren que sus métricas son indicadas para localizar puntos conflictivos en los desarrollos, que puede que no se haya logrado en ellos un nivel de abstracción suficiente. Su método fue evaluado por ellos mismos frente a proyectos en UNIX en 1984.

Estas métricas tienen algunos problemas:

- Las métricas de la complejidad de procedimientos son difíciles de interpretar excepto en términos de sus métricas componentes. Este problema es también compartido por las métricas de fan-in y fan-out.
- La métrica de complejidad de un procedimiento da un valor de cero para procedimientos no triviales si no tienen fan-in ni fan-out.

3.3. PRINCIPALES MÉTRICAS ORIENTADAS A OBJETOS

3.3.1. SISTEMAS ORIENTADOS A OBJETOS

3.3.1.1. INTRODUCCIÓN

A partir de 1990, principalmente, se comienza a imponer una nueva visión en los sistemas de software: La orientación a objeto. Este es un nuevo enfoque en el modo de entender la producción de software que pretende, ante todo, la reutilización del software, mayor productividad, mejora en la gestión del proyecto y mayor calidad en las aplicaciones entregadas al usuario.

Para comenzar, el enfoque mediante objetos permite al ingeniero de software crear modelos más parecidos al mundo real, lo que a su vez debe permitir que el cliente comprenda mejor los diseños informáticos y sea más capaz de validarlos. Esto, a su vez, permite que los diseños informáticos estén más cerca de las especificaciones iniciales del cliente.

Los desarrollos orientados a objetos se basan en una serie de primitivas fundamentales:

- *Las clases*.- Son las abstracciones de los tipos de objetos que realmente existen y tienen entidad. (p. e. la clase humanos)

- *Los objetos*.- Son las representaciones que, en el modelo, toman entidades concretas del mundo real. (p. e. los objetos Manuel o Pedro)

- *Los atributos*.- Definen las características que tiene una clase y el valor que dichas características toman en los objetos concretos. (p. e. estatura o edad)

- *Las relaciones*.- Son las representaciones de la jerarquía y de las interfaces que gobiernan el sistema real que se esté modelando. (p. e. padre o amigo)

Para realizar adecuadamente el modelo se necesitan una serie de herramientas que permitan definir el tipo de relaciones y que permitan plasmar adecuadamente en un lenguaje de programación el modelo que se ha creado, por tanto estas herramientas deben existir tanto en el análisis como en el diseño y en la codificación. Entre estas herramientas, las más importantes son:

- *La Encapsulación*.- Permite que los atributos de una clase o de un objeto le sean propios y no estén compartidos con otros. En las fases de diseño, esto facilita la creación del modelo. En las fases de programación permite crear código de forma transparente al resto de la aplicación (exceptuando las interfaces).

- *La Herencia*.- Permite modelar el hecho de que en la vida real muchas entidades se derivan de otras de orden superior. En las fases de diseño facilita la creación de esas jerarquías del mundo real que el modelo debe plasmar. En las fases de programación permite reutilizar mucho código, poniendo en las clases altas de la jerarquía la mayor parte del código común a una serie de clases.

- *El Polimorfismo*.- Da limpieza al modelo que se esté creando, permitiendo que una misma operación se pueda ejecutar sobre diferentes clases sin que ello suponga ambigüedad alguna. Esta limpieza se transmite a las fases de programación permitiendo que la misma función se pueda aplicar a diferentes objetos o sobre el mismo realizando operaciones diferentes.

A medida que se desarrolla la orientación a objeto aparecen nuevas herramientas (Tipos abstractos de datos, gestión de excepciones, etc.) que se van integrando en las diferentes fases del ciclo de vida donde son necesarias con el fin de potenciar este tipo de modelado.

Aunque no todos los enfoques orientados a objetos utilizan el mismo paradigma de desarrollo, todo ellos coinciden, de algún modo en fases de análisis y diseño y fases de codificación, pruebas, documentación, etc.

En lo que sigue del capítulo se han separado las fases de análisis y diseño y las de codificación por ser estas dos las que más desarrolladas están. En el último apartado sobre gestión de proyectos se comentan aspectos de la documentación y métricas orientadas a objetos.

3.3.1.2. ANÁLISIS Y DISEÑO ORIENTADO A OBJETOS

Las dos primeras fases de cualquier desarrollo son el análisis y el diseño. Durante la fase de análisis, el ingeniero de software debe captar y plasmar los requerimientos que ha hecho el cliente, con las restricciones impuestas por el cliente. El resultado final debe estar orientado hacia el problema y el objetivo debería ser que el propio cliente verificase el análisis y lo aprobara. Durante la fase de diseño, todos los requerimientos deben ser adaptados al hardware en concreto y al lenguaje de programación en que serán implementados, teniendo en cuenta las restricciones propias de la ingeniería informática (recursos, sistema operativo, etc. ..).

En cualquier caso, tanto en el análisis como en el diseño, las metodologías proponen la creación de modelos, esto es, distintas visiones del problema desde distintos puntos de vista. Al mismo tiempo se crean los diccionarios de datos correspondientes y la documentación que será el producto final de estas fases.

La principal metodología de análisis y diseño (la que más popularidad ha alcanzado en la comunidad de desarrolladores) es el Método Unificado (*UML*).

3.3.1.3. LENGUAJES ORIENTADOS A OBJETOS

Los lenguajes de programación orientados a objetos son las herramientas del programador para plasmar en código las especificaciones y diseños realizadas por el ingeniero de software. Estos lenguajes deben soportar una serie de primitivas (encapsulación, herencia, polimorfismo, etc.) que soportan las metodologías orientadas a objetos.

Con la eclosión de la tecnología orientada a objetos aparecieron también estos lenguajes, se calcula que en la actualidad hay más de cien lenguajes orientados a objeto de alto nivel e incluso hay ensambladores orientados a objetos (o al menos se venden como tales).

El origen de estos lenguajes es muy variado, unos son diseños específicos para los sistemas orientados a objetos y otros por el contrario son adaptaciones de lenguajes existentes procedimentales puros que van modificando su sintaxis para adaptarse a la nueva tecnología.

Una ventaja de la aparición de tantos lenguajes es que esto permitió a cada desarrollador proponer nuevas teorías sobre lo que debería ser un lenguaje OO de modo que los lenguajes de mayor éxito comercial se han tenido que ir adaptando de forma dinámica a estas nuevas tendencias. Tal es el caso de *C++* que en poco más de diez

años ha pasado por cuatro revisiones importantes (sin contar el original *C with classes*). En la Tabla 3-5 se puede ver una lista de los lenguajes OO más comunes.

LENGUAJE	*AÑO*	*DESARROLLADORES*
JAVA		Sun
ADA	1978 (1995)	Jean Ichbiah
AGORA		Programming Technology Lab de la Brussel Free University
BETA		Aarhus University (Dinamarca)
CLOS	1988	Daniel Bobrow (Pte. Comité CLOS)
C++	1980	Bjarne Stroustrup
DYLAN		
EIFFEL	1985	Bertrand Meyer
JAVA	1995	Sun
MODULA-3		
OBJECT PASCAL	1986	Appel Computer y Niklaus Wirth
OBJECTIVE-C		
OO-COBOL		
SMALLTALK	1972	Xeros Palo Alto Research Center

Tabla 3-5. Lenguajes de programación orientados a objetos más conocidos. Entre paréntesis está el año de su última revisión.

Además de los lenguajes de programación orientados a objetos, también se han desarrollado sistemas de bases de datos orientadas a objetos.

A la par que lenguajes también han aparecido algunas Bases de Datos Orientadas a Objetos (BDOO), que soportan objetos persistentes y que realizan las consultas en lenguajes también orientados a objetos.

Una de estas bases es *Thor* [@ZON96]. Esta BDOO está ampliamente distribuida y soporta almacenamiento persistente para los objetos. Hace el desarrollo de aplicaciones más fácil mediante una plataforma de alto nivel para el desarrollo de programas. Permite que el programador use diferentes lenguajes de programación estándar.

Los objetos en *Thor* se especifican e implementan en *Theta*, un lenguaje propio. A continuación se puede ver una lista no exhaustiva de estos productos:

- Sistemas de Bases de Datos Orientadas a Objetos:

COMPAÑÍA	PRODUCTO
A.D. Software	OOFile
Answer Software	MyBase
Basesoft Open Systems AB	EasyDB
GemStone (formerly Servio)	GemStone
Ibex computing (formerly Itasca Systems)	ITASCA
Mainstay	Phyla
NeoLogic Systems	NeoAccess, NeoShare
O2 Technology	O2
Object Design	ObjectStore
Objectivity	Objectivity
ONTOS (formerly Ontologic)	ONTOS VIA, DB, OIS
Persistent Data Systems	IDB Object Database
Poet Software	Poet

Versant	Object Technology
Xcc Software Technology Transfer	OBST+

- Sistemas de Bases de Datos Orientadas a Objetos Relacionales:

COMPAÑÍA	PRODUCTO
Oracle	Oracle
ADB Inc.	MATISSE
Cincom Systems	Total ORDB
Illustra Information Technologies	Illustra
Omniscience Object Technology	Omniscience ORDBMS
Persistence Software	Persistence
Raima	Raima Object Manager
Subtle Software	Subtleware
UniSQL	UniSQL

3.3.1.4. GESTIÓN DE PROYECTOS ORIENTADOS A OBJETOS

A la vista de los sistemas presentados en este capítulo (metodologías, herramientas CASE, lenguajes y bases de datos), se comprende que las métricas no han sido integradas en ellos de una forma natural y que la mayoría no están apoyados en ellas en ningún sentido. De este modo es difícil hacer una evaluación o una comparativa entre los distintos productos y es imposible estimar y controlar adecuadamente los proyectos.

Incluso en los entornos donde se utiliza algún tipo de métrica, es difícil la comparativa ya que los sistemas Orientados a Objetos se utilizan en cada empresa después de una adaptación particular a su equipo de trabajo y al entorno, haciendo muy difícil una evaluación de los proyectos de una manera estandarizada.

Cada metodología (desprovista de una métrica adecuada y adaptada) no permite ser comparada y se adaptará mejor a un proyecto que a otro, sería necesario conocerlas todas (o una buena parte de ellas) para tener una visión más acertada del problema. Sin embargo, la OO funciona en la vida real, y todo ello a pesar de que «*…no existen todavía criterios inamovibles de evaluación metodológica en OOA/OOD*» (Ricardo Devis).

Para realizar una buena gestión de proyectos orientados a objetos, es necesaria, pues, la ayuda de un sistema de apoyo integral del proyecto que guíe al ingeniero de software a través de todas las fases del ciclo de vida. Este sistema debe estar sostenido por métricas formales que puedan controlar de forma automática todo el complicado entramado de variables que se estime que influyen sobre el proyecto.

3.3.2. MÉTRICAS ORIENTADAS A OBJETOS

3.3.2.1. INTRODUCCIÓN

Como se había comentado en el capítulo anterior, cada día es más importante definir y poder contar tanto con Metodologías como con Sistemas de Métricas adecuados.

Hasta el momento actual, sin embargo, las metodologías diseñadas no han sido acompañadas de sistemas paralelos de métrica, por lo que todas éstas han sido añadidos posteriores realizados por personas distintas de los que desarrollaron aquellas.

Es importante, en cualquier caso, que se disponga de una métrica adecuada si queremos que la informática sea cada vez más una ingeniería y menos un arte. Por tanto es necesario medir y de ese modo poder comparar y estimar comportamientos y, por qué no, costes. Además la medida nos permite establecer un orden en las cosas y por tanto nuestro conocimiento aumenta.

En general la métrica nos permite: evaluar y comparar las diferentes metodologías, seleccionar bibliotecas y comparar el código perteneciente a las mismas, estimar el coste y la planificación de los proyectos, evaluar el impacto en productividad de las nuevas herramientas orientadas a objetos, elegir distintas posibilidades de diseño, etc.

En el estado actual de desarrollo, con la llegada de la Orientación a Objetos (OO) que comprende todas las etapas del desarrollo software, parece indicado que las metodologías soporten también la OO.

Sin embargo, existen dos enfoques a la hora de desarrollar métricas OO, el que aboga por mantener y adaptar las métricas clásicas (McCabe, Watson,...) y el que aboga por desarrollar nuevas métricas especialmente diseñadas para la OO (Chidamber, Kemerer,...).

También hay quien piensa en modelos híbridos o que aprovechen lo mejor de cada una de ellas, en palabras de Mario Piattini: «*personalmente creemos que existen atributos como pueden ser la fiabilidad o la mantenibilidad cuya medida prácticamente no ha cambiado con respecto a las medidas tradicionales; mientras que otros que están relacionados con los aspectos de complejidad en el que juegan un papel importante conceptos como el de encapsulamiento, la generalización, la agregación o el polimorfismo, sí requieren nuevas métricas. También se pueden redefinir de una manera Orientada al Objeto métricas clásicas como, por ejemplo, el acoplamiento y la cohesión, que pueden seguir resultando útiles, como se demuestra en LIBERHERR et al. [LIBE89] donde se definen reglas de estilo correcto utilizando estos conceptos*».

3.3.2.2. CALIDAD

Así pues, la consecución de niveles de calidad adecuados es un factor fundamental del desarrollo de software, pero es imprescindible establecer previamente dichos estándares para que permitan posteriormente definir los sistemas de métricas a utilizar.

Pero el establecimiento de dichos estándares no es una tarea fácil, ya que en ella intervienen los puntos de vista de los usuarios, los diseñadores y otros elementos humanos integrados en el proceso de elaboración de software.

Para que el sistema de métricas obtenga resultados aceptables debe seguir estándares de calidad como Goal-Question-Metric (GQM) [BASI88] o Quality Management System (QMS) [KITC86-1], que ayuden a conseguir modelos de calidad estableciendo estándares sobre elementos importantes como la facilidad de uso, la facilidad de mantenimiento o la reutilización.

Algunas métricas clásicas pueden ser de utilidad en el paradigma de OO, pero en muchos casos, es necesario definir nuevas métricas, como por ejemplo para medir conceptos que están ligados exclusivamente a la OO como la herencia, el polimorfismo, la encapsulación, etc.

En los próximos apartados se presentarán algunas de las métricas OO más conocidas.

3.3.2.3. TAYLOR

En Taylor [TAY93] se proponen como básicas las siguientes métricas para un sistema Orientado a Objeto:

- nº de servicios
- nº de objetos
- nº total de mensajes
- nº de objetos que reciben mensajes
- nº total de parámetros
- ratio de servicios públicos/privados

Además deben calcularse una serie de medidas estadísticas como media y varianza de algunos de los datos anteriores, ya que la opinión del propio autor es que los totales solos no dicen demasiado.

Para cada clase propone una serie de métricas (nº de servicios, de objetos componentes, de mensajes, de objetos receptores, de parámetros y ratio de servicios públicos/privados), y también en este caso es necesario el cálculo de la media y la varianza. Con esta información Taylor asegura que se pueden predecir el número de fallos, el número de líneas de código o la memoria necesaria.

La principal carencia de esta métrica radica en que Taylor sólo hace una declaración de necesidades, pero no explica cómo deben manejarse para obtener resultados, quedando éstos para la interpretación de los expertos. No obstante es importante el hecho de que no se centra sólo en los totales (como hacen el resto de las métricas) sino en otras medidas estadísticas.

3.3.2.4. BARNES Y SWIM

Los autores [BAR93] proponen una extensión del paradigma de la orientación a objeto y del entorno de programación que permita *heredar* métricas de software. Distinguen tres categorías de métricas diferentes: *basadas en objetivos*, *basadas en sintaxis* y *basadas en ejecución*.

Para las métricas basadas en objetivos proponen la especificación semántica y el control de excepciones siguiendo un modelo inspirado en Eiffel [MEY00] y en la programación por contrato.

Con este modelo se elaboran estadísticas encaminadas principalmente a medir la tolerancia a fallos del sistema, la capacidad de recuperación y otras muy relacionadas con

la fase de explotación y mantenimiento. En esta fase no se proponen demasiadas solucionas al tipo o calidad de las métricas y en el propio artículo, a la hora de presentar el prototipo de QOOL (ActQOOL) se reconoce que soporta este tipo de métricas de un modo muy superficial.

Para las métricas basadas en sintaxis si que desarrollan un modelo amplio, basadas siempre en métricas clásicas (McCabe, LOC, Número de mensajes, etc...).

Todas las métricas propuestas están ampliamente soportadas en ActQOOL y se especifica el método de heredarlas entre diferentes niveles de la jerarquía usando para ello redefinición de métodos, polimorfismo, herencia, etc.)

Por último las métricas basadas en ejecución se definen como imprescindibles en la fase de explotación y mantenimiento y se las dota de elementos que permitan en el futuro controlar los fallos de los asertos y enviar esta información al desarrollador para conseguir una atención al cliente personalizada y efectiva.

Éstas no están soportadas por ActQOOL en absoluto y sólo se las cita desde el punto de vista puramente experimental y académico, pero en palabras del propio Bradley Swim son un campo de experimentación muy interesante y no le consta que nadie haya seguido investigando en este terreno, a excepción de los profilers.

Barnes y Swim, en su artículo (1993), proponen un lenguaje Quality Object Oriented Language (QOOL) que permite un proceso de ingeniería de software dirigido por calidad. Mediante este lenguaje, además de los servicios de la clase se pueden definir otros para calcular las tres categorías de métricas anteriormente definidas.

Aunque las métricas que proponen sean las clásicas, lo más destacable es que proponen la integración de las métricas en los propios entornos de desarrollo.

3.3.2.5. CHIDAMBER Y KEMERER (MOOSE)

Los autores [CHI94] adoptan tres criterios a la hora de definir las métricas: capacidad de satisfacer propiedades analíticas, aspecto intuitivo a los profesionales y gestores y facilidad para su recogida automática. Proponen las siguientes métricas:

3.3.2.5.1. SERVICIOS PONDERADOS POR CLASE (WMC)

Dada una clase C con servicios M1, M2, ... , Mn de complejidad c1, c2, ... , cn respectivamente, esta métrica se calcula como

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

WMC es una medida de complejidad de la clase, pero no tiene en cuenta los atributos, pues los autores aseguran que introduce ruido sin aportar nada significativo.

3.3.2.5.2. PROFUNDIDAD DEL ÁRBOL DE HERENCIA (DIT)

Definida como la longitud desde el último nodo hasta la raíz del árbol.

3.3.2.5.3. NÚMERO DE HIJOS (NOC)

Es el número de subclases inmediatamente subordinadas de una clase en la jerarquía de clases.

3.3.2.5.4. ACOPLAMIENTO ENTRE CLASES DE OBJETOS

Es el número de clases con las que una clase concreta está acoplada. Una clase está acoplada si lo están sus objetos y un objeto está acoplado con otro si uno de ellos actúa sobre el otro.

3.3.2.5.5. RESPUESTA PARA UNA CLASE (RFC)

Se define como

$$RFC = | RS |$$

Donde RS es el conjunto de respuestas de la clase, esto es, el conjunto de servicios de la clase más el conjunto de servicios invocados por los de la clase.

3.3.2.5.6. FALTA DE COHESIÓN EN LOS SERVICIOS (LCOM)

Dada una clase C con n servicios M_1, M_2, ... , M_n; e $\{I_i\}$ el conjunto de atributos empleados por el servicio M_i, si hacemos

$$P = \{(I_i, I_j) \mid I_i \cap I_j = \varnothing\}\ y$$
$$Q = \{(I_i, I_j) \mid I_i \cap I_j <> \varnothing\}$$

(En caso de que todos los conjuntos $\{I_1\}$, ... , $\{I_n\}$ sean vacíos se toma P=0). Se define

$$LCOM = | p | - | q |, si\ P > Q$$
$$= 0\ en\ caso\ contrario$$

en definitiva es el número de pares de servicios cuya similitud es cero menos los que tienen similitud distinta de cero. Cuanto mayor es el número de servicios similares, más cohesionada resulta la clase.

3.3.2.6. MOOD. (METRICS FOR OBJECT ORIENTED DESIGN)

MOOD (Metrics for Object Oriented Design), definidas por Abreu y Melo, es un conjunto de métricas que opera a nivel de sistema [BRI96]. Mide elementos estructurales básicos de la OO, como encapsulación (MHF y AHF), herencia (MIF y AIF), polimorfismo (PF) y paso de mensajes (COF).

3.3.2.6.1. PROPORCIÓN DE MÉTODOS OCULTOS (MHF)

MHF (Method Hiding Factor), se define como la proporción de invisibilidad de los métodos de todas las clases frente al número total de métodos definidos en el sistema.

Por tanto MHF es la proporción entre los métodos definidos como protegidos o privados y el número total de métodos.

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

Donde:

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

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

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

Esto es, para todas las clases un método cuenta como 0 si puede ser usado por otra clase y 1 en caso contrario. *TC* es el número total de clases del sistema.

Esta es una medida de encapsulación.

Para calcular esta métrica no se consideran los métodos heredados.

3.3.2.6.2. PROPORCIÓN DE ATRIBUTOS OCULTOS (AHF)

AHF es una medida de la proporción de atributos ocultos de una clase. Es la proporción entre los atributos definidos como protegidos o privados y el número total de atributos.

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

Donde:

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

$$es\_visible(A_{mi}, C_j) = \begin{cases} 1 \Leftrightarrow j \neq i \wedge C_j \text{ puede acceder a } A_{mi} \\ 0 \qquad \text{en caso contrario} \end{cases}$$

TC es el número total de clases del sistema.

Idealmente el valor de esta métrica debería ser siempre de 100%, ya que ningún atributo debería ser público para no romper la encapsulación.

3.3.2.6.3. PROPORCIÓN DE MÉTODOS HEREDADOS (MIF)

MIF mide la proporción de métodos heredados frente al número de métodos totales.

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

Donde

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

siendo $M_a(Ci)$ el número de métodos que pueden ser invocados en relación a una clase, $M_d(C_i)$ es el número de métodos declarados en una clase, $M_i(C_i)$ es el número de métodos heredados (y no redefinidos en C_i).

Mediante esta métrica se obtiene información de la reutilización y de la dificultad de mantenimiento.

3.3.2.6.4. PROPORCIÓN DE ATRIBUTOS HEREDADOS (AIF)

AIF mide la proporción de atributos heredados frente al número de atributos totales.

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

Donde

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

siendo $A_a(Ci)$ el número de atributos que pueden ser invocados en relación a una clase, $A_d(C_i)$ es el número de atributos declarados en una clase, $A_i(C_i)$ es el número de atributos heredados (y no redefinidos en C_i).

Mediante esta métrica (al igual que la anterior MIF) se obtiene información de la reutilización y de la dificultad de mantenimiento.

3.3.2.6.5. PROPORCIÓN DE POLIMORFISMO (PF)

Mide la relación entre el número total de métodos redefinidos para una clase (número máximo de situaciones polimórficas distintas) frente al número total de métodos redefinidos realmente (número total de situaciones polimórficas).

$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=0}^{TC} \left[M_n(C_i) \times DC(C_i) \right]}$$

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 número de métodos redefinidos, $DC(C_i)$ el número de descendientes de C_i, $M_d(C_i)$ el número total de métodos definidos y TC el número total de clases.

3.3.2.6.6. PROPORCIÓN DE ACOPLAMIENTO (CF)

CF se define como la proporción entre el número total de acoplamientos en el sistema y el número total de éstos no imputables a la herencia.

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

Donde

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

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

3.3.2.7. LORENZ Y KIDD

En su libro "Object Oriented Software Metrics" [LOR94], Lorenz y Kidd presenta una suite de métricas aplicables a dos aspectos del desarrollo de software: gestión del proyecto y diseño orientado a objetos.

Cada una de las categorías principales es dividida, a su vez en subcategorías de métricas aplicables a partes concretas del desarrollo de software y por último en cada categoría se presentan una serie de métricas concretas, así por ejemplo:

- Métricas de Proyecto
  - Tamaño de la aplicación
    - Número de scripts de escenarios
    - Número de clases clave
    - Número de clases de soporte
    - Proporción de clases de soporte frente al número de clases clave
    - Número de subsistemas
  - Tamaño del equipo
    - Número de personas-día por clase
    - Número de clases por desarrollador
  - Plan de desarrollo

        - Número de iteraciones principales
        - Número de contratos completados
- Métricas de Diseño
    - Tamaño de métodos
        - Número de mensajes enviados
        - Significado
        - Número de instrucciones
        - Número de líneas de código
        - Tamaño medio de los métodos
    - Métricas internas de métodos
        - Complejidad de métodos
        - Cadenas de mensaje enviadas
    - Tamaño de las clases
        - Número de métodos públicos en una clase
        - Número de métodos en una clase
        - Número medio de métodos por clase
        - Número de atributos en una clase
        - Número medio de atributos por clase
        - Número de métodos de clase en una clase
        - Número de atributos de clase en una clase
    - Herencia de las clases
        - Nivel de anidación de la herencia de clases
        - Número de clases abstractas
        - Uso de herencia múltiple
    - Herencia de métodos
        - Número de métodos sobrescritos por una subclase
        - Número de métodos heredados por una clase
        - Número de métodos añadidos por una subclase
        - Índice de especialización
    - Métricas internas de clases
        - Cohesión de clases
        - Uso global
        - Media del número de parámetros por método
        - Uso de funciones “friend”

- Porcentaje de código funcional
- Media del número de líneas de comentario por método
- Número de problemas reportados por clase o contrato.

o Métricas externas de clases

- Acoplamiento de clases
- Número de veces que se reutiliza una clase
- Número de clases y métodos rechazados

Según los propios autores, las métricas son el resultado de una recolección de métricas (una existentes y otras modificadas) que fueron recogiendo y probando en proyectos reales desarrollados en Smalltalk y C++ en IBM.

3.3.2.8. CANT

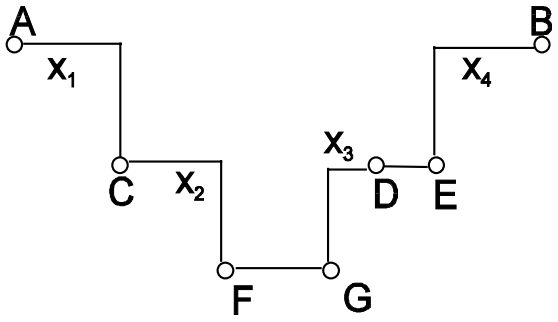


Ilustración 3-1. Modelo de perfil.

CANT et al. [CANT94] proponen un enfoque teórico basado en el análisis de modelos cognitivos para la comprensión del código por parte del programador. Se parte del reconocimiento de que los programadores utilizan dos técnicas diferentes *troceo* y *rastreo* de forma concurrente y sinérgica en la resolución de problemas. Esto es, cuando se está buscando un error o revisando un código, se hace por trozos pequeños de manera que durante la búsqueda hay que suspender de vez en cuando la lectura para buscar (rastrear) dependencias.

Se pueden representar gráficamente (como se ve en la Ilustración 3-1) los efectos de estas técnicas. En ella, las líneas horizontales representan el troceo y las verticales el rastreo. En el caso de la figura hay un trozo que va de A a B que necesita dos rastreos C-D y D-E de nivel inferior y al mismo tiempo el trozo C-D necesita un rastreo de nivel más inferior F-G. La complejidad de los trozos viene dada por su longitud (A-B tiene de complejidad x1+x4).

Las líneas verticales representan el trabajo necesario para llevar a cabo el rastreo. De este modo la dificultad en el programa para llevar a cabo una consulta centrada en el trozo i es

$$C_i = R_i + \sum_{j \in N} C_j + \sum_{j \in N} T_j$$

Siendo N el conjunto de todos los trozos de los que el trozo i depende directamente para una tarea y Ti es la dificultad de rastrear una dependencia concreta. La dificultad R es función de otras dificultades (el tamaño, la estructura de control, las expresiones booleanas, el formato visual, etc.). Por último la dificultad de rastreo es a su vez, también, función de varios aspectos (el efecto de ubicación sobre el rastreo, la dependencia espacial, etc.).

3.3.2.9. EL PROYECTO AMI

Como prueba de la importancia que las métricas están adquiriendo en la Ingeniería del Software (I.S.), se constata la atención cada vez más importante que tanto las empresas como los gobiernos están dando a este tema.

En este sentido la C.E., dentro del programa ESPRIT, creó el proyecto AMI [@AMI]. El objeto del proyecto es el establecimiento y difusión de un método práctico y validado para la implantación de un programa de métricas de software en las empresas.

AMI es un método validado que consta de una serie de fases de implementación (definición de objetivos, definición de métricas, plan de métricas y explotación de datos) y que no están destinadas a una métrica concreta, sino que todas tienen cabida, por supuesto también las orientadas a objetos.

El manual de AMI (ISBN 0 9522262 0 0), fue publicado por el consorcio AMI en 1992. Es una guía de gestión de software destinada a jefes de proyecto, responsables de calidad e ingenieros de software. En su contenido destacan las métricas de software de las que se hace una guía de buenas prácticas.

Desde su lanzamiento en marzo de 1992 se han distribuido 3000 copias en treinta países.

En cualquier caso el panorama actual de las métricas de software sigue siendo bastante desolador, esto puede ser debido, en parte, a la poca experiencia que sobre el tema tienen las empresas de software.

3.3.2.10. OTRAS MÉTRICAS

Con la explosión producida por la entrada de la orientación a objetos se ha producido también la explosión de métricas orientadas a objeto que pretenden ser la solución definitiva.

Entre las métricas que han visto la luz en los últimos años podemos contar, además de las ya vistas, con:

- Li y Henry [LI93], es una extensión de MOOSE con otras métricas de tamaño y diseño y un sistema de predicción para reusabilidad y mantenibilidad.

- Bieman *et al.* [BIE91] sistemas de métricas para cohesión, acoplamiento, reutilización y herencia de los sistemas orientados a objeto.

- MOSES de Henderson-Sellers y Edwards [HEN94].
- SOMA de Graham [GRA95].
- Laranjeira [LAR90], estimaciones de tamaño de software para sistemas orientados a objetos.
- Jenson y Bartley [JEN91], modelo de estimaciones paramétricas de esfuerzo de programación en un entorno OO.
- Rains [RAI91].
- Linda Rising [RIS93], métricas de ocultación de información.
- Martin [MAR94], medidas de la calidad del diseño orientado a objetos.
- Chen y Lu [CHEN93], medida de la complejidad de las clases y los objetos a partir de métricas del diseño orientado a objetos.
- Coppick y Cheatham [COPP92], complejidad de un objeto usando métricas tradicionales.
- Tegarden y Sheetz [SHEE93].
- Hay también toda una serie de métricas basadas en la teoría clásica de los puntos-función: *Object Function Point Analysis* [ZHAO95], *Object Point Analysis* [BANK91], [THOM94], *MK II function points for OO* [HOR94] y *3D function points*.
- Abbot *et al.* [ABB94], con una métrica para medida de la complejidad de los desarrollos orientados a objetos, más concretamente de la interacción entre objetos.
- Chung [CHU92], métrica basada en la herencia para el análisis de la complejidad en diseños orientados a objetos.

Ninguna de las métricas existentes son métricas formales de acuerdo a la teoría de la medida y sólo unas pocas se ajustan la teoría de la medición. Además se puede comprobar que casi todas se centran en alguna(s) etapa(s) del ciclo de vida y no en una visión integral de todo el desarrollo. También se puede comprobar que ninguna está pensada para tiempo de ejecución excepto la de Barnes & Swim [BAR93].

3.4. JMETRIC

JMetric [@JMET] es una herramienta realizada en Java que permite la medición sobre software realizado. Mediante JMetric se pueden recoger las métricas tradicionales de software en tiempo de diseño a nivel de diseño, proyecto, clase, método, etc.

Se basa en un parser realizado mediante JavaCC que inspecciona el código fuente y realiza diferentes informes de las métricas líneas de Código, Número de instrucciones, LCOM, y Complejidad Ciclomática.

La herramienta no se usa en tiempo de ejecución de la aplicación, sino en tiempo de compilación y sirve únicamente para medir sobre el código y no para intervenir en la ejecución.

Desde este punto de vista apenas tiene similitud con el sistema presentado en esta tesis. Ni siquiera es un profiler que pueda hacer medidas en tiempo de ejecución.

3.5. ESTUDIOS EN PROYECTOS REALES

Durante el desarrollo de esta tesis se han realizado una serie de estudios sobre proyectos reales en empresas con dos objetivos fundamentales, de un lado detectar los problemas más importantes para poner en marcha sistemas de métricas en desarrollos de software y de otro validar algún tipo de procedimiento para adaptar una suite de métricas a una empresa concreta con desarrollos reales.

El primer problema consistió en la selección de métricas y de criterios para elegir un conjunto de métricas significativas y del que inferir información sobre desarrollos futuros.

3.6. SELECCIÓN DE MÉTRICAS

Un problema difícil es el de seleccionar métricas. Ante la falta de consenso acerca de cuáles son las mejores métricas o las que una normativa obliga a aplicar, cada responsable de proyecto debe elegir las que le parezcan más indicadas.

En general no hay ni siquiera una guía de evaluación o de adecuación de las normas y es el sentido común el que debe imponerse. Algunas reglas generales pueden ser:

- No usar métricas compuestas a no ser que estén adecuadamente calibradas y validadas para un entorno particular.
- Elegir sólo aquellas métricas que sean significativas en términos de ingeniería del software y que tengan, por tanto, clara interpretación.
- Asegurarse de que se eligen métricas para cada fase, tanto para análisis y diseño como para implementación, pruebas y tiempo de ejecución.
- Cuando se eligen métricas en tiempo de ejecución, se debe hacer un estudio de los elementos que se desean medir y controlar y diseñar la métrica adecuada a cada caso.
- Usar siempre métricas estándar, comprobadas y calibradas antes que diseñar soluciones ad-hoc. Dejar estas métricas ad-hoc para los casos en que las métricas estándar son claramente inadecuadas.

- Antes de medir plantear y planificar adecuadamente los objetivos de la medición y usar las métricas que realmente tienen significado.

- Medir es una operación de elevado coste en general, por tanto se debe elegir una colección de métricas lo más reducida posible y se debe estimar el coste de realizar esa medición.

Las métricas usadas en los estudios fueron las métricas de Shyam R. Chidamber y Chris F. Kemerer [CHI94] y con ellas se estudiaron diferentes tipos de proyectos (todos ellos desarrollados con tecnologías orientadas a objetos).

3.7. RESULTADOS DE LOS ESTUDIOS

En los estudios realizados con métricas en proyectos reales, se han detectado, en general, deficiencias de aplicación ya que la mayor parte de los desarrollos parten de diseños insuficientes y por tanto es difícil aplicar métricas que están pensadas para diseños previos a la codificación y con objetivos bien definidos.

Un problema similar es tratado también para las métricas de acoplamiento y cohesión en fases muy iniciales [BALL01].

Por ejemplo, al utilizar frameworks de desarrollo pensados para prototipado rápido, las métricas clásicas OO son de difícil utilización, ya que generalmente estos diseños evolucionan mucho durante la codificación y los resultados obtenidos a priori no son aplicables al final de la implementación [TVF99].

Esto lleva a situaciones en que el resultado más interesante de la aplicación de las métricas se consigue al final del desarrollo del proyecto, pero en este momento muchas de las métricas no son de interés más que con objetivo de utilización estadística en proyectos futuros [TVF98].

Más interesante resulta la utilización de estas métricas cuando las partes del diseño realizado se hace en base a patrones de diseño, en este caso las soluciones implementadas tienen escasas variaciones con respecto al diseño realizado y entonces las métricas sí dan resultados extrapolables a otros proyectos [TVF98-1]. En este caso los resultados son mejores no ya sólo por la utilización de métricas, sino, evidentemente, por la utilización de los patrones de diseño que obligan al equipo a renunciar a algunas de las facilidades del framework en beneficio de un proyecto de mejor comprensión y de unas soluciones más reutilizables y fáciles de mantener.

La situación no es mejor en los entornos de monitorización industriales [SYS99]. En estos casos las aplicaciones suelen ser dependientes de los sistemas estándares de supervisión (SCADA), lo que hace que los requerimientos varíen durante el desarrollo de la supervisión, dando lugar a sistemas difíciles de diseñar y por tanto las métricas sobre esos diseños son difíciles de aplicar con resultados que permanezcan estables durante todo el diseño.

3.7.1. MEDICIONES DE SOFTWARE EN TIEMPO DE EJECUCIÓN

Sin embargo en desarrollos realizados sobre aplicaciones en tecnología web ([SYS02] y [TVF02]) si se ha detectado una necesidad de medir muchos eventos accesorios a lo que es puramente la programación de la lógica de negocio de dichas aplicaciones.

Es muy común (con objetivos diversos) tener que medir el número de accesos concurrentes, el tiempo de duración de la sesión, el porcentaje de sesiones que terminan inesperadamente y otros eventos similares. Todas estas mediciones llenarían el código de puntos de medida y de código adicional al puramente de la lógica. En estos casos las métricas tradicionales no han dado una solución satisfactoria. Se trata normalmente de métricas ad-hoc, de estructura sencilla y con objetivos claros de controlar o monitorizar la aplicación que no deberían estar integradas en el código de la lógica de las clases.

En algunos puntos se pueden tomar decisiones de control o también la activación de alarmas, por ejemplo cuando se detectan sesiones de menos de un tiempo de duración, ya que puede estar el servidor saturado o puede que se trate de intentos de accesos no deseados.

4. CARENCIAS DE LOS ACTUALES SISTEMAS DE MÉTRICAS

4.1. CAMPOS QUE RECOGEN LAS ACTUALES MÉTRICAS

Todos los desarrollos actuales de métricas están centrados, principalmente en el desarrollo de sistemas de métricas predictoras para establecer esfuerzos, calidad, costes y tiempos.

En casi todos los casos están centradas en aspectos de la metrología teórica avalada por resultados empíricos normalmente a través de estudios estadísticos y centradas en la metrología orientada a la caracterización del sujeto.

Todos estos estudios son de suma importancia y son una buena base para la definición de métricas universales que describan los procesos de desarrollos y la bondad del software, pero no estudian una parte importante de la medida y es el tiempo de ejecución.

En este sentido van orientados principalmente los profilers (programas de prueba de aplicaciones) los que tienen los mejores resultados, pero siempre vistos como una herramienta externa y nunca como una parte más del software elaborado.

Es opinión del autor que la estandarización en el uso de las métricas no será un hecho hasta que no se consiga que la medida sea una parte del desarrollo de las aplicaciones y en este sentido va encaminada esta tesis.

4.2. ESTUDIOS SOBRE MÉTRICAS DE SOFTWARE

La mayor parte de los estudios sobre métricas de software van orientados hacia la constatación de métricas concretas, diseño de un conjunto de métricas, aplicación de esas métricas a un conjunto de aplicaciones y elaboración de unas estadísticas con resultados más o menos interesantes.

Sin embargo en muy pocas ocasiones se habla del soporte de las métricas por el propio lenguaje o sistemas que permitan medir en tiempo de ejecución, lo que no quita importancia ni interés a los sistemas tradicionales.

El objetivo de esta tesis es diferente, no se trata de diseñar un conjunto de métricas, sino de diseñar el soporte para cualquier conjunto de métricas que se pueda aplicar en tiempo de ejecución, por tanto se define una metodología, un framework para los lenguajes actuales orientados a objetos y un lenguaje que soporte directamente el sistema de métricas.

4.3. MÉTRICAS EN TIEMPO DE EJECUCIÓN

Cabe hacer aquí un hueco para QOOL de Barnes & Swim [BAR93]. Los objetivos de QOOL de conseguir un lenguaje con integración de métricas es un enfoque realmente

interesante y coincidente en parte con lo que se expondrá en el siguiente capítulo de esta tesis.

Las principales diferencias entre QOOL y DMT residen en que aquellos estaban muy orientados a métricas de calidad, con indicaciones específicas del tipo de métrica, mientras DMT pretende ser generalista y adoptar cualquier tipo de métrica, estandarizada o ad-hoc para una solución concreta, no pretende estar orientada a calidad exclusivamente sino que los objetivos pueden ser muy variados y se basa en una metodología de diseño que en Java será soportada por un framework concreto, por tanto las coincidencias son evidentes, así como también lo son las diferencias, lo que justifica que no se trate en absoluto de la misma solución, más bien DMT incluiría los objetivos de QOOL.

4.4. NECESIDAD DE MÉTRICAS EN TIEMPO DE EJECUCIÓN

Es por tanto importante que las métricas se definan como parte del desarrollo de las aplicaciones lo que ofrece dos enfoques nuevos:

- La integración de la medición como parte del desarrollo y por tanto la necesidad de que los lenguajes de programación y las metodologías de desarrollo contemplen esta posibilidad.

- La modificación del estilo de programación y la aparición de accionadores (o elementos capaces de dirigir a la aplicación) lo que permitirá el diseño de aplicaciones con elementos similares a los tradicionales sistemas de control industrial de procesos.

Cambiando de este modo el enfoque de las métricas se consiguen varios efectos beneficiosos:

- La integración de las métricas dentro de los procesos de desarrollo de software al estar integrados dentro de las metodologías y de los lenguajes de programación.

- Delimitar las fronteras entre las mediciones inherentes al proceso y los algoritmos propios de la programación.

- La posibilidad de demostrar en ejecución las bondades del sistema.

- Aumentar las posibilidades de intervención en el proceso para adaptarlo mejor al entorno a través de los sistemas de monitorización y control.

Un desarrollo desde este punto de vista tendrá una serie de métricas para la fase de pruebas y un subconjunto del anterior para la fase de explotación, pero todas ellas habrán sido definidas de un modo ortogonal y por tanto su integración y su adaptación al entorno será mayor. Además, la separación entre elementos controlados y otros objetos dentro del sistema permite la interacción externa desde los objetos monitores y controladores, permitiendo actuar sobre una aplicación en fase de ejecución y hacer que modifique los elementos controlados.

Todo ello es posible con programación tradicional, pero la mayor aportación del nuevo sistema es el proceso metodológico y la separación clara entre ambos procesos: algoritmo y medición.

5. DMT. METODOLOGÍA PARA EL DISEÑO DE MÉTRICAS EN TIEMPO DE EJECUCIÓN

A lo largo de los últimos años, la tecnología orientada a objetos se ha desarrollado en diferentes segmentos de las ciencias de la computación como un medio para manejar la complejidad inherente a sistemas de muy diversos tipos. El modelo de objetos ha demostrado ser un concepto muy potente y unificador.

(Grady Booch)

5.1. MÉTRICAS DE SOFTWARE OO EN TIEMPO DE EJECUCIÓN

5.1.1. DEFINICIÓN DE MÉTRICA EN TIEMPO DE EJECUCIÓN

Podemos definir una métrica en tiempo de ejecución como el resultado formalmente expresado de la observación del comportamiento de un atributo (simple o complejo) que forma parte de la ejecución o de la evolución de una aplicación.

5.1.2. CONCLUSIONES DE LA DEFINICIÓN

Desde este punto de vista, el sistema que mida debe estar en ejecución al mismo tiempo que el sistema observado y medirá atributos internos del sistema observado.

No es necesario que el sistema observador esté en una máquina concreta y puede que el sistema esté ejecutándose en una máquina y la observación se haga desde otra independiente. A esto puede ayudar la orientación a objetos, abstrayendo el concepto de medición de su implementación.

En concreto y bajo la JVM [@JVM] es posible la implementación como objetos distribuidos o como agentes.

Por transitividad, estos atributos podrían corresponder a medidas del mundo real capturadas por el sistema. Sólo hay que imaginar una aplicación que corra capturando medidas de elementos del mundo real mediante tarjetas de adquisición de datos y que mediante el sistema de métricas y control tome decisiones que después transforma en acciones sobre el mundo real y las vuelve a emitir a través de una tarjeta de salida.

El planteamiento de una métrica de estas características requiere de una serie de premisas para que, tanto la medición como la monitorización se separen de la ejecución normal del sistema observado, tanto en su concepción como en su ejecución.

Desde el punto de vista de la ejecución, los frameworks pueden ser la solución. Desde el punto de vista de la separación conceptual, sólo nuevos lenguajes de programa-

ción expresamente diseñados para soportar este tipo de métricas pueden dar una solución aceptable.

5.2. REQUERIMIENTOS DE LAS MÉTRICAS OO PARA TIEMPO DE EJECUCIÓN

Así pues hemos identificado tres requerimientos conceptuales importantes para las RTM (métricas en tiempo de ejecución).

Estos tres requerimientos son:

1. Concurrencia de la ejecución del sistema de métricas y del sistema medido.

2. Separación de los elementos de ejecución y de medida, hasta el punto de que llegaran a estar en máquinas diferentes.

3. Separación conceptual de la implementación y por tanto del diseño de la aplicación que nace con el requerimiento de ser medida.

El primero de los requerimientos es puramente conceptual y es parte de la definición de RTM. Pero requiere, para ser realmente efectivo del segundo de los requerimientos y aún más el sistema que mide no debe interferir (o debe hacerlo en un porcentaje despreciable) en el sistema medido.

Por último, las aplicaciones que se van a medir deben ser diseñadas e implementadas con ese objetivo y por tanto se impone una metodología de diseño que soporte el diseño para ser medido.

Es importante destacar que el punto 3 de los requerimientos está muy cerca del concepto de Programación Orientada al Aspecto, ya que realmente lo que se identifica de forma clara es una aspecto de la aplicación independiente del modelo de negocio de la misma y que es su posibilidad de ser medida y controlada.

En [KICZ97] se establecen dos términos diferentes en la Programación Orientada al Aspecto: el Componente y el Aspecto. En este caso estamos claramente ante un aspecto de la programación: la medición de la aplicación y su control.

Una parte importante de la separación de aspectos ha sido estudiada para ser soportada en tiempo de ejecución [ORTIN02], sin embargo en el sistema que se presenta en esta tesis, la separación debe ser realizada ya en la fase de diseño, de ahí la importancia de establecer una metodología de diseño que permita esta separación y su posterior integración.

5.3. METODOLOGÍA DMT

5.3.1. INTRODUCCIÓN

Mediante esta metodología se va a proponer un modo de llegar desde la idea de medir una aplicación hasta la implementación de dicha idea.

Se van a realizar implementaciones en Java y también se propondrá un superconjunto de Java (JavaRTM) que sería capaz de soportar estos sistemas de métricas (RTM) de una manera mucho más sencilla.

5.3.2. DESCRIPCIÓN DE DMT

DMT es una Metodología para el diseño de aplicaciones de software orientado a objetos [AQU02] sobre el que deben tomarse medidas en tiempo de ejecución (en inglés DMT es Design Methodology for run-Time metrics).

Desde este punto de vista DMT partirá de UML como lenguaje de modelado y establecerá los pasos que deben seguirse desde el diseño conceptual hasta la implementación para dotar al sistema de un conjunto de mediciones que lo monitoricen e incluso que lo controlen.

Por tanto DMT consta de:

1. DMT-Language que ampliará los símbolos de UML para soportar los diagramas de diseño para las métricas y

2. DMT-Fases que establecerá las fases de diseño.

5.3.3. DMT-LENGUAJE, EL SISTEMA DE MODELADO DE DMT

EL sistema de modelado DMT es una extensión de UML para soportar sistemas de métricas en tiempo real. La simbología ha sido basada en el ISA [ISA-84] (para monitorización de procesos industriales) y ha sido reducida para que los símbolos representen de un modo más genérico los diferentes elementos.

En el apartado 5.4 se muestra la simbología desarrollada.

5.3.4. DMT-PHASES, LAS FASES DE DMT

Las fases de DMT son:

5.3.4.1. ABSTRACCIÓN DEL SISTEMA SIN MÉTRICAS

Los primeros pasos del diseño son similares a los que se sigan en cualquier otra aplicación, identificación de casos de uso, objetos, clases, responsabilidades, etc.

En esta fase se debe prever que el sistema irá monitorizado, pero no debe ser objetivo principal, ya que en ningún caso debe condicionar excesivamente el diseño.

Para esta fase se puede seguir el Método Unificado, BOOCH, OORAM o cualquier otra metodología equivalente.

El nivel de profundidad en esta fase dependerá del sistema a modelar y de la importancia que vayan a tener las métricas o el sistema de monitorización.

5.3.4.2. ANÁLISIS DE LAS MÉTRICAS A UTILIZAR

Una vez diseñado el sistema comienzan las fases propiamente de DMT.

El primer paso es una revisión del diseño anotando los puntos de control que se desea monitorizar o controlar.

Esta fase debe ser realizada con precaución para no realizar excesivas mediciones, ya que eso redundará en la eficiencia del sistema final.

Como criterio general, se deben medir aquellos puntos importantes para la calidad final de la aplicación y los puntos que se prevea que puedan autoadaptarse durante la ejecución.

Por ejemplo, en una aplicación donde se crean recursos de manera automática, pero el exceso de recursos redunde en la eficiencia final de la aplicación se deben tomar puntos de control de la eficiencia que habilitarán, destruirán o inhibirán la creación de nuevos recursos hasta que el sistema se vaya a una situación estable en la que el compromiso entre la velocidad de ejecución y la calidad de respuesta del software sean óptimos.

Con este fin hay que diseñar por un lado las métricas aplicables, ya que en la mayoría de los casos serán métricas ad-hoc para esos casos concretos y por otro lado los puntos en los que el sistema de control actuará sobre el sistema en diseño.

Una vez pensado el objeto a medir, se debe tener en cuenta que las métricas tienen unas características que deben de cumplir para que sean significativas [AQU97], por tanto se deben crear métricas que cumplan con las condiciones establecidas en [FEN91].

Por último hay que diseñar la monitorización y la posible intervención del usuario sobre el sistema para modificar su comportamiento.

5.3.4.3. DISEÑO DE LOS PUNTOS DE TOMA DE MUESTRA

Este en un paso muy importante de DMT.

Lo primero es identificar el objetivo de la toma de muestra. Hay que identificar el punto exacto en que se tomará la medida, el objetivo de esta medida y la forma en que esta medida influirá en el sistema de control.

Seguidamente hay que valorar en términos de eficiencia este punto de control, a modo de ejemplo, no se puede tomar una muestra en un punto para medir la velocidad de ejecución mediante un algoritmo que relentice excesivamente el sistema medido.

A continuación se diseña la métrica y por tanto el algoritmo de medición, se valora la métrica en términos de su adecuación a la teoría de la medición y se definen las unidades de medida. En términos generales una medida que carezca de una definición de unidad de medida debería ser revisada.

Se define en qué forma influirá la medida en el control del sistema y se definirá cómo debería ser vista por la monitorización.

En este momento se completan en los diagramas de clases del diseño los puntos de toma de muestra de acuerdo a los símbolos establecidos por DMT y se elaboran diagramas de estados, de actividades o pseudocódigo de los algoritmos que definen el comportamiento de los objetos que harán la función de métricas si su comportamiento no fuera trivial.

5.3.4.4. DISEÑO DE LOS PUNTOS DE CONTROL

Normalmente, cuando se decide medir es para realizar acciones que mejoren el comportamiento de la aplicación. Por tanto también hay que decidir sobre qué elementos u objetos del sistema accionará el control y de qué manera.

En este momento es importante saber qué puntos de medida influyen en el control y el algoritmo que defina esta influencia. Este paso consiste, pues es una revisión del paso anterior pero ahora vista desde el punto de vista global de cómo influyen todas las métricas definidas.

En este punto y si el sistema tiene una función de transferencia (Ver 5.9) conocida, se podrían establecer previsiones del funcionamiento del control del sistema. Éstos podrían ser resultados puramente numéricos o reglas. De este modo se puede prever como se va a comportar el sistema real.

Por último se completarán los diagramas de clases del paso anterior con los accionadores, excepciones y los lazos de control del sistema. También se implementarán diagramas de estados, de actividades o pseudocódigo que defina el comportamiento de los actuadores si su comportamiento no fuera trivial.

5.3.4.5. DISEÑO DE LA MONITORIZACIÓN

Para cada lazo de control que se establezca en el sistema y para métrica en solitario debe preverse un modo de monitorizar o visualizar la información que de ellos se obtiene. Estas visualizaciones consisten normalmente en valores actuales de la métrica, históricos de evolución, estadísticas, etc.

La visualización puede realizarse por cada punto de medida o como elaboración de la medición en varios puntos (métricas complejas) y debe dar una visión de la evolución del sistema.

Para cada bloque de medidas que se desee observar, se debe definir la interfaz donde se mostrarán los datos, normalmente visualización on-line o archivado de tomas de muestras en ficheros de log.

Al igual que en los pasos anteriores, en este momento se completarán los diagramas de clases con los puntos de monitorización, alarmas, etc. Además se implementarán diagramas de estados, actividades o pseudocódigo para definir la monitorización si el comportamiento de esta no es trivial.

5.3.4.6. PASO DEL SISTEMA A LAS CLASES DEL FRAMEWORK O AL LENGUAJE DE PROGRAMACIÓN QUE SOPORTE EL SISTEMA DE MÉTRICAS

Ahora se debe pasar todo el diseño anterior a clases y objetos si la implementación final se hace mediante el framework o bien se estudian las clases afectadas para poder implementar el sistema mediante JavaRTM (Ver 5.8) u otro lenguaje que implemente el control.

En el caso de utilizar directamente el framework, todos los diagramas de estados deben ser desarrollados de manera detallada eliminando los elementos de DMT y definiendo objetos y clases del framework o derivadas de él.

5.3.4.7. IMPLEMENTACIÓN

Por último se implementa el sistema en el lenguaje decidido.

En el caso de JavaRTM u otro lenguaje similar, los propios diagramas de clases extendidos con DMT dan suficiente información como se puede ver en los apartados 5.8.4.1, 5.8.4.2 y 5.8.4.3.

Para el caso de utilizar directamente el framework desde Java, se deben definir las clases que implementarán cada uno de los puntos definidos (métricas, actuadores, monitores, displays, etc.)

Una vez definidas se buscarán implementaciones estándar si existieran y en caso contrario se heredará de las clases abstractas y se implementará la clase que formará parte del sistema.

5.4. SIMBOLOGÍA DMT

5.4.1. BREVE PRESENTACIÓN DEL ISA

El ISA [ISA-84] es un estándar americano para diseño de los elementos de control que forman parte de los procesos industriales.

Se utiliza en los P&I's para definir tanto la posición de toma de muestra como los procesos que se llevan a cabo para el control y los puntos de acción sobre el sistema.

El parecido de estos procesos con los sistemas de métricas objeto de esta tesis ha hecho a este sistema adecuado para servir como inspirador de la simbología adoptada.

5.4.2. DIAGRAMAS EN UML

Una vez definido el sistema se obtendrán los diagramas de diseño en UML [RUM99]. Estos diagramas no tienen diferencias con cualquier otro diseño elaborado en UML a excepción del Diagrama de Clases que en una fase posterior incorporará las adiciones impuestas por el sistema de métricas ([AQU01] y [AQU02]).

5.4.2.1. DIAGRAMAS AFECTADOS POR LA SIMBOLOGÍA

Tres son los diagramas que se ven afectados en UML por esta extensión metodológica, El Diagrama de Clases, el Diagrama de Actividades y el Diagrama de Estados de las Clases.

El Diagrama de Clases se modifica con las extensiones propias del sistema de métricas, esto es, con los símbolos del apartado 5.4.3.

El Diagrama de estados definirá el proceso de control si existiere y si fuere no trivial.

Y el Diagrama de actividades describirá comportamientos no triviales complejos y también lazos de control completos.

En los próximos apartados se desarrolla esta ampliación a UML.

5.4.3. SÍMBOLOS

La simbología utilizada sobre los diagramas UML está inspirada en el ISA [ISA-84].

A diferencia del estándar americano, el enfoque que se ha seguido ha sido generalista, de acuerdo a los propios diagramas utilizados por UML y no ha sido definido para contener acepciones específicas a las métricas utilizadas. Dos han sido las razones que han llevado a este enfoque:

- De un lado la simplicidad con que UML representa los diagramas.
- La dificultad real para estandarizar métricas en software, cosa que no ocurre con los procesos industriales.

5.4.3.1. TOMA DE MUESTRA (MEDIDA)

5.4.3.1.1. MEDIDA SOBRE UN OBJETO O ELEMENTO

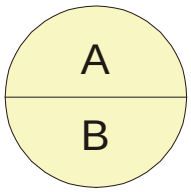


Ilustración 5-1. Medida sobre un objeto o elemento

Medida realizada sobre un objeto o elemento (clase, atributo, método). A es el nombre del objeto de la clase medida que se devuelve, en el framework de JavaRTM será `RTMMeasure`. B es el objeto observado o la explicación del elemento medido cuando lo que se observe no sea un objeto concreto.

Este elemento representa una medida primaria y simple sobre alguna parte de la aplicación. El resultado de la medida puede ser expresado en cualquier tipo de escala (ratio, nominal, ...).

Ejemplos de medidas son el número de usuarios en el sistema, el número de llamadas a un método, el tamaño de un objeto, etc.

5.4.3.1.2. AGRUPACIONES DE MEDIDAS

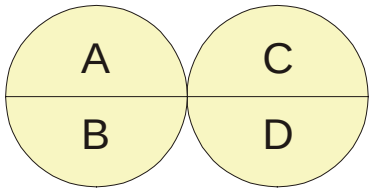


Ilustración 5-2. Agrupaciones de Medidas

Varias medidas que comparten un mismo fin. Pueden ser varias medidas sobre el mismo elemento, sobre la misma clase o varias medidas del mismo tipo que van a colaborar en una medida final compleja.

Ejemplos de este tipo de elemento se producen cuando se desea medir en el mismo objeto el número de accesos y el número de rechazos de acceso para poder obtener un ratio de accesos con éxito al sistema.

5.4.3.1.3. MEDIDA QUE SE VA A MONITORIZAR EN EL SISTEMA

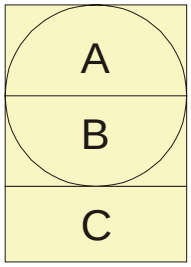


Ilustración 5-3. Medida a Monitorizar

Medida que se va a monitorizar en el sistema, esto es, cuyo valor debe poder ser observado desde la monitorización.

A y B significan lo mismo que en el apartado 5.4.3.1.1 y C es el display que se usará para la monitorización. C puede corresponderse con una fichero, una pantalla, un campo, una ventana, etc. de forma no exclusiva.

El sistema diseñado en esta tesis incluye la posibilidad de monitorización on-line del sistema bajo control, por tanto puede ser necesario ver la evolución de una medida que se esté tomando o simplemente su valor instantáneo con el objetivo de poder intervenir desde el sistema de control

5.4.3.1.4. COMPOSICIÓN DE MEDIDAS

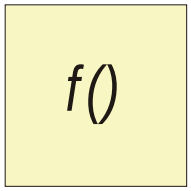


Ilustración 5-4. Composición de Medidas

Composición de medidas. Varias medidas que colaboran de forma casual o permanente para dar una sola salida.

La función *f()*, representa el algoritmo por medio del cual colaboran. También puede ser una colaboración muy compleja a través de diferentes objetos, en este caso *f()* pasa a ser simplemente un símbolo y las clases se representan en UML.

Ejemplos de utilización de este elemento se producen cuando la colaboración requiere de la aplicación de algoritmos matemáticos complejos, por ejemplo cuando el resultado final de la medida consiste en la integración de la evolución de varias medidas durante el tiempo de control.

5.4.3.2. OPERACIÓN LÓGICA

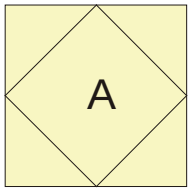


Ilustración 5-5. Operación lógica

Operación lógica o de monitorización que se realiza sobre el sistema.

El monitor del sistema tiene este símbolo. A es la especificación de lo que representa, puede ser el nombre de la clase o del componente que hace de monitor.

Ejemplos de este caso se producen cuando se desea intervenir en el control del sistema desde la interfaz de monitorización del usuario para cambiar el comportamiento de la aplicación. El procedimiento de intervención da lugar a una serie de operaciones lógicas (implementadas en el código) que no están previstas en el framework original y requiere de su especificación concreta en la fase de diseño.

5.4.3.3. ALARMAS

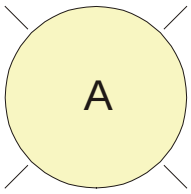


Ilustración 5-6. Alarma

Alarma que debe darse al operador del sistema o a algún elemento activo del sistema para que tome decisiones. Las alarmas se producen ante eventos que pueden ser corregidos sin parar el sistema.

A representa el texto de la alarma y/o el destino de dicha alarma.

Las alarmas son eventos que han sido programados en el sistema de control y que deben ser informados para que el usuario del sistema de control tome decisiones de intervención.

Ejemplos de alarmas se pueden producir cuando un objeto está midiendo la cantidad de recursos disponibles y hemos determinado que cuando el sistema haya ocupado un porcentaje concreto de recursos (p.e. 90%) el usuario del sistema de control sea avisado para impedir la concesión de nuevos recursos. En cualquier caso la decisión no se toma en automático, sino que es el usuario el que es avisado para poder intervenir.

5.4.3.4. ACTUADORES

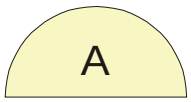


Ilustración 5-7. Actuador

Representan a los objetos que toman parte activa en el sistema actuando de manera automática.

Estos elementos son los que actuarán sobre los objetos que hayan sido determinados en la aplicación bajo control para ser intervenidos desde el sistema en automático o a través de la intervención del usuario del sistema de control.

Un ejemplo de utilización puede ser el control de accesos a un servidor con limitación de licencias de uso. Un elemento medirá el número de conexiones al sistema y cuando se alcance el número máximo de licencias de uso el actuador inhibirá la entrada de nuevos usuarios.

5.4.3.5. AGRUPACIÓN DE ACTUADORES

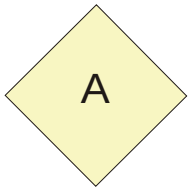


Ilustración 5-8. Agrupación de Actuadores

Cuando la misma acción dispara varios actuadores se encadenan a través de este símbolo.

Este elemento reparte el evento producido entre los diferentes actuadores.

Este elemento es necesario cuando un evento producido en el control debe intervenir en diferentes partes de la aplicación y no sólo en un punto concreto. La intervención se supone que es simultánea.

Un ejemplo de uso de este elemento se produce cuando el sistema está controlando la impresión de informes y da preferencia a los trabajos cortos.

- Al aparecer un trabajo largo se debe cambiar el objeto que controla la impresora para que almacene temporalmente el trabajo al mismo tiempo que se envía una alarma al usuario.

- Al quedar la impresora sin trabajos se vuelve a activar el trabajo largo y se envía otra alarma al usuario al mismo tiempo.

En ambos casos se necesita actuar sobre más de un objeto.

5.4.3.6. EXCEPCIONES

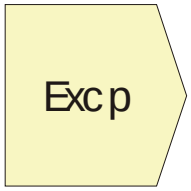


Ilustración 5-9. Excepción

Este símbolo representa un fallo severo en el sistema. El sistema por tanto lanzará una excepción y normalmente será parado.

Las excepciones se implementan con los elementos propios de los lenguajes.

Este elemento se puede utilizar por ejemplo, para expresar que el control que se hace sobre un elemento físico (dispositivo conectado al sistema) ha fallado y no se puede recuperar la conexión.

5.4.3.7. CONECTORES

Ilustración 5-10. Conector de punto de toma de muestra

Ilustración 5-11. Conector genérico

El conector consiste en una línea que parte de un punto y termina en una flecha.

El punto indica el lugar de medida y la flecha indica el elemento al que va dirigida dicha medida.

Cuando se trata de conexiones entre elementos del sistema de métricas o que parten de ellos para display o desde los actuadores hacia el sistema controlado no se coloca el punto inicial.

5.4.4. EJEMPLOS DE UTILIZACIÓN

5.4.4.1. CASO 1

La clase **Procesos** lanza la ejecución de threads, pero se ha detectado que en el sistema actual, un número de threads mayor que 4 provoca el enlentecimiento desmesurado del sistema, por lo que se desea que nunca se creen más de 4 threads.

Observese que en este caso no hay un elemento de agrupación de actuadores, ya que es el mismo actuador el que hace o bien activar o bien inhibir el sistema y por tanto no se produce concurrencia de operaciones.

La representación de este punto de control es como sigue:

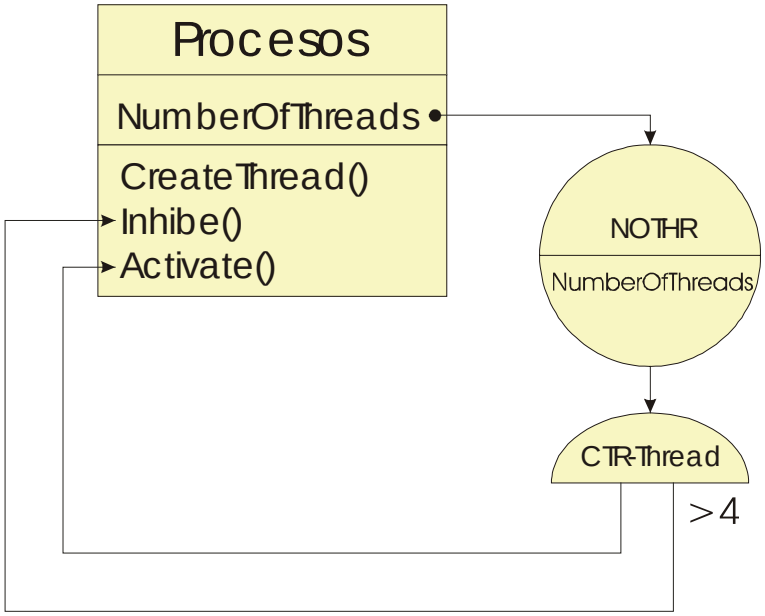


Ilustración 5-12. Lazo de control básico

El gráfico anterior representa un sistema sobre el que se va a medir el atributo *`NumberOfThreads`* de la clase `Procesos` y sólo se enviará `Activate()` si *`NumberOfThreads`* es menor o igual que 4, si es mayor se enviará el mensaje `inhibe()`.

Una aproximación a la descripción mediante diagramas de estados se puede ver en los siguientes diagramas:

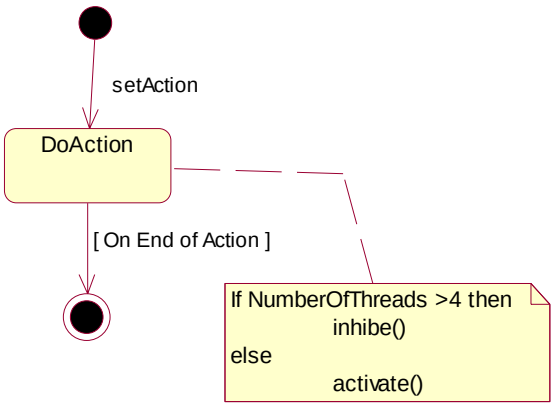


Ilustración 5-13. Diagrama de estados de CTR-Thread

Este diagrama representa el comportamiento de `CTR-Thread`, esto es, cada vez que se le pide que realice `setAction`(), comprobará el número de threads en ejecución y habilitará o inhibirá la posible creación de nuevas threads.

En este caso el comportamiento es trivial y no sería necesario representarlo mediante un diagrama.

También es importante el comportamiento que tenga la clase encargada de hacer las mediciones, en este caso `NOTHR`. Aunque de nuevo es un comportamiento trivial, en el diagrama siguiente se desarrolla el comportamiento dinámico de `NOTHR`.

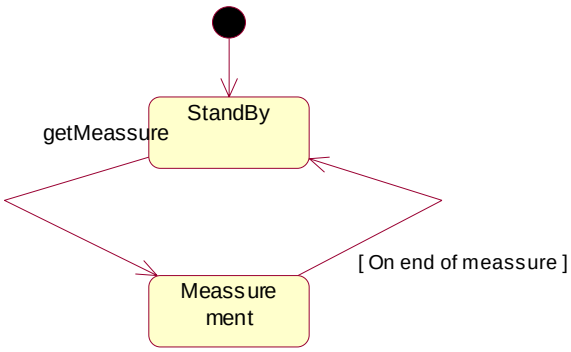


Ilustración 5-14. Diagrama de estados de NOTHR

5.4.4.2. CASO 2

En este caso, se tienen varios procesos `Productor` y un objeto `Consumidor` que envían y recogen la información de un objeto intermedio `Buffer`. Si el buffer está muy lleno se deben eliminar procesos `Productor` y crearlos en caso contrario.

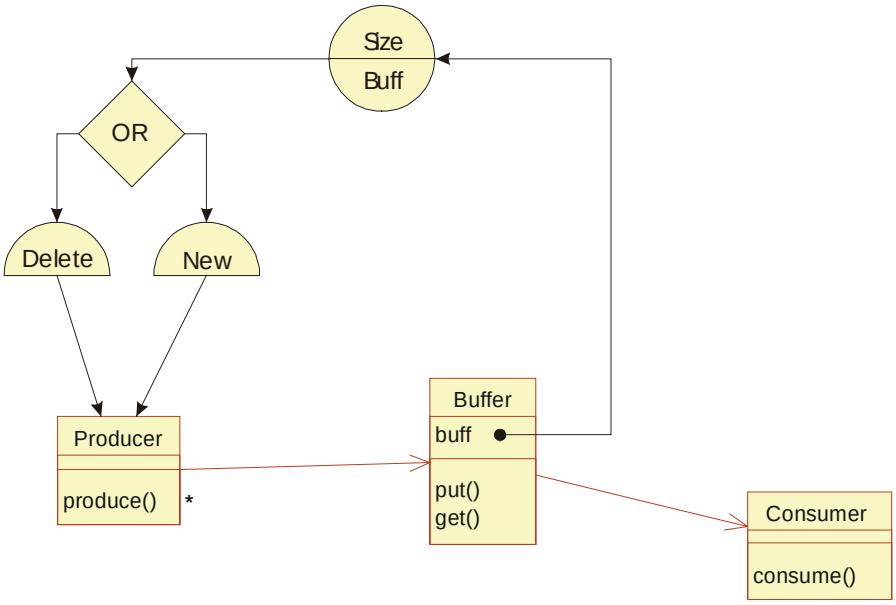


Ilustración 5-15. Lazo de control del productor-consumidor

En este caso la acción es compuesta pero exclusiva, o bien se producen nuevos objetos `Producer` o bien se borran.

El único proceso que en este caso es trivial, es el de producción o borrado de objetos, por tanto sólo se va a tener un diagrama de estados.

Al igual que en el ejemplo anterior define el comportamiento de la clase `OR`, siendo la acción en este caso la llamada al actuador `NEW` o al actuador `DELETE`.

No deben confundirse estos actuadores con los operadores `new` y `delete` presentes en el lenguaje C++, sino que pueden representar algoritmos de borrado y de creación muy complejos y ajustados a los requerimientos del sistema modelado.

El siguiente diagrama muestra este comportamiento.

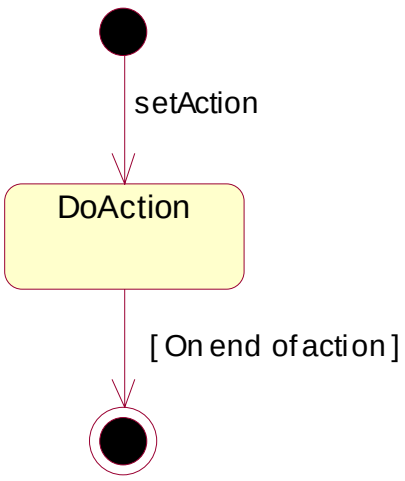


Ilustración 5-16. Diagrama de Estados

En muchos casos un simple diagrama de estados no representa todos los matices del comportamiento y entonces es importante realizar diagramas de actividades que amplíen la información.

En el caso del ejemplo anterior se hizo el algoritmo con un simple comentario, pero ahora es más complejo y es necesario utilizar un diagrama de actividades:

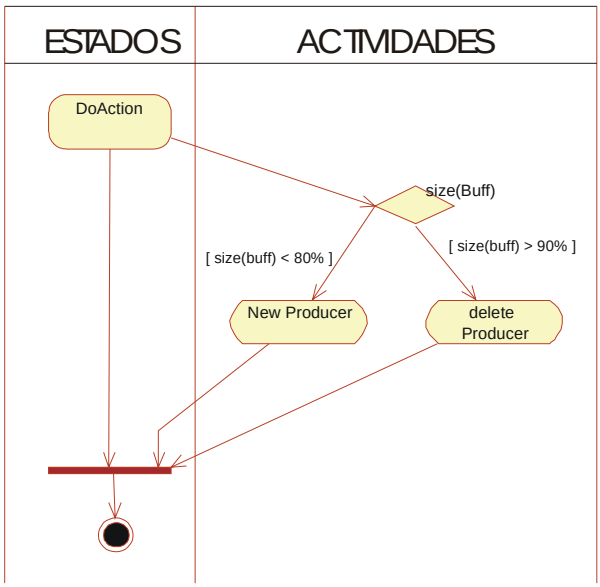


Ilustración 5-17. Diagrama de Actividades

5.5. FRAMEWORK PARA APLICAR DMT A DESARROLLOS EN JAVA JDMT

5.5.1. INTRODUCCIÓN

Siguiendo una tendencia casi habitual (ver JhotDraw [@GAHJD], Junit [@JUNIT]), la manera en que se deben implementar funcionalidades concretas dentro de los sistemas OO es mediante frameworks.

En este caso se ha seguido esta idea y se ha desarrollado un framework ([AQU01] y [AQU02]) completo que soporta la filosofía general del sistema, desde la toma de muestra, hasta el punto en que se vuelve a actuar sobre el sistema en función de las mediciones realizadas.

De este modo se describen todas las relaciones que deben tener los diferentes elementos que componen el framework y se define la filosofía de funcionamiento que debe soportar.

Esta filosofía se modela y se establecen las responsabilidades de cada uno de los componentes para que la colaboración tenga el efecto deseado.

El esquema general sigue la filosofía representada en el siguiente diagrama:

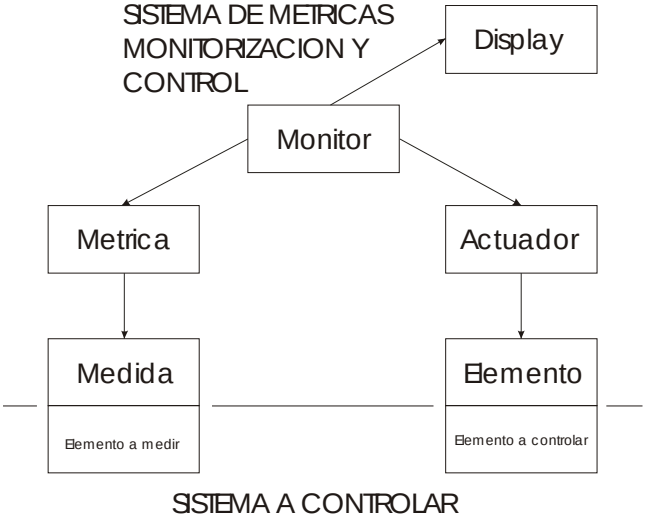


Ilustración 5-18. Esquema de la filosofía de funcionamiento del framework

Tanto el *elemento a medir* como el *elemento a controlar*, son parte del sistema que se desea controlar o monitorizar. La monitorización (*monitor*) hace de centro de control del sistema, define métricas (*metrica*) y elementos actuadores (*actuador*) que tomarán acciones sobre el sistema de nuevo en función de los valores medidos.

Las cajas *medida* y *elemento*, representan la parte del sistema de métricas y control que interacciona con el sistema a medir o a controlar.

Del esquema anterior se deduce que el sistema a medir debe ser modificado o adaptado para conectarse a los puntos de medida y de control. Por tanto el sistema tal y como se ha diseñado solo puede ser utilizado cuando se dispone de los códigos fuente.

Sería posible, sin embargo, una adaptación para utilizarlo con códigos objeto (bytecodes de java) siguiendo unos principios similares a los utilizados para desarrollar la herramienta de "Programación por contrato en ficheros .class" [AQU98]

El sistema de medida es independiente del sistema medido y esto permite que pueda estar funcionando en una thread independiente e incluso en otra máquina que se comunique con el sistema a través de RMI o CORBA.

5.5.2. DESCRIPCIÓN DEL FRAMEWORK

A la hora de dar un soporte software a todo el sistema se valoró entre la posibilidad de implementar una librería de componentes basados en patrones de diseño [LAN98] o en la realización de un framework. Esta última opción tiene ventajas adicionales, como por ejemplo dejar mayor libertad para las implementaciones, de modo que se pueden adaptar más fácilmente al sistema a medir.

En la elaboración del framework se ha seguido el esquema de diseñar una serie de interfaces que implementan el diseño básico del sistema y que se desarrollarán siguien-

do modelos de patrones de diseño [GAMM94]. El estilo ha sido inspirado por el JHotDraw [@GAHJD] y el JUnit [@JUNIT].

En el esquema se definen las relaciones básicas entre las interfaces pero no se implementa ninguna solución concreta, por tanto sólo se definen relaciones e interfaces.

En un segundo nivel y a través de clases abstractas se darán soluciones estandarizadas para las interfaces e implementaciones por defecto.

Por último una capa de clases estándar implementarán métricas básicas, monitores, actuadores, etc.

Este esquema básico de interfaces se puede ver en el siguiente diagrama:

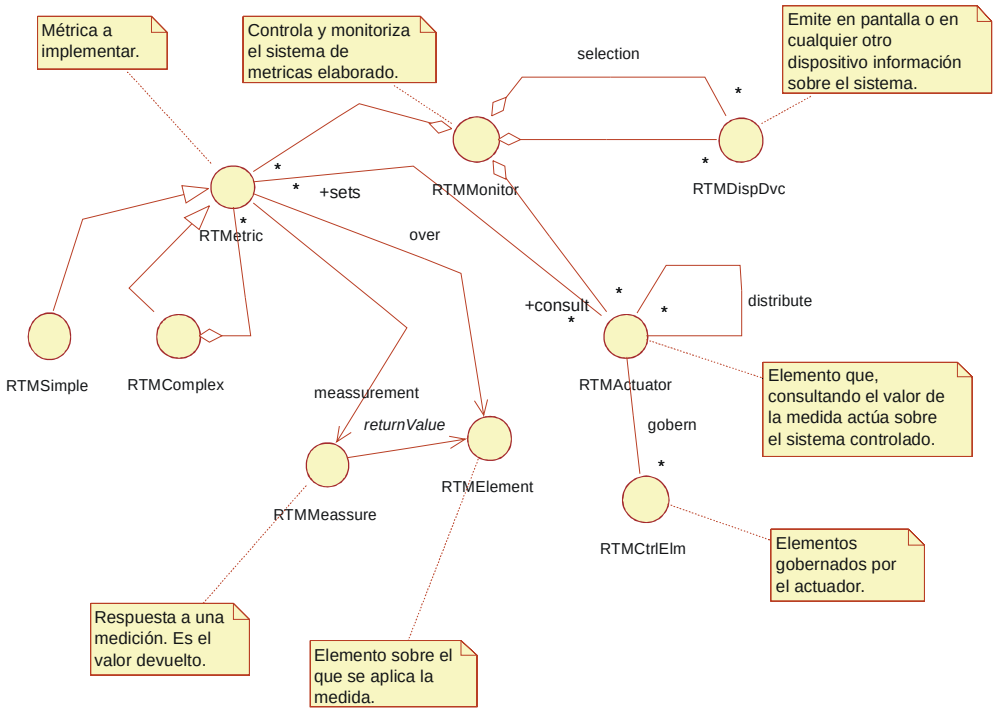


Ilustración 5-19. Diagrama de interfaces (roles) del framework

El esquema consta de nueve interfaces básicas que colaboran dentro del sistema: `RTMetric`, `RTMSimple`, `RTMComplex`, `RTMMeasure`, `RTMElement`, `RTMMonitor`, `RTMActuator`, `RTMCtrlElm` y `RTMDispDvc`.

El monitor (`RTMMonitor`) es el centro de control del sistema y de él depende el funcionamiento del resto. Básicamente debe identificar los sistemas de salida de datos (`RTMDispDvc`), las métricas a usar (`RTMetric`) y los actuadores (`RTMActuator`) y ponerlos en comunicación para que colaboren en los objetivos planteados al sistema de monitorización.

5.6. INTERFACES Y COLABORACIONES

5.6.1. RTMMONITOR

Es la interfaz principal del sistema y es el centro de control para cualquiera que utilice el framework. Registra todas la métricas, displays y los actuadores y monitoriza para el usuario aquellas medidas que se hayan definido.

La interfaz básica especifica métodos para registrar todos los elementos básicos (`RTMetric`, `RTMActuator`, `RTMDispDvc`) aunque no indica de qué manera se deben almacenar, decisión que deberá ser tomada en implementación y que dependerá de la complejidad del sistema.

Tampoco indica como deben relacionarse las distintas métricas, accionadores y displays (vistas) dentro de `RTMMonitor` y por tanto también esta política queda diferida a una posteior implementación.

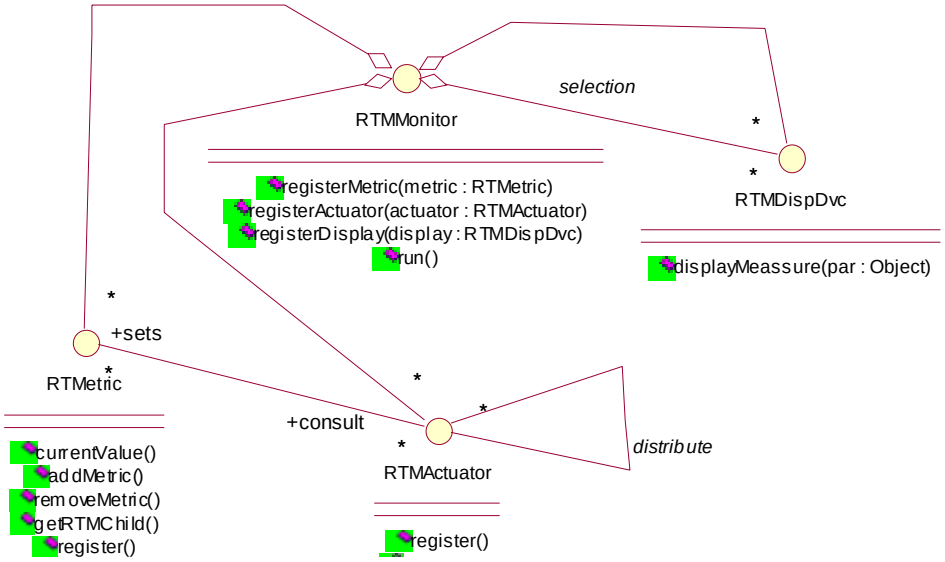


Ilustración 5-20. Subsistema Monitor

El método `run`() define el funcionamiento global del sistema. Tampoco a este nivel se define cual debe ser el modo de funcionamiento y dependerá del sistema que se esté monitorizando. Algoritmos adecuados pueden ser:

- Un polling que recorra uno a uno todos los elementos definiéndoles un slot para que realicen funciones atómicas.
- Sistemas independientes que corran en threads independientes y donde run() simplemente es llamado por una interrupción para actualizar los datos cada cierto tiempo.
- Cualquier otro algoritmo complejo y adaptado a la solución concreta del sistema que se está modelando.

De los párrafos anteriores se desprende la posibilidad de que `RTMMonitor` deba estar en un hilo de ejecución independiente. No es la única solución, pero sí es la mejor en gran parte de los casos.

No sólo en una thread, sino que en algunos casos es posible utilizarla como un agente móvil que resida en el ordenador de menor carga y se encargue de comunicarse con el resto del sistema a través de RMI o mediante un ORB.

La interfaz básica carece de comportamiento definido, pero en el paquete que acompaña al framework se define la clase `StRTMMonitor` que implementa un monitor básico que simplemente pone en comunicación a las métricas y los actuadores y en su método `run()` reparte slots para que todos los elementos funcionen al mismo tiempo.

En la siguiente figura se puede ver la jerarquía de la clase `StRTMMonitor` a partir de su interfaz básica en el framework.

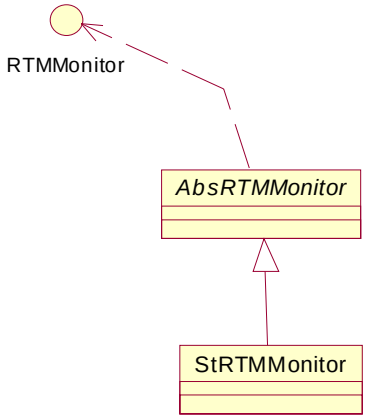


Ilustración 5-21. Desarrollo en clases

5.6.2. RTMETRIC

Es la interfaz de la métrica que se haya implementado. Puede ser simple o compleja y para implementar esta última se utiliza el patrón *composite* (`RTMSimple` y `RTMComplex`).

Se encarga de comunicarse con `RTMElement` que es la interfaz que debe cumplir el punto de medida en el sistema bajo observación y como resultado de una operación de medida recibirá un objeto con la interfaz `RTMMeasure`.

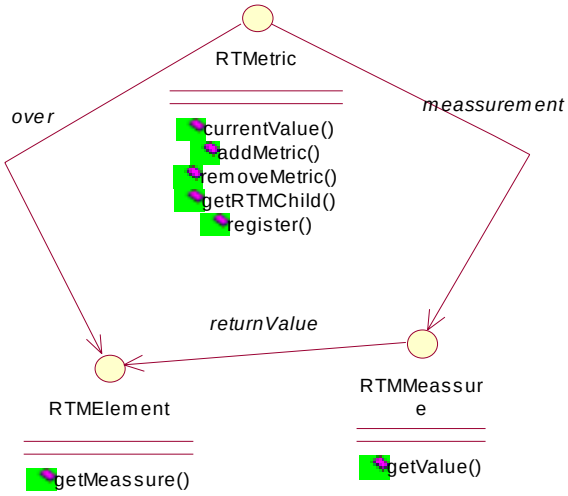


Ilustración 5-22. Subsistema de las métricas

La interfaz básica no define ninguna métrica concreta, sino sólo el soporte para definir métricas. Esto permite que se puedan establecer métricas ad-hoc en un diseño concreto.

El hecho de recibir un objeto `RTMMeasure` como resultado de la medida permite definir diferentes tipos de métricas sin importar el rango o la escala en la que miden, ya que el responsable final de ello será el resultado de la medida y no lo que se haga con ella desde la métrica.

La responsabilidad de la métrica `RTMetric` por tanto, es medir el elemento final y no decidir sobre el resultado de dicha medición.

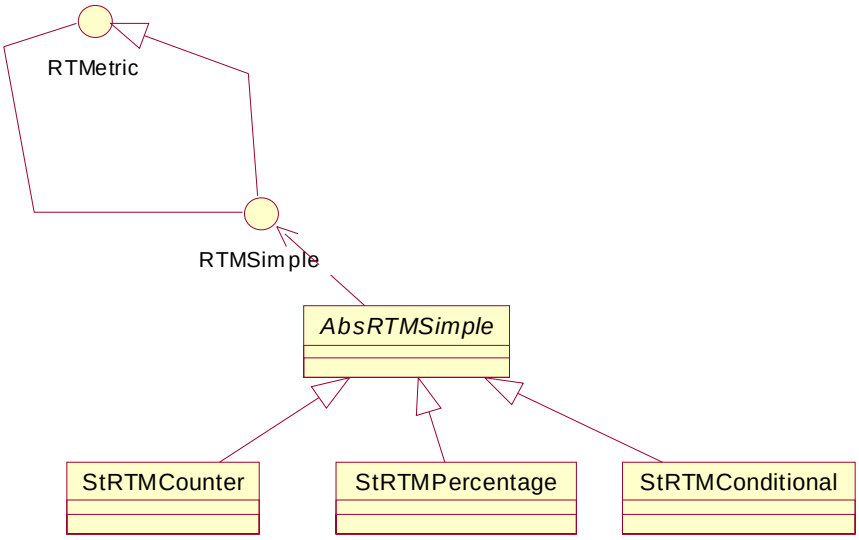


Ilustración 5-23. Desarrollo en clases

En el desarrollo del paquete que acompaña al framework se definen, sin embargo las clases `StRTMCounter`, `StRTMPercentage` y `StRTMConditional` que implementan tres tipos estándar de métricas básicas.

La clase `StRTMCounter` implementa un contador de eventos (entendiendo en este caso por evento cualquier suceso contabilizable, como por ejemplo número de pasadas por un sitio concreto del código, número de llamadas a un método, número de clientes conectados, etc.) Esta clase devolverá un objeto de la clase `StRTMInteger`.

`StRTMPercentage` implementa mediciones que devuelven valores reales, por ejemplo mediciones de tiempos, mediciones de elementos que se obtengan mediante aplicación de funciones reales, etc. Esta clase devolverá como medida un objeto de la clase `StRTMDouble`.

`StRTMConditional` implementa mediciones de tipo booleano, como por ejemplo comprobar la ocurrencia de sucesos, valores dentro de rangos, etc. Esta clase devolverá como medida un objeto de la clase `StRTMBoolean`.

5.6.3. RTMMEASURE

Es la interfaz de la medida realizada. Cada vez que se haga una medida se devolvera un objeto que cumpla esta interfaz para que pueda ser interrogado por el monitor.

Al igual que en los casos anteriores, en la interfaz no se define un comportamiento concreto de esta interfaz, pero si se acompaña al framework de un paquete básico que implementa varias medidas estándar: `StRTMInteger`, `StRTMDouble` y `StRTMBoolean`.

Por tanto, `RTMMeasure` ofrece al resto del sistema de control y monitorización una interfaz básica de acceso al resultado de la medida sin importar el tipo que deba tener el resultado de la medida y que será responsabilidad de quien tenga que tratar dicha medida.

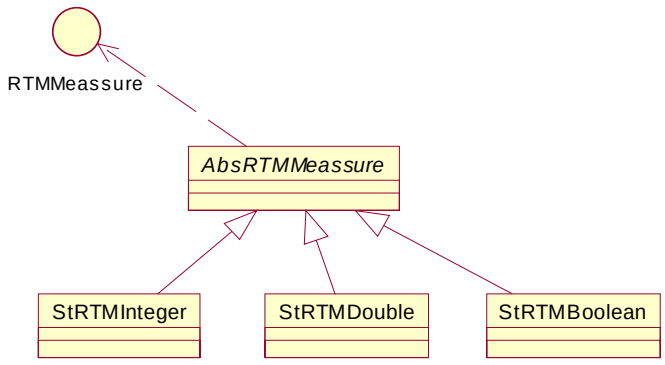


Ilustración 5-24. Desarrollo en clases de RTMMeasure

`StRTMInteger` maneja valores enteros y está asociada, como ya se ha dicho en párrafos anteriores, a la métrica `StRTMCounter`.

`StRTMDouble` maneja valores reales (double) y está asociada a `StRTMPercentage`.

`StRTMBoolean` maneja valores boléanos y está asociada a `StRTMConditional`.

5.6.4. RTMELEMENT

Cada objeto que contenga atributos o características que vayan a ser medidas deberá implementar esta interfaz para que `RTMMeasure` pueda interrogarlo.

Esta interfaz es, pues, la puerta de comunicación entre el sistema de métricas y el sistema medido.

La forma en que se conecta el sistema de monitorización a la aplicación que se desea controlar es haciendo una nueva clase a partir de la que se desea controlar y que implemente la interfaz `RTMElement` de modo que permita al sistema de control poder interrogar a dicha clase.

5.6.5. RTMACTUATOR

Implementa los actuadores que interaccionan con el sistema. A nivel de interfaz no está definido un patrón concreto para realizar este subsistema, pero en general puede ser un patrón *composite* para tener a todos los actuadores que colaboran con una `RTMetric` y un patrón *chain of responsibility* para pasar los eventos.

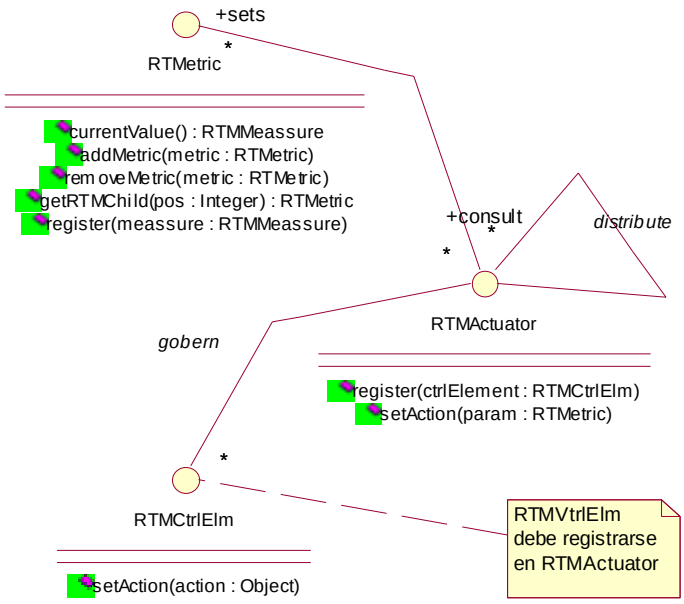


Ilustración 5-25. Subsistema de actuadores

La relación entre `RTMetric` y `RTMActuator` puede ser la de un *observer* de éste último para actuar sobre el sistema controlado en función de los cambios de valor de `RTMetric`, dependerá a su vez de la implementación que se haya realizado en `RTMMonitor` para relacionar las métricas y los actuadores.

Otros patrones que pueden encajar son *strategy*, *state*, *command*, etc.

Sólo se implementa una clase estándar en el paquete: `StRTMGeneric`.

Esta clase implementa por omisión para reenviar el evento de acción al objeto controlado y representado por `RTMElement`.

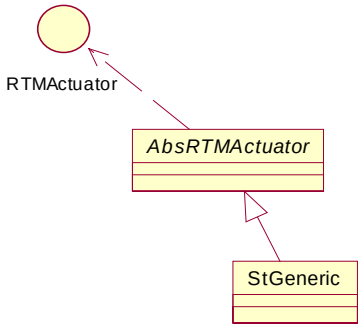


Ilustración 5-26. Desarrollo en clases

5.6.6. RTMCTRLELM

Esta interfaz debe ser implementada por el elemento bajo control del actuador, a través de ella, el `RTMActuator` puede controlar el comportamiento de este último. El patrón aquí puede ser de nuevo un *observer*.

Esta interfaz es la puerta de salida de las acciones de control sobre el sistema controlado.

Por sus características no hay implementación en las clases estándar.

5.6.7. RTMDISPDVC

Implementa la interfaz para componer la información que se mostrará al usuario. Aquí pueden encajar varios patrones, la relación entre esta interfaz y `RTMMonitor` puede implementarse a través de un *observer*, asimismo, en las relaciones entre los diferentes objetos de esta interfaz encajan varios patrones (*composite*, *chain of responsibility*, etc.)

5.7. IMPLEMENTACIÓN DE LA SOLUCIÓN

Para implementar la solución siguiendo la filosofía estándar del framework se debe seguir la norma de pasar por tres niveles de adaptación: Interfaz, Clase Abstracta y Clase Concreta para cada uno de los elementos que componen el framework.

Así pues, entre el nivel de interfaces básicas del framework y el nivel de clases concretas capaces de ser instanciadas en objetos, se puede pasar por un nivel intermedio de clases abstractas que definan de un modo más preciso la solución para un conjunto importante de diferentes sistemas, las responsabilidades por defecto e implementen algunos patrones de diseño que acerquen la solución concreta.

El esquema básico del framework consta de tres paquetes:

1. Nivel de interfaces, en el que se definen las relaciones principales entre los elementos del sistema de medida. Este nivel es realmente el framework.

2. Nivel de clases abstractas, en el que se implementan algunos comportamientos básicos del sistema. Este nivel no es de uso obligado y para soluciones concretas puede ser redefinido y adaptado. Simplemente en el paquete básico sirve para implementar la mayor parte de los casos más comunes, para dar soporte al siguiente nivel y para servir de ejemplo de desarrollo del framework.

3. Nivel de clases estándar, en el que se definen algunas métricas concretas, actuadores y monitores. En este nivel, ademas de dar un ejemplo de desarrollo completo del framework hasta el nivel de clase concreta, se han implementado algunas métricas comunes que se pueden utilizar en la mayor parte de los casos.

A medida que se sube en el nivel, las implementaciones son más concretas y menos generalistas, por lo que sólo el primer nivel es de cumplimiento obligado al desarrollar software utilizando el framework, esto es, para utilizar este framework sólo se debe respetar el nivel de interfaces, quedando el nivel de clases abstractas como una simple proposición de acercamiento a la solución y no como un paso obligado.

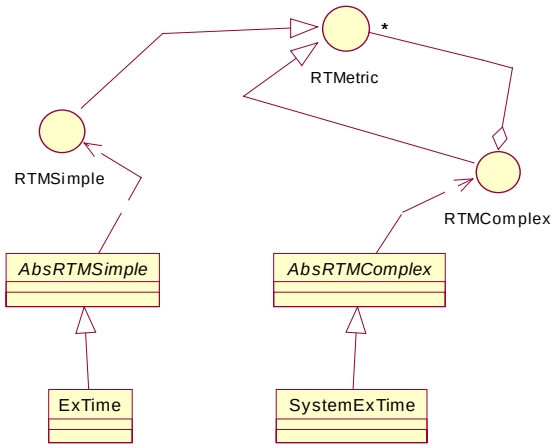


Ilustración 5-27. Ejemplo de desarrollo en clases de RTMetric

En general en las clases abstractas se propondrá una solución menos generalista, pero muy común a diferentes sistemas. Esto hace que en ciertos momentos este paso intermedio no sea de utilidad en algunos sistemas complejos y se deba reimplementar para ellos.

Como de ejemplo, `RTMetric` puede ser desarrollada en los tres niveles propuestos como se ve en el diagrama de la Ilustración 5-27.

En este diagrama se observan los tres niveles de desarrollo para las interfaces `RTMSimple` y `RTMComplex`.

Por último, las excepciones asociadas al framework se definen en un paquete concreto. Una solución podría haber sido utilizar excepciones estándar, pero parece más lógico poner un punto de partida para todas las excepciones asociadas al framwork, esta es la labor de `RTMException`.

Así pues y considerando todos los aspectos comentados en los párrafos anteriores, el paquete básico consta de los siguientes `packages`:

- es.uniovi.ootlab.metrics.framework
- es.uniovi.ootlab.metrics.abstracts
- es.uniovi.ootlab.metrics.metrics
- es.uniovi.ootlab.metrics.standards
- es.uniovi.ootlab.metrics.exceptions

5.7.1. ES.UNIOVI.OOTLAB.METRICS.FRAMEWORK

Este `package` contiene las interfaces básicas del framework: `RTMetric`, `RTMSimple`, `RTMComplex`, `RTMMeasure`, `RTMElement`, `RTMMonitor`, `RTMDispDvc`, `RTMActuator` y `RTMCtrlElm`.

Configura el esquema básico de funcionamiento del sistema de métricas.

5.7.2. ES.UNIOVI.OOTLAB.METRICS.ABSTRACTS

Este `package` contiene las clases abstractas del framework: `AbRTMetric`, `AbRTMSimple`, `AbRTMComplex`, `AbRTMMeasure`, `AbRTMElement`, `AbRTMMonitor`, `AbRTMDispDvc`, `AbRTMActuator` y `AbRTMCtrlElm`.

Implementa el nivel intermedio de definición de clases para la implementación del framework. La utilización de estas clases no es obligatoria y su comportamiento se puede modificar ad-hoc en cada diseño.

5.7.3. ES.UNIOVI.OOTLAB.METRICS.METRICS

Las clases estándar se han dividido en dos diferentes `package's`: `metrics` y `standards` con la intención de diferenciar a las métricas por su importancia dentro del sistema.

El `package` contiene las clases concretas del framework que implementan métricas concretas: `StRTMCounter`, `StRTMPercentage` y `StRTMConditional` y también las clases que implementan las medidas por su estrecha relación con las métricas: `StRTMInteger`, `StRTMDouble` y `StRTMBoolean`.

5.7.4. ES.UNIOVI.OOTLAB.METRICS.STANDARDS

El `package` contiene el resto de las clases concretas del framework que implementan elementos concretos: `StRTMMonitor` y `StRTMGeneric`.

5.7.5. ES.UNIOVI.OOTLAB.METRICS.EXCEPTIONS

Aunque no se ha hablado demasiado de este `package` hasta este momento, se ha definido una única clase para implementar las excepciones (ver párrafos anteriores).

Dentro del framework se utilizan excepciones en algunos métodos para definir situaciones de excepción y de error. Para ello se ha propuesto una clase que hereda de `Exception` y que puede ser ampliada a su vez mediante herencia para desarrollar toda una jerarquía de excepciones si fuera necesario.

La única clase de excepción propuesta es: `RTMException`.

5.7.6. DETALLES

En el CD-ROM que acompaña a esta tesis se encuentra un diseño detallado de cada interfaz y de cada clase desarrollada y en los apéndices se especifica el diseño.

5.8. LENGUAJES CON SOPORTE PARA MÉTRICAS

5.8.1. INTRODUCCIÓN

Al igual que el control de excepciones en C++ o la declaración de precondiciones en Eiffel [MEY00], las métricas integradas en el lenguaje deben tener un tipo de declaración exclusiva y bien diferenciada del resto del lenguaje, para recalcar su diferencia de cometidos.

Otra vez se vuelve a estar ante la separación de aspectos, ahora desde el punto de vista del propio lenguaje, en este caso se van a separar los aspectos hasta el punto de hacer dos sublenguajes diferentes: uno para la lógica de negocio y otro embebido en el anterior para definir la lógica (el aspecto) de la medición y el control de la aplicación.

En los próximos párrafos se va a describir una aplicación de Java para soportar un lenguaje de este tipo. A este lenguaje le llamaremos JavaRTM.

En el diseño de JavaRTM se ha usado los mismos criterios que en el desarrollo de la herramienta “Programación por contrato en aplicaciones Java” [GON00]. Por otro lado ha sido importante la experiencia obtenida en el desarrollo de otros lenguajes de programación [AQU94] para intentar separar adecuadamente los dos aspectos implicados: lógica y medición.

5.8.2. REQUERIIENTOS DE JAVARTM

Tres son los requerimientos principales que vamos a insertar en este lenguaje:

- Independencia entre el lenguaje imperativo y el definido para las métricas.
- El lenguaje de las métricas debe tener un carácter más declarativo que imperativo.
- Debe ser posible pasar de JavaRTM a Java + framework de métricas.

5.8.3. FRAMEWORK DE SOPORTE

Para soportar el sistema de métricas, el lenguaje debe contar con un framework básico de soporte, al igual que ocurre con los sistemas de control de excepciones en Java y en C++.

En este caso el sistema básico de framework que se utiliza es el mismo que se había diseñado para el soporte de estos elementos en Java ampliado con algunas clases implementadas por omisión, de manera que en los casos básicos la programación se reduzca considerablemente.

5.8.4. CASOS DE UTILIZACIÓN

5.8.4.1. CLASE A MEDIR

Se tiene una clase sobre la que se desea hacer una medición de un atributo, por tanto el lenguaje debe especificar el punto a medir, por ejemplo:

```
class A {
   private int i;
   public int getI {return i;}
}
```

En este caso se desea medir el atributo i. El diagrama de clases sería:

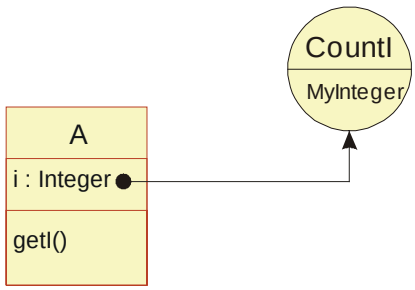


Ilustración 5-28. Ejemplo de diagrama con toma de muestra de una métrica

Por tanto la clase debe especificar el punto de control, la clase `RTMetric` que será realmente la métrica a aplicar, la clase `RTMMonitor` que será la encargada de supervisar la medición y la clase `RTMMeasure` correspondiente, por tanto quedaría:

```
class A {
   //# metrics begin:      anonymous
   //# metric class:          CountI     ## Clase RTMetric
```

```
   //# measure class:      MyInteger  ## Clase RTMMeasure
   //# element code:         { return MyInteger(i); }
   //# monitor class single: MyMonitor  ## Clase RTMMonitor
   //# metrics end:          anonymous

   private int i;
   public int getI() {return i;}
}
```

Esta parte es puramente declarativa y simplemente informa de cuales son las conexiones que tendrá el sistema. Por su parte la clase A será del tipo `RTMElement` y por tanto deberá implementar su interfaz, pero para simplificarlo se pone dentro de la declaración con la directiva `//# element code`.

Las clases `MyInteger`, `CountI` y `MyMonitor` deben ser implementadas o pertenecer a las clases estándar que acompañan al framework.

Se puede comprobar que el punto de medida es importante en este sistema de métricas y desde él parten todas las conexiones que deba implementar el diseño a partir del framework.

En este caso no se ha declarado en el diagrama específicamente el monitor, esto puede hacerse cuando su implementación resulte trivial y por tanto se refiera a uno de los estándares, pero debe ser declarado expresamente en el código.

5.8.4.2. CLASE QUE SE MIDE Y SE MONITORIZA

Ahora se desea que el valor medido active una alarma y además sea almacenado en un fichero de log.

Partiendo de la misma clase del apartado anterior, ahora el diagrama de clases sería:

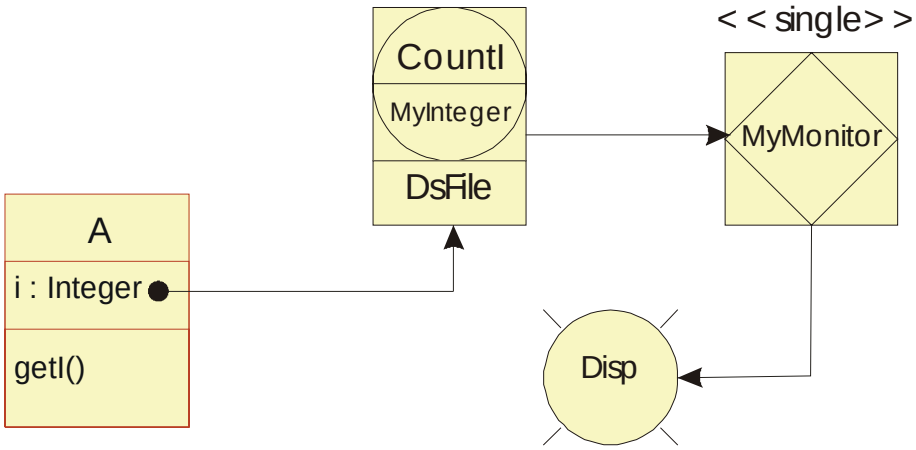


Ilustración 5-29. Desarrollo de toma de muestra con monitorización y alarma

Ahora la declaración de la clase sería:

```
class A {
   //# metrics begin:        anonymous
   //# metric class:         CountI     ## Clase RTMetric
   //# measure class:        MyInteger  ## Clase RTMMeasure
   //# element code:         { return MyInteger(i); }
   //# monitor class single: MyMonitor  ## Clase RTMMonitor
   //# display class:        Disp       ## Monitoriza alarma
   //# display class:        DsFile     ## Fichero de log File.log
```

```
   //# metrics end:

   private int i;
   public int getI() {return i;}
}
```

Ahora aparecen en el diagrama las clases `MyMonitor` y en la declaración aparece también `Disp` y `DsFile`, que serán las clases que estarán encargadas de la monitorización de la medida.

La clase `MyMonitor` aparece como single, lo que quiere decir que estará implementada como un *singleton* y será un solo objeto para todas las monitorizaciones que la usen. En el diagrama aparece como un stereotipo `<<single>>`.

5.8.4.3. CLASE QUE SE MONITORIZA PARA CONTROLAR EVENTOS EN OTRA CLASE

En este caso se mide y se monitoriza una clase para poder hacer control de eventos en otra clase diferente. Se medirá el atributo `i` de la clase `A` y se controlará que el atributo `j` de la clase `B` sea la mitad de `i`.

En este caso también se especifica que el monitor implementará un patrón singleton.

El diagrama de clases y control para este caso es como sige:

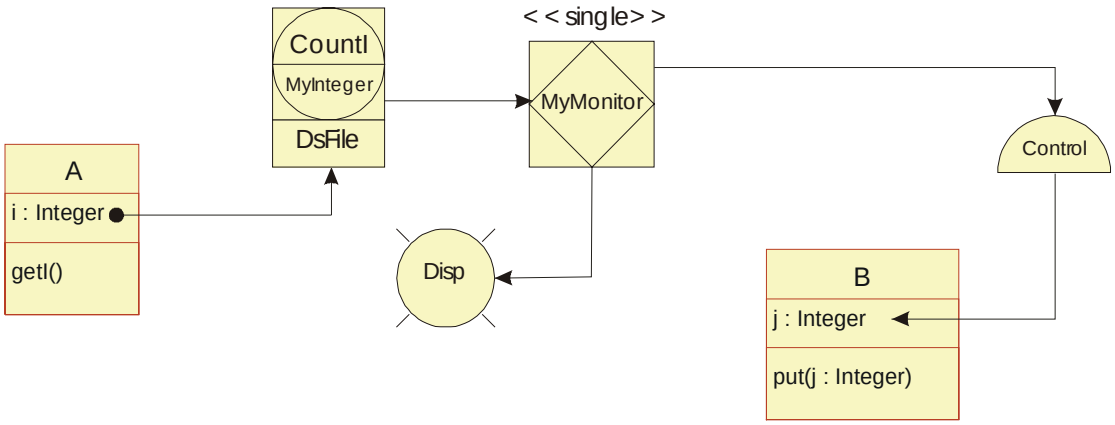


Ilustración 5-30. Ejemplo de toma de muestra con monitorización, alarma y actuador

Para este caso hay dos clases implicadas y la declaración sería:

```
class A {
   //# metrics begin:        A0001
   //# metric class:         CountI     ## Clase RTMetric
   //# measure class:        MyInteger  ## Clase RTMMeasure
   //# element code:         { return MyInteger(i); }
   //# monitor class single: MyMonitor  ## Clase RTMMonitor
   //# display class:        Disp       ## Monitoriza alarma
   //# display class:        DsFile     ## Fichero de log File.log
   //# actioner class:       Control    ## clase actioner
   //# controlled class:     B          ## clase controlada
   //# metrics end:          A0001

   private int i;
   public int getI() {return i;}
}
```

La clase `B` tendría el siguiente código:

```
class B {
   //# metrics begin:   A001
   //# element class:   A       ## clase element
   //# actioner class:  Control ## clase que realiza el control
   //# actioned code:   {put(((Integer)param).value/2);}
   //# metrics end:     A001

   private int i;
   public int getI() {return i;}
}
```

Hay dos elementos a tener en cuenta en este momento, el código de `//# actioner code`, será implementado por el propio objeto, dejando código más general a ser implementado por la clase Control.

Por otro lado, `param`, es el parámetro que recibirá el método encargado de realizar la acción, este nombre será palabra reservada y será del tipo más genérico que soporte el lenguaje (en Java será de la clase `Object`).

5.8.5. LENGUAJE

Así pues la proposición de ampliación del lenguaje Java se hará en base a directivas que comenzarán con `//#` y serán seguidas de declaraciones específicas.

En los siguientes párrafos se especifican las directivas. Para comprender completamente estas directivas es necesario entender el framework definido en 5.5, ya que el framework básico del sistema de métricas definido para el lenguaje es el mismo que el definido para Java.

5.8.5.1. //# METRICS BEGIN

Identifica un punto de medida, todo lo comprendido entre esta directiva y la `//# metrics end` es referente al mismo punto de control.

Cada clase puede definir tantos puntos de control como desee, siempre al principio de la clase, antes de comenzar la declaración específica de atributos y métodos.

La sintaxis es:

```
//# metrics begin ':' ( <identificador> | anonymous )
```

Si se especifica un identificador, este debe ser único y coincidirá con otros en otras clases que sean parte de la misma medida o del mismo `actioner`.

Si se especifica `anonynous` es que lo único que se desea es realizar la medida sin ninguna relación con otras medidas o con `actioners`.

5.8.5.2. //# METRICS END

Identifica el punto final de una declaración completa de un punto de medida o de un punto de declaración de `actioner`.

La sintaxis es:

```
//# metrics end ':' ( <identificador> | anonymous | <nada> )
```

Si se especifica identificador o `anonymous`, este debe coincidir con el de la directiva correspondiente `//# metrics begin`.

5.8.5.3. //# METRIC CLASS

Designa la clase que implementa la métrica, esta clase puede ser un estándar o un diseño ad-hoc para el sistema que se esté implementando.

Clases estándar son: `StRTMCounter`, `StRTMPercentage` y `StRTMConditional`.

El comando completo tiene la siguiente sintaxis:

```
//# metric class [single] ':' <nombre_de_la_clase>
```

Si se especifica `single` se entiende que la clase creará un único objeto de esta clase y por tanto se implementará mediante un patrón *singleton*.

El `<nombre_de_la_clase>` será el nombre de la clase que implementará `RTMetric`.

5.8.5.4. //# MEASURE CLASS

Designa la clase que implementa el resultado de la medida, esta clase puede ser un estándard o un diseño ad-hoc para el sistema que se esté implementando.

Clases estándar son: `StRTMInteger`, `StRTMDouble` y `StRTMBoolean`.

El comando completo tiene la siguiente sintaxis:

```
//# measure class [single] ':' <nombre_de_la_clase>
```

Si se especifica `single` se entiende que la clase creará un único objeto de esta clase y por tanto se implementará mediante un patrón *singleton*.

El `<nombre_de_la_clase>` será el nombre de la clase que implementará `RTMMeasure`.

5.8.5.5. //# MONITOR CLASS

Designa la clase que implementa el monitor o controlador del sistema, esta clase puede ser un estándard o un diseño ad-hoc para el sistema que se esté implementando.

Clase estándar es: `StRTMMonitor`, cuya unica función es registrar todos los elementos, ponerlos en comunicación y pasar el testigo entre ellos para que cooperen.

El comando completo tiene la siguiente sintaxis:

```
//# monitor class [single] ':' <nombre_de_la_clase>
```

Si se especifica `single` se entiende que la clase creará un único objeto de esta clase y por tanto se implementará mediante un patrón *singleton*. Es muy común que el monitor sea único por cada aplicación o al menos por cada subsistema de la aplicación,

por tanto la especificación de `single` será muy habitual cuando se refiera al monitor.

El `<nombre_de_la_clase>` será el nombre de la clase que implementará `RTMMonitor`.

5.8.5.6. //# DISPLAY CLASS

Designa la clase que implementa la vista de la medida, esta clase puede ser un estándard o un diseño ad-hoc para el sistema que se esté implementando. El sistema de métricas completo debe seguir el modelo MVC descrito en "Pattern-oriented software architecture" [BUS96].

Así como en las métricas, los estándares implementan la mayor parte de las métricas que se definirán, si embargo las vistas van a ser muy dependientes del sistema y por tanto lo habitual será implementarlas para cada sistema.

El comando completo tiene la siguiente sintaxis:

```
//# display class [single] ':' <nombre_de_la_clase>
```

Si se especifica `single` se entiende que la clase creará un único objeto de esta clase y por tanto se implementará mediante un patrón *singleton*. Es necesario un el diseño ad-hoc de esta clase para visualizar cada medida, implementado con unidades de medida y referencias al punto de medida por tanto puede ser un solo objeto por cada clase `RTMetric` para cada punto de medida.

El `<nombre_de_la_clase>` será el nombre de la clase que implementará `RTMDispDvc`.

5.8.5.7. //# ACTIONER CLASS

Designa la clase que implementa las acciones a realizar en el sistema en función de una o varias medidas, esta clase puede ser un estándar o más normalmente un diseño ad-hoc para el sistema que se esté implementando.

La única clase estándar es: `StRTMGeneric`, que simplemente reenvía la acción al elemento controlado convirtiendo el parámetro que se le pasa en una referencia a la clase más genérica del lenguaje, en Java la clase `Object`.

El comando completo tiene la siguiente sintaxis:

```
//# actioner class [single] ':' <nombre_de_la_clase>
```

Si se especifica `single` se entiende que la clase creará un único objeto de esta clase y por tanto se implementará mediante un patrón *singleton*.

El `<nombre_de_la_clase>` será el nombre de la clase que implementará `RTMActioner`.

5.8.5.8. //# CONTROLLED CLASS

Designa la clase que implementa la clase controlada por el sistema, esta clase sólo puede ser un diseño ad-hoc para el sistema que se esté implementando.

El comando completo tiene la siguiente sintaxis:

```
//# controlled class ':' <nombre_de_la_clase>
```

El `<nombre_de_la_clase>` será el nombre de la clase que se desea controlar con la medida que se está tomando. Si fueran varias clases se pondrá varias veces esta directiva.

Esta directiva sólo debe ser utilizada en la parte de la clase donde se realiza la medida, esto es en la clase `element`.

5.8.5.9. //# ELEMENT CLASS

Designa la clase sobre la que se hace la medida, esta clase sólo puede ser un diseño ad-hoc para el sistema que se esté implementando.

El comando completo tiene la siguiente sintaxis:

```
//# element class ':' <nombre_de_la_clase>
```

El `<nombre_de_la_clase>` será el nombre de la clase sobre la que se van a realizar las medidas que darán lugar a las acciones, si fueran varias clases se pondrá varias veces esta directiva.

Esta directiva sólo debe ser utilizada en la parte de la clase donde se realiza la acción, esto es en la clase `actioned`.

5.8.5.10. //# ACTIONED CODE

Designa el código que se implementara en la clase sobre la que se aplica acción después de realizadas las muestras.

El comando completo tiene la siguiente sintaxis:

```
//# actioned code ':' '{' <codigo> '}'
```

El `<codigo>` es código escrito en java asumiendo que toma como parámetro uno de nombre `param` y que es del tipo más genérico del lenguaje, en el caso de java será un `Object`.

Esta directiva sólo debe ser utilizada en la parte de la clase donde se realiza la acción, esto es en la clase `actioned`.

5.8.5.11. //# ELEMENT CODE

Designa el código que se implementara en la clase sobre la que se aplica la medida cuando sea requerida una toma de muestra.

El comando completo tiene la siguiente sintaxis:

```
//# element code ':' '{' <codigo> '}'
```

El `<codigo>` es código escrito en java asumiendo que debe retornar por todos los caminos un objeto de la clase definida en `//# measure class`.

Esta directiva sólo debe ser utilizada en la parte de la clase donde se realiza la medida, esto es en la clase `element`.

5.8.5.12. //#>

El contenido de esta línea es código que ha comenzado en una etiqueta anterior.

5.9. RESULTADOS TEÓRICOS DEL USO DE MÉTRICAS EN TIEMPO DE EJECUCIÓN

Con el sistema diseñado, se puede plantear que un sistema de métricas y control de aplicaciones OO tiene una estructura similar a la de cualquier otro sistema de control convencional, en donde se plantea una función de transferencia, un punto de lectura de la señal y una realimentación cuando hablamos de un lazo cerrado o sin realimentación si se trata de lazo abierto.

En el caso más general:

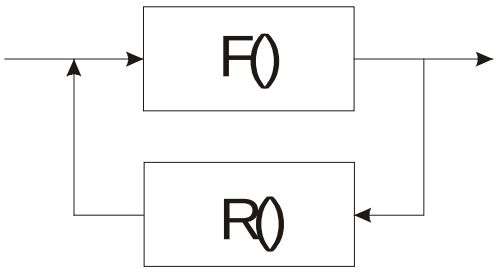


Ilustración 5-31. Función de transferencia y función de realimentación

Donde F() es la función de transferencia del sistema y R() es la función de realimentación.

La función de transferencia F() es difícil de conocer en los casos reales, ya que hay que tener en cuenta que los lenguajes OO son casi siempre imperativos lo que hace que tengan muchos efectos laterales.

La función R() será implementada por el usuario y será normalmente imperativa y consistirá en la adaptación del sistema a las modificaciones que se prevean en tiempo de diseño.

Tipos de funciones:

Atendiendo al tipo de funciones, tanto F() como R() pueden ser:

- Matemáticas
  - Lógicas
    - Predicados de 1er orden
    - Predicados de 2^{o} orden

        - Fuzzy
        - Algoritmos genéticos
        - Redes Neuronales
    - Funcionales
    - Imperativas sin efectos laterales desde el punto de vista de F()
    - Imperativas con efectos laterales
    - Intrínsecamente no controlables

Entendiendo por imperativos los lenguajes orientados a objetos y otros lenguajes convencionales.

Los modelos son más fáciles de encontrar en las implementaciones puramente matemáticas y más difíciles en las implementaciones imperativas con efectos laterales, por lo que las soluciones a R() serán diferentes en cada caso y no necesariamente matemáticas.

5.9.1. FUNCIONES MATEMÁTICAS

Se incluye en las funciones matemáticas a todas aquellas que carecen de efectos laterales y no sólo a las que simplemente hacen un cálculo numérico. El denominador común de este tipo de funciones es que aceptan unos parámetros y dan una salida idéntica para los mismos parámetros independiente de la historia de la aplicación.

En estos casos es posible definir F() en términos puramente numéricos o como un conjunto más o menos complejo de reglas, siendo aplicable en estos casos teorías matemáticas conocidas dentro del calculo numérico (usando aquí el término numérico en un sentido amplio) o como sistemas de inteligencia artificial (reglas, teoría de grafos, etc.).

Cuando F() es una función de este tipo es posible demostrar en tiempo de diseño el comportamiento deseado de la función siguiendo teorías matemáticas.

Debido a que este comportamiento es predecible y demostrable, es también más fácil encontrar una función R() que permita el control de los sucesos y que será normalmente otra función de este tipo (matemático).

5.9.2. FUNCIONALES

Aunque las F() funcionales pudieran considerarse dentro de las matemáticas, se les ha hecho una mención aparte para soslayar su contenido más puramente en el campo de la informática.

En este caso también es fácil predecir el comportamiento de la función en tiempo de diseño al carecer de efectos laterales, pero casi todos los lenguajes funcionales permiten llamadas a métodos no funcionales para poder conseguir algún efecto lateral, como por ejemplo emisión de interfaces para el usuario, escritura en ficheros, etc.

Luego los lenguajes OO con métodos de contenido funcional puro estarían en el caso de las matemáticas, pero no así todas aquellas llamadas no funcionales que provocarían que las F() funcionales pasasen a estar en alguno de los casos que vienen a continuación.

5.9.3. IMPERATIVAS SIN EFECTOS LATERALES DESDE EL PUNTO DE VISTA DE F()

En este caso estarían todas aquellas funciones y métodos en los que su parte demostrable es la que interesa desde el punto de vista de la medición y el control, pero que su contenido completo contiene efectos laterales.

En este tipo estarían incluidas las pertenecientes a lenguajes funcionales que permiten efectos laterales y las de lenguajes imperativos que permitan implementar procedimientos puramente funcionales.

Si se demuestra la independencia del algoritmo funcional de cualquier efecto lateral, estaríamos en el caso de algunas funciones matemáticas vista en apartados anteriores, pero si esta independencia no es demostrable estaríamos en el caso siguiente.

5.9.4. IMPERATIVAS CON EFECTOS LATERALES

Éste sería el caso más general de funciones, en este caso y en general el comportamiento no es demostrable matemáticamente, sino que puede tener que llegar a ser especificado mediante algoritmos o mediante lenguaje natural.

Este tipo de funciones F() no admiten demostración y por tanto la única forma de definirlas (mediante algoritmos p.e.) hace que sean difíciles de tratar y fuerzan a que R() sea también y en general una función del mismo tipo.

Cuando el código de la aplicación a controlar es conocido, el uso de estas funciones puede requerir de la modificación de estos algoritmos para que puedan ser controlados, cuando no son conocidas y se pretende su control, puede ocurrir incluso que ésta sea imposible.

Por ejemplo, si se pretende controlar el número de objetos que creará uno determinado cuyo algoritmo ha sido implementado de manera que la llamada a un solo método del objeto creador dispara en secuencia que cada contructor de los objetos creados cree, a su vez, como efecto lateral, el objeto siguiente de acuerdo a algún algoritmo determinado.

En este caso el control del método del objeto creador solo puede servir para que no se dispare una segunda secuencia, pero no se puede evitar que la primera se cree con un número no controlado de objetos.

5.9.5. INTRÍNSECAMENTE NO CONTROLABLES

En general, cuando un proceso se quiere controlar, tanto la causa (lo que se controla) como el efecto (lo que se mide), deben estar en ejecución simultáneamente. De este modo, así como las mediciones indican una desviación de los objetivos, los actuadores pueden estar corrigiendo el comportamiento de la causa para dirigir el sistema.

En el caso del software ocurre exactamente lo mismo, en general debemos tener concurrencia (real o simulada) de la causa y el efecto para que el control tenga sentido, en caso contrario (procesos en batch por ejemplo), es imposible operar sobre el proceso *causa*, puesto que cuando se procesa su salida, dicho proceso ya no está en ejecución, por tanto o bien se reprocesa (se filtra) su salida para hacerla controlable en el momento en que se está ejecutando el proceso *efecto*, o bien es imposible el control.

Pero en el caso del software se puede ir aún más lejos, ya que muchos procesos concurrentes tienen un cierto grado de proceso batch oculto, y en este caso, aun siendo (falsamente) concurrentes ambos procesos, no es posible el control.

6. CASOS DE ESTUDIO

6.1. INTRODUCCIÓN

En este apartado se hará una descripción de un caso de estudio solucionado tecnológicamente de diferentes formas, todas ellas dentro de las posibilidades del framework.

Los principales objetivos de esta parte son, de un lado presentar la viabilidad práctica del sistema definido en la tesis y de otro explorar las diferentes soluciones tecnológicas aplicables dentro del framework, tanto en utilización directa como mediante el lenguaje JavaRTM.

6.2. ENUNCIADO DEL PROBLEMA

EL sistema a analizar es un sistema productor-consumidor de mensajes. El productor crea mensajes de texto que envía a través de canal de comunicaciones y el receptor los consume.

El sistema será simulado y en esta simulación se han establecido dos restricciones:

- Si el mensaje es demasiado largo existe una cierta posibilidad de que se pierda porque el canal de comunicaciones tendría que realizar conexiones muy largas.
- Si el mensaje es demasiado corto también puede perderse debido a la baja prioridad que el canal establece para este tipo de mensajes.

Luego el sistema debe autoadaptarse a las condiciones de la línea y buscar el tamaño adecuado de mensaje para optimizar el rango de mensajes entregados con éxito.

Se comenzará presentando la solución sin utilizar el framework y después se presentarán diferentes soluciones con el framework.

6.3. CASO DE ESTUDIO 1

El caso de estudio primero presenta el problema planteado sin intervención de ningún sistema de métricas y control.

En este caso la implementación simplemente generará mensajes a enviar de un tamaño especificado en el código y los enviará por el canal de comunicaciones informando al usuario de si los mensajes han sido recibidos o no.

Cambiando el tamaño del mensaje se puede apreciar la mejor o peor respuesta del sistema dependiendo del tamaño de dicho mensaje comprobando cual es el tamaño ideal en función de las condiciones programadas en el canal de envío de mensajes.

Se hará un envío de 200 mensajes por cada ejecución.

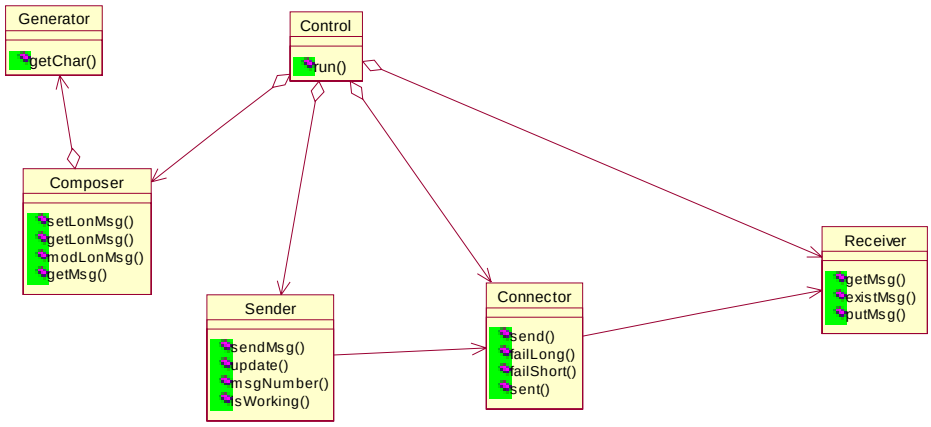


Ilustración 6-1. Diagrama de clases (Caso 1)

En el diagrama de clases se puede ver el diseño básico de la aplicación, un objeto de la clase `Control` es el encargado de la línea principal del programa.

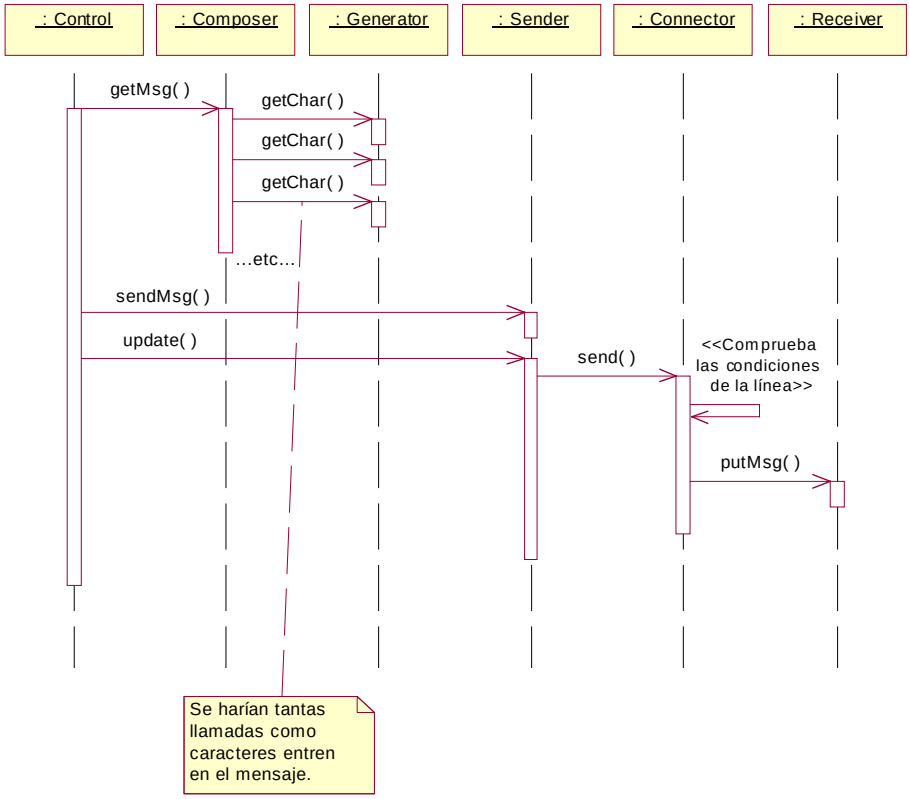


Ilustración 6-2. Diagrama de secuencias

Este objeto pedirá mensajes a un objeto de la clase `Composer` que a su vez hará que `Generator` genere caracteres (en este caso aleatoriamente) para el mensaje de texto a componer. Cada mensaje devuelto por el objeto `Composer` se envía al un objeto de la clase `Sender` para que lo envíe al `Receiver` a través de un `Conector`. En

el gráfico Ilustración 6-2 se puede ver un escenario de ejecución del envío de un mensaje.

Las condiciones de evaluación del canal se han hecho mediante porcentajes en función del tamaño del mensaje y números aleatorios:

```
// Long
longM = false;
double aux = msg.length / 10;
double pos = per + r.nextInt((int)aux);
if (pos >= 100)
   longM = true;

// Short
shortM = false;
aux = 1000 / msg.length;
pos = per + r.nextInt((int)aux);
if (pos >= 100)
   shortM = true;
```

Cambiando los valores de `per` se hace más o menos estrecho el valor del tamaño de mensaje que cruza el canal sin errores. Por ejemplo para un valor de 90 en `per` y para mensajes entre 25 y 200 caracteres, el resultado de la ejecución se muestra en el siguiente gráfico.

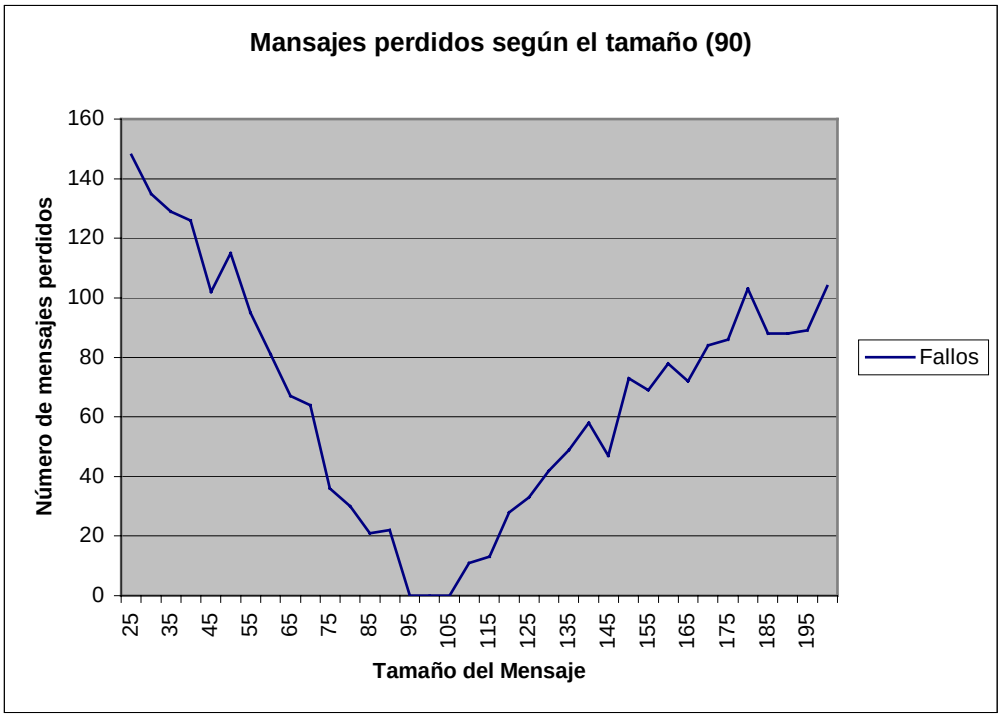


Ilustración 6-3. Gráfico de mensajes perdidos (90)

Como se puede apreciar, tan solo un pequeño hueco entre los 95 y los 105 caracteres por mensaje no ha perdido ningún mensaje en la ejecución.

En las condiciones actuales de la simulación este canal se mantiene en diferentes ejecuciones, pero en la realidad, un canal con este tipo de problemas podría ir variando ese estrecho canal en el tiempo y se necesitaría que la aplicación de envío de mensajes se autoadaptase a las condiciones del canal y estuviera buscando constantemente el hueco en el cual se pueden enviar los mensajes.

Entre las diferentes pruebas realizadas, cuando el valor `per` se va entorno a 85, el hueco se amplía considerablemente como se puede apreciar en el siguiente gráfico.

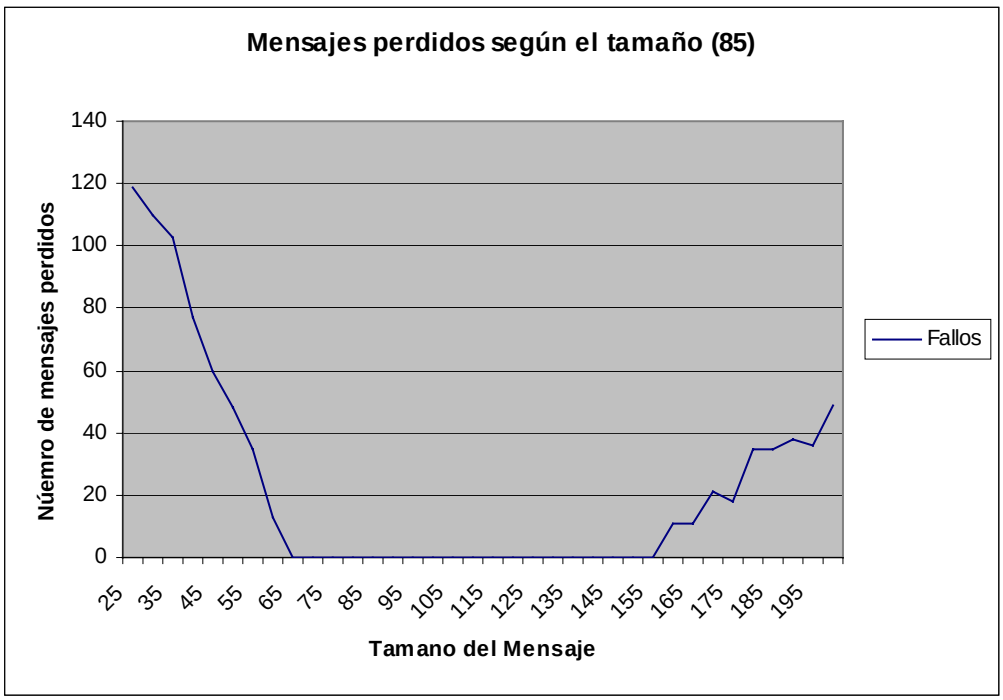


Ilustración 6-4. Gráfico de mensajes perdidos (85)

Sin embargo, cuando el valor de `per` sube hasta 95, no es posible encontrar ningún buen hueco y el número de mensajes perdidos permanece prácticamente constante.

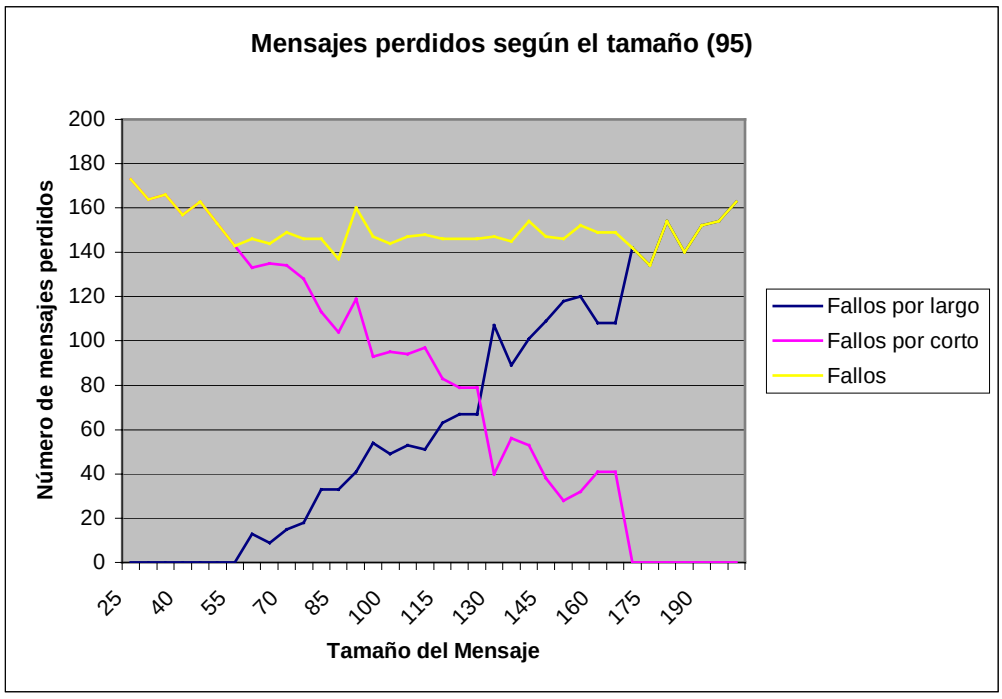


Ilustración 6-5. Gráfico de mensajes perdidos (95)

A la vista de las gráficas anteriores, y con el fin de demostrar la capacidad de adaptación del sistema así diseñado, se ha elegido un valor de `per` de 90 para el resto de los casos de estudio.

Este valor permite que haya algún hueco estable que será el que deberá buscar la aplicación y también hace que el hueco sea lo suficientemente estrecho como para que no sea fácil de encontrar.

6.4. CASO DE ESTUDIO 2

En este caso de estudio se va a plantear la primera solución al sistema de monitorización, medición y control del sistema general planteado en el caso anterior.

Para este primer acercamiento se utilizará un sistema de control nuevo que sea capaz de correr no solo el sistema actual, sino también todos los objetos necesarios para el sistema de monitorización y control.

Este planteamiento, que será utilizado también en algún otro caso de estudio, aunque no es la solución transparente que se puede conseguir con el framework diseñado, es un acercamiento importante y de fácil implementación.

En los casos en que la monitorización y el control sean partes integrantes del sistema definitivo y no simples medios de prueba, se pueden utilizar este tipo de enfoques por su simplicidad.

6.4.1. PLANTEAMIENTO GENERAL DE LA SOLUCIÓN

Para solucionar el problema del control y la supervisión se ha seguido la metodología DMT, definida en los capítulos anteriores y cuyas fases son:

- Abstracción del sistema sin métricas
- Análisis de las métricas a utilizar
- Diseño de los puntos de toma de muestra
- Diseño de los puntos de control
- Diseño de la monitorización
- Paso del sistema a las clases del framework
- Implementación

6.4.1.1. ABSTRACCIÓN DEL SISTEMA SIN MÉTRICAS

Para este paso se va a utilizar una versión reducida del Proceso Unificado y se documentarán (se escribirán) las soluciones utilizando el Lenguaje de Modelado Unificado (UML) como sistema estándar y de mayor utilización.

Una vez realizados los pasos de análisis y de diseño de la aplicación se obtienen (como se ha visto en párrafos anteriores) los diagramas de clases, de secuencia e iteración de objetos, etc. necesarios para la descripción del sistema a diseñar.

Este paso se ha realizado en los párrafos anteriores y de él se ha obtenido en siguiente diagrama de clases:

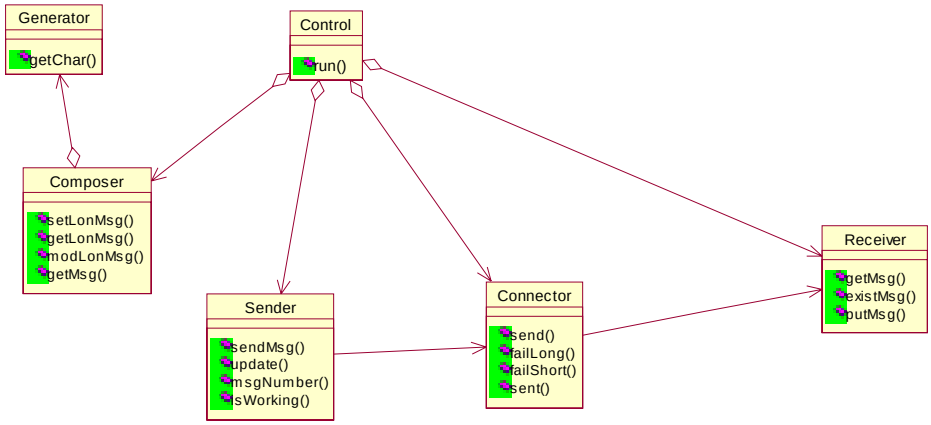


Ilustración 6-6. Diagrama de clases inicial (Caso 2)

Este diagrama es la base sobre la que se implementará el sistema de métricas y control.

6.4.1.2. ANÁLISIS DE LAS MÉTRICAS A UTILIZAR

El objetivo a controlar es el tamaño del mensaje. Cada mensaje está formado por una cadena de caracteres de una longitud entera, luego parece lógico utilizar una métrica entera.

En cualquier caso, estudiando un poco más profundamente el evento que se desea medir, lo que en realidad se busca es obtener algún valor que indique si el mensaje ha fallado por ser demasiado largo o demasiado corto o bien el mensaje ha sido enviado con éxito.

Desde este punto de vista, lo que se desea es controlar tres estados:

1. Mensaje no entregado por corto (valor de la métrica = -1)
2. Mensaje no entregado por largo (valor de la métrica = 1)
3. Mensaje entregado correctamente (valor de la métrica = 0)

En este caso se ha aplicado el principio de máxima sencillez y menor coste de proceso, si se hubiera medido el tamaño del mensaje y se hubiera intentado establecer si el fallo era por largo o por corto, posiblemente se hubiera necesitado procesar mayor información con lo que hubiera intervenido mucho más en el sistema.

Dentro del propio framework existe una métrica que puede servir para este proceso: `es.uniovi.ootlab.metrics.metrics.StRTMInteger`. En realidad esta

métrica sirve para medir cualquier variable entera dentro del sistema, será usada, por tanto, para codificar los valores de la respuesta del sistema a la entrega de mensajes.

En el proceso se ha reducido el tipo de escala de la métrica elegida, se trata de una escala de *ratio* pero en la que sólo se consideran tres valores enteros como válidos para definir las tres situaciones posibles conocidas.

6.4.1.3. DISEÑO DE LOS PUNTOS DE TOMA DE MUESTRA

En el punto anterior se ha definido qué medir, ahora hay que definir donde medir.

Para ello hay que recurrir al análisis original y buscar qué objetos tienen la responsabilidad de saber si el mensaje se ha entregado o no con éxito.

El objeto de la clase `Receiver` es el que tiene la responsabilidad de consumir el mensaje, sin embargo si no recibe el mensaje no tendrá siquiera confirmación de su existencia, luego parece lógico que no sea éste el punto de control.

El objeto de la clase `Sender` envía los mensajes al canal de comunicaciones, y obtiene respuesta de la entrega del mensaje interrogando al objeto de la clase `Connector` sobre la entrega del mensaje actual.

De hecho no introducirá un nuevo mensaje en el canal hasta que no reciba confirmación de que el canal está vacío de mensajes.

En cualquier caso, y tal y como ha sido concebida la solución base, habría que modificar mucho código para colocar el punto de control en este objeto, ya que habría que aumentar el número de consultas para saber no solo si el canal está vacío, sino también si ha fallado el mensaje y si ha sido por un mensaje corto o largo. Por el principio de sencillez y de mínima intervención en la solución adoptada tampoco éste parece el punto de toma de muestra adecuado.

Así pues el objeto de la clase `Connector` es el indicado para esta toma de muestra, ya que él recibirá los mensajes y tiene conocimiento en cada momento del destino del mensaje.

En la concepción básica es el objeto de la clase `Sender` el encargado de hacer los reintentos de envío cuando el mensaje falla, sin embargo el punto de toma de muestra está en el objeto de la clase `Connector`, lo que puede provocar reducciones del mensaje en función de los reintentos y no de los fallos.

De cualquier manera en los experimentos realizados, el comportamiento del sistema ha sido el esperado y no se ha complicado más el sistema con una nueva métrica que controle el número de reintentos, que sería la solución lógica en ese caso.

6.4.1.4. DISEÑO DE LOS PUNTOS DE CONTROL

El objeto sobre el que se debe operar el control es el que cree los mensajes y asigne su tamaño. Éste es el objeto de la clase `Composer`, que tiene dentro de su interfaz

dos operaciones para realizar la comprobación y el cambio del tamaño: `getLonMsg()` y `setLonMsg()`.

6.4.1.5. DISEÑO DE LA MONITORIZACIÓN

Por tanto el `Composer` será quien deba implementar la interfaz `RTMCtrlElm`. Y de este modo, el diagrama de control quedará como se puede ver en el siguiente diagrama.

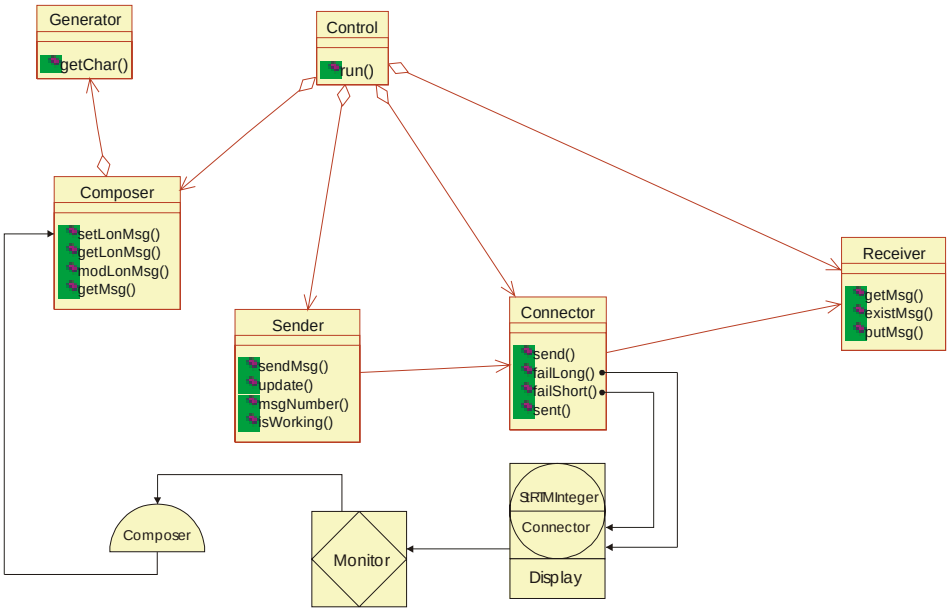


Ilustración 6-7. Diagrama de clases DMT (Caso 2)

Del gráfico anterior hay que hacer unas aclaraciones. En la parte de métricas y control se han colocado en el gráfico dos referencias (Composer y Connector) que en realidad no usarán ese nombre, sino que irán cambiando con cada caso de estudio. En el próximo apartado se colocarán los nombres reales.

6.4.1.6. PASO DEL SISTEMA A LAS CLASES DEL FRAMEWORK

El siguiente paso de la metodología es el paso a las clases del framework. En este paso se convierten los elementos gráficos en clases que implementarán el sistema.

En el diagrama de clases se ve este paso con referencia a todas las interfaces del framework.

A partir de las clases a medir y a controlar (`Connector` y `Composer`) se ha heredado en dos clases más específicas `ConnectorElement` y `ComposerCtrlElm` que implementan la conexión al framework para realizar las mediciones y el control.

En este caso no se ha heredado de las clases abstractas intermedias porque es necesario heredar directamente de las clases a medir y controlar y existe el objetivo de soportar solamente herencia simple. Las clases abstractas intermedias definidas como parte del framework sólo se utilizarían en el caso de haber partido directamente de clases controlables y/o medibles en el diseño original.

En este caso ha sido sustituida la clase `Control` por una nueva clase en este `package` para poder arrancar los objetos del control y la monitorización, ya que en la clase original no se contemplaba este arranque.

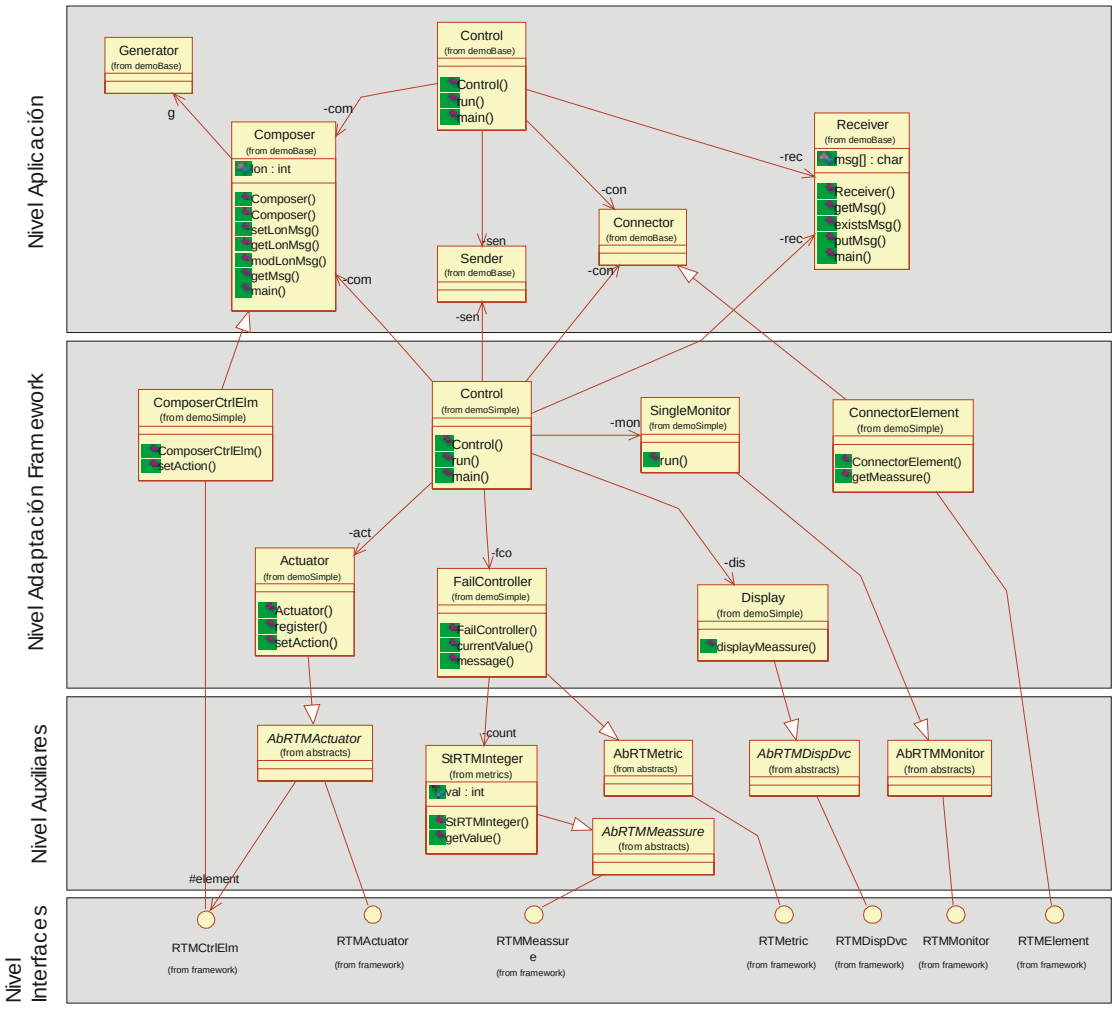


Ilustración 6-8. Diagrama de clases final (Caso 2)

El resto de las clases son especializaciones de las clases abstractas definidas en el nivel de clases abstractas del framework.

Los dos niveles inferiores (auxiliares e interfaces) son los propios del framework y son la base de soporte a la monitorización y el control.

Uno de los objetivos importantes que debe conseguirse en este nivel es que el sistema sería capaz de funcionar sin la monitorización y el control, esto se consigue recuperando directamente las clases del Nivel de Aplicación.

Todo el sistema de monitorización, medición y control está formado por los niveles de adaptación del framework, de auxiliares y de interfaces.

6.4.1.7. IMPLEMENTACIÓN

A partir del diseño anterior la implementación en Java es directa, aprovechando las clases del framework más el diseño del sistema original (demoBase).

En el CD que acompaña a este trabajo se puede ver la implementación completa.

6.4.2. RESULTADOS

Los resultados de la ejecución del control y la medición sobre el ejemplo son similares e independientes en todos los casos del sistema utilizado para el diseño y la implementación de la solución.

Tal y como se predijo en el capítulo anterior, el comportamiento del sistema tiene claras similitudes con los sistemas tradicionales de monitorización y control industriales. Variando el parámetro que ajusta el porcentaje de modificación del tamaño del mensaje se pueden obtener sistemas subamortiguados, sobreamortiguados u oscilantes. En algunos casos también se obtuvieron sistemas inestables.

En los siguientes gráficos se puede ver el resultado de la ejecución para diferentes valores de dicho porcentaje. Todos los sistemas han sido probados con un valor de `per` de 90 (ver las condiciones del Caso de Estudio 1, aptdo. 6.3)

6.4.2.1. PORCENTAJE DE MODIFICACIÓN DEL 20%

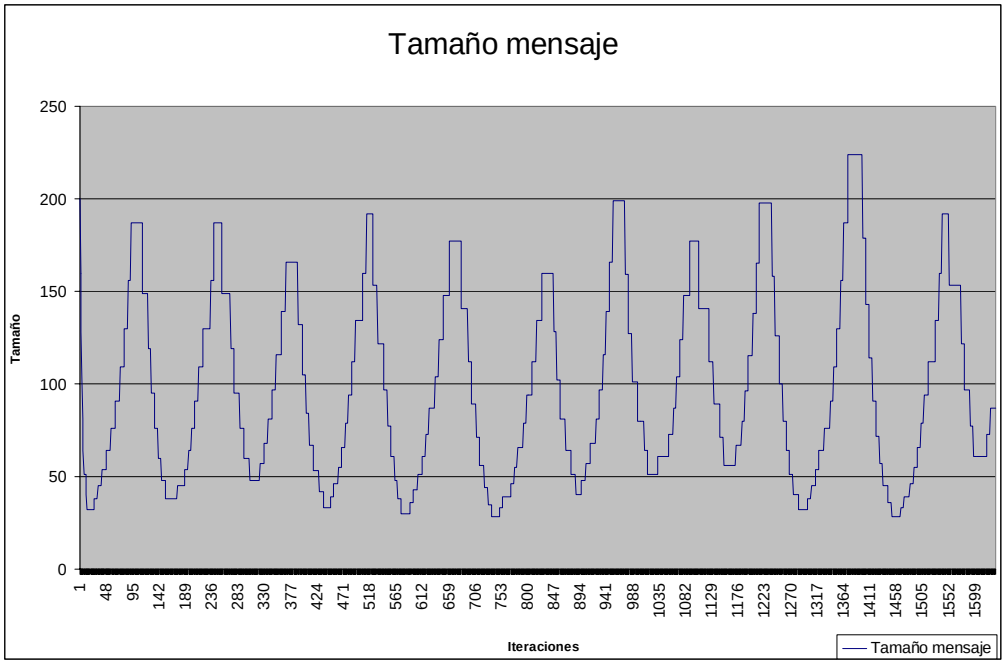


Ilustración 6-9. Sistema oscilante

Para este porcentaje de modificación y en las condiciones del sistema actual, éste resulta en un sistema oscilante como se puede apreciar en el gráfico (Ilustración 6-9. Sistema oscilante).

Sólo de manera casual se caería en el rango de valores aceptados para conseguir una ejecución estable, lo que en algunos ejemplos provocó que al tomar el tamaño del menaje, un valor muy próximo a 100, se quedara estable el resto de la ejecución, pero sólo de forma casual, ya que intrínsecamente el sistema es incapaz de controlar el tamaño del mensaje en estas condiciones.

En el gráfico se ha representado el tipo de ejecución más habitual en estas condiciones que es el sistema oscilante.

6.4.2.2. PORCENTAJE DE MODIFICACIÓN DEL 10%

En el caso de elección de un porcentaje de modificación del 10%, el sistema se comportó como sobreamortiguado.

Este comportamiento hace que se pase una o varias veces sobre el valor de referencia estable del sistema, cada vez con menos distancia de dicho valor de referencia hasta que por fin se consigue un valor estable que mantiene el sistema.

Al igual que en el caso anterior, en ejecuciones casuales se puede conseguir que el valor entre en un estado estable antes de haber pasado el valor de referencia incluso en la primera rampa de acercamiento, pero las ejecuciones habituales reproducen el modelo de sistema sobreamortiguado.

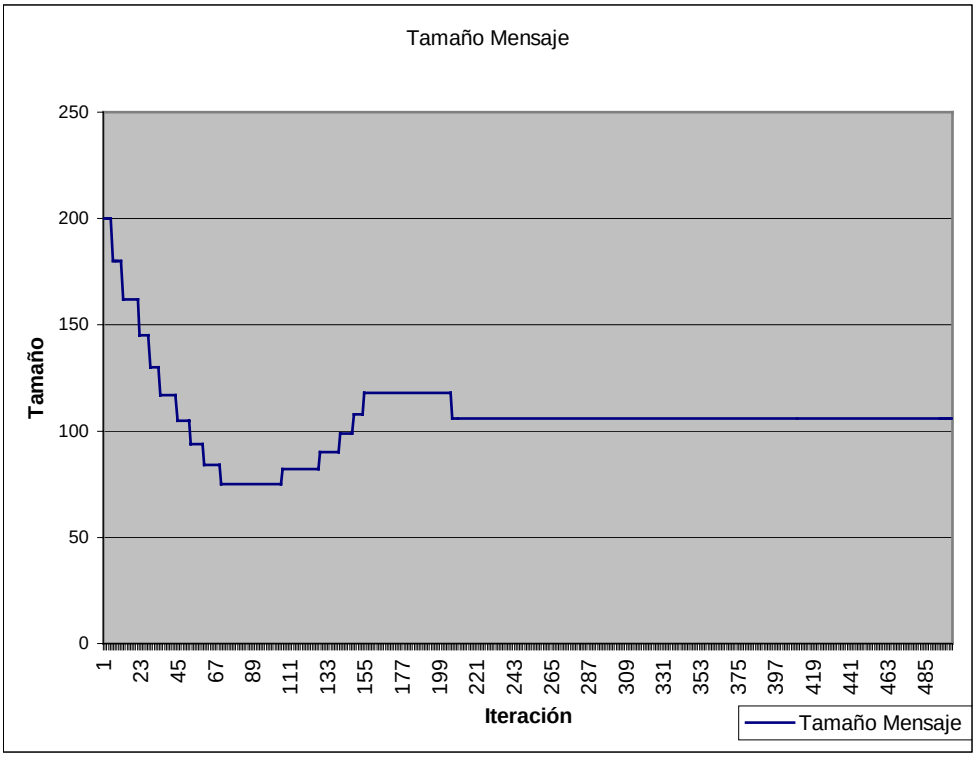


Ilustración 6-10. Sistema sobreamortiguado

En el gráfico (Ilustración 6-10. Sistema sobreamortiguado) se muestra una de las ejecuciones típicas del sistema descrito donde se puede ver que el valor del tamaño del mensaje sobrepasa dos veces el canal de valores estables, cada vez con menor distancia a este, hasta que al fin consigue un valor estable.

6.4.2.3. PORCENTAJE DE MODIFICACIÓN DEL 2%

Cuando el porcentaje de modificación del tamaño del mensaje se reduce se obtienen sistemas subamortiguados.

Al igual que en los sistemas de control de procesos industriales, estos sistemas suelen ser más lentos a la hora de obtener un valor estable, pero se tiene la ventaja de que siempre lo encuentran.

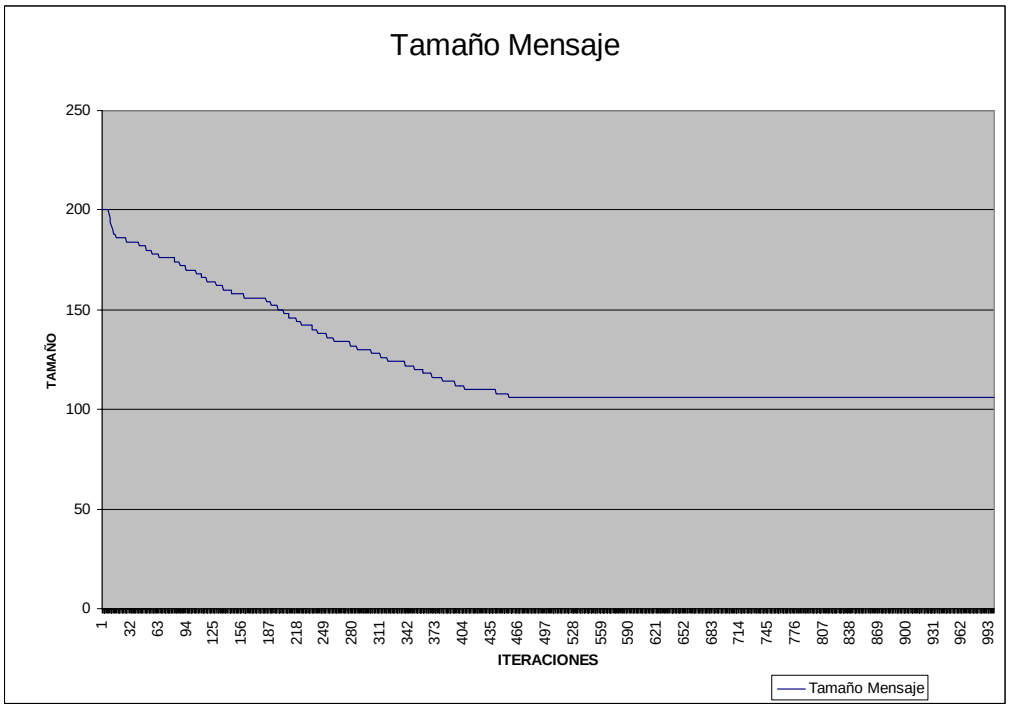


Ilustración 6-11. Sistema subamortiguado

En el gráfico anterior se puede ver un sistema subamortiguado generado por la ejecución de la aplicación.

Si las condiciones del canal de comunicaciones son perfectamente conocidas se pueden predecir valores del porcentaje de cambio del tamaño del mensaje que hagan que el sistema sea subamortiguado y que por tanto, siempre que exista, encuentre un valor estable.

6.5. CASO DE ESTUDIO 3

La segunda posible solución al sistema planteado puede basarse en la necesidad de llamar continuamente al sistema de control y monitorización.

Hay razones que pueden avalar la necesidad de llamar continuamente a dicho sistema o simplemente llamarlo cuando se producen ciertos eventos, por ejemplo en este caso cuando no se entrega un mensaje correctamente.

Si, como es el caso de estudio actual, la intención es simplemente realizar un sistema autoadaptativo sin importar la historia, entonces esta solución es mejor por tener un coste de computación claramente menor.

En este caso se ha decidido llamar al sistema de monitorización y control exclusivamente cuando se produce el evento del fallo en la entrega de un mensaje. Este evento es utilizado a su vez por `observedMonitor` para reenviarlo a todos los objetos que le observan.

Todos los pasos de la metodología DMT del caso de estudio anterior son válidos aquí a excepción de los dos últimos, esto es el paso a las clases del framework (que tiene variación en la relación entre estas clases) y la implementación que tiene, obviamente algunas variaciones.

6.5.1. PASO DEL SISTEMA A LAS CLASES DEL FRAMEWORK

El paso a clases del framework se puede observar en el diagrama siguiente.

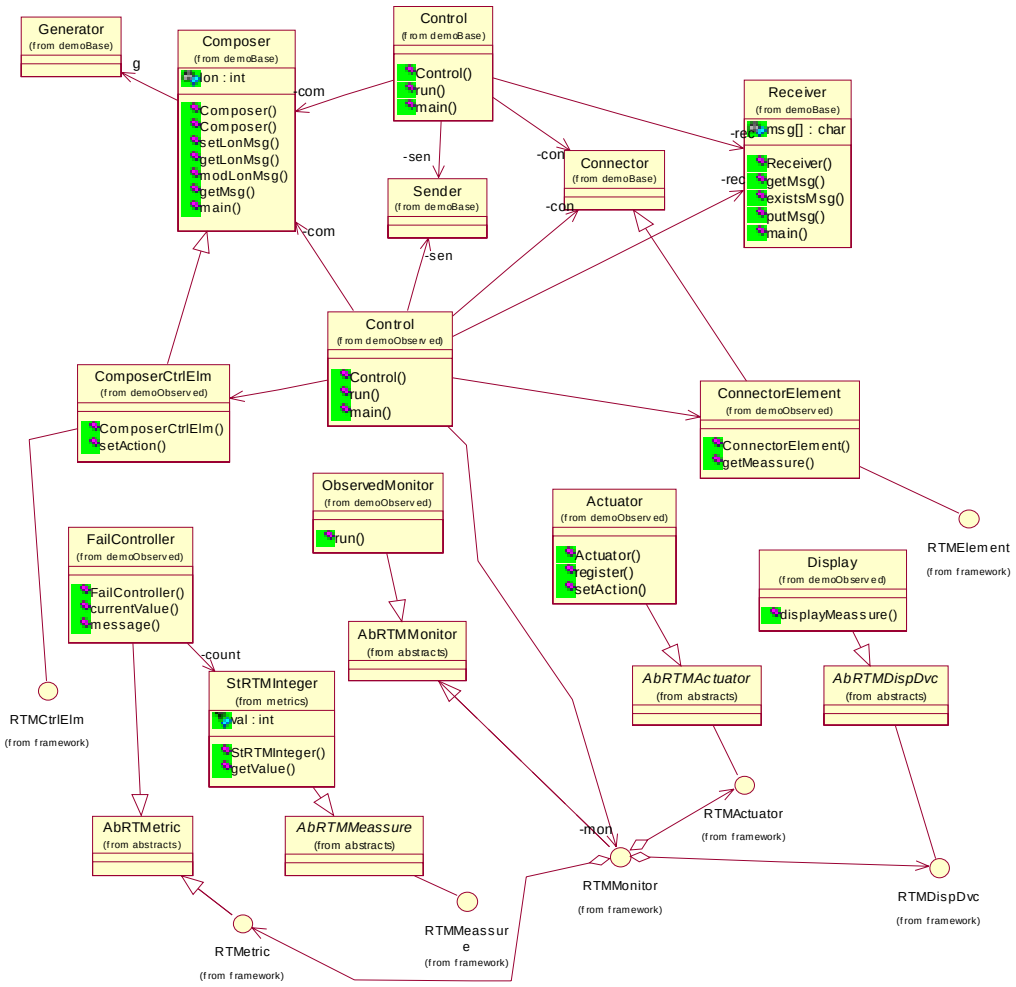


Ilustración 6-12. diagrama de clases (Caso 3)

En este diagrama se desea hacer notar que ahora la clase `Control` pierde toda relación con las clases más profundas del framework y sólo se relaciona con el monitor.

Para la creación de las clases ComposerCtrlElm y ConnectorElement se pueden barajar varias soluciones, aunque la más evidente es que sean creadas también por la clase Control como se puede ver en el diagrama.

El resto de las relaciones no dibujadas en el diagrama son las habituales entre las interfaces del framework ya definidas en el capítulo anterior.

6.5.2. IMPLEMENTACIÓN

La implementación de los algoritmos básicos de funcionamiento del sistema es similar a los definidos en pasos anteriores, a diferencia de que en este caso las llamadas al sistema de monitorización sólo se producen cuando falla la entrega de un mensaje.

```
…
      while ((sen.msgNumber() > 0)|| (sen.isWorking()))
      {
         try
         {
         …
         sen.update(); // Lanza DemoException en caso de fallo
         …
         }
         catch(DemoException e)
         {
            mon.run(); // Iteración del sistema de control
            if (con.failShort())
               System.out.println("Mensaje corto");
            else
               System.out.println("Mensaje Largo");
            System.out.println(e.toString());
         }
         System.out.println("Tamaño mensaje : "+com.getLonMsg());
      }
…
```

Por tanto la monitorización sólo se produce en ese caso y no durante toda la ejecución como se hacía antes. Esta es la razón de que el sistema sirva sólo cuando no importa hacer históricos de la ejecución, porque es ese caso es mejor solución la del caso de estudio anterior.

6.6. CASO DE ESTUDIO 4

Otra posibilidad es la de implementar la solución a través de un patrón *smart proxy* [GAM95].

Esta solución permite intervenir en el objeto de un modo casi transparente antes y después de la ejecución de los métodos, de manera que se pueden usar estas intervenciones como puntos de toma de muestras aunque no hubieran sido definidos en el sistema original.

Pero esta solución requiere replantearse una parte del diseño original y no sólo derivar de la clase observada, ya que el patrón requiere que todas las clases implementadas como proxy hereden de una interfaz común.

No es, por tanto, un sistema totalmente transparente, ya que requiere que el diseño original haya sido pensado de este modo. Aunque en ningún caso se ha conseguido una transparencia total, este sistema (que en un principio puede parecer la mejor so-

lución) tiene desventajas adicionales, como por ejemplo que hay que intervenir no solo en la implementación del sistema original, sino también en su diseño.

Puede ser, sin embargo, un buen sistema si ha sido previsto desde un principio, esto es, si en el diseño original ya se han previsto los puntos de medida y esas clases ya han sido implementadas siguiendo el patrón *smart proxy*.

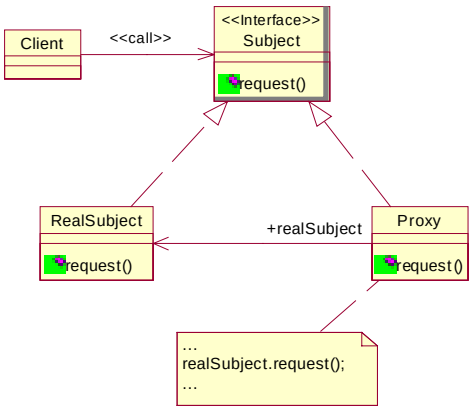


Ilustración 6-13. Diagrama del patrón proxy (Caso 4)

En este caso lo que hay que buscar es la clase a la que se le deba colocar el proxy para obtener la interfaz con el comportamiento deseado.

El objetivo en este caso será el mismo que en el caso de estudio anterior y se implementará sobre el conector para capturar las llamadas al envío de mensajes y comprobar si han fallado, en cuyo caso se lanzará el monitor para que opere sobre el sistema.

Así pues, las clases equivalentes son en este caso:

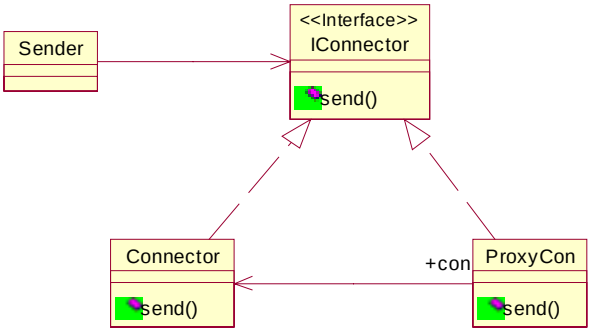


Ilustración 6-14. Diagrama de clases con proxy (Caso 4)

El problema de este tipo de implementación es que requiere una modificación importante del diseño obtenido por las metodologías tradicionales y requiere dicha modificación en las últimas fases de DMT lo que provoca que sea imposible de extender cuando no se tienen los códigos fuente.

Otra posible solución sobre Java es utilizar directamente la solución de la clase Proxy del API estándar de Java (a partir de la versión 1.3).

En cualquier caso requiere remodificar el diseño original o tener en cuenta en las fases de diseño tradicionales del sistema las clases que se van a medir (pero esto requeriría una ampliación de la metodología DMT que haría que se perdiese en parte el concepto de separación de aspectos con el que fue concebida).

En cualquiera de ambos casos, el sistema sería demasiado dependiente del sistema de métricas y control y no se obtendría una buena separación de aspectos en la filosofía general del diseño con DMT y por tanto no es adecuada a los principios con los que fue concebida DMT.

6.7. CASO DE ESTUDIO 5

Para este caso se ha decidido utilizar dos hilos (threads) de ejecución, uno para la aplicación ya desarrollada por el método de diseño tradicional e implementada en el `package demoBase` y otro para el sistema de métricas y control de la aplicación.

De este modo se consigue una mayor separación de aspectos, dejando la aplicación original casi intacta y diseñando el sistema de control con base en los puntos a medir y controlar, sin tener que intervenir en el resto de la aplicación de origen.

Este sistema así diseñado permitiría colocar incluso el código de control en una máquina diferente y hacer el control con una intervención mínima en la aplicación que se pretende monitorizar, cumpliendo de este modo con las tres premisas marcadas en el capítulo anterior:

1. Concurrencia de la ejecución del sistema de métricas y del sistema medido.

2. Separación de los elementos de ejecución y de medida, hasta el punto de que llegaran a estar en máquinas diferentes.

3. Separación conceptual de la implementación y por tanto del diseño de la aplicación que nace con el requerimiento de ser medida.

Todos los pasos de la metodología aplicados en los casos de estudio anteriores son válidos para el diseño hasta la aplicación del paso a las clases del framework.

6.7.1. PASO DEL SISTEMA A LAS CLASES DEL FRAMEWORK

En este caso se ha pasado el sistema a un modelo de clases e interfaces, de acuerdo al diagrama de clases de la Ilustración 6-15.

Se ha reutilizado gran parte del diseño de los casos de estudio anteriores, a excepción de las clases que implementan el monitor.

Pero este caso se ha optado por un diseño en el que la clase `TimeMonitor` (implementación del monitor) se apoya en la clase `SimpleThread` que soporta el segundo hilo de ejecución.

La interfaz de la clase `TimeMonitor` ha sido ampliada con los métodos `stop()` y `update()`.

El método `stop()` sirve para finalizar el proceso de monitorización y envía el mensaje `halt()` a la clase `SimpleThread` para terminar el hilo de ejecución que soporta la monitorización.

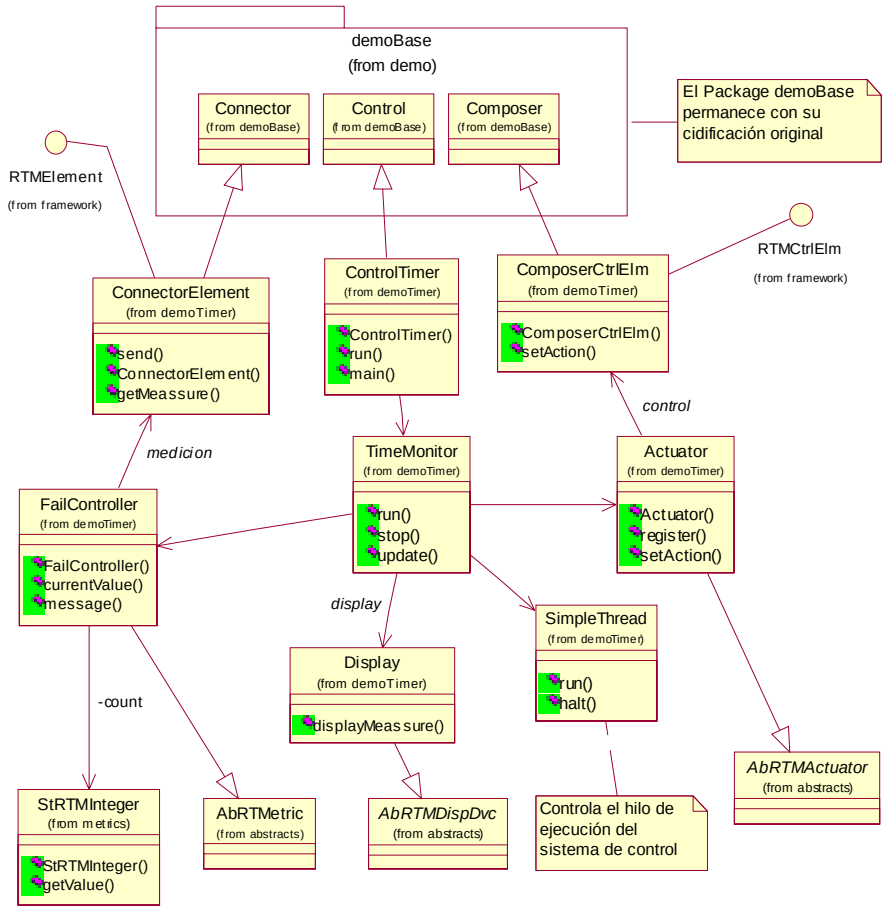


Ilustración 6-15. Diagrama de clases (Caso 5)

El método `update()` es enviado desde `SimpleThead` para realizar un ciclo de monitorización.

6.7.2. IMPLEMENTACIÓN

La implementación ha reutilizado gran parte de las clases de los casos de estudio anteriores, en concreto, las modificaciones importantes han sido realizadas en la clase `TimeControl` y en la clase `TimeMonitor`.

En la clase `TimeControl` se ha reimplementado la creación de los objetos participantes para poder crear todos los objetos del sistema de monitorización y se ha sobrecargado la línea principal de ejecución para poder registrar el elemento a medir en la métrica, el elemento a controlar en el actuador, los objetos del sistema de monitorización en el monitor y lanzar dicha monitorización.

```
public void run(TimerMonitor mon, int nMens) {
   // Control
   FailController fco;
```

```
      Actuator act;
      Display dis;
      // Control init
      fco = new FailController ();
      try {
         fco.register((RTMElement) con); // Registrar elemento a
medir
      }
      catch (RTMException e) {
         System.out.println(e.toString());
      }
      act = new Actuator();
      try {
         act.register((RTMCtrlElm) com); // Registrar elemento a
controlar
      }
      catch (RTMException e) {
         System.out.println(e.toString());
      }
      dis = new Display();
      mon.registerMetric(fco);  // Registrar objetos en el monitor
      mon.registerActuator(act);
      mon.registerDisplay(dis);
      mon.run(); // Lanzamiento del monitor
      super.run(nMens);
   }
```

Por lo que respecta a la clase `TimeMonitor`, se ha modificado para que lance las tomas de muestras y el control a intervalos no necesariamente sincronizados con el resto de la aplicación.

Lo que realmente se mide ahora es el incremento de fallos del sistema, de manera que si el número de fallos aumenta entre dos mediciones consecutivas es indicativo de que ha fallado desde el ultimo ciclo de medidas y por tanto se debe adaptar el tamaño del mensaje.

Para medir de manera asíncrona con el resto de la aplicación, el monitor ha sido arrancado en una thread diferente de la del programa. Por ello se ha creado la clase `SimpleThread` que arranca al monitor.

```
class SimpleThread extends Thread {
   private boolean running;
   private int slp;
   TimerMonitor t;
   public SimpleThread(TimerMonitor t, int slp)
   {
      this.t=t;
      this.slp = slp;
      running = true;
   }
   public void run()
   {
      while(running)
      {
         t.update();
         try
         {
         sleep(slp);
         }
         catch(InterruptedException e)
         {
            System.out.println(e.toString());
         }
      }
   System.out.println("Thread stopped");
   }
```

El método `run()` implementa el nievo hilo de ejecución y desde él se actualiza el monitor (`update()`). Con el objeto de no sobrecargar de ejecución este hilo y no ralentizar de manera importante el hilo principal, en cada ciclo se detiene la thread durante un tiempo que es configurable (`sleep()`).

6.8. CASO DE ESTUDIO 6

En este caso se hará el mismo modelo del caso de estudio 6 pero en la fase de implementación se utilizará el lenguaje de implementación JavaRTM definido en el capítulo anterior.

Desde este punto de vista todos los pasos de la metodología son idénticos a los del caso de estudio 6 a excepción de los dos últimos:

- Paso del sistema a las clases del framework.
- Implementación.

6.8.1. PASO DEL SISTEMA A LAS CLASES DEL FRAMEWORK

En esencia se consideran las mismas clases del caso anterior, pero en este caso habrá dos clases que nos generará automáticamente el nuevo lenguaje, por tanto se preverán las clases que no genera el lenguaje, esto es, todas menos `ConnectorElement` y `ComposerCtrlElm`.

6.8.2. IMPLEMENTACIÓN

En la fase de implementación se reimplementarán estas dos clases del sistema a controlar para que se adapten al nuevo sistema. Esta reimplementación se podría hacer reutilizando el código ya escrito y añadiéndole la parte nueva de JavaRTM.

Este modelo de programación impone que el código de la aplicación anterior sea conocido, al menos, para estas dos clases.

Para este caso, la declaración de las dos clases implicadas sería:

```
package es.uniovi.ootlab.metrics.demo.demoTimer;
import es.uniovi.ootlab.metrics.demo.demoBase.Connector;
import es.uniovi.ootlab.metrics.framework.*;
import es.uniovi.ootlab.metrics.metrics.*;

public class ConnectorElement extends Connector {
   //# metrics begin:        A0001
   //# metric class:         FailController  ## Clase RTMetric
   //# measure class:        StRTMInteger ## Clase RTMMEassure
   //# element code:
   //#> {
   //#>    return new StRTMInteger(fLong-fShort);
   //#> }
   //# monitor class single: StRTMMonitor ## Clase RTMMonitor
   //# display class:        Display   ## Monitoriza alarma
   //# actioner class:       Actuator     ## clase actioner
   //# controlled class:     Composer     ## clase controlada
   //# metrics end:          A0001

   // . . . Resto de la clase . . .
}
```

La clase `Composer` tendría el siguiente código:

```
package es.uniovi.ootlab.metrics.demo.demoTimer;

import es.uniovi.ootlab.metrics.demo.demoBase.Composer;
import es.uniovi.ootlab.metrics.framework.*;
import es.uniovi.ootlab.metrics.metrics.*;

public class ComposerCtrlElm extends Composer {
   //# metrics begin:   A001
   //# element class:   Connector  ## clase element
   //# actioner class:  Actuator   ## clase que realiza el control
   //# actioned code:
   //#> {
   //#>    modLonMsg(((Double)param).doubleValue());
   //#> }
   //# metrics end:     A001

   // . . . Resto de la clase . . .
}
```

El código generado por estas definiciones sería similar al que se poseía en los casos anteriores a excepción de que en este caso se usa la clase estándar `StRTMMonitor` que es parte de las utilidades del framework, pero que tiene una implementación similar a la `TimerMonitor` del caso anterior. (Ver Apéndice D).

7. CONCLUSIONES

7.1. CONCLUSIONES

7.1.1. HERRAMIENTA Y SEPARACIÓN DE ASPECTOS

Esta tesis doctoral ha querido profundizar en varios aspectos de las métricas que han quedado un tanto limitados de contenidos en otros trabajos anteriores. En concreto, la mayor parte de los trabajaos anteriores se han centrado en el diseño de nuevas métricas con objetivos muy diferentes, mientras que en este trabajo se ha buscado el enfoque de una separación de aspectos básica que permitiera diseñar métricas para tiempo de ejecución, de forma independiente del objeto a medir y se han diseñado herramientas básicas que soporten este enfoque.

La primera herramienta es una metodología, que viene acompañada de un sistema de elementos gráficos que se pueden unir a cualquier sistema de diagramas de diseño convencional de modo que doten al diseñador de un elemento con el cual puede ampliar cualquier diseño para determinar cómo debe medirse o cómo puede diseñarse la medición.

Adicionalmente se ha ampliado el sistema para conseguir que además de ser medible sea intervenible, monitorizable y controlable, siempre sin perder de vista el concepto de separación básica de aspectos de programación que permitía el enfoque inicial.

Como conclusión, con este nuevo enfoque se ha conseguido un sistema de medición, supervisión y control de aplicaciones software que puede ser instalado sobre cualquier diseño con un grado de separación de aspectos de programación bastante importante, lo que permite diseñar el control aparte del modelo de diseño del software originalmente concebido.

En cualquier caso, a pesar de las herramientas, aún es necesario diseñar con claros objetivos de medir, ya que en otro caso puede resultar incluso imposible realizar la medición.

En los experimentos realizados con los casos de estudio se ha detectado este problema, esto es, que si se diseña sin intención de medir, se pueden llegar a conseguir sistemas imposibles de medir. A modo de ejemplo, y basándose en los casos de estudio anteriores, si los mensajes se generan todos al principio de la aplicación y se almacenan en un array para después ser enviados, es imposible intervenir el sistema para modificar el tamaño cuando los mensajes empiecen a fallar, ya que ya han sido generados en ese momento y no es posible intervenir en el sistema para hacer una generación de mensajes con un tamaño diferente (mayor o menor del actual).

Por tanto sólo se podrá medir el sistema que haya sido diseñado con esa intención y en muchos casos la intención de medir, incluso con toda la separación de aspectos que integra el sistema diseñado, hará que los diseños originales deban ser retocados

con esa intención, incluso a pesar de que la separación de aspectos que el sistema proporciona debiera contribuir a no intervenir en el sistema.

Este tipo de sistemas son, por diseño, intrínsecamente no controlables, o al menos lo son de acuerdo a dicho diseño elegido.

7.1.2. METODOLOGÍA

Se ha diseñado una metodología de diseño que ha sido empleada en todos los casos de estudio con éxito y que sirve para diseñar sistemas medibles, monitorizables y controlables, a partir de modelos de diseño convencional.

En los experimentos realizados se ha comportado bien, dando lugar a sistemas bien diseñados que respetan el objetivo original de separación de aspectos de la programación.

En algunos casos, la implantación del sistema (la fase de implementación) ha requerido el rediseño de alguna parte del software original para hacer que pueda ser medible o controlable mediante el sistema diseñado para la metodlogía DMT y las herramientas accesorias (framework, JavaRTM, etc.).

Esto se debe a cuestiones puramente intrínsecas del diseño, como se expuso en el ejemplo del apartado anterior.

Adicionalmente se ha diseñado una sintaxis completa para diagramas de clases con información de los sistemas de control que hayan resultado de la aplicación de la metodología.

Ambos juntos, la metodología y los diagramas de clases ampliados permiten (como se ha visto en los casos de ejemplo) intervenir en sistemas diseñados para añadirles el sistema de monitorización y control y hacerlo de forma que, incluso en la fase de diseño, se vea la separación en los objetivos de ambos sistemas (aplicación y control).

7.1.3. FRAMEWORK

Con el objetivo de dar soporte de programación al sistema se ha diseñado un framework que implementa las ideas básicas de separación de aspectos de programación y de intervención mínima en el sistema monitorizado.

El diseño original ha sufrido algunos cambios mínimos durante los experimentos y, sobre todo, en el concepto de utilización.

En origen la idea era heredar desde las clases abstractas para aprovechar la implementación básica de algunos métodos que se prevé no cambien entre diferentes sistemas.

En el caso actual de Java, esto no ha sido posible en gran parte de los casos debido a que Java carece de herencia múltiple, lo que ha obligado a implementar directamente las interfaces aunque ello conlleve tener que repetir código en muchos casos. (Si se implementan varias métricas y controles)

La razón es que el sistema de métricas se basa en la herencia desde algunas clases del sistema, por lo que ya no es posible heredar de las implementaciones del framework.

En cualquier caso, el resultado es positivo, pero se pierde un poco la capacidad de automatización del sistema de monitorización y se requiere la intervención del usuario en estas adaptaciones.

El objetivo de este framework era doble: por un lado dar soporte a los diseños en Java (como se ha visto) y de otro no tener que definir nuevos lenguajes específicos para soportar la separación de aspectos ni tener que usar lenguajes específicos de este tipo de enfoque que, en el momento actual, aun no tienen una penetración importante en el mercado y son, en general desconocidos por los desarrolladores.

En este sentido se ha huido de soluciones como AspectJ [@APSCJ], ya que aún es un proyecto de investigación y no una herramienta de programación ampliamente utilizada.

Una solución a la utilización del enfoque original del framework se podría conseguir trasladando todo el sistema a un lenguaje que soporte herencia múltiple, como por ejemplo C++. En este caso si sería posible heredar de las clases del diseño de la aplicación y de las clases abstractas del framework al mismo tiempo aprovechando así implementaciones por omisión en muchos casos.

El resultado final del framework resulta en una herramienta utilizable en diseños reales, sencilla de entender y con mucha capacidad de reimplementación de soluciones y de reutilización de código.

Uno de los puntos que se puede mejorar en aplicaciones reales es la parte de las clases de display, que pueden convertirse en vistas de un modelo de arquitectura MVC. Esto produciría que el mismo sistema pudiera tener diferentes vistas de manera que se mejorase la representación de las mediciones y la monitorización.

7.1.4. LENGUAJE CON SEPARACIÓN DE ASPECTOS DE PROGRAMACIÓN

Se ha definido un lenguaje que permite separar los aspectos de programación del algoritmo propiamente y de monitorización y control en dos elementos diferentes, pero que, al mismo tiempo, no requiere conocer nuevos conceptos de programación demasiado alejados del origen en Java.

Al ser escrita la parte de monitorización dentro de comentarios, permite que si se compila el código con un compilador de Java tradicional (sin paso previo por el JavaRTM), la compilación dé lugar a la aplicación sin sistema de monitorización y control, pero que funciona de acuerdo al diseño original. De modo que los aspectos quedan absolutamente separados.

En cualquier caso, el alcance de este lenguaje ha quedado muy limitado cuando se trata de métricas que no sean las estándar definidas en el propio framework, ya que en este caso hay que implementar en Java el resto del sistema.

Lo que si ha quedado demostrado es que se pueden separar los aspectos de la aplicación y del control, que era uno de los objetivos del lenguaje.

Para más abundar en este hecho, JavaRTM se ha diseñado de manera que es casi declarativo, ya que sólo necesita poner código en dos cláusulas (ELEMENT CODE y ACTIONED CODE), siendo el resto de las cláusulas puramente declarativas de las condiciones o reglas que hacen colaborar al sistema.

En el prototipo de lenguaje implementado se ha conseguido llegar a la generación de las clases que intervienen en el sistema original a controlar, esto es las que deben implementar las interfaces `RTMElement` y `RTMCtrlElm`.

Este resultado es bueno pero se debe seguir trabajando, ya que el diseño del lenguaje original permite expresar mucha más información sobre la colaboración del resto de las clases de framework y se debería llegar a la implementación total del sistema basándose en las reglas declarativas de JavRTM, en el diseño de colaboraciones entre las interfaces del framework y en las implementaciones estándar de algunas clases.

7.1.5. RESULTADO DEL CONJUNTO DE LAS HERRAMIENTAS

El conjunto de todas las herramientas y prototipos desarrollados dentro del alcance de esta tesis aportan un elemento básico de desarrollo de aplicaciones reales con implementación de control, medida y monitorización incluidos.

Es posible la utilización del framework y de la metodología DMT en proyectos reales y el lenguaje JavaRTM es un apoyo importante, si bien es un elemento que debe seguir siendo investigado en varios sentidos, desde la ampliación hasta el replanteamiento en base a los nuevos lenguajes con separación de aspectos o incluso basándose en declaraciones específicas realizadas en otros lenguajes (por ejemplo en XML).

En resumen, se han logrado con éxito casi total todos los objetivos planteados al principio de la tesis, aunque el camino a la investigación aún queda abierto.

7.2. LÍNEAS DE INVESTIGACIÓN FUTURAS

7.2.1. SISTEMA DE MÉTRICAS Y CONTROL CON SEPARACIÓN DE ASPECTOS EN TIEMPO DE EJECUCIÓN

Dentro de las posibles líneas de investigación futura, una de las más importantes es la de conseguir una separación de aspectos (control-aplicación) en tiempo de ejecución, de modo que se pudiera colocar un lazo de control sobre cualquier aplicación en ejecución sin detener dicha ejecución.

Para ello se podrían aprovechar los resultados de Francisco Ortín [ORTIN02] y definir de nuevo un framework que permitiera, en una máquina reflectiva realizar dicho tipo de separación.

Esto permitiría, siempre en sistemas que no sean *intrínsecamente no controlables*, realizar diseños de control ad-hoc cuando la aplicación ya está en explotación, con un conocimiento mínimo del sistema original.

7.2.2. NUEVO LENGUAJE

Por otro lado, y a la vista de los resultados obtenidos con el lenguaje JavaRTM, se podría plantear el diseño de un nuevo lenguaje que, además, soportase la definición de sistemas de monitorización y control para tiempo de ejecución.

Una posibilidad para esta definición consiste en la creación de un lenguaje nuevo de definición de sistemas de monitorización y control realizada en XML y que, mediante un parser (SAX o JDOM), transformase las definiciones en un sistema de objetos capaces de interaccionar con la aplicación y realizar su control.

Para un sistema con separación en tiempo de ejecución, este nuevo sistema podría ser reimplementado de un modo muy similar al presentado en este trabajo, pero si se desease que el control fuera diseñado para tiempo de ejecución se necesitaría de nuevo una máquina que soportase reflectividad para poder interaccionar con los objetos.

Otra posible solución puede venir de la mano de lenguajes que soporten directamente la separación de aspectos (como por ejemplo AspectJ).

Por último no debe descartarse la posibilidad de extender el JavaRTM actual hasta soportar toda la potencia necesaria para hacer tales definiciones, de manera que en extensiones sencillas de lenguajes actuales también se pudiera hacer esta definición con un alto grado de separación de aspectos.

7.2.2.1. NUEVA HERRAMIENTA JRTMCOMPILER

Además, en este último caso, sería imprescindible ampliar el prototipo actual de herramienta de compilación hasta convertirla en un entorno completo de desarrollo que permita no sólo compilar clases separadas, sino estudiar un proyecto completo y establecer sobre él todos los lazos de monitorización, medición y control en función de las distintas reglas expresadas en los diferentes ficheros fuente (`*.jrtm`).

7.2.3. DEFINICIÓN DE MÉTRICAS PARA TIEMPO DE EJECUCIÓN

Otra posible línea de investigación reside en la propia definición de métricas para tiempo de ejecución que permitan obtener resultados estandarizados de rendimiento, adaptación a las especificaciones, gestión de recursos, etc.

Todas estas métricas nuevas estarían fuera de las clásicas que afectan principalmente al tiempo de diseño, aunque deberían tener algún tipo de relación con algunas de ellas, permitiendo, en tiempo de diseño, definir métricas para controlar partes del sistema en tiempo de ejecución, y además permitan predecir y demostrar estos comportamientos en ejecución, haciendo un software mucho más fácil de validar en las fases iniciales del diseño.

En este sentido se debería diseñar una suite completa de métricas que se pudiesen utilizar en lazos de control del sistema en ejecución y que tuviesen validación o comprobación en métricas de diseño, de manera que se pudiesen comparar los resultados en las dos fases del ciclo de vida de la aplicación.

7.2.4. AMPLIACIÓN DEL FRAMEWORK

Adicionalmente a punto 7.2.3 se podrían integrar todos estos nuevos sistemas de métricas en las implementaciones estándar del framework, permitiendo así la utilización de métricas con menos componente de improvisación y con resultados contrastables entre diferentes aplicaciones.

Esto permitiría un sistema de patrones de control y de métricas que varias aplicaciones diferentes podrían implementar con el objetivo de hacer los resultados comparables y contrastables.

Por otro lado el framework debería ser ampliado con el objetivo de conseguir la integración del sistema de monitorización y control para varias aplicaciones que cooperan, y en este sentido tendría mucha importancia utilizar arquitecturas de separación de vista y modelo (como MVC), de manera que se pudiera llegar incluso a la separación de aspectos dentro del propio sistema de control. Esta separación conllevaría, a su vez, a una separación del hilo de ejecución del sistema, de manera que el control y la monitorización pudiesen correr en hilos de ejecución diferentes e incluso en máquinas diferentes.

Una vez en máquinas diferentes, no sería difícil hacer que la monitorización recogiese información de diferentes sistemas que están en ejecución al mismo tiempo para poder hacer comparativas en tiempo real.

De este modo se puede conseguir que unos procesos intervengan en el comportamiento de otros sin que el modelo de intervención haya sido definido completamente en el diseño.

Como resultado, esto permitiría también, controlar modelos de productor-consumidor para optimizar buffers intermedios, incluso cuando todos estos procesos ocurren en nodos diferentes.

Para ello sistemas como el patrón proxy, RMI o los ORB's pueden ayudar dando el soporte de comunicaciones entre objetos de diferentes aplicaciones.

7.2.5. AMPLIACIÓN DE LA METODOLOGÍA

En secuencia, estos resultados podrían provocar la ampliación de la metodología haciendo que las métricas estándar se pudieran colocar en los propios símbolos, aumentando de este modo, la cantidad de información que proveen los gráficos asociados a los diagramas de clases ampliados.

Esto permitiría el intercambio de información entre diferentes desarrolladores al igual que ahora sucede con los diseños basados en UML estándar.

Para ello habría que definir el modelo de proposición al OMG para el reconocimiento de la extensión de UML con unos sistemas de diagramas y elementos semánticos adicionales que permitan la definición del control y la monitorización sobre procesos de software.

Por otro lado es necesario ampliar el soporte de la metodología para poder incluir sistemas como los descritos en los casos anteriores, en los cuales el monitor se convertirá en un auténtico centro de control de múltiples procesos que cooperan.

Además se debe investigar para adaptar la metodología a diferentes tipos concretos de proyectos (tiempo real, clusters, etc.) de manera que la metodología se convierta en un sistema universal de diseño de control sobre sistemas de software.

7.2.5.1. DISEÑO DE UNA HERRAMIENTA CASE (CASE-RTM)

En consecuencia se podría desarrollar una herramienta CASE que soportase UML y la ampliación DMT, de manera que se diese soporte a todo el proceso de diseño de la aplicación y que fuese posible la generación automática de código.

Esta herramienta debería definir una nueva vista del sistema que sería la resultante del sistema cooperativo con las definiciones de los lazos de control y monitorización, de manera que fuese posible en una primera instancia el paso de los sistemas definidos mediante DMT a sistemas de clases de UML de forma automática.

También debería permitir la definición de diferentes modelos de framework de soporte, de forma que se pudiesen integrar nuevos framework más evolucionados a medida que avancen las investigaciones.

7.2.5.1.1. GENERACIÓN AUTOMÁTICA DE CÓDIGO

La herramienta CASE, como ya se ha comentado, debería permitir la generación automática de código en dos sentidos principalmente: en tiempo de diseño y en tiempo de ejecución.

En tiempo de diseño, se podrían generar lenguajes convencionales (Java, C++, etc.) o también lenguajes específicos como JavaRTM o lenguajes independientes como la definición antes comentada en XML.

En tiempo de ejecución (o de explotación) sería importante para poder colocar nuevos lazos de control sobre sistemas que corren en máquinas reflectivas, de manera que ante comportamientos no previstos del sistema, se pudieran colocar nuevos lazos de control probados sobre el sistema originalmente diseñado.

7.2.5.2. XMI - MOF

XMI es una tecnología definida por el W3C [@W3C] para soportar, en un metalenguaje definido en paralelo a XML, el intercambio de diseño entre herramientas de desarrollo (CASE, entornos de desarrollo, etc.)

MOF (Meta Object Facility) es una tecnología definida por OMG [@OMG] para la definición de metasistemas de desarrollo de manera que permita el intercambio y la compartición de información (en el proceso de desarrollo) entre grupos de trabajo distribuido.

Por último, debería ser posible la ampliación de XMI para soportar este enfoque, de modo que se pudiera integrar en este lenguaje estándar de intercambio de diseños toda la información sobre el control de la aplicación y de este modo fuera relativamente sencillo el intercambio de diseños entre herramientas (CASE, entornos de desarrollo, definición de interfaces IDL, etc.)

Tampoco se debe olvidar una referencia a MOF. Mediante esta tecnología se podría n integrar las herramientas definidas para DMT en los metasistemas definidos MOF, de manera que se pudiera intercambiar información no solo entre herramientas que soporten el enfoque, sino también entre herramientas que no lo soporten y, de este modo, hacer que la conversión de los formatos fuera automática.

8. APÉNDICE A: REFERENCIAS

PUBLICACIONES

[ABB94] A PROPOSED DESIGN COMPLEXITY MEASURE FOR OBJECT-ORIENTED DEVELOPMENT
Abbott, D. H., T. D. Korson, et al
NO. TR 94-105, Clemson University, 1994

[ALBE79] MEASURING APLICATION DEVELOPMENT PRODUCTIVITY
Albertch A. J.
IBM Application Development Symposium (1979)

[ALBE83] SOFTWARE FUNCTION, SOURCE LINES OF CODE AND DEVELOPMENT EFFORT PREDICTION: A SOFTWARE SCIENCE VALIDATION
Albertch A. J. y Gaffney J. E.
IEEE transaction on software engineering, vol. 9, nº 6, Noviembre 1983, pp. 639-648

[AQU94] DISEÑO Y CONSTRUCCIÓN DEL LENGUAJE ANIMA OBJECT
Aquilino A. Juan
Actas del Congreso Internacional de Ingeniería de Proyectos. pp. 842-860. Oviedo, Octubre de 1994.

[AQU97] NECESIDAD DE SISTEMAS FORMALES DE MÉTRICAS PARA PROYECTOS DE SOFTWARE OO
Aquilino Adolfo Juan Fuente, Luis Joyanes Aguilar y Juan Manuel Cueva Lovelle
Actas del las III Jornadas sobre Tecnología de objetos. pp.107-117. 15-17 de Octubre de 1997

[AQU98] ESPECIFICACIÓN FORMAL DE COMPONENTES EN JAVA (PROGRAMACIÓN POR CONTRATO)
Aquilino A. Juan, et al..
Proceedings del IV Congreso sobre Tecnología de Objetos, pags. 131-138. (Bilbao 1998)

[AQU01] SISTEMA DE MÉTRICAS PARA TIEMPO REAL EN APLICACIONES JAVA
Aquilino A. Juan Fuente, Raúl Izquierdo Castanedo, Juan Manuel Cueva Lovelle, Benjamín López Pérez y Luis Joyanes Aguilar
SISOFT 2001. Simposio Iberoamericano de Sistemas de Información e Ingeniería de Software en la Sociedad del Conocimiento. Actas del Congreso ISBN: 95897030-3-8. Santa Fé de Bogotá. Colombia 29 - 31 Agosto, 2001.

[AQU02] RUN-TIME METRICS SYSTEM ON JAVA APPLICATIONS
Juan Fuente Aquilino A., Izquierdo Castanedo R., Cueva Lovelle J.M., López Pérez B. Y Joyanes Aguilar L.
Enviado a la revista SIGPLAN (pendiente de publicación)

[BALL01] THE PROBLEM OF COMPUTING COUPLING AND COHESION IN EARLY STEPS OF DESIGN WITH INSTANCE VARIABLES
González-Ballester, Manuel; Oller-Segura, Angel; Granja-Alvarez, Juan Carlos; Juan-Fuente, Aquilino Adolfo y Hodgson, Peter
International Conference on Information Systems, Analysis and Synthesis. Proceedings SCI 2001/ISAS 2001, VOLUME XI. Orlando, Florida 22-25 de Julio de 2001.

[BANK91] OUTPUT MEASUREMENT METRICS IN AN OBJECT-ORIENTED COMPUTER AIDED SOFTWARE ENGINEERING (CASE) ENVIRONMENT: CRITIQUE, EVALUATION AND PROPOSAL
Banker, R D, R J, Kauffman And R, Kumar
Proceedings of the 24th Hawaii International Conference on System Science,. Hawaii: IEEE Computer Society Press, 1991.

[BAR93] INHERITING SOFTWARE METRICS
G. Michael Barnes and Bradley R. Swim
Journal of Object-Oriented Programming, Vol 6, nº 7, pp. 27-34 (1993)

[BASI82] A METHODOLOGY FOR COLLECTING VALID SOFTWARE ENGINEERING DATA
Basili, V. R. and Weiss D. M.
University of Maryland Technical Report

[BASI88] THE TAME PROJECT: TOWARDS IMPROVEMENT-ORIENTED SOFTWARE ENVIRONMENTS
Basili, V. R. y Rombach, H. D.
IEEE Trans. on Software Engineering, vol. 14, nº 6, junio, 1988, pp. 758-773.

[BIE91] DERIVING MEASURES OF SOFTWARE REUSE IN OBJECT-ORIENTED SYSTEMS
Bieman, J M
Technical Report No. Cs-91-112, Department of Computer Science, Colorado State University, 1991

[BOEH81] SOFTWARE EGINEERING ECONOMICS
Barry Boehm
Prentice-Hall 1981

[BOO94] OBJECT-ORIENTED ANALYSIS AND DESIGN WITH APPLICATIONS.
Grady Booch.
Editorial Addison Wesley. (1994)

[BOO96] UNIFIED METHOD FOR OBJECT-ORIENTED DEVELOPMENT (DOCUMENTATION SET)
Grady Booch & James Rumbaugh.
Rational Software Corporation. (1996)

[BRI94] MOOD - METRICS FOR OBJECT-ORIENTED DESIGN
Brito e Abreu, F.
OOPSLA'94 Workshop Paper Presentation. 1994

[BRI96] EVALUATING THE IMPACT OF OBJECT-ORIENTED DESIGN ON SOFTWARE QUALITY.
Fernando Brito e Abreu.
Proc. Of the 3rd Int'l S/W Metrics Symposium (METRICS'96) (1996).

[BS5233] BS 5233; 1986 BRITISH STANDARD GLOSSARY OF TERMS USED IN METROLOGY (INCORPORATING BS 2643)
British Standards Institution.

[BS5750] BS 5759; ISO-9004 – 1987 QUALITY SYSTEMS; PART 0; PRINCIPAL CONCEPTS AND APPLICATIONS; SECTION 0.2 GUIDE TO MANAGEMENT AND QUALITY SYSTEM ELEMENTS
British Standards Institution.

[BUS96] PATTERN ORIENTED-SOFTWARE ARCHITECTURE. A SYSTEM OF PATTERNS.
Frank Buschmann et al.
Editorial wiley (1996).

[CANT94] APPLICATION OF COGNITIVE COMPLEXITY METRICS TO OBJECT-ORIENTED SOFTWARE
Cant, S. N. et al.
Journal of Object-Oriented Programming, Julio/agosto, pp. 52-62 (1994)

[CHEN93] ON THE GEOMETRY OF FEEDFORWARD NEURAL NETWORK ERROR SURFACES
Mei Chen, Haw-minn Lu and Robert HechtNielsen
Neural Computation, 5(6):910--927, 1993

[CHI91] TOWARDS A METRIC SUITE FOR OBJECT ORIENTED DESIGN
Shyam R. Chidamber, Chris F. Kemerer
Proc. OOPSLA 91, pp. 197-211

[CHI94] A METRIC SUITE FOR OBJECT ORIENTED DESIGN
Shyam R. Chidamber, Chris F. Kemerer
IEEE Transaction on Software Engineering. Vol. 20, nº 6, junio, pp. 476-493

[CHI95] OBJECT ORIENTED DESIGN METRICS.
Shyam R. Chidamber, Chris F. Kemerer
Edited by Mathias Lother (1995).

[CHU92] INHERITANCE-BASED METRIC FOR COMPLEXITY ANALYSIS IN OBJECT-ORIENTED DESIGN.
Chung, C. M. And M. C. Lee
Journal of Information Science and Engineering, 8(3), PP431-447, 1992

[COPP92] SOFTWARE METRICS FOR OBJECT-ORIENTED SYSTEMS
Coppick, Chris J. And Thomas J. Cheatham
CSC '92 Proceedings, PP. 317-322, ACM, 1992

[COUL83] SOFTWARE SCIENCE AND COGNITIVE PSYCOLOGY.
Coulter, N. S.
IEEE Transaction on Software Engineering, SE-9 (1983)

[DEMA82] CONTROLLING SOFTWARE PROJECTS
DeMarco T., Fitzsimmons A., Love T. A.
Computer Surveys, 10 pp. 3-18 – Prentice Hall Inc.

[DEV95] GESTIÓN DE PROYECTOS ORIENTADOS A OBJETOS: UNA INCRUSIÓN CRUENTA.
Ricardo Devis Botella.
Revista NOVATICA nº 114. (1995)

[FEN91] SOFTWARE METRICS: A RIGUROUS APPROACH
Norman E. Fenton
Thonson Computer Press, ISBN: 1-85032-242-2 (1991)

[FEN96] SOFTWARE METRICS: A RIGUROUS & PRACTICAL APPROACH
Norman E. Fenton, Shari Lawrence Pfleeger
Thonson Computer Press, ISBN: 1-85032-275-9 (1996)

[FIOK88] THEORY OF MEASUREMENT IN TEACHING METROLOGY IN ENGINEERING FACULTIES.
Fiok A., Jaworski J., Morawski R.Z., Olędzki J., Urban A.C
Measurement, April/June 1988, Vol. 6, No 2, 63:68.

[FITZ78] REVIEW AND EVALUATION OF SOFTWARE SCIENCE
Fitzsimons, A. and Love, T. A.
Computing Surveys, 10, pp. 3-18

[GAM95] DESIGN PATTERNS. ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE.
Erich Gamma *et al*.
Editorial Addison Wesley.(1995)

[GEE96] FORMAL MEASUREMENT IN OBJECT-ORIENTED SOFTWARE.
Geert Poels, Guido Dedene.
Katholieke Universiteit Leuven. (1996)

[GON00] INSERCIÓN DE LA PROGRAMACIÓN POR CONTRATO EN FICHEROS OBJETO DE LA JVM (.CLASS).
M. T. González, Aquilino A. Juan, et al.
Proceedings del congreso SEID 2000. (Orense 2000)

[GRA95] MIGRATING TO OBJECT TECHNOLOGY
Graham, I
Addison-Wesley (1995)

[GREM84] DETERMINANTS OF PROGRAM REPAIR MAINTENANCE REQUIREMENTS
Gremilion, L. L.
Comunications of the ACM 27, pp. 826-832 (1984)

[GUNN62] TECHNIQUES OF CLEAR WRITING
Gunning R.
McGraw-Hill (1962)

[HALS77] ELEMENTS OF SOFTWARE SCIENCE
Halstead, M.H.
New York, Elsevier North-Holland, (1977)

[HAR82] M. H. HALSTEAD'S SOFTWARE SCIENCE – A CRITICAL EXAMINATION
Harmer, P. G. and Frewin, G. D.
Proc. 6th International Conference on Software Engoneering, pp. 197-206 (1982)

[HEN94] BOOKTWO OF OBJECT-ORIENTED KNOWLEDGE: THE WORKING OBJECT
Henderson-Sellers, B and J, Edwards
Prentice Hall: Sydney, Australia, 1994.

[HENR81] SOFTWARE STRUCTURE METRICS BASED ON INFORMATION FLOW
Henry, S. and Kafura, D.
IEEE Transactions on Software Engineering, SE-7, pp. 510-518 (1981)

[HOA00] UNDERSTANDING ROBUST AND EXPLORATORY DATA ANALYSIS
David C. Hoaglin, Frederick Mosteller
John W. Tukey (Editor) ISBN: 0-471-38491-7 MAY 2000

[HOR94] POSITION PAPER FOR OOPSLA METRICS WORKSHOP
Horner Simon A.
OOPSLA Workshops on O-O Metrics, 1994.

[HUM86] CONTROL FLOW AS A MEASURE OF PROGRAMMING COMPLEXITY
Humpphrey R. A.
Alvey Club on Software Reliability and Metrics Newsletter (1986)

[ISA-84] ANSI/ISA-S5.1-1984 (R1992) INSTRUMENTATION SYMBOLS AND IDENTIFICATION
American National Standard
Instrument Society of America (1992)

[JEN91] PARAMETRIC ESTIMATION OF PROGRAMMING EFFORT: AN OBJECT-ORIENTED MODEL
R. Jenson and J. Bartley
Journal of Systems and Software, Vol. 15, 1992, pp. 107 - 114.

[JOH91] SOFTWARE ENGINEER'S REFERENCE BOOK.
John McDermid.
Butterworth-Heinemann Ltd. (1991)

[JON86] PROGRAMMING PRODUCTIVITY
Capers Jones
McGraw-Hill, 1986

[JON88] FEATURE POINTS (FUNCTION POINT LOGIC FOR REAL TIME AND SYSTEM SOFTWARE)
Capers Jones
IFPUG Fall 1988 Conference, Montreal, QuEbec, Oct. 1988.

[JON95] WHAT ARE FUNCTION POINTS?
Capers Jones *et al.*
Software Productivity Research (1995).

[KICZ97] ASPECT ORIENTED PROGRAMMING.
Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin.
Proceedings of ECOOP'97 Conference.Finlandia. Junio de 1997.

[KITC81] MEASURES OF PROGRAMMING COMPLEXITY
Kitchenham, B.
ICL Technical Journal, 2, 298-316 (1981)

[KITC84] PROGRAM HISTORY RECORDS; A SYSTEM OF SOFTWARE DATA COLLECTION AND ANALYSIS
Kitchenham, B.
ICL Technical Journal, pp. 103-114 (1984)

[KITC86] SOFTWARE METRICS AND INTEGRATED PROJECT SUPPORT ENVIRONMENTS
Kitchenham, B. and McDermid, J. A.
Software Engineering Journal, 2, pp. 114-126 (1986)

[KITC86-1] TEST SPECIFICATION AND QUALITY MANAGEMENT DESIGN OF A QMS SUBSYSTEM FOR QUALITY REQUIREMENTS SPECIFICATION
Kictchenham, B. A., Walker, J. D. y Domville, I.
Project Deliverable A27, Alvey project SE/031, noviembre, 1986.

[KITC95] TOWARDS A FRAMEWORK FOR SOFTWARE MEASUREMENT VALIDATION
Kitchenham B., Pfleeger S. L. y Fenton N. E.
IEEE Transaction on Software Engineering, vol. 21, n. 12, 1995, pp. 929-944 (1995)

[LAN98] YODA: CONSTRUCCIÓN DE APLICACIONES BASADA EN COMPONENTES
Fernandez Lanvín, Raul Izquierdo Castanedo, Aquilino Adolfo Juan Fuente y Juan Manuel Cueva Lovelle
IV Congreso sobre Tecnología de Objetos. Actas del las IV Congreso de Tecnología de objetos. pp.89-96. Bilbao 13-16 de Octubre de 1998.

[LAR90] SOFTWARE SIZE ESTIMATION OF OBJECT-ORIENTED SYSTEMS
L. Laranjeira
IEEE Transactions on Software Engineering, Vol. 16, No. 5, May 1990, pp. 510 -522.

[LI93] MEASUREMENT OF SOFTWARE MAINTENANCE AND RELIABILITY IN THE OBJECT ORIENTED PARADIGM
Henry, S M and M, Lettanzi
OOPSLA `93 Workshop on Processes and Metrics for Object Oriented Software Development. Washington DC: 1993.

[LIBE89] ASSURING GOOD STYLE FOR OBJECT-ORIENTED PROGRAMS
K.J. Liberherr and I.M. Holland
IEEE Software, Vol. 6, No. 5, September 1989, pp. 38 - 48.

[LIP89] C++ PRIMER
Stanley B. Lippman
Editorial Addison Wesley. (1989)

[LOR94] OBJECT-ORIENTED SOFTWARE METRICS
M. Lorenz and J. Kidd
Prentice Hall, Englewood Cliffs, New Jersey, 1994 ISBN:0-13-179292-X

[NOV92] NOVÁTICA TECNOLIMPICS (TECNOLOGÍAS DE LAS OLIMPIADAS)
Monográfico dedicado a las Olimpiadas de Barcelona 92
Varios autores.
Novática, nº 92 – 1992

[MAR94] OO DESIGN QUALITY METRICS - AN ANALYSIS OF DEPENDENCIES (POSITION PAPER)
Martin, R. 1994
Proc. Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94. 1994.

[MCC76] A COMPLEXITY MEASURE
McCabe T. J.
IEEE Transaction on Software Engineering, SE-2, pp. 308-320

[MEY00] OBJECT-ORIENTED SOFTWARE CONSTRUCTION, 2ND EDITION
Bertrand Meyer
ISBN: 0136291554
Editorial Prentice Hall

[ORTIN02] SISTEMA COMPUTACIONAL DE PROGRAMACIÓN FLEXIBLE DISEÑADO SOBRE UNA MÁQUINA ABSTRACTA REFLECTIVA NO RESTRICTIVA
Francisco Ortín
Tesis Doctoral. Universidad de Oviedo. 2002.

[PER91] DIRECCIÓN Y GESTIÓN DE PROYECTOS
Jaime Pereña Brand.
Diaz de Santos, S. A. (1991)

[PER92] MÉTRICAS, UN ENFOQUE CUANTITATIVO A LA GESTIÓN DE PROYECTOS DE SOFTWARE.
Ignacio Pérez y Pedro L. Ferrer.
Revista NOVATICA nº 99. (1992)

[PIA95] MÉTRICAS PARA EL DESARROLLO ORIENTADO A OBJETO.
Mario Piatini
Revista BASE INFORMÁTICA nº 26. (1995)

[PUTN78] A GENERAL EMPIRICAL SOLUTION TO THE MACRO SOFTWARE SIZING AND ESTIMATING PROBLEM
Putnam L. H.
IEEE transaction on software engineering, vol. 4, nº 4, Julio 1978, pp. 345-361

[RAI91] FUNCTION POINTS IN AN ADA OBJECT-ORIENTED DESIGN?
E. Rains
OOPS Messenger, Vol. 2, No. 4, October 1991, pp. 23 - 25

[RIS93] AN INFORMATION HIDING METRIC
Rising, L.
Proc. OOPSLA `93 Workshop on Processes and Metrics for Object Oriented Software Development. Washington DC: 1993

[ROB95] FUNCTION POINTS.
Robert Vienneau
DACS/Kaman Sciences Corporation (1995)

[RUB83] MACRO-ESTIMATION OF SOFTWARE DEVELOPMENT PARAMETERS: THE ESTIMACS SYSTEM
Rubin, H. A.
IEEE SOFTAIR, Conference on software development tools, Techniques and Alternatives (1983)

[RUM91] OBJECT-ORIENTED MODELING AND DESIGN (OMT).
James Rumbaugh *et al*.
Editorial Prentice Hall. (1991)

[RUM99] THE UNIFIED MODELING LANGUAJE. REFERENCE MANUAL
James Rumbaugh, Ivar Jacobson and Grady Booch
Addison Wesley, 1999 (ISBN: 0-201-30998-X)

[SACK68] EXPLORATORY EXPERIMENTAL STUDIES COMPARING ONLINE AND OFFLINE PROGRAMMING PERFORMANCE
Sackman, H.; Erickson, W.J.; & Grant, E.E.
Communications of the ACM 11, 1 (January 1968): 3–11.

[SHE88] A CRITIQUE OF CICLOMATIC COMPLEXITY AS A SOFTWARE METRIC
Shepperd M.
Software Engineering Journal, 3, pp. 30-36 (1988)

[SHEE91] MEASURING OBJECT-ORIENTED SYSTEM COMPLEXITY
Sheetz, S. D., D. P. Tegarden, Et Al.
Proc. Workshop in Information Technologies and Systems WITS'91. 1991

[STEV46] ON THE THEORY OF SCALES AND MEASUREMENT
Stevens S. S.
Science 103, pp. 677-680

[STEV59] MEASUREMENT, EMPIRICAL MEANINGFULNESS AND THREE-VALUED LOGIC.
Stevens S. S.
(C. W. Churchman and P. Ratoosh) John Wiley, pp. 129-143

[SYM88] FUNCTION POINT ANALYSIS: DIFFICULTIES AND IMPROVEMENTS
Symons, Charles, R
IEEE Transactions on Software Engineering, Volume 14, No. 1, January 1988, pp. 2-11.

[SYS99] INFORME SOBRE MÉTRICAS APLICADAS A PROYECTOS DE SUPERVISIÓN DE PROCESOS INDUSTRIALES
Aquilino A. Juan
Métricas Orientadas a Objetos aplicadas al desarrollo de sistemas de supervisión y control de procesos industriales
Documento restringido. SYSTELEC Sistemas Electrónicos, S.L. (1999)

[SYS02] INFORME SOBRE MÉTRICAS APLICADAS A PROYECTOS DE INTEGRACIÓN DE MONITORIZACIÓN INDUSTRIAL (*No Terminado*)
Aquilino A. Juan
Métricas Aplicadas al Desarrollo de Software Orientado a Objetos para Sistemas de Integración de Monitorización Industrial
Documento restringido. SYSTELEC Sistemas Electrónicos, S.L. (2002)

[TAY93] SOFTWARE METRICS FOR OBJECT TECHNOLOGY
Taylor D.A.
Object Magazine, marzo/abril 1993, pp 22-28

[THOM94] PROJECT ESTIMATION USING AN ADAPTATION OF FUNCTION POINTS AND USE CASES FOR OO PROJECTS
Thomson, N, R, Johnson et al.
Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94. 1994.

[TVF98] INFORME SOBRE MÉTRICAS APLICADAS A PROYECTOS DE SOFTWARE
Aquilino A. Juan
Métricas para la realización de Proyectos de Software Orientado a Objetos
Documento restringido. TVF Proyectos y Obras, S.A. (1998)

[TVF98-1] INFORME SOBRE MÉTRICAS APLICADAS A PATRONES Y COMPONENTES
Aquilino A. Juan
Proyecto de Investigación sobre Métricas aplicadas a Patrones y Componentes Software
Documento restringido. TVF Proyectos y Obras, S.A. (1998)

[TVF99] INFORME SOBRE MÉTRICAS APLICADAS A FRAMEWORKS
Aquilino A. Juan
Métricas para la realización de Proyectos de Software Orientado a Objetos con Frameworks
Documento restringido. TVF Proyectos y Obras, S.A. (1999)

[TVF02] INFORME SOBRE MÉTRICAS APLICADAS A ENTORNOS WEB (*No Terminado*)
Aquilino A. Juan
Métricas aplicadas al Desarrollo de Software Orientado a Objetos para entornos WEB
Documento restringido. TVF Proyectos y Obras, S.A. (2002)

[WALS77] A METHOD OF PROGRAMMING MEASUREMENT AND ESTIMATION
Walston C. E. y Felix C. P.
IBM Systems journal, vol 6, nº1, 1977, pp. 54-73

[VOLV74] THE COST OF DEVELOPING LARGE-SCALE SOFTWARE
Wolverton, R. Y.
IEEE Transaction on Computers, vol 23, nº 6, junio 1974, pp. 282-303

[ZHAO95] SOFTWARE SIZING FOR OO SOFTWARE DEVELOPMENT - OBJECT FUNCTION POINT ANALYSIS
Hau Zhao, Stockman Tony
2nd Guide Share Europe International Conference on Information and Communication Technology and its related Management, Berlin, 9 - 12 octobre, 1995, pp. 12

PÁGINAS WEB

[@AMI] AMI
http://sting.web.cern.ch/sting/ami.html

[@ASPCJ] Página web del desarrollo del proyecto AspectJ: "AspectJ is a seamless integration of aspect-oriented programming (AOP) with Java"
http://aspectj.org

[@CYR96] PARAMETRIC COST ESTIMATING. REFERENCE MANUAL. BASIC COCOMO.
Kelley Cyr.
http://www.jsc.nasa.gov/bu2/COCOMO.html(1996)

[@GAHJD] JHOTDRAW START PAGE.
Erich Gamma et al.
http://www.jhotdraw.org/

[@JMET] JMetric WEB PAGE
http://www.it.swin.edu.au/projects/jmetric/products/jmetric/

[@JUNIT] JUNIT WEB PAGE
http://www.junit.org/index.htm

[@JVM] THE JAVA WEB PAGE
http://java.sun.com/

[@OMG] OMG
Object Management Group
http://www.omg.org

[@W3C] W3C
World Wide Web Consortium
http://w3c.org

[@WAL96] SINGLE FACTOR COST MODELS.
Walston & Felix.
IBM Federal Systems Division (1996)
http://carbon.cudenver.edu/~jkarimi/is6260/single.html

[@ZON96] THOR
B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A.C. Myers, L. Shrira (1996)
http://citeseer.nj.nec.com/liskov96safe.html

9. APÉNDICE B. DEFINICIÓN DETALLADA DEL FRAMEWORK

En los próximos párrafos se presenta el resultado del análisis y del diseño del framework realizado. Los diagramas utilizados son en UML.

9.1. CASOS DE USO

9.1.1. CASOS

9.1.1.1. OBSERVAR MEDIDA

Visualizar una medida en el display asociado de un elemento interno de la aplicación.

9.1.1.2. CONFIGURAR

Intervención del usuario para cambiar algún parámetro que permita dirigir la ejecución.

9.1.1.3. TOMAR MEDIDA

Toma una medida del sistema al que se desea controlar o monitorizar.

9.1.1.4. ACTUAR SOBRE EL SISTEMA

Interviene en el sistema a controlar y cambia el estado de algún objeto para obtener un comportamiento predefinido.

9.1.2. ACTORES

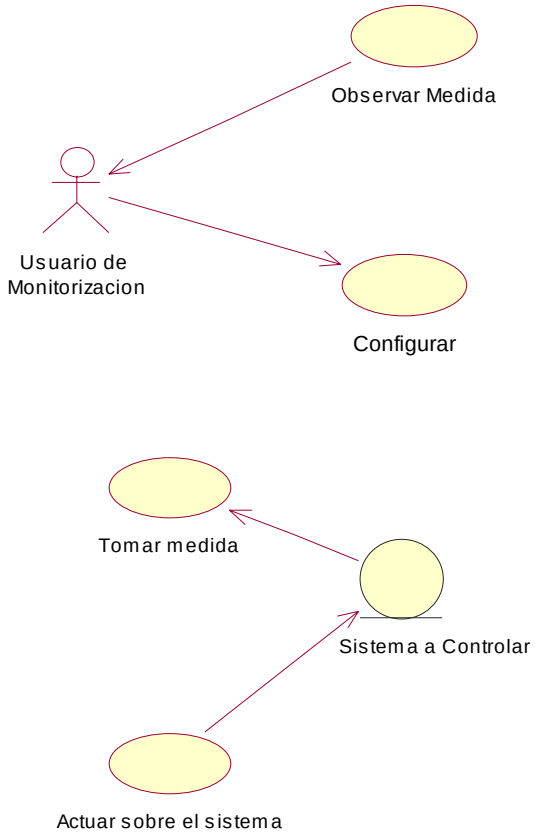
9.1.2.1. USUARIO DE LA MONITORIZACIÓN

Es el usuario que controla la ejecución de la aplicación bajo control y que puede intervenir en el sistema a través del sistema de monitorización.

9.1.2.2. SISTEMA A CONTROLAR

Es la aplicación que se desea monitorizar y controlar.

9.1.3. DIAGRAMA DE CASOS DE USO



9.2. ESCENARIOS

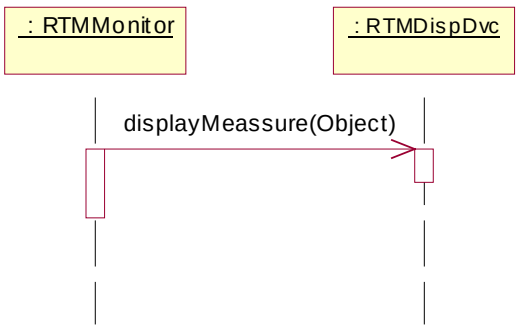
9.2.1. OBSERVAR MEDIDA

Iniciado por RTMMonitor.

Terminado por RTMDispDvc.

Precondiciones: previamente se debe haber leído la medida desde RTMetric.

Visualizar una medida en el display asociado de un elemento interno de la aplicación.



9.2.2. CONFIGURAR

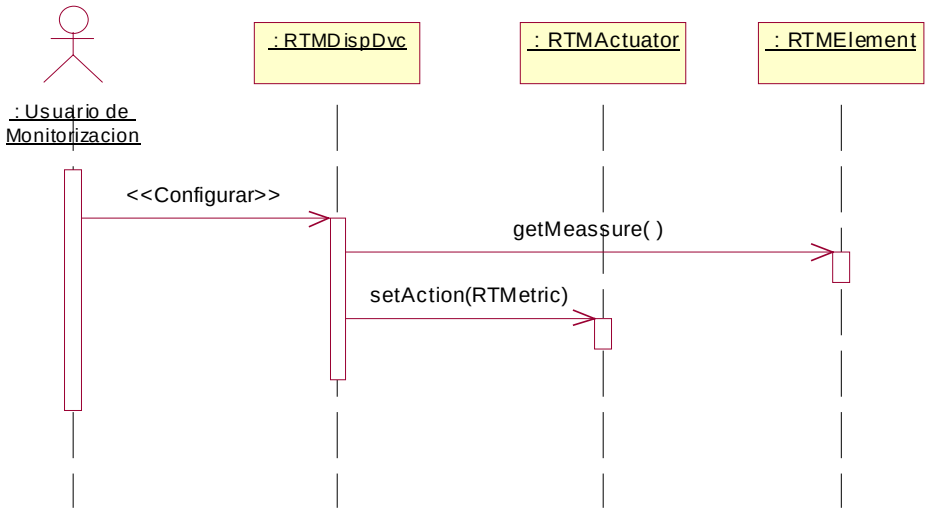
Iniciado por Usuario de Monitorización.

Terminado por RTMDispDvc.

Precondiciones: Ninguna

Notas: En implementación se debería contar con un patrón MVC para este tipo de implementación que no va a ser definido en el framework estándar.

Intervención del usuario para cambiar algún parámetro que permita dirigir la ejecución.

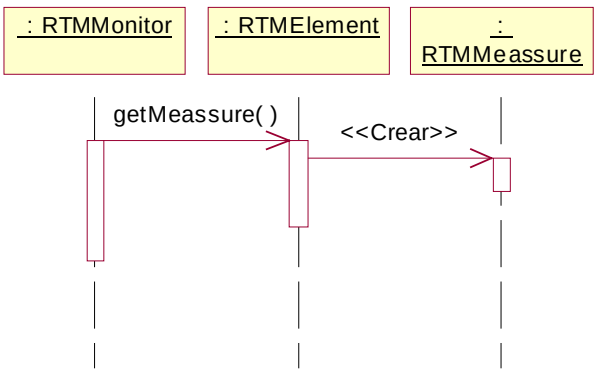


9.2.3. TOMAR MEDIDA

Inicializado por RTMMonitor.

Terminado por RTMMonitor.

Toma una medida del sistema al que se desea controlar o monitorizar.



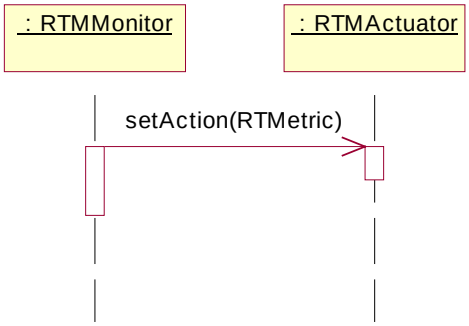
9.2.4. ACTUAR SOBRE EL SISTEMA

Inicializado por RTMMonitor

Terminado por RTMMonitor

Precondiciones: previamente se ha ejecutado una operación de lectura de métrica.

Actualiza el sistema para obtener un comportamiento específico.

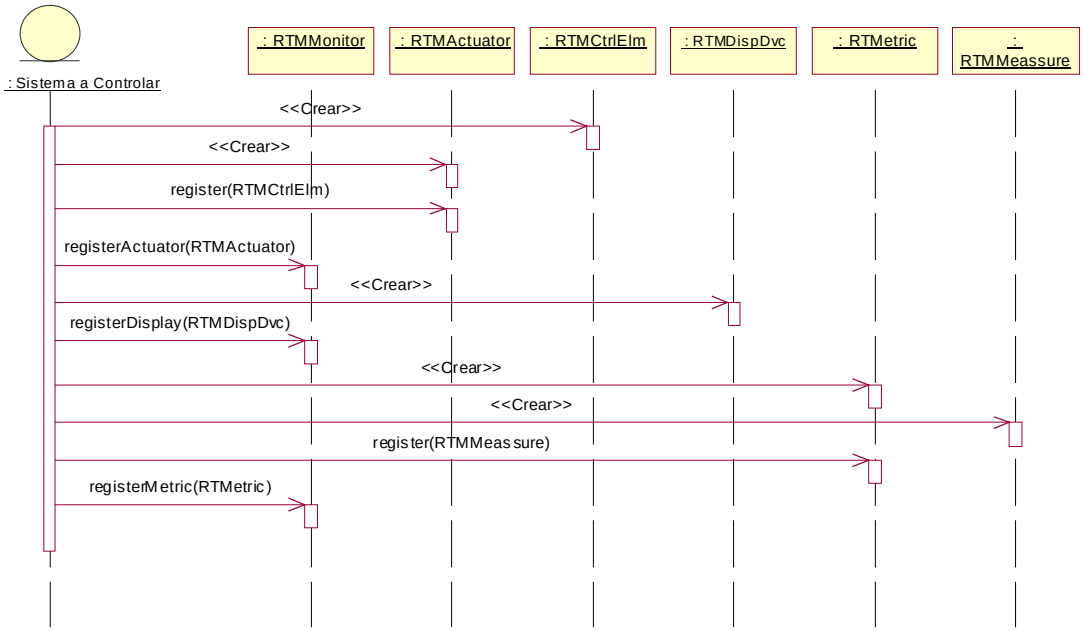


9.2.5. REGISTRAR OBJETOS EN EL MONITOR

Inicializado por la aplicación

Terminado por RTMMonitor

Se registrará un actuador, un elemento medido y un display relacionados.



9.3. DISEÑO

9.3.1. DISEÑO DE LAS INTERFACES (ES.UNIOVI.OOTLAB.METRICS.FRAMEWORK)

9.3.1.1. INTERFACE RTMACTUATOR

public interface RTMActuator

Los actuadores son elementos que hacen operaciones sonre el sistema. Operan para modificar el comportamiento del sistema de acuerdo a las mediciones tomadas.

Los objetos que controlan a otros objetos deben implementar esta interfaz.

Patrones de Diseño

RTMActuator se registra en RTMMonitor para monitorizar el comortamento y se relaciona con RTMEtric para obtener información del sistema. El elemento controlado debe ser registrado en RTMActuator para que este pueda actuar sobre él.

Ver también:

RTMetric, RTMMonitor, RTMCtrlElm

Métodos

void register(RTMCtrlElm ctrlElement)

Registra un RTMCtrlElm para ser controlado.

Detalle de los Métodos

register

public void register(RTMCtrlElm ctrlElement)

throws RTMException

Registra un RTMCtrlElm para ser controlado.

Parámetros:

ctrlElement - es el elemento a controlar.

9.3.1.2. INTERFACE RTMETRIC

Subinterfaces:

RTMComplex, RTMSimple

---

public interface RTMetric

Implementa cada métrica utilizada en el sistema.

Todas las métricas diseñadas en el sistema deben tener un objeto RTMetric para realizar las mediciones.

Patrones de Diseño

Implementa un composite junto con RTMSimple y RTMComplex.

También implementa un observer con RTMMonitor, siendo éste el elemento observado.

Envía un RTMElement para que éste devuelva una medida.

Ver también:

RTMSimple, RTMComplex, RTMMonitor, RTMElement

Métodos

void addMetric(RTMetric metric)

Añade una métrica cuando el objeto es un composite.

RTMMeasure currentValue()

Retorna la última medida observada en RTMElement.

RTMetric getChild(int pos)

Retorna un elemento RTMMetric que está en una posición concreta.

void register(RTMElement element)

Permite a un RTMElement identificarse en el objeto RTMetric.

void removeMetric(RTMetric metric)

Elimina una métrica del composite.

Detalle de los métodos

currentValue

public RTMMeasure currentValue()

Retorna la última medida observada en RTMElement.

Retorna:

Retorna un objeto RTMMeasure.

---

register

public void register(RTMElement element)

throws RTMException

Permite a un RTMElement identificarse en el objeto RTMetric.

Parámetros:

element - es el RTMElement a identificar.

---

addMetric

public void addMetric(RTMetric metric)

Añade una métrica cuando el objeto es un composite.

Parámetros:

metric - es el objeto a añadir.

---

removeMetric

public void removeMetric(RTMetric metric)

Elimina una métrica del composite.

---

getChild

public RTMetric getChild(int pos)

Retorna un elemento RTMMetric que está en una posición concreta.

Parámetros:

pos - es la posición en la cual está el objeto.

9.3.1.3. INTERFACE RTMSIMPLE

Superinterfaces:

RTMetric

---

public interface RTMSimple

extends RTMetric

Implementa todas las metricas simples que se utilicen en el sistema.

Patrones de Diseño

Implementa un composite junto con RTMetric y RTMComplex.

Ver también:

RTMetric, RTMComplex

Métodos heredados de es.uniovi.ootlab.metrics.framework.RTMetric

addMetric, currentValue, getChild, register, removeMetric

9.3.1.4. INTERFACE RTMCOMPLEX

Superinterfaces:

RTMetric

---

public interface RTMComplex

extends RTMetric

Implementa todas las métricas complejas que se utilicen en el sistema.

Patrones de diseño

Implementa un composite junto con RTMetric y RTMSimple.

Ver también:

RTMetric, RTMSimple

Métodos heredados de es.uniovi.ootlab.metrics.framework.RTMetric

addMetric, currentValue, getChild, register, removeMetric

9.3.1.5. INTERFACE RTMMEASURE

---

public interface RTMMeasure

Implementa la medida realizada.

En el caso mas simple puede ser numérico.

Patrones de Diseño

N/A.

Ver también:

RTMetric

Métodos

java.lang.Object getValue()

Retorna un objeto con la medida realizada.

Detalle de los Métodos

getValue

public java.lang.Object getValue()

Retorna un objeto con la medida realizada.

Retorna:

Retorna un Object.

9.3.1.6. INTERFACE RTMMONITOR

---

public interface RTMMonitor

Implementa el monitor que controla todo el sistema.

Es el elemento más importante del sistema y normalmente debe ser úncio.

Arranca RTMetric, lo lanza para hacer medida, informa a RTMDispDvc para colocar la información, arranca RTMActuator y establece la comunicación entre ellos y RTMetric para que colaboren.

Patrones de Diseño

Implementa un observer junto con RTMDispDvc's.

Ver también:

RTMetric, RTMActuator, RTMDispDvc

Métodos

void registerActuator(RTMActuator actuator)

Permite a los objetos RTMActutator ser identificados por el sistema.

void registerDisplay(RTMDispDvc display)

Permite a los objetos RTMDispDvc ser registrados como observers.

void registerMetric(RTMetric metric)

permite a los objetos RTMetric ser identificados por el sistema.

void run()

Arranca el sistema.

Detalle de los métodos

registerMetric

public void registerMetric(RTMetric metric)

permite a los objetos RTMetric ser identificados por el sistema.

Parámetros:

metric – es el objeto a ser identificado.

---

registerActuator

public void registerActuator(RTMActuator actuator)

Permite a los objetos RTMActutator ser identificados por el sistema.

Parámetros:

actuator - es el objeto a ser identificado.

---

registerDisplay

public void registerDisplay(RTMDispDvc display)

Permite a los objetos RTMDispDvc ser registrados como observers.

Parámetros:

display – es el objeto a ser registrado.

---

run

public void run()

Arranca el sistema.

9.3.1.7. INTERFACE RTMCTRLELM

---

public interface RTMCtrlElm

Los objetos controlados son aquellos sobre los que se realizan las acciones.

Cada objeto del sistema que deba ser controlado debe implementar esta interfaz.

Patrones de Diseño

N/A

Ver también:

RTMActuator

Métodos

void setAction(java.lang.Object action)

Realiza una acción ordenada por RTMActuator.

Detalle de los Métodos

setAction

public void setAction(java.lang.Object action)

Realiza una acción ordenada por RTMActuator.

Parámetros:

action - es la acción a ser realizada.

9.3.1.8. INTERFACE RTMDISPDVC

---

public interface RTMDispDvc

Emite la información en ficheros de log o en la pantalla.

Su responsabilidad es emitir la información sobre el sistema bajo la supervisión de RTMMonitor.

Cada elemento RTMCtrlDvc se debe registrar en RTMMonitor.

Patrones de Diseño

Implementa un observer de RTMMonitor.

Ver también:

RTMMonitor

Métodos

void displayMeasure(java.lang.Object par)

Realiza un display cuando lo decide RTMMonitor.

Detalle de los métodos

displayMeasure

public void displayMeasure(java.lang.Object par)

Realiza un display cuando lo decide RTMMonitor.

Parámetros:

par - es la información emitida.

9.3.1.9. INTERFACE RTMELEMENT

---

public interface RTMElement

Implementa el elemento a ser medido.

Es el punto de captura de mediciones en el sistema.

Patrones de Diseño

N/A.

Ver también:

RTMetric

Métodos

RTMMeasure getMeasure()

Realiza una medida sobre un elemento monitorizado.

Detalle de los métodos

getMeasure

public RTMMeasure getMeasure()

Realiza una medida sobre un elemento monitorizado.

Retorna:

Retorna un objeto RTMMeasure.

9.3.2. DIAGRAMA DE CLASES (INTERFACES)

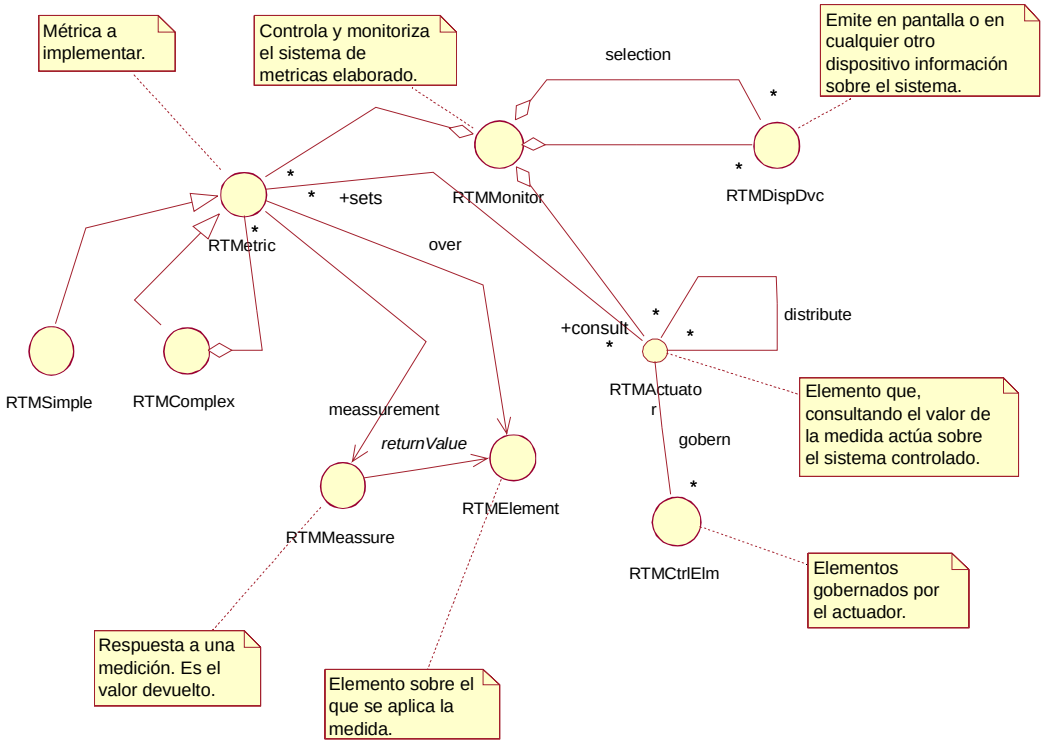


Ilustración 9-1. Diagrama de clases e interfaces del framework

9.3.3. DIAGRAMA DE PAQUETES

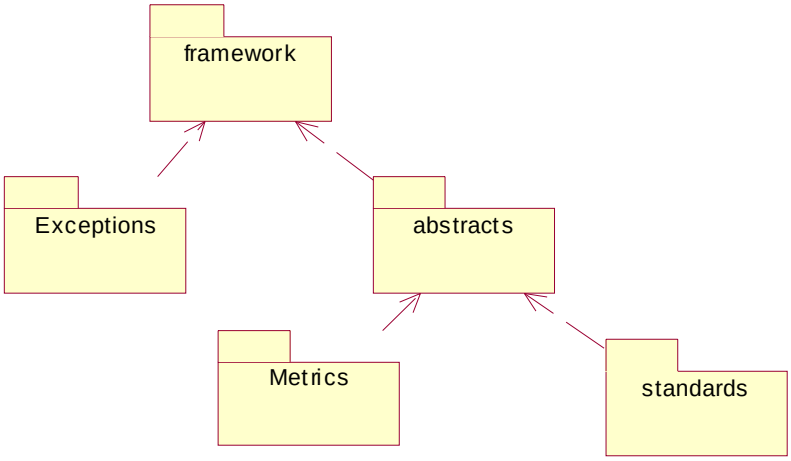


Ilustración 9-2. Diagrama de paquetes

10. APÉNDICE C. GLOSARIO

ACTUADOR Sistema que es capaz de intervenir en un objeto y modificar su estado.

ANÁLISIS En software, es una fase del ciclo de vida de la aplicación donde se identifican los requerimientos especificados por el cliente.

CICLO DE VIDA Son las fases por las que pasa un proyecto desde su origen hasta su conclusión (si es que existe). Hay distintos tipos de ciclo de vida, los dos más conocidos y usados son el clásico y el de prototipado.

CLASE En Orientación a Objetos, identifica un tipo de entidades modeladas a partir del mundo real. Una clase tiene atributos que identifican su estado y métodos que se aplican a esa clase.

CLASE ABS-TRACTA Clase que implementa alguna funcionalidad, pero en la que aún quedan interfaces que no están definidos. De este tipo de clases no es posible crear objetos.

CODIFICACIÓN (Ver programación)

CONTROL Posibilidad de interacción automática o manual en un sistema en evolución para dirigir el sistema hacia objetivos deseados.

CONTROLADOR Sistema que permite realizar acciones de control.

DERIVACIÓN Mecanismo que permite realizar herencia entre clases.

DIAGRAMA DE CLASES Diagrama que representa la estructura estática de clases que implementa un sistema.

DIAGRAMA DE ESTADOS Diagrama que representa el comportamiento dinámico de una clase y que se estructura en estados y transiciones.

DISEÑO (Ver Diseño de Software)

DISEÑO DE SFOTWARE En software, es una fase del ciclo de vida de la aplicación en que se adaptan a las disponibilidades tecnológicas los análisis realizados.

ESCALA Graduación para medir los efectos de diversos instrumentos. Hay distintos tipos de escala: nominales, ordinales, de intervalo, etc...

ESFUERZO En el contexto de este documento el esfuerzo es una variable mediante la cual medimos el tiempo que un programador medio necesita para completar una tarea.

FORMAL Se usa en concepto formal calificando a un procedimiento o a una actividad cuando éstas están avaladas por una teoría probada, de manera que en este momento el procedimiento o actividad resulta no ambiguo, está sujeto a reglas y puede ser predicho.

INGENIERÍA DEL SOFTWARE Conjunto de técnicas que permiten realizar aplicaciones siguiendo metodologías y estándares que garanticen la calidad del software

INTERFAZ Conjunto de métodos que responden a un comportamiento específico detectado durante la fase de diseño y que compone un role de comportamiento.

LENGUAJE DE PROGRAMACIÓN Lenguaje que permite codificar un diseño para, una vez traducido al lenguaje de la máquina, obtener un producto software ejecutable.

FRAMEWORK Esquema de funcionamiento de un sistema o de una parte de un sistema que permite la personalización para implementaciones concretas.

MEDICIÓN (*En inglés measurement*) La teoría de la medición es una disciplina filosófica. La medición se define en el estándar BSI (BS 5233) como «*las operaciones que tienen por objeto la determinación del valor de una cantidad*».

MEDIDA (*En ingles measure*) La teoría de la medida es una disciplina matemática. La definición depende de la teoría utilizada. Expresión numérica del resultado de medir una cosa.

MEDIDA DE SOFTWARE Medidas que se realizan sobre los procesos de la ingeniería del software, tanto en las fases de diseño como en las fases de explotación.

MÉTRICA Una métrica es un elemento matemático que debe tener una serie de propiedades específicas.

MÉTRICA DE SOFTWARE (*En inglés metric*) En lo referente al software podemos considerar las métricas como las medidas (en términos de cantidad) relacionadas con los productos software y con los procesos de producción, mantenimiento y explotación del software.

METROLOGÍA (*En inglés metrology*) Es la ciencia que estudia los sistemas de medición y medida. Según la definición del estándar BSI (BS 5233), metrología es «*el campo de conocimiento que concierne a la medida*».

MODELO Es una abstracción de la realidad. El hombre para poder abordar ciertos problemas muy complejos necesita abstraer de ellos aquellos aspectos que son fundamentales y obviar aquellos que aportan poca o nula información.

MONITOR Elemento que permite realizar la monitorización.

MONITORIZACIÓN Supervisión del comportamiento de un sistema que permite obtener información de su evolución en el tiempo.

OBJETO En Orientación a Objetos, identifica una instanciación de una clase. Es el modelo correspondiente de una entidad concreta del mundo real. Los valores de los atributos están instanciados para un objeto y no así (en general) para una clase.

PROGRAMACIÓN En software, es la fase del ciclo de vida de la aplicación en que las partes del proyectos identificadas como productos software se codifican en los lenguajes de programación indicados. A esta fase también se la conoce como codificación.

PROYECTO Es un plan para conseguir un objetivo. Un proyecto software convencional es un plan para realizar una aplicación de acuerdo a las especificaciones de su usuario final.

PRUEBAS En software, son las fases del ciclo de vida de la aplicación en que se testan los productos ya elaborados. Hay distintos tipos de prueba y en distintas fases del ciclo de vida. Las más comunes son las pruebas unitarias, de integración y las finales.

THREADS Son subprogramas que corren en paralelo al programa principal. Se suele hablar de hilos de ejecución.

11. APÉNDICE D. HERRAMIENTA JAVARTM

11.1. INTRODUCCIÓN

Dentro del alcance de esta tesis se ha desarrollado el prototipo de una herramienta para soportar la traducción de JavaRTM a Java: JRTMCompiler.

Al tratarse de un prototipo tiene funcionalidad limitadas por lo que respecta al soporte de la totalidad el framework, pero si soporta la traducción de las clases a las que el framework se conecta.

11.2. DESCRIPCIÓN

JRTMCompiler es una herramienta para soportar la precompilación a Java de las clases generadas por la metodología DMT.

A partir de los ficheros (`*.jrtm`) que contienen las descripciones de las clases en JavaRTM, la herramienta genera los ficheros fuentes de Java (`*.java`) y los ficheros de traza del proceso de precompilación (`*.tlog`).

Se trata de una precompilación en dos pasos, ya las definiciones deben ser generadas en el primer paso, junto con una tabla de símbolos que permitan, en el segundo paso la inserción de código en diferentes puntos de la clase final en Java.

En los ficheros `.tlog` se tiene una descripción del proceso y un volcado de la tabla de símbolos generada.

11.3. GUÍA DE USO

La herramienta presenta al usuario una interfaz para navegar por las unidades de disco del sistema en busca de los ficheros a precompilar (Ver Ilustración 11-2).

Una vez elegido el fichero a compilar se pulsa sobre el botón “Compile to Java” (ver Ilustración 11-1) y el proceso de compilación se aplica al fichero elegido.

Ilustración 11-1. Botones de compilar y salir de la aplicación.

En la barra de estado se informa al usuario del proceso y de la posibilidad de errores de compilación.

Para salir de la aplicación se pulsa el botón “Exit”.

Ilustración 11-2. Interfaz principal de la herramienta JRTMCompiler

Se puede elegir la inspección de diferentes tipos de ficheros relacionados con DMT (ver Ilustración 11-3):

- Ficheros fuente de JavaRTM (*.jrtm)
- Ficheros fuente de Java (*.java)
- Ficheros de traza de compilaciones (*.tlog)

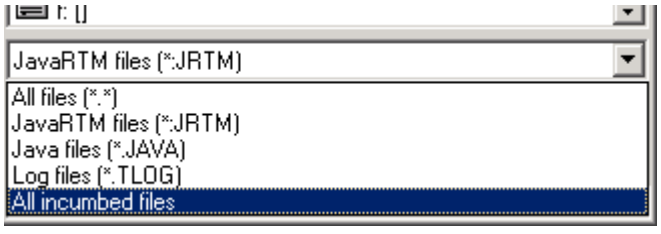


Ilustración 11-3. Diferentes tipos de ficheros que se pueden seleccionar.

En el siguiente apartado se muestra un ejemplo de compilación basado en el caso de estudio 6.

11.4. EJEMPLO

Para este ejemplo se parte de las clases `ConectorElement` y `ComposerCtrlElm` definidas en el caso de estudio 6 y que son respectivamente las clases a medir y controlada por el sistema.

El listado completo de dichas clases se muestra en los párrafos siguientes.

Clase `ConnectorElement`, del fichero `ConectorElement.jrtm`:

```
package es.uniovi.ootlab.metrics.demo.demoTimer;

import es.uniovi.ootlab.metrics.demo.demoBase.Connector;
import es.uniovi.ootlab.metrics.framework.*;
import es.uniovi.ootlab.metrics.metrics.*;

// Class ConnectorElement implementa
// el canal de conexión

public class ConnectorElement extends Connector {
   //# metrics begin:      A0001
   //# metric class:       FailController  ## Clase RTMetric
   //# measure class:      StRTMInteger    ## Clase RTMMEassure
   //# element code:
   //#> {
   //#>    return new StRTMInteger(fLong-fShort);
   //#> }
   //# monitor class single: StRTMMonitor ## Clase RTMMonitor
   //# display class:        Display      ## Monitoriza alarma
   //# actioner class:       Actuator     ## clase actioner
   //# controlled class:     Composer     ## clase controlada
   //# metrics end:          A0001

   int
      fLong = 0,
      fShort = 0;

   public void send(char [] msg)
   {
      super.send(msg);
      if (failLong())
         fLong++;
      else if (failShort())
         fShort++;
   }

   public Connector(double per, Receiver rec)
   {
      super(per,rec);
   }
}
```

Clase `ComposerCtrlElm`, del fichero `ComposerCtrlElm.jrtm`:

```
package es.uniovi.ootlab.metrics.demo.demoTimer;

import es.uniovi.ootlab.metrics.demo.demoBase.Composer;
import es.uniovi.ootlab.metrics.framework.*;
import es.uniovi.ootlab.metrics.metrics.*;

// Class ComposerCtrlElm implementa el sistema de
// generación de mensajes que puede ser controlado

public class ComposerCtrlElm extends Composer {
   //# metrics begin:   A001
   //# element class:   Connector  ## clase element
   //# actioner class:  Actuator   ## clase que realiza el control
   //# actioned code:
   //#> {
   //#>    modLonMsg(((Double)param).doubleValue());
```

```
   //#> }
   //# metrics end:     A001

   public ComposerCtrlElm(int lon)
   {
      super(lon);
   }
}
```

Al compilarlas mediante la herramienta JRTMCompiler se obtienen los ficheros siguientes.

Fichero `ConnectorElement.java`:

```
package es.uniovi.ootlab.metrics.demo.demoTimer;

import es.uniovi.ootlab.metrics.demo.demoBase.Connector;
import es.uniovi.ootlab.metrics.framework.*;
import es.uniovi.ootlab.metrics.metrics.*;

public class ConnectorElement extends Connector  implements RTME-
lement {

   synchronized public void setAction(Object param)
   {
      return new StRTMInteger(fLong-fShort);
   }

   int
      fLong = 0,
      fShort = 0;

   public void send(char [] msg)
   {
      super.send(msg);
      if (failLong())
         fLong++;
      else if (failShort())
         fShort++;
   }

   public Connector(double per, Receiver rec)
   {
      super(per,rec);
   }
}
```

Se ha resaltado en negrita el código generado durante la compilación y que corresponde a la definición del grupo `METTRICS A001` definido en el fichero `ConnectorElement.jrtm`.

Por lo que respecta al segundo fuente, genera el fichero `ComposerCtrlElm.java`:

```
package es.uniovi.ootlab.metrics.demo.demoTimer;

import es.uniovi.ootlab.metrics.demo.demoBase.Composer;
import es.uniovi.ootlab.metrics.framework.*;
import es.uniovi.ootlab.metrics.metrics.*;

public class ComposerCtrlElm extends Composer  implements
RTMCtrlElm {

   synchronized public void setAction(Object param)
   {
      modLonMsg(((Double)param).doubleValue());
   }

   public ComposerCtrlElm(int lon)
```

```
   {
      super(lon);
   }
}
```

También en este caso se ha resaltado en negrita el código generado durante la compilación.

Además se han generado don ficheros con a traza de la compilación, que se pueden ver en el CD que acompaña a esta tesis. En los ficheros de traza se refleja el proceso de compilación, los dos pasos del compilador, el código que genera y el que deja invariante y un display completo de la tabla de símbolos.

Si durante la compilación se generasen errores, éstos se reflejarían en el fichero de traza, como se puede ver en el párrafo siguiente, que ha sido generado a partir de un fichero fuente con errores:

```
FICHERO DE TRAZA.

PRIMER PASO...

Línea de código nativo Java.
METRICS BEGIN : A001

<<< E R R O R >>> '//#>' mal colocado. No hay ningún elemento de
código iniciado.
CODIGO> Double par;
ELEMENT CLASS : Connector

<<< E R R O R >>> Etiqueta desconocida: CLAS:

<<< E R R O R >>> Etiqueta desconocida: ELEMEN

<<< E R R O R >>> Faltan ':'
ACTIONER CLASS : ##
ACTIONED CODE : [IMPLEMENTS RTMCrtlElm]
CODIGO>   {
.......

SIMBOLO (4):
Nombre : A001
Tipo : METRICS END
FIN SIMBOLO (4).

FIN DE LA TABLA DE SÍMBOLOS.

HAY 4 ERRORES


FIN DEL FICHERO DE TRAZA.
```

Como se puede comprobar cada error se genera en la posición en que se detecta en el código fuente de JavaRTM y al final se ha hecho un resumen del número de errores detectados.

También se puede ver en este ejemplo de fichero de traza, que por cada línea de código Java invariante, se genera una línea del fichero de traza indicando que en esa línea había código nativo.

Por otro lado se puede ver, también, la información que se guarda en la tabla de símbolos sobre un elementos de JavaRTM (en este caso sobre METRICS END).

12. APÉNDICE E. ÍNDICE ALFABÉTICO