

# Estimación de costes de un proyecto informático

Métricas de productividad y modelos  
de estimación de costes

Miquel Barceló García

P03/75069/02142



# Índice

<b>Introducción</b>	5
<b>Objetivos</b>	7
<b>1. Métricas del software</b>	9
1.1. Métricas del producto y del proceso	10
1.2. Métricas de calidad y de productividad	11
1.3. El esfuerzo y la medida de la productividad	13
1.3.1. Los problemas terminológicos del hombre-mes	13
1.3.2. El mito del hombre-mes	14
1.4. Métricas orientadas al tamaño y a la función	15
1.4.1. Las líneas de código	16
1.4.2. Los puntos de función	21
1.4.3. Relación entre puntos de función y líneas de código	26
1.4.4. Una perplejidad final	28
<b>2. Estimación de costes de un proyecto informático</b>	29
2.1. Momento de la estimación y grado de exactitud	30
2.2. Los métodos de estimación posibles según Barry W. Boehm	31
2.3. Modelos de estimación	33
2.3.1. Modelos históricos	35
2.3.2. Modelos empíricos o estadísticos	36
2.3.3. Modelos teóricos: el modelo de Putman	38
2.3.4. Modelos compuestos: los modelos COCOMO de Barry W. Boehm	40
2.3.5. El uso de estándares de productividad	44
2.4. Práctica real de la estimación	47
2.4.1. Un ejemplo: contabilidad sencilla en un PC	49
<b>Resumen</b>	53
<b>Actividades</b>	55
<b>Ejercicios de autoevaluación</b>	55
<b>Solucionario</b>	56
<b>Glosario</b>	56
<b>Bibliografía</b>	58



## Introducción

Cuando se inicia un proyecto informático se conocen muy pocas de las funcionalidades que se deben implementar. A pesar de todo, a veces debe realizarse una estimación de los costes necesarios para construir un *software* de aplicación, del cual, al empezar, no se conoce ni siquiera el alcance que pueda llegar a tener.

De hecho, un *software* se especifica a medida que se construye (o, dicho de otra manera, la especificación o el análisis es una de las primeras etapas de las que consta el complejo proceso de construcción de *software* de aplicación en la informática de gestión). Sin embargo, aunque esto sea así, a menudo antes de empezar se debe llevar a cabo una primera estimación de costes que, simplemente, sea objetiva y que tenga unas mínimas posibilidades de convertirse en realidad.

Por tanto, debemos disponer, en primer lugar, de diferentes unidades de medida que nos puedan ser útiles para caracterizar y medir varias funcionalidades del *software* que se quiere implementar: el tamaño del proyecto, la evaluación de la calidad y la fiabilidad, etc. Y, además, necesitamos unidades de medida que nos permitan asociar estas funcionalidades con el esfuerzo que cuesta implementarlas; es decir, debemos tener referencias sobre la productividad en la construcción de *software*.

En este módulo didáctico analizamos algunas de las muchas métricas del *software*, con atención especial a las métricas de tamaño (LOC), funcionalidades (puntos de función) y productividad (esfuerzo), que son las más adecuadas para llevar a cabo una buena estimación de los costes de construcción de un *software* de aplicación determinado.

En la construcción de *software* interviene un conjunto de costes muy variado, sin embargo, tal como se menciona en el módulo “El proyecto informático de construcción de *software*”, dominan los costes del personal técnico que interviene en la construcción del *software*, a menudo caracterizados como jefes de proyecto, analistas y programadores. Por tanto, estimar el trabajo de los diferentes profesionales involucrados en un proyecto en horas o días es la manera habitual de considerar los costes más relevantes en el proceso de construcción de *software* de aplicación.

Sin embargo, el hecho de disponer de unidades de medida no es suficiente. Es necesario también conocer los diferentes modelos que, desde hace unos treinta años, se utilizan para estimar los costes de un proyecto informático.

Cabe mencionar que algunos de estos modelos están prácticamente obsoletos y no son nada adecuados para la informática de nuestros días. De hecho, du-

rante la última década, los cambios en la construcción del *software* han sido tantos y tan rápidos que, a menudo, los modelos de estimación disponibles no parecen los más adecuados para la compleja y difícil tarea de estimar los costes. Es cierto que se revisan los modelos más importantes, pero la falta de datos concretos y generalizables provoca que todavía no se hayan sustituido del todo los modelos tradicionales.

En este módulo, además de exponer las métricas o las unidades de medida más habituales, presentamos de manera crítica los diferentes modelos de estimación que existen y que forman el ámbito de conocimiento que debe tener un buen jefe de proyecto sobre el tema de la estimación de costes.

El módulo termina con un ejemplo práctico y concreto sobre el proyecto de construir una pequeña aplicación de contabilidad.

## Objetivos

En este módulo presentamos la primera de las etapas de la gestión de un proyecto informático: la estimación previa de los posibles costes en la construcción de *software* de aplicación en la informática de gestión. Con el estudio de los materiales asociados a este módulo didáctico, el estudiante debe poder alcanzar los siguientes objetivos:

1. Conocer el problema de las métricas de *software*, sobre todo las que sirven para medir el tamaño de un proyecto, las funcionalidades que se quieren implementar y, especialmente, la productividad que se puede alcanzar.
2. Comprender lo que se denomina el *mito del hombre-mes*, es decir, las dificultades asociadas a la unidad más habitual utilizada para medir el esfuerzo humano en la construcción de *software* (por ejemplo, la persona-mes). Aunque se trata del producto de personas por meses, es necesario saber que no son factores intercambiables, ya que no se pueden efectuar todo tipo de combinaciones y debe darse un reparto concreto y específico del esfuerzo a lo largo de los meses del proyecto.
3. Conocer las características, las ventajas y los inconvenientes de las métricas de tamaño (líneas de código) y funcionalidad (puntos de función) y su posible interrelación.
4. Conocer la orden de magnitud de las ratios habituales de productividad en la construcción de *software* de aplicación en la informática de gestión.
5. Comprender que el grado de incertidumbre en la estimación de costes de un proyecto informático es francamente elevado cuando empieza el proyecto, pero a medida que éste avanza se puede reducir del todo.
6. Conocer de manera crítica los diferentes modelos de estimación de costes en la construcción de *software* que se han utilizado en los últimos años, las características y los puntos fuertes y débiles que tienen.
7. Entender un posible proceso práctico de estimación de costes con un sistema ascendente (*bottom-up*) a partir de una descomposición del proyecto en varias tareas y la estimación directa de cada tarea por separado.





## 1. Métricas del *software*

Para efectuar una estimación correcta de los costes de un proyecto informático de construcción de *software*, se debe disponer de unidades de medida que nos permitan relacionar las diferentes actividades de la construcción del *software* con el esfuerzo en horas de trabajo o con el coste monetario que representa tener su construcción. Ésta es una cuestión que se intenta resolver a partir de las diferentes métricas de productividad o, si se quiere, de las diversas unidades con las que se puede medir la construcción del *software*.

De hecho, medir el *software* no es en absoluto una actividad sencilla. Se puede decir que toda magnitud física se puede medir, pero el *software* es, sobre todo, el resultado de una actividad básicamente intelectual, tanto en el origen como en el resultado final. De hecho, la dificultad de medir el *software* ha provocado el nacimiento de varias métricas o unidades de medida que intentan cuantificar diferentes aspectos que pueden ser relevantes en el *software*.

Una **métrica** es, pues, una asignación de valor a un atributo de una entidad propia del *software*, ya sea un producto o un proceso.

En esta definición de métrica se incluyen tres conceptos que conviene especificar de manera diferenciada:

- Cuando hablamos de un **atributo** nos referimos a una determinada característica que pretendemos medir y cuantificar\*.
- Cuando hablamos de un **producto** nos referimos, por ejemplo, a un código, un diseño, una especificación, etc.
- Cuando se habla de un **proceso** se hace referencia a una etapa de la construcción del *software* y a la manera de llevarlo a cabo: un diseño, unas pruebas, etc.

\* Como podrían ser el esfuerzo, la fiabilidad, la mantenibilidad, la calidad o el tamaño.


Reproducimos a continuación una definición estricta de lo que es una métrica de *software* y que podéis encontrar en el *Standard Glossary of Software Engineering Terms* del IEEE.

Una **métrica de *software*** es una medida cuantitativa del grado en el que un sistema, un componente o un proceso posee un determinado atributo.

Existe una variedad impresionante de métricas de *software* y, de hecho, muchos autores han ido sugiriendo diferentes unidades de medida para evaluar y cuantificar diversos aspectos considerados importantes en el *software*.

En general, se puede decir que las diferentes métricas intentan evaluar tres grandes magnitudes generales: la **calidad** y el **tamaño** (a menudo del producto) y la **productividad** (frecuentemente del proceso de construcción del producto), aunque lo hacen desde muchos puntos de vista diferentes.

A menudo las métricas se utilizan para evaluar un determinado aspecto del *software* que ya existe, pero también para estimar *a priori* determinadas magnitudes de un *software* que aún está por elaborar. Cabe decir, de entrada, que no se puede llevar a cabo una estimación posible de proyectos informáticos futuros si no se dispone, *desde el inicio*, de los datos obtenidos en la evaluación de proyectos anteriores.


En el momento de realizar la estimación de costes, las métricas de productividad son las que tienen más relevancia en la calificación de un proyecto informático. Por ello, aquí nos centraremos muy especialmente en dichas métricas. Como veremos, se dan diversos modelos para efectuar esta estimación, pero, evidentemente, todos se basan, de una manera o de otra, en los datos obtenidos en la evaluación de proyectos anteriores. La elección de una métrica en concreto suele ser un aspecto primordial en los diferentes modelos de estimación. 

Podéis ver diferentes modelos de estimación de costes en el apartado 2 de este módulo didáctico.

Cabe mencionar que, a pesar de disponer de métricas bien escogidas e incorporadas a un proceso de construcción de *software*, continúa habiendo problemas.

En definitiva, se debe tener un criterio para adoptar y utilizar una métrica de *software*. Como siempre que se mide, el hecho de disponer de unidades de medida e, incluso, de estándares, no aleja, más bien al contrario, la inevitable necesidad de pensar. Disponer de métricas puede ser una ayuda, pero no lo es todo.

### 1.1. Métricas del producto y del proceso

Una primera subdivisión de las muchas métricas que existen las clasifica en los dos tipos siguientes: 


1) Las **métricas del producto**, que miden diferentes aspectos del *software* obtenido, a menudo a partir de código fuente expresado en un lenguaje informático determinado\*.

#### La falibilidad de las métricas

La aceptación, que es muy frecuente, de las líneas de código como unidad de medida del tamaño de un proyecto informático no evita errores de apreciación (considerar o no los comentarios o las líneas en blanco), de recuento (cada vez hay más líneas de código generadas de manera automática por las herramientas de desarrollo), de generalización (para un mismo tratamiento, se necesita un número diferente de líneas de código en función del lenguaje de programación utilizado), etc.

\* Lenguajes de programación, de diseño, de especificación, etc.

2) Las **métricas del proceso**, que intentan medir determinados atributos que hacen referencia al entorno de desarrollo del *software* y tienen en cuenta la manera de construirlo.

A menudo, las métricas se pueden utilizar solas o combinadas, es decir, como indicadores que permiten proporcionar una visión resumida del producto de *software* o del proceso de su construcción: 

1) Los **indicadores de producto** referidos, evidentemente, al *software* obtenido pueden servir para evaluar la calidad del proceso de construcción de *software*, siempre de manera comparativa con estándares de mercado o con niveles alcanzados en otros productos.

Cabe mencionar que los indicadores de producto a menudo no están realmente disponibles hasta que no lo está el producto en sí, es decir, *a posteriori*, cuando el *software* de aplicación ya se encuentra en disposición de entrar en la etapa de explotación.

2) Los **indicadores de proceso** han de permitir conocer la eficacia de una instalación simplemente por comparación con los indicadores que es capaz de conseguir respecto de los que la literatura técnica publica.

En la última década se ha puesto de moda crear, en el marco de lo que se denomina la *madurez del proceso del software*<sup>\*</sup>, al menos en las grandes instalaciones, el *programa de métricas de software*<sup>\*\*</sup>, que, una vez decididas unas determinadas unidades de medida, intenta introducir en el proceso de desarrollo y construcción de *software* la buena práctica de medir diferentes aspectos y cuantificar al máximo posible buena parte de los aspectos de la gestión de la construcción de *software* de aplicación. El hecho de disponer de estas informaciones objetivas es de gran ayuda en la calificación de proyectos informáticos futuros.

#### Como indicador de producto...

... se puede utilizar, por ejemplo, el número de errores de un *software*, aunque es también un indicador claro de la calidad del proceso de construcción utilizado para obtener este *software*.

#### Como indicador de proceso...

... si se utiliza, por ejemplo, el número de líneas de código por persona y mes, como la cifra más o menos estándar publicada en los años noventa sería de 350 líneas de código por persona y mes en un proyecto medio, una instalación que sólo consiguiera 200 sabría que está todavía lejos de lo que debería ser factible en el estado actual del arte.

\* En inglés, *Software Process Maturity*.

\*\* En inglés, *Software Metrics Program*.

## 1.2. Métricas de calidad y de productividad

Las diferentes unidades de medida para cuantificar el *software* intentan determinar sus características, que se pueden resumir en calidad del producto y productividad del proceso.

La **calidad de un producto**, según la definición del estándar ISO 8402, es el conjunto de funcionalidades y características de un producto o servicio que se centran en su capacidad de satisfacer las necesidades, ya sea implícitas o bien explicitadas claramente, de un cliente o usuario.

En el caso de la informática de gestión, a menudo estas necesidades provienen de un contrato entre el cliente y el proveedor de *software*, aunque, como se

dice en otro módulo, es muy difícil que todas las necesidades queden explicadas convenientemente en los requerimientos.

Las **métricas de calidad** se asocian a unos determinados atributos medibles, no siempre evidentes, para reconocer la presencia o ausencia de calidad.

A menudo, las métricas de calidad forman parte de lo que se denominan *medidas indirectas del producto*, junto con las métricas de funcionalidad, complejidad, eficiencia, fiabilidad, facilidad de mantenimiento y tantas otras capacidades en cierta manera difíciles de medir objetivamente.

Las **métricas de productividad** recogen la eficiencia del proceso de producción de *software* y relacionan el *software* que se ha construido con el esfuerzo que ha costado elaborarlo.

En el caso de la productividad, es mucho más fácil encontrar medidas directas, como las diferentes maneras de determinar el tamaño del producto obtenido (las líneas de código o los puntos de función, por ejemplo) relacionándolo con el tiempo y/o el esfuerzo que ha costado obtenerlo.

#### Las unidades de medida como indicadores

A menudo las diferentes unidades de medida se combinan en indicadores resumidos, como ejemplos podemos encontrar, entre otros, los siguientes:

- Líneas de código por persona y día (indicador de productividad).
- Horas para implementar un punto de función (indicador de productividad).
- Número de errores por cada mil líneas de código (indicador de calidad).
- Pesetas por cada millar de líneas de código (indicador de coste).
- Páginas de documentación por cada mil líneas de código (indicador de documentación).

En la terminología del estándar de métricas de productividad de *software* del Instituto de Ingenieros Eléctricos y Electrónicos (*IEEE Standard for Software Productivity Metrics*) de 1993, se habla de **primitiva** para hacer referencia al nivel más bajo en el que se recogen los datos. Evidentemente, podemos encontrar primitivas de dos tipos:

- **De salida\***, que recogen las características finales del *software*.
- **De entrada\*\***, que miden lo que ha sido necesario tener y utilizar para construir el *software*.

Una **ratio de productividad** es una relación que se suele establecer para tener en cuenta la productividad entre las primitivas de salida, es decir, las unidades que miden la salida del proceso de construcción de *software* y las primitivas de entrada, que miden las entradas en el proceso de construcción de *software*.

#### Algunas medidas directas...

... serían las líneas de código, el coste, el esfuerzo, el número de errores, la velocidad de ejecución, la memoria ocupada por un programa, los puntos de función de Albrecht, etc.

\* En inglés, *output primitives*.  
\*\* En inglés, *input primitives*.

#### Primitivas de entrada y de salida


Algunos ejemplos de primitivas de salida son las líneas de código, el número de errores, los puntos de función, el número de páginas de documentación, etc.

Por otra parte, ejemplos de primitivas de entrada son el esfuerzo en horas de trabajo de una persona, el coste en pesetas, etc.

### 1.3. El esfuerzo y la medida de la productividad


Con vistas a tratar la productividad en el proceso de construcción de *software* de aplicación, independientemente de cuáles sean las primitivas de salida utilizadas (generalmente, las líneas de código o los puntos de función que estudiaremos más adelante), las primitivas de entrada intentan a menudo medir el esfuerzo o el coste que representa el hecho de construirlo. La productividad se expresa finalmente en una ratio que relaciona las primitivas de entrada y de salida.

Podéis ver las líneas de código y los puntos de función en el subapartado 1.4 de este módulo didáctico.

Como se explica en otro módulo, casi siempre el coste se asocia o es proporcional al esfuerzo que supone el trabajo, ya que los recursos humanos que intervienen en un proyecto informático son, hoy en día, los más importantes. Aunque conviene no olvidar que pueden darse otros costes para el *hardware*, el *software* de desarrollo, etc., a menudo se hace la simplificación de asociar coste y esfuerzo en un proyecto informático, aunque se expresen en unidades diferentes. 

Podéis ver los recursos de un proyecto informático en el subapartado 1.4 del módulo "El proyecto informático de construcción de *software*" de esta asignatura.

La **medida habitual del coste** es, evidentemente, la cantidad de dinero que cuesta producir un *software* determinado; mientras que las **medidas habituales del esfuerzo** se reducen siempre al trabajo que lleva a cabo un profesional en una determinada unidad de tiempo: un día (persona-día), un mes (persona-mes) o un año (persona-año).

Cabe mencionar que, como siempre que se efectúan generalizaciones de este tipo, se piensa en un profesional de cualificación media, con una buena formación o una experiencia adecuada para la tarea que debe realizar y una capacidad de trabajo media. Es evidente que en un equipo de proyecto intervienen todo tipo de profesionales, algunos más eficientes o más motivados que otros. El jefe de proyecto ha de tener muy en cuenta estas particularidades concretas, sobre todo a la hora de repartir las tareas que se deben llevar a cabo. Sin embargo, en el tratamiento general de las métricas de productividad se piensa en un profesional genérico con capacidades, motivación y dedicación medias. 

#### 1.3.1. Los problemas terminológicos del hombre-mes

En los proyectos informáticos se han utilizado diferentes denominaciones de la unidad de esfuerzo, como las siguientes:

1) En primer lugar, se habló de hombre-mes como la tarea que llevaba a cabo un hombre durante un mes de trabajo. Ésta es la denominación habitual, que se puede encontrar a menudo abreviada con la sigla MM, proveniente de la denominación inglesa *man-month*.

##### La denominación hombre-mes...

... se utiliza en libros del todo clásicos, como el de Brooks (*The Mythical Man-Month*), a partir de la experiencia de uno de los primeros grandes proyectos informáticos de construcción de *software*: el sistema operativo 360 de IBM en los años sesenta.

2) Más adelante, pareció que la denominación era claramente sexista y se buscaron otros nombres como éstos:

- **Persona-mes:** un mes de trabajo de una persona del equipo con independencia del género.
- **Staff-month:** un mes de trabajo de una persona del equipo de proyecto\*.
- **Engineering-month:** un mes de trabajo en la actividad de ingeniería del *software*, que incluye, como ya sabemos, el análisis, el diseño, la programación y las pruebas para producir una determinada aplicación o un sistema de información.

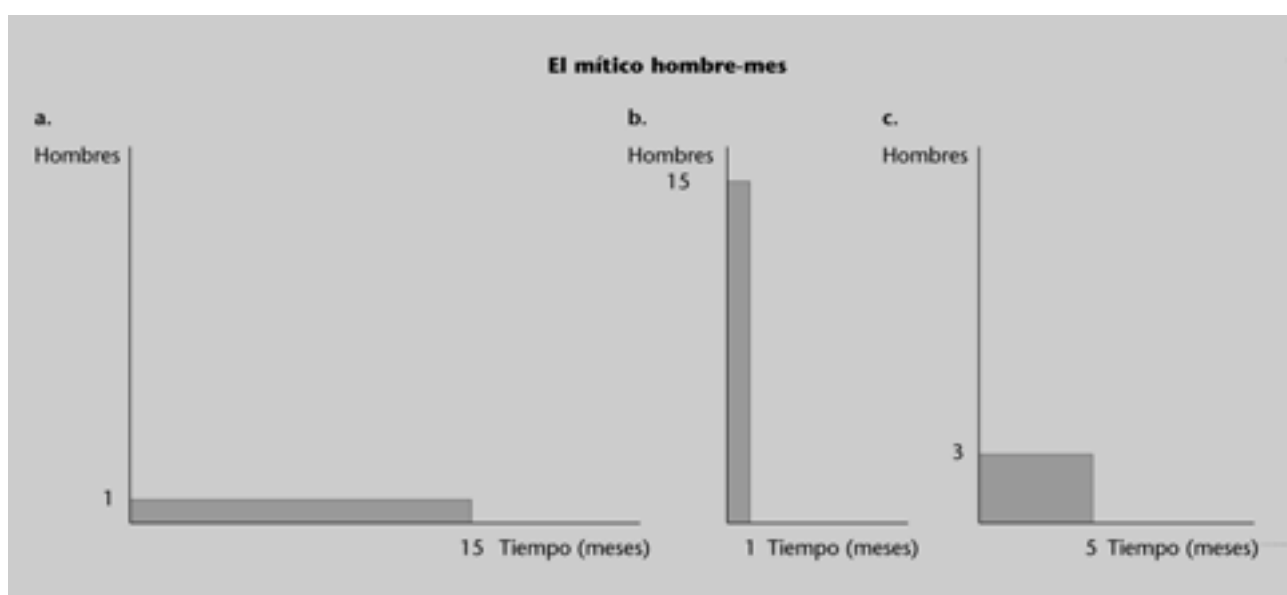
\* En inglés, *staff*.

Podéis ver las etapas de la actividad de ingeniería de *software* en el subapartado 1.8 del módulo "El proyecto informático de construcción de *software*".

Desgraciadamente, la tradición se mantiene y es muy habitual oír hablar todavía de un proyecto informático de, por ejemplo, 50 hombres-mes, término que la práctica profesional parece haber estabilizado en nuestro entorno. En este texto, hechas estas aclaraciones previas, seremos más bien eclécticos en la denominación de esta unidad de medida del esfuerzo en un proyecto informático. !

### 1.3.2. El mito del hombre-mes

Los problemas que plantea la unidad persona-mes no proceden sólo de querer usar un lenguaje políticamente correcto. Inconscientemente, una unidad que es el producto de dos (número de personas y meses) sugiere que debe ser posible intercambiar personas y meses. Esto no es cierto, y no saberlo es uno de los errores en los que puede incurrir el jefe de un proyecto informático.



Por ejemplo, la figura anterior muestra diferentes maneras de imaginar un esfuerzo de 15 hombres-mes. Este esfuerzo se puede alcanzar tanto con una per-

sona que trabaje durante quince meses, como con quince personas que trabajen conjuntamente durante un mes. O, también, por ejemplo, con tres personas que trabajen durante cinco meses.

Matemáticamente es cierto y, en los tres casos, el producto es el mismo:

$$1 \cdot 15 = 15 \cdot 1 = 3 \cdot 5 = 15.$$

Pero en la práctica de la construcción de *software* este intercambio no es cierto. Posiblemente la tercera opción está más próxima a la realidad que las otras dos.

Éste es el posible error de interpretación que quiere ayudar a corregir el artículo más importante de los que Brooks recoge en la obra ya histórica *The Mythical Man-Month*.

En determinados trabajos se da una distribución del esfuerzo a lo largo del tiempo que les es, en cierta manera, característica. Esto ocurre en la construcción de *software* y es necesario saberlo.

#### Lectura recomendada

Podéis consultar el trabajo de Brooks sobre el mito del hombre-mes en la obra siguiente:

F. Brooks (1975). *The Mythical Man-Month*. Reading (Massachusetts): Addison-Wesley.

#### Ejemplo de distribución característica del esfuerzo a lo largo del tiempo

La distribución del esfuerzo de construcción de *software* se puede comparar con la tarea de traer a un ser humano al mundo, proceso que, habitualmente, requiere el esfuerzo de 9 meses- mujer y admite sólo una distribución temporal muy clara: una mujer con un embarazo de nueve meses. No se concibe que alguna vez se haya dado a luz a un ser humano a partir del esfuerzo conjunto de nueve mujeres durante un mes, o de tres mujeres durante tres meses, a pesar de que el producto final parece, matemáticamente, el mismo:

$$9 \cdot 1 = 1 \cdot 9 = 3 \cdot 3 = 9 \text{ meses-mujer.}$$

Cabe decir que, en este ejemplo tan claro, no se tienen en cuenta excepciones como los sietemesinos (7 meses-mujer de esfuerzo), o los gemelos (4,5 meses-mujer de esfuerzo).

En resumidas cuentas, llegamos a la conclusión de que los meses y los hombres (o las mujeres) no son intercambiables.

En el caso del esfuerzo para construir *software*, aunque puedan darse pequeñas desviaciones, también se tiene un tipo característico de distribución del esfuerzo a lo largo del tiempo que ya se muestra en otro módulo.

Podéis ver la figura "Ciclo de vida en cascada" del subapartado 1.8 del módulo "El proyecto informático de construcción de *software*" de esta asignatura.

### 1.4. Métricas orientadas al tamaño y a la función

Con el objetivo de medir la productividad del proceso de construcción de *software* de aplicación, debe darse la posibilidad de determinar primitivas de salida que sean útiles para relacionarlas después con el volumen del trabajo o el esfuerzo (primitiva de entrada) que representa desarrollar un producto de *software* determinado. Y eso es francamente difícil.

El **tamaño del *software***, construido a partir de las líneas de código que éste incluye, es una medida tradicional para evaluar el volumen de trabajo.

Pero debe conocerse, de entrada, que el tamaño es una medida que, a veces, no es lo suficientemente indicativa de lo que realmente cuesta construir un *software*.


#### **Las líneas de código no siempre son una medida indicativa del esfuerzo**

A veces, algoritmos laboriosos y difíciles de encontrar se concretan en muy pocas líneas de código (pongamos treinta o cuarenta líneas), que provocan la imposibilidad de hacerse una idea del volumen de trabajo que representa pensar, diseñar y disponer finalmente de un algoritmo determinado a partir del disminuido resultado final. En este caso, recurrir a una medida del tamaño como las líneas de código puede ser poco correcto.


Por ello se han propuesto también otras unidades de medida que, en lugar del tamaño, intentan tener en cuenta la dificultad intrínseca de construir un *software* determinado; es decir, métricas que se refieren no al tamaño del producto final obtenido, sino a las funcionalidades que éste incorpora. Un ejemplo suficientemente conocido es el de los puntos de función de Albrecht que después estudiaremos con detalle.

Es necesario decir, sin embargo, que el problema de optar por una métrica orientada al tamaño o a la función no es tan grave si se tiene en cuenta que, al menos en el caso de la informática de gestión, se da una especie de “estándar de complejidad sencilla” que no hace muy difíciles los algoritmos y que, en cierta manera, permite relacionar métricas orientadas al tamaño, como las líneas de código, con métricas orientadas a la función, como los puntos de función, como veremos también más adelante.

Podéis ver los puntos de función de Albrecht en el subapartado 1.4.2 de este módulo didáctico.

En la práctica, a pesar de los problemas y defectos que estudiaremos a continuación, la medida de líneas de código y/o de puntos de función es la más habitual para determinar el volumen de un proyecto informático, es decir, son las principales primitivas de salida que se utilizan para calcular ratios de productividad. 

#### **1.4.1. Las líneas de código**

La medida más utilizada para determinar el tamaño de un proyecto informático ha sido, durante mucho tiempo, la de las líneas de código del *software* final obtenido. A menudo, en la literatura especializada se utilizan diversas denominaciones según las interpretaciones de cada uno respecto de la definición de éstas. A continuación presentamos algunas de éstas: 

a) **LOC:** líneas de código. Es la más habitual y antigua.

LOC es la sigla de la expresión inglesa *Lines of Code*.



b) **KLOC**: miles de líneas de código. Es la unidad de medida que adoptan la mayoría de modelos clásicos de estimación de costes en la calificación de un proyecto informático.

c) **DSI**: instrucciones de código fuente realmente entregadas, y su múltiplo **KDSI**, que surgieron para que se tuviera en cuenta que, en los nuevos entornos de desarrollo y construcción de *software*, buena parte de las líneas de código no siempre las ha escrito el equipo del proyecto, sino que las han generado las herramientas de productividad del entorno de programación.

DSI es la sigla de la expresión inglesa *Delivered Source Instructions*.

d) **NCSS**: líneas de código fuente sin tener en cuenta los comentarios, y su múltiplo **KNCSS**, por el hecho de considerar que en un buen proceso de construcción los programas incluyen líneas de comentario o que una línea de tratamiento se puede escribir en diferentes líneas de código para aumentar la legibilidad y mejorar la mantenibilidad del *software*.

NCSS es la sigla de la expresión inglesa *Non Comment Source Statements*.

e) **NSLOC**: nuevas líneas de código fuente, tal como se realiza en el modelo COCOMO 2 para que se tenga en cuenta que sólo deben contarse las líneas de código nuevas sin contar las que incorpore automáticamente el entorno de programación, o, lo más importante en este caso, el hecho de que parte del código final puede proceder de la reutilización de rutinas u objetos ya disponibles que no se han de volver a escribir, pero que se incorporan al proyecto informático en curso.

NSLOC es la sigla de la expresión inglesa *New Source Lines of Code*.

### Las diferentes denominaciones de la medida de las líneas de código


Las diferentes denominaciones que existen cuando se habla de las líneas de código recogen el hecho de que tenemos bastantes dificultades y dudas a la hora de medirlas. Éstos son algunos de los planteamientos que hacen difícil la valoración:

- ¿Debemos contar o no las líneas de comentario?
- ¿Qué ocurre cuando una única instrucción se escribe, por legibilidad, en dos o más líneas de código fuente?
- ¿Se deben tener en cuenta las líneas que haya generado automáticamente el entorno de programación (formateador de pantallas, gestor de formularios de pantalla, gestor de acceso a la base de datos, etc.)?
- ¿Se deben tener en cuenta las líneas de código de rutinas o los objetos reutilizados procedentes de otras aplicaciones?
- ¿Es la medida basada en las líneas de código indiferente al lenguaje de programación utilizado?
- ¿Qué pasa con las ratios de productividad establecidas cuando cambian los lenguajes de programación utilizados?

Cada modelo de estimación o cada programa de métricas de *software* elige cuál de las muchas interpretaciones posibles se debe tener en cuenta.

De manera simplificada, aquí consideraremos que las LOC (que a menudo se cuentan de mil en mil, en KLOC) son las nuevas líneas de código fuente –que incluyen las directivas de compilación\*, las declaraciones de las estructuras de


\* Por ejemplo, las COPY del COBOL.

datos y el código ejecutable– excluyendo los comentarios y las líneas generadas automáticamente por el entorno de programación o fruto de reutilización. Cada línea física de código se cuenta una sola vez y cada fichero incluido automáticamente\*\* se cuenta también una sola vez. 

**\*\* Como en el caso del COPY del COBOL.**

### Líneas de código, productividad y lenguajes de programación

Las líneas de código se empezaron a utilizar como unidad de medida durante los años setenta y ochenta. A menudo se asocian a ratios de productividad relacionándolas con el esfuerzo medido casi siempre como el trabajo que realiza un profesional en una unidad de tiempo determinada: un día (persona-día), un mes (persona-mes) o un año (persona-año).

 Podéis ver las ratios de productividad en el subapartado 2.1.2 del módulo "El proyecto informático de construcción de software" y en el subapartado 1.2 de este módulo didáctico.

Las **ratios de productividad** también proceden de los años setenta y ochenta (de la informática y de los lenguajes que existían entonces). Estas ratios, que actúan como estándares de la profesión y que han sido mencionadas de paso, son las siguientes:

- a) 10 LOC por día y persona ocupada en el proyecto, según Barry W. Boehm a principios de los años ochenta.
- b) 350 NCSS por persona y mes, según datos de comienzos de los años noventa en los proyectos de la empresa Hewlett Packard recogidos por Robert O. Grady.

El problema principal de las líneas de código con vistas al estudio de la productividad es que los lenguajes de programación han ido cambiando con el tiempo y que, evidentemente, el número de líneas de código que es necesario para implementar una misma funcionalidad depende del lenguaje de programación utilizado.

En su libro de 1986, el especialista Caper T. Jones ofrecía datos que ilustran varios aspectos de la productividad asociada a los diferentes lenguajes de programación:

Esfuerzo para desarrollar la misma cantidad de líneas de código en diferentes lenguajes			
Lenguajes	Tamaño (LOC)	Esfuerzo (persona-mes)	LOC por persona-año
Ensamblador	10.000	40	3.000
Macroensamblador	10.000	42	2.857
C	10.000	44	2.727
COBOL	10.000	48	2.500
Pascal	10.000	51	2.354
Ada	10.000	52	2.308
BASIC	10.000	53	2.264

Fuente: C.T. Jones, 1986.

#### Lectura complementaria

Encontraréis datos sobre la productividad asociada a los diferentes lenguajes de programación en la obra siguiente:

C. T. Jones (1986).  
*Programming Productivity*.  
Nueva York: McGraw-Hill.

Está claro que con lenguajes de alto nivel habrá que escribir menos para obtener la misma funcionalidad y, por tanto, la superior “productividad aparente” (más LOC por persona-año) de los lenguajes de bajo nivel desaparece.

Por tanto, posiblemente es más adecuada la tabla siguiente, también extraída del libro de Jones, que ofrece resultados parecidos, pero que parte del esfuerzo que es necesario para desarrollar, en diferentes lenguajes, unos programas que implementen una misma funcionalidad.

<b>Esfuerzo y líneas de código para desarrollar la misma funcionalidad en diferentes lenguajes</b>			
<b>Lenguajes</b>	<b>Tamaño (LOC)</b>	<b>Esfuerzo (persona-mes)</b>	<b>LOC por persona-año</b>
Ensamblador	10.000	40	3.000
Macroensamblador	6.666	28	2.856
C	5.000	22	2.727
COBOL	3.333	26	2.500
Pascal	2.500	13	2.307
Ada	2.222	12	2.222
BASIC	2.000	11	2.181

Fuente: C.T. Jones, 1986.

Los datos de Jones son suficientemente compatibles con los otros estándares de productividad que hemos ido mencionando. Si tenemos en cuenta el COBOL, e imaginamos unos 225 días de trabajo al año exceptuando las fiestas, las vacaciones y los fines de semana ( $365 - 14 - 22 - 52 \cdot 2 = 225$ ), la productividad de la que habla Jones es de 11,11 LOC por persona-día, que está próximo a las 10 que decía Boehm en la misma época y es un poco inferior a las casi 16 LOC por persona-día que representan las 350 NCSS de Grady establecidas unos cuantos años después.

El COBOL es con mucho el lenguaje más usado en las aplicaciones de gestión.

Por tanto, la idea de una productividad media entre 10 y 16 LOC por persona-día con lenguajes como, por ejemplo, el COBOL es la conclusión final que se puede extraer de los datos disponibles a mitad de los años ochenta.

Cabe mencionar que estos datos provienen, como ya hemos dicho, de los años setenta y ochenta y no recogen lo que se puede conseguir con lenguajes de programación de productividad más elevada o con las facilidades de los nuevos entornos de programación y sus ayudas.

En los últimos años han aparecido una serie de lenguajes muy fáciles de utilizar que unen, poco a poco, las ventajas de la orientación a objetos con los de la llamada *programación visual*. Nacidos en el mundo de la microinformática para ayudar a la realización de programas con interfaz visual bajo el paradigma

#### Ejemplos de programas con interfaz visual

Entre los muchos ejemplos disponibles de programas bajo el paradigma WIMP, debemos destacar PowerBuilder, Visual Basic o Delphi, que son –posiblemente– los más utilizados actualmente.


WIMP (*windows, icons, mouse y pop-up menu*; es decir, ventanas, iconos, ratón y menús desplegables), constituyen la manera más moderna y más evolucionada de lo que se ha llamado RAD (*rapid application development*, es decir, el desarrollo rápido de aplicaciones).

El problema es que todavía no existen datos suficientes para tener en cuenta estos nuevos lenguajes, que, eso sí, parecen haber aumentado mucho la productividad en la construcción de *software* de aplicación.

### Líneas de código y el conjunto del proyecto informático

En cualquier caso, incluso pensando en lenguajes clásicos en la informática de gestión como el COBOL, es evidente que si nos hablan de un programador que es capaz de codificar sólo entre 10 y 16 LOC en un día (es decir, en ocho horas de trabajo), la primera idea es que se trata de un profesional poco eficiente. Ahora bien, cuando se habla de las líneas de código como métrica de tamaño de un proyecto informático, no se piensa únicamente en la actividad de codificación que lleva a cabo el programador.

Las LOC de las que se habla aquí, además de coincidir físicamente con las líneas de código fuente de los programas construidos, incorporan, a los efectos de la medida de la productividad, todos los otros trabajos que ha sido necesario realizar para llegar a la codificación.

Otra vez son los datos que ofrece Caper T. Jones en uno de sus libros (1986) los que nos pueden ayudar a aclarar el significado de las LOC. Jones habla de la *productividad aparente de una persona* a partir de las LOC que se pueden implementar por persona y año. Pero debemos saber cuál es el trabajo que incorpora cada una de las medidas o estándares. 

#### La productividad aparente del COBOL

En el caso concreto del COBOL o en el de un lenguaje parecido, la productividad aparente varía mucho en función de las actividades que tengamos en cuenta, como vemos a continuación:

- Si sólo se tiene en cuenta la codificación y, además, medida en sólo un día de trabajo (y convenientemente pasada a su equivalente anual), se obtienen 25.000 LOC por persona-año.
- Si se tienen en cuenta el diseño, la codificación y las pruebas de sólo un módulo o rutina y se realiza la medida durante un mes (y después, como antes, se pasa a su equivalente anual), se obtiene una medida de 12.000 LOC por persona-año.
- Si se trata del diseño, la codificación y las pruebas de un programa completo que consta de diferentes módulos o rutinas, se obtienen 6.000 LOC por persona-año.
- Cuando se tiene en cuenta la actividad de un programador durante todo un año en la cual, evidentemente, se incluyen las interrupciones entre proyectos y, también, otras actividades no directamente relacionadas con la programación, se obtienen 4.000 LOC por persona-año.

- Si se incluyen también todas las actividades previas a la programación y las pruebas (el estudio de oportunidades, el análisis de requerimientos, el diseño, la codificación, la integración, todo tipo de pruebas, las actividades de control y gestión de la calidad, la documentación externa e interna y el mantenimiento de un proyecto), se obtiene una medida de 2.500 LOC por persona-año.
- Si a las actividades anteriores se añade también el apoyo para un centenar de usuarios y su formación, la medida que se obtiene es de 750 LOC por persona-año.
- Finalmente, cuando se tiene en cuenta el esfuerzo total en *software* de una organización o empresa durante todo un año, incluyendo los proyectos cancelados o fracasados, los desarrollos nuevos y el mantenimiento y las ampliaciones de sistemas viejos de información, se obtienen 250 LOC por persona-año.

Destacamos, entre éstas, la medida de las 2.500 LOC por persona-año, que es la más adecuada porque tiene en cuenta todas las actividades que corresponden al proyecto de construcción de *software* de aplicación que nos ocupa. El dato coincide, en orden de magnitud, con el intervalo entre 10 y 16 LOC por persona-día que la informática tradicional considera el estándar de productividad habitual en grandes proyectos de informática de gestión con lenguajes tradicionales como el COBOL.

### 1.4.2. Los puntos de función

Como veremos, existen diferentes modelos de estimación de las cargas de un proyecto informático. La mayoría utilizan como medida de las primitivas de salida las líneas de código en cualquiera de las versiones posibles. Otra alternativa, cada día más utilizada, es la medida de los puntos de función que definió Albrecht en un primer artículo del año 1979 y que fue revisado (y relacionado también con la métrica de las líneas de código) en el año 1983 por el mismo Albrecht, ayudado por Gaffney.

El recuento de los puntos de función, como veremos, deja algunos aspectos a la interpretación de quien lleva a cabo la medida y ello ha generado diferentes artículos y trabajos que, manteniendo la idea central de Albrecht, intentan ayudar a adaptar el recuento de puntos de función a la realidad cambiando la informática de gestión: bases de datos relacionales, nuevos lenguajes y entornos de programación, nuevas herramientas de programación visual, orientación a objetos, etc.

Desde el año 1984, el denominado *grupo internacional de usuarios de los puntos de función*, IFPUG (de la denominación inglesa *International Function Points User Group*) publica periódicamente la manera correcta de evaluar los diferentes aspectos discrecionales del recuento de puntos de función de Albrecht. La versión 3.0, fechada en el año 1990, y la 4.0, que es de 1994, son las más utilizadas hoy y la diversa literatura técnica actual sobre puntos de función hace referencia a ellas a menudo.

Actualmente, la mayoría de especialistas estarían de acuerdo en afirmar que la métrica de los puntos de función es la más utilizada. El trabajo del IFPUG la pone en cierta manera al día, mientras que las métricas basadas en las líneas de código no siempre se han adaptado bien a los nuevos lenguajes de programación o a las herramientas RAD.

Podéis ver diferentes modelos de estimación de costes en el subapartado 2.3 y las diferentes versiones de medidas de líneas de código en el subapartado 1.4.1 de este módulo didáctico.


#### Lectura complementaria

Podéis consultar la obra siguiente:

A.J. Albrecht; J.E. Gaffney Jr. (1983). "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation". *IEEE Transactions on Software Engineering* (vol. SE-9, núm. 6, noviembre, pág. 639-648).

En la dirección electrónica [www.ifpug.org](http://www.ifpug.org) podéis encontrar información de la versión más actual del IFPUG en Internet, aunque para obtener determinados datos o manuales se ha de ser socio.

WEB

A pesar de todo, continúa vivo el problema de que la métrica fue pensada durante los años setenta, cuando la informática era bastante diferente de lo que es hoy en lenguajes, tipo de aplicaciones, importancia de las bases de datos, en el papel decisivo de las comunicaciones y de la distribución de sistemas, etc. EL IFPUG ha decidido mantener inalterada la estructura de la métrica, lo que no facilita su utilización actual. En concreto, el IFPUG recomienda, siguiendo las directrices que establece, construir procedimientos normalizados y uniformes en cada instalación para proceder al recuento de los puntos de función siempre de una misma manera homogénea. 

Como veremos, también será necesario establecer para cada instalación cuál es la ratio de productividad (puntos de función por persona-mes o por hora de trabajo) que son imprescindibles para una estimación y calificación correctas de proyectos informáticos futuros.

### Determinación de los puntos de función

El recuento de los puntos de función se elabora a partir de determinadas características funcionales, que pueden ser de datos o de transacción:

1) Las **características funcionales de datos** son las que presentamos a continuación:

a) **Ficheros lógicos internos (LIF)**: grupo de datos interrelacionados lógicamente y que pueden ser identificados claramente incluso por los usuarios, o a veces información de control pero que, en cualquier caso, siempre se mantienen dentro de los límites de la aplicación, es decir, son internos y propios de la aplicación. Son los ficheros de la aplicación (o, si se quiere, las tablas de la base de datos).

LIF es la sigla de la expresión  
inglesa *Logical Internal File*.

b) **Ficheros de interfaces externas (EIF)**: grupo de datos interrelacionados lógicamente y que pueden ser identificados por los usuarios, o a veces información de control pero, en este caso, proceden de fuera de los límites de la aplicación y se mantienen al margen de ésta. Representan los ficheros o las tablas que la aplicación utiliza pero que no crea ni mantiene.

EIF es la sigla de la expresión  
inglesa *External Interface File*.

2) Las **características funcionales de transacción** son las siguientes:

a) **Entradas externas (EI)**: hacen referencia a los tratamientos que procesan datos o información de control introducidos en la aplicación desde fuera de sus límites. Son, evidentemente, las entradas a la aplicación.

EI es la sigla de la expresión  
inglesa *External Input*.

b) **Salidas externas (EO)**: cualquier proceso elemental que genere datos o información de control que salga de los límites de la aplicación. Equivale a las salidas que genera la aplicación.

EO es la sigla de la expresión  
inglesa *External Output*.

c) **Consultas externas (EQ)**: proceso elemental formado por una combinación de entrada y salida que permite la recuperación de datos. La salida no

EQ es la sigla de la expresión  
inglesa *External Query*.

debe contener datos derivados y se pueden utilizar los ficheros lógicos internos. Son las consultas internas de la aplicación, que a menudo se resuelven con consultas en los ficheros o tablas propias.

Siguiendo una serie de directrices más o menos objetivas que hoy elabora y mantiene el IFPUG, cada una de estas características funcionales queda definida como si fuera de complejidad simple, media o compleja. En la tabla siguiente se muestra la ponderación de cada característica funcional y la disposición de los cálculos para obtener, gracias a pesos de ponderación fijados por Albrecht a finales de los años setenta, lo que se denomina **total de puntos de función sin ajustar (TUFPS)**:

TUFP es la sigla de la expresión inglesa *Total Unadjusted Function Points*.

Puntos de función sin ajustar (TUFPS)					
Tipo	Descripción	Complejidad			Total
		Simple	Media	Compleja	
EI	Entradas externas	× 3	----- × 4	----- × 6	-----
EO	Salidas externas	× 4	----- × 5	----- × 7	-----
LIF	Ficheros lógicos internos	× 7	----- × 10	----- × 15	-----
EIF	Interfaces lógicas externas	× 5	----- × 7	----- × 10	-----
EQ	Consultas externas	× 3	----- × 4	----- × 6	-----
Total de puntos de función sin ajustar (TUFPS)					-----

#### Los pesos de ponderación

La ponderación de las características funcionales de la tabla refleja claramente las preocupaciones de finales de los años setenta, cuando, por ejemplo, se pensaba que el tratamiento de un fichero complejo daba prácticamente cinco veces más trabajo que resolver el tratamiento de una entrada sencilla (eso es lo que, de hecho, quieren decir, comparativamente, los factores 15 para el LIF complejo y 3 para la EI sencilla).

El mismo Albrecht ya pensó que las características funcionales indicadas no agotaban todas las posibilidades de definir funcionalmente un *software* de aplicación. Por ello, añadió que se debían considerar también las características generales del sistema. En la formulación establecida en el año 1979 (y mantenida todavía por el IFPUG sin modificaciones) se dan catorce características funcionales y se evalúa cada una de 0 a 5 (el valor medio es, pues, 2,5).

La tabla siguiente detalla las características generales del sistema y una disposición adecuada a su cálculo:

Características generales del sistema: grado total de influencia (TDI)		
ID	Característica del sistema	Total
C1	Comunicación de datos	-----
C2	Funciones distribuidas	-----
C3	Rendimiento	-----
C4	Configuración muy utilizada	-----
C5	Frecuencia de transacciones	-----
C6	Entrada <i>on-line</i> de datos	-----
C7	Eficiencia de usuario final	-----
C8	Actualización <i>on-line</i>	-----
C9	Procesos complejos	-----
C10	Reusabilidad	-----

Características generales del sistema: grado total de influencia (TDI)		
ID	Característica del sistema	Total
C11	Facilidad de instalación	-----
C12	Facilidad de operación	-----
C13	Instalación múltiple	-----
C14	Facilidad de cambios	-----
Grado total de influencia (TDI)		-----

Estas características generales afectan al resultado de los puntos de función de manera que la suma de los valores de éstas (que se encuentra siempre entre 0 y 70) se denomina **grado total de influencia (TDI)**.

TDI es la sigla de la expresión inglesa *Total Degree of Influence*.

Con el valor del grado total de influencia se determina el **ajuste por la complejidad** del proceso (PCA) según la fórmula siguiente:

PCA es la sigla de la expresión inglesa *Processing Complexity Adjustment*.

$$PCA = 0,65 + (0,01 \cdot TDI).$$

Finalmente, los **puntos de función (FP)** se obtienen ajustados y definitivos según el cálculo siguiente:

FP es la sigla de la expresión inglesa *Function Points*.

$$FP = TUFP \cdot PCA,$$

donde *PCA* es el valor de la expresión anterior.

Continúa vigente un debate general sobre la idoneidad tanto de las características funcionales (EI, EO, EQ, LIF, EIF), como de las características generales, que, de hecho, se establecieron a finales de los años setenta y, muy posiblemente, podrían no ser las más adecuadas para reflejar las funcionalidades de las aplicaciones de hoy día.

Aunque el IFPUG ha decidido mantener las mismas características y ponderaciones que escogió Albrecht sin modificar los pesos de ponderación, periódicamente va publicando guías para ayudar a la determinación de los valores de las características generales y, también, del grado de complejidad de las características funcionales.

Por falta de espacio, no podemos incluir aquí las recomendaciones completas del IFPUG, pero sí indicaremos algunas a modo de ejemplo, extraídas de la versión 4.0 (1994) del manual del IFPUG:

1) La **complejidad funcional sobre los ficheros** (tanto los LIF como los EIF) se determina a partir de dos conceptos:

- El número de elementos, o campos de datos, de un fichero (DET).
- El número de registros elementales diferentes (RET).

DET es la sigla de la expresión inglesa *Data Element Type*.  
RET es la sigla de la expresión inglesa *Record Element Type*.



La complejidad de las características funcionales sale, en este caso, de una tabla como la que presentamos a continuación:

Medida de la complejidad de los LIF y los EIF			
	1 a 19 DET	20 a 50 DET	51 DET o más
1 RET	Baja	Baja	Media
2 a 5 RET	Baja	Media	Alta
6 RET o más	Media	Alta	Alta

2) De manera similar, la **complejidad de las características generales** se puede evaluar a partir de las guías que publica el IFPUG. Por ejemplo, en la versión 4.0, la primera característica funcional (comunicación de datos) se determina con los valores siguientes:

Característica funcional: comunicación de datos	
0	La aplicación es puramente <i>batch</i> o se ejecuta en un único PC
1	La aplicación es <i>batch</i> , pero tiene entrada de datos o impresión remota
2	La aplicación es <i>batch</i> , pero tiene entrada de datos e impresión remotas
3	La aplicación incluye entrada de datos <i>on-line</i> o TP (teleproceso) <i>front-end</i> para un proceso <i>batch</i> o un sistema de consulta
4	La aplicación es más que un <i>front-end</i> , sin embargo, soporta sólo un tipo de protocolo de comunicaciones de TP
5	La aplicación es más que un <i>front-end</i> y soporta más de un tipo de protocolo de comunicaciones de TP

## Los puntos de función y la productividad

Para tener datos de la productividad a partir de los puntos de función, se suele utilizar una ratio de productividad que indica el número de horas de trabajo (siempre de un profesional medio) necesarias para implementar un punto de función.

Evidentemente, como ya hemos dicho antes en el caso de las líneas de código, en este trabajo se reflejan todas las tareas necesarias, desde el estudio de oportunidad hasta la puesta en marcha, pasando por el análisis de los requerimientos, el diseño de la solución técnica, la programación y las pruebas.

Aunque siempre se deben tener los datos de productividad habituales en una determinada instalación, Jones nos ofrece como patrón de referencia los resultados en su empresa Software Productivity Research (SPR), donde se descompone un proyecto informático en diferentes actividades (hasta

### Lectura recomendada


Encontraréis los datos que ofrece Jones sobre su empresa en la obra siguiente:  
**C.T. Jones** (1996). "Software Estimating Rules of Thumb". *IEEE computer* (marzo, pág. 116-118).

un máximo de 25 para todo tipo de proyectos). Los datos se resumen en la tabla que presentamos a continuación:

Productividad en horas por punto de función				
Actividad	Descripción	Horas/FP	Ponderación (%)	Total horas/FP
01	Requerimientos	0,75	5	3,75
04	Plan del proyecto	0,26	2	0,52
05	Diseño inicial	0,75	8	6,00
06	Diseño detallado	0,88	10	8,80
08	Codificación	2,64	40	105,60
15	Documentación de usuario	1,89	10	18,90
16	Prueba unitaria	0,88	10	8,80
17	Prueba funcional	0,88	4	3,52
18	Prueba de integración	0,75	4	3,00
19	Prueba del sistema	0,66	4	2,64
25	Gestión del proyecto	1,32	3	3,96
<b>Media</b>				<b>1,6549</b>

Fuente: C.T. Jones, 1996.

Teniendo en cuenta sólo las actividades típicas de los proyectos informáticos en la informática de gestión, las horas por punto de función que marcan la productividad en cada caso y una ponderación de su importancia final en el conjunto del proyecto, Jones obtiene una media de 1,65 horas por punto de función, hecho que no deja de ser un valor muy optimista, conseguido por una empresa que hace mucho tiempo que observa con detalle todos los procesos de mejora de la productividad.

A todos los efectos, una productividad que tenga entre dos o tres horas por punto de función podría ser la más habitual en la mayoría de las instalaciones modernas. 

### 1.4.3. Relación entre puntos de función y líneas de código

Aunque los puntos de función y las líneas de código sean diferentes, es posible encontrar una especie de equivalencia entre los unos y las otras. De hecho, ya existen miles de proyectos informáticos que se han medido utilizando puntos de función y, también, líneas de código, hecho que ha permitido obtener ratios de conversión de una unidad a la otra.

Se han publicado diferentes equivalencias entre FP y LOC; la que proponía Caper T. Jones es la siguiente:

LOC por punto de función	
Lenguaje	LOC/FP
Ensamblador	320
Macroensamblador	213
C	150
COBOL	106
Fortran	106
Pascal	91
RPG	80
PL/I	80
Ada	71
BASIC	64

Fuente: C.T. Jones, 1986.

Otros autores pretenden incluso encontrar fórmulas de equivalencia, como hace Bernard Londeix, que en el caso concreto del lenguaje COBOL propone la fórmula de conversión siguiente:

$$LOC = 118,7 \cdot FP - 6,490.$$

Sin embargo, a pesar del interés por la exactitud, cabe mencionar que las imprecisiones en la determinación de las LOC y, sobre todo, de los FP (a pesar de los estándares y las recomendaciones del IFPUG), convierten en muy razonable la propuesta del mismo Caper T. Jones.

La propuesta de Caper T. Jones en su columna mensual en la revista *IEEE Computer*, en el año 1996, concretamente en el mes de marzo, establecía la equivalencia siguiente:

“Regla 1: un punto de función = 100 sentencias de código fuente lógico.”

En aquel mismo artículo, este experto justificaba la equivalencia de esta manera:

“El ratio entre sentencias de código fuente lógico sin comentarios al punto de función va desde más de 300 sentencias por punto de función para los lenguajes básicos de ensamblador, hasta menos de 20 para los lenguajes orientados a objetos y la mayoría de los generadores de programas. Como los lenguajes de procedimientos (*procedural*) como C, COBOL, Fortran y Pascal están próximos a una relación de 100 en 1, este valor puede servir como factor de conversión basto.”

“Esta regla tiene un gran margen de error y son necesarias correcciones significativas cuando se trata de lenguajes de programación orientados a objetos, generadores de aplicaciones o aplicaciones donde se utilice una cantidad significativa como código reutilizado.”

C.T. Jones (1996, pág. 116).

Cabe mencionar que, en este contexto, Jones considera más bien las líneas de código lógicas que las físicas (que dependen del estilo de cada programador)

#### Lectura complementaria

Podéis consultar la obra siguiente:

**B. Londeix** (1987). *Cost estimation for software development*. Reading (Massachusetts): Addison- Wesley.

#### Conversión de LOC en FP

- Para lenguajes de procedimiento, como C, COBOL, Fortran o Pascal: 100 LOC por FP.
- Para lenguajes orientados a objeto y generadores de aplicaciones: 20 LOC por FP.

y, sobre todo, que los puntos de función hacen referencia a la versión 3.0 del recuento de puntos de función según el IFPUG.

#### 1.4.4. Una perplejidad final

Aunque parezca mentira, no existe un acuerdo total entre los datos que Jones ofrece y los otros disponibles que hemos comentado, procedentes de Boehm y Grady. Es más, la diferencia es muy acusada.

Podéis ver los datos de Boehm y Grady con relación a las líneas de código en el subapartado 1.4.1 de este módulo didáctico.

Si, de manera general, un punto de función son 100 LOC de COBOL y, según hemos visto, hoy en día se implementa en dos o tres horas, en una jornada de trabajo de ocho horas se deberían implementar entre 260 y 400 LOC, cantidad que se encuentra muy por encima de lo que dicen Boehm y Grady (entre 10 y 15 LOC por persona-día).

Podéis ver la relación que se da entre los puntos de función y la productividad en el subapartado 1.4.2 de este módulo didáctico.

En cualquier caso, no hay nada que hacer. Este tipo de contradicciones sobre los datos de productividad son habituales en las métricas de productividad del *software*, que, evidentemente, acusan el paso de los años y, sobre todo, las características propias (no siempre iguales) de las instalaciones y los proyectos informáticos de donde se han tomado los datos.

Como ya se ha dicho, la única manera segura de poder tener unos estándares de productividad buenos y dignos de ser utilizados es no depender de lo que dicen libros y artículos (con datos obtenidos en condiciones a menudo bien diferentes) y disponer de los datos de productividad propios\*, datos que pueden haber sido obtenidos en proyectos anteriores del mismo tipo (en cuanto a aplicación y tecnología) y con equipos de desarrollo de calificaciones y características personales conocidos. Por este motivo, como ya se ha dicho en otro módulo, es tan importante disponer de la documentación de gestión y de los datos de proyectos informáticos anteriores. !

**\* A partir de KLOC o de los puntos de función.**

Podéis ver el subapartado 1.6 del módulo "El proyecto informático de construcción de *software*" de esta asignatura.

## 2. Estimación de costes de un proyecto informático

Ya sabéis que la gestión de un proyecto informático de gestión empieza con la calificación del proyecto, que pretende, en primer lugar, obtener una idea del volumen de trabajo que costará construir la aplicación (estimación) y, en segundo lugar, planificar en el tiempo las diferentes actividades que es necesario llevar a cabo (planificación).

Podéis ver las etapas de un proyecto informático en el subapartado 1.2 del módulo "El proyecto informático de construcción de *software*" de esta asignatura.

La primera de estas etapas se conoce también con el nombre de **estimación de costes de un proyecto informático**, ya que a partir del esfuerzo de trabajo estimado se obtendrá el presupuesto. Del mismo modo, después de la planificación y el reparto en el calendario de las tareas que se deben a realizar, se obtienen los plazos, etapa que también se conoce con el nombre de **tiempo de desarrollo del proyecto**. Y es necesario recordar que las funcionalidades, los plazos y el presupuesto definen un proyecto informático de construcción de *software*.

Como ya conocéis de otro módulo, los costes que intervienen en un proyecto informático son variados y complejos, pero la tendencia a la disminución del coste del *hardware* y al aumento del coste del *software* provoca que aquí nos podamos centrar en los costes de personal para la construcción del *software* de aplicación. La mayor parte del coste del *software* se encuentra hoy en el coste de las horas de análisis, diseño, programación y prueba que se deben utilizar para obtenerlo. Por ello, cuando aquí se habla de *estimación de costes* se hace referencia, exclusivamente, al esfuerzo humano que ha sido necesario, es decir, a las horas de trabajo requeridas para construir el *software*.

Podéis ver el apartado 1 del módulo "El proyecto informático de construcción de *software*" de esta asignatura.

De las dos etapas de la calificación, la de la estimación de costes y determinación del volumen de trabajo que se debe llevar a cabo es claramente la más difícil y compleja. Existe mucha bibliografía sobre el tema, pero, en cierta manera, sirve de poco. Cuando el jefe de proyecto se plantea por primera vez evaluar el trabajo necesario para construir un *software* determinado sólo dispone de unos pocos datos de lo que ha de ser la aplicación futura y la realidad es que no sabe en absoluto lo suficiente como para llegar a una estimación correcta. Y este problema, simplemente, no tiene solución.

Ya hace muchos años que Tom de Marco dejó claro este punto. En un libro lleno de disquisiciones interesantes y con mucho sentido común, de Marco considera que el hecho de esperar obtener de la bibliografía técnica datos lo bastante fiables como para llevar a cabo una buena estimación es una actitud comparable a la que toman los personajes de la obra de teatro *Esperando a Godot* del premio Nobel Samuel Beckett. En la obra, un clásico del teatro moderno, dos personajes pasan todo el tiempo esperando a alguien, un tal Godot, que, simplemente, no llega nunca.

### Lectura recomendada

Os recomendamos que leáis el capítulo "El dilema de la estimación" de la obra siguiente:

**T. de Marco** (1982). *Controlling Software Projects* (vol. 2, pág. 9-17). Englewood Cliffs: Yourdon Press (Prentice Hall).

En palabras de de Marco:

“No hay modelos de coste transportables. Si esperas que alguien en otro lugar desarrolle un conjunto de fórmulas que puedas utilizar para prever el coste en tu propia instalación, probablemente tendrás que esperar para siempre.”


T. de Marco (1982, pág. 155).

Como ya se ha dicho sobradamente, es imprescindible disponer de datos concretos sobre proyectos anteriores realizados en una instalación determinada. Si no se tienen estos datos, se deben utilizar los de la bibliografía técnica\*, aunque no es siempre razonable fiarse demasiado. Quizá por ello, el propio de Marco propone las dos definiciones sucesivas muy realistas de lo que es una estimación, aunque parezcan más bien cínicas:

1) **Definición implícita de estimación:** una estimación es la predicción más optimista que tiene una probabilidad no nula de llegar a ser cierta.

2) **Definición propuesta por de Marco:** una estimación es una predicción que tiene la misma probabilidad de estar por encima que de estar por debajo del resultado real.

En resumidas cuentas, estimar la carga de trabajo que es necesario llevar a cabo para la construcción de *software* de aplicación es una tarea que normalmente debe fracasar.

A pesar de todo, para empezar, se debe realizar una estimación y, dejando bien claro que en estos aspectos vale más la experiencia que las fórmulas, aquí intentaremos comentar cómo se puede estimar *a priori* el trabajo necesario para construir un *software* de aplicación determinado en la informática de gestión. 

## 2.1. Momento de la estimación y grado de exactitud

El problema, evidentemente, es que cuando se ha de llevar a cabo la primera estimación todavía no se dispone de las especificaciones completas de la aplicación que es necesario construir. Se ignora la mayor parte del detalle de las funcionalidades que caracterizan el proyecto. Precisamente, es en la etapa de análisis y diseño cuando estas especificaciones quedarán totalmente determinadas y se podrá conocer el alcance real del proyecto informático.

Se ha intentado calcular cuál es la desviación que se presenta entre la estimación del volumen de trabajo que se debe realizar y el volumen que se debe llevar a cabo realmente al final. El gráfico de la página siguiente mues-

\* Por ejemplo, la productividad de 10 a 16 LOC por persona-día o 2 horas para implementar un punto de función.

### Lectura recomendada

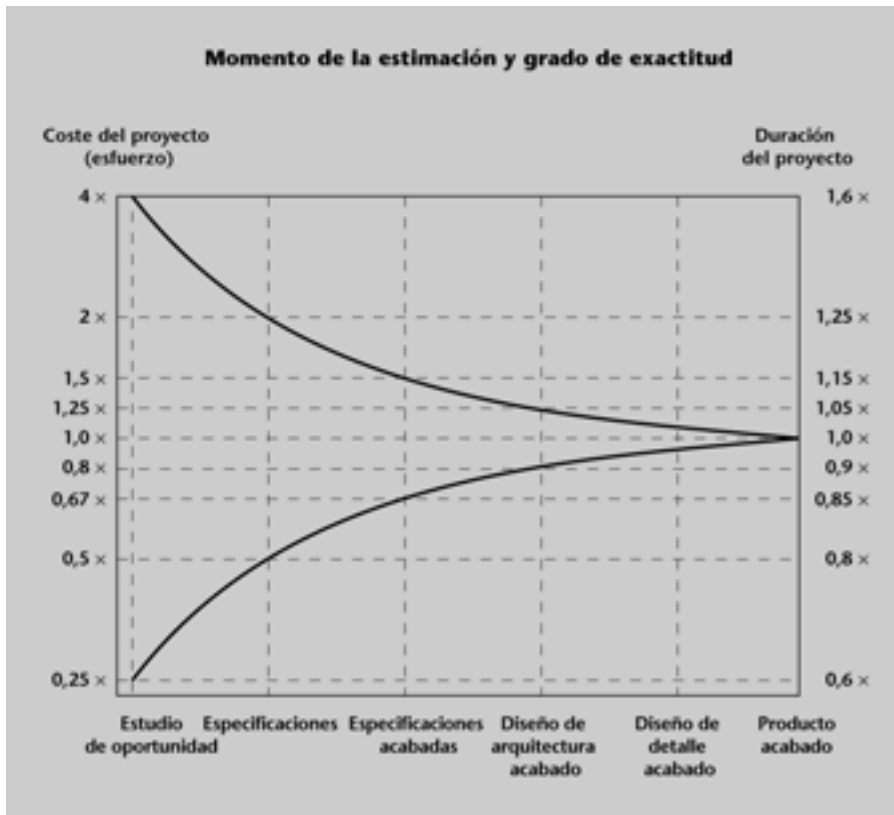
Encontraréis las definiciones de estimación propuestas por de Marco en la obra siguiente:

T. de Marco (1982). *Controlling Software Projects* (pág. 14). Englewood Cliffs: Yourdon Press (Prentice- Hall).

### En cuanto a la experiencia...

... el jefe de proyecto debería ser como el demonio, de quien se dice que sabe más por viejo que por demonio.

tra cómo, a medida que avanza el proyecto informático y se conocen con más detalle las especificaciones, el margen de error va disminuyendo:



Fuente: Gráfico adaptado de Cost Models for Future Life Cycle Process: COCOMO 2.0 de B.W. Boehm y otros (1995).

Cabe destacar (visible en la escala de ordenadas de la izquierda) cómo la primera estimación, realizada en el momento de la definición inicial del producto, puede tener errores de estimación hasta cuatro veces por encima o por debajo de lo que después será la carga real de trabajo.

De manera similar, tal como se ve en la escala de ordenadas de la derecha, la duración estimada del proyecto (es decir, el tiempo de desarrollo que marca los plazos que caracterizan al proyecto) tiene un margen de error que va entre el 160% y el 60% de la duración real, siempre comparando la realidad final con esta primera estimación tan precaria realizada sólo a partir de la definición inicial del producto.

## 2.2. Los métodos de estimación posibles según Barry W. Boehm

En el libro de Barry W. Boehm (1981), que se ha convertido en un clásico sobre el tema, se exponen siete maneras diferentes de proceder a la estimación inicial de costes de un proyecto informático. Algunas pueden parecer extrañas e incluso poco serias, sin embargo, si se tiene en cuenta la realidad de mercado y la práctica real, los siete procedimientos que menciona Boehm (evidentemente, no todos recomendables) son bastante esclarecedores.

A continuación se exponen los siete métodos que menciona Boehm, con comentarios complementarios, tal vez ajenos a la voluntad inicial del autor:

1) **Utilización de modelos:** Boehm los denomina **modelos algorítmicos** y proporcionan uno o más algoritmos que dan una estimación del coste del *software* como función de un número de variables determinado que se consideran influyentes en este coste. Estos modelos, como veremos, pueden ser también de base estadística, teórica o compuestos.

2) **Juicio de expertos en el tipo de proyecto:** método que supone basarse fundamentalmente en la experiencia de uno o más profesionales expertos que ya hayan actuado en diferentes ocasiones como jefes en proyectos bastante parecidos al que nos preocupa.

3) **Analogía con otro proyecto parecido:** estrategia que pretende ampararse en los datos disponibles obtenidos en otro proyecto de construcción de *software* que sea en cierta manera análogo al que nos ocupa en aspectos clave como la tecnología, el tipo de aplicación, el personal que interviene, etc.

4) **Utilizar todos los recursos disponibles:** caso habitual que se da cuando no se gestiona un proyecto informático y se destinan los recursos que parecen necesarios en cada momento del desarrollo. Con respecto a este punto, Boehm hace una referencia explícita a lo que se conoce como la *ley de Parkinson*, según la cual el trabajo siempre se expande hasta ocupar todos los recursos disponibles. Esta ley es fruto de la observación y el ingenio de Cyril Northcote Parkinson.

5) **Precio ganador:** estrategia que propone simplemente una estimación de cargas (esfuerzo, coste y/o tiempo de desarrollo) que provoque que la oferta sea irresistible, con independencia de si después se puede cumplir o no. O se iguala automáticamente el coste al valor más bajo para conseguir un contrato, o se escoge automáticamente un tiempo de desarrollo y unos plazos que, con independencia del trabajo se deba llevar a cabo, permitan al producto ser el primero en llegar al mercado.

#### Una estrategia habitual

Desgraciadamente, la estrategia del precio ganador es en demasiadas ocasiones la realidad del mundo mercantilizado en el que nos movemos: si dos o tres empresas proveedoras deben realizar una oferta de lo que puede costar (en tiempo y dinero) construir una aplicación informática determinada, es casi seguro que se llevará el contrato la que realice una oferta más barata o la que ofrezca un tiempo de desarrollo inferior. El hecho de que después puedan surgir problemas para cumplir lo que se ha prometido es, simplemente, consecuencia de la estrategia con la que se ha conseguido el contrato.

6) **Estimación global descendente\*:** método que intenta obtener un dato único y global para todo el proyecto informático a partir de diferentes propiedades globales del producto final (tamaño, complejidad, dificultad técnica, nivel de calidad y fiabilidad, etc.) que, después, se va descomponiendo.

#### Lectura recomendada

La idea de que el trabajo, si no se controla, ocupa siempre todos los recursos disponibles, se expone de manera muy divertida en la obra siguiente:

**Cyril Northcote Parkinson** (1962). *Al patrimonio por el matrimonio (tercera ley de Parkinson)*. Bilbao: Deusto.

\* En inglés, *top-down*.



7) **Descomposición en actividades (WBS) y estimación ascendente\***: método que descompone el conjunto del proyecto en diferentes actividades\*\* y, una vez estimado el esfuerzo para cada una de éstas, obtiene por agregación el esfuerzo total del proyecto.

\* En inglés, *bottom-up*.  
\*\* En inglés, *Work Breakdown Structure*.

Cabe destacar que, al margen del realismo de algunos de los métodos mencionados por Boehm, los seis primeros representan métodos estratégicos y tácticos que, a menudo, dan lugar a una estimación única de la totalidad del proyecto. Como veremos más adelante, el método más operativo es el séptimo, que permite, además, obtener un desglose del proyecto informático en actividades, hecho que nos será muy útil tanto en la planificación, como en el seguimiento y control del proyecto. !

Podéis ver la operatividad del método de descomposición en actividades y estimación ascendente en el subapartado 2.4 de este módulo didáctico. Podéis ver también la planificación, el seguimiento y el control en el módulo "La gestión de un proyecto informático" de esta asignatura.

A pesar de todo, la mayoría de los métodos que recogen los libros (y éste no será en absoluto una excepción) tratan la estimación de costes por métodos y modelos algorítmicos o con base estadística. En este caso, como veremos enseguida, se ofrecen fórmulas que suelen ofrecer como resultado de la estimación una única cifra global para todo el proyecto informático.

Podéis ver los modelos de estimación en el subapartado 2.3 de este módulo didáctico.

### 2.3. Modelos de estimación

La idea de los modelos de estimación es la de proporcionar sistemas y métodos generales para proceder a realizar la estimación de costes en la construcción de *software* de aplicación.

Antes de disponer de modelos, lo que se hacía era recurrir a la apreciación personal del jefe de proyecto, que a menudo basaba sus estimaciones en una experiencia propia no siempre objetiva o reflejada en datos concretos.

A medida que se introducen los programas de métricas de *software*, se empieza a disponer de datos reales de proyectos ya elaborados. Ello permite escapar de la subjetividad personal e intentar expresar en modelos y fórmulas todo aquello que el estudio estadístico o teórico de los datos disponibles nos permite deducir con respecto a la productividad en la construcción de *software* de aplicación.

Podéis ver los programas de métricas de *software* en el subapartado 1.1 de este módulo didáctico.

Los diferentes **modelos de estimación de costes y/o esfuerzos en la construcción de *software*\*** se pueden dividir en cinco grandes grupos principales: modelos con base histórica, con base estadística, teóricos, compuestos y basados en estándares. A continuación los presentamos brevemente: !

\* En la denominación inglesa, *Cost Estimating Models*.

1) Los **modelos de base histórica** son los más antiguos y, en cierta manera, primitivos. A menudo se basan en la analogía con otros proyectos parecidos y se fundamentan casi exclusivamente en la experiencia profesional (la "historia") de los que efectúan la estimación.

2) Los **modelos de base estadística** intentan superar la experiencia histórica concreta de unos cuantos profesionales y, a partir del estudio estadístico de los datos reales disponibles tomados de un conjunto más o menos numeroso de proyectos ya acabados, obtienen fórmulas que relacionan las diferentes unidades de medida del *software*, a menudo las líneas de código (LOC) y el esfuerzo (generalmente medido en hombre-mes).

3) Los **modelos de base teórica** proporcionan otro enfoque. Estos modelos, más que basarse en datos estadísticos disponibles no siempre suficientemente abundantes y/o generalizables, lo que hacen es partir de una serie de ideas generales sobre el proceso de construcción de *software* y, sobre esta teoría, elaboran fórmulas que relacionan diferentes métricas de *software*.


4) Los **modelos compuestos** han arraigado a partir de los trabajos de Barry W. Boehm. Estos modelos intentan obtener las ventajas de los dos sistemas anteriores: estadísticos y teóricos. Es decir, se parte de una serie de planteamientos teóricos y se complementan o corrigen con datos estadísticos obtenidos de proyectos reales ya acabados.

Dado que a menudo los modelos mencionados anteriormente son poco fiables y ofrecen un grado de error nada despreciable, la práctica profesional ha provocado que muchas instalaciones\* hayan adoptado un sistema más simple y menos complicado: la **elaboración de estándares de productividad** obtenidos en proyectos anteriores de la misma instalación. Para llevar a cabo la estimación de nuevos proyectos se parte de estos estándares.


\* Generalmente, las instalaciones mayores y con más experiencia.

También al margen de los estándares propios de cada instalación, en la bibliografía técnica se encuentran las que podríamos denominar *recomendaciones prácticas a primera vista*\*, que han sido elaboradas por especialistas prestigiosos y que pueden actuar como referencia general futura para los estándares propios de una instalación que empieza a desarrollarse.

\* *Rules of thumb*, en la denominación inglesa.

Cabe señalar que la mayoría de los modelos disponibles se han elaborado durante los años setenta y ochenta y, evidentemente, responden a una informática bastante diferente de la que se lleva a cabo en la actualidad, que ha mejorado bastante la productividad por medio de nuevos entornos de programación, nuevos lenguajes visuales y orientados a objetos, nuevas herramientas de ayuda, etc. Pero, en cualquier caso, los resultados que ofrecen los modelos de estimación más algorítmicos pueden actuar siempre como hito superior de la estimación que se quiere realizar. 

También cabe mencionar que la mayoría de modelos disponibles suelen utilizar las líneas de código (LOC) como base para obtener otras medidas: el esfuerzo medido en personas-mes, los meses de construcción del proyecto, el número de profesionales que deberían estar involucrados, el número de errores que se puede esperar tener, el número de páginas de documen-

 Podéis ver las LOC con relación al lenguaje de programación utilizado en el subapartado 1.4.3 de este módulo didáctico. Podéis ver también las herramientas RAD en el subapartado 1.4.1 de este módulo didáctico.


tación que es necesario realizar, etc. En la mayoría de modelos todo parte de las LOC, que, como ya hemos visto, son una métrica que depende demasiado de los lenguajes de programación (y éstos han cambiado mucho en los últimos años) y que hoy en día a menudo son difíciles de medir, sobre todo con las actuales herramientas RAD, que generan gran parte de líneas del código final.

### 2.3.1. Modelos históricos

Tal vez el método más utilizado todavía para realizar la estimación de costes es el que se basa en la experiencia personal de quien la lleva a cabo (por ejemplo, el jefe de proyecto). Como podéis suponer, se trata de un método marcadamente subjetivo y muy propenso a errores. Puede ocurrir que quien realice la estimación no tenga en cuenta de manera adecuada todos los aspectos que pueden intervenir en el proyecto y también puede suceder que la experiencia disponible no se corresponda en absoluto con el tipo de proyecto o de aplicación que se debe efectuar.

Este sistema se puede aplicar tanto a métodos descendentes, como a métodos ascendentes. En el segundo caso, se realizan estimaciones independientes de cada uno de los subproyectos o de las fases (e incluso de las actividades concretas, si ya se tiene claro cuáles serán) para obtener la estimación del proyecto completo sumando las estimaciones de las diferentes partes.

### Combinación de estimaciones individuales

Puede ocurrir que, cuando el proyecto tiene una cierta envergadura, se pida la opinión de más de un experto y se tenga que combinar dos o tres estimaciones. En este caso, existen diferentes técnicas para intentar encontrar una combinación adecuada, sobre todo para evitar que, en reuniones conjuntas, el estimador con más personalidad imponga su criterio sobre el resto. A continuación mencionamos tres de estas técnicas: 

**a) Media.** La manera más sencilla de llegar a una combinación adecuada es, simplemente, calcular la media entre las diferentes estimaciones obtenidas por separado.

**b) Media ponderada.** Otro procedimiento consiste en tener en cuenta que pueden darse estimaciones pesimistas y optimistas y que se deben tener en cuenta de una manera diferente a otras estimaciones intermedias. Lo que se hace, pues, en lugar de calcular la media habitual, es establecer una media ponderada, que podría ser, por ejemplo, en el caso de tres estimaciones,

la que recomendaba Pressman en las primeras ediciones de su libro *Ingeniería del Software*:

$$e = (o + 4n + p)/6,$$

donde  $e$  es la estimación final,  $o$  es la estimación más optimista,  $p$  la más pesimista y  $n$  la que ha quedado en el medio.

c) **Técnicas de Delphi.** Éste es el procedimiento más serio y complejo. Fue elaborado por la Rand Corporation en 1948 para evitar las reuniones de grupo entre estimadores, debido a la peligrosa dinámica que tenían. La idea es que diferentes estimadores den por separado su estimación después de haber estudiado el proyecto. Estos resultados se trasladan anónimamente a los otros estimadores, que, de esta manera, pueden matizar y rehacer la propia estimación considerando lo que han propuesto los demás. El proceso se repite tantas veces como sea necesario hasta conseguir una cierta convergencia que sea al menos satisfactoria a los ojos del coordinador del proceso. Si así no se llega a la convergencia deseada, el coordinador discute personalmente con cada estimador las diferencias encontradas con el fin tratar de forzar una solución aceptable para todo el mundo.

### 2.3.2. Modelos empíricos o estadísticos

La idea de los modelos empíricos es aprovechar el estudio estadístico de los datos reales disponibles medidos en un conjunto más o menos numeroso de proyectos ya acabados. Se intenta buscar fórmulas que relacionen el esfuerzo con las diferentes variables que puedan incidir en éste.

En primer lugar, en los años sesenta, se intentó aproximar los datos estadísticos disponibles con expresiones lineales del tipo siguiente:

$$E = C_o + \sum_i c_i \cdot x_i$$

En esta expresión,  $E$  es el esfuerzo, a menudo medido en personas-mes,  $x_i$  son las diferentes variables o los atributos que se cree que inciden en el coste, a menudo denominadas CDA, mientras que  $c_i$  son los coeficientes que resultan de la aproximación estadística a partir de los datos disponibles procedentes de proyectos anteriores ya acabados y bien medidos.

CDA es la sigla de la expresión inglesa *Cost Driver Attributes*.

De hecho, bien pronto se observó que los modelos lineales no eran correctos y rápidamente se abandonó este planteamiento para intentar encontrar fórmulas de tipo exponencial de la manera siguiente:

$$E = (a + b \cdot L^c) \cdot m(x),$$

donde  $E$  es el esfuerzo medido en *personas-mes*,  $L$  es el tamaño estimado del proyecto en KLOC, mientras que  $a$ ,  $b$  y  $c$  son los coeficientes que resultan de la aproximación estadística de los datos disponibles.

El término adicional  $m(x)$ , que no todos los modelos utilizan, es un elemento de ajuste para tener en cuenta más datos que simplemente el tamaño del proyecto expresado en KLOC. De hecho, el término  $m(x)$  es aquí el equivalente a las características generales que completaban y permitían ajustar los puntos de función de Albrecht deducidos de las características funcionales. La idea básica es que el esfuerzo tal vez no depende sólo del tamaño (las líneas de código) y que se dan otros factores que influyen en este proceso.

Podéis ver las características funcionales y las generales de los puntos de función de Albrecht en el subapartado 1.4.2 de este módulo didáctico.

La mayoría de modelos empíricos se obtuvieron durante los años setenta, dato que se debe tener en cuenta, ya que se utilizaban datos de proyectos que no se parecen a los proyectos de ahora. Muy a menudo los datos de esfuerzo que ofrecen los modelos empíricos son exagerados, ya que hoy la productividad en la construcción de *software* es bastante más alta\*. Siempre se debe tener en cuenta recordar la base de datos de gestión de los proyectos que se utilizaron para establecer los coeficientes del modelo.

\* Recordemos, sin embargo, que no llega al nivel que se ha alcanzado en el caso del *hardware*.

Presentamos, a continuación, un par de ejemplos de modelos estadísticos que ya se han convertido en clásicos.

### 1) El modelo de Walston y Felix

El primer ejemplo de modelo estadístico es el estudio, publicado en el año 1977, que Walston y Felix realizaron tomando como base sesenta proyectos acabados entre 1973 y 1977. Se trataba, posiblemente, de los primeros proyectos disponibles, no eran homogéneos (aparecían diferentes lenguajes de programación, diferentes ordenadores y aplicaciones diversas) y, además, eran de varios tamaños (de 12 a 11.758 meses-hombre y de 460 a 467.000 LOC). Los coeficientes eran  $a = 0$ ,  $b = 5,2$  y  $c = 0,91$  y la fórmula en este caso es la siguiente:

$$E = 5,2 \cdot L^{0,91}.$$

También se obtenían otras informaciones, como la duración en meses del proyecto ( $D$ ), el número de personas implicadas ( $P$ ) y el número de páginas de documentación ( $DOCK$ ), gracias a las fórmulas siguientes:

- $D = 4,1 \cdot L^{0,36}.$
- $P = 0,54 \cdot E^{0,6}.$
- $DOCK = 40 \cdot L^{1,01}.$

### 2) El modelo de Bailey y Basili

Otro ejemplo clásico es el de Bailey y Basili, publicado en el año 1981, recogiendo datos de dieciocho proyectos más homogéneos, todos del Goddard

Space Center de la NASA y programados en Fortran. En este caso, los coeficientes son  $a = 5,5$ ,  $b = 0,73$  y  $c = 1,16$ ; es decir, se obtiene la expresión siguiente:

$$E = 5,5 + 0,73 \cdot L^{1,16}.$$

Debería sorprender el hecho de que el exponente de las KLOC sea inferior a la unidad en el caso de Walston y Felix, mientras que es claramente superior a la unidad en el caso de Bailey y Basili. La razón es que, en el estudio de Walston y Felix, entre las LOC se contaba hasta un máximo del 50% de comentarios, mientras que en el estudio de Bailey y Basili, como en la gran mayoría de los modelos estadísticos, no se incluyen los comentarios en el recuento de las LOC.

También se da una gran variedad de fórmulas obtenidas a partir de datos procedentes de diferentes proyectos, entre ellas algunas que relacionan el esfuerzo con los puntos de función de Albrecht.

Cabe tener en cuenta que los resultados de esfuerzo que se obtienen en los diferentes modelos no tienen en absoluto por qué coincidir, ya que reflejan datos extraídos de proyectos diferentes y no siempre comparables ni homogéneos. De hecho, los modelos estadísticos mencionados se consideran ahora obsoletos del todo y proporcionan sólo un margen altísimo para una estimación razonable. Modelos posteriores, como el COCOMO de Boehm, han mejorado y han actualizado los datos y las fórmulas.

#### El modelo de Albrecht y Gaffney...

... es un ejemplo de modelo que relaciona el esfuerzo con los puntos de función de Albrecht. Este modelo propone la fórmula siguiente:

$$E = -13,39 + 0,054FP.$$

Podéis ver los modelos COCOMO de Barry W. Boehm en el subapartado 2.3.4 de este módulo didáctico.

### 2.3.3. Modelos teóricos: el modelo de Putman

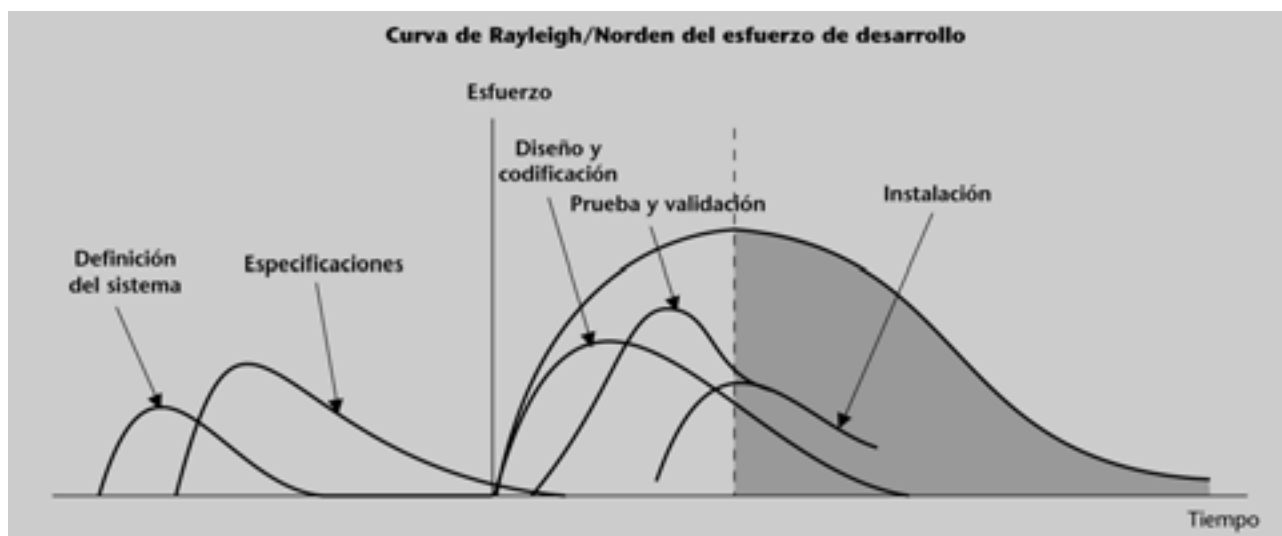
En un artículo de 1978, Lawrence H. Putman comenzó a plantear la necesidad de modelos teóricos que, prescindiendo de la posibilidad de disponer o no de datos de proyectos ya acabados, tuvieran en cuenta la manera de distribuir el esfuerzo en el desarrollo de un proyecto de construcción de *software*.

Putman adoptó la distribución de esfuerzos en el tiempo que marca la forma de la curva de Rayleigh/Norden que se muestra en la figura siguiente:

#### Lectura recomendada

Podéis consultar la obra siguiente:

**L.H. Putman** (1978). "A General Empirical Solution to the Macro Software Sizing and Estimation Problem". *IEEE Transactions on software engineering* (vol. SE-4, núm. 4, julio, pág. 345-361).




El estudio de la curva permite, entre otras cosas, relacionar el esfuerzo ( $K$ , en personas-año), las líneas de código ( $L$ , en KLOC) y el tiempo total de desarrollo del proyecto ( $t_D$ , en meses) según la fórmula:

$$L = C_k \cdot K^{(1/3)} \cdot t_D^{(4/3)}$$

La idea es que, a partir de un esfuerzo ( $K$ ) y un tiempo de desarrollo ( $t_D$ ), la fórmula indica la productividad en la construcción de *software* y nos da el número de KLOC que se obtienen.

La fórmula incluye una constante denominada **constante tecnológica** ( $C_k$ ), que, en palabras de Putman, es en cierta manera una medida del estado de la tecnología que se aplica al proyecto. La constante  $C_k$  refleja el efecto en la productividad de una multitud de factores como el *hardware* y sus limitaciones, la complejidad de los programas, la experiencia del personal (tanto técnicos como usuarios), el entorno de desarrollo y programación, etc. Esto es lo que provoca que el método teórico de Putman no se convierta nunca en obsoleto. Se trata de ir cambiando los valores de la constante tecnológica a medida que cambian los entornos de desarrollo y programación. El problema es que conocer el valor correcto de  $C_k$  no es en absoluto sencillo.

Al principio, Putman indicaba una serie discreta de veinte valores de  $C_k$  que podían situarse entre 610 y 57.314. La  $C_k$ , pues, podía moverse desde un valor bastante bajo, 610, hasta otro que es casi cien veces más alto. Es decir, el papel de esta constante es totalmente fundamental en el modelo. 

En el año 1981, Putman presentaba por primera vez su modelo insertado en una herramienta automatizada denominada **SLIM**, que, a partir de diferentes preguntas, determina la constante tecnológica correspondiente.

El modelo de Putman, en su última versión, identifica la fórmula anterior como la denominada **ecuación del software** y, además, descompone la constante tecnológica  $C_k$  en dos nuevas constantes:

$$C_k = P/B^{0,333},$$

donde  $P$  es el **parámetro de productividad** (que sería de 28.000 en el caso de aplicaciones de la informática de gestión) y  $B$  es un **factor especial de habilidad** que se incrementa lentamente a medida que crecen la necesidad de integración, las pruebas, la garantía de calidad, la documentación y otras exigencias que disminuyen la productividad.

En la práctica, hoy el modelo de Putman sólo puede ser bien utilizado por aquellos que adquieren el SLIM, la herramienta informática que implementa el modelo completo. Esta herramienta se actualiza constantemente con las sucesivas adaptaciones de las nuevas tecnologías de desarrollo de *software*. El SLIM es capaz, después de una larga batería de preguntas, de determinar  $C_k$  y el resto de datos que

#### Lectura recomendada

Podéis consultar la obra siguiente:

L. Putman; W. Myers (1992). *Measures for excellence*. Englewood Cliffs: Yourdon Press.

#### Valores de $B$ habituales

En aplicaciones pequeñas de 5 a 15 KLOC,  $B = 0,16$  y en aplicaciones mayores, de más de 70 KLOC,  $B = 0,39$ .

interesan para la estimación de un proyecto. Pero la manera de llevarlo a cabo se encuentra en el interior de la herramienta informatizada.

En cualquier caso, es bueno saber que el modelo de Putman se probó y validó con una serie homogénea de proyectos del Departamento de Defensa de EE.UU. Las conclusiones a las que se llegó mostraban que el modelo es bastante correcto para proyectos muy grandes y que el resultado es mucho más impreciso y dudoso en el caso de proyectos pequeños y medios. !

Por otra parte, sin disponer de la herramienta informática SLIM, el estimador que quiera utilizar el modelo de Putman ha de intentar determinar por su cuenta una constante tecnológica ( $C_k$ ) de gran efecto en el resultado final: un trabajo que es prácticamente imposible sin el SLIM. !

### 2.3.4. Modelos compuestos: los modelos COCOMO de Barry W. Boehm

COCOMO es el modelo de construcción de costes más conocido y utilizado de los modelos algorítmicos compuestos que se basan sobre todo en datos estadísticos, pero también en ecuaciones analíticas y en un ajuste fruto de la opinión de expertos.

El modelo es obra de Barry W. Boehm y se publicó por primera vez en el libro de 1981 *Software Engineering Economics*. Después ha tenido varias versiones, como Ada COCOMO, desarrollado a mitad de los años ochenta para proyectos programados con Ada, y, sobre todo, el actual proyecto COCOMO II, en el que Boehm colabora en su desarrollo, en el Center for Software Engineering (CSE) de la Universidad del Sur de California (University of Southern California, USC).

Cabe mencionar que las diversas versiones del *software* tienen objetivos diferentes y que la última versión (COCOMO II) no sustituye en absoluto a las anteriores (COCOMO 81 y Ada COCOMO), que son válidas siempre que se utilicen los paradigmas de construcción de *software* y el ciclo de vida y las herramientas para los cuales se definieron.

Al ser el modelo más aceptado, famoso y utilizado, es fácil encontrar muchas implementaciones informatizadas como, por ejemplo, el COSTAR (*Software Cost Estimation Tool*), una herramienta ya clásica de la empresa Softstar Systems, que, a partir del COSTAR 5.0, implementa ya el COCOMO II.

#### El COCOMO clásico de 1981

El COCOMO clásico lo forman, en realidad, tres modelos diferentes, que tienen en cuenta diferentes grados de complejidad: !

1) El **COCOMO básico** es un modelo estático válido para obtener una estimación rápida del esfuerzo (meses-hombre) en función del tamaño (KLOC) al inicio del ciclo de vida.

COCOMO es un acrónimo de la expresión inglesa *Cost Constructive Model*.

WEB

En las páginas web del Center for Software Engineering (<http://sunset.usc.edu/cse/pub/tools/>) de la Universidad del Sur de California (USC) se pueden encontrar herramientas informatizadas que implementan los diferentes modelos COCOMO, tanto el de 1981 como el COCOMO II.

WEB


En la página web de la empresa Softstar Systems (<http://www.softstarsystems.com>), además de muchos enlaces interesantes, podéis obtener libremente versiones de demostración del modelo COSTAR.



2) El **COCOMO intermedio** añade al cálculo del esfuerzo en función del tamaño, el efecto de unos atributos que influyen en el coste (CDA), con los cuales se quiere tener en cuenta el tipo de aplicación y tecnología, las calificaciones y la experiencia del personal, el entorno de diseño y programación y las herramientas de las que dispone, etc.


3) El **COCOMO adelantado** incorpora todas las características de la versión intermedia, pero en lugar de evaluar los CDA con un único valor para todo el ciclo de vida, tiene en cuenta diferentes CDA para cada fase\* de la construcción del *software*.

\* El análisis, el diseño, la programación, las pruebas, etc.

Las ecuaciones del modelo COCOMO tienen siempre la forma general que mostramos a continuación: 

- $E = a \cdot L^b \cdot CDA$ ,
- $T = c \cdot E^d$ ,

donde  $E$  es el esfuerzo (en *personas-mes*),  $L$  son las líneas de código (en *KLOC*),  $T$  es el tiempo de desarrollo del proyecto (en meses) y  $CDA$  los *cost driven attributes* que, recordémoslo, no se utilizan en el modelo básico. Finalmente,  $a$ ,  $b$ ,  $c$  y  $d$  son coeficientes que el modelo proporciona como resultado del análisis de los datos de sesenta y tres proyectos realizados entre 1965 y 1980, escritos en cinco lenguajes diferentes, de tamaños que varían de 2 a 1.000 KLOC y con una productividad que se sitúa entre 28 y 250 LOC por persona-mes.

Además de los tres modelos, COCOMO tiene en cuenta varios tipos de proyecto, ya que no se obtienen los mismos datos de productividad en todos los casos. En la nomenclatura que utiliza Boehm, los tres tipos de proyecto son los que presentamos a continuación: 

a) **Orgánico\***. Proyectos relativamente pequeños y sencillos, con poca innovación tecnológica, donde trabaja un pequeño equipo formado por gente que tiene suficiente experiencia en el tipo de aplicación concreto y que, además, debe tener requerimientos poco rígidos.

\* En inglés, *organic mode*.

b) **Semiacoplado\***. Proyectos de nivel intermedio en tamaño, complejidad y sofisticación técnica, donde varios equipos diferentes colaboran para construir una aplicación con requerimientos medianamente rígidos.

\* En inglés, *semidetached*.

c) **Encajado\***. Proyectos de un tamaño y complejidad francamente elevados, donde intervienen varios equipos diferentes y los requerimientos, tanto de *hardware* como de *software*, son muy rígidos.

\* En inglés, *embedded mode*.

Se debe tener en cuenta que, en la informática de gestión, posiblemente sólo tengan sentido los proyectos orgánicos o semiacoplados y que, tal como dice Pressman, los proyectos encajados sugieren otro tipo de sistemas informáticos con requerimientos muy rígidos\*.

\* Como, por ejemplo, el sistema de control de navegación de un avión.

Los coeficientes que corresponden al modelo básico, a menudo más que suficiente para una primera estimación de pequeñas aplicaciones en la informática de gestión, se recogen en la tabla siguiente:

Coeficientes del modelo COCOMO básico				
Proyecto	Coeficientes			
	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
Orgánico	2,4	1,05	2,5	0,38
Semiacoplado	3,0	1,12	2,5	0,35
Encajado	3,6	1,20	2,5	0,32

En el modelo intermedio los coeficientes son los de esta otra tabla:

Coeficientes del modelo COCOMO intermedio				
Proyecto	Coeficientes			
	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
Orgánico	3,2	1,05	2,5	0,38
Semiacoplado	3,0	1,12	2,5	0,35
Encajado	2,8	1,20	2,5	0,32

En este modelo, además de no cambiar la relación entre esfuerzo y tiempo (coeficientes *c* y *d*, respectivamente), puede ser sorprendente la tendencia descendente del coeficiente *a*. Sería como si el esfuerzo disminuyera al ser más complejo el proyecto. Realmente no es así, ya que el modelo intermedio utiliza, además, los atributos que influyen en el coste (CDA).

En el modelo intermedio, los atributos que influyen en el coste (CDA) son quince parámetros que recogen características del producto que se va a realizar, del ordenador en el que se trabaja y que ejecutará la aplicación, del personal que forma el equipo del proyecto y del proyecto mismo. Cada uno de estos quince parámetros puede tomar diferentes valores: muy bajo, bajo, nominal, alto, muy alto y extraalto\* que varían en torno a la unidad. La tabla siguiente muestra los valores de los CDA.

\* En inglés, *very low, low, nominal, high, very high, y extra high*.

COCOMO intermedio: atributos que influyen en el coste (CDA)							
		Valores					
		Muy bajo	Bajo	Nominal	Alto	Muy alto	Extraalto
Atributos del producto							
RELY	Fiabilidad requerida	0,75	0,88	1,00	1,15	1,40	-
DATA	Volumen de la base de datos	-	0,94	1,00	1,08	1,16	-
CPLX	Complejidad	0,70	0,85	1,00	1,15	1,30	1,65
Atributos del ordenador							
TIME	Limitaciones de tiempo de ejecución	-	-	1,00	1,11	1,30	1,65


COCOMO intermedio: atributos que influyen en el coste (CDA)							
		Valores					
		Muy bajo	Bajo	Nominal	Alto	Muy alto	Extraalto
<b>Atributos del ordenador</b>							
STOR	Limitaciones de volumen de memoria	-	-	1,00	1,06	1,21	-
VIRT	Volatilidad de la máquina virtual	-	0,87	1,00	1,15	1,30	-
TURN	Tiempo de respuesta	-	0,87	1,00	1,07	1,15	1,65
<b>Atributos del personal</b>							
ACAP	Capacidad de análisis	1,46	1,19	1,00	0,86	0,71	1,65
AEXP	Experiencia en la aplicación	1,29	1,13	1,00	0,91	0,82	1,56
PCAP	Capacidad de programación	1,42	1,17	1,00	0,86	0,70	-
VEXP	Experiencia en la máquina virtual	1,21	1,10	1,00	0,90	-	-
LEXP	Experiencia en los lenguajes de programación	1,14	1,07	1,00	0,95	-	-
<b>Atributos del proyecto</b>							
MODP	Uso de prácticas modernas	1,24	1,10	1,00	0,91	0,82	-
TOOL	Uso de herramientas de <i>software</i>	1,24	1,10	1,00	0,91	0,83	-
SCED	Exigencias de planificación	1,23	1,08	1,00	-	1,10	-

Es necesario saber también que existen varias hipótesis que están en la base del modelo COCOMO.

#### Algunas hipótesis del modelo COCOMO

Algunas de las hipótesis que utiliza el modelo COCOMO son, entre otras, las que mencionamos a continuación:

- No se tienen en cuenta los comentarios en el recuento de las KLOC.
- Se admite la equivalencia siguiente: 1 mes-hombre son 152 horas de trabajo.
- Se considera que los requerimientos son estables.
- Se acepta que el proyecto está bien gestionado.

En resumidas cuentas, el modelo COCOMO de Boehm es el más serio y completo de los que existen, aunque los resultados que se obtienen pueden haber quedado obsoletos por la evolución y los cambios que ha sufrido la informática en los últimos veinte años. 

#### El COCOMO II de los años noventa y a partir del 2000

El nuevo modelo COCOMO II tiene como objetivo principal desarrollar un modelo de estimación de costes y planificación del *software* especialmente adecuado para los ciclos de vida utilizados en los años noventa y a partir del 2000. Aquí no entraremos en la exposición de los detalles, ya que hemos optado por referirnos al ciclo de vida tradicional en la construcción del *software* y este ciclo ya está bien cubierto por el COCOMO clásico.

El modelo COCOMO II todavía se encuentra en fase de elaboración y de prueba y, además, es muy complejo. De hecho, como ya pasaba en el COCOMO clásico, el COCOMO II incluye tres modelos que corresponden a diferentes fases y modalidades del futuro ciclo de vida:

1) **Modelo de composición de aplicaciones\***: incluye el uso de prototipos para disminuir los riesgos potenciales que surgen con las interfaces gráficas de usuario típicas de herramientas RAD y otras herramientas actuales de productividad y de la orientación a objetos. En este modelo se definen unos puntos objeto\*\* que vendrían a ser una adaptación y modernización de los puntos de función de Albrecht ya vistos.

\* En inglés, *Application Composition Model*.  
\*\* En inglés, *Object Points*.

2) **Modelo de diseño primerizo\***: intenta obtener una primera aproximación en las fases iniciales del ciclo de vida, cuando todavía se conocen pocas de las características y datos definitivos del proyecto. Utiliza como primitivas de salida tanto las líneas de código como los clásicos puntos de función de Albrecht sin ajustar (TUPP).

\* En inglés, *Early Design Model*.


3) **Modelo de postarquitectura\***: versión más completa, corresponde a la modernización del COCOMO tradicional de 1981. Se aplica cuando se considera que el proyecto dispone ya de requerimientos estables. Por otra parte, también utiliza como primitivas de salida las líneas de código y los puntos de función de Albrecht sin ajustar (TUPP) y, además, tiene en cuenta indicadores de la reutilización de *software*, cinco factores de escala y hasta diecisiete factores específicos diferentes.

\* En inglés, *Post-Architecture Model*.

Es importante destacar cómo en el proyecto COCOMO II, en el modelo postarquitectura, se pueden utilizar las nuevas líneas de código (NSLOC\*) cuando se trata de estimar los costes de una aplicación nueva, o las líneas de código adaptadas (ASLOC\*\*) cuando se trata de estimar una aplicación en la cual se utilizará básicamente el diseño o el código de otras aplicaciones ya disponibles. Es decir, el COCOMO II incluye ya como aspecto central el fenómeno de la reutilización del *software*.

\* NSLOC es la sigla de la expresión inglesa *New Source Lines of Code*.  
\*\* ASLOC es la sigla de la expresión inglesa *Adapted Source Lines of Code*.


Por otra parte, también es necesario destacar el eclecticismo del COCOMO II al utilizar como primitivas de salida tanto las tradicionales líneas de código que ya utilizaba el COCOMO clásico de 1981, como los puntos de función de Albrecht o su evolución natural en el mundo de la orientación a objetos, los puntos objeto.

Tal vez es prematuro apuntarlo, pero, cuando esté terminado, COCOMO II se puede convertir en el nuevo modelo de referencia, tal como lo ha sido el COCOMO clásico durante tantos años. 

### 2.3.5. El uso de estándares de productividad

No es fácil, en la práctica, encontrar cuál es el modelo que más se ajusta a la realidad del proyecto informático que todavía está por empezar y que

preocupa al jefe de proyecto que debe llevar a cabo la estimación de las cargas y los costes.

Como herramienta para estudiar el proceso de construcción del *software*, los modelos que ofrecen fórmulas son interesantes, pero el gran inconveniente que tienen es que la mayoría parte de una base de datos de proyectos lo suficientemente antiguos como para que sus resultados no siempre sean útiles. 

Por esto, la práctica profesional de cada día ha provocado que muchas instalaciones\*, a pesar de tener como referencia los datos que ofrecen los modelos, hayan decidido adoptar un sistema mucho más simple y menos complicado: elaborar estándares de productividad propios.

\* Generalmente, las instalaciones mayores y con más experiencia.


Para tener estándares de productividad propios es necesario recoger datos de proyectos anteriores de la misma instalación y establecer cuál es el esfuerzo que corresponde a las diferentes actividades que pueden formar parte del proceso de construcción del *software*. Para realizar la estimación de proyectos nuevos se parte de estos datos convertidos en estándar de referencia.

#### A la hora de estimar el esfuerzo...

... y planificar las tareas de programación es útil saber (haciendo la media de los datos de proyectos anteriores) cuánto ha tardado en programarse y probarse, por ejemplo, una transacción que saca tres formularios diferentes por pantalla y que consulta y actualiza cinco tablas de la base de datos.

La idea general es cuantificar varias unidades de medida que tengan o que parezcan tener una cierta relación con el esfuerzo (el número de formularios de pantalla, el número de tablas de la base de datos, el número y la complejidad de los tratamientos, etc.) y/o disponer de una larga serie de actividades tipo habituales en el proceso de construcción de *software*, de las cuales conocemos el coste (extraído, evidentemente, de proyectos anteriores).

En el fondo es como construir una especie de características funcionales como las de los puntos de función de Albrecht, pero teniendo en cuenta las condiciones concretas de una instalación determinada y del tipo de aplicación que se suele construir. En lugar de cuantificar cada actividad o conjunto de estas características funcionales en líneas de código o puntos de función y buscar modelos que conviertan estas métricas en esfuerzo (meses-hombre), es suficiente disponer directamente, para cada actividad, del esfuerzo estándar que ha requerido en otros proyectos anteriores.

Podéis ver las características funcionales de los puntos de función de Albrecht en el subapartado 1.4.2 de este módulo didáctico. 

#### Uso de estándares de productividad

Es imprescindible la utilización de estándares de productividad cuando, por ejemplo, se utiliza el *outsourcing* ('externalización') y se encarga la programación de una nueva aplicación a una empresa externa. En este caso, es necesario tener datos claros que permitan a las dos empresas (la que quiere el proyecto y la que va a efectuar la programación) ponerse de acuerdo en el coste que se pagará por el trabajo de programación que se debe realizar.

También, al margen de los estándares propios de cada instalación, en la bibliografía técnica se encuentran las **recomendaciones prácticas a primera vista**.

#### Lectura recomendada

Las recomendaciones de Robert O. Grady del grupo de métricas de Hewlett Packard se pueden encontrar en la obra siguiente:


**R.O. Grady** (1992). *Practical software metrics for project management and process improvement* (vol. 2, pág. 13-20). Englewood Cliffs: Prentice Hall (Hewlett Packard Professional Books).

Como ya hemos dicho, se trata de resúmenes del saber acumulado por especialistas prestigiosos que pueden actuar como referencia general futura para los estándares propios de una instalación que empieza.

Tal como recuerda Caper T. Jones en un artículo, las recomendaciones prácticas a primera vista siguen siendo populares, pero nunca son exactas y no substituyen a los métodos formales de estimación del *software*.

En cualquier caso, es interesante saber que, según la opinión de un experto como Jones, se dan las equivalencias prácticas siguientes:

- 1 FP = 100 LOC.
- FP elevado a 0,4 = meses de desarrollo.
- $FP/150$  = número de personas que son necesarias para el desarrollo.
- $FP/500$  = número de personas necesarias para el mantenimiento futuro.
- FP elevado a 1,15 = número de páginas de documentación.
- FP elevado a 1,2 = número de casos de prueba que se realizan.
- FP elevado en 1,25 = potencial de errores (en proyectos nuevos).
- FP elevado a 0,25 = número de años que seguirá en uso la aplicación.

Las recomendaciones prácticas a primera vista pueden ser más generales, como las que, de acuerdo con Jones, os ofrecemos a continuación: 

- Si no se vigilan los requerimientos crecientes, los usuarios conseguirán que aumenten en un 1% cada mes de desarrollo del proyecto: en un proyecto de dos años se da, al final, un 24% más de requerimientos de los que existían al principio.
- Cada inspección o control de errores encuentra y también corrige el 30% de los errores que se dan.
- Los proyectos invierten aproximadamente un 18% de su esfuerzo total en determinar los requerimientos y efectuar las especificaciones, un 19% en el diseño, un 34% en la codificación y un 29% en las pruebas.
- Para mantener y mejorar el *software*, se llevan a cabo esfuerzos dos o tres veces superiores a los necesarios para crearlo.
- Se da aproximadamente un defecto sin detectar por cada diez que se encuentran en las pruebas antes de distribuir una versión del *software*.

#### Lectura recomendada

C.T. Jones (1996). "Software Estimating Rules of Thumb". *IEEE computer* (marzo, pág. 116-118).


#### Según estas equivalencias...

... una aplicación de 1.000 FP supone 16 meses de desarrollo, 6,6 personas en el proyecto y hasta 106 meses de esfuerzo. (En este sentido, si estos datos no encajan del todo con la productividad de dos horas para implementar un punto de función tal como el mismo Jones proponía en un artículo de mayo de 1996 que hemos recordado en el apartado 1.4.2 es, simplemente, un ejemplo más de las perplejidades que este asunto de la estimación de los costes del *software* desencadena, tal como ya hemos comentado en el subapartado 1.4.4.)

## 2.4. Práctica real de la estimación

En general se suelen considerar más serios y seguros los modelos que incorporan fórmulas, pero no debemos olvidar que, muy a menudo, eso no resuelve en absoluto el problema real de la estimación.

Los modelos algorítmicos nos permiten, por ejemplo, saber cuál será el esfuerzo necesario para obtener un determinado número de líneas de código. También nos muestran los meses empleados en la construcción del proyecto, el número de profesionales que deberían estar involucrados, el número de errores que se estima obtener, el número de páginas de documentación que se deben llevar a cabo, etc. Sin embargo, todo parte de un determinado número global de líneas de código o de la estimación del total de puntos de función que se deben implementar.


Si bien, lo cierto es que, aun así, los métodos algorítmicos no resuelven en absoluto el problema de la estimación porque nadie nos dice de dónde sale el número de LOC que utilizaremos en las fórmulas, ni la persona que lo determina. 

Por otra parte, en la práctica, disponer de una cifra de esfuerzo única no es en absoluto el camino más fácil para las etapas posteriores en la gestión de un proyecto informático: la planificación y el seguimiento del proyecto. Saber sólo que un proyecto debe durar seis meses no nos permite conocer si, por ejemplo, a tres meses del comienzo se van cumpliendo o no las previsiones realizadas (faltan detalles).

Si tenemos en cuenta que, posteriormente a la estimación de costes, es necesario planificar en el tiempo y efectuar el seguimiento de todo el conjunto de actividades o tareas que forman el proyecto informático, el hecho de tener una estimación global única es poco útil. Es mucho más interesante un planteamiento ascendente que nos permita, antes de efectuar la estimación, descomponer el proyecto informático en diferentes actividades. Este paso previo lo convierte en mucho más planificable y controlable.

Existen varias maneras de descomponer un proyecto informático y, aunque aquí sólo nos interesa una, mencionamos a continuación los nombres más habituales:

a) **WBS**: descomposición estructural de los trabajos que se deben realizar, es decir, la lista estructurada de todas las actividades y tareas de un proyecto.



Podéis ver el seguimiento del proyecto informático en el apartado 2 del módulo “La gestión de un proyecto informático” de esta asignatura.

WBS es la sigla de la expresión inglesa *Work Breakdown Structure*.

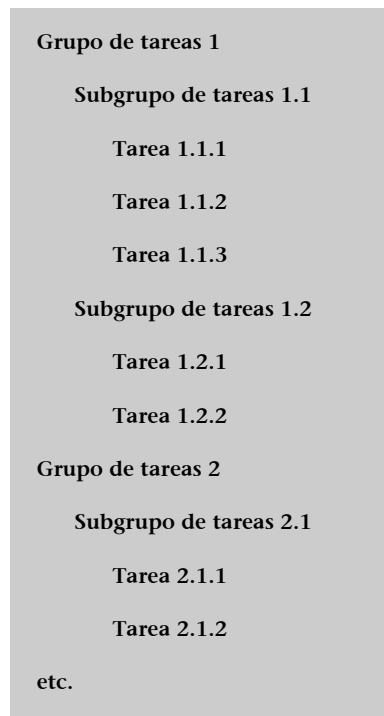
b) **PBS**: resultado que se espera obtener, es decir, la descomposición en partes del producto final. En el caso del *software* de aplicación podrían ser tablas de la base de datos, formularios, programas, módulos, transacciones, etc. Cabe mencionar que la PBS es de gran interés para el diseño técnico, pero no tanto para la estimación de costes y la planificación de un proyecto.

PBS es la sigla de la expresión inglesa *Product Breakdown Structure*.

c) **OBS**: descomposición de la organización para atender a todas las tareas que componen la WBS. De hecho, cada elemento terminal de la OBS (posiblemente, cada uno de los profesionales que intervienen en el proyecto) ha de quedar encargado de una o más tareas, las cuales forman los elementos finales de la WBS.

OBS es la sigla de la expresión inglesa *Organization Breakdown Structure*.

Aunque es muy habitual utilizar una descomposición en forma de árbol para cualquier descomposición estructural (ya sea de tareas, producto u organización) a menudo es suficiente con una lista estructurada en forma jerárquica, como el ejemplo siguiente para una teórica WBS:




En la práctica, la **estimación de costes de un proyecto informático** se efectúa, pues, a partir de una descomposición del proyecto en las diferentes tareas o actividades que se deben realizar (WBS) y la estimación individual del esfuerzo que cuesta cada tarea concreta.

Para llevar a cabo esta descomposición WBS, el jefe de proyecto utiliza su experiencia y, sobre todo, la analogía con otros proyectos más o menos parecidos, para establecer una posible lista de tareas que es necesario llevar a cabo en el proyecto.

En este caso, la tradición establece que la estimación de cada tarea se realice, no en líneas de código o puntos de función, sino directamente en unidades de esfuerzo, por ejemplo, personas-día. Se trata de utilizar los estándares de productividad de los que se ha hablado o, en caso de que no existan, el sentido común y las recomendaciones prácticas a primera vista.

Podéis ver los estándares de productividad y las recomendaciones prácticas a primera vista en el subapartado 2.3.5 de este módulo didáctico.



La incertidumbre de este proceso práctico de estimación de costes se encuentra en el posible acierto de la descomposición WBS, además de en la bondad de los estándares de productividad y en las recomendaciones prácticas disponibles. Cabe mencionar que los errores de estimación son muy frecuentes y, en cierta manera, completamente inevitables. 

### 2.4.1. Un ejemplo: contabilidad sencilla en un PC

Aplicaremos el método que hemos descrito en el subapartado anterior a un ejemplo sencillo, una pequeña aplicación de contabilidad que podría quedar suficientemente descrita de la manera siguiente: se quiere implementar una miniaplicación contable destinada a ejecutarse en un microordenador personal tipo PC compatible. Los requerimientos que se exigen se exponen a continuación:


- 1) La aplicación debe soportar un sistema de contabilidad general por partida doble, susceptible de incorporar también cuentas corrientes de clientes, a partir de la utilización del Plan General de Contabilidad.
- 2) Se utilizarán unos códigos de cuenta y subcuenta de hasta ocho dígitos, organizados jerárquicamente en cuatro niveles. Los niveles quedan determinados respectivamente por la primera cifra (grupo), las dos primeras cifras (subgrupo), las tres primeras cifras (cuentas) y todo el código (subcuenta):

Plan General de Contabilidad		
Nivel	Código	Concepto
Grupo	4	Acreedores y deudores por operaciones de tráfico
Subgrupo	43	Clientes
Cuenta	430	Clientes
Cuenta	435	Clientes de cobro dudoso
Subcuenta	43512345	Joan Belloch Pi (cliente número 12345)

- 3) Los movimientos afectan siempre a las subcuentas (ocho dígitos) y se deben hacer repercutir automáticamente en pirámide a los niveles de orden superior (tres, dos y un dígitos, respectivamente).
- 4) Los asentamientos han de tener la opción de múltiples contrapartidas hasta un máximo de diez líneas por asentamiento.
- 5) La aplicación incluirá necesariamente:
  - a) Tratamientos interactivos:
    - Gestión del plan de cuentas (alta, baja, modificación y consulta).
    - Entrada de movimientos contables.
    - Consulta por pantalla del saldo y los movimientos de una subcuenta determinada (extracto).

## b) Tratamientos diferidos:

- Listado del plan de cuentas.
- Listado del diario.
- Listado del mayor (o de unas cuentas concretas).
- Listado mensual de Resultados de pérdidas y ganancias.
- Listado mensual del Balance de comprobación.
- Listado mensual del Balance de situación.
- Cierre del mes.

Cabe mencionar que a menudo no se dispone ni siquiera de una descripción como ésta que, a pesar de su brevedad, es bastante completa si se conoce qué es una contabilidad realizada por ordenador (es decir, si el jefe de proyecto tiene experiencia en este tipo de aplicaciones). Enunciados como éste (que, de hecho, no existen en la realidad), permiten efectuar muy fácilmente una descomposición de tareas (WBS), a menudo centrada en la descomposición de tratamientos (que no de programas) que prácticamente ya da el enunciado. 

En cualquier caso, es evidente que este ejemplo es limitado y que la contabilidad que hemos presentado es sólo mensual. Falta lo que podría ser una segunda fase del proyecto, con el cierre del ejercicio anual y la apertura del nuevo ejercicio.

El resultado de la estimación se indica a continuación con la descomposición estructural de las tareas que se deben llevar a cabo (WBS) y, para cada tarea, la estimación de esfuerzo en personas-día:

Estimación de costes del proyecto				
Descomposición estructural de tareas		Personas-día		
		JP	A	P
<b>Gestión del proyecto</b>				
01	Estimación y planificación del proyecto por primera vez	1		
02	Seguimiento, control y recalificación del proyecto (dos meses)	3		
<b>Análisis y diseño de arquitectura</b>				
03	Análisis funcional y especificación		3	
04	Diseño de la base de datos		1	
05	Diseño de formularios y listados		2	
<b>Tratamiento de gestión del plan de cuentas</b>				
06	Redacción del cuaderno de cargas		0,5	
07	Programación			2
<b>Tratamiento de la entrada de movimientos contables</b>				
07	Redacción del cuaderno de cargas		1	
08	Programación			4
<b>Tratamiento de la consulta de saldo y movimientos de subcuentas</b>				
09	Redacción del cuaderno de cargas		0,5	
10	Programación			3

Estimación de costes del proyecto				
Descomposición estructural de tareas		Personas-día		
		JP	A	P
<b>Tratamiento del listado del plan de cuentas</b>				
11	Redacción del cuaderno de cargas		0,5	
12	Programación			2
<b>Tratamiento del listado de diario</b>				
13	Redacción del cuaderno de cargas		0,5	
14	Programación			2
<b>Tratamiento del listado de mayor</b>				
15	Redacción del cuaderno de cargas		0,5	
16	Programación			3
<b>Tratamiento del listado de la cuenta de pérdidas y ganancias</b>				
17	Redacción del cuaderno de cargas		1	
18	Programación			3
<b>Tratamiento del listado de Balance de comprobación</b>				
19	Redacción del cuaderno de cargas		1	
20	Programación			3
<b>Tratamiento del listado de Balance de situación</b>				
21	Redacción del cuaderno de cargas		0,5	
22	Programación			2
<b>Tratamiento del cierre del mes</b>				
23	Redacción del cuaderno de cargas		1	
24	Programación			3
<b>Prueba de integración del sistema</b>				
25	Integración de los programas y prueba general del sistema		5	5
<b>Documentación</b>				
26	Documentación general del sistema		3	
27	Diseño del sistema de ayudas ( <i>help</i> )		2	
28	Implementación del sistema de ayudas ( <i>help</i> )			4
<b>Cierre del proyecto</b>				
29	Cierre del proyecto y recogida final de datos	1		
	<b>Total (días)</b>	<b>5</b>	<b>23</b>	<b>36</b>

Con vistas a la posterior planificación del proyecto con la asignación de recursos, es decir, la asignación de tareas a personas concretas (relación del OBS con la WBS), se han considerado qué tareas forman parte del trabajo del jefe de proyecto (JP), el analista (A) o el programador (P). En la práctica puede haber muchos más profesionales involucrados, pero los tres niveles mencionados responden a tres niveles diferenciados de sueldo en la práctica profesional actual y son una buena aproximación para una futura estimación de los costes


monetarios. Posiblemente, en una aplicación tan pequeña sólo intervenga un único profesional, sin embargo, con vistas a la generalización para casos mayores, vale la pena mantener esta separación de las tareas y los profesionales.

Se ha considerado también que el analista, una vez realizado el análisis y el diseño de arquitectura, elabora las especificaciones individuales y concretas de cada programa que se debe realizar, que en la profesión se denomina *cuaderno de cargas* y es la documentación en la que se basa el programador para efectuar su tarea. En esta documentación se describen el tratamiento, los formularios y los listados que se deben utilizar en el programa y la parte de la base de datos que ha de utilizar el programa.

A partir del cuaderno de cargas, el programador diseña, codifica y prueba el programa individualmente (esta necesidad de probar cada programa explica los días que se han estimado para la programación: es necesario realizar un juego de prueba, comprobar y documentar los resultados, etc.). La tarea del programador termina con la documentación exhaustiva del programa y de los resultados conseguidos con el juego de prueba. Queda para el final la prueba de la integración global de toda la aplicación.

Se ha considerado que el jefe de proyecto dedica dos o tres horas cada semana al seguimiento y control del proyecto (no parece en absoluto que en un proyecto tan pequeño sean necesario más tiempo). El proyecto, del orden de 60 personas-día de trabajo, puede tener un tiempo de desarrollo de un par de meses (cabe recordar que en un mes se dan, aproximadamente, unos 22 días laborables), hecho que justifica las tres jornadas del jefe de proyecto en seguimiento y control.

Se ha realizado una estimación optimista de las tareas de programación pensando en herramientas modernas como las herramientas RAD ya mencionadas y, también, en una buena asignación de los recursos (por ejemplo, se puede pensar en sólo dos días para la programación y prueba del listado del Balance de situación si lo efectúa el mismo programador que se ha encargado del Balance de comprobación). A pesar de este optimismo, se ha considerado que ninguna tarea se puede estimar en menos de media jornada de trabajo, que en la mayoría de casos es una estimación muy realista en el ámbito del trabajo profesional asalariado.




Podéis ver las herramientas RAD en el subapartado 1.4.1 de este módulo didáctico.

## Resumen

La actividad de la primera estimación de costes de un proyecto informático es y será siempre problemática, sobre todo por el hecho de tener que estimar el esfuerzo necesario para implementar una aplicación cuando todavía no se conocen con detalle las funcionalidades que se han de cubrir.

La mayoría de métodos de estimación utilizados hasta ahora se centran en dos grandes unidades de medida: el número de líneas de código (LOC) para medir el tamaño y los puntos de función (FP) para medir las funcionalidades. Se da una relación empírica entre LOC y FP, y la tendencia actual parece que es el predominio de los puntos de función e, incluso, la creación de una nueva unidad para la orientación a objetos: los puntos objeto.

Cabe recordar que el hecho de que el esfuerzo se mida en personas-mes no permite intercambiar mecánicamente personas y meses, ya que el esfuerzo para desarrollar un proyecto informático tiene una distribución característica en el tiempo como la que marca la curva de Rayleigh/Norden.

Los diferentes modelos de estimación de costes se encuentran hoy un poco obsoletos y todavía no se han desarrollado completamente otros modelos que recojan los cambios de herramientas de desarrollo y de productividad que ha alcanzado la informática en la última década. De momento, el IFPUG actualiza la interpretación de los puntos de función de Albrecht creados en el año 1979 y el modelo COCOMO de Barry W. Boehm, que ha dominado el campo de la estimación de costes desde 1981, se está actualizando en un COCOMO II que tenga en cuenta los nuevos ciclos de vida y la nueva informática. 



## Actividades

1. Buscad en cualquier portal o buscador de Internet el nombre COCOMO y podréis ver la gran cantidad de referencias que se dan sobre el modelo más famoso y conocido de la estimación algorítmica de costes en proyectos informáticos. En muchas, se pueden obtener herramientas informáticas que ayudan a utilizar el modelo de Barry W. Boehm.
2. Si tenéis acceso a alguna instalación o entidad que construya *software* de aplicación en la informática de gestión, haced lo posible para conocer los estándares de productividad que aplica o, si no existen, el sistema de estimación de costes que utiliza. Puede ser muy útil comparar estos estándares de productividad con los que menciona la bibliografía técnica y que se han comentado en este módulo.
3. Después de haber consultado cuáles son los precios-hora habituales de mercado para un jefe de proyecto, un analista y un programador, convertid la estimación de esfuerzo del ejemplo de la contabilidad sencilla en un PC de personas-día a pesetas (o euros). Teniendo en cuenta que se trata de una aplicación muy sencilla, la cifra final obtenida os dará una idea del coste que representa desarrollar proyectos de construcción de *software* a medida y os hará entender la tendencia actual a la utilización de paquetes (*packages*) y a la reutilización de todo el *software* que se pueda en la construcción de aplicaciones nuevas.

## Ejercicios de autoevaluación

1. ¿Es cierto que la estimación inicial de costes de un proyecto informático con métricas basadas en las funcionalidades (puntos de función de Albrecht, por ejemplo) es más objetiva que la estimación efectuada con métricas basadas en el tamaño del proyecto (KLOC, por ejemplo)?
2. Cuando un proyecto informático se retrasa con relación a su planificación inicial, ¿se puede decir que la causa es siempre un defecto de la estimación de costes?
3. ¿Qué se incluye cuando se habla de medir el tamaño de un proyecto informático en líneas de código?
4. ¿En un mismo proyecto, el número de líneas de código escritas en COBOL es mayor o más pequeño que las escritas en un lenguaje RAD?
5. ¿Cómo se miden los puntos de función de Albrecht?
6. ¿A partir de los años noventa, qué se debe realizar para utilizar bien el modelo teórico de Putman?
7. ¿El modelo COCOMO de Boehm, de 1981, es un ejemplo clásico de modelo de estimación histórico?
8. ¿Son fiables los modelos algorítmicos de estimación de costes?
9. Con vistas a la práctica de la gestión de un proyecto informático, ¿cuál es el problema principal que plantean los modelos tradicionales de estimación?
10. Una vez obtenida una descomposición de tareas de un proyecto (WBS), ¿se podrían utilizar los modelos algorítmicos para efectuar la estimación de costes de cada una de las tareas?

## Solucionario

### Ejercicios de autoevaluación

1. El hecho de utilizar KLOC o puntos de función es, sólo, escoger una métrica determinada. El problema de la estimación se encuentra en la dificultad de acertar el alcance de un proyecto informático (tamaño y/o funcionalidades) justo cuando empieza. Por lo tanto, la estimación basada en KLOC no es más objetiva que la basada en puntos de función de Albrecht.
2. La causa de un retraso en el proyecto no es siempre una mala estimación de costes. Pueden darse otros problemas de desarrollo como, por ejemplo, un programador incompetente que no realice su trabajo a tiempo, aunque haya sido bien estimado. En el caso de la informática de gestión no se debe olvidar nunca el problema de los requerimientos crecientes.
3. Cuando se hace referencia a medir el tamaño de un proyecto informático en líneas de código, se incluyen todas las actividades del proyecto y no sólo la codificación. Es decir, es necesario tener en cuenta el trabajo que conlleva el estudio de oportunidad, el análisis de requerimientos, el diseño, la codificación, la integración, todo tipo de pruebas, las actividades de control y gestión de la calidad, la documentación externa e interna, etc.
4. En un mismo proyecto, el número de líneas de código escritas en COBOL es mayor que el número de líneas escritas en un lenguaje RAD.
5. Para medir los puntos de función de Albrecht se efectúa un recuento de las características funcionales (entradas, salidas, consultas, ficheros internos e interfaces), teniendo en cuenta si tienen una complejidad baja, media o alta. Una vez ponderadas estas cifras con los pesos que fijó Albrecht, se obtienen los puntos de función sin ajustar, que se corrigen teniendo en cuenta las características generales del sistema (catorce valores que deben estimarse, cada uno entre 0 y 5) que determinan el grado total de influencia, hecho que da el valor final de los puntos de función mediante las fórmulas siguientes:
  - $PCA = 0,65 + (0,01 \cdot TDI)$ .
  - $FP = TUPP \cdot PCA$ .
6. Para utilizar el modelo teórico de Putman correctamente a partir de los años noventa, se debe adquirir la herramienta de ayuda SLIM, que incorpora todo el modelo y lo mantiene actualizado.
7. El modelo COCOMO, aunque es de 1981, es un modelo que forma parte de la historia, pero no es un ejemplo clásico de modelo de estimación histórico, sino el paradigma de los modelos de estimación denominados *modelos compuestos*.
8. Los modelos algorítmicos de estimación de costes son fiables si el proyecto que se quiere evaluar es semejante a los proyectos que sirvieron para establecer el modelo y los coeficientes de sus fórmulas. Desgraciadamente, los modelos clásicos de estimación de costes se obtuvieron hace ya muchos años y la informática ha cambiado demasiado para esperar que los datos obtenidos de los modelos sean completamente aplicables hoy en día.
9. Los modelos tradicionales de estimación de costes suelen dar una única cifra para todo el proyecto, tanto con respecto a LOC, FP, o al esfuerzo necesario para implementar el proyecto. En la práctica, la planificación y el seguimiento se realizan mejor si se dispone de una descomposición del proyecto en diferentes actividades y tareas que se puedan planificar y controlar una por una.
10. Los modelos algorítmicos se obtuvieron en proyectos más bien muy grandes y, por lo tanto, sus fórmulas no son aplicables a tareas pequeñas. Así pues, no podemos utilizar estos modelos para llevar a cabo la estimación de costes de cada una de las tareas en las que hayamos descompuesto un proyecto.

## Glosario

**CDA** *f* Cada una de las variables de los modelos empíricos que se cree que inciden en el coste.

**COCOMO** *m* Modelo algorítmico de estimación de costes muy conocido y utilizado, desarrollado por Barry W. Boehm a partir de 1981. Actualmente está en curso una revisión con el nombre COCOMO II.

**DET** *m* Número de elementos, o campos de datos, de un fichero.



**DSI** *f* Instrucciones de código fuente entregadas realmente.

**EI** *f* Entradas externas que hacen referencia a los tratamientos que procesan datos o información de control que se introduce a la aplicación desde fuera de sus límites.

**EIF** *f* Ficheros de interfaces externas.

**EO** *f* Salidas externas.

**EQ** *f* Consultas externas.

**estándares de productividad** *m* Datos propios de una instalación sobre la productividad en la construcción de *software*, obtenidos a partir de proyectos anteriores, tipificando los resultados.

**FP** *m* Puntos de función.

**hombre-mes o persona-mes** *m* y *f* Unidad de medida que representa un mes de trabajo de una persona del equipo de profesionales que trabaja en un proyecto informático.

**IFPUG** *m* Grupo internacional de usuarios de los puntos de función.

**indicador de software** *m* Combinación de diferentes métricas que proporciona una visión resumida del producto o del proceso de construcción de *software*.

**KDSI** *f* Miles de DSI.

**KLOC** *f* Miles de líneas de código.

**LIF** *m* Ficheros lógicos internos.

**líneas de código** *f* Unidad de medida (que a menudo se cuenta de mil en mil, en KLOC) que incluye en el recuento, además de las líneas codificadas, las directivas de compilación, las declaraciones de las estructuras de datos y el código ejecutable; no se cuentan los comentarios y las líneas generadas automáticamente por el entorno de programación o que son fruto de reutilización.

**sigla:** LOC

**LOC** *f* Véase líneas de código

**métrica de software** *f* Asignación de valor a un atributo de una entidad propia del *software*, ya sea un producto o un proceso.

**métricas de productividad** *f* Métricas que recogen la eficiencia del proceso de producción de *software* y relacionan el *software* construido con el esfuerzo que ha costado obtenerlo.

**NCSS** *f* Líneas de código fuente que no tienen en cuenta los comentarios.

**NSLOC** *f* Nuevas líneas de código fuente.

**OBS** *f* Véase organization breakdown structure

**organization breakdown structure** *f* Descomposición de la organización para atender todas las tareas del proyecto.

**sigla:** OBS

**PBS** *f* Véase product breakdown structure

**PCA** *m* Ajuste por la complejidad del proceso.

**product breakdown structure** *f* Descomposición en partes del producto final: tablas de la base de datos, formularios, programas, módulos, transacciones, etc.

**sigla:** PBS

**puntos de función** *m* Métrica creada por Albrecht en el año 1979 que pretende recoger las funcionalidades como medida de la dificultad de construir un *software*. El método de cálculo realiza el recuento de unas características funcionales (entradas, salidas, consultas, ficheros internos e interfaces), teniendo en cuenta la complejidad, y, después, corrige el valor obtenido mediante la estimación de hasta catorce características generales que determinan el entorno de trabajo y del proyecto.

**RAD** *m* Véase rapid application development

**rapid application development** *m* Desarrollo rápido de aplicaciones que permiten nuevas herramientas como, por ejemplo, PowerBuilder, Visual Basic o Delphi, que seguramente son las más utilizadas actualmente.

sigla: RAD

**RET** *m* Número de registros elementales diferentes.

**TDI** *m* Grado total de influencia.

**TUFP** *m* Total de puntos de función sin ajustar.

**WBS** *f* Véase work breakdown structure

**WIMP** *m y f* Sigla de *windows, icons, mouse y pop-up menu*, es decir, ventanas, iconos, ratón y menús desplegables. Es el paradigma de la manera más moderna y más evolucionada para construir diálogos persona-ordenador.

**work breakdown structure** *f* Descomposición estructural de los trabajos que se deben realizar, es decir, la lista estructurada de todas las actividades y tareas de un proyecto.

sigla: WBS

## Bibliografía

**Albrecht, A.J.** (1979). "Measuring Application Development Productivity" (pág. 83-92). *Proceedings joint SHARE/GUIDE/IBM applications development symposium* (14 a 17 de octubre de 1979). Monterrey.

**Albrecht, A.J.; Gaffney Jr., J.E.** (1983). "Software Function, Source Lines of Code, and Development Effort Prediction: **A Software Science Validation**". *IEEE transactions on software engineering* (vol. SE-9, núm. 6, noviembre, pág. 639-648).

**Boehm, B.W.** (1981). *Software Engineering Economics*. Englewood Cliffs: Prentice Hall.

**Boehm, B.W.** y otros (1995). "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0". J.D. Arthur y S.M. Henrio (ed.). *Annals of Software Engineering, Special Volume On Software Process and Product Measurement* (vol. 1, pág. 57-94). Amsterdam: J.C. Baltzer, AG, Science Publishers.

**Brooks, F.** (1975). *The Mythical Man-Month*. Reading: Addison-Wesley.

**Cantor, M.R.** (1998). *Object-oriented Project Management with UML*. Nueva York: John Wiley & Sons.

**Grady, R.O.** (1992). *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliffs: Prentice Hall (Hewlett Packard Professional Books).

**Horowitz E.** y otros (1997). *USC COCOMO II, 1997 Reference Manual*. Los Ángeles: USC- SCE Technical Report.

**IEEE** (1993). *IEEE Standard for Software Productivity Metrics*. Nueva York: IEEE.

**Jones C.T.** (1986). *Programming Productivity*. Nueva York: McGraw-Hill.

**Jones, C.T.** (1996). "Software Estimating Rules of Thumb". *IEEE computer* (marzo, pág. 116-118).

**Jones, C.T.** (1996). "Activity-based Software Costing". *IEEE computer* (mayo, pág. 103-104).

**Londeix, B.** (1987). *Cost Estimation for Software Development*. Reading (Massachusetts): Addison-Wesley.

**Marco, T. de.** (1982). *Controlling Software Projects*. Englewood Cliffs: Yourdon Press (Prentice Hall).

**Möller, K.H.; Paulish, D.J.** (1993). *Software Metrics: A Practitioner's Guide to Improved Product Development*. Londres: Chapman & Hall, IEEE Press.

**Pressman, R.S.** (1998). *Ingeniería del software: un enfoque práctico* (4.<sup>a</sup> ed.). Madrid: McGraw- Hill.

**Putman, L.H.** (1978). "A General Empirical Solution to the Macro Software Sizing and Estimation Problem". *IEEE Transactions on Software Engineering* (vol. SE-4, núm. 4, julio, pág. 345-361).

**Putman, L.H.** (1981). "SLIM, A Quantitative Tool for Software Cost and Schedule Estimation (A Demonstration of a Software Management Tool)". *Proceedings of the NBS/IEEE/ACM software tool fair* (marzo, pág. 49-57).

**Putman L.; Myers, W.** (1992). *Measures for Excellence*. Englewood Cliffs: Yourdon Press.

**Roetzheim, W.H.; Beasley, R.A.** (1998). *Software Project Cost & Schedule Stimating: Best Practices*. Upper Saddle River: Prentice Hall.

**Royce, W.** (1998). *Software Project Management: A Unified Framework*. Reading: Addison-Wesley (Object Technology Series).

**Sommerville, I.** (1996). *Software Engineering* (5.<sup>a</sup> ed.). Reading: Addison-Wesley.

**Walston C.E; Felix C.p.** (1977). "A Method Of Programming Measurement And Estimation". *Ibm Systems journal* (núm. 1, pág. 54-73).

