F. J. García-Peñalvo, "Definición del proyecto docente," en Proyecto Docente. Profesor Titular de Escuela Universitaria. Perfil Ingeniería del Software y Orientación a Objetos. Área de Conocimiento de Ciencia de la Computación e Inteligencia Artificial pp. 107-210, Salamanca, España: Departamento de Informática y Automática. Universidad de Salamanca, 2000.

Capítulo 4
Definición del Proyecto Docente

Es un hecho ampliamente asumido que la Informática es hoy en día un factor social de gran relevancia. El objetivo de la titulación donde se circunscribe la plaza objeto de concurso, la *Ingeniería Técnica en Informática de Sistemas*, es la formación de profesionales (*ingenieros técnicos en Informática*) que se puedan incorporar a un mercado laboral, actualmente en plena demanda de estos titulados, con unas garantías plenas de calidad en cuanto a la función que deberán desempeñar en sus puestos de trabajo.

Actualmente, y a consecuencia de la falta de madurez que todavía sufre la Informática, es difícil precisar qué se entiende por un ingeniero informático, ya sea técnico o superior. Esta bruma que rodea a la figura de los informáticos es aprovechada por multitud de personas, ya sean tituladas o no, para acogerse a puestos laborales donde se interacciona de una manera u otra con un computador. Es por este motivo por el que se debe hacer un esfuerzo por diferenciar a los *ingenieros en Informática*[12] de los informáticos en general.

En este sentido, y con un ámbito internacional, en 1998 se formó el ***Software Engineering Cordinating Committee*** (SWECC) bajo el auspició de **IEEE-CS** y **ACM**. Su misión es cuidar la evolución de la Ingeniería del Software como una disciplina profesional dentro del mundo de la Informática. Para lo cual se pretende documentar el cuerpo de conocimientos de la disciplina, recomendar un criterio de acreditación de los titulados, desarrollar un modelo de currículo, mantener un código ético y definir un conjunto de estándares. Actualmente, los proyectos que se encuentran en marcha son:

[12] Como ya se ha comentado antes en este proyecto docente, cada vez existe una mayor tendencia en el ámbito internacional a denominar Ingeniería del Software a la titulación que en España equivaldría a la Ingeniería Informática, sin entrar en las diferenciaciones entre titulados de primer o segundo ciclo en nuestro país.

- ***Software Engineering Body of Knowledge (SWEBOK) – Cuerpo de conocimiento de la Ingeniería del Software:*** Tiene como objetivos *clarificar y definir los límites de la ingeniería del software con respecto a otras disciplinas* y *ofrecer los fundamentos para el desarrollo de una propuesta curricular y el material para la certificación de los profesionales.* Este proyecto ha obtenido como resultado la guía con el cuerpo de conocimiento de Ingeniería del Software, que tras pasar por la denominada **Straw Man Version** (*versión de hombre de paja*) [Bourque et al., 1998], actualmente se encuentra en la **Stone Man Version v0.6** (*versión del hombre de piedra*) [Abran et al., 2000] y se pretende que en el año 2001 se llegue a la **Iron Man Versión** (*versión del hombre de hierro*).
- ***Software Engineering Education Project – Proyecto de Educación en Ingeniería del Software [ACM/IEEE-CS, 1999a]:*** Reúne dos subproyectos, *un modelo de acreditación para programas universitarios* [ACM/IEEE-CS, 1998] y *el plan para el proyecto de educación en Ingeniería del Software (Plan for the Software Engineering Education Project – SWEEP)* [ACM/IEEE-CS, 1999b]. Hasta mediados del año 2000 no se tendrá una propuesta curricular sobre en el campo de la Ingeniería del Software.
- ***Software Engineering Code of Ethics and Professional Practice – Código de Ética y Práctica Procesional de Ingeniería de Software [ACM/IEEE-CS, 1999c]:*** El código ético de la profesión fue aprobado por las asociaciones **ACM** e **IEEE-CS** en su versión 5.2 [ACM/IEEE-CS, 1999c][13] como el estándar para la enseñanza y la práctica de la Ingeniería del Software. Este código ha sido desarrollado por un grupo dirigido por **Donald Gotterbarn,** siendo propuesto tras varias versiones[14] y después de revisar los códigos éticos de otras sociedades. El código ético contiene ocho principios relacionados con el comportamiento y las decisiones tomadas por los profesionales. Un breve resumen del mismo se presenta en el Cuadro 4.1.

Tras la presentación de los esfuerzos en pro de la definición de la profesión de *ingeniero informático* o *ingeniero del software* realizados por ACM e IEEE-CS, se vuelve a centrar la atención en el contexto nacional, y más concretamente en el de la Universidad de Salamanca, para definir de una forma más precisa el perfil de los titulados en *Ingeniería Técnica en Informática de Sistemas*; de forma que éstos sean profesionales que *utilicen un sólido conjunto de fundamentos de Ciencias de la Informática para solventar problemas reales, interaccionando con clientes y usuarios,*

[13] En [Dolado, 1999] el **Dr. D. Javier Dolado**, de la Universidad de San Sebastián, ha traducido el código ético en su versión 5.2 al español.

[14] La versión del código ético más conocida, con anterioridad a la aprobación de la versión 5.2, fue la versión 3.0 [Gotterbarn et al., 1997a], [Gotterbarn et al., 1997b]. Precisamente en [Gotterbarn et al., 1999a] y en [Gotterbarn et al., 1999b] en se comenta la versión 5.2 comparándola con la versión 3.0.

debiendo hacer uso una capacidad de comunicación oral y escrita correcta y fluida, y actuando siempre de acuerdo al código ético marcado por su profesión.

Los códigos éticos tienen una función esencial para caracterizar una profesión, y para que una disciplina adquiera el carácter de profesión debe poseer un código de conducta.

Se pueden resumir las principales funciones de los códigos éticos en los siguientes apartados [Bowyer, 1996]:

1) Simbolizar una profesión.
2) Proteger los intereses del grupo.
3) Inspirar buena conducta.
4) Educar a los miembros de la profesión.
5) Disciplinar a sus afiliados.
6) Fomentar las relaciones externas.
7) Enumerar los principios morales básicos.
8) Expresar los ideales a los que se debe aspirar.
9) Mostrar las reglas básicas de comportamiento
10) Ofrecer guías de comportamiento.
11) Enumerar derechos y responsabilidades.

Los códigos de conducta van más allá de la pura normativa legal, ya que ayudan a guiar el comportamiento en multitud de situaciones para las que no existe referencia legal.

El código propuesto por ACM y IEEE-CS tiene como objetivo documentar las responsabilidades y obligaciones éticas y profesionales, en un intento de educar y aleccionar a los ingenieros del software, a la vez que informar al público sobre las responsabilidades que son importantes para la profesión. Este código ético prima el bienestar y la calidad de vida del público en general, en cuanto a todas las decisiones relacionadas con la Ingeniería del Software [Gotterbarn et al., 1999b].

Los ingenieros informáticos deben responsabilizarse que al llevar a cabo sus actividades lo hagan con beneficio y respeto a la profesión que ejercen. De acuerdo a sus compromisos con la salud, la seguridad y el bien público, los ingenieros informáticos deben seguir los siguientes ocho principios [Gotterbarn, 1999]:

1. **Sociedad:** Los ingenieros del software actuarán de manera coherente con el interés general.
2. **Cliente y empresario:** Los ingenieros del software deberán actuar de tal modo que se sirvan los mejores intereses para sus clientes y empresarios, y consecuentemente con el interés general.
3. **Producto:** Los ingenieros del software deberán garantizar que sus productos y las modificaciones relacionadas con ellos cumplen los estándares profesionales de mayor nivel que sea posible.
4. **Juicio:** Los ingenieros del software deberán mantener integridad e independencia en su valoración profesional.
5. **Gestión:** Los gestores y líderes en Ingeniería del Software suscribirán y promoverán un enfoque ético a la gestión del desarrollo y el mantenimiento del software.
6. **Profesión:** Los ingenieros del software deberán progresar en la integridad y la reputación de la profesión, de forma coherente con el interés público.
7. **Compañeros:** Los ingenieros del software serán justos y apoyarán a sus compañeros.
8. **Persona:** Los ingenieros del software deberán participar en el aprendizaje continuo de la práctica de su profesión y promoverán un enfoque ético en ella.

Cuadro 4.1. Resumen del código ético de ACM/IEEE-CS, versión 5.2

En este capítulo se va a presentar qué[15] puede aportar a los futuros titulados las actividades a realizar desde la plaza a concurso, y que vienen marcadas por su perfil: *docencia en materias de Ingeniería del Software y Orientación a Objetos*. Para lo que primero se van a identificar las asignaturas relacionadas con el perfil de la plaza dentro del plan de estudios vigente, para presentar sus características, interrelaciones con otras

[15] El cómo se detalla en el capítulo 5 dedicado a la programación docente de las asignaturas.

asignaturas del plan de estudios y su situación tanto en el ámbito nacional como internacional.

4.1 El perfil de formación

Las actividades a realizar por la persona que ocupe la plaza a concurso se centran en impartir docencia en materias relacionadas con las disciplinas de Ingeniería del Software y de Tecnología de Objetos[16] en la *Ingeniería Técnica en Informática de Sistemas* dentro de la Universidad de Salamanca.

El siguiente paso es identificar qué asignaturas se ajustan al perfil de formación dentro del plan de estudios vigente en dicha titulación (*Plan de 1997, aprobado en el BOE de 4 de Noviembre de 1997*). En el *Capítulo 2* de este Proyecto Docente ya se presentó este plan de estudios, y más concretamente en la Tabla 2.14 se incluía una relación de las asignaturas troncales y obligatorias del mismo, mientras que en la Tabla 2.15 se hacía lo propio con las asignaturas optativas. Una descripción más detallada del plan de estudios vigente para la titulación de *Ingeniería Técnica en Informática de Sistemas* se encuentra en la *Guía Académica de la Facultad de Ciencias* para el Curso 1999-2000 [USAL, 1999].

De las asignaturas que se incluyen en este plan de estudios hay dos que se ajustan perfectamente al perfil propuesto para la plaza a concurso: ***Ingeniería del Software*** y ***Programación Orientada a Objetos***. Ambas asignaturas se imparten en el tercer curso de la *Ingeniería Técnica*, en el quinto y sexto cuatrimestre respectivamente.

La asignatura de *Ingeniería del Software* es una asignatura obligatoria de 6 créditos, 4,5 de ellos teóricos y el 1,5 restantes prácticos. Por su parte la asignatura de *Programación Orientada a Objetos* es una asignatura optativa de 6 créditos repartidos al cincuenta por ciento entre teoría y práctica.

Dado que el plan de estudios vigente comenzó su andadura en el curso 1997-1998, en el presente curso académico (1999-2000) es la primera vez que se cursan estas asignaturas, si bien ambas tenían asignaturas equivalentes, con las mismas características de nombre, obligatoriedad y carga docente en el plan de estudios anterior (Plan de 1992), como se puede comprobar en [USAL, 1998], apareciendo éstas como asignaturas sin docencia en el presente curso académico para aquellos alumnos que, matriculados en el Plan de 1992, no hubieran superado alguna de ellas, y quedando establecido el mecanismo de convalidación oportuno entre estas asignaturas del Plan de 1992 y sus homónimas del Plan de 1997.

[16] Nótese que a la Tecnología de Objetos se la puede considerar como un paradigma concreto de desarrollo, y por tanto como una parte de la Ingeniería del Software.

4.2 Ingeniería del Software

4.2.1 Introducción

Previamente al desarrollo de la propuesta concreta del programa de la asignatura *Ingeniería del Software*, que se desarrolla en el *Capítulo 5*, se presenta una visión global de la Ingeniería del Software como disciplina mediante un análisis de su marco histórico y conceptual. El objetivo no es realizar una exposición exhaustiva, sino mostrar aquellos conceptos, aportaciones y resultados que se consideran relevantes para el contenido de la asignatura objeto de la propuesta docente. De esta forma los contenidos se adaptarán a las tendencias actuales consolidadas en la teoría, tecnología, práctica y aplicación del software a los sistemas basados en computadores.

El desarrollo de un proyecto docente para una determinada disciplina implica, en primer lugar, la identificación clara y precisa del conjunto de conceptos y conocimientos científico/técnicos que la misma engloba. En el caso de la Ingeniería del Software es una tarea que presenta algunas dificultades debido al amplio conjunto de materias que abarca y a la constante evolución a la que se ve sometida como consecuencia de los rápidos cambios que sufre la tecnología del software. Una primera aproximación a esta tarea puede venir dada por la definición de Ingeniería del Software.

Antes de hacer un repaso por algunas de las muchas definiciones que de Ingeniería del Software se han dado, se va comentar el origen y polémica que el propio término ha suscitado.

La introducción del término Ingeniería del Software se produjo en la primera conferencia sobre Ingeniería del Software patrocinada por la OTAN, celebrada en Garmisch (Alemania) en octubre de 1968 [Naur and Randell, 1969], atribuyéndose la paternidad del término a **Fritz Bauer** [Randell, 1998].

Fue tal la aceptación de esta conferencia que se consiguió un nuevo patrocinio de la OTAN para una segunda conferencia sobre Ingeniería del Software, que tendría lugar un año más tarde en Roma (Italia) [Buxton and Randell, 1970], con unos resultados menos esperanzadores que los producidos en la primera conferencia. De hecho, no se produjo ninguna petición de que continuara la serie de conferencias de la OTAN, lo cual no influyó para que a partir de entonces se utilizara con gran profusión el nuevo término para describir los trabajos realizados, aunque quizás sin un consenso real sobre su significado.

En el contexto educativo, sin duda alguna, lo que más controversia ha levantado es el propio nombre del término, centrándose la discusión en la pregunta *¿es la Ingeniería del Software realmente una Ingeniería?* [Tomayko, 2000].

Los argumentos que se dan para sustentar una respuesta negativa se pueden resumir en dos categorías. La primera reuniría a aquéllos que se ciñen a la definición literal de Ingeniería dada por algunos diccionarios o sociedades profesionales, que se centran en que en estas definiciones se hace mención a productos tangibles derivados del uso efectivo de materiales y fuerzas naturales, mientras que el software ni es tangible, ni utiliza materiales y fuerzas naturales para su concepción.

La segunda categoría estaría formada por los que arguyen que una disciplina ingenieril evoluciona desde una profesión, y la profesión relacionada con el software no ha evolucionado lo suficiente para ser considerada una Ingeniería.

Por el contrario, son muchos los que están a favor de la utilización y difusión del término *Ingeniería del Software*, tomando como un estándar de facto la utilización reiterada del término en la bibliografía especializada.

Quizás la defensa más fuerte y adecuada de la Ingeniería del Software como Ingeniería venga de la mano de **Mary Shaw** que justifica que si tradicionalmente se ha definido Ingeniería como "*la creación de soluciones rentables a problemas prácticos mediante la aplicación del conocimiento científico para la construcción de cosas al servicio de la humanidad*" [Shaw, 1990], entonces el desarrollo del software es un problema ingenieril apropiado, porque involucra "*la creación de soluciones rentables económicamente para problemas prácticos*" [Shaw and Tomayko, 1991].

Una vez hechas estas disquisiciones sobre el término y sus controversias, se va a proceder a exponer una muestra de las numerosas definiciones que de Ingeniería del Software se pueden encontrar en la bibliografía:

> **"Ingeniería del software es el establecimiento y uso de principios sólidos de ingeniería, orientados a obtener software económico que sea fiable y trabaje de manera eficiente en máquinas reales"**
>
> **Fritz Bauer,** *First NATO Software Engineering Conference*, Garmisch (Germany), 1968; en **[Buxton et al., 1976]**[17]

> **"La aproximación sistemática al desarrollo, operación, mantenimiento y retirada del software"**
>
> **IEEE "Standard Glossary of Software Engineering Terminology" [IEEE, 1983]**

> **"Es la disciplina tecnológica y de gestión que concierne a la producción y mantenimiento sistemático de productos software que son desarrollados y modificados a tiempo y dentro de los costes estimados"**
>
> **[Fairley, 1985]**

[17] También en [Bauer, 1972].

"Tratamiento sistemático de todas las fases del ciclo de vida del software. Se refiere a la aplicación de metodologías para el desarrollo del sistema software"

Asociación Española para la Calidad.
"*Glosario de Términos de Calidad e Ingeniería del Software*" **[AECC, 1986]**

"Construcción de software multi-versión por un equipo de varias personas"

[Parnas and Weiss, 1987]

"La aplicación disciplinada de principios, métodos y herramientas de ingeniería, ciencia y matemáticas para la producción económica de software de calidad"

[Humphrey, 1989]

"La utilización de metodologías, herramientas y técnicas para resolver los problemas prácticos que surgen en la construcción, desarrollo, soporte y evolución del software"

Institute for Information Technology, NRC Canada, 1990

"Una disciplina cuyo objetivo es la producción de software de calidad, que se entrega en plazo, se ajusta al presupuesto y que satisface sus requisitos"

Vanderbilt University, "*Software Engineering*"
Aksen Assoc., 1990

"La aplicación de un enfoque sistemático, disciplinado y cuantificable para el desarrollo, la operación y el mantenimiento del software; es decir, la aplicación de la ingeniería al software"

"*Standard Glossary of Software Engineering Terminology*". IEEE Std 610.12-1990
[IEEE, 1999]

"Disciplina tecnológica y de gestión concerniente a la invención, producción sistemática y mantenimiento de productos software de alta calidad, desarrollados a tiempo y al mínimo coste"

[Frakes et al., 1991]

"Las actividades sistemáticas implicadas en el diseño, implantación y prueba de software para optimizar su producción y soporte"

Canadian Standards Association.
"*CSA Information Technology Vocabulary*"**, 1992**

"Aplicación de herramientas, métodos y disciplinas para producir y mantener una solución automatizada de un problema real"

[Blum, 1992]

> **"Aquella forma de ingeniería que aplica principios de informática y matemática a la resolución de problemas software de forma eficiente en cuanto al coste"**
>
> **[Humphrey, 1993]**

> **"Aplicación de principios científicos para la transformación ordenada de un problema en una solución software funcional, así como en el consiguiente mantenimiento del software hasta el final de su vida útil"**
>
> **[Davis, 1993]**

> **"Es la aplicación de herramientas, métodos y disciplinas de forma eficiente en cuanto al coste, para producir y mantener una solución a un problema de procesamiento real automatizado parcial o totalmente por el software"**
>
> **[Horan, 1995]**

> **"La aplicación de métodos y conocimiento científico para crear soluciones prácticas y rentables para el diseño, construcción, operación y mantenimiento del software y los productos asociados, al servicio de las personas"**
>
> Adaptado de la definición de Ingeniería de
> **Mary Shaw y David Garlan** en **[Shaw and Garlan, 1996]**

En lugar de con definiciones escuetas, el *Instituto de Ingeniería del Software* y la *Sociedad Británica de Informática* presentan su visión de lo qué es la Ingeniería del Software con descripciones más elaboradas.

> 1. Definición central:
>
> - Ingeniería es la aplicación sistemática de conocimiento científico para la creación y construcción de soluciones rentables a problemas prácticos al servicio de la humanidad.
> - La Ingeniería del Software es la forma de ingeniería que aplica principios propios de la Ciencia de la Informática y Matemáticas para conseguir soluciones rentables a problemas software.
>
> 2. Elaboraciones e interpretaciones:
>
> - La creación y construcción del software debe incluir el mantenimiento. Debe cubrirse el ciclo de vida del software completo.
> - La rentabilidad implica no sólo dinero, sino tiempo, calendario y recursos humanos. También implica obtener buenos valores por los recursos invertidos; incluyendo la calidad cuando las medidas se consideren oportunas.
> - La Ingeniería del Software no se limita a aplicar sólo principios de la Ciencia de la Informática y las Matemáticas, sino cualquier principio del que pueda sacar ventaja.
> - La Ingeniería del Software necesita contar con principios y técnicas de gestión para llevar a cabo sus actividades de desarrollo.
>
> 3. Distinción entre el uso actual del término "Ingeniería del Software" y la definición que se adecua a la misión del SEI:
>
> - Actualmente, el término "Ingeniería del Software" tiene múltiples conjuntos de significados conflictivos y pobremente entendidos, que van desde la programación a la gestión del diseño del sistema.
> - Actualmente, el término "Ingeniería del Software" es más una aspiración que una descripción.
>
> **Software Engineering Institute (SEI) [Ford, 1990]**

"La Ingeniería del Software requiere la comprensión y aplicación de principios de ingeniería, habilidades de diseño, buenas prácticas de gestión, fundamentos de la Ciencia de la Informática y formalismos matemáticos. Es tarea de la Ingeniería del Software juntar estas áreas de trabajo tan dispares y utilizarlas en las fases de obtención de los requisitos, especificación, diseño, verificación, implementación, prueba, documentación y mantenimiento de sistemas software complejos y de gran tamaño. El ingeniero del software debe cumplir el papel del arquitecto del sistema complejo, tomando en cuenta las necesidades y requisitos del usuario, la viabilidad, el coste, la calidad, la confianza, la seguridad y las restricciones temporales. La necesidad de ajustar la importancia relativa de estos factores de acuerdo a la naturaleza del sistema y de su aplicación confiere una fuerte dimensión ética a las tareas del ingeniero del software, sobre quien puede depender la seguridad y bienestar de otros, y para quien, como en medicina o en derecho, se requiere un sentido de moralidad profesional para su trabajo.

El ingeniero del software debe ser capaz de estimar el coste y la duración del proceso de desarrollo del software, así como determinar la consecución de corrección y confianza. Tales medidas y estimaciones pueden involucrar conocimientos de conceptos financieros y de gestión, al mismo nivel que el manejo de los fundamentos matemáticos. Se necesita el uso preciso de las notaciones formales y de las palabras para expresarlas con el grado de precisión requerido a otros ingenieros y a clientes formados. En la mayoría de las circunstancias las hebras técnicas, teóricas y de gestión de un proyecto de Ingeniería del Software no pueden separarse unas de las otras.

Para construir grandes productos y conseguir una alta productividad, el ingeniero requiere el uso de herramientas software de desarrollo y de elementos reutilizables que garanticen su subsiguiente modificación y mantenimiento con seguridad.

La actividad profesional del ingeniero del software abarca el rango de tareas involucradas en el ciclo de vida de un sistema software. La obtención de requisitos, especificación, diseño, verificación y construcción son tareas críticas para conseguir la calidad del producto y son todas ellas responsabilidad del ingeniero del software.

Dado que el software determina el comportamiento de un autómata, el ingeniero del software necesita tener conocimientos de hardware digital y de comunicaciones. Aunque la Ingeniería del Software como disciplina puede ser calificada como independiente del área de aplicación, su realización debe ser en el contexto de aplicaciones específicas. El ingeniero del software debe, por tanto, ser capaz de colaborar con otros profesionales que le brindarán capacidades complementarias en la labor de especificar, diseñar y construir sistemas hardware-software que se ajusten a las necesidades del cliente, haga uso de las soluciones hardware y software en una óptima combinación y ofrezca una interfaz de usuario con una calidad adecuada.

La mayoría del software se construye en equipo, frecuentemente con equipos interdisciplinarios. La habilidad para trabajar cerca los unos de los otros es esencial.

Algunos de los métodos y de las herramientas intelectuales de la Ingeniería del Software están en proceso de desarrollo y se espera que tengan que cambiar de forma rápida. Los ingenieros del software, por tanto, necesitan tener unos buenos fundamentos teóricos que les sirvan de base par aprender y usar nuevos métodos en el futuro, y la mentalidad que les permita actualizar de forma permanente los conocimientos que necesitan para su labor profesional"

British Computer Society and the Institution of Electrical Engineers [BCS, 1989]

No sólo hace falta decir lo qué es la Ingeniería del Software, sino que es conveniente recalcar lo qué **no** es; así la Ingeniería del Software no es el diseño de programas que se implementan en otras áreas ingenieriles, ni es simplemente una forma de programar más organizada que la que prevalece entre aficionados, principiantes o personas con falta de educación y entrenamiento específico.

Esta variedad de definiciones refleja las diferentes concepciones existentes sobre la Ingeniería del Software. A modo de ejemplo, esta diferencia de alcance y concepción de la Ingeniería del Software como disciplina se aprecia revisando los contenidos del libro de **Ian Sommerville** [Sommerville, 1985], donde la Ingeniería del Software está limitada al desarrollo de la programación mientras que el libro de **Roger S. Pressman** [Pressman, 1987] de esa misma época ya considera el análisis y diseño de sistemas. En su última edición hasta la fecha Sommerville [Sommerville, 1996] amplía su concepto de Ingeniería del Software a aquellas actividades relacionadas con la especificación, desarrollo, gestión y evolución del software, no incorporando ningún capítulo específico sobre la práctica o los lenguajes de programación.

El concepto de Ingeniería del Software surge de la distinción entre programación de pequeños proyectos (*programming in the small*) y programación de grandes proyectos (*programming in the large*) y el reconocimiento de que la Ingeniería del Software está

relacionada con esta última. Este primer concepto fue rápidamente ampliado para incorporar a la Ingeniería del Software todas aquellas tareas relacionadas con la automatización de los Sistemas de Información y con la Ingeniería de Sistemas en general.

El enfoque de la Ingeniería del Software que se adopta en este proyecto es el relacionado con los problemas que se presentan en el desarrollo de grandes sistemas software bajo la perspectiva de los sistemas de información a los que dan servicio. Esto viene motivado por el hecho de que aquellos enfoques de la Ingeniería del Software más relacionados con el diseño y la programación de módulos o componentes software están asignados a otras asignaturas del plan de estudios como *Programación, Laboratorio de Programación, Estructuras de Datos* o *Programación Orientada a Objetos*.

4.2.2 Marco histórico de la Ingeniería del Software

El contexto en el cual se han desarrollado las técnicas del software está íntimamente ligado a la evolución solapada de los sistemas informáticos y de la programación, cuyos hitos o avances más importantes han configurado las eras o etapas que se detallan a continuación. No se intenta hacer una clasificación cronológica exacta ya que los desarrollos han estado muy solapados e incluso ideas que surgieron en una fecha determinada no serian aceptadas ampliamente quizá hasta 10 años después [Goldberg, 1986], [Pressman, 1992].

Ninguna de estas eras puede decirse que hayan terminado formalmente, sobre todo si se hace caso del estudio [Redwine, 1985] que afirma que el tiempo necesario para aceptar una idea que implique un cambio tecnológico es de 15 a 20 años. Esta afirmación ha sido corroborada más recientemente por otros muchos autores, como **Klaus Dittrich** del grupo del conocido "***manifiesto de Atkinson***" [Atkinson et al., 1989].

Durante la **primera era** (1.950-1.96x), la programación de ordenadores se concibe más como un arte que como una técnica sistematizada y compleja. En esta época la importancia se centra en el hardware, sometido a un cambio continuo, considerando al software como un *añadido*. El software se desarrolla utilizando únicamente la intuición, con escasos métodos sistematizados y prácticamente sin ningún control en su desarrollo. La mayoría de los sistemas utilizaban una orientación *batch* (*excepciones a esto son el sistema de reserva de American Airlines y los sistemas americanos de defensa en tiempo real, como SAGE*). El ordenador estaba dedicado a la ejecución de un programa simple que a su vez resolvía una situación específica.

En la **segunda era** (1.96x-197x) la multiprogramación y los sistemas multiusuario introdujeron nuevos conceptos en la interacción hombre-máquina. Los sistemas en tiempo real podían recoger, analizar y transformar datos procedentes de múltiples orígenes, controlando procesos y produciendo resultados en milisegundos en vez de en

minutos. Los avances en las máquinas de memoria secundaria *on-line* llevaron a la primera generación de sistemas de gestión de bases de datos.

La segunda era también se caracterizó por el uso de *productos software* o paquetes de software y por el comienzo de las "*software house*" o "*casas de servicios*". El software empezó a ser desarrollado pensando en una distribución más amplia y para un mercado multidisciplinar.

Una de las primeras contribuciones a la ingeniería del software, en el sentido de mejorar la fiabilidad, dirección y productividad en el desarrollo del software, fue el artículo de **E. W. Dijkstra** "*Go to Statement Considered Harmful*", que apareció en la revista **Communications of the ACM** (Vol. 11, N.3) en 1968, y acentuaba los beneficios de utilizar los conceptos de la programación estructurada. Otra contribución importante de esta época fue la obra de **Weinberg** "*The Psychology of Computer Programming*" [Weinberg, 1971]; este clásico de la literatura del software introdujo las ideas de "*programación sin ego*", nombres mnemónicos de variables y en general la necesidad de la claridad y un buen estilo en la programación.

Según fue creciendo el número de sistemas basados en ordenador las bibliotecas de software se fueron extendiendo. Los proyectos produjeron cientos de miles de instrucciones fuente. Pero los problemas también empezaron a aparecer: todos estos sistemas informáticos y todas estas instrucciones fuente tenían que ser mantenidos para corregir errores "*oscuros*" (detectados tardíamente), adaptarse a los cambios en los requisitos de los usuarios o adaptarse al nuevo hardware que adquirían las organizaciones. El esfuerzo necesario para el mantenimiento de los sistemas informáticos, y sobre todo del software asociado, comenzó a absorber recursos de forma alarmante y, peor aún, la naturaleza personalizada y la ausencia o escasez de técnicas generales de diseño y análisis, hacía que muchos de estos sistemas fuesen prácticamente inmantenibles: había empezado una **crisis del software**, reconocida por primera vez oficialmente en la reunión de la **OTAN** de 1968 [Naur and Randell, 1969], [Boehm, 1976], [Goldberg, 1986], [Randell, 1998].

En la **tercera era** (1.97x-1.98x) aparecen los sistemas distribuidos - varios computadores, realizando cada uno sus funciones concurrentemente y comunicándose unos con otros - dando lugar a un incremento importante de la complejidad de los sistemas informáticos. Esta época se caracteriza por el uso generalizado del microprocesador que, gracias al abaratamiento y aumento de potencia de éstos, permite la realización de funciones complejas a un coste excepcionalmente bajo.

Durante esta era, la crisis del software se acentuó, detectándose que aproximadamente el **50%** de los presupuestos de los centros de procesamiento de datos se dedica a mantenimiento [Goldberg, 1986], con lo que la productividad en el desarrollo de nuevos sistemas se vio notablemente dañada. La intensificación de la crisis provoca como respuesta una mayor toma en consideración de la necesidad de un proceso de Ingeniería para el desarrollo del software. Como consecuencia de la

importancia que adquiere la Ingeniería del Software hacen su aparición las primeras metodologías, destacando las propuestas por Jackson [Jackson, 1975], Warnier [Warnier, 1974] y DeMarco [DeMarco, 1979] por ser las que mayor difusión y utilización alcanzan. Por otra parte, este acercamiento del proceso de construcción del software a los procesos de ingeniería clásicos, conduce a la aplicación de técnicas de gestión de proyectos como PERT y CPM a los proyectos de desarrollo de software.

La consecuencia que se extrae de esta época es que es mejor utilizar alguna metodología disciplinada, no importa cual, que no utilizar ninguna [Basili, 1991].

Una **cuarta era** (1.98x-...) ha venido caracterizada por la introducción de sistemas de sobremesa, y la adopción de tecnologías y herramientas que proporcionan el soporte necesario para mejorar la calidad y la productividad en el desarrollo del software. Entre ellas se pueden destacar: *las herramientas CASE, los entornos de programación, el prototipado rápido (usado independientemente o con el ciclo de vida tradicional)*, *la tecnología orientada a objetos, la reutilización sistemática del software, y los lenguajes de cuarta generación (4GL)*.

Es también en este período cuando empiezan a darse a conocer las aproximaciones formales al desarrollo de software a través de especificaciones algebraicas y lenguajes de especificación ejecutables [Goguen and Meseguer, 1988], como OBJ [Goguen et al., 1992] o ACT ONE [Ehrig and Mahr, 1985], técnicas y métodos orientados a modelos como Z [Spivey, 1989], [Diller, 1990] o VDM [Jones, 1990] y álgebras de procesos como CSP [Hoare, 1985], aunque algunas de estas técnicas son muy anteriores (*como VDM, que fue desarrollado por los laboratorios de investigación de IBM de Viena, en 1973*).

En los últimos años, han cobrado una gran importancia los modelos de proceso y la preocupación por la mejora en la calidad tanto del producto software como del proceso para mejorarlo. De estos modelos se puede destacar la norma ISO-9000, en su aplicación a la construcción de software [Layman, 1994], [Schmauch, 1994], la iniciativa "BOOTSTRAP" [Lebsanft and Synspace, 1994], que consiste en un método para analizar y evaluar cuantitativamente determinados atributos de calidad del proceso (la evaluación permite conseguir un perfil detallado acerca de la calidad de la organización donde se aplica); el conocido modelo de madurez y capacidad de la Universidad Carnegie-Mellon (Pittsburg, PA - USA), CMM (*Capability Maturity Model*) [Paulk et al., 1993a], [Paulk et al., 1993b] o el método SPICE (*Software Process Improvement & Capability Evaluation*) [Dorling, 1993], [Konrad et al., 1995] que tiene como objetivo convertirse en un estándar ISO mundial que integre a las iniciativas anteriores.

Los sistemas basados en microprocesadores de 32 ó 64 bits, la computación paralela, el desarrollo de sistemas de Inteligencia Artificial (*todavía hoy no excesivamente extendidos en el mercado*) y las nuevas tecnologías (*láser, fibra óptica...*) en las comunicaciones (*Internet*), junto con herramientas de programación visual,

desarrollo y ejecución de aplicaciones remotas, están condiciendo a **la transición hacia una quinta era**. En ingeniería del software se espera que esta nueva era venga marcada por el desarrollo y perfeccionamiento de los entornos de programación y herramientas integradas de apoyo a metodologías, por la continuación y mejora de las técnicas asociadas al prototipado y reusabilidad del software y por la aplicación de técnicas de "Ingeniería del Conocimiento" al desarrollo de software, especialmente como apoyo a la Ingeniería de Requisitos en la captura, estructuración y reutilización del conocimiento (*requisitos*) en dominios de aplicación [Sutcliffe and Maiden, 1998].

4.2.3 Marco conceptual de la Ingeniería del Software

En la introducción se señaló que el enfoque de la Ingeniería del Software que aquí se presenta es el relacionado con los Sistemas de Información a los que da servicio. Bajo esta perspectiva, un sistema software es una parte de un sistema mayor que lo engloba como componente. La Ingeniería del Software, por lo tanto, será solamente una parte del diseño del sistema en la que los requisitos del software han de ajustarse a los requisitos del resto de los elementos que constituyen este sistema. Por este motivo, el ingeniero del software ha de estar implicado en el desarrollo de los requisitos del sistema completo, comprendiendo el dominio de actividad en su totalidad.

Atendiendo a esta concepción de la Ingeniería del Software, es importante remarcar el papel que desempeña la Teoría General de Sistemas como antecedente conceptual en el que se apoya la teoría sobre los Sistemas de Información a los que la Ingeniería del Software intenta aportar soluciones. En el marco de la Teoría General de Sistemas, el análisis de sistemas tiene como objetivo general la comprensión de los sistemas complejos para abordar su modificación de forma que se mejore el funcionamiento interno para hacerlo más eficiente, para modificar sus metas... Las modificaciones pueden consistir en el desarrollo de un subsistema nuevo, en la agregación de nuevos componentes, en la incorporación de nuevas transformaciones... En general, el análisis de sistemas establece los siguientes pasos a seguir:

1. ***Definición del problema***. *En este paso se identifican los elementos de insatisfacción, los posibles cambios en las entradas y/o salidas al sistema y los objetivos del análisis del sistema.*
2. ***Comprensión y definición del sistema***. *En este paso se identifica y descompone el sistema jerárquicamente en sus partes constituyentes o subsistemas junto con las relaciones existentes entre los mismos.*
3. ***Elaboración de alternativas***. *En este paso se estudian las diferentes alternativas existentes para la modificación y mejora del sistema, atendiendo a los costes y perspectivas de realización.*
4. ***Elección de una de las alternativas definidas en el paso anterior.***
5. ***Puesta en práctica de la solución elegida.***
6. ***Evaluación del impacto de los cambios introducidos en el sistema.***

Muchas de las técnicas y métodos actuales de la Ingeniería del Software intentan dar respuesta a este tipo de cuestiones.

Los sistemas, de los que el software forma parte, se denominan sistemas informáticos o sistemas de computadora. En este sentido **Roger S. Pressman** [Pressman, 1997] considera que la Ingeniería del Software ocurre como consecuencia de un proceso denominado Ingeniería de Sistemas de Computadora. La Ingeniería de Sistemas de Computadora se concentra en el análisis, diseño y organización de los elementos en un sistema que pueden ser un producto, un servicio o una tecnología para la transformación de información o el control de información. De igual forma, este autor denomina al *Proceso de Ingeniería del Software* como **Ingeniería de la Información** cuando el contexto de trabajo de Ingeniería se enfoca a una empresa, y lo denomina **Ingeniería de Producto** cuando el objetivo es construir un producto. El término genérico de Ingeniería de Sistemas es el que utiliza para unificar estos dos tipos de ingenierías. La Ingeniería de Sistemas establece, por lo tanto, el papel que ha de asignarse al software y los enlaces que unen al software con otros elementos de un sistema basado en computadora.

Desde una perspectiva más general, **J. L. Le Moigne** [Le Moigne, 1973] concibe los sistemas formados por tres subsistemas interrelacionados: *el de decisión, el de información* y *el físico*. El sistema de decisión procede a la regulación y control del sistema físico decidiendo su comportamiento en función de los objetivos marcados. El sistema físico transforma un flujo físico de entradas en un flujo físico de salidas. En interconexión entre el sistema físico y el sistema de gestión se encuentra el sistema de información. El sistema de información está compuesto por diversos elementos encargados de almacenar y tratar las informaciones relativas al sistema físico a fin de ponerlas a disposición del sistema de gestión. El Sistema Automatizado de Información (SAI) es un subsistema del sistema de información en el que todas las transformaciones significativas de información son efectuadas por máquinas de tratamiento automático de las informaciones.

Basándose en las ideas anteriores, se considera a la Ingeniería del Software como la disciplina que se ocupa de las actividades relacionadas con los sistemas informáticos o sistemas de información en los que el software desempeña un papel relevante. Estos sistemas de información han de ser fiables, es decir, que su realización se lleve a cabo de forma correcta conforme a unos estándares de calidad y, además, que su desarrollo se realice en el tiempo y coste establecidos. Este último aspecto es crucial, y existe una gran variedad de informes, publicaciones y datos que avalan la gran dependencia que las organizaciones tienen hoy en día de los sistemas software.

En muchas ocasiones la Ingeniería del Software se ha querido limitar al desarrollo de grandes proyectos informáticos. Sin embargo, tal y como afirma **Barry B. Boehm** "*se hacen planos para una casa tanto si esta es grande como si es pequeña*". La filosofía que subyace en esta frase, es que cualquier desarrollo de software ha de seguir

un proceso. Como afirma **Roger S. Pressman** [Pressman, 1997]: "*El fundamento de la Ingeniería del Software es la capa proceso. El proceso de la Ingeniería del Software es la unión que mantiene juntas las capas de tecnología y que permite un desarrollo racional y oportuno de la Ingeniería del Software*". El proceso define un marco de trabajo en el que se establece el control de gestión de los proyectos software y el contexto en el que se aplican los métodos técnicos y se producen los resultados del trabajo.

En el proceso de construcción de sistemas informáticos se pueden distinguir dos fases genéricas, independientemente del paradigma de ingeniería elegido: ***la definición*** y ***el desarrollo***.

Durante la fase de *definición* se identifican los requisitos claves del sistema y del software. Durante la misma se desarrollan un **Análisis de Sistemas**, en el que se define el papel de cada elemento en el sistema automatizado de información, incluyendo el que jugará el software, y un **Análisis de Requisitos** en el que se especifican todos los requisitos de usuario que el sistema tiene que satisfacer. Esta fase está orientada al **QUÉ**: *qué información ha de ser procesada, qué función y rendimiento se desea, qué interfaces han de establecerse, qué ligaduras de diseño existen y qué criterios de validación se necesitan para definir un sistema correcto*. En la fase de definición existe un paso complementario que consiste en la planificación del proyecto software, en el que se asignan los recursos, se estiman los costes y se planifican las tareas y el trabajo.

Por el contrario la fase de *desarrollo* está orientada al **CÓMO**, y el primer paso de esta fase corresponde al **Diseño del Software**. En el diseño del software *se trasladan los requisitos del software a un conjunto de representaciones que describen la estructura de datos, arquitectura del software y procedimientos algorítmicos y que permiten la construcción física de dicho software*. Los otros dos pasos de la fase de desarrollo corresponden a la **Codificación** y a la **Prueba del Software**.

Además de las fases de definición y desarrollo del software, existe una tercera fase dentro de la construcción de los sistemas que corresponde a la fase de **Mantenimiento** de los mismos. Esta fase se enfoca *a los cambios asociados con la corrección de errores, con las adaptaciones requeridas por la evolución del entorno del software y las modificaciones debidas a los cambios de requisitos del usuario para mejorar el sistema*.

La fase de definición es crucial en el desarrollo de un sistema software pues en ella quedan establecidas y determinadas explícitamente las necesidades y limitaciones del usuario y del sistema. La especificación de requisitos ha de realizarse antes de que la construcción del sistema de comienzo, pues de lo contrario podría ocurrir que las necesidades se simplifiquen completamente, las limitaciones se olviden o las interdependencias se pasen por alto durante la fase de desarrollo.

Para la comprensión y validación del sistema en estudio, se elaboran modelos. Estos modelos han de separar las especificaciones conceptuales y lógicas (¿qué ha de

cumplir el sistema?), de las especificaciones físicas, (¿cómo lo hará dependiendo de los recursos hardware y software?). Un concepto esencial en el desarrollo de este proceso es el de "***nivel de abstracción***". Aplicando este concepto, el desarrollo de un sistema de información consiste en definir una jerarquía apropiada de niveles de abstracción. Cada nivel produce un modelo del sistema que se describe mediante un lenguaje apropiado. El desarrollo comienza con niveles de abstracción altos, poco detallados, y termina con los de máximo detalle que sirven como base para la construcción directa del sistema ejecutable.

Una última consideración a tener en cuenta en el proceso de construcción de sistemas informáticos, es que la operatividad del producto final depende en gran medida de su conocimiento. Esto se consigue con la elaboración detallada y precisa de la documentación de apoyo: *documentación de sistema, manual de usuario, instrucciones de instalación, guías de entrenamiento, manual de operación...*

El uso de una **metodología** permite el dominio del proceso descrito, definición, desarrollo, implementación y mantenimiento, lo que asegurará el éxito de los proyectos informáticos. En general, una metodología es "*el conjunto de métodos que se siguen en una investigación científica o en una exposición doctrinal*" [DRAE, 1995]. Se puede decir que una metodología es un enfoque, una manera de interpretar la realidad o la disciplina en cuestión, que en este caso particular correspondería a la Ingeniería del Software. A su vez, un método, es un procedimiento que se sigue en las ciencias para hallar la verdad y enseñarla. Es un conjunto de técnicas, herramientas y tareas que, de acuerdo a un enfoque metodológico, se aplican para la resolución de un problema.

Desde el punto de vista específico de la Ingeniería del Software, la metodología describe como se organiza un proyecto, el orden en el que la mayoría de los trabajos tienen que realizarse y los enlaces entre ellos, indicando asimismo cómo tienen que realizarse algunos trabajos proporcionando las herramientas concretas e intelectuales. En concreto, se puede definir **metodología de Ingeniería del Software** como "*un proceso para producir software de forma organizada, empleando una colección de técnicas y convenciones de notación predefinidas*" [Rumbaugh et al, 1991].

En la actualidad se pueden distinguir seis escuelas principales de pensamiento en relación con las técnicas y métodos de desarrollo de Ingeniería del Software:

1. **Orientadas a procesos**: Si se parte de que la Ingeniería del Software se fundamenta en el modelo básico **entrada/proceso/salida** de un sistema[18]; de forma que los datos se introducen en el sistema y éste responde ante ellos transformándolos para obtener salidas. Estas metodologías se enfocan fundamentalmente en la parte de proceso y, por esto, se describen como un enfoque de desarrollo de software orientado al proceso. Utilizan un enfoque de descomposición descendente para evaluar los procesos del espacio del

[18] Este modelo básico es utilizado por todas las metodologías estructuradas.

problema y los flujos de datos con los que están conectados. Este tipo de metodologías se desarrolló a lo largo de los años 70. Los creadores de este tipo de métodos fueron **Edward Yourdon y Larry Constantine** [Yourdon and Constantine, 1975], [Yourdon and Constantine, 1979], [Yourdon and Constantine, 1989], [Yourdon, 1989]; **Tom DeMarco** [DeMarco, 1979]; **Gane y Sarson** [Gane and Sarson, 1977], [Gane and Sarson, 1979]. Representantes de éste grupo son las metodologías de análisis y diseño estructurado como **YSM** (Yourdon Systems Method) [Yourdon Inc., 1993], **SSADM** (Structured Systems Analysis and Design Method) [Ashworth and Goodland, 1990] o **METRICA v.2.1** [MAP, 1995].

2. **Orientadas a datos**: Al contrario que en el caso anterior, estas metodologías se centran más la parte de **entrada/salida** dentro del modelo básico **entrada/proceso/salida.** En estas metodologías las actividades de análisis comienzan evaluando en primer lugar los datos y sus interrelaciones para determinar la arquitectura de datos subyacente. Cuando esta arquitectura está definida, se definen las salidas a producir y los procesos y entradas necesarios para obtenerlas. Ejemplos representativos de este grupo son los métodos **JSP** (Jackson Structured Programming) y **JSD** (Jackson Structured Design) [Jackson, 1975], [Jackson, 1983], [Cameron, 1989], la construcción lógica de programas **LCP** (Logical Construction Program) de [Warnier, 1974] y el **DESD** (Desarrollo de Sistemas Estructurados de Datos), también conocido como metodología **Warnier-Orr**, [Orr, 1977], [Orr, 1981].

3. **Orientadas a estados y transiciones**: Estas metodologías están dirigidas a la especificación de sistemas en tiempo real y sistemas que tienen que reaccionar continuamente a estímulos internos y externos (eventos o sucesos). Las extensiones de las metodologías de análisis y diseño estructurado de **Ward y Mellor** [Ward and Mellor, 1985] y de **Hatley y Pirbhai** [Hatley and Pirbhai, 1987] son dos buenos ejemplos de estas metodologías.

4. **Diseño basado en el conocimiento**: Es una aproximación que se encuentra aún en una fase temprana de desarrollo. Utiliza técnicas y conceptos de Inteligencia Artificial para especificar y generar sistemas de información. El método **KADS** (Knowledge Acquisition and Development Systems) [Wielinga et al. 1991] y la metodologías **IDEAL** [Gómez et al., 1998] son ejemplos de esta categoría.

5. **Orientadas a objetos**: Estas metodologías se fundamentan en la integración de los dos aspectos de los sistemas de información: datos y procesos. En este paradigma un sistema se concibe como un conjunto de objetos que se comunican entre sí mediante mensajes. El objeto encapsula datos y

operaciones. Este enfoque permite un modelado más natural del mundo real y facilita enormemente la reusabilidad. Algunos representantes de este grupo son las metodologías **OOA/D** de **Grady Booch** [Booch, 1994], **OMT** (Object Modeling Technique) [Rumbaugh et al., 1991], **OOSE** (Object Oriented Software Engineering) de [Jacobson et al., 1993], **FUSION** propuesta por [Coleman et al., 1994], **MOSES** de [Henderson-Sellers and Edwards, 1994a] o **RUP** (Rational Unified Process) [Jacobson et al., 1999].

6. **Basadas en métodos formales**: Estas metodologías implican una revolución en los procedimientos de desarrollo, ya que a diferencia de todas las anteriores, estas técnicas se basan en teorías matemáticas que permiten una verdadera aproximación científica y rigurosa al desarrollo de sistemas de información y software asociado.

Existen otras clasificaciones de las metodologías, por ejemplo en [Piattini et al., 1996] éstas se clasifican en función de tres parámetros *el enfoque, el tipo de sistema* y *la formalidad*.

Atendiendo a todo lo expuesto parece imprescindible incluir en la asignatura de Ingeniería del Software los conceptos relacionados con los actores implicados en el desarrollo de proyectos, aspectos del desarrollo y la calidad de los productos finales, así como los procedimientos, herramientas y métodos de trabajo a disposición del analista para poder construir el software de calidad que el usuario necesita. Teniendo en cuenta que entre los enfoques metodológicos mencionados anteriormente la Orientación a Objetos es una de las líneas más prometedoras y que las orientadas a procesos y las orientadas a estados y transiciones siguen siendo las más utilizadas y difundidas en la actualidad, estos serán los enfoques metodológicos que se incluirán en el programa de la asignatura.

Una vez que se ha introducido cual es el marco conceptual de la asignatura, se va a perfilar de una manera más concreta el conjunto de conocimientos que entrarían dentro del área de influencia de la Ingeniería del Software: *su cuerpo de conocimiento*.

4.2.3.1 El cuerpo de conocimiento de la Ingeniería del Software

Como se indicó al comienzo de este capítulo una de las tareas básicas para la definición de una profesión es el establecimiento del conjunto de conocimientos que el profesional debe poseer para el adecuado ejercicio de su labor profesional. Este cuerpo de conocimiento es fundamental para constituir el resto de los elementos que conformarán la profesión, esto es, una propuesta curricular y una política de certificación de los estudios y de los profesionales.

Ante esta necesidad se han propuesto diferentes alternativas, algunas de las cuales van a ser presentadas someramente en los siguientes apartados.

SWEBOK propuesto por IEEE-CS y ACM

De las diferentes propuestas el proyecto **SWEBOK** (*Software Engineering Body of Knowledge*), patrocinado por **IEEE-CS** y **ACM**, es el que acabará acatándose como estándar internacional, aunque a fecha de hoy todavía no se ha finalizado estando en la segunda fase (*versión del hombre de piedra* [Abran et al., 1999], [Abran et al., 2000]) de las tres fases de que consta el proyecto, esperando que finalice a lo largo del año 2001, como se aprecia en la Figura 4.1.

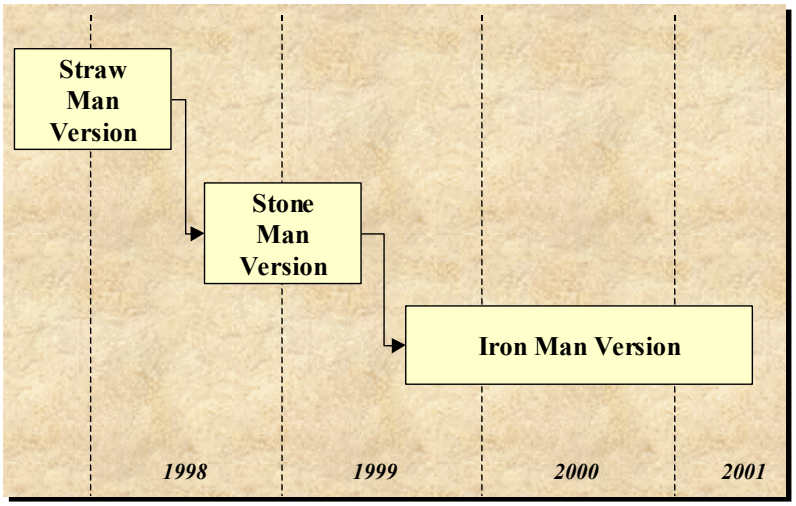

Figura 4.1. Duración aproximada del proyecto SWEBOK

Los objetivos de este proyecto se resumen en los siguientes cinco puntos [Bourque et al., 1999a]:

1. Caracterizar los contenidos de la Ingeniería del Software como disciplina.
2. Ofrecer un acceso al cuerpo de conocimiento de la Ingeniería del Software.
3. Promover una vista consistente de la Ingeniería del Software para todo el mundo.
4. Clarificar el lugar, y establecer los límites, de la Ingeniería del Software con respecto a otras disciplinas, tales como la Ciencia de la Informática, la Gestión de Proyectos, la Ingeniería de Computadores o las Matemáticas.
5. Ofrecer las bases para el desarrollo de una propuesta curricular y una política de certificación, relacionadas ambas con la Ingeniería del Software.

El producto que resulte de este proyecto no será el cuerpo de conocimiento en sí, sino más bien una guía de él. El conocimiento ya existe, lo que se busca es el consenso para determinar el subconjunto de conceptos esenciales que caracterizan a la Ingeniería del Software.

La guía que, como ya se ha mencionado, actualmente se encuentra en la parte final de su segunda fase (*versión del hombre de piedra*), se divide en diez áreas de conocimiento, con una serie de especialistas responsables de cada una de ellas, que se recogen en la Tabla 4.1.

Área de Conocimiento	Especialistas
Gestión de la configuración del software	J.A. Scott y D. Nisse (Lawrence Livermore Laboratory, USA)
Construcción del software	T. Bollinger (The Mitre Corporation, USA)
Diseño del software	G. Tremblay (Université du Québec à Montreal, Canadá)
Infraestructura de la Ingeniería del Software	D. Carrington (The University of Queensland, Australia)
Gestión de la Ingeniería del Software	S.G. MacDonell y A.R. Gray (University of Otago, Nueva Zelanda)
Proceso de Ingeniería del Software	K. El Emam (National Research Council, Canadá)
Evolución y mantenimiento del software	T.M. Pigoski (Techsoft, USA)
Análisis de la calidad del software	D. Wallace y L. Reeker (National Institute of Standards and Technology, USA)
Análisis de los requisitos del software	P. Sawyer y G. Kotonya (Lancaster University, Reino Unido)
Prueba del Software	A. Bertolino (National Research Council, Italia)

Tabla 4.1. Las áreas de conocimiento del SWEBOK y sus responsables

Además, se han considerado siete disciplinas relacionadas con la Ingeniería del Software: *Ciencias cognitivas y factores humanos, Ingeniería de computadores, Ciencia de la Informática, Gestión y ciencia de la gestión, Matemáticas, Gestión de proyectos* e *Ingeniería de Sistemas*.

El proyecto SWEBOK especifica las unidades de conocimiento, así como los temas pertenecientes a dichas áreas de conocimiento, que se considerará el conocimiento esencial que debe poseer un ingeniero del software. Éstos deben también poseer ciertos conocimientos de las disciplinas relacionadas, pero no es cometido del SWEBOK especificar estos conocimientos.

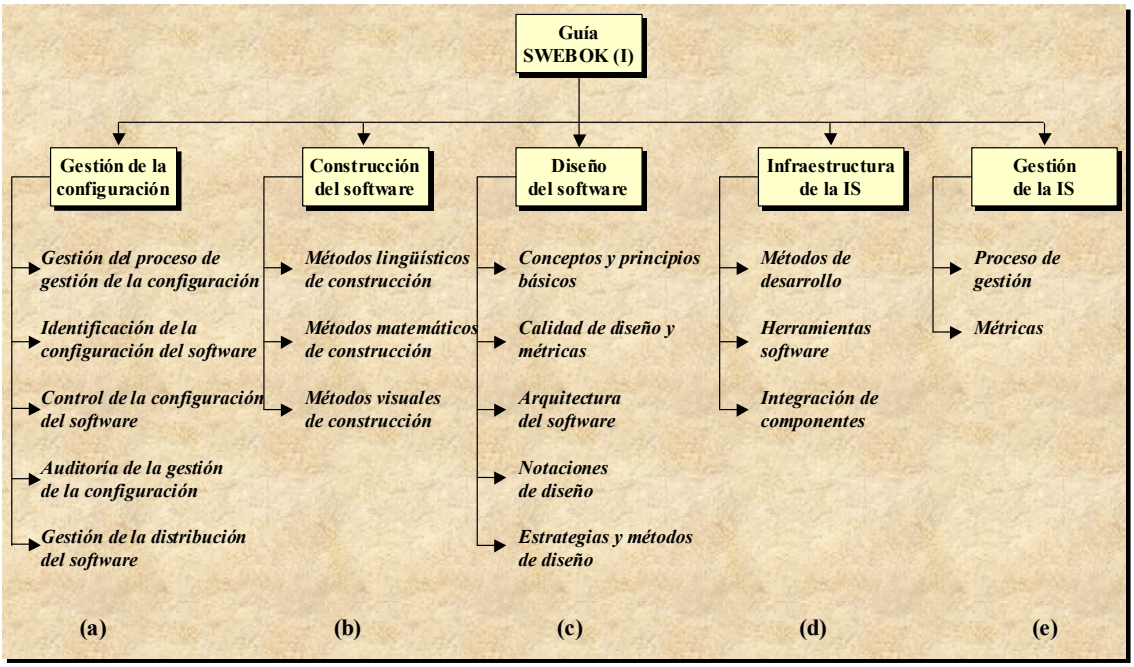

Figura 4.2. Estructura de la guía SWEBOK (parte I)

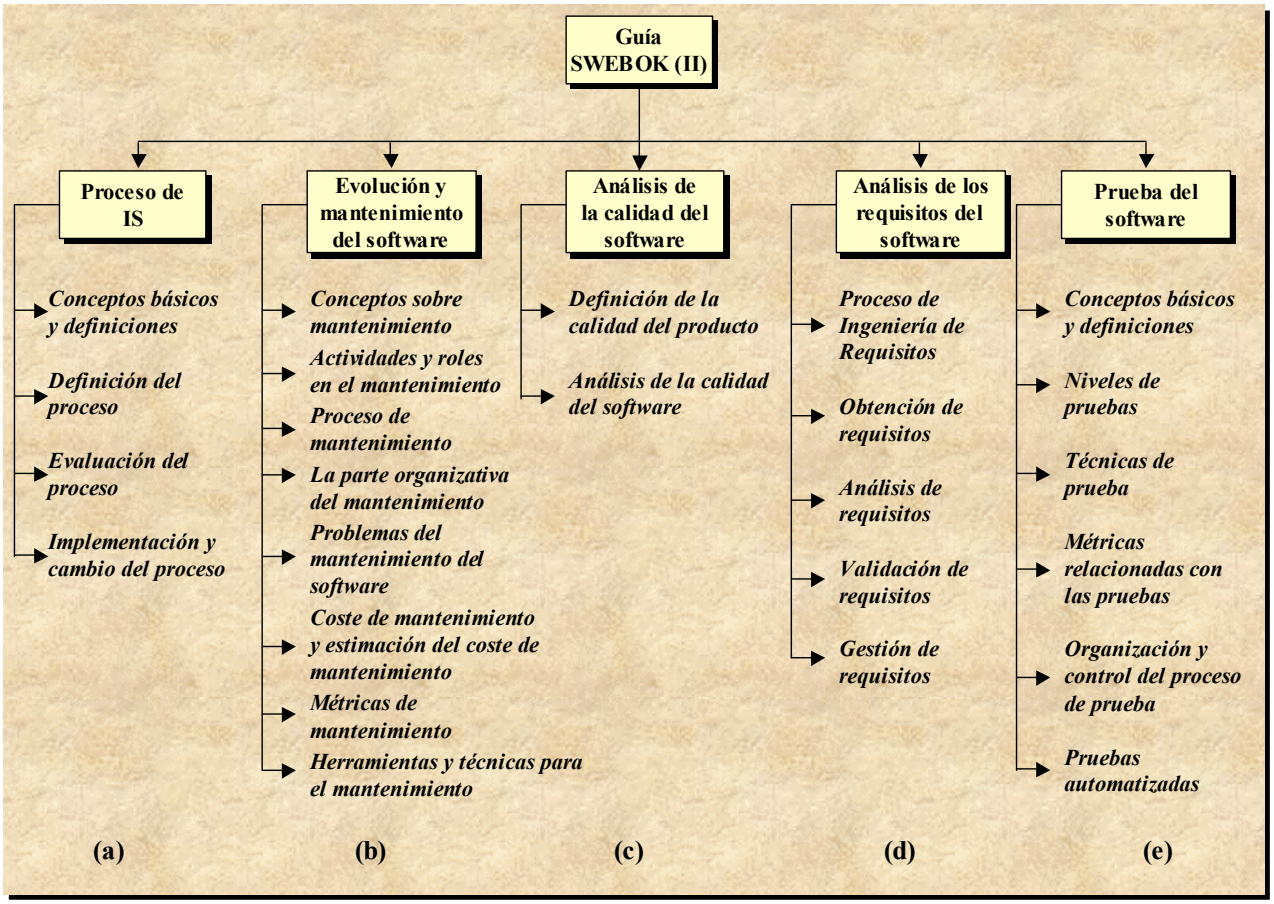

Figura 4.3. Estructura de la guía SWEBOK (parte II)

La guía del SWEBOK se organiza de forma jerárquica, descomponiendo cada área de conocimiento en un conjunto de temas con nombres fácilmente reconocibles, como se puede apreciar en la Figura 4.2 y en la Figura 4.3. Una breve descripción de cada una de las diez áreas de conocimiento se presenta en el Cuadro 4.2.

El número de áreas de conocimiento no ha sido siempre de diez, y muy probablemente este número variará en la versión definitiva de la guía. En la *Straw Man Version* [Bourque et al., 1998] fueron quince las áreas de conocimiento consideradas; éstas se obtuvieron del estudio de libros de texto sobre Ingeniería del Software, del estudio de programas de cursos universitarios y masters en Ingeniería del Software, y especialmente el ciclo de vida **ISO/IEC 12207**[19] [ISO/IEC, 1995] fue considerado como una entrada principal para esta versión de la guía, marcando las bases y el vocabulario para la clasificación de los diferentes temas relacionados con el ciclo de vida.

[19] Adoptado por IEEE/EIA y por ISO/IEC como un estándar.

Gestión de la configuración del software: Es la disciplina que identifica la configuración de puntos discretos en el tiempo para lograr un control sistemático de sus cambios y mantener la integridad y la trazabilidad a través del ciclo de vida del sistema.

Construcción del software: Es el acto fundamental de la Ingeniería del Software. Requiere el establecimiento de un diálogo entre el ingeniero y el computador, entre representantes de dos mundos totalmente diferentes. Para establecer los temas de esta unidad de conocimiento se adoptan dos vistas complementarias de la construcción del software. La primera de ellas establece las tres interfaces principales para la creación de software: *lingüista, matemática* y *visual*. La segunda establece que, para cada uno de los estilos, los temas a tratar se organicen de acuerdo a cuatro principios de organización: *reducción de la complejidad, anticipación de la diversidad, estructuración de la validación* y *uso de estándares externos*.

Diseño del software: Transforma los requisitos, típicamente expresados en términos del dominio del problema, en descripciones que explican cómo resolver el problema. Describe cómo el sistema se descompone y se organiza en componentes, describiendo las interfaces entre ellos.

Infraestructura de la Ingeniería del Software: Describe tres subáreas que discurren de forma horizontal a través de otras áreas de conocimiento: *los métodos de desarrollo, las herramientas software* y *la integración de componentes*.

Gestión de la Ingeniería del Software: Une la gestión del proceso y la parte de métricas del proceso. La gestión del proceso casa con la noción de "*gestión a la larga*", esto es, trata de la organización de las fases del ciclo de vida. La parte de métricas aborda *la medida de los objetivos del programa, la medida de la selección, la recolección de datos* y *el desarrollo de modelos*.

Proceso de la Ingeniería del Software: Cubre la definición, implementación, medida, gestión, cambio y mejora de los procesos software.

Evolución y mantenimiento del software: Estudia los procesos relacionados con la modificación de un producto software después de su entrega, para corregir faltas, mejorar su rendimiento u otros atributos, o adaptar el producto a otro entorno. Pero sin embargo, normalmente más que considerar que un sistema software está terminado, se piensa que evoluciona constantemente, de ahí la inclusión de la parcela de evolución del software.

Análisis de la calidad del software: Discute sobre el aseguramiento de la calidad que todo producto resultado de un proceso de Ingeniería del Software debe tener.

Análisis de los requisitos del software: Se divide en cinco subáreas que se corresponden aproximadamente con las actividades del proceso que se desarrollan iterativa y concurrentemente, más que de forma secuencial. La parcela del *proceso de ingeniería de requisitos* presenta e introduce las otras cuatro. La subárea de *obtención de requisitos* cubre la captura, descubrimiento o adquisición de éstos. La parte de *análisis* trata de solventar los conflictos entre los requisitos, así como la frontera del sistema. La *validación de requisitos* busca conflictos, omisiones y ambigüedades; a la vez que asegura que los requisitos siguen un estándar de calidad. Por último, la subárea de *gestión de requisitos* permite mantener los requisitos presentes en todo el ciclo de vida, adaptándolos a los cambios sufridos por el proyecto.

Prueba del software: Consiste en la verificación dinámica del comportamiento de los programas ante un conjunto finito de casos de prueba.

Cuadro 4.2. Descripción de las áreas de conocimiento del SWEBOK en la Stone Man Version v0.5

En la Tabla 4.2 se recoge la equivalencia entre las áreas de conocimiento propuestas en las dos versiones de la guía SWEBOK existentes hasta la fecha.

Área de Conocimiento en la Stone Man Version	Áreas de Conocimiento correspondientes en la Straw Man Version	Notas
Gestión de la configuración del software	Gestión de la configuración*	
Construcción del software	Codificación*	
Diseño del software	Diseño detallado*	Se decidió cubrir todo el diseño desde el principio, no distinguiendo entre diseño arquitectónico y diseño detallado. Esta distinción aparece en el estándar 12207
Infraestructura de la Ingeniería del Software	Métodos de desarrollo (orientados al objeto, formales, prototipado) Entornos de desarrollo de software	Los entornos de desarrollo (herramientas) y la reutilización se consideraron los dos elementos principales del proceso de infraestructura. Los métodos se incluyeron porque con frecuencia las herramientas se construyen para implementar métodos determinados
Gestión de la Ingeniería del Software	Proceso de gestión* Medida/Métricas	La definición de este proceso en el estándar 12207 hace mención del uso de datos cuantificables para la toma de decisiones, por eso la parte de *métricas* se ha agrupado con la de *proceso de gestión*
Proceso de Ingeniería del Software	Proceso de mejora*	
Evolución y mantenimiento del software	Proceso de mantenimiento*	
Análisis de la calidad del software	Aseguramiento de la calidad* Verificación y validación* Confiabilidad del software	Todas las áreas relacionadas con la calidad se han agrupado en una sola área de conocimiento
Análisis de los requisitos del software	Análisis de requisitos*	
Prueba del Software	Prueba*	
	Presentación y definición de la Ingeniería del Software	Se incluyó en la Straw Man Version porque aparecía en todos los libros de texto sobre Ingeniería del Software, y se necesita algún tipo de introducción en la Stone Man Version, pero no ha sido incluida en ninguna área como tal. Cuando se establezca el formato final de la Stone Man Version se decidirá como introducirla

* Área de la Straw Man Version basada en el estándar ISO/IEC 12207.

Tabla 4.2. Equivalencia entre la lista de áreas de conocimiento de la Stone Man Version y de la Straw Man Version de la guía SWEBOK [Bourque et al., 1999b]

En la Tabla 4.3 se establece la correspondencia entre las áreas de conocimiento propuestas en la Stone Man Version v0.5 de la guía SWEBOK y el estándar ISO/IEC 12207.

Área de Conocimiento de la Stone Man	Estándar ISO/IEC 12207	
Análisis de los requisitos del software	Análisis de requisitos	PROCESOS PRINCIPALES
Diseño del software	Diseño arquitectónico Diseño detallado	
Construcción del software	Codificación Integración	
Prueba del Software	Prueba Instalación Soporte a la aceptación Proceso de operación Explotación del software Soporte operativo a los usuarios	
Evolución y mantenimiento del software	Proceso de mantenimiento	
Gestión de la configuración del software	Gestión de la configuración	PROCESOS DE SOPORTE
Análisis de la calidad del software	Aseguramiento de la calidad Verificación y validación Revisión conjunta Auditoría	
Gestión de la Ingeniería del Software	Procesos de gestión	PROCESOS DE LA ORGANIZACIÓN
Infraestructura de la Ingeniería del Software	Proceso de infraestructura	
Proceso de Ingeniería del Software	Proceso de mejora	

Tabla 4.3. Correspondencia entre las áreas de conocimiento de la guía SWEBOK Stone Man Version v0.5 y el estándar ISO/IEC 12207 [Bourque et al., 1999b]

SWE-BOK propuesto para la FAA

Este cuerpo de conocimiento (denotado por las siglas SWE-BOK) es el resultado de un trabajo patrocinado por la **FAA** (Federal Aviation Administration) de EEUU como parte de un proyecto para mejorar los procesos de adquisición, desarrollo y mantenimiento del software de dicha entidad.

Este cuerpo de conocimiento pretende contribuir al trabajo que está realizando el SWECC (Software Engineering Coordination Committee) bajo el patrocinio de ACM e IEEE-CS para el desarrollo de la Ingeniería del Software y su madurez como disciplina.

Este cuerpo de conocimiento se recoge en [Hilburn et al., 1999], donde el término *conocimiento* se utiliza para describir el espectro completo del contenido de la

disciplina: *información, terminología, artefactos, datos, roles, métodos, modelos, procedimientos, técnicas, prácticas, procesos* y *bibliografía*.

Este cuerpo de conocimiento se estructura en tres niveles de abstracción: *categorías de conocimiento, áreas de conocimiento* y *unidades de conocimiento*, para lograr así un balance entre la simplicidad y la claridad y el nivel apropiado de detalle en la descripción del conocimiento. En el Cuadro 4.3 se recogen las definiciones que se manejan en esta estructuración, mientras que en la Figura 4.4 se pueden apreciar estos niveles de abstracción de forma gráfica.

Conocimiento: Término utilizado para describir el espectro completo de los contenidos de la disciplina: *información, terminología, artefactos, datos, roles, métodos, modelos, procedimientos, técnicas, prácticas, procesos* y *bibliografía*.

Cuerpo de conocimiento: Descripción jerárquica del conocimiento sobre la Ingeniería del Software que organiza y estructura el conocimiento en tres niveles de jerarquía: *categorías de conocimiento, áreas de conocimiento* y *unidades de conocimiento*.

Categoría de conocimiento: Una subdisciplina de la Ingeniería del Software que es generalmente reconocida como una parte significativa de este cuerpo de conocimiento de la Ingeniería del Software. Son elementos estructurales de alto nivel, utilizados para organizar, clasificar y describir el conocimiento sobre Ingeniería del Software. Cada una de ellas está compuesta por un conjunto de áreas de conocimiento.

Área de conocimiento: Una subdivisión de una categoría de conocimiento que representa el conocimiento de la Ingeniería del Software que está lógicamente cohesionado y relacionado con la categoría de conocimiento mediante la herencia o la agregación. Cada una de ellas está compuesta de un conjunto de unidades de conocimiento.

Unidad de conocimiento: Una subdivisión de un área de conocimiento que representa un componente básico del cuerpo de conocimiento de la Ingeniería del Software que tiene una descripción explícita. Para el propósito de este cuerpo de conocimiento, cada una de estas unidades es atómica; esto es, no se subdivide en elementos más básicos.

Cuadro 4.3. Definiciones manejadas en el SWE-BOK de la FAA [Hilburn et al., 1999]

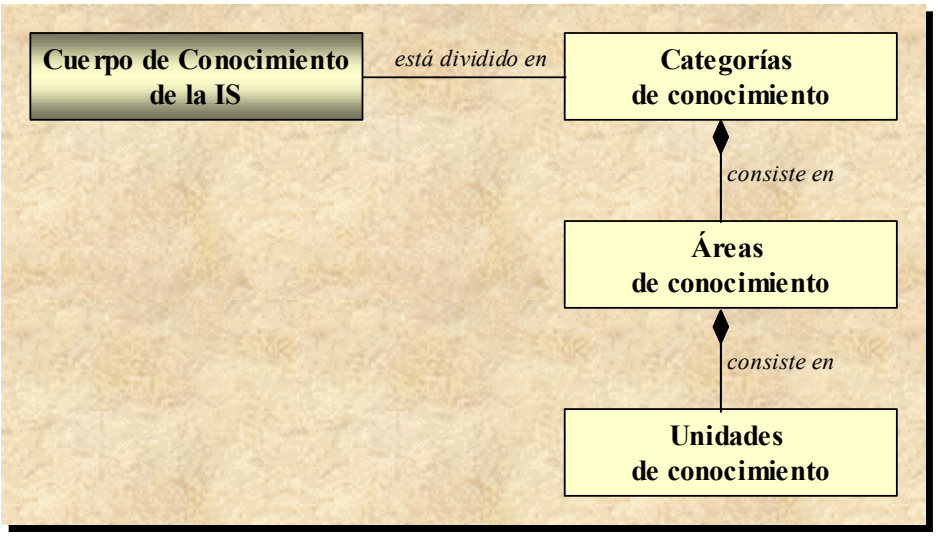

Figura 4.4. Niveles de abstracción en la arquitectura del SWE-BOK de la FAA [Hilburn et al., 1999]

Este cuerpo de conocimiento está formado por cuatro categorías de conocimiento: *Fundamentos de Informática, Ingeniería del Producto Software, Gestión del Software* y *Dominios Software*. En las siguientes tablas se resume este cuerpo de conocimiento.

1. FUNDAMENTOS DE INFORMÁTICA	
Descripción:	*Esta categoría concierne al conocimiento, conceptos, teoría, principios, métodos, propiedades y aplicaciones de Informática que forman la base para el desarrollo de software y de la disciplina de la Ingeniería del Software*
ÁREAS DE CONOCIMIENTO	**UNIDADES DE CONOCIMIENTO**
1.1 Algoritmos y estructuras de datos	*1.1.1 Estructuras de datos básicas* *1.1.2 Diseño de algoritmos* *1.1.3 Análisis de algoritmos*
1.2 Arquitectura del computador	*1.2.1 Sistemas digitales* *1.2.2 Organización de un sistema computacional* *1.2.3 Arquitecturas alternativas* *1.2.4 Comunicaciones y redes*
1.3 Fundamentos matemáticos	*1.3.1 Lógica matemática y prueba de sistemas* *1.3.2 Estructuras matemáticas discretas* *1.3.3 Sistemas formales* *1.3.4 Combinatoria* *1.3.5 Probabilidad y estadística*
1.4 Sistemas operativos	*1.4.1 Fundamentos de sistemas operativos* *1.4.2 Gestión de procesos* *1.4.3 Gestión de memoria* *1.4.4 Seguridad y protección* *1.4.5 Sistemas distribuidos y de tiempo real*
1.5 Lenguajes de programación	*1.5.1 Teoría de lenguajes de programación* *1.5.2 Paradigmas de programación* *1.5.3 Diseño e implementación de lenguajes de programación*

Tabla 4.4. Categoría de Fundamentos de Informática

2. INGENIERÍA DEL PRODUCTO SOFTWARE	
Descripción:	*Esta categoría se refiere a un conjunto de actividades bien definidas e integradas para producir productos software consistentes. Incluye actividades técnicas, que involucran la documentación de estos productos y el mantenimiento de la traza y la consistencia entre ellos. También se refiere al control de la transición entre las diferentes fases del ciclo de vida del software, así como a las actividades que se necesitan para ofrecer productos software de alta calidad a los clientes.*
ÁREAS DE CONOCIMIENTO	**UNIDADES DE CONOCIMIENTO**
2.1 Ingeniería de requisitos del software	*2.1.1 Obtención de requisitos* *2.1.2 Análisis de requisitos* *2.1.3 Especificación de requisitos*
2.2 Diseño del software	*2.2.1 Diseño arquitectónico* *2.2.2 Especificación abstracta* *2.2.3 Diseño de la interfaz* *2.2.4 Diseño de las estructuras de datos* *2.2.5 Diseño de algoritmos*
2.3 Codificación del software	*2.3.1 Implementación de código* *2.3.2 Reutilización de código* *2.3.3 Estándares de codificación y documentación*
2.4 Prueba del software	*2.4.1 Pruebas de unidad* *2.4.2 Pruebas de integración* *2.4.3 Pruebas del sistema* *2.4.4 Pruebas de rendimiento* *2.4.5 Pruebas de aceptación* *2.4.6 Pruebas de instalación* *2.4.7 Documentación de la prueba*
2.5 Explotación y mantenimiento	*2.5.1 Instalación y explotación del software* *2.5.2 Operaciones de mantenimiento del software* *2.5.3 Proceso del mantenimiento del software* *2.5.4 Gestión del mantenimiento del software* *2.5.5 Reingeniería del software*

Tabla 4.5. Categoría de Ingeniería del Producto Software

3. GESTIÓN DEL SOFTWARE	
Descripción:	*Esta categoría casa con los conceptos, métodos y técnicas para la gestión de los productos y proyectos software. Incluye actividades relacionadas con la gestión de proyectos, gestión de riesgos, calidad del software y gestión de la configuración.*
ÁREAS DE CONOCIMIENTO	**UNIDADES DE CONOCIMIENTO**
3.1 Gestión del proyecto software	*3.1.1 Planificación del proyecto* *3.1.2 Organización del proyecto* *3.1.3 Estimación del proyecto* *3.1.4 Calendario del proyecto* *3.1.5 Control del proyecto*
3.2 Gestión de riesgos del software	*3.2.1 Análisis del riesgo* *3.2.2 Planificación de la gestión del riesgo* *3.2.3 Monitorización del riesgo*
3.3 Gestión de la calidad del software	*3.3.1 Aseguramiento de la calidad del software* *3.3.2 Verificación y validación* *3.3.3 Métricas del software* *3.3.4 Sistemas dependientes*
3.4 Gestión de la configuración software	*3.4.1 Identificación de la configuración del software* *3.4.2 Control de la configuración del software* *3.4.3 Auditoría de la configuración del software* *3.4.4 Contabilidad del estado de la configuración del software*
3.5 Gestión del proceso software	*3.5.1 Gestión cuantitativa del proceso del software* *3.5.2 Mejora del proceso del software* *3.5.3 Evaluación del proceso software* *3.5.4 Automatización del proceso software* *3.5.5 Ingeniería del proceso software*
3.6 Adquisición del software	*3.6.1 Gestión de la obtención* *3.6.2 Planificación de la adquisición* *3.6.3 Gestión del rendimiento*

Tabla 4.6. Categoría de Gestión del Software

4. DOMINIOS SOFTWARE	
Descripción:	*Esta categoría tiene que ver con el conocimiento de dominios específicos que involucran la utilización y aplicación de la Ingeniería del Software*
ÁREAS DE CONOCIMIENTO	
4.1 Inteligencia artificial	
4.2 Sistemas de bases de datos	
4.3 Interacción hombre-máquina	
4.4 Computación numérica y simbólica	
4.5 Simulación por computadora	
4.6 Sistemas de tiempo real	

Tabla 4.7. Categoría de Dominios Software

SE-BOK propuesto por el WGSEET

El fin último del **WGSEET** (*Working Group on Software Engineering Education and Training*) es desarrollar un modelo de currículo para la Ingeniería del Software que pueda ser aplicado en todo, o en parte, en el desarrollo de programas educativos especializados en Ingeniería del Software. Esta propuesta se recoge en [Bagert et al., 1999] (y de forma más esquemática en [Hilburn et al., 1998]).

Como parte importante de este proyecto está la definición de un cuerpo de conocimiento de la Ingeniería del Software que sirva de base a la propuesta curricular. Este cuerpo de conocimiento (denotado por SE-BOK en este trabajo) se organiza en cuatro áreas de conocimiento: *el área central, el área de fundamentos, el área de conceptos recurrentes* y *el área de soporte*.

SE-BOK [Bagert et al., 1999]	
ÁREAS DE CONOCIMIENTO	**COMPONENTES DE CONOCIMIENTO**
Área Central – *incluye aquellos componentes que definen la esencia de la Ingeniería del Software*	• Requisitos del software • Diseño del software • Construcción del software • Gestión de proyectos software • Evolución del software
Área de Fundamentos – *incluye aquellos componentes que sirve de base a las áreas central y de conceptos recurrentes*	• Fundamentos de Informática • Factores humanos • Dominios de aplicación
Área de Conceptos Recurrentes – *son elementos que discurren por todos los componentes de conocimiento del área central*	• Ética y profesionalismo • Procesos software • Calidad del software • Modelado de software • Métricas del software • Herramientas y entornos • Documentación
Área de Soporte – *incluye otros campos de estudio que ofrecen los conocimientos necesarios para completar la educación de los ingenieros del software*	• Educación general • Matemáticas • Ciencias naturales • Ciencias sociales • Empresariales • Ingeniería • ...

Tabla 4.8. SE-BOK propuesto por el WGSEET

Comparativa

De los diferentes cuerpos de conocimiento que se han presentado, el propuesto por **IEEE-CS/ACM** es el que más respaldo internacional tiene y, al igual que el propuesto por el **WGSEET**, tiene entre sus cometidos servir de base para la definición de una propuesta curricular para la disciplina de la Ingeniería del Software.

El propuesto para la **FAA** es el que abarca unos contenidos más amplios, buscando definir y evaluar las competencias software que se necesitan en una organización con unas dependencias altas de los sistemas software.

El **SWEBOK** de **IEEE-CS/ACM** es la única propuesta que establece la frontera de la Ingeniería del Software identificando otras disciplinas relacionadas, ya que las otras propuestas de una u otra forma introducen en el cuerpo de conocimiento temas relacionados con otras disciplinas. Una de las intersecciones que más se repite es con la parte de **Gestión de Proyectos**, que por otra parte tiene su propio cuerpo de conocimiento [Duncan, 1996].

4.2.4 La enseñanza de la Ingeniería del Software

Tras la revisión anterior sobre los conceptos relacionados con la disciplina de la Ingeniería del Software, se van a efectuar algunas consideraciones de tipo general sobre la enseñanza de la Ingeniería del Software.

Sin duda, una característica de la Ingeniería del Software como disciplina universitaria, frente a otras disciplinas más establecidas, es el dinamismo con el que cambian sus conceptos y herramientas. Por ello es importante atender a las recomendaciones sobre la enseñanza de la Ingeniería del Software que realizan los organismos y organizaciones internacionales relacionados con la Ciencia de la Informática, los Sistemas de Información y la propia Ingeniería del Software. Atender a este tipo de recomendaciones permite que los conocimientos transmitidos y las habilidades adquiridas por los alumnos respondan a las necesidades profesionales reales y no queden obsoletos en poco tiempo.

Del análisis realizado en el apartado anterior se puede deducir que la Ingeniería del Software consiste en un gran número de actividades interrelacionadas que resultan muy difíciles de conjugar bajo un único epígrafe. Por ello, y sobre todo con fines educativos, es imprescindible introducir una cierta organización en esos contenidos, que permita la definición de un programa educativo consistente y ordenado.

En este sentido en [Ardis and Ford, 1989], [Ford, 1991a] se identifican para la formación en Ingeniería del Software los puntos de vista del proceso de ingeniería y de los productos resultantes. A continuación se exponen brevemente las características más importantes de esta doble visión en formación en Ingeniería del Software.

❖ **Punto de vista del proceso**

El proceso de la Ingeniería del Software incluye un amplio rango de actividades realizadas por los ingenieros de software; pero a lo largo de este rango muchos aspectos de estas actividades son similares. Los elementos del proceso pueden considerarse en dos dimensiones: *la actividad* y *el aspecto*.

- *Actividad.* Las actividades del proceso se dividen en cuatro grupos: *desarrollo, control, dirección* y *operaciones*.
    - ***Actividades de Desarrollo.*** Creación o producción del software de los componentes del sistema, incluyendo análisis de requisitos, especificación, diseño, implementación y prueba.
    - ***Actividades de Control***. Estas actividades están más relacionadas con el control del desarrollo que con la producción del software. Las dos actividades principales del control hacen referencia a la evolución del software y a la calidad del software. En este apartado se consideran las actividades relacionadas con la dirección de la configuración, el control de los cambios, el mantenimiento, el control de la calidad, las pruebas, la evaluación, la verificación y la validación.
    - ***Actividades de Dirección***. Implica la dirección ejecutiva, administrativa, y supervisora de un proyecto de software, incluyendo las actividades técnicas que soportan el proceso ejecutivo de decisión. Las actividades que se pueden considerar aquí son las de planificación del proyecto, localización de recursos, organización de equipos de trabajo, estimación de costes y aspectos legales.
    - ***Actividades de Operación***. Relacionado con el uso de los sistemas de software. Estas actividades incluyen formación del personal en el uso del sistema, planificación para la entrega e instalación de sistemas, cambio desde el sistema antiguo (manual o automático) al nuevo, operación del software y retirada del sistema.
- *Aspecto*. Las actividades de la Ingeniería se dividen tradicionalmente en actividades analíticas y sintéticas. Se consideran seis aspectos de estas actividades para recoger esta distinción: *abstracción, representación, métodos, herramientas, medición* y *comunicaciones*.
    - ***Abstracción.*** Incluye los principios fundamentales y los modelos formales (Modelos del proceso de desarrollo del software, máquinas de estados finitos y redes de Petri, modelo COCOMO...).
    - ***Representación***. Incluye notaciones y lenguajes (Ada, Tablas de Decisión, diagramas de flujo, PERT).

- ***Métodos.*** Incluye métodos formales, practicas actuales, y metodologías (OOD, Programación estructurada...).
- ***Herramientas***. Incluye los conjuntos de herramientas software individuales e integradas (e implícitamente los sistemas hardware donde se ejecutan). Pueden mencionarse las de propósito general (correo electrónico y proceso de textos), las herramientas relativas al diseño e implementación (compiladores y editores sensitivos a la sintaxis) y las herramientas de control de proyectos.
- ***Medición***. Los aspectos de medición incluyen análisis de medidas y evaluación de los productos y el proceso software, así como el impacto del software en la organización; en esta categoría hay que incluir las métricas y los estándares. Esta área merece ser tenida en cuenta en la formación ya que los ingenieros de software, al igual que los ingenieros de los campos tradicionales, necesitan conocer qué medir, cómo medirlo y cómo utilizar los resultados para analizar y evaluar cómo progresan los procesos y productos.
- ***Comunicación***. La comunicación es el último aspecto. Todas las actividades de los ingenieros de software implican comunicación tanto oral como escrita, así como producción de documentación. Un ingeniero de software debe tener unas buenas habilidades en las técnicas generales de comunicación y una comprensión de las formas apropiadas de documentación para cada actividad [Levine et al., 1991].

❖ **Punto de vista del producto**

A menudo es conveniente discutir las actividades y aspectos en el contexto de una determinada clase de sistema de software; por ejemplo la programación concurrente y la secuencial tienen características diferenciadoras. Así, se añaden dos nuevas dimensiones a la estructura organizacional del contenido curricular: *las clases de sistemas software* y *los requisitos del sistema*.

- *Clases de sistemas software*. De las distintas clases que pueden ser consideradas, un grupo se define en función de las relaciones del sistema con su entorno, y sus elementos (partes) están descritos por términos tales como procesamiento por lotes, interactivo, reactivo, tiempo real. Otro grupo tiene elementos descritos en términos tales como distribuido, concurrente, o red. Otro esta definido en función de las características internas, tales como orientado a tablas, orientado a procesos o basado en conocimientos. También, se incluyen áreas de aplicación genéricas o específicas, sistemas de aviónica, sistemas de comunicaciones, sistemas operativos, sistemas de base de datos.

- *Requisitos del sistema*. La discusión de los requisitos del sistema generalmente se centra en los requisitos funcionales, pero existen otras categorías que merecen atención. Identificar y reunir esos requisitos es el resultado de las actividades realizadas a lo largo del proceso de Ingeniería del Software. Ejemplos de estos requisitos son: accesibilidad, adaptabilidad, disponibilidad, compatibilidad, exactitud, eficiencia, tolerancia a fallos, integridad, interoperabilidad, mantenibilidad, rendimiento, portabilidad, protección, formalidad, reusabilidad, robustez, seguridad, comprobabilidad y usabilidad.

Otra forma de enfocar la enseñanza de la Ingeniería del Software es mediante un enfoque basado en un proyecto para que el alumno se aproxime al trabajo tal cual ocurre (*o debiera ocurrir*) en la realidad empresarial [Tomayko, 1987], [Shaw and Tomayko, 1991].

En Sicología se distinguen dos tipos de conocimientos: *declarativo* y *procedural* [Norman, 1988]. El primero es fácil de transcribir y de enseñar; sin embargo, el segundo es imposible de transcribir y difícil de enseñar, siendo más sencillo de transmitir mediante demostración y de aprender por la práctica. Muchos de los procesos de la Ingeniería del Software dependen del conocimiento procedural. Por este motivo se recomienda una importante parte de experiencia adquirida mediante proyectos [Ardis and Ford, 1989].

En [Shaw and Tomayko, 1991] se presentan diferentes modelos de cursos de Ingeniería del Software que toman el proyecto como eje conductor de los mismos. Los modelos que se enuncian son:

- ***Modelo de Ingeniería del Software como producto***. Es el modelo al que se ajustan los cursos compuesto exclusivamente por clases teóricas. Son cursos que concentran en, aproximadamente, un cuatrimestre los conceptos de la Ingeniería del Software. Su mayor desventaja es la ausencia de parte práctica, y por tanto de experiencia. Estos cursos se adecuan a lo que se denomina *énfasis por el ciclo de vida*, que se ajusta a la forma de organizar los libros de texto sobre Ingeniería del Software, especialmente siguiendo el ciclo de vida en cascada, como claramente se aprecia en el libro de **Richard Fairley** [Fairley, 1985] o en ediciones anteriores de libros tan clásicos como el de **Roger S. Pressman** [Pressman, 1987] o **Ian Sommerville** [Sommerville, 1989].
- ***Modelo de la aproximación por seminarios.*** Es similar al anterior en el sentido de que la base del curso son las clases teóricas, pero se distingue en que en este modelo de curso, se reserva un tiempo para que los alumnos presenten trabajos que ellos mismos han realizado sobre algún tema concreto, manejando la bibliografía oportuna.

- ***Modelo de proyecto para grupos pequeños.*** Este modelo de curso incluye la realización de un proyecto de pequeñas dimensiones como parte del curso. Es muy seguido porque divide el curso en trabajo de clase y trabajo de proyecto. Los alumnos se dividen en grupos de entre tres y cinco personas, debiendo abordar un proyecto que les sea familiar y puedan terminar en el tiempo asignado al curso. Este curso permite obtener algunas experiencias derivadas de la aplicación de lo explicado en las clases teóricas, pero es deficiente en el sentido de que no les entrena para el trabajo en proyectos grandes.
- ***Modelo de proyecto para grandes equipos.*** Es la mejor opción para aprender las técnicas que se utilizan en los proyectos reales. La idea es realizar un proyecto con toda la clase. Típicamente se elige un proyecto consistente en el desarrollo de un producto software, idealmente destinado a un cliente real. Los alumnos se organizan en un solo equipo de desarrollo, asumiendo cada uno de ellos el rol que le sea asignado, y que coincidirá con los roles que aparecen en los entornos industriales reales. Cada alumno mantendrá el rol durante todo el curso, y aprenderá las actividades de los otros roles a través de la interacción con los otros miembros del equipo. Es inviable de llevar a cabo cuando el número de alumnos es alto y la asistencia no es obligatoria. Además, la relación con las actividades propias de otros roles no es tan intensa como las que uno lleva a cabo.
- ***Modelo de proyecto único.*** El curso entero se dedica a la realización de un proyecto. Suele llevarse a cabo cuando la Ingeniería del Software se divide en dos asignaturas independientes, una dedicada por completo a la teoría y la otra a la práctica.

Con independencia del modelo que se siga existen varios tipos de proyectos. En [Shaw and Tomayko, 1991] se hace un repaso de ellos, encontrando:

- ***El proyecto de "juguete".*** La clase se divide en equipos de 3 a 5 personas; cada equipo recibe una labor predefinida por el responsable de la asignatura. Puede existir una variante donde cada equipo crea su propia especificación. Las ventajas son que los alumnos aplican las enseñanzas de la Ingeniería del Software trabajando en equipo, aunque no se satisfagan los requisitos por completo o no se llegue a terminar la implementación, y los proyectos sean fáciles de gestionar. Se puede llegar a situaciones en las que alumnos de segundo ciclo actúen como gestores de los proyectos realizados por alumnos de primer ciclo. La mayor desventaja es que se omiten técnicas propias de los proyectos grandes como puede ser la gestión de la configuración. Es una práctica muy extendida.
- ***Proyecto basado en componentes.*** Los alumnos se organizan en grupos que desarrollan componentes software. Se establecen después una serie de

proyectos que se llevarán a cabo con los componentes realizados, para ello debe haber una primera etapa de *adquisición de componentes*, una segunda de integración de los mismos y una tercera de *modificación* o *adaptación* a las necesidades del grupo. Es un tipo de proyecto en el que se manejan técnicas propias de proyectos grandes, y los alumnos se acostumbran a diseñar componentes reutilizables. Su mayor desventaja está en su gestión, muy parecida al caso anterior.

- ***Proyecto para un cliente externo.*** Los estudiantes trabajan en componentes que deberán integrarse para la realización de un producto para un cliente externo, que deberá pasar los criterios de aceptación oportunos acordados con el cliente. Se puede llevar a cabo en pequeños equipos o con un gran equipo. Es el que más se acerca a la realidad, debiendo hacer uso de todas las técnicas propias de un proyecto de grandes dimensiones, donde la relación con el cliente será uno de los puntos más sobresalientes.
- ***Proyectos individuales.*** Cada equipo tiene un proyecto diferente. Los equipos pueden tener clientes externos, que con frecuencia son facilitados por el responsable de la asignatura. Puede convertirse en una caos desde la perspectiva de gestión de la globalidad de los proyectos.

4.3 Programación Orientada a Objetos

4.3.1 Introducción

Al igual que se hizo con la Ingeniería del Software, se va a realizar un repaso global por la Orientación a Objetos, presentando su marco histórico y conceptual.

La Orientación a Objetos forma parte de la Ingeniería del Software como un paradigma de desarrollo con su propio marco conceptual que involucra terminología, métodos, modelos, procedimientos, técnicas, prácticas y procesos.

Los métodos de la Orientación a Objetos han sido reconocidos en el ámbito de las tecnologías de la información, como la mejor filosofía para abordar la reutilización y la extensibilidad del software.

En una primera aproximación se puede decir que el término *orientado a objetos* significa que el software se organiza como una colección de objetos discretos que contienen tanto estructuras de datos como un comportamiento. Esto se opone frontalmente a la programación convencional, donde las estructuras de datos y la funcionalidad sólo se relacionan débilmente.

Tradicionalmente se asocia la Orientación a Objetos con la Programación Orientada a Objetos, es decir centrando la atención en los lenguajes de programación. Cierto es que los lenguajes de programación orientados a objetos son útiles para eliminar algunas

restricciones propias de los lenguajes de programación tradicionales. Sin embargo, enfatizar la fase de codificación supone un retroceso de la Ingeniería del Software al priorizar los mecanismos de implementación frente al proceso de pensamiento subyacente al cual sirven de base [Rumbaugh et al., 1991]. Así, este paradigma se extiende a todas las fases del ciclo de vida con un modelo común subyacente a todas estas fases: *el modelo objeto*.

Se entenderá, entonces, por *desarrollo orientado a objetos* la forma de pensar acerca del software basándose en abstracciones del mundo real; donde la palabra desarrollo hace alusión directa al bloque inicial de fases del ciclo de vida del software: *análisis, diseño* e *implementación.*

Aparecen así los conceptos de **Programación Orientada a Objetos** (POO)**, Diseño Orientado a Objetos** (DOO) y **Análisis Orientado a Objetos** (AOO)**.**

> **"La programación orientada a objetos es un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son, todas ellas, miembros de una jerarquía de clases unidas mediante relaciones de herencia"**
>
> **Grady Booch, [Booch, 1994]**

Cabe destacar tres partes importantes en esta definición:

1. La POO utiliza *objetos*, no algoritmos, como sus bloques lógicos de construcción fundamentales.
2. Cada objeto es una *instancia* de alguna *clase*.
3. Las clases están relacionadas con otras clases por medio de relaciones de *herencia*.

Mientras que los métodos de programación ponen el énfasis en el uso correcto y efectivo de mecanismos particulares del lenguaje de programación que se utiliza, los métodos de diseño enfatizan la estructuración correcta y efectiva de un sistema complejo, definiéndose como sigue.

> **"El diseño orientado a objetos es un método de diseño que abarca el proceso de descomposición orientada a objetos y una notación para descubrir los modelos lógico y físico, así como los modelos estático y dinámico del sistema que se diseña"**
>
> **Grady Booch, [Booch, 1994]**

Se destacan dos aspectos de esta definición:

1. El DOO da lugar a una descomposición orientada a objetos.
2. Utiliza diversas notaciones para expresar diferentes modelos del diseño lógico (*estructura de clases y objetos*) y físico (*arquitectura de módulos y*

procesos) de un sistema, además de aspectos estáticos y dinámicos del sistema.

El modelo de objetos ha influido en las fases iniciales del ciclo de vida del desarrollo del software. El análisis orientado a objetos enfatiza la construcción de modelos del mundo real, utilizando una visión del mundo orientada a objetos, pudiéndose definir como:

> **"El análisis orientado a objetos es un método de análisis que examina los requisitos desde la perspectiva de las clases y los objetos que se encuentran en el vocabulario del dominio del problema"**
>
> **Grady Booch, [Booch, 1994]**

Toda la Orientación a Objetos gira en torno a dos conceptos fundamentales: *el objeto* y *la clase*. Ambos son dos conceptos estrechamente relacionados, pero que no deben confundirse nunca [Holland et al., 1997]. Para terminar esta introducción se van a presentar algunas de las definiciones que, tanto de objeto como de clase, pueden encontrarse en la bibliografía especializada.

Un objeto modela una parte de la realidad. Con el concepto de objeto, se modela la permanencia e identidad de conceptos percibidos. Así un objeto puede definirse como:

> **"Un objeto representa un elemento, unidad o entidad individual e identificable, ya sea real o abstracta, con un papel bien definido en el dominio del problema"**
>
> **[Smith and Tockey, 1988]**

> **"Una colección de operaciones que comparten un estado"**
>
> **Peter Wegner, [Wegner, 1990]**

> **"Concepto, abstracción, o cosa con frontera y significado débil, perteneciente al problema que se trata; instancia de una clase"**
>
> **[Rumbaugh et al., 1991]**

> **"Un objeto es un encapsulado de un conjunto de operaciones o métodos que pueden ser invocados externamente y de un estado que puede recordar los efectos de los métodos"**
>
> **[Blair et al., 1991]**

> **"Entidad conceptual que es identificable, tiene características que comporten un estado interno y tiene unas operaciones que pueden cambiar el estado del sistema local, y que también pueden solicitar operaciones de objetos relacionados"**
>
> **[Champeaux et al., 1993]**

> **"Un objeto tiene estado, comportamiento e identidad; la estructura y el comportamiento de objetos similares están definidos en su clase común; los términos instancia y objeto son intercambiables"**
>
> **Grady Booch, [Booch, 1994]**

> **"Un concepto, abstracción o cosa que puede ser individualmente identificada y tiene significado en una aplicación. Un objeto es una instancia de una clase"**
>
> **[Blaha and Premerlani, 1998]**

> **"Una entidad delimitada precisamente y con identidad, que encapsula estado y comportamiento. El estado es representado por sus atributos y relaciones, el comportamiento es representado por sus operaciones, métodos y máquinas de estados. Un objeto es una instancia de una clase"**
>
> **[OMG, 1999]**

Por su parte las clases sirven como plantillas para la creación de objetos, especificando un comportamiento a todas sus instancias. Se puede definir como:

> **"Una clase es una plantilla desde la que sus objetos pueden ser creados. Contiene la definición de los descriptores de estado y los métodos de los objetos"**
>
> **[Blair et al., 1991]**

> **"Es un conjunto de objetos que comparten una estructura común y un comportamiento común"**
>
> **Grady Booch, [Booch, 1994]**

> **"Descripción abstracta de los datos y del comportamiento de una colección de objetos similares"**
>
> **Timothy Budd, [Budd, 1991]**

> **"Descripción de un grupo de objetos con propiedades similares, comportamientos comunes, interrelaciones comunes y semántica común"**
>
> **[Rumbaugh et al., 1991]**

> **"Una clase es un tipo abstracto de datos equipado con una posible implementación"**
>
> **Bertrand Meyer, [Meyer, 1997]**

La Orientación a Objeto se distribuye en las dos asignaturas objeto de este proyecto docente de la siguiente forma: el AOO entrará a formar parte del temario de la asignatura de ***Ingeniería del Software***, mientras que la parte de POO y especialmente el DOO serán el cometido de la asignatura de la ***Programación Orientada a Objetos***.

4.3.2 Marco histórico de la Orientación a Objetos

Los pasos por los que ha pasado la evolución histórica de la Orientación a Objeto han sido similares a los de otros métodos de desarrollo; esto es, en primer lugar el énfasis estuvo centrado en las técnicas de programación, para posteriormente ir centrando la atención en las fases de desarrollo de mayor nivel de abstracción, diseño y análisis, y actualmente en los procesos de negocio [Pancake, 1995].

Lo que no ha sido tan normal ha sido su evolución temporal, pues tras los primeros pasos dados a finales de la década de los sesenta, sufrió una parada significativa en su evolución hasta la segunda mitad de la década de los ochenta (como dato significativo, entre junio de 1978 y diciembre de 1985, sólo aparecen nueve artículos – de unos cuatrocientos – en ACM SIGPLAN Notices, mencionando la tecnología orientada a objetos de una forma significativa [Sharp et al., 2000]), que resurge con tremenda fuerza, especialmente a partir de la primera conferencia OOPSLA en 1986, dando lugar a una gran diversidad, que podría calificarse de caótica en algunas parcelas (como por ejemplo los métodos de análisis y diseño). En los últimos años de la década de los noventa se ha producido una cierta madurez en la tecnología de objetos, que camina hacia la unificación y los primeros estándares.

El repaso que se va a hacer por la historia de la Orientación a Objetos se ha divido en tres apartados. En el primero de ellos se va a buscar el origen del concepto de objeto. En el segundo se va a sintetizar la historia de los lenguajes de programación orientados a objetos. Por último, en el tercero se va a presentar la evolución de los métodos de desarrollo en la tecnología de objetos.

4.3.2.1 Origen del concepto de objeto

El término *objeto* surge de forma independiente en varios campos de la informática, casi simultáneamente a principios de los setenta, para referirse a nociones que eran diferentes en su apariencia pero relacionadas entre sí. Todas estas nociones se inventaron para manejar la complejidad de los sistemas software de tal forma que los objetos representaban componentes de un sistema descompuesto modularmente o bien unidades modulares de representación del conocimiento [Yonezawa and Tokoro, 1987].

Levy añade que los siguientes acontecimientos contribuyeron a la evolución de los conceptos orientados a objetos [Levy, 1984]:

- *Avances en la arquitectura de los computadores, incluyendo los sistemas de capacidades y el apoyo en hardware para conceptos de sistemas operativos.*
- *Avances en los lenguajes de programación, como se demostró en Simula, Smalltalk, CLU y Ada.*
- *Avances en metodología de la programación, incluyendo la modularización y la ocultación de la información.*

A esta lista de contribuciones podrían añadirse las tres siguientes [Booch, 1994]:

- *Avances en los modelos de bases datos.*

- *Investigación en Inteligencia Artificial.*
- *Avances en Filosofía y Ciencia Cognitiva.*

El concepto de un objeto tuvo sus inicios en el hardware con la aparición de arquitecturas basadas en descriptores y, posteriormente, arquitecturas basadas en capacidades [Ramamoorthy and Sheu, 1988]. Estas arquitecturas representaron una ruptura con las arquitecturas clásicas de **Von Neumann**, surgiendo como consecuencia de los intentos realizados para eliminar el hueco existente entre las abstracciones de alto nivel del los lenguajes de programación y las abstracciones de bajo nivel de la propia máquina. Según sus inventores estas arquitecturas ofrecen ventajas del tipo: *mejor detección de errores, mejora de la eficiencia de ejecución, menos tipos de instrucciones, compilación más sencilla* y *reducción de los requisitos de almacenamiento*. Algunos computadores con arquitectura orientada a objetos fueron el Burroughs 5000, el Plessey 250, el Intel 432, el IBM System/38 o el Rational R10000.

Muy relacionados con las arquitecturas orientadas a objetos están los sistemas operativos orientados a objetos. El trabajo de **Dijkstra** con el sistema de multiprogramación THE fue el primero que introdujo la idea de construir sistemas como máquinas de estados en capas [Dijkstra, 1968]. Otros sistemas operativos pioneros en la tecnología de objetos fueron UCLA Secure UNÍS (para el PDP 11/45 y 11/70), StarOS y Medusa (para Cm* de CMU) o el iMAX (para el Intel 432). Sistemas operativos actuales, como Microsoft Windows NT, parecen seguir el camino de los objetos.

Sin duda alguna la contribución más importante al modelo de objetos estriba en los denominados lenguajes de programación basados en objetos y orientados a objetos. Las ideas fundamentales de clase y objeto aparecieron por primera vez en el lenguaje Simula 67.

En 1972 **David Parnas** introduce la idea de ocultación de la información [Parnas, 1972], y en la década de los setenta varios investigadores, destacando **Liskov y Zilles** [Liskov and Zilles, 1977], **Guttag** [Guttag, 1980] y **Mary Shaw** [Shaw, 1984], fueron pioneros en el desarrollo de mecanismos de tipos de datos abstractos. **Hoare** contribuyó a esos desarrollos con su propuesta de una teoría de tipos y subtipos.

Aunque la tecnología de Bases de Datos ha evolucionado un tanto independientemente de la Ingeniería del Software, también ha contribuido al modelo de objetos [Atkinson and Buneman, 1987], especialmente mediante las ideas de la aproximación entidad-relación al modelado de datos [Rumbaugh, 1988]. En el modelo entidad-relación, propuesto inicialmente por **Peter Chen** [Chen, 1976], el mundo se modela en términos de sus entidades, los atributos de éstas y las relaciones entre esas entidades.

En el campo de la Inteligencia Artificial, los avances en representación del conocimiento han contribuido a una comprensión de las abstracciones orientadas a objetos. En 1975, **M. Minsky** propuso por primera vez una teoría de marcos para

representar objetos del mundo real tal y como los perciben los sistemas de reconocimiento de imágenes y el lenguaje natural [Barr and Feigenbaum, 1981]. Desde entonces se han utilizado los marcos como fundamento arquitectónico para diversos sistemas inteligentes.

Por último la Filosofía y la Ciencia Cognitiva han contribuido al avance del modelo de objetos. La idea de que el mundo podía verse en términos de objetos o procesos procede de los filósofos griegos, más concretamente de la Teoría de las Ideas desarrollada por **Platón**, a partir de las enseñanzas de **Sócrates**, y estudiada y ampliada por **Aristóteles** (en la tabla se tiene una equivalencia entre los conceptos de la Teoría de las Ideas y la Orientación a Objetos, para una mayor información consultar [Alhir, 1998]). Así se puede decir que la Orientación a Objetos se acerca más al enfoque Aristotélico-Tomista[20], frente al enfoque Kantiano de los métodos estructurados. Asimismo, en el siglo XVII se tiene a **Descartes** defendiendo que los seres humanos aplican de forma natural una visión orientada a objetos del mundo [Stillings et al., 1987]. Ya en el siglo XX, **Rand** amplía estos conceptos en su filosofía de la epistemología objetivista [Rand, 1979].

TEORÍA DE LAS IDEAS	PARADIGMA OBJETUAL
Método Socrático (la recolección)	Abstracción (*como forma de obtener conocimiento*)
Ideas, universalidad y sustancias secundarias	Clases (*y encapsulación*)
Cosas, particulares y sustancias primarias	Objetos (*y encapsulación)*
Procedimiento de captura y división y el principio de uno sobre muchos	Herencia (*y polimorfismo*)
El argumento del tercer hombre	Abstracción (*como marco conceptual para el modelado que involucra múltiples niveles de abstracción*)

Tabla 4.9. Correspondencia entre la Teoría de las Ideas y la Orientación a Objeto

4.3.2.2 Historia de los lenguajes de programación orientados a objetos

Los lenguajes de programación de alto nivel pueden agruparse en cuatro generaciones [Booch, 1994]; según soporten abstracciones matemáticas, algorítmicas, de datos u orientados a objetos. Un lenguaje se considera *basado en objetos* si soporta directamente abstracción de datos y clases, mientras que un lenguaje *orientado a objetos* es aquél que, además de estar basado en objetos, proporciona soporte para la herencia y el polimorfismo.

[20] En la realidad no existen datos o procesos independientes. Santo Tomás de Aquino sostiene que el número, considerado abstractamente, no existe fuera de la mente humana; "*la verdad consiste en la adecuación del entendimiento con las cosas*" (Suma Teológica. I, c.16, a.3.).

Aristóteles indica que "*no se pueden separar, a los objetos en movimiento, por ejemplo*" puesto que "*hay una multitud de accidentes que son esenciales a las cosas, en tanto que cada uno de ellos reside esencialmente en ellas*" (Metafísica. 7ª edición, Madrid, Espasa-Calpe, 1972, p.284.).

El primer lenguaje orientado a objetos data de finales de la década de los sesenta fue **Simula 67** [Dahl et al., 1970], [Dahl and Hoare, 1972], [Birtwistle et al., 1973]. Este lenguaje fue diseñado en 1967 por **Ole-Johan Dhal** y **Kristen Nygaard** en el *Norwegian Computing Center* en Oslo. Ellos partieron de un lenguaje denominado Simula 1 [Dahl and Nygaard, 1966], que fue desarrollado en 1962 para la realización de simulaciones discretas; siendo un conjunto de procedimientos más un preprocesador para el lenguaje **ALGOL 60**. El lenguaje Simula al que se le atribuye el *honor* de ser el primer lenguaje orientado a objetos es el Simula 67. El nombre que se le dio intentaba no romper la continuidad con el primer lenguaje, manteniendo así el enlace con la comunidad de usuarios existentes, pero, como el propio **Nygaard** reconoce, fue un error porque aunque Simula 67 era un lenguaje de carácter general, todo el mundo lo asociaba a un lenguaje para llevar a cabo simulaciones de eventos discretos solamente. El nombre de Simula 67 fue acortado a Simula en 1986, existiendo un estándar del lenguaje desde 1987 [SIS, 1987]. La historia de Simula contada por sus creadores se encuentra en [Nygaard and Dahl, 1981].

Uno de los descendientes directos de Simula fue **Beta** [Madsen et al., 1993], diseñado en Escandinavia con la colaboración de **Kristen Nygaard**. Introduce una construcción denominada *patrón* para unificar los conceptos de clase, procedimiento, función, tipo y corrutina.

Uno de los lenguajes de programación que adoptó los conceptos de clase y mensaje propios de Simula fue **Smalltalk**. Se desarrolla a principios de la década de los setenta en el *Xerox PARC* (*Palo Alto Research Center*) como fruto de una labor sinérgica de un grupo de trabajo, en el que aparecen nombres tan importantes como **Alan Kay, Adele Goldberg** o **Daniel Ingalls** como principales responsables. Smalltalk representa tanto un lenguaje de programación como un entorno de desarrollo de software. Se le considera un lenguaje orientado a objetos puro, en el sentido de que todo en él se ve como un objeto (*desde las propias clases hasta los tipos de datos básicos de otros lenguajes, como puede ser el tipo entero, son objetos*).

Los conceptos de Smalltalk no sólo han influido decisivamente en el diseño de posteriores lenguajes de programación orientados a objetos, sino también en el aspecto y sensación de interfaces gráficas de usuario como las de **Macintosh**, **Windows** o **Motif**, o en la definición de una de las arquitecturas básicas de los entornos gráficos de usuario, la ***arquitectura Modelo-Vista-Controlador (MVC)*** [Krasner and Pope, 1988]**.**

Como lenguaje combina la influencia de Simula con un estilo libre, sin ataduras de tipos, propio de **Lisp**. Hace un gran énfasis en la ligadura dinámica, no haciendo ningún chequeo de tipos.

Hay cinco versiones identificables de Smalltalk, referenciadas por su año de aparición: Smalltalk-72 [Goldberg and Kay, 1976], Smalltalk-74, Smalltalk-76 [Ingalls, 1978], Smalltalk-78 y Smalltalk-80 [Goldberg and Robson, 1983]. Las dos primeras sentaron gran parte de los fundamentos del lenguaje, incluyendo las ideas de paso de

mensajes y polimorfismo. Las versiones posteriores convirtieron a las clases en *ciudadanos de primera clase*, completando la visión de que todo lo que hay en el entorno puede tratarse como un objeto. Hay un importante dialecto de Smalltalk proporcionado por Digital, Smalltalk/V, muy parecido a Smalltalk y disponible para máquinas IBM PC (entornos Windows y OS/2) y Macintosh.

Las referencias por excelencia de Smalltalk son [Goldberg and Robson, 1983], [Goldberg, 1985], [Lalonde and Pugh, 1990] y [Lalonde and Pugh, 1990].

No todos los lenguajes orientados a objetos fueron creados desde cero, sino que muchos han visto la luz como evolución de otro ya existente. Quizás los ejemplos más destacados sean **Object Pascal, Objective-C, C++** y **Ada 95**.

Object Pascal fue diseñado por desarrolladores de **Apple Computer** (algunos de los cuales estuvieron implicados en el desarrollo de Smalltlak), en conjunción con **Niklaus Wirth**, el padre de Pascal. El antecesor inmediato de Object Pascal fue *Clascal*, una versión orientada a objetos del Pascal para el computador *Lisa*. Object Pascal se puso a disposición del público en 1986 y fue el primer lenguaje de programación orientado a objetos soportado por el Macintosh Programmer's Workshop (MPW), el entorno de desarrollo para la familia Apple de computadores Macintosh.

Object Pascal es el esqueleto de un lenguaje orientado a objetos. No proporciona métodos de clase, variables de clase, herencia múltiple ni metaclases. Conceptos que se excluyen de forma deliberada en un intento de suavizar la curva de aprendizaje de los que se aproximan por primera vez a la programación orientada a objeto [Schmucker, 1986].

La referencia principal para Object Pascal es [Apple, 1989]. En la actualidad, la orientación a objetos en derivados del Pascal sigue viva gracias a los esfuerzos de Borland (ahora Inprise) por incorporar las extensiones de objetos a su Turbo Pascal (versiones para MS-DOS y MS - Windows) y al desarrollar posteriormente Delphi (para entornos Windows), cuya versión actual es la 5.0.

Objective-C [Cox, 1984], [Cox and Novobilski, 1990] fue diseñado por **Brad J. Cox** en Stepstone Corporation. Es una definición ortogonal a Smalltalk sobre la base conceptual del lenguaje C. Fue el lenguaje de programación fundamental para el sistema operativo y las estaciones de trabajo NEXTSTEP. Aunque en parte relegado a un segundo plano por el éxito de C++, este lenguaje aún retiene un importante número de usuarios activos.

Como sucede en Smalltalk, Objective-C, pone un énfasis especial en el polimorfismo y en la ligadura dinámica, aunque las versiones actuales han dado un salto atrás con respecto al modelo original de Smalltalk para ofrecer tipos estáticos como una opción (y para algunos de ellos también ligadura estática).

C++ fue desarrollado por **Bjarne Stroustrup** en los AT&T Bell Laboratories a principios de la década de los ochenta. En el Cuadro 4.4 se recogen las fechas más significativas en la evolución de C++.

Figura 4.5. Bjarne Stroustrup

Es un lenguaje híbrido, con comprobación estricta de tipos, en el cual algunas entidades son objetos y otras no. Es una extensión de C, que además de añadir capacidades orientadas a objetos, sirve para mejorar algunas deficiencias del lenguaje C.

1979	Mayo	Comienza el trabajo con C con Clases
	Octubre	La primera implementación de C con Clases en uso
1980	Abril	Primer artículo interno para Bell Labs sobre C con Clases
1982	Enero	Primer artículo externo sobre C con Clases
1983	Agosto	Primera implementación de C++ en uso
	Diciembre	Se le da el nombre de C++
1984	Enero	Primer manual de C++
1985	Febrero	Primera versión externa de C++ (Release E)
	Octubre	Cfront Release 1.0 (primera versión comercial)
	Octubre	The C++ Programming Language [Stroustrup, 1986]
1986	Agosto	The "*whatis paper*" [Stroustrup, 1986b]
	Septiembre	Primera conferencia OOPSLA
	Noviembre	Primer *port* comercial a PC de Cfront (Cfront 1.1, Glockenspiel)
1987	Febrero	Cfront Release 1.2
	Noviembre	Primera conferencia USENIX (Santa Fe, Nuevo Mexico)
	Diciembre	Primera versión de GNU C++ (1.13)
1988	Enero	Primera versión de Oregon Software C++
	Junio	Primera versión de Zortech C++
	Octubre	Primer *workshop* USENIX de desarrolladores en C++
1989	Junio	Cfront release 2.0
	Diciembre	Reunión organizativa ANSI X3J16 (Washington, DC)
1990	Mayo	Primera versión de Borland C++
	Mayo	Primera reunión técnica ANSI X3J16 (Somerset, NJ)
	Mayo	The Annotated C++ Reference Manual [Ellis and Stroustrup, 1990]
	Julio	Plantillas aceptadas (Seattle, WA)
	Noviembre	Excepciones aceptadas (Palo Alto, CA)
1991	Junio	The C++ Programming Language 2nd [Stroustrup, 1991]
	Junio	Primera reunión ISO WG21 (Lund, Sweden)
	Octubre	Cfront release 3.0 (incluye plantillas)
1992	Febrero	Primera versión del DEC C++ (incluye plantillas y excepciones)
	Marzo	Primera versión de Microsoft C++
	Mayo	Primera versión de IBM C++ (incluye plantillas y excepciones)
1993	Marzo	RTTI aceptado (Munich, Alemania)
	Julio	Espacios de nombres aceptado (Munich, Alemania)
1994	Agosto	Registro del *draft* del comité ANSI/ISO
1997	Noviembre	Aprobado el estándar internacional del lenguaje de programación C++

Cuadro 4.4. Fechas relevantes en la evolución de C++

El antecesor inmediato de C++ fue el denominado *C con Clases*, desarrollado por el propio Stroustrup en 1980, recibiendo enormes influencias de los lenguajes Simula y C.

Han existido varias versiones principales del lenguaje C++. La versión 1.0 (y sus versiones menores) añadían características básicas de orientación a objetos al C, como la herencia simple y el polimorfismo, además de la comprobación de tipos y sobrecarga. La versión 2.0, que aparece en 1989, mejoró las versiones anteriores de diversas formas, como la introducción de la herencia múltiple, sobre la base de una amplia experiencia con el lenguaje por parte de una comunidad de usuarios relativamente grande. La versión 3.0, aparecida en 1991, introduce las plantillas (*manejo de clases parametrizadas*) y el manejo de excepciones. Las últimas aportaciones al estándar de C++ han venido de la mano de la incorporación de espacios de nombres e identificación de tipos en tiempo de ejecución. En noviembre de 1997 se aprobó el estándar internacional de C++, que queda recogido en [ISO/IEC, 1998], teniéndose acceso libre a algunos de los borradores de los informes técnicos que se han ido elaborando, por ejemplo el de diciembre de 1996 en [X3J16/WG21, 1996].

En [Stroustrup, 1994] se recogen de mano del padre de C++ las decisiones de diseño tomadas para el desarrollo del lenguaje.

La tecnología inicial de traductores para C++ implicaba el uso de un preprocesador para C, llamado *cfront*. Puesto que este traductor generaba código C como representación intermedia, era posible transportar C++ a prácticamente todas las arquitecturas UNIX con bastante sencillez. Ahora, están disponibles los traductores de C++ y compiladores nativos para los conjuntos de instrucciones de casi todas las arquitecturas.

La bibliografía de referencia sobre C++ es muy abundante; de los numerosos títulos existentes se van a destacar dos [Ellis and Stroustrup, 1990] y [Stroustrup, 1997][21]. Como punto discordante se recomienda la lectura de [Joyner, 1996], donde se hace un recorrido crítico de C++ como lenguaje de programación orientado a objetos, comparándolo con otros lenguajes orientados a objetos.

Eiffel [Meyer, 1992] es un lenguaje orientado a objetos diseñado por **Bertrand Meyer**. Fue ideado no sólo como un lenguaje de programación, sino también como una herramienta de Ingeniería del Software. Aunque se diseñó desde cero, recibe influencias de Simula.

El lenguaje soporta ligadura dinámica y comprobación estática de tipos, ofreciendo flexibilidad en el diseño de la interfaz de una clase, pero aprovechando la seguridad respecto al tipo que proporciona la comprobación estática de tipos. Hay varias características significativas que apoyan el desarrollo de software seguro y de calidad, incluyendo clases parametrizadas, aserciones y excepciones.

[21] Esta edición se ajusta al estándar aprobado.

Como otros lenguajes anteriores a la Orientación a Objetos, Lisp ha servido como influencia para muchos lenguajes de programación orientados a objetos, lo cual no es de extrañar porque Lisp ofrece muchos mecanismos que ayudan a la implementación de los conceptos de la Orientación a Objetos: *una aproximación dinámica para la creación de objetos, gestión automática de memoria con "recolección de basura", fácil implementación de estructuras arbóreas, selección de operaciones en tiempo de ejecución...*

En la década de los ochenta hubo tres frentes que coparon la atención de la orientación a objeto en torno a Lisp: **Loops** [Bobrow and Stefik, 1982], desarrollado por **Xerox**, inicialmente para el entorno *Interlisp*; **Flavors** [Cannon, 1980], desarrollado en el **MIT**, disponible en varias arquitecturas orientadas a Lisp; **Ceyx** [Hullot, 1984], desarrollado en el **INRIA**.

Loops introdujo un concepto interesante: *la programación orientada a los datos*; de forma que se podía asociar una rutina a cada elemento de datos (por ejemplo a un atributo). La ejecución de la rutina era disparada no sólo por la llamada explícita, sino también cuando el elemento era accedido o modificado. Esto abría la puerta a la computación conducida por eventos y a arquitecturas software más descentralizadas.

La unificación de estas propuestas vino de la mano de **CLOS** (*Common Lisp Object Oriented System*) [Paepcke, 1993], una extensión orientada a objetos de Common Lisp, convirtiéndose en el primer lenguaje orientado a objetos en contar con un estándar ANSI.

Aunque fue implementado de forma híbrida en un principio, CLOS está tan bien integrado con las características de Common Lisp que tiene la mayoría de las ventajas de los lenguajes de programación orientados a objetos puros. Esto se debe a que todos los objetos de datos, incluyendo los átomos y las listas de Lisp, son miembros de una clase. Los métodos correspondientes a primitivas de Lisp pertenecen a una estructura de herencia. Como resultado, no hay una distinción práctica entre primitivas de Lisp y objetos.

CLOS ofrece una rica colección de metadatos a los cuales se puede acceder y pueden ser actualizados en tiempo de ejecución. Se pueden definir clases nuevas, y se pueden añadir dinámicamente métodos a las clases.

El sistema del lenguaje CLOS no impone el concepto de encapsulación. Se recomienda a los programadores que definan y documenten la interfaz pública de todas las clases, y que utilicen únicamente las características indicadas de las otras clases, pero nada impide que el código de una clase acceda directamente a los detalles de implementación de otra clase. Esta falta de aplicación de las convenciones concuerda con la política general de Lisp, consistente en ofrecer la mayor flexibilidad y el mayor espacio posible para la experimentación.

La última revolución dentro del seno de los lenguajes de programación orientados a objetos ha sido **Java**. Desarrollado por un equipo de **Sun Microsystems**, este lenguaje ha provocado ríos de tinta desde su aparición en escena a finales de 1995, especialmente por su íntima relación con Internet.

Los orígenes de Java como lenguaje de programación se deben más al azar que al fin para el cual se utiliza hoy en día [Mohedano, 1998]. En 1990, Sun organiza un equipo de seis personas (entre las que se encontraban **James Gosling, Bill Joy** y **Patrick Naughton**), al que se denominó *Green*, para que desarrollasen algo innovador. Este grupo centra su atención en los dispositivos electrónicos de gran consumo, buscando una forma de comunicarlos y que pudieran ejecutar programas simples suministrados a través de una red. Dadas las características intrínsecas de estos aparatos, que se resumen en heterogeneidad y limitaciones de memoria, se dedican a diseñar un nuevo lenguaje de programación orientado a objetos que James Gosling denominó **Oak**. Este lenguaje estuvo listo en el verano de 1991, justo el mismo verano en que el servicio WWW iniciaba su expansión por Internet. En octubre de 1991 el grupo había crecido y Sun decidió crear una compañía autónoma que se denominó *First Person*, con las vistas puestas en la televisión interactiva, pero no cuajó el intento. Durante 1993 y parte de 1994 el grupo desarrolló diferentes proyectos en Oak, a la vez que buscaba a quién venderle esta tecnología, no consiguiendo ningún contrato. A finales de 1994 el equipo se disuelve, aunque algunas personas del grupo prosiguen en el intento de aplicar la tecnología Oak a los sistemas de escritorio basados en red. La difusión que por 1994 experimentaba el mundo web, abrió una puerta de solución a la tecnología desarrollada, ya que la filosofía cliente-servidor se ajustaba perfectamente a lo que se había desarrollado. En junio de 1994 surge la idea de utilizar a Oak como un producto en sí mismo, distribuyéndolo a través de Internet, a la vez que se baraja la idea de crear un sistema operativo basado en el lenguaje, consolidando la independencia de plataforma. Sun apoya en el otoño de 1994 la idea de regalar el lenguaje Oak, pero antes de su lanzamiento se dan cuenta de que ya existe un lenguaje con ese nombre, y se lo cambian por Java, presentándose oficialmente en la revista **SunWorld** en mayo de 1995. A lo largo de 1999 ha aparecido la última revisión de Java, la especificación de *Java 1.2*, también conocida como **Java 2**. En [Naughton, 1996] se cuenta la historia de este lenguaje según uno de sus autores, Patrick Naughton.

Java es un lenguaje que en su sintaxis recuerda mucho a C++ (y por tanto a C), pero sus semejanzas con C++ no van mucho más allá [Joyner, 1999].

Uno de los puntos más interesantes de Java es que no sólo es un lenguaje de programación, es una plataforma que se compone de un *lenguaje de programación* de propósito general orientado a objetos; de *archivos de clase* encargados de almacenar los ***bytecodes***, resultado de compilar los programas java, facilitando la independencia de plataforma y la movilidad entre redes; de una *máquina virtual* que se encarga de ejecutar los programas Java; y una *interfaz de programación de aplicaciones*.

Existe una numerosa bibliografía sobre Java, entre los que dan una visión general de Java como lenguaje de programación se pueden citar [Naughton, 1996], [Arnold and Gosling, 1997], [SUN, 2000]. Una buena referencia de consulta avanzada de Java 1.2 es [Jaworski, 1998].

Existen otros muchos lenguajes de programación orientados a objetos, tales como ***Oberon*** [Wirth and Reiser, 1992] – diseñado por **Niklaus Wirth** como sucesor de Modula-2; ***Modula-3*** [Harbison, 1992] – es un proyecto de Digital, que parte de Modula-2; ***Self*** [Ungar et al., 1992] – no está basado en clases sino en prototipos, teniendo en la herencia una relación entre objetos más que entre tipos; ***Ada 95*** [Ada95-Web] – es el resultado de añadir capacidades de Orientación a Objetos al lenguaje Ada 83; cuyo repaso sobrepasa el cometido de este apartado. Existen diversos fuentes bibliográficas que repasan y comparan diversos lenguajes orientados a objetos, entre ellos cabe destacar el capítulo 15 de [Rumbaugh et al., 1991], el apéndice A de [Booch, 1994], el capítulo 32 de [Meyer, 1997] o [Rans, 1999].

La Figura 4.6 ilustra la evolución de algunos de los principales lenguajes de programación orientados a objetos.

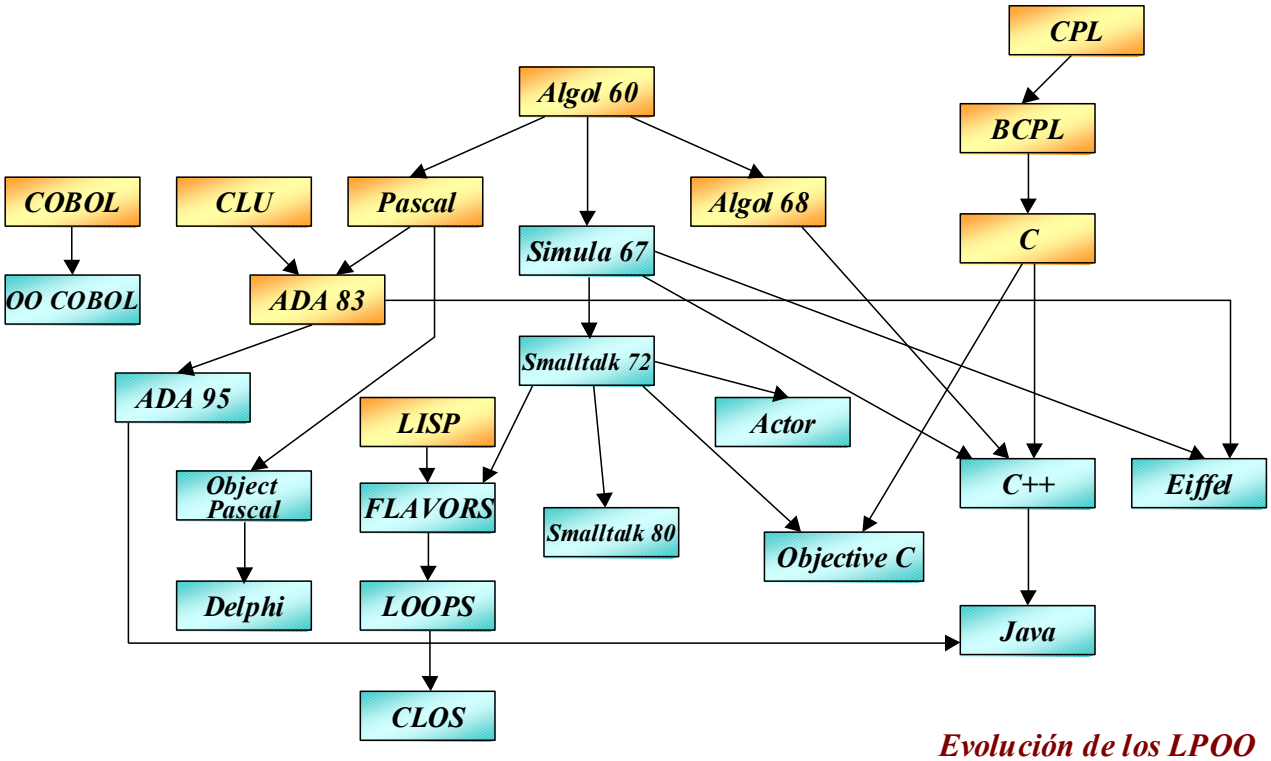

Figura 4.6. Evolución de los lenguajes de programación orientados a objetos

4.3.2.3 Evolución de los métodos orientados a objetos

Al igual que sucede en el paradigma estructurado, la tecnología de objetos debe dar cobertura a todo el ciclo de desarrollo de los sistemas software, es decir, al análisis, al

diseño y a la implementación; aunque se debe resaltar que estas fases se solapan y presentan unas formas más difuminadas que en el desarrollo estructurado.

Cuando a mediados de la década de los ochenta resurge la Orientación a Objetos, comienzan a surgir multitud de propuestas metodológicas con una orientación objetual, sólo en el período comprendido entre 1989 y 1994 el número de lenguajes de modelado orientados a objetos pasa de 10 a más de 50. Precisamente esta diversidad de métodos y notaciones ha sido una de las mayores barreras con que se encontraron las empresas y los departamentos de informática para la adopción de la Orientación a Objetos, conociéndose a este fenómeno como la *guerra de los métodos* [García y Pardo, 1998].

La cantidad de métodos que surgen en esta primera *hornada* se denominan metodologías o métodos de primera generación entre los que cabe destacar **Synthesis** [Bailin, 1989]**, RDD** (*Responsibility Driven Design*) [Wirfs-Brock et al., 1990], **OMT** (*Object Modeling Technique*) [Rumbaugh et al., 1991], **OOD** (*Object Oriented Design*) [Booch, 1991], **OOSE** (*Object-Oriented Software Engineering*) – **Objectory** [Jacobson et al., 1993], o la metodología de **Shlaer y Mellor** [Shlaer and Mellor, 1992].

Hacia la mitad de década de los noventa aparecen en escena las nuevas versiones de los métodos más consolidados, así como nuevas incorporaciones que surgen como una acumulación de las características más destacadas de los métodos tradicionales junto con algunos añadidos: las metodologías o métodos de segunda generación. Pertenecientes a esta generación se tiene, entre otros, a **OMT-2** [Rumbaugh, 1996], **OOram** [Reenskaug et al., 1996], **OOA/D** (*Object-Oriented Análisis and Design*) [Booch, 1994], **Fusion** [Coleman et al., 1994]**, Moses** (Methodology for Object-Oriented Software Engineering of Systems) [Henderson-Sellers and Edwards, 1994a], [Henderson-Sellers and Edwards, 1994b]**, Syntropy** [Cook and Daniels, 1994] o **Medea** [Piattini, 1994].

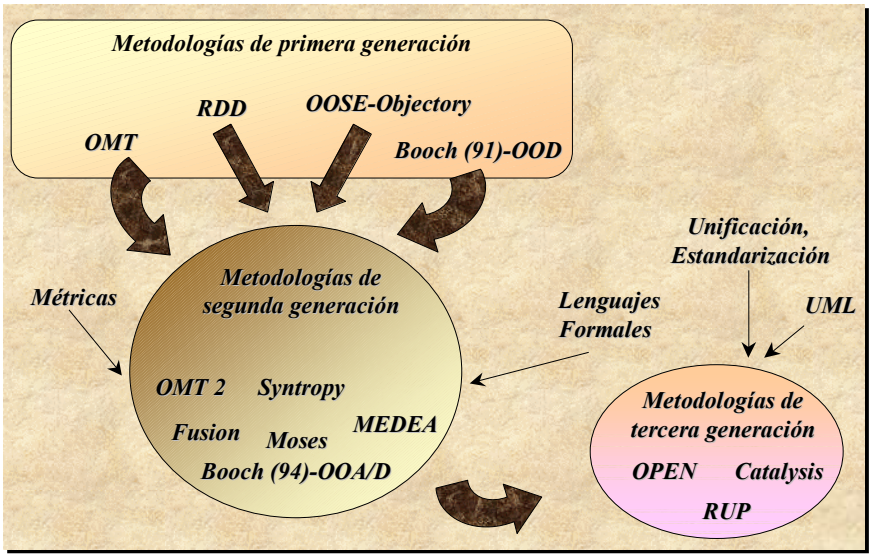

Figura 4.7. Generaciones de las metodologías orientadas a objetos

Ante esta situación de diversidad, en la parte final de la década de los noventa se empiezan a escuchar los primeros rumores sobre la necesidad de una unificación y una estandarización, esto ha dado lugar a la tercera generación de metodologías, de entre las que destacan **RUP** (*Rational Unified Process*) [Jacobson et al., 1999], **OPEN** (*Object-Oriented Process, Environment and Notation*) [Graham et al., 1997], [Henderson-Sellers et al., 1998], [Firesmith et al., 1998] o **Catalysis** [D'Souza and Wills, 1999].

En la Figura 4.7 se presenta un esquema de la evolución sufrida por las metodologías y métodos de diseño orientados al objeto.

El catalizador principal de esta unificación ha sido sin lugar la aparición en escena del lenguaje de modelado **UML** (*Unified Modeling Language*) de ***Rational Software Corporation***, que fue adoptado el 17 de noviembre de 1997 como lenguaje de modelado estándar por el **OMG** (*Object Management Group*) en su versión 1.1 [Rational et al., 1997]. OMG ha propuesto la especificación de UML para su estandarización internacional por parte de **ISO** (*International Organization for Standardization*).

UML se desarrolla en el seno de ***Rational Software Corporation*** con el apoyo de diversos colaboradores a lo largo de su historia, convirtiéndose en el elemento unificador de los lenguajes de modelado de los métodos OOA/D de **Grady Booch** [Booch, 94], OMT-2 de **James Rumbaugh** [Rumbaugh, 1996] y OOSE de **Ivar Jacobson** [Jacobson et al., 1993].

Grady Booch, director científico de Rational Software Corporation desde prácticamente su creación en 1980, empieza a planificar su estrategia a favor de la unificación de métodos. Así, para la comunidad de los métodos orientados al objeto la gran noticia en el OOPSLA'94 fue que James Rumbaugh había abandonado General Electric para unirse a Grady Booch en Rational Software Corporation para fusionar sus métodos. De esta manera el desarrollo de UML comienza en octubre de 1994, cuando Booch y Rumbaugh empiezan a trabajar para unificar los dos métodos que más repercusión habían alcanzado en la escena del ADOO, OOA/D y OMT. Como primer fruto de esta colaboración aparece en octubre de 1995 la primera versión pública de la descripción de la unión de sus métodos. Esta versión se presenta en el OOPSLA'95 con el nombre de Método Unificado versión 0.8 [Booch and Rumbaugh, 1995].

El siguiente paso en el proceso de unificación se produce a finales de 1995 cuando Rational compra a la empresa Objectory, y su fundador, el prestigioso Ivar Jacobson, se une a Rational como vicepresidente de Ingeniería de Negocio para acoplar su metodología OOSE al método unificado de Booch y Rumbaugh. Tras la incorporación de Jacobson a Rational, se les conoce a Booch, Rumbaugh y Jacobson por *los tres amigos*.

El primer paso que se da después de la incorporación de Jacobson es la creación de un lenguaje de modelado unificado debido principalmente a dos razones. En primer lugar el lenguaje de modelado permite que los tres métodos puedan evolucionar hacia

un punto común de forma independiente. Por otro lado, la unificación de la semántica y de la notación les permite colocarse en un lugar de privilegio en el mercado de los métodos orientados al objeto, debe tenerse en cuenta que la mayoría de la gente que dice utilizar un método orientado al objeto realmente lo que usa es la notación asociada a dicho método, olvidándose de los procesos asociados al método, así pues, contar con una notación gráfica universal era fundamental para la unificación de sus métodos, y de hecho la notación de UML se convierte en el estándar de facto de las notaciones en tecnología de objetos antes de su aceptación como estándar oficial.

La primera versión de este lenguaje de modelado aparece en junio de 1996 con el nombre de UML 0.9 [Booch et al., 1996a]. En septiembre de este mismo año aparece la versión 0.91 de UML [Booch et al., 1996b].

Durante 1996 invitan a otros expertos a colaborar con ellos, entre los colaboradores se encuentran nombres muy ilustres y de reconocido prestigio dentro de la comunidad de la Ingeniería del Software y más particularmente de la Orientación al Objeto. Citarlos a todos sería largo, pero como muestra se pueden mencionar a personajes de tanto renombre como *Peter Coad, Derek Coleman, Ward Cunningham, David Ambly, Eric Gamma, David Harel, Richard Helm, Ralph Johnson, Stephen Mellor, Bertrand Meyer, Jim Odell, Kenny Rubin, sally Shlaer, JohnVlissides, Paul Ward, Rebbeca Wirfs-Brock, Edward Yourdon* entre otros.

Además, se crea **OTUG** (*Object Technology Users Group*) como foro de discusión; una lista de correo en la que se discute y se comparten ideas sobre la tecnología de objetos, teniendo como trasfondo a UML.

Tras la salida a la luz de UML la comunidad de ADOO queda divida en dos grupos principales, aquellos que se unen a la estela de UML y aquellos que forman una coalición anti UML. Esta situación la aprovecha OMG para en 1996 crear el **OOAD SIG** (*Object-Oriented Analysis and Design Special Interest Group*) que se encargara de canalizar los esfuerzos para conseguir un estándar. De esta forma se organiza la ***OMG Task Force RFP*** (*Request For Proposal*), es decir una petición de propuestas para la creación de un estándar del lenguaje de modelado para los métodos de ADOO. Esta petición de propuestas sugería un lenguaje de modelado que contara con un metamodelo, una notación, una sintaxis y una semántica.

Rational va a responder a esta petición de propuesta con la versión 1.0 de UML en enero de 1997 [Booch et al., 1997a], [Booch et al, 1997b]. Esta versión está avalada por un conjunto de empresas de mucho prestigio dentro del mundo de la informática: *Digital Equipment Corporation, HP, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI, y Unisys*.

Como era de esperar, la respuesta de Rational no fue la única, OMG recibió en enero de 1997 las respuestas de IBM & ObjecTime [Cook et al., 1997], Platinum Technology [Rivas et al., 1997], Ptech, Taskon & Reich Technologies y Softeam.

Durante los primeros meses de 1997 se hicieron predicciones de todo tipo y para todos los gustos. La notación de UML se había convertido en un estándar de facto apoyado por empresas de especial relevancia en el sector informático, lo cual colocaba a UML en una posición de privilegio, pero su oscura semántica y su metamodelo parecían no estar a la altura de otras propuestas, lo cual hacía peligrar su adopción como estándar por OMG en detrimento del resto de las propuestas. Así todo apuntaba a dos posibles soluciones. La primera de ellas, y quizás la peor, daba como resultado dos estándares: UML de Rational y OML de Platinum Technology. La segunda de ella apuntaba por una solución mixta que contemplase aspectos de todas las propuestas, siendo la que se llevó a cabo; así se unen al equipo liderado por Rational el resto de los equipos que enviaron propuestas, dando lugar a la versión 1.1 de UML [Rational et al., 1997], que fue enviada a OMG el 1 de septiembre de 1997, contribuyendo todos con sus ideas a la generación de una respuesta única.

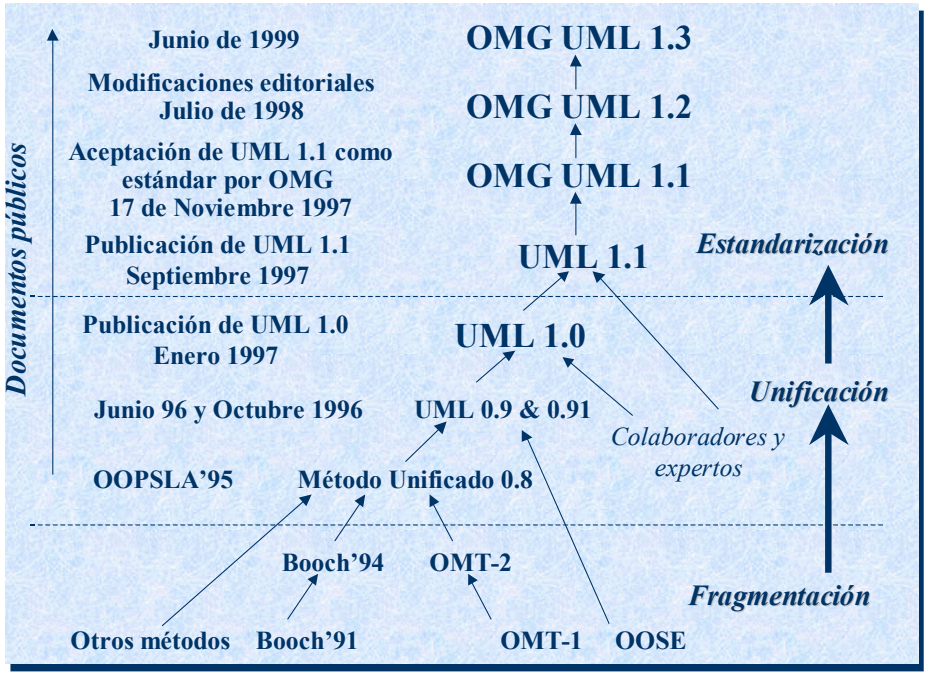

Figura 4.8. Evolución de UML

Esta propuesta es aceptada por OMG como estándar de lenguaje de modelado el 17 de noviembre de 1997, pasándose a denominar al lenguaje de modelado **OMG UML 1.1**. En julio de 1998 aparece la versión **OMG UML 1.2** [OMG, 1998], presentando sólo cambios editoriales. Casi un año más tarde, en junio de 1999 aparece **OMG UML 1.3** [OMG, 1999] con algunos cambios significativos, especialmente en lo tocante a la semántica, siendo ésta la versión en vigor a fecha de hoy[22].

[22] En la actualidad se está trabajando en una serie de cambios menores, que se verán reflejados en una versión 1.4 de UML (UML Revisión Task Force 1.4) que aparecerá a lo largo del año 2000, y en una revisión más profunda que aparecerá como UML 2.0. para el 2001 [Kobryn, 1999]. Una mayor información sobre estos trabajos se puede encontrar en la página web de UML Revison Task Force (http://uml.shl.com).

El camino seguido hasta la versión actual de OMG UML se puede resumir en la Figura 4.8.

Las referencias básicas de UML las constituyen los libros escritos por los autores principales del mismo [Rumbaugh et al., 1999], [Booch et al., 1999] y [Jacobson et al., 1999], donde los dos primeros están dedicados por completo al lenguaje de modelado, mientras que el tercero se dedica al proceso unificado.

4.3.3 Marco conceptual de la Orientación a Objetos

Los sistemas y el pensamiento orientado a objetos se asientan sobre una colección de conceptos que, en su mayor parte, aparecen en diferentes campos y desarrollos de las Ciencias de la Informática con anterioridad a que la Orientación a Objetos fuera un paradigma aceptado como tal.

Desde que los primeros principios de la Orientación a Objeto fueran establecidos en el campo de los lenguajes de programación, fundamentalmente con Simula y Smalltalk, una gran diversidad de campos de la computación han utilizado la tecnología de objetos en el desarrollo de sus teorías; esto ha provocado la existencia de un número de polisemias y sinónimos en relación con el conjunto central de los conceptos inherentes a la Orientación a Objetos. En [Wirfs-Brock and Johnson, 1990] se hace un esfuerzo por presentar los conceptos básicos del paradigma, clarificando su significado con respecto a diferentes interpretaciones.

Para solucionar esta confusión conceptual se establece un *idioma común* que permita la comunicación entre equipos de desarrollo de software con tecnología de objetos. El conjunto de conceptos y propiedades, que configuran este *idioma común* es lo que se conoce como **Modelo Objeto**, y que puede definirse como sigue:

> **"Un Modelo Objeto es un marco de referencia conceptual, en el que se establece el conjunto básico de los conceptos, la terminología asociada y el modelo de computación de los sistemas software soportados por la tecnología orientada al objeto. Este conjunto básico de conceptos deberá incluir como mínimo, los de abstracción, encapsulación, jerarquía y modularidad; y deberá considerar el sistema de información como un conjunto de entidades conceptuales modeladas como objetos e interactuando entre ellas"**
>
> **[Marqués, 1995]**

Desde la perspectiva de la evolución de los modelos objetos, puede considerarse que el primero de ellos, reconocido como tal, es el propuesto por **Anita K. Jones** [Jones, 1987] en el año 1978, definiendo el modelo objeto como un concepto y una herramienta, que proporciona las líneas directrices para caracterizar las entidades abstractas, en los términos en los que se conciben.

A partir de esta primera propuesta formal de modelo objeto, y ante la ausencia de un modelo común de referencia y de estándares universalmente aceptados, para estos

modelos, se ha asistido a una proliferación de propuestas de modelos objetos [Sernadas et al., 1989], [Wand, 1989], [Castellanos et al., 1991], [Bertino and Martino, 1993], [Marqués, 1995] entre otros. Además, se puede constatar que la mayoría de los trabajos en la Orientación a Objetos definen de forma implícita o explícita su propio modelo objeto, como por ejemplo el de UML [OMG, 1999] o el de OPEN [Firesmith et al., 1998].

Según **Grady Booch** [Booch, 1994] todo modelo objeto está formado por cuatro elementos fundamentales: *abstracción, encapsulamiento, modularidad* y *jerarquía*; y por tres elementos secundarios: *tipos, concurrencia* y *persistencia*. "*Fundamentales*" significa que un modelo que carezca de cualquiera de estos elementos no es orientado a objetos. Con "*secundarios*" quiere decirse que cada uno de ellos es una parte útil del modelo objeto, pero no esencial.

La *abstracción* se utiliza para combatir la complejidad. Surge de un reconocimiento de las similitudes entre ciertos objetos, situaciones o procesos del mundo real, y la decisión de concentrarse en esas similitudes e ignorar por el momento las diferencias. Una abstracción se centra en la visión externa de un objeto, sirviendo para separar el comportamiento esencial de un objeto de su implantación.

Se puede definir abstracción como:

> **"Representación de las características esenciales de algo sin incluir antecedentes o detalles irrelevantes"**
>
> **Ian Graham, [Graham, 1994]**

> **"Las características esenciales que distinguen a una entidad de todas las demás entidades. Una abstracción define una delimitación relativa a la perspectiva del observador"**
>
> **[OMG, 1999]**

Mientras que la abstracción ayuda a las personas a pensar sobre lo que están haciendo, el *encapsulamiento* permite que los cambios realizados en los programas sean fiables con el menor esfuerzo. El encapsulamiento genera una cápsula, una barrera conceptual sobre la información y servicios de un objeto, haciendo que estos permanezcan juntos. El encapsulamiento es un medio para conseguir el principio de ocultación de la información [Parnas, 1972].

Se puede definir encapsulamiento como:

> **"Un principio de estado que agrupa datos y procesos permitiendo ocultar a los usuarios de un objeto los aspectos de implementación, ofreciéndoles una interfaz externa mediante la cual poder interaccionar con el objeto"**
>
> **[Piattini et al., 1996]**

> **"Técnica de modelado e implementación que separa los aspectos externos de un objeto de los internos, detalles de implementación de un objeto"**
>
> **[Rumbaugh et al., 1991]**

La *modularidad* permite la fragmentación de un programa en componentes individuales, lo que puede reducir la complejidad en algún grado. Dicha fragmentación crea una serie de fronteras bien definidas y documentadas dentro del programa. Estas interfaces tienen una importancia incalculable para la comprensión del programa.

La modularidad se puede definir como:

> **"La modularidad es la propiedad que tiene un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados"**
>
> **Grady Booch, [Booch, 1994].**

La abstracción es un concepto sumamente útil, pero demasiado extenso, excepto para los casos triviales. El encapsulamiento y la modularidad son medios para manejar las abstracciones. Con frecuencia, un conjunto de abstracciones forma una jerarquía.

La jerarquía se puede definir como:

> **"La jerarquía es una clasificación u ordenación de abstracciones"**
>
> **Grady Booch, [Booch, 1994]**

Las dos jerarquías principales son la jerarquía de clases, formada mediante relaciones de herencia entre las clases, y la jerarquía de partes, formada mediante relaciones de agregación.

Los objetos se comunican a través del *paso de mensajes*. Esto elimina la duplicación de datos, garantizando que no se propaguen los efectos de los cambios en las estructuras de datos encapsuladas dentro del objeto sobre otras partes del sistema. Así, un objeto accede a otro enviándole un mensaje, cuando esto ocurre, el receptor ejecuta el método correspondiente al mensaje. Un mensaje consiste en un nombre de un método más cualquier argumento adicional. El conjunto de mensajes a los que un objeto responde caracteriza su comportamiento.

Se define mensaje como:

> **"Llamada a una operación o a un objeto, en la que se incluye el nombre de la operación y una lista de valores de argumentos"**
>
> **[Rumbaugh et al., 1991]**

> **"Una comunicación entre objetos que transmite información con la expectativa de desatar una acción. La recepción de un mensaje es, normalmente, considerado como un evento"**
>
> **[OMG, 1999]**

El concepto de tipo se deriva de las teorías sobre los tipos abstractos de datos. Se incluye el tipo como elemento separado de modelado de objetos porque el concepto de tipo pone énfasis en el significado de la abstracción en un sentido muy distinto.

Se puede definir *tipo* como:

> **"Los tipos son la puesta en vigor de la clase de los objetos, de modo que los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse sólo de formas muy restringidas"**
>
> **Grady Booch, [Booch, 1994]**

Un concepto muy ligado a la teoría de tipos es el *polimorfismo*, que indica que un solo nombre puede denotar objetos de muchas clases diferentes que se relacionan por una superclase común. Cualquier objeto denotado por este nombre es, por tanto, capaz de responder a algún conjunto de operaciones. Lo opuesto al polimorfismo es el *monomorfismo*, que se encuentra en todos los lenguajes con comprobación estricta de tipos y ligadura estática. Existe el polimorfismo denominado de *inclusión* cuando interactúan las características de la herencia y el enlace dinámico. Es una de las características más potentes de los lenguajes de programación orientados a objetos después de su capacidad para soportar la abstracción, y es lo que distingue a la programación basada en tipos abstractos de datos.

Se puede definir polimorfismo como:

> **"La posibilidad de que una variable o una función adopte diferentes formas en tiempo de ejecución o, más específicamente, a la posibilidad de referirse a instancias de varias clases"**
>
> **Ian Graham, [Graham, 1994]**

Un sistema que implique *concurrencia* puede tener muchos hilos de control (*algunos transitorios y otros permanentes durante toda la ejecución del sistema*). Mientras que la Programación Orientada a Objetos se centra en la abstracción de datos, encapsulamiento y la herencia, la concurrencia se centra en la abstracción de procesos y la sincronización; los objetos unifican los dos puntos de vista anteriores.

Se define la concurrencia como

> **"Concurrencia es la propiedad que distingue un objeto activo de uno que no está activo"**
>
> **Grady Booch, [Booch, 1994]**

Todo objeto software ocupa una cierta cantidad de espacio, y existe durante una cierta cantidad de tiempo. Así pues, se puede definir la *persistencia* como:

> **"La persistencia es la propiedad de un objeto por la que su existencia trasciende el tiempo (es decir, el objeto continúa existiendo después de que su creador deja de existir) y/o el espacio (es decir, la posición del objeto varía con respecto al espacio de direcciones en el que fue creado)"**
>
> **Grady Booch, [Booch, 1994]**

4.3.3.1 La Orientación a Objetos en los cuerpos de conocimiento de la Ingeniería del Software

La Orientación a Objeto es un paradigma de desarrollo, y como tal se contempla dentro de la Ingeniería del Software, por lo tanto está presente en los cuerpos de conocimiento de la Ingeniería del Software.

En el SWEBOK propuesto por IEEE/ACM [Abran et al., 1999], [Abran et al., 2000] los métodos orientados a objetos aparecen en varias áreas de conocimiento:

- **Área de Construcción de Software:** Se realiza una taxonomía de los métodos de construcción de software, de forma que se distinguen métodos de construcción *lingüísticos, matemáticos* y *visuales*, donde cada uno de los cuales se estudia con respecto a cuatro principios (*reducción de la complejidad, anticipación de la diversidad, estructuración de la validación* y *uso de estándares externos*) [Bollinguer, 1999]. La Orientación al Objeto está presente de una forma relevante tanto en los métodos de construcción de software lingüísticos como en los visuales (Tabla 4.10).

	Reducción de la complejidad	**Anticipación de la diversidad**	**Estructuración de la validación**	**Uso de estándares externos**
Lingüísticos	• Plantillas • Patrones de diseño • Bibliotecas de componentes y *frameworks* • Encapsulación y tipos abstractos de datos	• Ocultación de la información • Autodocumen-tación • Conjuntos de métodos completos y suficientes • Herencia • Lenguajes "pegamento" para el uso de componentes	• Modularidad • Guías de estilo	• Lenguajes de programación estándares • Lenguajes de especificación de datos estándares (XML) • Estándares de comunicación entre procesos (COM, CORBA) • Software basado en componentes
Visuales	• Programación orientada a objetos • Programación visual	• Clases • Separación entre la interfaz gráfica y la funcionalidad	• Diseño completo y suficiente de los métodos de las clases	• Lenguajes de programación orientados a objetos estándares • Modelos de interfaces visuales estandarizados

Tabla 4.10. Presencia de la Orientación al Objeto en el área de Construcción de Software del SWEBOK de IEEE/ACM

- **Diseño del Software:** La Orientación a Objeto está presente en todos los temas que aborda esta área de conocimiento (*Conceptos básicos y principios, Calidad de diseño y métricas, Arquitectura software, Notaciones de diseño* y *Estrategias y métodos de diseño*) [Tremblay, 1999], como se recoge en la Tabla 4.11.

	Elementos del paradigma objetual
Conceptos básicos y principios	*Abstracción* *Cohesión y acoplamiento* *Separación de conceptos* *Encapsulación* *Ocultación de la información* *Interfaz vs implementación* *Modularidad* *Refinamiento*
Calidad de diseño y métricas	*Atributos de calidad* (reusabilidad, portabilidad...) *Métricas de diseño orientado a objetos*
Arquitectura software	**Estilos arquitectónicos:** *Abstracción de datos y organización OO* *Sistemas distribuidos (CORBA, DCOM)* **Descripciones arquitectónicas:** *Patrones* ***Frameworks* orientados al objeto**
Notaciones de diseño	**Diseño arquitectónico:** *Diagramas de clase* *CRC* *Diagramas de despliegue y de procesos* **Diseño detallado:** *Diagramas de colaboración* *Diagramas de secuencia* *Diagramas de transición de estados*
Estrategias y métodos de diseño	**Estrategias generales:** *Diseño Orientado a Objetos:* Método de Coad y Yourdon Fusion RUP Diseño por contrato Método de Shlaer y Mellor RDD

Tabla 4.11. Presencia de la Orientación al Objeto en el área de Diseño del Software del SWEBOK de IEEE/ACM

- **Infraestructura de la Ingeniería del Software:** La Orientación a Objetos aparece representada de nuevo en las tres subáreas de esta área de conocimiento: *Métodos de desarrollo, Herramientas software* e *Integración de componentes* [Carrington, 1999] (Tabla 4.12).

<table>
<tr><th>Subárea</th><th colspan="3">Elementos relacionados con la OO</th></tr>
<tr><td>Métodos de desarrollo</td><td colspan="3">Se clasifican en heurísticos, formales y de prototipado. Dentro de los primeros se hace mención explícita de los OO</td></tr>
<tr><td>Herramientas software</td><td colspan="3">Herramientas CASE que soportan el ciclo de desarrollo en la construcción de software OO</td></tr>
<tr><td rowspan="2">Integración de componentes</td><td>Definición de componentes</td><td>Modelos de referencia</td><td>Reutilización</td></tr>
<tr><td>Especificación de la interfaz
Especificación de protocolos
COTS</td><td>Patrones
Frameworks</td><td>Tipos de reutilización
Repositorios</td></tr>
</table>

Tabla 4.12. Presencia de la Orientación al Objeto en el área de Infraestructura de la Ingeniería Software del SWEBOK de IEEE/ACM

- **Área de Análisis de los Requisitos del Software:** Donde se hace mención explícita a las técnicas de Análisis Orientado a Objetos [Sawyer and Kotonya, 1999].
- **Área de Pruebas del Software:** Donde se hace referencia a las técnicas para realizar las pruebas a software construido bajo el paradigma orientado a objeto y software basado en componentes [Bertolino, 1999].

Por lo que respecta al **SWE-BOK** realizado para la **FAA** [Hilburn et al., 1999], también aparece incluida la Orientación a Objetos en diversos apartados como se muestra en la Tabla 4.13.

<table>
<tr><th>Categoría de Conocimiento</th><th>Área de Conocimiento</th><th>Unidad de Conocimiento</th></tr>
<tr><td>Fundamentos de Informática</td><td>Lenguajes de Programación</td><td>Paradigmas de programación
Donde se reconoce a la POO como un paradigma de programación</td></tr>
<tr><td rowspan="6">Ingeniería del Producto Software</td><td rowspan="2">Ingeniería de Requisitos del Software</td><td>Análisis de requisitos
Métodos de modelado OO</td></tr>
<tr><td>Especificación de requisitos
Lenguajes de modelado OO</td></tr>
<tr><td rowspan="2">Diseño del Software</td><td>Diseño arquitectónico
Técnicas de DOO</td></tr>
<tr><td>Especificación abstracta
Notaciones y técnicas de DOO</td></tr>
<tr><td rowspan="2">Codificación del Software</td><td>Implementación del código
POO</td></tr>
<tr><td>Reutilización de código
Reutilización vs OO</td></tr>
</table>

Tabla 4.13. Orientación a Objetos en el SWE-BOK

Por último, el **SE-BOK** propuesto por el **WGSEET** [Bagert et al., 1999] al no describir con tanto detalle los componentes de conocimiento de cada una de las áreas de conocimiento propuestas, no puede establecer a priori la presencia de la Orientación a Objetos en este cuerpo de conocimiento. Sin embargo, como este cuerpo de conocimiento viene acompañado de una propuesta curricular, con la descripción de los cursos, puede verse como la Orientación a Objetos tiene un papel destacado en lo que se denomina el **área central**, especialmente en los componentes de conocimiento *requisitos del software, diseño del software* y *construcción del software*.

4.3.4 La enseñanza de la Programación Orientada a Objetos

La Orientación a Objetos es en sí misma una sub-disciplina de la Ingeniería del Software, que requiere que sus conceptos básicos queden asentados de forma sólida para su correcta aplicación en el desarrollo de sistemas software, donde se quiere obtener ventaja de todo su potencial [D'Souza, 1996].

Para la enseñanza de la Orientación a Objetos se aconseja seguir un modelo de aprendizaje constructivista, más que un modelo de aprendizaje objetivista [Hadjerrouit, 1999]. Esto es así porque en la aproximación objetivista se ve el proceso de aprendizaje como una *transmisión pasiva de conocimientos*, donde no se necesita ningún conocimiento previo; el resultado de este enfoque es que el alumno acaba sin una adecuada comprensión de la base conceptual del paradigma objetual, con malos hábitos de programación y con serios malentendidos sobre la tecnología de objetos. Por su parte la aproximación constructivista presenta el aprendizaje como un *proceso activo de construcción*, donde los alumnos construyen su conocimiento sobre la base de su conocimiento previo, siendo necesario probar lo que se conoce y evaluar si entra en conflicto con los conceptos que están adquiriendo.

Desde un punto de vista constructivo de la Orientación a Objetos, se requieren tres tipos de conocimientos: *los conceptos propios del modelo objeto, un lenguaje de programación orientado a objetos* donde plasmar los conceptos y *supuestos con problemas* de donde obtener el conocimiento específico del dominio [Hadjerrouit, 1999].

El enfoque constructivo, se asemeja a la estructura de los ciclos de vida del desarrollo de software en la tecnología de objetos, ya que es iterativo e incremental. Esto es así, porque el conocimiento y el entendimiento se va adquiriendo, ampliando y madurando en una serie de etapas [Meyer, 1996].

Otro aspecto a tener en cuenta es el momento en el que se introduce la Orientación a Objetos en el currículo del alumno. Hay defensores de introducir la Orientación a Objeto desde el primer momento en la formación de los titulados en Informática [Tewari and Friedman, 1992], [Decker and Hirshfield, 1994], [Adams, 1996],

[Woodman et al., 1996], [Hadjerrouit, 1999]; aunque siendo conscientes que el aprendizaje de los conceptos propios de la tecnología de objetos es difícil para los nuevos alumnos porque requiere una forma de pensar diferente y más profunda en términos de computación [Hadjerrouit, 1999].

Sobre los conocimientos que se requieren para iniciarse en la Orientación a Objetos, hay quien opina que aquéllos que tienen experiencia en el desarrollo de sistemas software, típicamente en el paradigma basado en procedimientos, aprenden fácilmente la sintaxis de los lenguajes orientados a objetos, pero esto no significa que hayan captado la esencia del diseño orientado a objetos y sean capaces de plasmar dichos diseños con los lenguajes aprendidos [Rosson and Carroll, 1996]; de hecho hay estudios que indican que el conocimiento de lenguajes de tipo procedimental puede llegar ser una barrera conceptual para introducirse plenamente en la Orientación a Objetos [Hadjerrouit, 1999].

La forma más extendida para introducir la tecnología de objetos en los currículos de Informática es mediante los cursos de introducción a la programación, donde los lenguajes procedimentales y los métodos de diseño estructurado se verían desplazados por sus homónimos orientados al objeto.

Por supuesto, hay voces que no están de acuerdo con que los conceptos del paradigma de la Orientación a Objetos se introduzcan por la programación, defendiendo que los principios del modelo de objetos para realizar análisis y diseño deben cubrirse con gran detalle antes de llegar a su implementación real en un lenguaje de programación [Northrop, 1992]; o quién no está de acuerdo con que la orientación a objeto se centre sólo en la programación, defendiendo un enfoque más amplio que cubra todo el ciclo vital, con una perspectiva de Ingeniería del Software [Bézivin et al., 1992].

En la Universidad española hay muy pocos casos en los que la Orientación a Objeto aparezca en el primer curso de los planes de estudio de las Ingenierías Técnicas y Superiores en Informática. En su lugar la introducción de la tecnología de objetos se enfoca desde dos perspectivas, complementarias, a saber:

- Inmersos dentro de alguna asignatura de Ingeniería del Software, donde principalmente se estudian las características de los ciclos de vida que pueden seguir los desarrollos de sistemas software bajo el paradigma objetual, así como alguna propuesta metodológica orientada a objetos (*haciendo más hincapié en la parte del lenguaje de modelado, UML típicamente, que en la parte de proceso*).
- Como componentes centrales de asignaturas de programación avanzada, normalmente de carácter optativo, que se centran en técnicas de diseño y programación orientadas a objetos.

El primer enfoque cae dentro de la enseñanza de la Ingeniería del Software [Miller and Mingins, 1998], tal y como se presentó en el apartado 4.2.4 de este capítulo, siendo

una aproximación de carácter más general y abstracto que la que se da en el segundo enfoque, cuando el centro de atención se pone en los modelos del dominio de la solución.

En el contexto de la Universidad de Salamanca se sigue la aproximación más generalizada en España, esto es, en la Ingeniería Técnica en Informática de Sistemas, la Orientación a Objeto se reparte en las dos asignaturas que marcan el perfil de la plaza a concurso: *Ingeniería del Software* y *Programación Orientada a Objetos*.

En la primera de ellas, principalmente, se introducen los conceptos del modelo objeto, para posteriormente hacer una somera introducción del análisis y diseño orientado a objetos a la vez que se presentan los principios básicos de UML. En la segunda de ellas se repasan los conceptos del modelo objeto, pero con una mayor profundidad, haciendo hincapié en las decisiones de diseño y se utiliza un lenguaje de programación como medio (no como fin) para que el alumno asimile el paradigma objetual de una forma constructiva.

La asignatura de Programación Orientada a Objetos se presta a la utilización de los patrones pedagógicos (*reusable pedagogical design patterns*) [Lilly, 1996], [Proto-Patterns, 1999], [Bergin, 1998a], [Bergin, 1998b]. En concreto el proyecto de patrones pedagógicos [Proto-Patterns, 1999] recoge prácticas efectivas de docentes en tecnología de objetos. Estos patrones deben ser fáciles de repetir y de adaptar. Cada patrón debe ser descrito de manera que sea fácilmente instanciable para diferentes *lecciones* y por diferentes *educadores*. Muchos de estos patrones se refieren de forma explícita a temas de tecnologías de objetos, como por ejemplo [Bellin, 1999], [Prieto and Victory, 1999] o [Vaitkevitchius, 1999], aunque otros muchos se pueden aplicar a la docencia de cualquier asignatura, como por ejemplo [Bergin, 1998c], [Jalloul, 1999] o [Manns, 1999]. Los antipatrones son otra herramienta que se puede aplicar en la pedagogía [Dodani, 1999].

4.4 Revisión de los currículos internacionales

Se puede definir currículo como "*un plan para educar estudiantes, ofreciéndoles las características y el conocimiento necesarios para vivir y practicar competentemente una profesión. El currículo debe anticiparse al mundo cambiante en que los alumnos graduados vivirán y trabajarán*" [Denning, 1992].

En este apartado se analizan las propuestas más importantes en currículos internacionales en Informática, realizadas por instituciones de alto prestigio dentro del mundo informático. Para ello se han identificado tres categorías de programas que se estudiarán por separado: *Ciencia de la Informática (CS), Sistemas de Información (IS) o Gestión de Sistemas de Información (MIS)* e *Ingeniería del Software (SE)*.

Cada una de estas áreas presenta su propio enfoque, pero claramente tienen un núcleo de conocimiento común [Parrish et al., 1998].

La revisión que aquí se hace se centra en destacar la presencia de las materias objeto de este Proyecto Docente en las diferentes propuestas curriculares.

4.4.1 Currículos centrados en la Ciencia de la Informática

La educación en *Ciencia de la Informática e Ingeniería* ha sido un área activa durante toda la historia de la disciplina. En particular desde el establecimiento de los primeros Departamentos de Ciencias de la Computación a mediados de la década de los sesenta, se puso una especial atención al reto de educar a los alumnos en un campo tan sumamente cambiante y que evoluciona con tanta rapidez como es la Informática [Tucker et al., 1996].

Las propuestas curriculares más relevantes que aparecen en el campo de la Ciencia de la Informática o Ciencia de la Computación tienen como protagonistas a las dos asociaciones más prestigiosas en el mundo informático, ACM y IEEE-CS. Estas organizaciones publican por separado diferentes propuestas curriculares entre los años 1968 y 1983 (*la evolución de los currículos en la Ciencia de la Informática se detalla en el Cuadro 4.5*), para terminar aunando esfuerzos para definir una propuesta curricular única en el campo de la Ciencia de la Informática e Ingeniería, la propuesta de ACM/IEEE-CS de 1991 (también conocida como *Computing Curricula 1991*) [Tucker et al., 1991a] convirtiéndose en una referencia internacionalmente aceptada para el establecimiento de los Planes de Estudio de las titulaciones de Informática en la Universidad, incluyendo a la Universidad en España.

Aunque ACM parte de una tradición más proclive a la Ciencia de la Informática, en la última década del siglo XX se ha acercado a la parte de Ingeniería, convergiendo con IEEE-CS, de orientación tradicionalmente ingenieril. De hecho, ACM se está abriendo incluso al mundo de los Sistemas de Información, cosa que nunca ha hecho IEEE.

En 1968, ACM publica el primer currículo informático, denominado *Currículo 68* [ACM, 1968], donde sus propios autores manifiestan que iba destinado a los que pretendían dedicarse a la investigación dentro de la Informática.

En 1978, ACM revisa su propuesta curricular [Austing et al., 1979], de forma que este informe, con sólo unas pequeñas modificaciones [Koffman et al., 1984], [Koffman et al., 1985], constituye un estándar para la educación en Informática, hasta 1991.

En 1983, IEEE-CS publica un currículo independiente para Ciencia de la Informática e Ingeniería [Education Activities Board, 1983].

En 1986 se presenta un currículo para las escuelas de *artes liberales* [Gibbs and Tucker, 1986].

Dado que en los trabajos publicados por las dos principales sociedades de Informática tenían bastantes partes en común, deciden aunar esfuerzos y elaborar una única propuesta avalada por ambas sociedades. Así, en 1985 se crea un grupo de trabajo dirigido por **Peter Denning** y formado por miembros de ACM e IEEE-CS; este grupo publica un informe en diciembre de 1988 [Denning et al., 1988]. Se decide continuar con el trabajo para desarrollar las recomendaciones para un currículo completo, para lo que se crea otro grupo de trabajo conjunto en febrero de 1988 (*The Joint ACM/IEEE-CS Curriculum Task Force*), que da lugar al Computing Curricula 1991 [Tucker et al., 1991a].

A finales de 1998, la **ACM Education Board** y la **Educational Activities Board de IEEE-CS** establecieron un nuevo grupo de trabajo para preparar el *Curriculum 2001*.

Cuadro 4.5. Evolución de las propuestas curriculares en CS

La propuesta ACM/IEEE-CS de 1991 sigue actualmente en vigor, esperándose una nueva propuesta para el año 2001, Currículo 2001 [Roberts et al., 1999], [Chang et al., 1999], y será objeto de un estudio pormenorizado en el siguiente subapartado.

4.4.1.1 La propuesta conjunta ACM/IEEE-CS (Curricula 91)

La recomendación conjunta (ACM/IEEE-CS 91) [Tucker et al., 1991a], [Tucker et al., 1991b] incluye un currículo general en Informática, adaptable a las necesidades concretas de cada institución que imparta titulaciones como "*Computer Science*", "*Computer Engineering*", "*Computer Science and Engineering*" y otras similares; pretende ser flexible y adaptarse a los nuevos cambios tecnológicos que se vayan produciendo. El informe constituye un conjunto de guías, más que una prescripción de cursos concretos, en una titulación universitaria de Informática. El contenido de este informe puede resumirse en los siguientes puntos:

- Una identificación de objetivos de un plan de estudios universitario en la disciplina de Informática.
- Una definición de qué es la Informática como disciplina vista en forma bidimensional "área/proceso", que comprende la definición de nueve áreas temáticas y tres procesos, uno para cada una de las áreas (*Teoría, Abstracción* y *Diseño*).
- Una colección de material curricular avanzado y suplementario que posibilita la profundización en el estudio en algunas de las nueve áreas identificadas. El desarrollo de estos materiales variará en función de los intereses y posibilidades de cada institución que los imparta.
- Un conjunto de consideraciones pedagógicas y curriculares que gobiernen la materialización de los anteriores requisitos comunes y materiales avanzados/suplementarios en una titulación universitaria completa. Se incluyen aspectos como *el papel de los laboratorios, la programación, las matemáticas, conceptos éticos, sociales y profesionales* y hace explícita la noción de los "**conceptos recurrentes**" comunes a diferentes áreas, independientes de una tecnología en particular y repetidas por toda la disciplina informática.

A lo largo del documento se insiste en que un currículo es algo más que un conjunto de cursos y por eso tiene que conocerse no solamente la materia básica, sino también tienen que comprenderse los tres procesos o puntos de vista: teoría, abstracción y diseño.

Influencias sobre el *Computig Curricula 91*

Los siguientes puntos resumen los aspectos más significativos de algunos de los trabajos anteriores que han tenido influencia sobre ACM/IEEE-CS91:

- El informe de **ACM de 1968** [ACM, 1968] supone una de las primeras tentativas de definir el ámbito de la Informática como disciplina. Además de incluir un currículo, define los principales campos de la Informática; concretamente identifica tres áreas: *estructuras de información y procesos, sistemas de proceso de información* y ***metodologías***. Propone un currículo central compuesto de cuatro cursos básicos (*algoritmos y programación, estructura de computadores y sistemas, estructuras discretas* y *cálculo numérico*) y de cuatro cursos más avanzados (*estructuras de datos, lenguajes de programación, organización de computadores* y *programación de sistemas*). Estos cursos enfatizaban el análisis numérico y el hardware, omitiendo la Ingeniería del Software [Tucker and Wegner, 1994].
- El informe de **ACM de 1978** [Austing et al., 1979], proporcionaba descripciones detalladas de cursos universitarios en Informática, subrayando el término "*Computer Science*" (al igual que el anterior de 1968) y la importancia de la programación. La visión aquí es muy dependiente de la máquina, y solamente en uno de los cursos "*opcionales*" avanzados se hace referencia a un curso de "***Diseño y Desarrollo de Software***" que podría incluirse en el currículo; incluye técnicas "top-down" y estructuradas de diseño, formación de equipos de desarrollo y gestión de proyectos.
- El informe de **la Educational Activities Board de IEEE-CS de 1983**, describía una serie de áreas para Informática, considerando la orientación "*Computer Science and Engineering*". En este informe se hacían recomendaciones sobre material de laboratorio y sobre el propio laboratorio como soporte a las clases teóricas; utilizaba una estructura modular para organizar los temas y construir los cursos, proponiéndose como una base para configurar planes de estudios adaptados a distintas situaciones y centros particulares. En este informe se introduce la **Ingeniería del Software** en varios niveles: *como área temática básica*, *como Laboratorio específico de Ingeniería del Software* y *como área temática avanzada*, con un carácter dominado por las técnicas y métodos estructurados, propios de la época, pero subrayando ya la importancia de la fase de requisitos, modelado conceptual, prototipado y validación y verificación.
- El informe de 1989 "*Computing as a discipline*" [Denning et al., 1989], defiende la integración del trabajo de laboratorio con las clases teóricas, subrayando la importancia de introducir aspectos de *diseño* en el currículo y propone que en los cursos introductorios se dé una visión global de toda la carrera, incidiendo en los conceptos fundamentales, mientras que en los cursos superiores se debe ir profundizando en cada uno de esos temas. En este informe se proponen ya las nueve áreas temáticas que, posteriormente, aparecen en el documento ACM/IEEE-CS 91, distinguiendo para cada una

de ellas ***Teoría*** (*matemáticas subyacentes*), ***Abstracción*** (*modelado, análisis*) y ***Diseño*** (*especificar soluciones, estudiar alternativas*).

Objetivos del programa de graduación. Perfil de los graduados

El programa debe preparar a los estudiantes para la comprensión de las materias relacionadas con la computación en dos aspectos: *como una disciplina académica* y *como una profesión dentro de su contexto social*. Los estudiantes deben adquirir las habilidades para poder mantenerse actualizados y evaluar las ideas nuevas. Entre estas habilidades se incluyen el leer publicaciones, la asistencia a seminarios y la evaluación de su contenido, así como el trabajo en equipo en proyectos relacionados con problemas de actualidad.

En consecuencia, el primer objetivo del programa de formación ha de ser el proporcionar una cobertura básica, amplia y coherente de la disciplina de la computación. Los estudiantes deben comprender las relaciones existentes entre las diferentes áreas de la computación.

El programa debe preparar a los estudiantes para aplicar sus conocimientos a problemas específicos y con restricciones para elaborar soluciones. Esto incluye la *capacidad de definir un problema claramente*, *determinar su posibilidad de tratamiento, determinar la oportunidad de consultar con expertos, evaluar y elegir una estrategia de solución adecuada; estudiar, especificar, diseñar, implementar, probar, modificar y documentar esa solución, evaluar alternativas y realizar análisis de riesgos del diseño, integrar tecnologías alternativas en la solución y comunicar la solución tanto a los compañeros, a los profesionales de otros campos como al público en general*. Esto incluye la capacidad de trabajo en equipos durante todo el proceso de resolución de problemas.

El programa debe proporcionar la suficiente exposición del amplio cuerpo de teoría que subyace en el campo de la computación, de forma que los estudiantes aprecien la profundidad intelectual y los elementos abstractos que continuarán retando a los investigadores en el futuro.

Los graduados tienen que tener presente la alta tasa de cambio tecnológico, la tasa de crecimiento relativo en la teoría de la computación y la interacción delicada que tiene lugar entre las dos.

Principios subyacentes en el diseño curricular

En esta propuesta curricular se identifican nueve áreas de conocimientos y tres procesos que caracterizan las diferentes metodologías operativas utilizadas en la investigación y el desarrollo de la computación.

Cada una de las áreas identificadas tiene una base teórica significativa, abstracciones significativas y realizaciones significativas de diseño e implementación.

Las nueve áreas son: *Algoritmos y estructuras de datos; Arquitectura; Inteligencia artificial y robótica; Bases de datos y recuperación de la información; Comunicación hombre-máquina; Computación numérica y simbólica; Sistemas operativos; Lenguajes de Programación* y *Metodología e Ingeniería del Software*. De todas ellas las relacionadas con este Proyecto Docente son:

- ***Comunicación Hombre-Máquina***. El propósito principal de esta área es la transferencia eficiente de información entre el hombre y la máquina. Se incluyen gráficos, factores humanos que afectan a la interacción eficiente, y la organización y visualización de la información para una utilización efectiva por parte de las personas.
- ***Metodología e Ingeniería del Software***. El propósito principal de esta área es la especificación, el diseño y la producción de grandes sistemas software. El interés se centra en los principios de programación y del desarrollo del software, la verificación y la validación del software, y la especificación y producción de sistemas software que sean seguros y fiables.

La Informática es vista simultáneamente como una disciplina matemática, científica y de ingeniería. Los diferentes profesionales en cada una de las áreas emplean diferentes metodologías operativas, o procesos, en el curso de sus trabajos de investigación, desarrollo y aplicación.

El primero de estos procesos es el llamado **teoría**, está fundamentado en las matemáticas, y se utiliza en el desarrollo de teorías matemáticas coherentes. Los principales elementos que componen este proceso son: *definiciones y axiomas*, *teoremas*, *pruebas* e *interpretación de resultados*.

El segundo proceso, denominado **abstracción**, está enraizado en las ciencias experimentales, y consta de los siguientes elementos: *recolección de datos y formulación de hipótesis*, *modelado y predicción*, *diseño de un experimento*, *análisis de resultados*. Cuando las personas hacen abstracción están modelando algoritmos, estructuras de datos o arquitecturas, por ejemplo; prueban hipótesis acerca de esos modelos, toman decisiones de diseño alternativas... A los estudiantes hay que introducirles en la abstracción a través de las clases y de los laboratorios.

El tercer proceso, llamado **diseño**, está enraizado en la Ingeniería y se utiliza en el desarrollo de un sistema o dispositivo para resolver un determinado problema. Consta de las siguientes partes: *requisitos*, *especificaciones*, *diseño*, *implementación* y *pruebas*. Cuando los profesionales de la computación diseñan, han de implicarse en la conceptualización y la realización de sistemas en el contexto de las restricciones del mundo real. Los estudiantes aprenden diseño mediante la experiencia directa y mediante el estudio de los diseños de otros. Los proyectos de laboratorio se orientan hacia el diseño, proporcionando a los estudiantes una experiencia de primera mano en el desarrollo de un sistema o un componente de un sistema para resolver un problema

particular. Estos proyectos de laboratorio enfatizan en la síntesis de las soluciones prácticas a problemas y, por tanto, obligan a los estudiantes a evaluar las alternativas, costes y rendimientos en el contexto de las restricciones del mundo real. Los estudiantes desarrollan la capacidad de realizar estas evaluaciones viendo y discutiendo ejemplos de diseños, así como recibiendo información sobre sus propios diseños.

Conceptos recurrentes

Existen ciertos conceptos fundamentales que aparecen de forma recurrente en el diseño de los currículos de computación, y que representan un papel importante en el diseño de los cursos individuales. Estos conceptos son *ideas, materias, principios* y *procesos* que ayudan a unificar una disciplina académica en su substrato. En la propuesta curricular ACM/IEEE-CS91 se han identificado doce conceptos, a saber:

- **Ligadura**. El proceso de concretar abstracciones mediante la asociación de propiedades adicionales a dicha abstracción. Por ejemplo la asociación de procesos a procesadores o la creación de instancias concretas a partir de descripciones abstractas.
- **Complejidad de los grandes problemas**. Los efectos del crecimiento no lineal de la complejidad cuando crece el tamaño del problema.
- **Modelos conceptuales y formales**. Diferentes modos de formalizar, caracterizar, visualizar y concebir ideas o problemas. Los ejemplos incluyen modelos conceptuales del tipo de los tipos abstractos de datos y los modelos semánticos, y lenguajes visuales utilizados en la especificación y diseño de sistemas, tales como flujos de datos y diagramas entidad-relación.
- **Consistencia y compleción**. Incluye la consistencia de un conjunto de axiomas que sirven como especificación formal, la consistencia de la teoría con los hechos observados y la consistencia interna de un lenguaje. La compleción incluye la suficiencia de un conjunto de axiomas dados para capturar todos los comportamientos que se desea, la suficiencia funcional de los sistemas software y hardware, y la capacidad de un sistema para comportarse adecuadamente en condiciones de error o situaciones no previstas.
- **Eficiencia**. La medida de los costes relativos de los recursos tales como espacio, tiempo, dinero o personal.
- **Evolución**. El hecho del cambio y sus implicaciones.
- **Niveles de abstracción**. La naturaleza y uso de la abstracción en computación; la utilización de abstracciones para manejar la complejidad, estructurar sistemas, ocultar detalles y capturar patrones recurrentes. La capacidad de representar una entidad o sistema por abstracción tiene diferentes niveles de detalle y especificidad.

- **Ordenación en el espacio**. De los conceptos de localización y proximidad en la disciplina de la Informática. Además de la localización física, como en las redes o en la memoria de un computador, incluye el emplazamiento de la organización (por ejemplo, de procesadores y procesos; definiciones de tipo y las operaciones asociadas) y la localización conceptual (por ejemplo, el ámbito del software, la cohesión y el acoplamiento).
- **Ordenación en tiempo**. El concepto de tiempo en la ordenación de los eventos. Esto incluye el tiempo *como un parámetro en los modelos formales* (como por ejemplo en la lógica temporal), *como sinónimo de sincronización de procesos* o *como una parte esencial en la ejecución de los algoritmos*.
- **Reutilización**. La capacidad de una técnica, concepto o componente de sistema para ser reutilizado en un nuevo contexto o situación.
- **Seguridad**. La capacidad de los sistemas software y hardware para responder y defenderse de las peticiones no autorizadas, inapropiadas o imprevistas.
- **Decisiones y consecuencias**. El fenómeno de las decisiones en Informática y de las consecuencias que acarrean; los efectos técnicos, económicos, culturales que se derivan de la selección de una determinada alternativa de diseño.

El papel de los laboratorios

Un currículo de formación en Informática se compone, idealmente, de un programa integrado de clases teóricas y experiencias de laboratorio.

El papel de los laboratorios es muy importante en el diseño de un currículo en formación Informática, presentando las siguientes características y ventajas:

- Los laboratorios permiten demostrar la aplicación de los principios en el diseño, implantación y prueba de los sistemas software.
- Los laboratorios enfatizan las técnicas que utilizan las herramientas actuales y conducen hacia métodos experimentales adecuados incluyendo la presentación oral y escrita de los hallazgos.
- Los ejercicios de laboratorio han de estar diseñados para mejorar la experiencia del estudiante en las metodologías del software mediante el desarrollo de diseños e implantaciones.
- Las experiencias del laboratorio incrementan la capacidad de resolver problemas, las habilidades analíticas y la capacitación profesional.

Se diferencian dos tipos de laboratorios, *laboratorios abiertos* y *laboratorios cerrados*. Los primeros consisten en una asignación de trabajo que se llevará a cabo sin supervisión. Un laboratorio cerrado conlleva la asignación de trabajo de forma

planificada, estructurada y supervisada. La finalización de un trabajo de laboratorio debe de estar acompañada de un informe oral y/o escrito por parte del estudiante.

La unidad de conocimiento

El currículo en Informática se organiza sobre la base de unidades del conocimiento. Se entiende por unidad de conocimiento *la designación de una colección coherente de materias dentro de una de las nueve áreas principales de la disciplina de la computación*.

Para cada unidad de conocimiento se especifican los contenidos bajo el epígrafe de materias, los laboratorios propuestos, la posible relación con otras unidades de conocimiento, las unidades de conocimiento que son prerrequisito y si la unidad del conocimiento es requisito para otras unidades de conocimiento.

Las unidades de conocimiento que se definen en la propuesta curricular ACM/IEEE-CS91, relacionadas con el presente Proyecto Docente caen dentro del área de Metodología e Ingeniería del Software; cuyas unidades de conocimiento a su vez se recogen en la Tabla 4.14.

IS: Metodología e Ingeniería del Software *(se recomiendan unas 44 horas teóricas)*
IS1: Conceptos fundamentales para la resolución de problemas
IS2: El proceso de desarrollo del software
IS3: Requisitos y especificaciones del software
IS4: Diseño e implementación del software
IS5: Verificación y validación

Tabla 4.14. Unidades de conocimiento para el área de Metodología e Ingeniería del Software

De las unidades de conocimiento de esta área se describen aquéllas que se estima que deben estar contempladas en las asignaturas que se ajustan al perfil y al entorno en el que se va a desarrollar la docencia objeto de la plaza a concurso.

- **IS2.- El proceso de desarrollo del software.**

  Introducción a los modelos y elementos relacionados con el desarrollo de software de calidad. Utilización de entornos y herramientas que faciliten el diseño y la implantación de grandes sistemas software. El papel y la utilización de los estándares.

  1) Materias.

  a. Modelos del ciclo de vida del desarrollo del software.
  b. Objetivos del diseño del software.
  c. Documentación.
  d. Gestión y control de configuraciones.
  e. Elementos de la fiabilidad del software.

f. Mantenimiento.
g. Herramientas de especificación y diseño, herramientas de implementación.

2) Laboratorios (abiertos).

a. Implementar un prototipo para una especificación determinada.
b. Dado un diseño de software y una implementación intermedia de un desarrollo iterativo, completar la implementación correspondiente a la siguiente iteración.
c. Crítica de un conjunto de documentación determinado.
d. Dada una implementación, unas especificaciones y un conjunto de nuevas especificaciones, modificar el código de acuerdo a las nuevas especificaciones.

3) Relacionado con: Especificaciones y requisitos del software.

4) Prerrequisitos: Tipos abstractos de datos.

5) Requisito para: Diseño e implementación del software.

- **IS3.- Especificaciones y requisitos del software.**

Introducción al desarrollo de especificaciones formales e informales para la definición de los requisitos de un sistema software.

1) Materias.

a. Especificaciones informales.
b. Especificaciones formales, especificaciones algebraicas de los TAD, precondiciones y postcondiciones.

2) Laboratorios.

a. Producir un documento de análisis de requisitos.
b. Producir un documento con las especificaciones formales correspondientes a un conjunto de especificaciones informales.

3) Relacionado con: El proceso de desarrollo del software, control de tipos, semántica de los lenguajes.

4) Prerrequisitos: Tipos abstractos de datos.

5) Requisito para: Diseño e implementación del software, verificación y validación del software.

- **IS4.- Diseño e implementación del software.**

Introducción a los paradigmas principales que rigen el diseño y la implementación de grandes sistemas software.

1) Materias.

a. Diseño dirigido por funciones/procesos.
b. Diseño ascendente, soporte para reutilización.
c. Estrategias de implementación (por ejemplo, ascendente, descendente, equipos)
d. Elementos de la implementación, mejora de rendimientos, depuración.

2) Laboratorios (abiertos).

a. Realizar un diseño objeto para un conjunto de especificaciones.
b. Implementar el diseño anterior
c. Dado una especificación de problema y un conjunto de módulos ejecutables con sus especificaciones, realizar un diseño ascendente, reutilizando lo más posible.
d. Realizar una implementación descendente para un diseño software dado.

3) Relacionado con: Bases de datos, paradigmas de programación.

4) Prerrequisitos: IS2, IS3.

Otras consideraciones

La propuesta curricular ACM/IEEE-CS91 propone un curso optativo de nivel avanzado en Ingeniería del Software, centrado en los métodos y herramientas necesarios para incrementar la calidad, además de reducir el coste y la complejidad de mantenimiento de los sistemas software. Este curso tiene como prerrequisitos, además de diversas unidades de las áreas de algoritmos y estructuras de datos, cálculo numérico, lenguajes de programación y asuntos tanto éticos como sociales de la profesión, así como todo el área de metodología e Ingeniería del Software.

Se incluye una propuesta detallada de un programa universitario en Informática con énfasis en la Ingeniería del Software (*Implementation G: A Program in Computer Science – Software Engineering Emphasis*) [Tucker et al., 1991a].

En cuanto a la tecnología de objetos, cabe decir que tiene muy poca presencia en esta propuesta curricular (unas diez horas teóricas divididas en cuatro áreas [Osborne, 1992]), quizás porque ésta se centra en describir de una forma general el esqueleto básico de la Informática como disciplina, más que en establecer su cuerpo de conocimiento.

Aunque no es perfecta, esta propuesta curricular ha sido la que mayor influencia ha tenido en un plano internacional, incluyendo a España. Durante todos los años que lleva en vigor el *Computing Curricula 91*, se han propuesto diferentes modificaciones para incorporar o enfatizar diferentes aspectos, como la tecnología de objetos, la Ingeniería del Software o los Sistemas de Información; algunas de estas propuestas son: [Scragg et al., 1994], [Knight et al., 1994], [Shackelford and LeBlanc, 1994], [Hirmanpour et al., 1995], [Reynolds and Fox, 1996], [Pham, 1997] o [Jackson et al., 1997].

4.4.1.2 La propuesta conjunta ACM/IEEE-CS (Curricula 2001)

Reconocidas las limitaciones del *Computing Curricula 91*, de nuevo ACM e IEEE-CS se unen para crear un nuevo grupo de trabajo que defina una nueva propuesta curricular en el campo de la Ciencia de la Informática, que se espera esté terminada en el año 2001.

Actualmente este grupo de trabajo ha adoptado diez principios [Chang et al., 1999]:

1. La futura propuesta curricular debe perseguir el doble objetivo de formar buenos investigadores y buenos profesionales.
2. La Informática debe integrar la Matemática, la Ciencia y la Ingeniería.
3. Las unidades de conocimiento están disponibles en el proceso del diseño currículo. Estas unidades deben actualizarse para adaptarse a los nuevos conceptos que han aparecido en los últimos diez años.
4. La propuesta curricular debe ofrecer una guía para el diseño de cursos individuales. Se espera que en este sentido debe ser más efectivo que su antecesor.
5. El currículo 2001 debe identificar un conjunto relativamente pequeño de conceptos centrales y propiedades que son requeridos por todos los estudiantes.
6. El currículo 2001 debe ofrecer guías para cursos avanzados.
7. La propuesta curricular debe tener ámbito internacional.
8. La propuesta curricular debe ser el resultado de una participación significativa con la industria.
9. Debe tener en cuenta la práctica profesional.
10. Debe satisfacer las necesidades de los programas universitarios.

El grupo de trabajo ha identificado una serie de áreas de conocimiento clave, que deberán estar representadas en el informe final; éstas son: *matemáticas y ciencia, estructuras discretas, algoritmos y complejidad, arquitectura, sistemas inteligentes, gestión de la información, interacción hombre-máquina, gráficos por computador y visualización, sistemas operativos, fundamentos de programación, lenguajes y traducción,* ***Ingeniería del Software****, redes, ciencia computacional* y *asuntos sociales, éticos, legales y profesionales*.

4.4.1.3 Modelo de currículo para las artes liberales

Este modelo de currículo [Gibbs and Tucker, 1986] presenta una aproximación alternativa a la disciplina de la Informática, que pone un gran énfasis en que la Ciencia de la Informática tiene un cuerpo coherente de principios científicos. Define la Informática como un estudio sistemático de propiedades formales, implementación y aplicación de algoritmos y estructuras de datos.

La Ingeniería en esta propuesta curricular está ausente; en una posterior revisión [Walker and Schneider, 1996] se le asignan trece horas a la Ingeniería del Software (frente a las cuarenta y cuatro que recomendaba el *Computing Curricula 91*). Se presenta también un curso optativo que lleva por nombre *Ingeniería del Software*.

Existe alguna propuesta para la creación de un programa de estudios centrado en la Ingeniería del Software para su desarrollo en las Escuelas de Artes Liberales, como por ejemplo [Tymann et al., 1994].

4.4.2 Currículos centrados en los Sistemas de Información

Los Sistemas de Información es una parte esencial de las organizaciones. Son sistemas complejos que requieren tanto experiencia técnica como de organización para el diseño, el desarrollo y la gestión.

Hay una estrecha relación entre los Sistemas de Información y la Ciencia de la Informática y la Ingeniería del Software. Sin embargo, hay grandes diferencias ya que los Sistemas de Información se concentran en la parte organizativa y en la aplicación de las tecnologías de la información para sus objetivos.

Ninguna de las propuestas curriculares conjuntas de ACM/IEEE-CS entra en los Sistemas de Información, ya que son ajenos a la aplicación de las tecnologías de la información a los Sistemas de Información en las empresas u organizaciones; es decir, estarían cercanos a lo que en España se denomina *Ingeniería Técnica en Informática de Gestión*.

Dentro de un plan de estudios centrado en los Sistemas de Información, la Ingeniería del Software es una disciplina instrumental y su importancia relativa dependerá del grado de orientación tecnológica que se le quiera dar a la titulación, así hay titulaciones basadas en Sistemas de Información más orientadas al mundo empresarial, mientras otras están más orientadas al desarrollo de aplicaciones; estas últimas son las que utilizan conocimientos de Ingeniería del Software.

El desarrollo de currículos para Sistemas de Información comienza a principios de la década de los setenta, con el plan de estudios definido por la ACM [Ashenhurst, 1972]. Las razones de esta incursión en terrenos tradicionalmente ajenos fueron [Camps, 1999]:

a) El sector de los Sistemas de Información era y es el sector de aplicación de la Informática que más técnicos solicita y emplea.

b) En EEUU algunos departamentos de Ciencias de la Computación impartían enseñanzas propias de los Sistemas de Información.

La evolución histórica de los modelos de currículos para los Sistemas de Información se resume en el Cuadro 4.6.

La presencia de la Ingeniería del Software en estos modelos de currículos empieza a ser significativa a partir del modelo IS'90 definido por la **DPMA** (*Data Processing Management Association*) [Longenecker and Feinstein, 1991].

En el último modelo de currículo **IS'97** definido por la **ACM** (*Association for Computing Machinery*), la **AIS** (*Association for Information Systems*) y la **AITP** (*Association of Information Technology Professionals*) formalmente **DPMA** [Davis et al., 1997] presenta veinte subáreas significativas, de las que las áreas de **Análisis, diseño e implementación de Sistemas de Información** y de **Gestión de Proyectos** serían las más directamente relacionadas con la Ingeniería del Software, aunque muchas otras tendrían una relación más colateral. Por su parte el curso que más directamente se relaciona con la Ingeniería del Software es el que lleva el epígrafe **SI'97.7-Análisis y diseño lógico**, estando también relacionados los cursos con epígrafes **SI'97.8, SI'97.9** y **SI'97.10**; la descripción de estos cursos se encuentra en la Tabla 4.15**.**

	ÁMBITO	TEMAS
SI'97.7-Análisis y diseño lógico	Ofrece una comprensión del proceso de desarrollo y modificación de los sistemas software. Permite a los alumnos evaluar y escoger una metodología. Enfatiza la comunicación entre las partes interesadas. ADOO. Modelado de datos. Ciclo de vida estándar.	*Fases del ciclo de vida; técnicas de entrevista; JAD; DOO; prototipado; diseño de bases de datos; análisis de riesgos; gestión de proyectos; métricas de calidad del software; evaluación y adquisición de paquetes software; código de ética profesional.*
SI'97.8-Diseño físico e implementación con SGBD	Diseño e implementación de Sistemas de Información con un SGBD.	*Modelado de datos: herramientas y técnicas; paradigma estructurado y OO; modelos de bases de datos; CASE; repositorios; datawarehouse; IGU; cliente-servidor; conversiones; mantenimiento; formación de usuarios.*
SI'97.9-Diseño físico e implementación con entornos de programación	Diseño físico, programación, prueba e implantación de un sistema. Implementación OO y cliente-servidor utilizando entornos de desarrollo.	*Selección del entorno de programación cliente-servidor; construcción de software: estructurado, orientado a eventos y OO; pruebas; calidad del software; formación de usuarios; gestión de la configuración; mantenimiento; ingeniería inversa y reingeniería.*
SI'97.10-Gestión de proyectos y práctica	Cubre los factores necesarios de la gestión del desarrollo de sistemas. Cubre tanto los aspectos técnicos como los de comportamiento. Se centra en la gestión del desarrollo de sistemas de nivel empresarial.	*Gestión del ciclo de vida; integración de sistemas y bases de datos; gestión de redes y cliente-servidor; métricas para la gestión de proyectos y para la evaluación del rendimiento de sistemas; gestión de recursos humanos; análisis de coste-beneficio; técnicas de presentación; gestión de cambios.*

Tabla 4.15. Descripción de los cursos más relacionados con la Ingeniería del Software del IS'97

En el modelo curricular para graduados **MSIS 2000**, definido por la ACM y la AIS [Gorgone et al., 1999] presenta una línea profesional directamente relacionada con la Ingeniería del Software, **Analista y diseñador de sistemas**, que trataría temas de: *metodologías de diseño avanzadas (ADOO, RAD, Prototipado), Gestión de proyectos avanzada, Integración de sistemas* y *Consultoría de Sistemas de Información.*

La cronología de los diferentes currículos definidos en el campo de los Sistemas de Información comienza a comienzo de la década de los setenta con una propuesta curricular liderada por ACM. Los hechos más relevantes se presentan a continuación:

- Mayo de 1972: **ACM Graduate Professional Programs in Information Systems** [Ashenhurst, 1972].
- Diciembre de 1973: **ACM Undergraduate Programs in Information Systems** [Couger, 1973].
- Marzo de 1981: **ACM Educational Programs and Information Systems** [ACM, 1981].
- 1981 **DPMA Curriculum for Undergraduate Information Systems Education** [DPMA, 1981].
- En el año 1982, ACM elabora un nuevo informe para la enseñanza de los Sistemas de Información, el currículo **ACM-IS-82** [Nunamaker et al., 1982]. En él se hacen las siguientes distinciones entre el currículo en Ciencias de la Computación y el currículo para Sistemas de Información:
    - El currículo de Sistemas de Información enseña conceptos y procesos de Sistemas de Información, en dos contextos; conocimientos de organización y gestión, y conocimientos técnicos sobre Sistemas de Información. Por el contrario, las Ciencias de la Computación tienden a ser enseñadas en un entorno de matemáticas, algoritmos y tecnología.
    - En cuanto a conocimientos técnicos, el currículo de Sistemas de Información pone substancial énfasis en la capacidad para desarrollar la estructura de un Sistema de Información para una organización (institución/empresa) y para diseñar e implementar aplicaciones. Al titulado en Ciencias de la Computación se le suele hablar menos de análisis de requisitos de información y de consideraciones organizativas, pero adquiere mayores conocimientos en desarrollo de algoritmos, programación, software de sistemas y hardware.
- Durante 1990 se desarrolla el IS'90 propuesto por DPMA [Longenecker and Feinstein, 1991].
- En 1994 aparece el IS'94 de DPMA [Longenecker et al., 1994].
- Quince años después del currículo **ACM-IS-82**, en 1997 ACM vuelve a tratar los planes de estudio para los Sistemas de Información, cuando conjuntamente con la AITP (asociación norteamericana de profesionales de las tecnologías de la información, antes DPMA) y la AIS (asociación norteamericana de profesionales de los Sistemas de Información) propusieron un nuevo currículo para los Sistemas de Información, el IS'97 [Davis et al., 1997], de amplia repercusión.
- Actualmente se está trabajando una nueva actualización denominada IS'2000.
- Tanto el IS'97 como el IS'2000 son planes de estudios para estudiantes no graduados. Existe una propuesta de master especializado en Sistemas de Información, denominado MSIS2000 [Gorgone et al., 1999].

Cuadro 4.6. Cronología de los currículos en Sistemas de Información

4.4.3 Currículos centrados en la Ingeniería del Software

En su mayor parte, los currículos centrados en la Ciencia de la Informática están más orientados a la formación de científicos en computación que ingenieros, mientras que la realidad empresarial e industrial demanda profesionales que sean capaces de afrontar con éxito los proyectos que paulatinamente quedan inconclusos o con carencias significativas, suponiéndoles graves pérdidas económicas.

Para poder formar profesionales que afronten con garantías los problemas reales de las empresas se necesitan programas que hagan hincapié en la Ingeniería del Software, dado que los programas existentes centrados en las Ciencias de la Computación prestan poca atención a la Ingeniería del Software [Jalics and Golden, 1995], [Vaughn, 2000].

Existen diferentes propuestas curriculares que se centran en la Ingeniería del Software, ya sea en el contexto de los estudios de graduación o postgrado, cuya influencia está derivando en la definición de un cuerpo de conocimiento propio de la Ingeniería del Software que sentará la base para el establecimiento de una propuesta curricular de ámbito internacional que tenga a ésta como el objetivo central de la misma.

Se presentan a continuación las propuestas curriculares elaboradas por el **Instituto de Ingeniería del Software** (SEI – Software Engineering Institute) en la Universidad Carnegie-Mellon (CMU) en USA y por el **WGSEET** (Working Group on Software Engineering Education and Training).

4.4.3.1 Las propuestas del SEI-CMU

En 1985 la Universidad Carnegie Mellon presenta un currículo general en Informática [Shaw, 1985] donde la *Ingeniería del Software* aparece como una asignatura en el nivel intermedio, denominada "*Organización de programas*" (*"programming in the small", ampliación de TAD, POO, reusabilidad, especificaciones formales e informales...*) y en los cursos avanzados en forma de dos asignaturas: "*Ingeniería del Software*" (*"programming in the large", diseño avanzado y especificación, descomposición en módulos, CASE, prototipado, modelado...*) y "*Laboratorio de Ingeniería del Software*" (*desarrollo de proyectos en grupo, colaboración con la Industria*). En este currículo aparece el concepto de "*nociones recurrentes*" que después aparecerían como novedad en el currículo ACM/IEEE-CS91.

Con posterioridad el Instituto de Ingeniería del Software, SEI a partir de ahora, también en la Carnegie Mellon, realiza diferentes currículos específicos en Ingeniería del Software para estudios de graduación y de postgrado.

Programas de postgrado

El SEI establece primeramente estudios de postgrado, *masters*, centrados en Ingeniería del Software, que se detallan en diferentes informes y artículos entre 1989 y 1991

[Gibbs, 1989], [Ardis and Ford, 1989], [Ford, 1991a], siendo este último el de mayor difusión.

En el informe de 1991 se realiza un desarrollo muy completo y detallado de las áreas temáticas de Ingeniería del Software. El citado informe incluye un modelo de currículo, numerosa bibliografía sobre Ingeniería del Software y descripciones de revistas de investigación de la disciplina. Incluye también descripciones detalladas de los contenidos de la serie de audiovisuales (cintas de vídeo) que constituyen a su vez un ejemplo de una implementación del modelo de currículo. Se describen, además, los programas para postgraduados en Ingeniería del Software de 23 universidades de todo el mundo.

El material se organiza en cursos universitarios, e incorpora el concepto de "*unidad de conocimiento*", al igual que el ACM/IEEE-CS91. También como esta última propuesta, el currículo se organiza en una serie de materias fundamentales optativas avanzadas o complementarias (20-30% de un currículo), y proyectos prácticos de desarrollo (al menos un 30% del trabajo total del estudiante).

La secuencia que sugiere (aunque en esto no es restrictivo) es la de una aproximación docente, comenzando con una visión de proceso para colocar cada actividad individual en su contexto, para seguir con aspectos de gestión del software y actividades de control, finalizando con las actividades concretas de desarrollo y la visión producto.

En el currículo se incluyen materias tanto de Ingeniería del Software, como, de una forma más amplia, de Ingeniería de Sistemas. Las materias fundamentales se distribuyen en 21 unidades, sin recomendación temporal, describiendo para cada unidad sus contenidos, aspectos de la actividad que son más importantes y objetivos educativos de la unidad. A continuación se ofrece una relación de dichas unidades.

- El proceso de Ingeniería del Software
- Evolución del software
- Generación de software
- Mantenimiento de software
- Comunicación técnica
- Gestión de configuraciones software
- Conceptos de calidad del software
- Aseguramiento de la calidad del software
- Organización y gestión de proyectos de software
- Economía de proyectos software
- Conceptos de operación del software
- Análisis de requisitos
- Especificación
- Diseño de sistemas
- Diseño de software

- Implementación de software
- Pruebas del software
- Integración de sistemas
- Sistemas de tiempo real "empotrados"
- Interfaces hombre-máquina
- Asuntos de la profesión

Actualmente el master en Ingeniería del Software que se imparte en la Universidad Carnegie Mellon consta de tres elementos básicos [Garlan et al., 1997]:

1. **Un núcleo curricular:** Formado por los cursos sobre los que recaen los fundamentos de Ingeniería del Software, haciendo un énfasis especial en el análisis, diseño y gestión de grandes sistemas software. Los cursos que conforman este currículo son:
    a. **Modelos de sistemas software**
    b. **Métodos de desarrollo de software**
    c. **Gestión del desarrollo del software**
    d. **Análisis de elementos software**
    e. **Arquitecturas de sistemas software**
2. **Desarrollo de un proyecto:** Durante la duración del master, los alumnos planifican e implementan un proyecto software de tamaño significativo para un cliente externo. El trabajo se hace en equipo supervisado por profesores.
3. **Cursos de especialización:** De carácter optativo, que permiten que los estudiantes desarrollen una experiencia más profunda en una de las siguientes especialidades, entre las que se incluyen *sistemas de tiempo real, interfaces hombre-máquina* y *mejora del proceso software*.

Programas de graduación

En el SEI se llega a la conclusión de que la incesante demanda de ingenieros del software no se puede cubrir sólo con los estudiantes de postgrado, lo que hace necesaria la creación de programas de graduación en Ingeniería del Software. Estos programas quedan documentados en [Ford, 1990], [Ford, 1991b], [Ford, 1994].

El esquema de este currículo es:

1. **Matemáticas y Ciencia.** Con los objetivos de preparar a los alumnos para participar en una sociedad cada día más tecnológica, así como de ofrecerles los fundamentos necesarios para afrontar el resto de las asignaturas de la titulación. Se recomiendan dos asignaturas de matemáticas discretas, una de estadística y probabilidad, dos de cálculo, una de métodos numéricos, una de física, una de química y una de biología.

2. **Ciencia de Ingeniería y Diseño de Ingeniería.** Con los siguientes cursos:
    a. **Análisis.** Ofrece al alumno el conocimiento para realizar modelos y razonar sobre el proceso software. Algunos de los temas a tratar son: *desarrollo formal de algoritmos y programas (incluyendo la verificación formal de los mismos); técnicas de abstracción y modelado; sistemas formales y su aplicación a la Ingeniería del Software; métricas, análisis de algoritmos; análisis de rendimiento...*
    b. **Arquitecturas software.** Abordan la solución de problemas recurrentes a un alto nivel. Algunos temas pueden ser: *representación de datos, información y conocimiento; gestión de recursos; sistemas expertos; sistemas de tiempo real embebidos; sistemas concurrentes, paralelos o distribuidos...*
    c. **Hardware.** En la Ingeniería del Software no todo es software, debiendo un ingeniero del software comprender el hardware que está presente en los sistemas informáticos. Se tratan temas relacionados con *lenguajes ensambladores; arquitectura de computadores; sistemas digitales; redes...*
    d. **Proceso software.** Se encarga de transmitir el conjunto de herramientas, métodos y prácticas que se utilizan en la creación de software. Temas de este curso son: *análisis de requisitos; especificación y métodos formales; diseño; técnicas de implementación y lenguajes; verificación y validación; evolución del software; evaluación de productos y procesos software; gestión y organización de equipos de trabajo; temas éticos y profesionales.*
3. **Humanidades, Ciencias Sociales y Optativas.** Para completar la formación del alumno.

4.4.3.2 La propuesta del WGSEET

Cuando se presentaron los cuerpos de conocimiento sobre Ingeniería del Software, ya se comentó la iniciativa del WGSEET para la definición de un modelo de currículo para esta disciplina, que se recoge en [Bagert et al., 1999].

Esta propuesta curricular se centra en la generación de nuevos graduados que tengan en la Ingeniería del Software la base de conocimientos para afrontar una vida profesional condicionada por las necesidades de su entorno industrial y empresarial.

Arquitectura del currículo

En el diseño de este currículo intervienen una serie de elementos básicos, como se puede apreciar en la Figura 4.9.

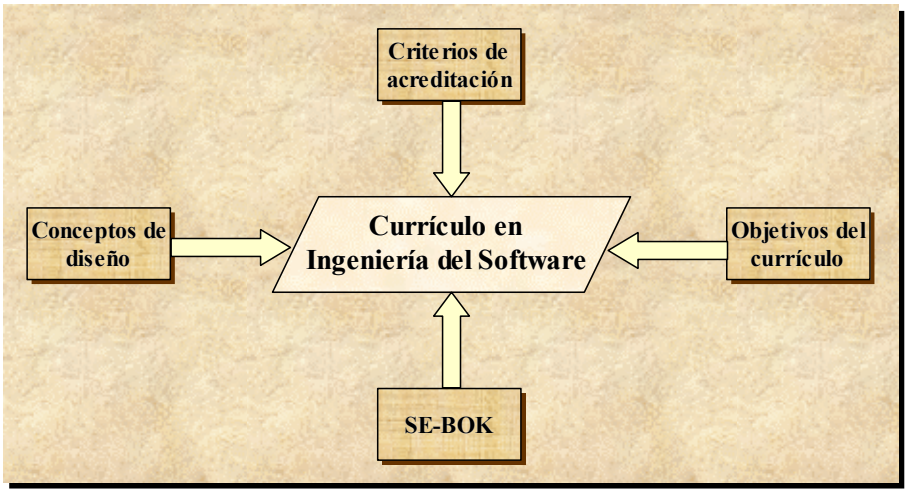

Figura 4.9. Arquitectura del currículo en Ingeniería del Software del WGSEET [Bagert et al., 1999]

La determinación de los objetivos del currículo se convierte en una actividad prioritaria en el diseño del mismo. Conlleva consideraciones sobre la misión de la institución que lo va a implantar y sobre la estrategia que se quiere seguir. En un nivel más fundamental, todos los afectados por la definición del currículo (*profesores, alumnos, empresarios...*) debieran estar representados en el momento de establecer estos objetivos.

El cuerpo de conocimientos ofrece la base de fundamentos para fijar el contenido del currículo, mientras que los criterios de acreditación introducen las bases para asegurar la calidad y la evaluación del mismo.

Por último, los conceptos de diseño construyen un marco filosófico para el desarrollo de currículos efectivos en Ingeniería del Software, así como para la definición del modelo de currículo del WGSEET.

Conceptos de diseño

La lista de conceptos que aparecen en esta sección constituye los principios básicos para el modelo de currículo en Ingeniería del Software propuesto por el WGSEET.

1. Soporta el desarrollo de varios programas de estudios, con la característica común de centrarse en la Ingeniería del Software (*Ingeniería del Software, Ciencias de la Computación, Sistemas de Información...*).
2. Introduce a los alumnos, desde el comienzo de sus estudios, en la naturaleza, ámbito e importancia de la Ingeniería del Software.
3. El modelo incorpora dos niveles de educación en Ingeniería del Software:
    a. La denominada Ingeniería del Software a pequeña escala (*Software Engineering in the Small*)

i. Aplica los principios de la Ingeniería del Software en el desarrollo de productos software realizados de forma individual o en pequeños grupos.

ii. El desarrollo del software con estas características se encuentra concentrado en los primeros cursos.

b. La denominada Ingeniería del Software a gran escala (*Software Engineering in the Large*)

i. Aplica los principios de la Ingeniería del Software en el desarrollo de productos software realizados en equipo.

ii. El desarrollo del software con estas características se encuentra localizado en cursos avanzados o en pequeños proyectos realizados para empresas.

4. El modelo establece un balance entre *producto* y *proceso*.

   a. Las actividades relacionadas con el producto incluyen métodos, técnicas y propiedades usados en la construcción de los elementos software asociados en un producto software (*planes de desarrollo, planes de aseguramientos de la calidad, especificación de requisitos, especificación de diseño, código, documentación...*).

   b. Las actividades relacionadas con el proceso incluyen los estándares, procedimientos, guías, métricas y técnicas de análisis y soporte que ofrecen el marco adecuado para llevar a cabo las actividades de construcción del producto de forma efectiva y eficiente.

5. El modelo ofrece una guía para el desarrollo de programas de graduación que pueden ser acreditados por organizaciones externas.

6. Se incluyen asuntos éticos, sociales y profesionales que estén directamente relacionados con la práctica profesional de la Ingeniería del Software.

7. El modelo enfatiza el concepto de trabajo en grupo.

8. Ofrece especialización en dominios de aplicación concretos (*sistemas empotrados, bases de datos y Sistemas de Información, sistemas inteligentes, redes...*).

9. Asegura la práctica de la Ingeniería del Software:

   a. Prácticas en laboratorios planificadas.

   b. Proyectos de Ingeniería del Software a realizar en laboratorio.

   c. Educación cooperativa y prácticas en empresas.

Contenidos del currículo

Aunque la organización y puesta en marcha de varios currículos centrados en la Ingeniería del Software puede diferir de unos a otros, todos ellos deben ofrecer asignaturas que incluyan conocimientos sobre *Matemáticas, Fundamentos de Ciencias de la Computación, Ingeniería del Software, Educación General, Ciencias Naturales* y *Conocimientos de Dominios Específicos*.

En la siguiente lista se describe lo que, como mínimo, el WGSEET considera que se debe incluir en cada área:

1. **Fundamentos de Ciencias de la Computación**
    a. Programación, estructuras de datos y algoritmos.
    b. Conceptos de lenguajes de programación.
    c. Organización de computadores.
    d. Sistemas software (*gestión de procesos y recursos, computación concurrente y distribuida, redes*).
2. **Matemáticas**
    a. Matemática discreta.
    b. Estadística y probabilidad.
3. **Ciencias naturales**
4. **Educación general**
    a. Comunicación (oral y escrita).
    b. Ciencias sociales y humanidades.
5. **Ingeniería del Software**
    a. Ingeniería de requisitos.
    b. Diseño del software.
    c. Calidad del software.
    d. Arquitecturas software.
    e. Construcción del software.
    f. Evolución del software.
    g. Métodos formales.
    h. Interfaces hombre-máquina.
    i. Organización, planificación de proyectos.
    j. Proceso software.
    k. Ética y profesionalidad en la Ingeniería del Software.

Módulos propios de la Ingeniería del Software

Los módulos que se presentan en la Tabla 4.16, pueden hacerse corresponder cada uno de ellos con una sola asignatura, o combinarse varios de ellos en una asignatura.

Módulo	Título	Descripción
IS1	*Introducción a la Ciencia de la Informática para ingenieros del software 1*	Introducción a la programación, normalmente en el paradigma procedimental. Las características básicas de la Ingeniería del Software se integran en este curso.
IS2	*Introducción a la Ciencia de la Informática para ingenieros del software 2*	Introducción a las estructuras de datos y al paradigma objetual. Se continúa con la introducción de conceptos de Ingeniería del Software.
IS3	*Introducción a la Ingeniería del Software*	Descripción general de la Ingeniería del Software como disciplina; introduce los principios fundamentales y las metodologías.
IS4	*Ética y profesionalismo*	Cubre material sobre aspectos históricos, sociales y económicos de la Ingeniería del Software. Incluye el estudio de las responsabilidades y riesgos profesionales, así como sobre la propiedad intelectual.
IS5	*Requisitos del software*	Introduce los conceptos básicos y principios de la ingeniería de requisitos: sus herramientas, técnicas y métodos para el modelado del software.
IS6	*Diseño del software*	Métodos y técnicas utilizados en la fase de diseño. Énfasis en el diseño orientado a objetos.
IS7	*Calidad del software*	Aseguramiento de la calidad y gestión de la configuración.
IS8	*Construcción y evolución del software*	Examina problemas, métodos y técnicas asociadas con la construcción del software, dado un diseño de alto nivel y teniendo en cuenta el mantenimiento del mismo.
IS9	*Proyecto*	Permite a los estudiantes poner en práctica los conocimientos adquiridos en el resto de módulos, al entrar a formar parte de un equipo de desarrollo que se encarga de la realización de un proyecto.

Tabla 4.16. Módulos relacionados con la Ingeniería del Software en el currículo WGSEET

Para una descripción más detallada de cada uno de estos módulos, incluyendo un índice de primer nivel de los contenidos de cada uno de ellos, se recomienda la consulta de [Bagert et al., 1999].

Influencias de otras propuestas curriculares

La propuesta curricular del WGSEET recibe influencias de otras propuestas anteriores, que también coinciden en el objetivo de establecer un currículo independiente para la disciplina de Ingeniería del Software. De todas ellas se van presentar las dos más relevantes.

Thomas B. Hilburn propone un modelo conceptual de currículo en Ingeniería del Software [Hilburn, 1997] que se basa en tres pilares fundamentales:

- ***Las personas:*** Un currículo debe soportar las actividades que formen al alumno para trabajar y comunicarse con sus semejantes de una forma efectiva. Las capacidades a considerar son: *educación general, capacidad de comunicación, trabajo en grupo* así como *ética y profesionalismo.*
- ***Proceso:*** Los estudiantes deben darse cuenta de la necesidad de utilizar un proceso definido para desarrollar productos software, distinguiendo el proceso para la creación de software de carácter individual, el proceso para trabajar en equipo y el proceso que involucra a los estándares seguidos por una organización.
- ***Tecnología:*** Los alumnos deben adquirir los conocimientos tecnológicos y científicos necesarios para el desarrollo de software de calidad, incluyendo conocimientos de *Matemáticas y Ciencia, Ciencias de la Computación* y *Desarrollo de Software*.

A. J. Cowling propone un modelo de currículo en Ingeniería del Software multi-dimensional [Cowling, 1998], que establece las siguientes dimensiones:

- **Los diferentes niveles de abstracciones que definen los componentes hardware y software**. Establece que el ámbito de la Ingeniería del Software está en la construcción de sistemas en los que su parte principal está constituida por componentes software. Permite distinguir a los ingenieros del software de aquéllos que trabajan en la parte de arquitecturas de computadores.
- **El balance de los contenidos en Informática con respecto a otras ramas de la Ciencia y la Ingeniería**. Permite distinguir entre la Ingeniería del Software y otras disciplinas.
- **El balance entre la teoría, el modelado y su aplicación práctica**. La base de esta tercera dimensión es la observación de que la Ingeniería del Software no es la mera construcción de sistemas software, sino que la construcción de éstos se hace acorde a un método de Ingeniería.
- **El balance entre la parte técnica y la parte no técnica**. Las tres primeras dimensiones se centran en la parte técnica, pero una característica importante de la Ingeniería es que consiste en algo más que un conjunto de aspectos técnicos. Se necesitan contemplar aspectos económicos, sociales y sicológicos.

4.5 Referencias

[Abran et al., 1999] Abran, Alain (Co-Excutive Editor), Moore, James W. (Co-Excutive Editor), Bourque, Pierre (Editor), Dupuis, Robert (Editor) and Tripp, Leonard L. (Project Champion). "*Guide to the Software Engineering Body of Knowledge. A Stone Man Version*". Version 0.5. ACM/IEEE-CS, October, 1999.

[Abran et al., 2000] Abran, Alain (Co-Excutive Editor), Moore, James W. (Co-Excutive Editor), Bourque, Pierre (Editor), Dupuis, Robert (Editor) and Tripp, Leonard L. (Project Champion). "*Guide to the Software Engineering Body of Knowledge. A Stone Man Version*". Version 0.6. ACM/IEEE-CS, February, 2000. Available on line at http://www.swebok.org [Última vez visitado 21-3-2000].

[ACM, 1968] ACM Curriculum Committee on Computer Science. "*Curriculum '68: Recommendations for Academic Programs in Computer Science*". Communications of the ACM, 11(3):151-197. March, 1968.

[ACM, 1981] ACM Committee on Computer Curricula of ACM Education Board. "*ACM Recommended Curricula for Computer Science and Information Processing Programs in Colleges and Universities*". ACM, New York, 1981.

[ACM/IEEE-CS, 1998] ACM/IEEE-CS. "*Accreditation Criteria for Software Engineering*". http://www.acm.org/serving/se/Accred.htm. [Última vez visitado 28/12/1999]. September, 1998.

[ACM/IEEE-CS, 1999a] ACM/IEEE-CS. "*Software Engineering Education Project*". http://www.acm.org/serving/se/SWEE.htm. [Última vez visitado 28/12/1999]. 1999.

[ACM/IEEE-CS, 1999b] ACM/IEEE-CS. "*1999 Plan for the Software Engineering Education Project (SWEEP)*". Draft 0.5. http://www.acm.org/serving/se/sweep.htm. [Última vez visitado 28/12/1999]. April, 1999.

[ACM/IEEE-CS, 1999c] ACM/IEEE-CS. "*Software Engineering Code of Ethics and Professional Practice*". Version 5.2. http://computer.org/tab/sweec/code.htm. [Última vez visitado 28/12/1999]. 1999.

[Ada95-Web] "*Ada 95 Reference Manual: Language and Standard Library*". http://lglwww.epfl.ch/Ada/LRM/9X/rm9x/rm9x-toc.html [Última vez visitado 8/1/2000].

[Adams, 1996] Adams, Joel C. "*Object-Centered Design. A Five-Phase Introduction to Object-Oriented Programming in CS1-2*". In Proceedings of the twenty-seventh SIGCSE technical symposium on Computer Science Education - SIGCSE '96. (Feb. 15-18, 1996, Philadelphia, PA, USA). Pages 78-82. ACM. 1996.

[AECC, 1986] Asociación Española para la Calidad. "*Glosario de Términos de Calidad e Ingeniería del Software*". AECC, 1986.

[Alhir, 1998] Alhir, Sinan Si. "*The Object-Oriented Paradigm*". http://home.earthlink.net/~salhir/theobjectorientedparadigm.html. [Última vez visitado 23/12/1999]. October, 1998.

[Apple, 1989] Apple Computer Inc. "*Macintosh Programmer's Workshop Pascal 3.0 Reference*". Apple Computer, 1989.

[Ardis and Ford, 1989] Ardis, Mark and Ford, Gary. "*1989 SEI Report on Graduate Software Engineering Education*". Technical Report CMU/SEI-89-TR-21 (ESD-TR-89-

29), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, PA 15213 (USA). June, 1989.

[Arnold and Gosling, 1997] Arnold, Ken and Gosling, James. "*The Java Programming Language*". 2nd edition. Addison-Wesley, 1997.

[Ashenhurst, 1972] Ashenhurst, R. L. (Editor). "*Curriculum Recommendations for Graduate Professional Programs in Information Systems*". ACM, 1972.

[Ashworth and Goodland, 1990] Ashworth, C. and Goodland, M. "*SSADM: A Practical Approach*". McGraw-Hill, 1990.

[Atkinson and Buneman, 1987] Atkinson, M. and Buneman, P. "*Types and Persistence in Database Programming Languages*". ACM Computing Surveys, 19(2). 1987.

[Atkinson et al., 1989] Atkinson, Malcom, Bancilhon, François, DeWitt, David, Dittrich, Klaus, Maier, David, Zdonik, Stanley. "*The Object-Oriented Database System Manifesto*". In Proceedings of the First International Conference on Deductive and Object-Oriented Databases, Kyoto (Japan). 1989. Also in Deductive and Object-Oriented Databases, Elsevere Science Publishers, Amsterdam, Netherlands, 1990.

[Austing et al., 1979] Austing, Richard, Barnes, Bruce, Bonnette, Della, Engel, Gerald and Stokes, Gordon. "*Curriculum '78: Recommendations for the Undergraduate Program in Computer Science*". Communications of the ACM, 22(3):147-166. March, 1979.

[Bagert et al., 1999] Bagert, Donald J., Hilburn, Thomas B., Hislop, Greg, Lutz, Michael, McCracken, Michael and Mengel, Susan. "*Guidelines for Software Engineering Education. Version 1.0*". Working Group on Software Engineering Education and Training (WGSEET). August, 1999.

[Bailin, 1989] Bailin, Sidney C. "*An Object-Oriented Requirements Specification Method*". Communications of the ACM, 32(5):608-623. Mayo, 1989.

[Barr and Feigenbaum, 1981] Barr, A. and Feigenbaum, E. "*The Handbook of Artificial Intelligence*". Vol. 1. William Kaufmann, 1981.

[Basili, 1991] Basili, V. R. "*The Future Engineering of Software: A Management Perspective*". IEEE Computer, 24(9):90-96. September, 1991.

[Bauer, 1972] Bauer, F. L. "*Software Engineering*". Information Processing 71. Amsterdam: North Holland, 1972.

[BCS, 1989] The British Computer Society and The Institution of Electrical Engineering. "*A Report on Undergraduate Curricula for Software Engineering Curricula*". June, 1989.

[Bellin, 1999] Bellin, David. "*Pedagogical Pattern #4. Brainstorming Pattern*". Version 2.0. In [Proto-Patterns, 1999]. http://www-lifia.info.unlp.edu.ar/ppp/pp4.htm. [Última vez visitado, 20/8/1999]. July, 1999.

[Bergin, 1998a] Bergin, Joseph. "*Pedagogical Patterns*". http://csis.pace.edu/~bergin/PedPat1.2.html. [Última vez visitado, 3/8/1999]. October, 1998.

[Bergin, 1998b] Bergin, Joseph. "*Six Pedagogical Patterns*". http://csis.pace.edu/~bergin/fivepedpat.html. [Última vez visitado, 3/8/1999]. October, 1998.

[Bergin, 1998c] Bergin, Joseph. "*Pedagogical Pattern #32. Spiral Pattern*". Versión 1.2. In [Proto-Patterns, 1999]. http://www-lifia.info.unlp.edu.ar/ppp/pp32.htm. [Última vez visitado, 20/8/1999]. October, 1998.

[Bertino and Martino, 1993] Bertino, E. and Martino, L. "*Object-Oriented Database Systems. Concepts and Architectures*". Addison-Wesley, 1993.

[Bertolino, 1999] Bertolino, A. "*KA Description of Software Testing V. 0.5*". In [Abran et al., 1999], 1999.

[Bézivin et al., 1992] Bézivin, Jean, Roux, Olivier and Royer, Jean-Claude. "*Teaching Object-Oriented Programming or Using the Object Model to Teach Software Engineering*". In addendum to the proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum) - OOPSLA '92. (Oct. 18-22, 1992, Vancouver, British Columbia, Canada). Pages 269-275. ACM. 1992.

[Birtwistle et al., 1973] Birtwistle, Graham M., Dahl, Ole-Johan, Myhrhaug, Bjorn and Nygaard, Kristen. "*Simula Begin*". Studentliteratur, 1973.

[Blaha and Premerlani, 1998] Blaha, Michael and Premerlani, William. "*Object-Oriented Modeling and Design for Database Applications*". Prentice Hall, 1998.

[Blaha et al., 1988] Blaha, Michael, Premerlani, William and Rumbaugh, J. "*Relational Database Design Using an Object-Oriented Methodology*". Communications of the ACM, 31(4):414-427. April, 1988.

[Blair et al., 1991] Blair, G., Gallagher, J., Hutchinson, D. and Shepard, D. "*Object-Oriented Languages, Systems and Applications*". Halsted Press, 1991.

[Blum, 1992] Blum, B. I. "*Software Engineering, A Holistic View*", Oxford University Press, New York, 1992.

[Bobrow and Stefik, 1982] Bobrow, Daniel G. and Stefik, Mark J. "*LOOPS: an Object-Oriented Programming System for Interlisp*". Xerox PARC, 1982.

[Boehm, 1976] Boehm, B. W. "*Software Engineering*". IEEE Transactions on Computers. C-25(12). December, 1976.

[Bollinguer, 1999] Bollinguer, Terry. "*Software Construction (Versión 0.5)*". In [Abran et al., 1999], 1999.

[Booch, 1991] Booch, Grady. "*Object Oriented Design with Applications*". The Benjamin/Cummings Publishing Company, 1991.

[Booch, 1994] Booch, Grady. "*Object Oriented Analysis and Design with Applications*". 2nd Edition. The Benjamin/Cummings Publishing Company, 1994.

[Booch and Rumbaugh, 1995] Booch, Grady and Rumbaugh, James. "*Unified Method for Object-Oriented Development*". Documentation set, version 0.8. Rational Software Corporation, 1995.

[Booch et al., 1996a] Booch, Grady, Jacobson, Ivar and Rumbaugh, James. "*The Unified Modeling Language for Object-Oriented Development*". Documentation set, version 0.9 Addendum. Rational Software Corporation, June 1996.

[Booch et al., 1996b] Booch, Grady, Jacobson, Ivar and Rumbaugh, James. "*The Unified Modeling Language for Object-Oriented Development*". Documentation set, version 0.91 Addendum UML update. Rational Software Corporation, September 1996.

[Booch et al., 1997a] Booch, Grady, Jacobson, Ivar and Rumbaugh, James. "*The Unified Modeling Language for Object-Oriented Development*". Documentation set, version 1.0. Rational Software Corporation, 13 January 1997.

[Booch et al., 1997b] Booch, Grady, Jacobson, Ivar and Rumbaugh, James. "*The Unified Modeling Language for Object-Oriented Development*". Documentation set, version 1.0.1. Rational Software Corporation, 19 March 1997.

[Booch et al., 1999] Booch, Grady, Rumbaugh, James and Jacobson, Ivar. "*The Unified Modeling Language User Guide*". Object Technology Series. Addison-Wesley, 1999.

[Bourque et al., 1998] Bourque, Pierre, Dupuis, Robert, Abran, Alain, Moore, James W., Tripp, Leonard, Shyne, Karen, Pflug, Bryan, Maya, Marcela and Tremblay, Guy. "*Guide to the Software Engineering Body of Knowledge. A Straw Man Version*". ACM/IEEE-CS, September, 1998.

[Bourque et al., 1999a] Bourque, Pierre, Dupuis, Robert, Abran, Alain, Moore, James W. and Tripp, Leonard. "*The Guide to the Software Engineering Body of Knowledge*". IEEE Software, 16(6):35-44. November-December, 1999.

[Bourque et al., 1999b] Bourque, Pierre, Dupuis, Robert, Abran, Alain, Moore, James W., Tripp, Leonard and Frailey, Dennis. "*Approved Baseline for a List of Knowledge Areas for the Stone Man Version of the Guide to the Software Engineering Body of Knowledge*". Technical Report. January, 1999.

[Bowyer, 1996] Bowyer, Kevin W. "*Ethics and Computing: Living Responsibly in a Computerized World*". IEEE Computer Society Press, 1996.

[Budd, 1991] Budd, Timothy. "*An Introduction to Object-Oriented Programming*". Addison-Wesley, 1991.

[Buxton and Randell, 1970] Buxton, J. N. and Randell, B. (Editors). "*Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27-31 October, 1969*". Brussels: Scientific Affairs Division, NATO. April, 1970.

[Buxton et al., 1976] Buxton, J. M., Naur, P. and Randell, B. (Editors) "*Software Engineering Concepts and Techniques*". Proceedings of 1968 NATO Conference on Software Engineering, Van Nostrand Reinhold, 1976.

[Cameron, 1989] Cameron, J. "*JSP & JSD: The Jackson Approach to Software Development*". 2nd edition. IEEE Computer Society Press, 1989.

[Camps, 1999] Camps Paré, Rafael. "*¿Qué Informática Se Enseña en la Universidad? (Primera Parte)*". Novática. Nº 141: 48-51. Septiembre/Octubre, 1999.

[Cannon, 1980] Cannon, H. I. "*Flavors*". Technical Report, MIT Artificial Intelligence Laboratory, Cambridge (Mass.), 1980.

[Carrington, 1999] Carrington, David. "*SWEBOK Knowledge Area Description for Software Engineering Infraestructure (version 0.5)*". In [Abran et al., 1999], 1999.

[Castellanos et al., 1991] Castellanos, M. G., Saltor, F. and García-Solaco, M. "*The Development of Semantic Concepts in BLOOM Model Using an Object Metamodel*". Technical Report LSI-91-22. Dept. de Llenguatges i Sistemes Informátics. Universidad Politécnica de Cataluña, 1991.

[Coleman et al., 1994] Coleman, D., Arnold, P., Bodoff, S., Dolin, C., Hayes, F. and Jeremaes, P. "*Object-Oriented Development: The Fusion Method*". Prentice-Hall, 1994.

[Cook and Daniels, 1994] Cook, S. and Daniels, J. "*Designing Object Systems: Object Oriented Modelling with Syntropy*". Prentice-Hall, 1994.

[Cook et al., 1997] Cook, Steve, Selic, Bran, Gangopadhyay, Dipayan, Gheorge, Serban, Gullekson, Garth, Hogg, John, McGee, Jim, Meier, Mike, Mitra, Subrata, Saaltink,

Mark, Warmer Jos and Wills Alan. "*OMG OA&D RFP OMG OA&D RFP Response*". Document Version 1.0. IBM Corporation and ObjecTime Limited. 10 January 1997.

[Couger, 1973] Couger, J. (Editor) "*Curriculum Recommendations for Undergraduate Programs in Information Systems*". Communications of the ACM, 16(12): 727-749. December, 1973.

[Cowling, 1998] Cowling, A. J. "*A Multi-Dimensional Model of the Software Engineering Curriculum*". In Proceedings of the 11th Conference on Software Engineering Education and Training – CSEE&T'98. (February 22-25, 1998, Atlanta, GA, USA). Pages 44-55. IEEE Computer Society. 1998.

[Cox, 1984] Cox, Brad J. "*Message/Object Programming: An Evolutionary Change in Programming Technology*". IEEE Software, 1(1):50-69. January, 1984.

[Cox and Novobilski, 1990] Cox, Brad J. and Novobilski, Andrew J. "*Object-Oriented Programming: An Evolutionary Approach*". 2nd edition. Addison-Wesley, 1990.

[Champeaux et al., 1993] Champeaux, Dennis, Lea, Doug and Faure, Penelope. "*Object-Oriented System Development*". Addison Wesley. 1993.

[Chang et al., 1999] Chang, Carl K., Engel, Gerald, King, Willis, Roberts, Eric, Shackelford, Russ, Sloan, Robert H. and Srimani, Pradip K. "*Curricula 2001: Bringing the Future to the Classroom*". Computer, 32(9):85-88. September, 1999.

[Chen, 1976] Chen, Peter. "*The Entity-Relationship Model: Toward a Unified View of Data*". ACM Transactions on Database Systems, 1(1):9-36. March, 1976.

[Dahl and Hoare, 1972] Dahl, Ole-Johan and Hoare, C. A. R. "*Hierarchical Program Structures*". In Dahl, Dijkstra, Hoare, *Structured Programming*, Academic Press, pages 175-220. 1972.

[Dahl and Nygaard, 1966] Dahl, Ole-Johan and Nygaard, Kristen. "*SIMULA — An Algol-Based Simulation Language*". Communication of the ACM, 9(9):671-678. September, 1966.

[Dahl et al., 1970] Dahl, Ole-Johan, Myrhaug, Bjorn and Nygaard, Kristen. "*(Simula 67) Common Base Language*". Norsk Regnesentral (Norwegian Computing Center), Publication N. S-22, Oslo. October, 1970.

[Davis, 1993] Davis, Alan M. "*Software Requirements. Objects, Functions and States*". Prentice-Hall International, 1993.

[Davis et al., 1997] Davis, Gordon B., Gorgone, John T., Couger, J. Daniel, Feinstein, David L. and Longenecker, Jr. Herbert E. (Editors) "*IS'97 Model Curriculum and Guidelines for Undergraduate Degree Programs in Information Systems*". ACM, AIS and AITP, 1997.

[Decker and Hirshfield, 1994] Decker, Rick and Hirshfield, S. "*The Top 10 Reasons Why Object-Oriented Programming Can't Be Taught in CS1*". In Proceedings of the twenty-fifth annual SIGCSE symposium on Computer Science Education (SIGCSE '94). (March 10-11, 1994, Phoenix, AZ – USA). Pages 51-55. ACM. 1994.

[DeMarco, 1979] DeMarco, Tom "*Structured Analysis and System Specification*". Prentice-Hall, 1979.

[Denning, 1992] Denning, Peter J. "*Educating a New Engineer*". Communications of the ACM, 35(12). December, 1992.

[Denning et al., 1988] Denning, Peter J., Comer, Douglas E., Gries, David, Mulder, Michael C., Tucker, Allen B., Turner, A. Joe and Young, Paul R. "*Report of the ACM Task Force on the Core of Computer Science*". ACM Press, 1988.

[Denning et al., 1989] Denning, Peter J., Comer, Douglas E., Gries, David, Mulder, Michael C., Tucker, Allen B., Turner, A. Joe and Young, Paul R. "*Computing as a Discipline*". Communications of the ACM, 32(1):9-23. January, 1989.

[Dijkstra, 1968] Dijkstra, E. "*The Structure of 'THE' Multiprogramming System*". Communications of the ACM, 11(5). May, 1968.

[Diller, 1990] Diller, A. "*Z: An Introduction to Formal Methods*". John Wiley & Sons, 1990.

[Dodani, 1999] Dodani, Mahesh. "*OO Learning AntiPatterns: Rewriting Data and Functional Thinkers into Object Technology Developers*". Journal of Object-Oriented Programming (JOOP), 11(8):59-63. January, 1999.

[Dolado, 1999] Dolado, Javier. "*El Código de Ética y Práctica Profesional de la Ingeniería del Software de la ACM/IEEE Computer Society*". Novática. Nº 140. Julio-Agosto, 1999.

[Dorling, 1993] Dorling, Alec. "*Software Process Improvement and Capability Determination*". Software Quality Journal, 12(4): 209-224. December, 1993.

[DPMA, 1981] Data Processing Management Association. "*DPMA Model Curriculum, 1981*". Published by DPMA, Chicago, 1981.

[DRAE, 1995] Real Academia Española. "*Diccionario de Real Academia*". Vigésimo primera edición. Espasa-Calpe. Edición electrónica, versión 21.1.0. 1995.

[Duncan, 1996] Duncan, William R. "*A Guide to the Project Management Body of Knowledge*". PMI Standards Committee. Project Management Institute, Four Campus Boulevart, Newtown Square, PA 19073-3299, USA. 1996.

[D'Souza, 1996] D'Souza, Desmond F. "*Objects: Education vs. Training*". ICON Computing Inc. http://www.iconcomp.com/papers/education-vs-training/EducationvsTraining.frm.html. [Última vez visitado, 24/5/1999]. 1996.

[D'Souza and Wills, 1999] D'Souza, Desmond F. and Wills, Alan Cameron. "*Objects, Components, and Frameworks with UML. The Catalysis Approach*". Object Technology Series. Addison-Wesley, 1999.

[Education Activities Board, 1983] Education Activities Board. "*The 1983 Model Program in Computer Science and Engineering*". Technical Report 932. IEEE Computer Society. December, 1983.

[Ehrig and Mahr, 1985] Ehrig, H. and Mahr, B. "*Fundamentals of Algebraic Specification 1*". Springer-Verlag, EATCS Nº6, 1985.

[Ellis and Stroustrup, 1990] Ellis, Margaret and Stroustrup, Bjarne. "*The Annotated C++ Reference Manual*". Addison Wesley, 1990.

[Fairley, 1985] Fairley, R. "*Software Engineering Concepts*". McGraw-Hill, 1985.

[Firesmith et al., 1998] Firesmith, Donald, Henderson-Sellers, Brian and Graham, Ian. "*OPEN Modeling Language (OML) Reference Manual*". Cambridge University Press, 1998.

[Ford, 1990] Ford, Gary. "*1990 SEI Report on Undergraduate Software Engineering Education*". Technical Report CMU/SEI-90-TR-3 (ESD-TR-90-204), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, PA 15213 (USA). March, 1990.

[Ford, 1991a] Ford, Gary. "*1991 SEI Report on Graduate Software Engineering Education*". Technical Report CMU/SEI-91-TR-2 (ESD-TR-91-2), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, PA 15213 (USA). April, 1991.

[Ford, 1991b] Ford, Gary. "*The SEI Undergraduate Curriculum in Software Engineering*". In Proceedings of the twenty-second SIGCSE technical symposium on Computer Science Education – SIGCSE'91. (March 7-8, 1991, San Antonio, Texas, USA). Pages 375-385. ACM. 1991.

[Ford, 1994] Ford, Gary. "*A Progress Report on Undergraduate Software Engineering Education*". Technical Report CMU/SEI-94-TR-11 (ESC-TR-94-011), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, PA 15213 (USA). May, 1994.

[Frakes et al., 1991] Frakes, William B., Fox, Christopher, Nejmeh and Brian A. "*Software Engineering in the UNIX/C Environment*". Prentice Hall, 1991.

[Gane and Sarson, 1977] Gane, C. and Sarson, T. "*Structured Systems Analysis and Design*". Improved Systems Technologies, Inc., 1977.

[Gane and Sarson, 1979] Gane, C. and Sarson, T. "*Structured Systems Analysis: Tools and Techniques*". Prentice-Hall, 1979.

[García y Pardo, 1998] García Peñalvo, Francisco José y Pardo Aguilar, Carlos. "*UML 1.1. Un Lenguaje de Modelado Estándar para los Métodos de ADOO*". Revista Profesional para Programadores (RPP), Editorial América-Ibérica, V(1):57-61. Enero, 1998.

[Garlan et al., 1997] Garlan, David, Gluch, David P. and Tomayko, James E. "*Agents of Change: Educating Software Engineering Leaders*". IEEE Computer, 30(11):59-65 November, 1997.

[Gibbs, 1989] Gibbs, Norman E. "*The SEI Education Program: The Challenge of Teaching Future Software Engineers*". Communications of the ACM, 32(5):594-605. May, 1989.

[Gibbs and Tucker, 1986] Gibbs, N. E. and Tucker, A. B. "*Model Curriculum for a Liberal Arts Degree in Computer Science*". Communications of the ACM, 29(3):202-210. March, 1986.

[Gómez et al., 1998] Gómez, A., Juristo, N., Montes, C. y Pazos, J. "*Ingeniería del Conocimiento*". Madrid. Ceura, 1998.

[Goguen and Meseguer, 1988] Goguen, J. A. and Meseguer, J. "*Order-Sorted Álgebra I*". Technical Report, SRI International, Stanford University, 1988.

[Goguen et al., 1992] Goguen, J. A., Winkler, T., Meseguer, J., Futatsugui, K. and Jouannaud, J. P. "*Introducing OBJ*". SRI-CSL Report. Draft of January, 1992.

[Goldberg and Kay, 1976] Goldberg, Adele and Kay, Alan. "*Smalltalk-72 Instruction Manual*". Technical Report SSL-76-6, Xerox Palo Alto Research Center. March, 1976.

[Goldberg and Robson, 1983] Goldberg, Adele and Robson, David. "*Smalltalk-80: The Language and its Implementation*". Addison-Wesley, 1983.

[Goldberg, 1985] Goldberg, Adele. "*Smalltalk-80: The Interactive Programming Environment*". Addison-Wesley, 1985.

[Goldberg, 1986] Goldberg, R. "*Software Engineering: An Emerging Discipline*". IBM Systems Journal. 25(3/4), 1986.

[Gorgone et al., 1999] Gorgone, John T., Gray, Paul, Feinstein, David L., Kasper, George M., Luftman, Jerry N., Stohr, Edward A., Valacich, Joseph and Wigand, Rolf T.

"*MSIS 2000 Model Curriculum and Guidelines for Graduate Degree Programs in Information Systems*". ACM and AIS. November, 1999.

[Gotterbarn, 1999] Gotterbarn, D. "*How the New Software Engineering Code of Ethics Affects You*". IEEE Software, 16(6):58-64. November/December, 1999.

[Gotterbarn et al., 1997a] Gotterbarn, D., Miller, K. and Rogerson, S. "*Software Engineering Code of Ethics*". Communications of the ACM, 40(11):110-118. November, 1997.

[Gotterbarn et al., 1997b] Gotterbarn, D., Miller, K. and Rogerson, S. "*Software Engineering Code of Ethics, Version 3.0*". IEEE Computer, 30(11):88-92. November, 1997.

[Gotterbarn et al., 1999a] Gotterbarn, D., Miller, K. and Rogerson, S. "*Computer Society and ACM Approve Software Engineering Code of Ethics*". IEEE Computer, 32(10):84-88. October, 1999.

[Gotterbarn et al., 1999b] Gotterbarn, D., Miller, K. and Rogerson, S. "*Software Engineering Code of Ethics Is Approved*". Communications of the ACM, 42(10):102-107. October, 1999.

[Graham, 1994] Graham, Ian. "*Object-Oriented Methods*". 2nd edition. Addison-Wesley, 1994.

[Graham et al., 1997] Graham, Ian, Henderson-Sellers, Brian and Younessi, Houman. "*The Open Process Specification*". Addison Wesley (Open Series), 1997.

[Guttag, 1980] Guttag, J. "*Abstract Data Types and the Development of Data Structures*". In *Programming Language Design*. Computer Society Press, 1980.

[Hadjerrouit, 1999] Hadjerrouit, Said. "*A Constructivist Approach to Object-Oriented Design and Programming*". In Proceedings of the 4th annual SIGCSE/SIGCUE on Innovation and technology in computer science education (ITiCSE '99). (June 27-July 1, 1999, Cracow, Poland). Pages 171-174. ACM. 1999.

[Harbison, 1992] Harbison, Samuel P. "*Modula-3*". Prentice Hall, 1992.

[Hatley and Pirbhai, 1987] Hatley D. J. and Pirbhai, A. "*Strategies for Real-Time System Specification*". Dorset House, 1987.

[Henderson-Sellers and Edwards, 1994a] Henderson-Sellers, B. and Edwards, J. M. "*BOOKTWO of Object-Oriented Knowledge: The Working Object*". Prentice-Hall, 1994.

[Henderson-Sellers and Edwards, 1994b] Henderson-Sellers, B. and Edwards, J. M. "*MOSES: A Second Generation Object-Oriented Methodology*". Object Magazine, pp. 68-73. June, 1994.

[Henderson-Sellers et al., 1998] Henderson-Sellers, Brian, Simons, Anthony, Younessi, Houman. "*The Open Toolbox of Techniques*". Open Series. Addison Wesley, 1998.

[Hilburn, 1997] Hilburn, Thomas B. "*Software Engineering Education: A Modest Proposal*". IEEE Software, 14(6):44-48. November/December, 1997.

[Hilburn et al., 1998] Hilburn, Thomas B., Bagert, Donald J., Mengel, Susan and Oexmann, Dale. "*Software Engineering Across Computing Curricula*". In Proceedings of the 6th annual conference on the teaching of computing/3rd annual conference on integrating technology into computer science education on Changing the delivery of computer science education, ITiCSE '98. (Aug. 17-21, 1998, Dublin City Univ., Ireland). Pages 117-121. ACM. 1998.

[Hilburn et al., 1999] Hilburn, Thomas B., Hirmanpour, Iraj, Khajenoori, Soheil, Turner, Richard and Qasem, Abir. "*A Software Engineering Body of Knowledge Version 1.0*". Technical Report CMU/SEI-99-TR-004 (ESC-TR-99-004), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, PA 15213 (USA). April, 1999.

[Hirmanpour et al., 1995] Hirmanpour, Iraj, Hilburn, Thomas B. and Kornecki, Andrew. "*A Domain Centered Curriculum. An Alternative Approach to Computing Education*". In Proceedings of the 26th SISCSE technical symposium on Computer Science Education, SIGCSE '95. (March 2-4, 1995, Nashville, TN, USA). ACM. 1995.

[Hoare, 1985] Hoare, C. A. R. "*Communicating Sequential Processes*". Prentice-Hall, 1985.

[Holland et al., 1997] Holland, Simon, Griffiths, Robert and Woodman, Mark. "*Avoiding Object Misconceptions*". In Proceedings of the twenty-eighth SIGCSE technical symposium on Computer Science Education (SIGCSE'97). (Feb. 27-Mar. 1, 1997, San Jose, CA – USA). Pages 131-134. 1997.

[Horan, 1995] Horan, Peter "*Software Engineering - A Field Guide*". Deakin University. http://www.cm.deakin.edu.au/~peter/SEweb/field_gu.html**.** December 1995.

[Hullot, 1984] Hullot, Jean-Marie. "*Ceyx, Version 15: I — une Initiation*". Rapport Technique no. 44, INRIA, Rocquencourt, 1984.

[Humphrey, 1989] Humphrey, W. S. "*Managing the Software Process*". Addison-Wesley, 1989.

[Humphrey, 1993] Humphrey, W. S. "*Software Engineering*" in Ralston, A. and Reilly, E.D. (eds.), *Encyclopedia of Computer Science,* Van Nostrand Reinhold, p. 1218,1993.

[IEEE, 1983] IEEE. "*Standard Glossary of Software Engineering Terminology*". ANSI/IEEE Std. 729-1983. IEEE, 1983.

[IEEE, 1999] IEEE. "*IEEE Software Engineering Standards Collection 1999 Edition. Volume 1: Customer and Terminology Standards*". IEEE Computer Society Press, 1999.

[Ingalls, 1978] Ingalls, Daniel H. "*The Smalltalk-76 Programming System: Design and Implementation*". In Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages. ACM. January, 1978.

[ISO/IEC, 1995] ISO/IEC. "*Information Technology – Software Life Cycle Processes*". Technical ISO/IEC 12207:1995(E), 1995.

[ISO/IEC, 1998] ISO/IEC. "*Programming Languages – C++*". Technical ISO/IEC 14882. September, 1998.

[Jacobson et al., 1993] Jacobson, I., Christerson, M., Jonsson, P. and Övergaad, G. "*Object Oriented Software Engineering: A Use Case Driven Approach*". Addison-Wesley, 1992. Revised 4th printing, 1993.

[Jacobson et al., 1999] Jacobson, I., Booch, G. and Rumbaugh, J. "*The Unified Software Development Process*". Object Technology Series. Addison-Wesley, 1999.

[Jackson, 1975] Jackson, M. A. "*Principles of Program Design*". Academic Press, 1975.

[Jackson, 1983] Jackson, M. A. "*System Development*". Prentice-Hall, 1983.

[Jackson et al., 1997] Jackson, Ursula, Manaris, Bill and McCauley, Renée. "*Strategies for Effective Integration of Software Engineering Concepts and Techniques into the Undergraduate Computer Science Curriculum.*". In Proceedings of the twenty-eighth SIGCSE technical symposium on Computer Science Education, SIGCSE '97. (Feb. 27-Mar. 1, 1997, San José, CA). Pages 360-364. ACM. 1997.

[Jalics and Golden, 1995] Jalics, P. and Golden, D. "*A Profile of Undergraduate Computer Science Curricula*". Computer Science Education, 6(2):179-192. November, 1995.

[Jalloul, 1999] Jalloul, Ghinwa. "*Pedagogical Pattern #48. Academic to Industrial Project Link (LINK) Pattern*". Version 1.0. In [Proto-Patterns, 1999]. http://www-lifia.info.unlp.edu.ar/ppp/pp48.htm. [Última vez visitado, 20/8/1999]. July, 1999.

[Jaworski, 1998] Jaworski, Jaime. "*Java 1.2 Unleashed*". Sams, 1998.

[Jones, 1987] Jones, A. K. "*The Object Model: A Conceptual Tool for Structuring Software*". In Gerald E. Peterson, editor, *TUTORIAL: Object-Oriented Computing*, volume 2: Implementations. IEEE Computer Society Press, 1987.

[Jones, 1990] Jones, C. B. "*Systematic Software Construction Using VDM*". Prentice-Hall, 1990.

[Joyner, 1996] Joyner, Ian. "*C++?? A Critique of C++ and Programming and Language Trends of the 1990s*". 3rd edition http://www.progsoc.uts.edu.au/~geldridg/cpp/cppcv3/cppcv3.pdf. [Última vez visitado 7/1/2000]. 1996.

[Joyner, 1999] Joyner, Ian. "*Objects Unencapsulated. Eiffel, Java and C++??*". Object and Component Technology Series, Prentice-Hall, 1999.

[Knight et al., 1994] Knight, John C., Prey, Jane C. and Wulf, Wm. A. "*Undergraduate Computer Science Education: A New Curriculum Philosophy & Overview*". In Proceedings of the twenty-fifth annual SIGCSE symposium on Computer science education, SIGCSE '94. (March 10-11, 1994, Phoenix, AZ - USA). Pages 155-159. ACM. 1994.

[Kobryn, 1999] Kobryn, Cris. "*UML 2001: A Standardization Odyssey*". Communications of the ACM, 42(10):29-37. October, 1999.

[Koffman et al., 1984] Koffman, E., Miller, P. and Wardle, C. "*Recommended Curriculum for CS1: 1984*". Communications of the ACM, 27(10):998-1001. October, 1984.

[Koffman et al., 1985] Koffman, E., Miller, P. and Wardle, C. "*Recommended Curriculum for CS2: 1984*". Communications of the ACM, 28(8):815-818. August, 1985.

[Konrad et al., 1995] Konrad, Michael D., Paulk, Mark C., and Graydon, Allan W. "*An Overview of SPICE's Model for Process Management*". In Proceedings of the Fifth International Conference on Software Quality, Austin, TX, 23-26 October 1995.

[Krasner and Pope, 1988] Krasner, G. E. and Pope, S. T. "*A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*". Journal of Object-Oriented Programming (JOOP), SIGS Publications, 1(3):26-49. August-September, 1988.

[Lalonde and Pugh, 1990] Lalonde, Wilf R. and Pugh, John R. "*Inside Smalltalk*". Vol. 1. Prentice-Hall, 1990.

[Lalonde and Pugh, 1991] Lalonde, Wilf R. and Pugh, John R. "*Inside Smalltalk*". Vol. 2. Prentice-Hall, 1991.

[Larman, 1998] Larman, Craig. "*Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design*". Prentice Hall, 1998.

[Layman, 1994] Layman, B. "*ISO-9000 Standards and Existing Quality Models: How They Relate*". American Programmer Review, pp. 9-15. February, 1994.

[Le Moigne, 1973] Le Moigne, J. L. "*Les Systemes D'Information dan les Organizations*". Presses Universitaires de France, 1973.

[Lebsanft and Synspace, 1994] Lebsanft, E. and Synspace, A. G. "*BOOTSTRAP: Experiences with Europe's Software Process Assesment & Improvement Method*". In Proceedings of the 1st World Congress for Software Quality, 1994.

[Levine et al., 1991] Levine, Linda, Pesante, Linda H. and Dunkle, Susan B. "*Technical Writing for Software Engineers*". Curriculum Module, SEI-CM-23, Software Engineering Institute. Carnegie Mellon University, Pittsburgh, PA 15213 (USA). November, 1991.

[Levy, 1984] Levy, H. "*Capability-Based Computer Systems*". Digital Press, 1984.

[Liskov and Zilles, 1977] Liskov, B. and Zilles, S. "*An Introduction to Formal Specifications of Data Abstractions*". Current Trends in Programming Methodology: Software Specification and Design. Vol. 1. Prentice-Hall, 1977.

[Longenecker and Feinstein, 1991] Longenecker, Jr. Herbert E. and Feinstein, David L. (Editors) "*IS'90 The DPMA Model Curriculum for a Four Year Undergraduate Degree for the 1990s*". DPMA Data Processing Management Association Model Curricula for the 1990s. 505 Busee Highway, Park Ridge, IL. 60068. 1991.

[Longenecker et al., 1994] Longenecker, Jr. Herbert E., Fournier, Robert, Reaugh, William R. and Feinstein David L. (Editors) "*IS'94 The DPMA Two Year Model Curriculum for IS Professionals*". DPMA Data Processing Management Association Model Curricula for the 1990s. 505 Busee Highway, Park Ridge, IL. 60068. 1994.

[Madsen et al., 1993] Madsen, Ole Lehrmann, Møller-Pedersen, Birger, Nygaard, Kristen. "*Object-Oriented Programming in the BETA Programming Language*". Addison-Wesley, Wokingham (U.K.), 1993.

[Manns, 1999] Manns, Mary Lynn. "*Pedagogical Pattern #8. Lab-Discussion-Lecture-Lab (LDLL) Pattern*". Version 1.0. In [Proto-Patterns, 1999]. http://www-lifia.info.unlp.edu.ar/ppp/pp8.htm. [Última vez visitado, 20/8/1999]. July, 1999.

[MAP, 1995] Ministerio de las Administraciones Públicas. "*Metodología Métrica 2.1*". Volúmenes 1-3. Editorial Tecnos, 1995.

[Marqués, 1995] Marqués Corral, José Manuel. "*Jerarquías de Herencia en el Diseño de Software Orientado al Objeto*". Tesis Doctoral. Facultad de Ciencias, Universidad de Valladolid, 1995.

[Meyer, 1992] Meyer, Bertrand. "*Eiffel: The Language*". Prentice Hall Object-Oriented Series, 1991; second revised printing, 1992.

[Meyer, 1996] Meyer, Bertrand. "*Teaching Object Technology*". IEEE Computer, 29(12):117. December, 1996.

[Meyer, 1997] Meyer, Bertrand. "*Object Oriented Software Construction*". 2nd Edition. Prentice Hall, 1997.

[Miller and Mingins, 1998] Miller, Jan and Mingins, Christine. "*Putting the Practice into Software Education*". In Proceedings oh the 1998 International Conference on Software Engineering: Education and Practice (SEEP'98). (26-29 January 1998, Dunedin – New Zeland). Pages, 200-208. IEEE Computer Society. 1998.

[Mohedano, 1998] Mohedano, José Eduardo. "*Java en la Historia: El Azar Mueve el Mundo*". Sólo Programadores. Especial Monográfico Nº3: 6-10. Octubre, 1998.

[Naughton, 1996] Naughton, Patrick. "*The Java Handbook*". McGraw-Hill, 1996.

[Naur and Randell, 1969] Naur, P. and Randell, B. (Editors). "*Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 October 1968*". Brussels: Scientific Affairs Division, NATO. January, 1969.

[Norman, 1988] Norman, D. A. "*The Psychology of Everyday Things*". New York: Basic Books, Inc. 1988.

[Northrop, 1992] Northrop, Linda M. "*Finding an Educational Perspective for Object-Oriented Development*". In addendum to the proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum) - OOPSLA '92. (Oct. 18-22, 1992, Vancouver, British Columbia, Canada). Pages 245-249. ACM. 1992.

[Nunamaker et al., 1982] Nunamaker, Jay F., Couger, J. D. and Davis, Gordon B. "*Information System Curriculum Recommendations for the 80s: Undergraduate and Graduate Programs*". Communications of the ACM 25(11). November, 1982.

[Nygaard and Dahl, 1981] Nygaard, Kristen and Dahl, Ole-Johan. "*The Development of the SIMULA Language*". In *History of Programming Languages*, Richard L. Wexelblat editor. Pages 439-493. Academic Press, 1981.

[OMG, 1998] OMG. "*OMG Unified Modeling Language Specification. Version 1.2*". Object Management Group Inc. ftp://ftp.omg.org/pub/docs/ad/98-12-02-pdf. July, 1998.

[OMG, 1999] OMG. "*OMG Unified Modeling Language Specification. Version 1.3*". Object Management Group Inc. http://uml.shl.com:80/docs/UML1.3/99-06-08-pdf. June, 1999.

[Orr, 1977] Orr, K. T. "*Structured System Development*". Yourdon Press, 1977.

[Orr, 1981] Orr, K. T. "*Structured Requirements Definition*". Ken Orr & Associates, 1981.

[Osborne, 1992] Osborne, Martin. "*The Role of Object-Oriented Technology in the Undergraduate Computer Science Curriculum*". In addendum to the proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum) - OOPSLA '92. (Oct. 18-22, 1992, Vancouver, British Columbia, Canada). Pages 303-308. ACM. 1992.

[Paepcke, 1993] Andreas Paepcke (editor). "*Object-Oriented Programming: The CLOS Perspective*". MIT Press, Cambridge (Mass.), 1993.

[Pancake, 1995] Pancake, Cherri M. "*The Promise and the Cost of Object Technology: A Five Years Forecast*". Communications of the ACM 38(10):32-49. October, 1995.

[Parnas, 1972] Parnas, David L. "*On the Criteria To Be Used in Descomposing Systems into Modules*". Communications of the ACM, 15(12):1053-1058. December, 1972.

[Parnas and Weiss, 1987] Parnas, David Lorge and Weiss, D. M. "*Active Design Reviews: Principles and Practices*". Journal of Systems and Software, 7(4): 259-265, December 1987.

[Parrish et al., 1998] Parrish, A., Borie, R., Cordes, D., Dixon, B., Hale, D., Hale, J., Jackson, J. and Sharpe, S. "*Computer Engineering, Computer Science and Management Information Systems: Partners in a Unified Software Engineering Curriculum*". In Proceedings of the 11th Conference on Software Engineering Education & Training – CSEET'98. (February 22-25, 1998. Atlanta, GA – USA). Pages 67-75. IEEE Computer Society. 1998.

[Paulk et al., 1993a] Paulk, Mark C., Curtis, Bill, Chrissis, Mary Beth and Weber, Charles V. "*Capability Maturity Model for Software, Version 1.1*" Technical Report CMU/SEI-93-

TR-24, Software Engineering Institute. Carnegie Mellon University, Pittsburgh, PA 15213 (USA), February 1993.

[Paulk et al., 1993b] Paulk, Mark C., Curtis, Bill, Chrissis, Mary Beth and Weber, Charles V. "*Capability Maturity Model, Version 1.1*". IEEE Software, 10(4):18-27. July, 1993.

[Pham, 1997] Pham, Binh. "*The Changing Curriculum of Computing and Information Technology in Australia*". In Proceedings of the second Australasian conference on Computer science education, ACSE '97. (July 2-4, 1997, The Univ. of Melbourne, Australia). ACM. 1997.

[Piattini, 1994] Piattini Velthuis, Mario G. "*Definición de una Metodología para el Desarrollo de Bases de Datos Orientadas al Objeto Fundamentadas en Extensiones del Modelo Relacional*". Tesis Doctoral. Facultad de Informática, Universidad Politécnica de Madrid. 1994.

[Piattini et al., 1996] Piattini Velthuis, Mario G., Calvo-Manzano, José A., Cervera, Joaquín y Fernández, Luis. "*Análisis y Diseño Detallado de Aplicaciones Informáticas de Gestión*". Ra-ma. 1996.

[Pressman, 1987] Pressman, Roger S. "*Software Engineering: A Practitioner's Approach*". 2nd edition. McGraw Hill, 1987.

[Pressman, 1992] Pressman, Roger S. "*Software Engineering. A Practitioner's Approach*". 3rd Edition. McGraw Hill, 1992.

[Pressman, 1997] Pressman, Roger S. "*Software Engineering: A Practitioner's Approach*". 4th Edition. McGraw Hill, 1997.

[Prieto and Victory, 1999] Prieto, Máximo and Victory, Pablo. "*Pedagogical Pattern #20. Identity Pattern*". Version 1.0. http://www-lifia.info.unlp.edu.ar/ppp/pp20.htm. [Última vez visitado, 20/8/1999]. July, 1999.

[Proto-Patterns, 1999] "*The Pedagogical Patterns Project. Successes in Teaching Object-Technology (PROTO-PATTERNS)*". http://www-lifia.info.unlp.edu.ar/ppp/index.html. [Última vez visitado, 20/8/1999]. July, 1999.

[Ramamoorthy and Sheu, 1988] Ramamoorthy, C. and Sheu, P. "*Object-Oriented Systems*". IEEE Expert, 3(3). Fall, 1988.

[Rand, 1979] Rand, Ayn. "*Introduction to Objectivist Epistemology*". New American Library, 1979.

[Rans, 1999] Rans, Michael. "*A History of Object-Oriented Programming Languages and their Impact on Program Design and Software Development*". http://users.ox.ac.uk/~ball0370/documents/oo.pdf [Última vez visitado, 22/12/1999]. November, 1999.

[Randell, 1998] Randell, Brian. "*Memories of the NATO Software Engineering Conference*". In *Anecdotes Column*, James E. Tomayko (Editor). IEEE Annals of the History of Computing, 20(1):51-54. January-March, 1998.

[Rational et al., 1997] Rational Software Corporation, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing IntelliCorp, i-Logix, IBM, ObjecTime. Platinum Technology, Ptech, Taskon, Reich Technologies and Softeam. "*UML Proposal to the Object Management. In Response to the OA&D Task Force's RFP-1*". UML 1.1 Referece Set 1.1. 1 September 1997.

[Redwine, 1985] Redwine, S. T. "*Software Technology Maduration*". In Proceedings of the 1st International Conference on Software Engineering, Washington D. C. (USA). IEEE, 1985.

[Reenskaug et al., 1996] Reenskaug, Trygve, Wold, Per and Lehne, Odd Arild. "*Working with Objects. The OOram Software Engineering Method*". Manning Publications Co./Prentice Hall, 1996.

[Reynolds and Fox, 1996] Reynolds, Charles and Fox, Christopher. "*Requirements for a Computer Science Curriculum Emphasizing Information Technology Subject Area: Curriculum Issues*". In Proceedings of the twenty-seventh SIGCSE technical symposium on Computer Science Education - SIGCSE '96. (Feb. 15-18, 1996, Philadelphia, PA, USA). Pages 247-251. ACM. 1996.

[Rivas et al., 1997] Rivas, Erick, DeSilva, Dilhar, McDaniel, Terrie and Atkinson, Colin. "*Object Analysis and Design Facility*". Response to OMG/OA&D RFP-1. Version 1.0. Platinum Technology, Inc. January, 1997.

[Roberts et al., 1999] Roberts, Eric, Shackelford, Russ, LeBlanc, Rich and Denning, Peter J. "*Curriculum 2001: Interim Report from the ACM/IEEE-CS Task Force*". In Proceedings of the thirtieth SIGCSE technical symposium on Computer science education, SIGCSE '99. (March 24-28, 1999, New Orleans, LA - USA). Pages 343-344. ACM. 1999.

[Rosson and Carroll, 1996] Rosson, Mary Beth and Carroll, John M. "*Scaffolded Examples for Learning Object-Oriented Design*". Communications of the ACM, 39(4):46-47. April, 1996.

[Rumbaugh, 1996] Rumbaugh, James. "*OMT Insights. Perspectives on Modeling from the Journal of Object-Oriented Programming*". SIGS Books Publications, 1996.

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

[Rumbaugh et al., 1999] Rumbaugh, James, Jacobson, Ivar and Booch, Grady. "*The Unified Modeling Language Reference Manual*". Object Technology Series. Addison-Wesley, 1999.

[Sawyer and Kotonya, 1999] Sawyer, Pete and Kotonya, Gerald. "*SWEBOK: Software Requirements Engineering Knowledge Area Description*". In [Abran et al., 1999], 1999.

[Schmauch, 1994] Schmauch, C. "*ISO9000 for Software Developers*". IEEE Computer Society Press, 1994.

[Schmucker, 1986] Schmucker, K. "*Object-Oriented Language for the Macintosh*". Byte, 11(8). August, 1986.

[Scragg et al., 1994] Scragg, Greg, Baldwin, Doug and Koomen, Hans. "*Computer Science Needs and Insight-Based Curriculum*". In Proceedings of the twenty-fifth annual SIGCSE symposium on Computer science education, SIGCSE '94. (March 10-11, 1994, Phoenix, AZ - USA). Pages 150-154. ACM. 1994.

[Sernadas et al., 1989] Sernadas, A., Fiadeiro, J., Sernadas, C. and Eric, H.-D. "*The Basic Building Blocks of Information Systems*". In *Information Systems Concepts*. North Holland, Namur, 1989.

[Shackelford and LeBlanc, 1994] Shackelford, Russell L. and LeBlanc, Richard J. "*Integrating 'Depth First' and 'Breadth First' Models of Computing Curricula*". In Proceedings of the twenty-fifth annual SIGCSE symposium on Computer science

education, SIGCSE '94. (March 10-11, 1994, Phoenix, AZ - USA). Pages 6-10. ACM. 1994.

[Sharp et al., 2000] Sharp, Helen, Robinson, Hugh and Woodman, Mark. "*Software Engineering: Community and Culture*". IEEE Software, 17(1):40-47. January/February, 2000.

[Shaw, 1984] Shaw, Mary. "*Abstraction Techniques in Modern Programming Languages*". IEEE Software, 1(4). October, 1984.

[Shaw, 1985] Shaw, Mary (editor). "*The Carnegie-Mellon Curriculum for Undergraduate Computer Science*". Springer-Verlag, 1985.

[Shaw, 1990] Shaw, Mary. "*Prospects for an Engineering Discipline of Software*". IEEE Software, 7(6):15-24. November, 1990.

[Shaw and Garlan, 1996] Shaw, Mary and Garlan, David. "*Software Architecture: Perspectives on a Emerging Discipline*". Prentice-Hall, 1996.

[Shaw and Tomayko, 1991] Shaw, Mary and Tomayko, James E. "*Models for Undergraduate Project Courses in Software Engineering*". Technical Report CMU/SEI-91-TR-10 (ESD-91-TR-10), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, PA 15213 (USA). August, 1991.

[Shlaer and Mellor, 1992] Shlaer, S. and Mellor, S. "*Object Lifecycles: Modeling the World in States*". Yourdon Press, 1992.

[SIS, 1987] SIS. "*Data Processing - Programming Languages — SIMULA*". Standardiseringskommissionen i Sverige (Swedish Standards Institute), Svensk Standard SS 63 61 14, 20 May, 1987.

[Smith and Tockey, 1988] Smith, M. and Tockey, S. "*An Integrated Approach to Software Requirements Definition Using Objects*". Seattle, WA: Boeing Commercial Airplane Support Division, 1988.

[Sommerville, 1985] Sommerville, Ian. "*Software Engineering*". 2nd edition. Addison-Wesley, 1985.

[Sommerville, 1989] Sommerville, Ian. "*Software Engineering*". 3rd edition. Addison-Wesley, 1989.

[Sommerville, 1996] Sommerville, Ian. "*Software Engineering*". 5th edition. Addison-Wesley, 1996.

[Spivey, 1989] Spivey, J. M. "*The Z Notation. A Reference Manual*". International Series in Computer Science. Prentice-Hall International, 1989.

[Stillings et al., 1987] Stillings, N., Feinstein, M., Garfield, J., Rissland, E., Rosenbaum, D., Weisler, S. and Baker-Ward, L. "*Cognitive Science: An Introduction*". The MIT Press, 1987.

[Stroustrup, 1986] Stroustrup, Bjarne. "*The C++ Programming Language*". 1st Edition, Addison Wesley, 1986.

[Stroustrup, 1986b] Stroustrup, Bjarne. "*What Is Object-Oriented Programming?*". In Proceedings of 14th ASU Conference. August 1986.

[Stroustrup, 1991] Stroustrup, Bjarne. "*The C++ Programming Language*". 2nd Edition, Addison Wesley, 1991.

[Stroustrup, 1994] Stroustrup, Bjarne. "*The Design and Evolution of C++*". Addison Wesley, 1994.

[Stroustrup, 1997] Stroustrup, Bjarne. "*The C++ Programming Language*". 3rd Edition, Addison Wesley, 1997.

[SUN, 2000] SUN Microsystems. "*The Java Tutorial. A Practical Guide for Programmers*". http://java.sun.com/docs/books/tutorial/index.html. [Última vez visitado, 16/3/2000]. February, 2000.

[Sutcliffe and Maiden, 1998] Sutcliffe, Alistair and Maiden, Neil. "*The Domain Theory for Requirements Engineering*". IEEE Transactions on Software Engineering, 24(3): 174-196. March, 1998.

[Tewari and Friedman, 1992] Tewari, Raj and Friedman, Frank. "*The Impact of Object-Oriented Software Engineering in the Introductory Computer Science Curriculum*". In addendum to the proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum) - OOPSLA '92. (Oct. 18-22, 1992, Vancouver, British Columbia, Canada). Pages 289-292. ACM. 1992.

[Tomayko, 1987] Tomayko, James E. "*Teaching a Project-Intensive Introduction to Software Engineering*". Technical Report CMU/SEI-87-TR-20 (ESD-TR-87-171), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, PA 15213 (USA). August, 1987.

[Tomayko, 2000] Tomayko, James E. "*A Historian's View of Software Engineering*". In Proceedings of the Thirteenth Conference on Software Engineering and Training. (6-8 March, 2000. Austin, Texas (USA)). Pages 101-108. IEEE Press, 2000.

[Tremblay, 1999] Tremblay, Guy. "*Knowledge Area Description for Design (version 0.5)*". In [Abran et al., 1999], 1999.

[Tucker and Wegner, 1994] Tucker, Allen B. and Wegner, Peter. "*New Directions in the Introductory Computer Science Curriculum*". In Proceedings of the twenty-fifth annual SIGCSE symposium on Computer science education, SIGCSE '94. (March 10-11, 1994, Phoenix, AZ - USA). Pages 11-15. ACM. 1994.

[Tucker et al., 1991a] Tucker, Allen B., Barnes, Bruce H., Aiken, Robert M., Barker, Keith, Bruce, Kim B., Cain, J. Thomas, Conry, Susan E., Engel, Gerald L., Epstein, Richard G., Lidtke, Doris K., Mulder, Michael C., Rogers, Jean B., Spafford, Eugene H. and Turner, A. Joe. "*Computing Curricula 1991*". ACM Press. February, 1991.

[Tucker et al., 1991b] Tucker, Allen B., Barnes, Bruce H., Aiken, Robert M., Barker, Keith, Bruce, Kim B., Cain, J. Thomas, Conry, Susan E., Engel, Gerald L., Epstein, Richard G., Lidtke, Doris K., Mulder, Michael C., Rogers, Jean B., Spafford, Eugene H. and Turner, A. Joe. "*A Summary of the ACM/IEEE-CS Joint Curriculum Task Force Report. Computing Curricula 1991*". Communications of the ACM, 34(6):68-84. June, 1991.

[Tucker et al., 1996] Tucker, Allen B., Astrachan, Owen, Bruce, Kim, Cupper, Robert, Denning, Peter, Drysdale, Scot, Horton, Tom, Kelemen, Charles, McGeoch, Cathy, Patt, Yale, Proulx, Viera, Rada, Roy, Rasala, Richard, Roberts, Eric, Rudich, Steven, Stein, Lynn, Van Loan, Charles. "*Strategic Directions in Computer Science Education*". ACM Computing Surveys, 28(4):836-845. December, 1996.

[Tymann et al., 1994] Tymann, Paul T., Lea, Douglas and Raj, Rajendra K. "*Developing an Undergraduate Software Engineering Program in a Liberal Arts College*". In Proceedings of the twenty-fifth annual SIGCSE symposium on Computer Science

Education (SIGCSE '94). (March 10-11, 1994, Phoenix, AZ – USA). Pages 276-280. ACM. 1994.

[Ungar et al., 1992] Ungar, David, Smith, Randall B., Chambers, Craig and Hölzle, Urs. "*Object, Message and Performance: How they Coexist in Self*". IEEE Computer 25(10): 53-64. October, 1992.

[USAL, 1998] Universidad de Salamanca. "*Guía Académica Curso 1998-1999. Facultad de Ciencias*". Facultad de Ciencias - Universidad de Salamanca. 1998.

[USAL, 1999] Universidad de Salamanca. "*Guía Académica Curso 1999-2000. Facultad de Ciencias*". Facultad de Ciencias - Universidad de Salamanca. 1999.

[Vaitkevitchius, 1999] Vaitkevitchius, Raimundas. "*Pedagogical Pattern #25. BASE-and-Supplementary-Languages in Lectures (BSLL)*". In [Proto-Patterns, 1999]. http://www-lifia.info.unlp.edu.ar/ppp/pp25.htm. [Última vez visitado, 20/8/1999]. July, 1999.

[Vaughn, 2000] Vaughn, Rayford B. Jr. "*Software Engineering Degree Programs*". Crosstalk, The Journal of Defense Software Engineering, 13(3):7-9. March, 2000.

[Walker and Schneider, 1996] Walker, Henry M. and Schneider, G. Michael. "*A Revised Model Curriculum for a Liberal Arts Degree in Computer Science*". Communications of the ACM, 39(12):85-95. December, 1996.

[Wand, 1989] Wand, Y. "*How to Integrate Object Orientation with Structured Analysis and Design*". IEEE Software, 6(2). March, 1989.

[Ward and Mellor, 1985] Ward, P. and Mellor, S. "*Structured Development for Real-Time Systems*". Vols. 1-3. Yourdon Press, 1985.

[Warnier, 1974] Warnier, J. "*Logical Construction of Programs*". Van Nostrand Reinhold, 1974.

[Wegner, 1990] Wegner, Peter. "*Concepts and Paradigms of Object-Oriented Programming*". OOPS Messenger, 1(1). August, 1990.

[Weinberg, 1971] Weinberg, G. F. "*The Psychology of Computer Programming*". Van Nostrand Reinhold, 1971.

[Wielinga et al. 1991] Wielinga, B. J., Schreiber, A. T. and Breuker, J. A. "*KADS: A Modeling Approach to Knowledge Engineering*". Technical Report ESPRIT Project P5248 KAD-II, 1991.

[Wirfs-Brock and Johnson, 1990] Wirfs-Brock, Rebecca and Johnson, Ralph E. "*Surveying Current Research in Object-Oriented Design*". Communications of the ACM, 33(9):104-124. September, 1990.

[Wirfs-Brock et al., 1990] Wirfs-Brock, Rebecca, Wilkerson, Brian and Wiener, Lauren. "*Designing Object-Oriented Software*". Prentice-Hall. 1990.

[Wirth and Reiser, 1992] Wirth, Niklaus and Reiser, Martin. "*Programming in Oberon — Steps Beyond Pascal and Modula*". Addison-Wesley, Reading (Mass.), 1992.

[Woodman et al., 1996] Woodman, Mark, Davies, Gordon and Holland, Simon. "*The Joy of Software – Starting with Objects*". In Proceedings of the twenty-seventh SIGCSE technical symposium on Computer Science Education - SIGCSE '96. (Feb. 15-18, 1996, Philadelphia, PA, USA). Pages 88-92. ACM. 1996.

[X3J16/WG21, 1996] ANSI X3J16 and ISO WG21. "*Working Paper for Draft Proposed International Standard for Information Systems—Programming Language C++*".

ftp://research.att.com/dist/c++std/WP/CD2. [Última vez visitado 7/1/2000]. X3J16/96–0225 (WG21/N1043). December, 1996.

[Yonezawa and Tokoro, 1987] Yonezawa, A. and Tokoro, M. "*Object-Oriented Concurrent Programming: An Introduction*". In *Object-Oriented Concurrent Programming*. Cambridge, MA: The MIT Press. 1987.

[Yourdon, 1989] Yourdon, Edward. "*Modern Structured Analysis*". Prentice Hall, 1989.

[Yourdon Inc., 1993] Yourdon Inc. "*Yourdon™ Systems Method. Model-Driven Systems Development*". Prentice Hall International Editions. 1993.

[Yourdon and Constantine, 1975] Yordon, E. and Constantine, L. "*Structured Design*". 1st edition. Prentice Hall, 1975.

[Yourdon and Constantine, 1979] Yordon, E. and Constantine, L. "*Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*". Yourdon Press, 1979.

[Yourdon and Constantine, 1989] Yordon, E. and Constantine, L. "*Structured Design*". 2nd edition. Yourdon Press, 1989.