

Calidad en la Industria del Software. La Norma ISO-9126

María Antonieta Abud Figueroa*

Introducción

Hoy en día las compañías de todo el mundo industrializado reconocen que la calidad del producto se traduce en ahorro de costos y en una mejora general. La industria de desarrollo de software no es la excepción, por lo que en los últimos años se han realizado intensos trabajos para aplicar los conceptos de calidad en el ámbito del software.

Hablar de calidad del software implica la necesidad de contar con parámetros que permitan establecer los niveles mínimos que un producto de este tipo debe alcanzar para que se considere de calidad. El problema es que la mayoría de las características que definen al software no se pueden cuantificar fácilmente; generalmente, se establecen de forma cualitativa, lo que dificulta su medición, ya que se requiere establecer métricas que permitan evaluar cuantitativamente cada característica dependiendo del tipo de software que se pretende calificar.

En este sentido se han realizado muchos trabajos que establecen propuestas para el establecimiento de los factores cualitativos que afectan la calidad del software. Entre los principales están los factores de calidad de McCall [1][4] y aquellos propuestos por Hewlett-Packard (FURPS: Funcionality,

Usability, Reliability; Performance, Supportability) [4].

Además se han hecho varios intentos por estandarizar los mecanismos de evaluación de calidad del software. Entre los principales están la familia de normas ISO 9000 (en especial la ISO 9001 y la ISO 9003-2)[5], el modelo de niveles madurez CMM (Capability Maturity Model)[7], el estándar para el aseguramiento de planes de calidad del IEEE 730:1984 [7], el plan general de garantía de calidad del Consejo Superior de Informática MAP[7] y la norma ISO/IEC 9126 [3], que es objeto de este estudio.

En este trabajo se expone un esquema general del estándar ISO 9126, con el fin de mostrar los elementos que deben considerarse en la evaluación de calidad de los productos de software de acuerdo a este estándar, de modo que todo aquel que se interese

en aplicar modelos de calidad en la producción de software pueda generar sus propias métricas bajo la guía de este estándar.

Modelo de Calidad Establecido por el estándar ISO 9126

La ISO, bajo la norma ISO-9126, ha establecido un estándar internacional para la evaluación de la calidad de productos de software el cual fue publicado en 1992 con el nombre de “*Information technology –Software product evaluation: Quality*

characteristics and guidelines for their use”, en el cual se establecen las características de calidad para productos de software.

El estándar ISO-9126[7] establece que cualquier componente de la calidad del software puede ser descrito en términos de una o más de seis características básicas, las cuales son: funcionalidad, confiabilidad, usabilidad, eficiencia, mantenibilidad y portatilidad; cada una de las cuales se detalla a través de un conjunto de subcaracterísticas que permiten profundizar en la evaluación de la calidad de productos de software. La tabla 1 muestra la pregunta central que atiende cada una de estas características.

Características	Pregunta central
Funcionalidad	¿Las funciones y propiedades satisfacen las necesidades explícitas e implícitas; esto es, el qué . . . ?
Confiabilidad	¿Puede mantener el nivel de rendimiento, bajo ciertas condiciones y por cierto tiempo?
Usabilidad	¿El software es fácil de usar y de aprender?
Eficiencia	¿Es rápido y minimalista en cuanto al uso de recursos?
Mantenibilidad	¿Es fácil de modificar y verificar?
Portatilidad	¿Es fácil de transferir de un ambiente a otro?

Tabla 1. Características de ISO-9126 y aspecto que atiende cada una.

Características Propuestas por ISO-9126

A continuación se detalla cada una de las características que establece el estándar ISO-9126.

C1. Funcionalidad

En este grupo se conjunta una serie de atributos que permiten calificar si un producto de software maneja en forma adecuada el conjunto de funciones que satisfagan las necesidades para las cuales fue diseñado. Para este propósito se establecen los siguientes atributos:

*Maestra en Sistemas de Información por el Instituto Tecnológico y de Estudios Superiores de Monterrey-Campus Morelos. Profesor-Investigador en la División de Estudios de Posgrado e Investigación del Instituto Tecnológico de Orizaba. Correo electrónico: mabud@itorizaba.edu.mx.

- **Adecuación.** Se enfoca a evaluar si el software cuenta con un conjunto de funciones apropiadas para efectuar las tareas que fueron especificadas en su definición.
- **Exactitud.** Este atributo permite evaluar si el software presenta resultados o efectos acordes a las necesidades para las cuales fue creado.
- **Interoperabilidad.** Permite evaluar la habilidad del software de interactuar con otros sistemas previamente especificados.
- **Conformidad.** Evalúa si el software se adhiere a estándares, convenciones o regulaciones en leyes y prescripciones similares.
- **Seguridad.** Se refiere a la habilidad de prevenir el acceso no autorizado, ya sea accidental o premeditado, a los programas y datos.

C2. Confiabilidad

Aquí se agrupan un conjunto de atributos que se refieren a la capacidad del software de mantener su nivel de ejecución bajo condiciones normales en un periodo de tiempo establecido. Las subcaracterísticas que el estándar sugiere son:

- **Nivel de Madurez.** Permite medir la frecuencia de falla por errores en el software.
- **Tolerancia a fallas.** Se refiere a la habilidad de mantener un nivel específico de funcionamiento en caso de fallas del software o de cometer infracciones de su interfaz específica.
- **Recuperación.** Se refiere a la capacidad de restablecer el nivel de operación y recobrar los datos que hayan sido afectados directamente por una falla, así como al tiempo y el esfuerzo necesarios para lograrlo.

C3. Usabilidad

Consiste de un conjunto de atributos que permiten evaluar el esfuerzo necesario que deberá invertir el usuario para utilizar el sistema.

- **Comprensibilidad.** Se refiere al esfuerzo requerido por los usuarios para reconocer la estructura lógica del sistema y los conceptos relativos a la aplicación del software.
- **Facilidad de Aprender.** Establece atributos del software relativos al esfuerzo que los usuarios deben hacer para aprender a usar la aplicación.
- **Operabilidad.** Agrupa los conceptos que evalúan la operación y el control del sistema.

C4. Eficiencia

Esta característica permite evaluar la relación entre el nivel de funcionamiento del software y la cantidad de recursos usados. Los aspectos a evaluar son:

- **Comportamiento con respecto al Tiempo.** Atributos del software relativos a los tiempos de respuesta y de procesamiento de los datos.
- **Comportamiento con respecto a Recursos.** Atributos del software relativos a la cantidad de recursos usados y la duración de su uso en la realización de sus funciones.

C5. Mantenibilidad

Se refiere a los atributos que permiten medir el esfuerzo necesario para realizar modificaciones al software, ya sea por la corrección de errores o por el incremento de funcionalidad. En este caso, se tienen los siguientes factores:

- **Capacidad de análisis.** Relativo al esfuerzo necesario para diagnosticar las deficiencias o causas de fallas, o para identificar las partes que deberán ser modificadas.
- **Capacidad de modificación.** Mide el esfuerzo necesario para modificar aspectos del software, remover fallas o adaptar el software para que funcione en un ambiente diferente.
- **Estabilidad.** Permite evaluar los riesgos de efectos inesperados

debidos a las modificaciones realizadas al software.

- **Facilidad de Prueba.** Se refiere al esfuerzo necesario para validar el software una vez que fue modificado.

C6. Portatibilidad

En este caso, se refiere a la habilidad del software de ser transferido de un ambiente a otro, y considera los siguientes aspectos:

- **Adaptabilidad.** Evalúa la oportunidad para adaptar el software a diferentes ambientes sin necesidad de aplicarle modificaciones.
- **Facilidad de Instalación.** Es el esfuerzo necesario para instalar el software en un ambiente determinado.
- **Conformidad.** Permite evaluar si el software se adhiere a estándares o convenciones relativas a portatibilidad.
- **Capacidad de reemplazo.** Se refiere a la oportunidad y el esfuerzo usado en sustituir el software por otro producto con funciones similares.


Conclusiones

El mundo globalizado exige cada vez más la aplicación de estándares internacionales que garanticen la calidad de los productos. Por esta razón, es necesario que todo aquel que se dedica al desarrollo de software incluya en sus procesos, estándares de calidad que permitan certificarse en alguno de los modelos.

Aquí se ha presentado un estándar, el ISO-9126, el cual establece una guía para la evaluación de la calidad del software, sin embargo es necesario que cada empresa dedicada a producir software trabaje en establecer su modelo de calidad que le permita valorar el nivel de excelencia de sus productos, en el que deberán incluirse instrumentos de medición que permitan calificar cuantitativamente cada una de las características aquí

presentadas. Es importante mencionar, que dependiendo de los distintos tipos de aplicaciones las

métricas podrán variar, ya que aunque las características expuestas son comunes a la totalidad de

los productos, cada software particular requiere una evaluación específica. 

Bibliografía

[1] Cervera Paz, Ángel. *El modelo de McCall como aplicación de la calidad a la revisión del software de gestión empresarial*. Universidad de Cádiz, obtenido el 24 de abril del 2001 del sitio web <http://www.monografias.com/trabajos5/call/call.html>

[2] Fairley, Richard. *Ingeniería de Software*, 2ª Edición. Editorial Mc Graw Hill. México. 1987.

[3] International Organization for Standardization. *Norma de gestión de la calidad y garantía de la calidad, parte 3*. Sitio web <http://alarcos.inf-cr.uclm.es/doc/calidad/ISO%209000-3.doc>.

[4] Pressman, Roger S. *Ingeniería de Software, Un enfoque práctico*, 4ª. Edición. Editorial Mc Graw Hill. México. 1998.

[5] Quintanilla Osorio, Gloria. “La implantación de ISO 9001 en el desarrollo de software”, *Revista Soluciones Avanzadas*, Septiembre (1999), p.31.

[6] Rodríguez G., González J., Dávila Gladys. “La norma ISO 9001 en una fábrica de software a la medida”, *Revista Soluciones Avanzadas*, julio (1998), p.27.

[7] Sanders, Joc & Eugene Curran. *Software Quality. A Framework for Success in Software Development and Support*, Addison Wesley.





Metodología de diseño, desarrollo y evaluación de software educativo

Tesis de Magíster en Informática. (versión resumida)
Facultad de Informática. UNLP



Ing. Zulma Cataldi

**Directores: Dr. Ramón García-Martínez
Ing. Raúl Pessacq**

ISBN 960-34-0204-2
2000

liema@mara.fi.uba.ar

Contenidos

Índice	2
Resumen	4
Abstract	4
Introducción	5
Objetivos	6
Estructura de la tesis	6
Estado del Arte	8
1.Las teorías del aprendizaje y el diseño de software educativo.	8
1.1. Resumen	8
1.2. El condicionamiento operante: La instrucción programada	8
1.3. Los ambientes constructivistas de aprendizaje	8
1.4. El cognitivismo y los mapas conceptuales.	9
1.5. La Teoría Uno y la teoría de las inteligencias múltiples	10
1.6. Las cogniciones repartidas o distribuidas.	11
1.7. Aprender a aprender	11
1.8. Los desarrollos actuales de software	12
1.9. La aparición del software educativo	12
1.10. La problemática actual	13
2. El software educativo	13
2.1. Resumen	13
2.2. Definiciones	14
2.3. Las Tipologías	14
2.4. Clasificación de los programas didácticos	15
2.5. Las funciones del software educativo	16
2.6. El rol docente y los usos del software.	16
2.7. Las funciones del profesor y los materiales didácticos	17
2.8. Los objetivos educativos	17
2.9. Las actividades de comprensión a desarrollar por los alumnos	18
2.10. La motivación	18
2.11. La organización y presentación de los contenidos	19
2.12. La comunicación: Las interfaces humanas.	19
2.13. La planificación didáctica.	21
3. La ingeniería de software	21
3.1. Resumen	21
3.2. Fundamentos	22
3.3. Los procesos del ciclo de vida del software	23
3.3.1. El modelo en cascada	23
3.3.2. El modelo incremental, de refinamiento sucesivo o mejora iterativa.	24
3.3.3. Prototipado evolutivo	24
3.3.4. El modelo en espiral de Boehm	24
3.3.5. Los modelos orientados al objeto	25
3.4. La necesidad de una metodología de desarrollo	26

3.4.1. Evolución de las metodologías de desarrollo	27
3.4.2. Características y clasificación de las metodologías	27
3.4.2.1. Metodologías estructuradas	27
3.5. El ciclo de vida y los procesos	28
3.5.1. La planificación de la gestión proyecto	29
3.5.2. La identificación de la necesidad	29
3.5.3. El proceso de especificación de los requisitos	29
3.5.4. El proceso de diseño	30
3.5.5. El proceso de implementación	30
3.5.6. El proceso de instalación	31
3.5.7. Los procesos de mantenimiento y retiro	31
3.5.8. El proceso de verificación y validación	31
3.5.9. El proceso de la gestión de la configuración	32
3.5.10. Los procesos de desarrollo de la documentación y de formación	32
3.5.11. La selección de un ciclo de vida	32
3.6. El concepto de la calidad	32
3.6.1. La calidad en ingeniería de software	33
3.6.2. La calidad desde el aspecto organizacional. La familia ISO 9000	33
3.6.3. El concepto de calidad del software	35
3.6.4. Métricas de calidad del software	35
3.6.5. Las diferentes aproximaciones	36
3.6.6. La verificación y la validación del software	36
3.6.7. Las revisiones del software	37
4. La evaluación de software educativo.	37
4.1. Resumen	37
4.2. La evaluación	37
4.3. La evaluación interna	38
4.4. La evaluación externa	38
4.5. Los instrumentos de evaluación	39
4.6. Las propuestas de selección y evaluación de software educativo	39
Conclusiones del Estado del Arte	41
Descripción de la Problemática	43
5. Presentación de la problemática	43
5.1. Resumen	43
5.2. La problemática	43
5.3. El diseño y desarrollo del software educativo	43
5.4. La evaluación del software educativo	43
5.5. La calidad en el software educativo	44
Solución Propuesta	45
6. Propuesta de metodología de diseño y desarrollo	45
6.1. Resumen	45
6.2. La elección del ciclo de vida	45

6.3. La matriz de actividades	47
6.4. El equipo de trabajo	53
6.5. El primer diseño del programa	53
6.6. Acerca del diseño	54
6.7. La documentación	54
6.8. Otras cuestiones	54
Propuesta de Evaluación	55
7.1. Resumen	55
7.2. Desarrollo	55
7.2.1. Prototipo V1 (versión 1)	56
7.2.2. Prototipo V2 (versión 2)	56
7.2.3. Evaluación del prototipo versión final (vfinal)	57
7.3. Evaluación interna	57
7.4. Evaluación externa	57
7.5. La calidad de los programas de software: un problema interdisci- plinario	57
7.5.1. La propuesta: ¿qué medir en el software educativo?	57
7.5.2. La calidad desde la perspectiva pedagógica	58
7.5.3. Algo más acerca de la evaluación de los programas educativos	59
7.5.4. La integración de perspectivas	59
	60
8. Evaluación contextualizada	
8.1. Resumen	60
8.2. Formulación de la tesis y etapas preparatorias	61
8.3. Desarrollo de la experiencia y valuación estadística	
Conclusiones Finales	64
1. Aportes del presente trabajo	64
2. Líneas de trabajo futuras.	64
Anexos	66
Anexo I: Planilla de evaluación de la interface de comunicación. Proto- tipo v1	66
Anexo II: Evaluación de contenidos y pertinencia. Prototipo v2	66
Anexo III: Evaluación Prototipo versión final (vfinal)	67
Anexo IV: Evaluación Externa de Producto Final	68
Anexo V: Aplicación de Criterios y Subcriterios de Calidad	68
Referencias Bibliográficas	69

Resumen

La presente tesis se orienta a realizar una contribución en el área de metodología para el diseño, desarrollo y evaluación de software.

En particular, la metodología que se propone es aplicable al proceso de desarrollo de software educativo, contemplándose en las distintas etapas metodológicas aspectos de naturaleza pedagógica que no son tenidos en cuenta en las metodologías convencionales.

Debido a la diversidad y multiplicidad de las actividades que se requieren para elaborar el producto de software, la metodología da soporte a un desarrollo tecnológico interdisciplinario, que tiene como pilares a la ciencia informática y a las ciencias de la educación.

Abstract

The present thesis is guided to carry out a contribution in the area of methodologies for design, development and software evaluation.

In particular, the methodology proposed in this work is applicable to the process of educational software development. Pedagogical, aspects are contemplated in the different stages of the proposed methodology. This aspects are not kept in mind in the conventional methodologies.

Due to the diversity and multiplicity of the activities that are required to elaborate the software product, the proposed methodology gives support to an interdisciplinary technological development that has basis on the computer science and the education sciences.

Introducción

En el trabajo de tesis se plantea una metodología para el diseño, desarrollo y evaluación del software educativo. La misma se basa en la sinergia de dos campos del saber aparentemente disímiles: la ingeniería de software por un lado y las teorías de aprendizaje modernas por el otro, que convergen en la generación de un producto deseable: el software educativo.

El presente trabajo pretende contribuir a las crecientes investigaciones que en estos últimos años se vienen realizando, tratando de desarrollar un software que contemplase los objetivos educativos, sin desmedro de las pautas de calidad en software.

Por lo tanto, la elección de este tema de tesis reúne tres tipos de interés que todo trabajo de estas características debe comprender:

- Un interés pedagógico: ya que mediante el uso del software apropiado los alumnos adquirirán distintas capacidades a través de las estrategias de enseñanza utilizadas. Sin querer dejar de lado las líneas conductistas, los diseños en la actualidad se basan en las teorías de Bruner (1988), Ausubel y Novak (1983), Perkins (1995) y Gardner (1995), entre otros.
- Interés profesional: puesto que se enmarca en los lineamientos actuales de la ingeniería del software y los desarrollos realizados durante los últimos años en cuanto a normativa a utilizar en el diseño de los productos software.
- Un interés económico-social: ya que esta investigación pretende ser un aporte más al mejoramiento del nivel educativo del país que afectará todas las áreas productivas

Es un trabajo de relevancia en el ámbito educativo y tecnológico, con la derivación socio-económica consecuente. Es un desarrollo de base conceptual y se lo prevé como una herramienta metodológica aplicable tanto en el ámbito educativo como en el no educativo.

Es el deseo de la autora que este aporte sea además, una aproximación para la fijación de directrices aplicadas a las tareas concernientes al desarrollo de software para el área educativa y criterios de evaluación de los productos educativos.

El software educativo durante los últimos años, ha tenido un creciente desarrollo y gran parte del mismo ha sido realizado en forma desorganizada y poco documentada, y considerando el aumento exponencial que sufrirá en los próximos años, surge la necesidad de lograr una metodología disciplinada para su desarrollo, mediante los métodos, procedimientos y herramientas, que provee la ingeniería de software para construir programas educativos de calidad, siguiendo las pautas de las teorías del aprendizaje y de la comunicación subyacentes.

Las primeras ideas sobre desarrollo de software educativo aparecen en la década de los 60, tomando mayor auge después de la aparición de las microcomputadoras a fines de los 80.

Los programas se han desarrollado según tres líneas distintas. La primera corresponde a los lenguajes para el aprendizaje y de ella nace el Logo, como un lenguaje que fue utilizado en un sentido constructivista del aprendizaje. Es de decir, el alumno no descubre el conocimiento, sino que lo construye, sobre la base de su maduración, experiencia física y social (Bruner, 1988). Su evolución continua en la actualidad hacia otras formas de interacción llamadas micromundos. A partir de ahí se ha desarrollado infinidad de software de acuerdo a las diferentes teorías, tanto conductistas, constructivistas como cognitivistas (Gallego, 1997).

La segunda línea corresponde a la creación de lenguajes y herramientas que sirven para la generación del producto de software educativo. Ella, se inicia con la aparición de los lenguajes visuales, los orientados a objetos, la aplicación de los recursos multimediales (Nielsen, 1995) y las herramientas de autor, el campo del desarrollo del software se ha hecho muy complejo, razón por la cual se necesita de una metodología unificada para su desarrollo.

Por último, surgen los productos propiamente dichos que nacen con la enseñanza asistida por computadora (EAC) u ordenador (EAO) que dio la aparición del software educativo, y que a su vez se difundió según tres líneas de trabajo: como tutores (enseñanza asistida por computadoras), como aprendices y herramientas. (Schunk, 1997).

Se pueden enumerar algunos de los problemas detectados que aún subsisten, como la mistificación de las herramientas informáticas aplicadas por los técnicos, la falta de capacitación docente en el tema específico y que las reglas y los pasos metodológicos para la creación de software en general se modifican evolutivamente.

Es por ello, que se quiere presentar una solución informática para el diseño, desarrollo y evaluación tanto interna como externa, mediante la aplicación de las métricas correspondientes, para determinar los parámetros básicos del proyecto de software educativo, teniendo en cuenta los requerimientos particulares del mismo en cuanto a los aspectos pedagógicos. En este enfoque disciplinado para el desarrollo de dicho software, se pretende aplicar los métodos, procedimientos y herramientas de la ingeniería del software, los cuales ayudan a asegurar la calidad del mismo.

El software educativo, tiene características particulares en cuanto a la comunicación con el usuario (Gallego, 1997), las cuales no se pueden cuantificar mediante métricas porque están relacionadas con conductas de

aprendizajes o actos de significado, pero las reglas en la construcción de un programa son las mismas ya sea para el ámbito educativo, comercial, de investigación, u otros.

La eficacia del producto constituye a su vez un alto riesgo debido a que sólo puede ser medida después de finalizado y probado por los alumnos (Fainholc, 1994), por ello es fundamental la instancia de evaluación tanto interna como externa (Marquès, 1995; Sancho, 1994; Reeves, 1997; Meritxell, 1996), y la contextualizada para el logro del producto deseado.

Algunos autores como Marquès (1995) sostienen que las metodologías específicas a utilizar para el diseño del software educativo se pueden englobar bajo el nombre de ingeniería de software educativo.

Objetivos

Se han determinado una serie de objetivos que a continuación se detallan.

Objetivos pertinentes a la construcción del objeto de estudio

- Definir qué es software educativo
- Ofrecer un estudio crítico de la situación actual

Objetivo general

- Construir una metodología disciplinada para el desarrollo del software educativo, mediante la identificación de los métodos, los procedimientos, y las herramientas, que provee la ingeniería de software para el desarrollo de programas educativos de calidad, siguiendo las pautas de la teoría educativa subyacente.

Objetivos particulares

- Justificar un modelo apropiado para el ciclo de vida del software educativo.
- Desarrollar una metodología de evaluación interna y externa del software educativo a fin de lograr un producto de calidad.

Estructura de la tesis

La tesis se divide básicamente en seis grandes partes: Introducción, Estado del Arte, Descripción del Problema, Descripción de la Solución Propuesta, Parte Experimental y Conclusiones. Estas a su vez se subdividen en capítulos.

Introducción: Aquí se describe la presentación de la problemática contextualizada, los objetivos propuestos, y la estructura de la tesis.

Estado del Arte: En la “**Introducción**”, se describe como se efectuó el relevamiento realizado en cuanto a la evolución de los desarrollos de software educativo, sin base pedagógica en sus inicios y luego con la base de las teorías del aprendizaje que los sustentan. Se intenta dar un panorama de cómo evolucionaron hasta nuestros días tales desarrollos y qué aspectos deberían tomarse de la ingeniería de software para mejorar los diseños. Se pretende dar un panorama neutral de los trabajos más relevantes realizados en el área, presentando la mayor parte de las herramientas disponibles y sus fundamentos teóricos para considerarlas como un punto de partida para el desarrollo de la solución propuesta. En la sección 1, denominado “**Las teorías del aprendizaje y el diseño de software educativo**”, se presenta una síntesis diacrónica de las teorías del aprendizaje más conocidas y se las relaciona con la aparición del software educativo. Se plantea el panorama actual acerca de las problemáticas vigentes a causa de algunos aspectos divergentes en la construcción de los programas educativos. La sección 2, llamado “**El software educativo**”, es una síntesis de las tipologías y características principales de los programas educativos, los diferentes roles que pueden tener los profesores que los utilizan y los procesos de comprensión que se intenta desarrollar o incentivar en los alumnos. Se describen las interfaces de comunicación usuario–software y se considera el contexto de aplicación y uso de los programas mediante el empleo de una buena planificación didáctica.

En la sección 3, “**La ingeniería de software**” se describe desde la ingeniería de software, la gran variedad de modelos o paradigmas de ciclo de vida, para el desarrollo de los programas, ya sea para un gran proyecto de software, o un simple programa. Se describen las metodologías, los métodos, los procedimientos y las herramientas que utiliza la ingeniería de software. Se incluye un apartado especial acerca de las métricas utilizadas en la determinación de la calidad, como de la normativa vigente en cuanto a productos lógicos.

La sección 4 corresponde a “**Evaluación del software educativo**”, y en él se detallan todos los aspectos a tener en cuenta para una aplicación óptima a nivel áulico. Se resumen los trabajos considerados más relevantes en cuanto a evaluación que en general toman la forma de listas de control o valoración ponderativa de algunos criterios.

En “**Conclusiones del Estado del Arte**”: se quieren señalar simplemente, aquellos aspectos a tener en cuenta para los futuros diseños de los programas educativos, que han detectado otros investigadores, y las posibles soluciones a algunas de las problemáticas planteadas.

Descripción del Problema: En esta sección que consta de 1 capítulo 5; “**Presentación de la problemática**”, se describe el problema actual de los desarrollos de los programas educativos que se pretende resolver.

Descripción de la Solución Propuesta: Esta parte de la tesis se divide en dos capítulos. El primero, es la sección 6 “**Propuesta de metodología de diseño y desarrollo**”, y aquí se consideran las características de

los programas educativos, para definir un modelo de ciclo de vida y una teoría educativa apropiados. Se presentan las actividades a realizar en cada una de las etapas a fin de establecer los recursos materiales y humanos indispensables para cada una de ellas y se analiza la configuración del equipo de desarrollo y los roles cada uno de los participantes en los procesos.

La sección 7 es la **“Propuesta de evaluación”**, sobre la base de la doble evaluación de los productos de software educativo, que debe considerar aspectos técnicos y pedagógicos, se evalúa un software desarrollado a partir de la propuesta de la sección anterior, y se detallan las evaluaciones interna y externa del producto. Se resumen los resultados de las evaluaciones realizadas, de los prototipos presentados y del producto final, llevados a cabo por los grupos de alumnos evaluadores. Se evaluaron progresivamente: la interface de comunicación en el primer caso, el funcionamiento de las bases de imágenes, vídeos y efectos, en el segundo y finalmente el producto final con, el agregado de la voz, los textos y la música. Se dedica un apartado a evaluar aquellos aspectos que se consideran importantes para desarrollar un software de calidad.

Parte Experimental: En la sección 8: **“Parte experimental”**: se realiza una evaluación contextualizada, contrastando un producto realizado con la metodología propuesta, con otro desarrollado sin una metodología explícita.

Finalmente se presentan las **Conclusiones** a esta propuesta y experiencia y se dejan bosquejadas algunas posibles líneas de investigación futuras.

Estado del Arte

1. Las teorías de aprendizaje y el diseño de software educativo.

1.1. Resumen

En esta sección se presenta la aparición y la evolución del software educativo a la luz de algunas de las teorías del aprendizaje más representativas. En este paralelismo, sólo se mencionan aquellas teorías que darán los marcos conceptuales para los desarrollos de los programas didácticos en función de las aplicaciones deseadas.

Partiendo del conductismo de Skinner, (sección 1.2) se pasa por el constructivismo (sección 1.3) y las diversas líneas de la psicología cognitiva (secciones 1.4, 1.5, 1.6) considerando también los aspectos metacognitivos (sección 1.7).

Se quieren destacar también, los cambios paradigmáticos producidos a partir del conductismo, el constructivismo y la psicología cognitiva y su repercusión en la construcción de software educativo (secciones 1.8 y 1.9). A partir de aquí es donde se concatena la noción de secuenciación de contenidos de Coll con la programación estructurada modular.

Pero, es con la irrupción de la computadoras personales a bajo costo, cuando se masiviza el uso de los programas educativos, y el uso de la computadora como tutor, herramienta o aprendiz, según Schunk, teniendo como sustrato la teorías de aprendizaje mencionadas. Por último, se da una síntesis de la problemática actual en el área (sección 1.10).

1.2. El condicionamiento operante: La instrucción programada.

A comienzos de la década de los 60 se pensó que una de las posibles soluciones a algunos de los problemas educativos de esa época, consistía en la aplicación de los avances tecnológicos a la enseñanza. Sin embargo, la introducción de los instrumentos tecnológicos no fue acompañada con una teoría acerca de la enseñanza y del aprendizaje.

Skinner (1958, 1963) formuló su teoría conductista del condicionamiento operante en los años treinta y, durante los primeros años de su carrera se interesó por la educación elaborando las “*máquinas de enseñanza*” y los “*sistemas de instrucción programada*”. El cambio conductual en el “*condicionamiento operante*” se da a través del refuerzo diferencial por aproximaciones sucesivas hacia la forma de comportamiento deseada, mediante el proceso de moldeamiento para modificar la conducta.

Durante los años sesenta aparecen una corriente de “*programadores*” (Deterline, 1969), que empezaron a “*programar*” de una manera muy fácil, y, que careciendo de formación docente, tomaban un libro de texto, borraban alguna palabra de una frase elegida y la sustituían por una línea horizontal, para que el alumno anotara allí su respuesta. Repetían la frase varias veces por cada cuadro, pero borrando una palabra diferente cada vez.

En esta época es cuando comienzan los estudios referidos a la elaboración de lo que se considera una buena “*programación didáctica*”. La elaboración de una programación se iniciaba con el establecimiento de los objetivos generales en función del curriculum de los alumnos, se construía el programa, elaborando la serie de secuencias a seguir en “*cuadros*”. Luego, se estudiaba el tipo de respuesta más adecuada y la clase de feedback¹ a lograr. El paso siguiente era la evaluación y revisión del programa sobre la base de las respuestas de los alumnos.

En este período, cobran interés los objetivos operacionales y conductuales a partir de un trabajo de Mager (1967), que se usó como un manual para los escritores de enseñanza programada. El objetivo debe describir una conducta observable y sus productos o logros.

Las décadas de los sesenta y setenta, destacan a una serie de autores dedicados a la definición, la elaboración y la redacción de objetivos conductuales tales como Gagné (1970), quien da una tipología de los aprendizajes, y para cada uno de ellos reconoce estadios o fases, que son las condiciones psicológicas para un aprendizaje eficaz (Fernández Pérez, 1995). El aprendizaje ocurre así, a través de transformaciones de la información.

1.3. Los ambientes constructivistas de aprendizaje

Las primeras ideas sobre desarrollo de software educativo aparecen en la década de los 60, tomando mayor auge después de la aparición de las microcomputadoras a fines de los 80.

El uso de software educativo como material didáctico es relativamente nuevo, los primeros pasos fueron dados por el lenguaje Logo, que a partir de su desarrollo en el MIT (Instituto Tecnológico de Massachusetts) fue utilizado en numerosas escuelas y universidades.

¹ retroalimentación

Se desarrolla una línea de software que corresponde a los lenguajes para el aprendizaje y de ella nace el Logo, que fue utilizado en un sentido constructivista del aprendizaje.

Es decir, como sostiene Bruner: *"el punto crucial y definitorio del aprendizaje, del conocimiento de algo nuevo, radica en la posibilidad humana de abstraer en los objetos algunos pocos rasgos para construir criterios de agrupamiento de los objetos abstraídos"*, a pesar de que con frecuencia acontece que los rasgos comunes son muchos menos y menores, que los rasgos que los diferencian como plantea Fernández Pérez (1995). En otras palabras, hace del proceso de formación de conceptos una instrumentalización cognitiva.

El alumno no descubre el conocimiento, sino que lo construye, en base a su maduración, experiencia física y social (Bruner 1988), es decir el contexto o medio ambiente.

Según Bruner, algunas de las habilidades a adquirir son: la capacidad de identificar la información relevante para un problema dado, de interpretarla, de clasificarla en forma útil, de buscar relaciones entre la información nueva y la adquirida previamente.

Hablar de ambientes de enseñanza constructivistas significa concebir el conocimiento desde la perspectiva de Piaget (1989) mediante desarrollos cognitivos basados en una fuerte interacción entre sujeto y objeto, donde el objeto trata de llegar al sujeto, mediante cierta perturbación de su equilibrio cognitivo, quien trata de acomodarse a esta nueva situación y producir la asimilación del objeto, con la consecuente adaptación a la nueva situación. En este esquema conceptual piagetiano, se parte de la acción, esencial, ya sea para la supervivencia, como para el desarrollo de la cognición. *"La postura constructivista psicogenética acepta la indisolubilidad del sujeto y del objeto en el proceso de conocimiento. Ambos se encuentran entrelazados, tanto el sujeto, que al actuar sobre el objeto, lo transforma y a la vez se estructura a sí mismo construyendo sus propios marcos y estructuras interpretativas"* (Castorina, 1989).

A partir de aquí, se ha desarrollado infinidad de software de acuerdo a las diferentes teorías, tanto conductuales, constructivistas y posteriormente cognitivistas (Gallego 1997).

1.4. El cognitivismo y los mapas conceptuales.

El cognitivismo tiene sus raíces en la ciencia cognitiva y en la teoría de procesamiento de la información. Howard Gardner psicólogo de Harvard, sostiene que el nacimiento de la psicología cognitiva es de 1956. Es a partir de esta fecha que se empieza a gestar el movimiento que algunos llaman revolución cognitiva y que a juicio de Lachman et al. (1979) *"constituyó un verdadero cambio de paradigmas en el sentido kuhniano"*. (Hernández, 1997)

El cognitivismo es una teoría de aprendizaje donde la mente es un agente activo en el proceso de aprendizaje, construyendo y adaptando los esquemas mentales o sistemas de conocimiento. Bruner (1991) sostiene que la revolución cognitiva tenía como objetivo principal recuperar la mente, después de la época de la *"glaciación conductista"* (Hernández, 1998). En los inicios del modelo cognitivo, señala Bruner, había una firme intención en la realización de esfuerzos para indagar acerca de los procesos de construcción de los significados y producciones simbólicas, empleados para conocer la realidad circundante. Sin embargo, el papel creciente de la informática y las computadoras incorporó un planteamiento basado en la metáfora de las computadoras.

Dentro de la teoría cognitiva los psicólogos del procesamiento de la información usan la analogía de la computadora para explicar el aprendizaje humano, con el supuesto básico de que todo aprendizaje consiste en formar asociaciones entre estímulo y respuesta. Según Gardner (1987) y Rivière (1987): *"el paradigma del procesamiento de la información dentro de la psicología educativa, se inserta en la gran tradición racionalista de la filosofía, que otorga cierta preponderancia al sujeto en el acto del conocimiento"*.

Vigotzkii (1978), por otra parte desde su modelo sociocultural, destaca las actividades de aprendizaje con sentido social, atribuyendo gran importancia al entorno sociocomunicativo del sujeto para su desarrollo intelectual y personal. Sostiene que el cambio cognitivo, se da en la ZDP (zona de desarrollo próximo) o sea la distancia entre el nivel real de desarrollo y el nivel posible, mediante la resolución de problema mediado por un adulto o tutor, siendo a veces el aprendizaje repentino, en el sentido *gestáltico*³ del *insight*.⁴

Entre las ideas de Vigotzkii, se deriva un concepto muy importante que es el que Bruner denomina *"andamiaje"* educativo que consiste en brindar apoyo, y en el caso de la computadora como herramienta, para permitir ampliar el alcance del sujeto y la realización de tareas que de otro modo serían imposibles y usarlos selectivamente cuando se necesitan.

Rogers (1984) habla de *"la facilitación del aprendizaje que aparece como una potencialidad natural de todo ser humano"*. Dice que *"el aprendizaje significativo"* tendrá lugar cuando el sujeto perciba al tema como importante para sus propios objetivos o satisfaciendo alguna de sus características o necesidades personales sociales. El término significativo también puede ser entendido siguiendo a Ausubel (1983), como un contenido que tiene una estructuración lógica interna y como aquel material que puede ser aprendido de manera significativa por el sujeto. Rogers afirma que *"el aprendizaje social más útil en el mundo es el aprendizaje*

² En referencia a Kuhn T. (1980) en *La estructura de las revoluciones científicas*. México. FCE.

³ En referencia a la teoría de la forma

⁴ En el sentido de visión repentina

del proceso de aprendizaje, que significa adquirir una actitud continua de apertura frente a las experiencias e incorporar a sí mismo el proceso de cambio".

"El conocimiento elaborado a través de conceptos teóricos de las diferentes disciplinas, requiere también desarrollos en la recepción en los alumnos para una comprensión significativa" (Ausubel, 1983). Esta denominación de "comprensión significativa o aprendizaje significativo" tiene para Ausubel un sentido muy particular: incorporar información nueva o conocimiento a un sistema organizado de conocimientos previos en el que existen elementos que tienen alguna relación con los nuevos.

El alumno que carece de tales esquemas desarrollados, no puede relacionar significativamente el nuevo conocimiento con sus incipientes esquemas de comprensión, por lo que, ante la exigencia escolar de aprendizaje de los contenidos disciplinares, no puede sino incorporarlos de manera arbitraria, memorística, superficial o fragmentaria. Este tipo de conocimiento es difícilmente aplicable en la práctica y, por ello, fácilmente olvidado.

El nuevo material de aprendizaje solamente provocará la transformación de las creencias y pensamientos del alumno cuando logre "movilizar los esquemas ya existentes de su pensamiento". Al alumno se le debe enseñar de tal manera, que pueda continuar aprendiendo en el futuro por sí solo. Ausubel y sus colaboradores, según expresa Coll (1994), "concretan las intenciones educativas por la vía del acceso a los contenidos, lo cual exige tener un conocimiento profundo de los mismos para armar un esquema jerárquico y relacional".

Según Novak y Ausubel, (1997) todos los alumnos pueden "aprender significativamente un contenido, con la condición de que dispongan en su estructura cognoscitiva o cognitiva, de conceptos relevantes e inclusores". Cabe recordar la frase: "El factor más importante que influye en el aprendizaje es lo que el alumno ya sabe. Averigüese esto y enséñese consecuentemente", (tal como Ausubel, Novak y Hanesian expresan en el prefacio de su libro "Psicología Educativa. Un punto de vista cognoscitivo"), esencial para construir herramientas o indicadores diagnósticos de la estructura cognitiva de los alumnos. El contenido del aprendizaje debe ordenarse de tal manera que los conceptos más generales e inclusivos se presenten al principio. Esto favorece la formación de conceptos inclusores en la estructura cognoscitiva de los alumnos que facilitan, posteriormente, el aprendizaje significativo de los otros elementos del contenido.

Para lograr una diferenciación progresiva del conocimiento del alumno y una "reconciliación integradora" posterior, las secuencias de aprendizaje tienen que ordenarse partiendo de los conceptos más generales y avanzando de forma progresiva hacia los conceptos más específicos.

"El aprendizaje significativo, es un aprendizaje globalizado en la medida en que el nuevo material de aprendizaje pueda relacionar de forma sustantiva y no arbitraria con lo que el alumno ya sabe", (Coll, 1994), con calidad de lo aprendido y duración del almacenamiento.

Los mapas conceptuales, adaptados de Novak (1984), surgen como una herramienta base para representar las relaciones significativas entre conceptos. Actualmente son el fundamento para la red semántica base para el desarrollo del software educativo cognitivista.

El mapa de base, es el punto de partida para el acuerdo entre los especialistas de las diferentes áreas que intervienen en dicho desarrollo. Esta base proveerá un camino de navegación libre de ambigüedades e incoherencias. Usando recursos hipermediales, se pueden construir documentos interrelacionados siguiendo una estructura jerárquica de modo que el alumno navegue pasando desde las informaciones más inclusivas a las más específicas.

1.5. La Teoría Uno y la Teoría de las inteligencias múltiples

David Perkins (1995), codirector del Proyecto Zero del Centro de Investigación para el Desarrollo Cognitivo, de Harvard, en su Teoría Uno afirma que "la gente aprende más cuando tiene una oportunidad razonable y una motivación para hacerlo". Puede parecer imposible que este enunciado tan trivial, dice el autor, implique alguna mejora en la práctica educativa, pero basándose en el sentido común, se podrían señalar las siguientes condiciones:

- Información clara.
- Práctica reflexiva.
- Realimentación informativa.
- Fuerte motivación intrínseca y extrínseca.

La Teoría Uno intenta simplemente ser un punto de partida: "Dada una tarea que se desea enseñar, si se suministra información clara sobre la misma mediante ejemplos y descripciones, si se ofrece a los alumnos tiempo para practicar dicha actividad y en pensar cómo encargarla, si se provee de realimentación informativa con consejos claros y precisos para que el alumno mejore el rendimiento y se trabaja desde una plataforma de fuerte motivación intrínseca y extrínseca, es probable que se obtengan logros considerables en la enseñanza." (Perkins, 1995).

La Teoría Uno no es un método de enseñanza, sino un conjunto de principios que todo método válido de enseñanza debe satisfacer. Sobre esto Perkins afirma en su obra: "La Escuela Inteligente", que cualquier método válido de enseñanza encarna a la Teoría Uno y amplía sus principios para adaptarse a las necesidades

particulares del estudiante y del momento. *"Una buena enseñanza requiere métodos distintos para ocasiones distintas"*: la Teoría Uno debe subyacer a todos ellos. Mortimer Adler⁵, en *"La Escuela Inteligente"* destaca tres modos de enseñar: la instrucción didáctica, el entrenamiento y la enseñanza socrática. (Perkins, 1995).

La Teoría Uno es compatible con el conductismo y con el constructivismo, pero no enfatiza la importancia de que el alumno elabore sus ideas con un alto grado de autonomía a fin de alcanzar la verdadera comprensión. Puede considerarse como un mojón que marca el primer hito hacia otras teorías más elaboradas y aún usando sólo sus dos versiones más simples: con la instrucción didáctica y el entrenamiento, se obtendrían resultados considerablemente mejores que los actuales.

En el caso de desarrollos del software educativo, se pueden incorporar, como sostiene Perkins, representaciones potentes, mediante imágenes mentales y utilizar modelos, de tal modo de estimular la motivación de los alumnos e intentar desarrollar actividades mentales como:

- evaluar y discriminar lo específico de lo particular,
- construir, crear,
- evaluar necesidades, procesos, resultados,
- investigar otras posibilidades de solución,
- resolver problemas inéditos,
- transferir conocimiento de y hacia otras áreas,
- sintetizar, globalizar, analizar, etc.

Perkins habla acerca de la conexión importante que existe entre la pedagogía de la comprensión (o el arte de enseñar a comprender) y las imágenes mentales, por lo que se puede decir que la relación es bilateral.

Esta relación recíproca existente puede ayudar al alumno a adquirir imágenes mentales, con lo cual desarrolla su capacidad de comprensión y al exigirles actividades de comprensión (como por ejemplo: predecir, explicar, resolver, ejemplificar, generalizar) se hará que construyan imágenes mentales, para lo que afirma Perkins que: *"se alimentan unas a otras como si fueran el Yin y el Yan de la comprensión"*. En cuanto a la transferencia, la idea es aprender en una situación determinada y luego aplicar lo aprendido en otra muy diferente.

Una enseñanza comprensiva para favorecer el desarrollo de los procesos reflexivos, es la mejor manera de generar la construcción del conocimiento no frágil.

Por otra parte, el psicólogo Howard Gardner (1993), quien enunció la "Teoría de las Inteligencias Múltiples" sostiene que la inteligencia humana posee siete dimensiones diferentes y a cada una de ellas corresponde un sistema simbólico diferente y un modo de representación: lógico-matemática, lingüística, musical, espacial, cinético-corporal, interpersonal e intrapersonal.

Gardner sostiene que la práctica educativa se centra fundamentalmente en las inteligencias matemática y lingüística y que dado el carácter múltiple de la inteligencia humana se debe ampliar el horizonte a fin de dar cabida a las diversas habilidades de las personas, proponiendo a los alumnos proyectos que admitan modos alternativos de expresión simbólica, creando proyectos grupales que inviten a los alumnos a trabajar con el lenguaje de los medios de comunicación y con sistemas simbólicos por los que sientan una mayor afinidad e induciendo una mayor diversidad de sistemas simbólicos en las diferentes áreas del saber.

La teoría de las inteligencias múltiples supera a la Teoría Uno en tanto que hace hincapié en la diversidad de la capacidad humana en la consecuente necesidad de diversificar las oportunidades y los caminos pedagógicos. (Perkins, 1995).

1.6. Las cogniciones repartidas o distribuidas.

Respecto de la relación persona-herramienta que interactúan para dar lugar al proceso cognitivo, Perkins (1985) dice que la cognición humana, siempre se produce de una manera física, social y simbólicamente repartida. Las personas piensan y recuerdan con la ayuda de toda clase de instrumentos físicos e incluso construyen otros nuevos con el fin de obtener ayuda. Las personas piensan y recuerdan por medio del intercambio con los otros, compartiendo información, puntos de vista y postulando ideas.

Libedinsky (1995) en el marco pedagógico de la utilización de tecnologías en el ámbito educativo, dice que uno de los principios clave que puede operar es el de las cogniciones repartidas. *"Cuando se examina la conducta humana en la resolución de problemas de la vida real y en entornos laborales, la gente parece pensar en asociación con otros y con la ayuda de herramientas provistas por la cultura, las cogniciones parecerían no ser independientes de las herramientas con las que se resuelve un problema. Las cogniciones parecerían distribuirse físicamente con nuestros útiles y herramientas, entre ellas la computadora, socialmente con quienes compartimos las tareas intelectuales y simbólicamente desde las palabras, gráficos y mapas conceptuales, entre otros, como medios de intercambio entre la gente. Los recursos físicos y sociales, participan en la cognición no sólo como fuente sino como vehículo del pensamiento"*. (Libedinsky, 1995).

1.7. Aprender a aprender

⁵ Citado por David Perkins (1995), en *"La Escuela Inteligente"*. Gedisa.

La metacognición se refiere al "*conocimiento de los propios procesos cognitivos*", es una forma de conocimiento que tiene como rasgo diferencial su referencia al sistema humano de procesamiento de información, es decir, conocer qué son, cómo se realizan, cómo se potencian o interfieren los procesos cognitivos como la percepción, la atención, la memorización, la lectura, etc.

Es el conocimiento que ha desarrollado el alumno acerca de sus experiencias almacenadas y de sus propios procesos cognoscitivos, así como de su conocimiento estratégico y la forma apropiada de uso. (Flavell, 1993). El conocimiento metacognitivo es de aparición relativamente tardía en casi todos los dominios del aprendizaje escolar. Básicamente, la metacognición tiene que ver con el conocimiento que cada uno tiene de sus propios procesos cognitivos, abarcando también, el control activo y la regulación de tales procesos, lo cual implica tener conciencia de las propias fortalezas y debilidades acerca del funcionamiento intelectual de cada uno.

1.8. Los desarrollos actuales de software

Una *segunda línea* en los desarrollos de software, corresponde a la creación de lenguajes y herramientas para la generación del producto de software educativo. Ella, se inicia con la aparición de los lenguajes visuales, los orientados a objetos, la aplicación de los recursos multimediales (Nielsen 1995) y las herramientas de autor, complejizando el campo del desarrollo del software, razón por la cual se necesita de una metodología unificada para su desarrollo.

Los lenguajes de programación han experimentado en los últimos años un notable auge. El por qué del crecimiento evolutivo, a partir de los lenguajes de máquina y ensambladores, debe buscarse en el intento por acercarse a los lenguajes naturales de las personas. Surgen así, los lenguajes de alto nivel o evolucionados, a partir del FORTRAN en 1955, desarrollado por IBM; el Cobol, se creó en 1960, como un intento del Comité CODASYL de lenguaje universal para aplicaciones comerciales, el PL/I, que surge en los sesenta para ser usado en los equipos de IBM 360.

El Basic surge en 1965, lenguaje ampliamente usado en el ámbito educativo y en 1970 aparece el Pascal, creado por el matemático Niklaus Wirth, basándose en el Algol de los sesenta. Este lenguaje en particular aporta los conceptos de programación estructurada, tipo de datos y diseño descendente. La evolución continúa hacia otros más modernos como el C, creado en 1972 por Denis Ritchie y el ADA, cuya estandarización se publicó en 1983 (Alcalde et al., 1988).

Los lenguajes se incorporaron rápidamente al ámbito educativo, porque se consideró que permitían ayudar a mejorar el pensamiento y acelerar el desarrollo cognitivo. Los estudios en este aspecto si bien sostienen que se pueden lograr habilidades cognitivas no indican que se facilite la transferencia hacia otras áreas del saber. (Liguori, 1995)

1.9. La aparición del software educativo

Por último aparecen *los productos propiamente* dichos de software educativo, con la difusión de las computadoras en la enseñanza, según tres líneas de trabajo, computadoras como tutores (enseñanza asistida por computadoras o EAC), como aprendices y como herramienta. (Schunk 1997).

La enseñanza asistida por computadora (EAC) o enseñanza basada en computadora (EBC) es un sistema que se utiliza sobre todo para efectuar ejercicios, cálculo, simulaciones y tutorías. Los programas de ejercicios son fáciles de realizar y los alumnos proceden a manejarlos en forma lineal en su repaso de información. Las tutorías presentan información y retroalimentación, de acuerdo a la respuesta de los estudiantes, que en este caso son programas ramificados.

Una aplicación interesante de las computadoras son las simulaciones por que permiten al alumno ponerse en contacto con una situación real que de otro modo nunca podría hacerlo, tal es el caso de los simuladores de vuelo o de una planta nuclear. Se presenta artificialmente una situación real y con gran uso de recursos gráficos e interactivos. El hecho de usar simulaciones por computadora, en la enseñanza tradicional ha logrado cambios positivos en los alumnos, en cuanto a la resolución de problemas, ya que brindan la posibilidad de acceso a la enseñanza de temas de difícil comprensión y demostración.

Como aprendices, sostiene Schunk (1997) que las computadoras permiten que los estudiantes aprendan a programar, facilitando el desarrollo de habilidades intelectuales tales como reflexión, razonamiento y resolución de problemas. Lepper (1985) sostiene que las computadoras pueden enseñar ciertas habilidades que no son posibles con los métodos tradicionales, y el aprender a programar ayuda a la resolución de problemas al modelado y división del problema en partes más pequeñas. También a la detección y corrección de errores.

Esta es la filosofía del Logo de Papert, al dar las órdenes en el Logo mediante conjunto de instrucciones que producen ciertas configuraciones, combinando comandos con procedimientos. Las investigaciones actuales destacan que la motivación es un aspecto clave que favorece el procesamiento profundo y no sólo el superficial. (Hopper y Hannafin, 1991).

La otra aplicación es la utilización de las computadoras como herramientas, mediante el uso de procesadores de textos, bases de datos, graficadores, planillas de cálculo y programas de comunicación, etc. Son herra-

mientas que ayudan a ordenar, procesar, almacenar, transmitir información, y que pueden mejorar el aprendizaje de acuerdo al uso que de ellas haga el docente.

1.10. La problemática actual

Existen una serie de problemas detectados y que aún subsisten, en la construcción y uso de mediadores pedagógicos, quizás el más relevante sea el intento de desmistificación de las herramientas informáticas aplicadas por los técnicos, la falta de capacitación docente en el tema específico y el desarrollo tecnológico que se modifica rápida y evolutivamente, así como las reglas y los pasos metodológicos para la creación de software.

Es por ello, que se quiere presentar una propuesta informática para el diseño, desarrollo y evaluación tanto interna como externa, mediante la aplicación de las métricas correspondientes, para determinar los parámetros básicos del proyecto de software educativo, teniendo en cuenta los requerimientos particulares del mismo en cuanto a los aspectos pedagógicos.

En este enfoque disciplinado para el desarrollo de dicho software, se pretende aplicar los métodos, procedimientos y herramientas de la ingeniería del software, los cuales ayudan a asegurar la calidad del mismo.

Como la cantidad y la variedad de software educativo crece muy rápidamente, existe una necesidad de evaluarlo, cada vez mayor, para saber si es adecuado a los propósitos educativos. Los docentes necesitan saber cuándo y cómo un programa puede usarse para mejorar su enseñanza, y los alumnos necesitan saber cómo podrían mejorar sus aprendizajes, en este punto son los vendedores deberían asesorar de acuerdo a las necesidades de uso, y entre varios programas similares en el mercado cuál usar.

Los diseñadores de software educativo necesitan definir criterios a partir de los cuales puede evaluarse y posteriormente llevar a cabo una estrategia de evaluación práctica.

Por otra parte se debe diferenciar qué se pretende englobar con el término calidad: ya sea calidad del software desde el punto de vista técnico o calidad del producto desde el punto de vista educativo o ambas. Debe quedar claro que la calidad es un concepto *"multidimensional y polisémico"* porque es el resultado de una larga lista de factores que van desde la tecnología, los contenidos, el docentes, el currículum, etc.

La calidad del software educativo es cambiante desde la perspectiva de los objetivos. Asimismo, algunos investigadores (Campos 1996, Underwood 1990) mencionan que existe una gran controversia en lo que se refiere a determinar cuándo un software se considera *"educativo"*, qué se debería evaluar en un software educativo y qué se considera como un *"software educativo de calidad"*.

Algunos sostienen que la calidad de un software educativo debería responder a un modelo de aprender, un modelo de alumno y el rol de la tecnología en el aula, más bien, a un modelo curricular (Flagg, 1990). Así, no es lo mismo evaluar un software que se utilizará como refuerzo de una clase magistral, que un software de apoyo al trabajo colaborativo de los alumnos.

El problema de la determinación de la calidad en medios de comunicación es un problema recurrente, para el cual numerosos investigadores intentaron definir criterios de calidad del software y compilar clasificaciones y catálogos de ellos. La idea era traducir estos catálogos en listas de verificación que pudieran ser de uso práctico para los docentes al juzgar los medios de comunicación educativos (Baumgartner y Payr, 1996).

Un programa educativo bien diseñado y utilizado ayuda a lograr los *"objetivos educativos"*, entre los que se pueden mencionar: incrementar la calidad de la enseñanza que se ofrece a los estudiantes, reducir los costos de la misma, facilitar el acceso a la educación a mayor número de personas, etc.

Existe una diversidad de estudios que denotan la necesidad del uso de herramientas fáciles de usar y bien documentadas para evaluar tanto el software como sus interfaces (Norman y Drapper 1988; Norman 1988; Winograd 1996).

2. El software educativo

2.1. Resumen

En esta sección se presenta una definición de software educativo (sección 2.2), su tipología (sección 2.3) y su clasificación (sección 2.4). Se describen las principales funciones de los programas educativos (sección 2.5). Posteriormente, se analiza el rol del docente al aplicar los diferentes programas, de acuerdo al estilo docente y a la función de los mismos (sección 2.6). Desde el triángulo didáctico se aborda el problema del cambio del rol docente hacia los mediadores pedagógicos (sección 2.7). Luego, se consideran los objetivos educativos a lograr en las intervenciones didácticas (sección 2.8) y los procesos de pensamiento a desarrollar en los alumnos (sección 2.9), considerando aspectos tales como la motivación (sección 2.10), la organización de los contenidos (sección 2.11) y el diseño de las interfaces de comunicación (sección 2.12).

Finalmente, se exponen los puntos claves que debe tener en cuenta una buena planificación didáctica para el uso de los mediadores pedagógicos (sección 2.13).

2.2. Definiciones

Se define como software educativo a “los programas de computación realizados con la finalidad de ser utilizados como facilitadores del proceso de enseñanza” y consecuentemente de aprendizaje, con algunas características particulares tales como: la facilidad de uso, la interactividad y la posibilidad de personalización de la velocidad de los aprendizajes.

Marquès (1995) sostiene que se pueden usar como sinónimos de “software educativo” los términos “programas didácticos” y “programas educativos”, cen-trando su definición en “aquellos programas que fueron creados con fines didácticos, en la cual excluye todo software del ámbito empresarial que se pueda aplicar a la educación aunque tengan una finalidad didáctica, pero que no fueron realizados específicamente para ello”.

Características	Descripción
Facilidad de uso	En lo posible autoexplicativos y con sistemas de ayuda
Capacidad de motivación	Mantener el interés de los alumnos
Relevancia curricular	Relacionados con las necesidades del docente
Versatilidad	Adaptables al recurso informático disponible
Enfoque pedagógico	Que sea actual: constructivista o cognitivista.
Orientación hacia los alumnos	Con control del contenido del aprendizaje
Evaluación	Incluirán módulos de evaluación y seguimiento.

Tabla 2.1: Características principales de los programas educativos, clasificación según Marquès (1998a).

En la Tabla 2.1 se pueden observar algunas de las características principales de los programas educativos. Se da por sentado que los programas deben usarse como recursos que incentiven los proceso de enseñanza y de aprendizaje, con características particulares respecto de otros materiales didácticos y con un uso intensivo de los recursos informáticos de que se dispone. (Marquès, 1998b).

2.3. Las Tipologías

Los programas educativos se pueden clasificar según diferentes tipologías. En la Tabla 2.2 se puede ver algunas de ellas de acuerdo a diferentes criterios.

Se debe considerar que un aspecto clave de todo buen diseño es tomar en cuenta las características de la interface de comunicación, la que deberá estar de acuerdo con la teoría comunicacional aplicada y con las diferentes estrategias para el desarrollo de determinados procesos mentales.

Por otra parte, cuando el software se desarrolla a partir de un lenguaje de programación, ya sea convencional, orientado a eventos u objetos, se tiene que considerar que se fundamenta en la estructura del algoritmo que lo soporta, cuyo diseño deberá reunir algunas características esenciales como la modularidad y el diseño descendente (como se verá en la sección 3).

Gran parte de los programas educativos pertenecen a un sub-grupo denominado hipermediales, y en ellos las bases de datos de imágenes fijas o en movimiento, vídeo clips y sonidos juegan un rol fundamental a la hora de diseñar el programa.

Tipologías según:		
Los contenidos	Temas, áreas curriculares	
Los destinatarios	Por niveles educativos, edad, conocimientos previos	
Su estructura	Tutorial, base de datos, simulador constructor, herramienta	
Sus bases de datos	Cerrados o abiertos	
Los medios que integra	Convencional hipermedia, realidad virtual	
Su inteligencia	Convencional, sistema experto	
Los objetivos educativos que pretende facilitar	Conceptuales, actitudinales, procedimentales	
Las procesos cognitivos que activa	Observación, identificación, construcción memorización, clasificación, análisis, síntesis, deducción, valoración, expresión, creación, etc.	
El tipo de interacción que propicia	Recognitiva, reconstructiva, intuitiva, constructiva	(Kemmis, 1970)
Su función en el	Instructivo, revelador, conjetural, emancipador ⁶	

⁶ Squires y Mc Dougall (1994) postulan estos cuatro paradigmas.

aprendizaje		
Su comportamiento	Tutor, herramienta, aprendiz	(Taylor, 1980)
El tratamiento de los errores	Tutorial, no tutorial	
Sus bases psicopedagógicas sobre el aprendizaje	Conductista, constructivista, cognitivista	(Gros Begoña, 1997)
Su función en la estrategia di-dáctica	Informar, motivar, orientar, ayudar, proveer recursos, facilitar prácticas, evaluar	
Su diseño	Centrado en el aprendizaje, centrado en la enseñanza, proveedor de recursos	

Tabla 2.2. Algunas tipologías, según Marquès (1998)

2.4. Clasificación de los programas didácticos

Una clasificación factible de los programas puede ser: tutoriales, simuladores, entornos de programación y herramientas de autor.

Los programas *tutoriales*, son programas que *dirigen* el aprendizaje de los alumnos mediante una teoría subyacente conductista de la enseñanza, guían los aprendizajes y comparan los resultados de los alumnos contra patrones, generando muchas veces nuevas ejercitaciones de refuerzo, si en la evaluación no se superaron los objetivos de aprendizaje.

En este grupo, se encuentran los programas derivados de la enseñanza programada, tendientes al desarrollo de habilidades, algunos de ellos son lineales y otros ramificados, pero en ambos casos de base conductual, siendo los ramificados del tipo interactivos.

Se han desarrollado modelos cognitivistas, donde se usa información parcial, y el alumno debe buscar el resto de la información para la resolución de un problema dado.

Dentro de esta categoría, están los sistemas tutoriales expertos o inteligentes, que son una guía para control del aprendizaje individual y brindan las explicaciones ante los errores, permitiendo su control y corrección.

Los programas *simuladores*, ejercitan los aprendizajes inductivo y deductivo de los alumnos mediante la toma de decisiones y adquisición de experiencia en situaciones imposibles de lograr desde la realidad, facilitando el aprendizaje por descubrimiento.

Los *entornos de programación*, tales como el Logo, permiten construir el conocimiento, paso a paso, facilitando al alumno la adquisición de nuevos conocimientos y el aprendizaje a partir de sus errores; y también conducen a los alumnos a la programación.

Las *herramientas de autor*, también llamadas "*lenguajes de autor*" permiten a los profesores construir programas del tipo tutoriales, especialmente a profesores que no disponen de grandes conocimientos de programación e informática, ya que usando muy pocas instrucciones, se pueden crear muy buenas aplicaciones hipermediales.

Algunos autores consideran que las *bases de datos para consulta*, son otro tipo de programas educativos, porque facilitan la exploración y la consulta selectiva, permitiendo extraer datos relevantes para resolver problemas, analizar y relacionar datos y extraer conclusiones. (Marquès, 1995).

Quedarían por analizar los programas usados como *herramientas de apoyo* tales como los procesadores de textos, planillas de cálculo, sistemas de gestión de bases de datos, graficadores, programas de comunicación, que no entran dentro de la clasificación de educativos, pero muchas veces son necesarios para la redacción final de trabajos, informes y monografías.

En la búsqueda permanente del mejoramiento de los procesos de enseñanza y de aprendizaje, se encuentra una herramienta poderosísima en los *sistemas hipermediales*, como un subconjunto del software educativo en general.

Se puede definir un sistema hipermedial como la combinación de hipertexto y multimedia.

Se entiende por *hipertexto al sistema de presentación de textos extensos con o sin imágenes donde se puede adicionar sonido, formando una red con nodos que son unidades de información, con enlaces y arcos dirigidos hacia otros nodos, la red no es más que un grafo orientado, que se aparta de la forma secuencial tradicional del libro. Multimedia es la presentación de la información con grandes volúmenes de texto, con imágenes fijas, dibujos con animación y vídeo digital. Por lo tanto la hipermedia es la combinación de hipertexto y multimedia.* (Nielsen, 1995).

Algunos autores como García López (1997) sostienen que a pesar de que el multimedia interactivo es anterior a la aparición de las redes y a la realidad virtual y que el prefijo "*hiper*" engloba también a dichas fusiones interactivas.

2.5. Las funciones del software educativo

Las funciones del software educativo, están determinadas de acuerdo a la forma de uso de cada profesor. En la Tabla 2.3, se describen en forma sintética algunas de las funciones que pueden realizar los programas.

2.6. El rol docente y los usos del software.

El estilo docente ha cambiado a causa de la introducción de las computadoras en el aula, desde el tradicional suministrador de información, mediante clases magistrales a facilitadores, pudiendo de este modo realizar un análisis más preciso del proceso de aprendizaje de sus alumnos y una reflexión acerca de su propia práctica.

Función	Descripción
Informativa	Presentan contenidos que proporcionan una información estructurado-ra de la realidad. Representan la realidad y la ordenan. Son ejemplos, las bases de datos, los simuladores, los tutoriales.
Instructiva	Promueven actuaciones de lo estudiantes encaminadas a facilitar el logro de los objetivos educativos, el ejemplo son los programas tutoriales.
Motivadora	Suelen incluir elementos para captar en interés de los alumnos y enfiarlo hacia los aspectos más importantes de las actividades.
Evaluadora	Al evaluar implícita o explícitamente, el trabajo de los alumnos.
Investigadora	Los más comunes son: las bases de datos, los simuladores y los entornos de programación.
Expresiva	Por la precisión en los lenguajes de programación, ya que el entorno informático, no permite ambigüedad expresiva.
Metalingüística	Al aprender lenguajes propios de la informática.
Lúdica	A veces, algunos programas refuerzan su uso, mediante la inclusión de elementos lúdicos.
Innovadora	Cuando utilizan la tecnología más reciente.

Tabla 2.3: funciones del software educativo según Marquès (1995)

Los “mediadores pedagógicos”, son el vínculo entre los estudiantes (sujetos) y los contenidos. La concepción tradicional de docente informante, ha cambiado hacia el facilitador o guía y tutor, y una nueva perspectiva es el uso de mediadores tales como los programas educativos, sean o no hipermediales, con toda la gama de posibles matices intermedios.

Cuando se desea aplicar un software educativo en un contexto áulico, se debe tener en cuenta, que para algunas asignaturas resulta más difícil incorporar el recurso informático al aula. Estas formas de incorporación están directamente relacionadas con las diferentes actitudes del docente, de acuerdo a su estilo, como se puede observar en la Tabla 3.4.

Magistral o de informante	El docente deja de ser la fuente principal de información de la clase.
Auxiliar	El docente conserva su función de informante, articulando diferentes medios.
Aplicativa	Se integra el rol del docente y se consolida el trabajo individual y grupal
Interactiva	Se favorece la comunicación, la construcción conjunta del conocimiento.

Tabla 3.4: el rol docente y el software educativo.

Los nuevos entornos de enseñanza y aprendizaje, exigen nuevos roles en profesores y alumnos, la perspectiva tradicional en todos los niveles educativos y especialmente en la educación superior del profesor como fuente única de información se ha transformado hacia un del profesor guía y consejero acerca del manejo de las fuentes apropiadas de información y desarrollador de destrezas y hábitos conducentes a la búsqueda, selección y tratamiento de la información.

Los estudiantes ya no son *receptores pasivos*, sino que se convierten en *alumnos activos* en la búsqueda, selección, procesamiento y asimilación de información.

La concepción tradicional ha cambiado hacia una *cultura del aprendizaje*, o sea una educación generalizada y una formación permanente, dentro de una avalancha constante de información. Es en esta cultura del aprendizaje, en la que el profesor debe encarar el rol de *gerenciador de los saberes y desarrollador de habilidades* que permitan a sus alumnos utilizar el análisis crítico y reflexivo.

2.7. Las funciones del profesor y los materiales didácticos

Los materiales didácticos, se pueden definir como *"el conjunto de medios materiales que intervienen en el acto didáctico, facilitando los procesos de enseñanza y de aprendizaje"*. Sus fines centrales persiguen facilitar la comunicación entre el docente y el estudiante para favorecer a través de la intuición y el razonamiento un acercamiento comprensivo de las ideas a través de los sentidos. (Eisner, 1992). Estos materiales didácticos constituyen la variable dependiente del proyecto pedagógico y del entorno de aprendizaje que se trate.

La utilización de software educativo como material didáctico, cambia la manera en la cual los profesores estimulan el aprendizaje en sus clases; cambia el tipo de interacción entre alumnos y docentes y por lo tanto cambia el rol y las funciones del profesor. En la Tabla 2.5 se presenta un resumen de dichas funciones:

Función	Características
Como proveedor de recursos	Muchas veces el profesor tiene que adaptar los materiales de un cierto paquete educativo a las características de la clase y a los fines que él plantea en ese momento.
Como Organizador	Cuando se usan computadoras, hay muchas formas de organizar su uso en el aula y variando de acuerdo a los diferentes estilos docentes. También se debe tener en cuenta la graduación del tiempo de interacción con las máquinas, ya que es en los diálogos en clase donde se produce gran parte del aprendizaje.
Como tutor	Hay profesores que usan un software para centrar las actividades. El profesor trabaja con un sólo alumno o un grupo pequeño, realizando actividades de tutoría como: razonar y buscar modelos o respuestas.
Como Investigador	A nivel áulico, el uso de software puede dar a los profesores ideas sobre los procesos de aprendizaje y de las dificultades de sus alumnos. En este papel de investigadores, los docentes, usan al software como una herramienta diagnóstica.
Como facilitador	Esta es la responsabilidad principal del docente, como facilitadores del aprendizaje de los estudiantes y la que no debe olvidarse, con la aparición de las demás funciones que surgen con la introducción del uso de las computadoras en el aula.

Tabla 2.5: Las funciones del profesor (Squires y Mc Dougall, 1994)

2.8. Los objetivos educativos⁷

Se entiende por objetivo *"algo"*⁸ que se quiere lograr, o sea un estado al cual se quiere arribar (aunque no siempre hay que pensarlo desde el punto de vista conductual). A fin de enunciar correctamente un objetivo, de manera que sea tal y no la mera expresión de un deseo, es necesario que existan en él los tres elementos siguientes:

- *Intención*: Es el fin de todo objetivo. La intención debe ser clara y estar concretamente expresada en el enunciado del objetivo, debe enunciar con toda certeza y precisión qué se propone alcanzar. La intención debe ser no sólo concreta, sino también real. Un objetivo, al enunciar una intención debe proponer un fin concreto, de esta manera se podrá determinar con toda exactitud cuándo se logró alcanzar el fin propuesto. Si la intención no es concreta, nunca se podrá saber si el objetivo está cumplido.
- *Medida*: Es el elemento que vuelve al objetivo mensurable y esa cualidad de ser mensurable es la otorga la certeza de cumplimiento.
- *Plazo*: Es el período durante el cual debe lograrse el objetivo.

La formulación de los objetivos sirve para:

- *Fijar la situación actual*: El hecho de determinar un estado final a lograr, obliga, indefectiblemente, a fijar una situación actual. Si se quiere lograr algo en el futuro, se debe partir de una determinada situación en el presente. Aquí es donde se pone de manifiesto la importancia de la evaluación inicial o diagnóstica.⁹
- *Determinar el estado final a lograr*: Por medio de evaluaciones sumativas¹⁰ o finales.
- *Determinar las estrategias a emplear*: Si se tiene una situación actual y un estado final, es evidente que se hace necesario un accionar que permita lograrlo. Las estrategias son opciones alternativas con un gran número de posibilidades diferentes. En la selección de las mismas se tienen en cuenta, además de su efi-

⁷ Esta sección forma parte de un Trabajo presentado a la Dra. Beatriz Fainholc en el *Seminario de Sistemas Multimediales Aplicados a la Educación*. UTN. Buenos Aires. 1998.

⁸ depende de qué sea este *"algo"*, serán distintos los aspectos buscados.

⁹ diseñada en base a una serie ejercicios específicos, para saber en qué etapas de su desarrollo se encuentra el alumno, especialmente si está en la etapa de desarrollo formal, según la visión de piagetiana.

¹⁰ de producto final tal como se refieren Bork (1986) y Coll (1994).

ciencia: costo, tiempo de acción, sencillez, facilidades operativas, requerimiento de laboratorio, de biblioteca, sistemas informáticos, etc.

- *Medir los resultados*: Mediante evaluaciones formativas¹¹ (de procesos) y sumativas o finales. Puede ser parcial y sumativa, no necesariamente formativa.

2.9. Las actividades de comprensión a desarrollar por los alumnos

Entre las *actividades de comprensión* o "*procesos de pensamiento*" que los alumnos pueden desarrollar al interactuar con los programas educativos, se pueden mencionar:

- Explicar relaciones causa efecto.
- Formular conclusiones válidas.
- Describir limitaciones de los datos.
- Confrontar conocimientos nuevos con previos.
- Clasificar y seleccionar información.
- Producir, organizar y expresar ideas.
- Elaborar mapas conceptuales (teniendo en cuenta la reconciliación integradora y la diferenciación progresiva)
- Integrar el aprendizaje en diferentes áreas.
- Inferir correctamente.
- Evaluar el grado de adecuación de las ideas.
- Presentar argumentos pertinentes frente a fenómenos.
- Defender un punto de vista y fundamentar criterios.
- Resolver problemas elaborando estrategias metacognitivas.

La comprensión, implica el compromiso reflexivo del alumno con el contenido de enseñanza y la habilidad para articular significativamente el material comunicado por acciones de guía (Cediproe, 1998).

Entre *los objetivos de los programas educativos* se pueden mencionar:

- Crear expectativas en el estudiante y estimular la planificación de su aprendizaje.
- Dirigir la atención del estudiante y permitir que inicie su aprendizaje por diferentes caminos de acceso. (tiene gran importancia desde lo cognitivo).
- Asegurar situaciones de aprendizaje significativo.
- Aprovechar la posibilidad de usar imágenes, animaciones, simulaciones y sonidos.
- Desarrollar y hacer consciente el uso de diferentes estrategias:
 - de procesamiento de la información.
 - de producción y uso de la información.
 - De recreación de la información.
- Estimular la generalización y transferencia de lo aprendido.
- Ofrecer situaciones de resolución de problemas.
- Proveer retroalimentación constante e informar acerca de los progresos en el aprendizaje. (Zangara, 1998).

2.10. La motivación

Alessi y Trollip (1985), consideran que existe una motivación extrínseca independiente del programa utilizado, y una intrínseca inherente en la instrucción y recomiendan criterios para su promoción, como el uso de juegos, de exploración, de desafíos, incentivación de la curiosidad del estudiante, teniendo en cuenta un balance entre la motivación y el control del programa aplicado.

Las bases teóricas pueden ser provistas por alguna de las teorías de la motivación permitiendo crear desafíos, curiosidad, control y fantasía y con un diseño motivacional que mantenga la atención a través del mismo. Los estudiantes deben poder ver la utilidad de resolución de problemas.

Ausubel (1987) sostiene que el papel de la motivación en el aprendizaje es uno de los problemas más controvertidos de los teóricos de la psicología, y que aún las posiciones son muy encontradas. En la Tabla 2.6, se pueden ver la clasificación de los diferentes tipos de motivación.

Tipos	Características
<i>Intrínseca</i>	Es la que proviene del interior del sujeto por su compromiso con la tarea.
<i>Relacionada con el yo</i>	Se relaciona con la autoestima, con el no percibirse inferior que los demás
<i>Centrada en la</i>	Se relaciona con la satisfacción afectiva que produce la aceptación, aprobación

¹¹ de proceso o parcial, tal como se refieren Bork (1986) y Coll (1994).

valoración social	o aplauso por parte de personas consideradas superiores.
Extrínseca	Centrada en recompensas externas, se relaciona con premios y/o castigos

Tabla 2. 6: Tipos de motivación (Guiraud, 1997)

La motivación intrínseca es superior a la extrínseca y para lograrla, quizás la manera más eficaz es mediante el entusiasmo propio del docente por lo que hace.

Para ello se debe considerar la creación de nuevos intereses en los alumnos como uno de los objetivos de la intervención pedagógica, teniendo en cuenta la escala motivacional de Maslow¹² con necesidades fisiológicas, de supervivencia, de seguridad, de amor, de pertenencia, de aceptación, de autoestima, de autorrealización.

2.11. La organización y presentación de los contenidos

La selección de los contenidos, es uno de los problemas recurrentes en educación que comienzan con el planteo del docente de qué enseñar, para qué enseñar y cómo enseñar.

En el análisis del “*qué enseñar*”, de acuerdo a los “*principios básicos*”, ejes de todo el desarrollo, el docente que va a desarrollar software o que trabaja en un equipo de desarrollo, debe seleccionar la información a presentar y transmitir, determinando los contenidos y también su organización que dependerá de la subdivisión del eje temático principal en bloques de contenido y en sub-bloques.

La organización en bloques y sub-bloques se realizará de tal forma que permitan de navegación en sentido horizontal, vertical y transversal y deberán estar de acuerdo a las diferentes estrategias de búsqueda que se preparen desde alguna de las visiones de los diferentes paradigmas educativos.

Esta organización será acorde con el diseño de las pantallas más adecuado en cada caso, para la presentación de los contenidos.

2.12. La comunicación: Las interfaces humanas.

Gallego y Alonso (1997), ofrecen una guía metodológica para el diseño pedagógico de la interface de navegación, destacando la necesidad de un diseño adecuado tanto de la organización de los contenidos como de las estrategias de enseñanza y de aprendizaje. Esta interface es fundamental, ya que es el sistema de recursos mediante el cual el usuario interactúa con el sistema informático. Estos recursos implican tener en cuenta aspectos técnicos, de funcionamiento de la interface y también los cognitivos y emocionales resultantes de la interacción usuario-computadora.

El diálogo entre el usuario y el sistema informático debe ser lo más sencillo posible y debe proveerle los recursos necesarios para la navegación y obtención de la información buscada.

La interface es el elemento clave de comunicación o aspecto fundamental de diseño y presentación de los contenidos. Actualmente, se diseñan interfaces orientadas al usuario, lo más cercanas posible al lenguaje humano, incluyendo el modo de presentar la información en la pantalla y las funcionalidades brindadas al usuario para interactuar con el programa.

Según Gallego y Alonso (1997), las características principales de una interface orientada al usuario deben ser:

- Facilidad de manejo: la mejor interface de usuario es aquella que requiere el menor esfuerzo de aprendizaje.
- Originalidad: para promover la motivación y exploración.
- Homogeneidad: requiere de una interface con funciones claras para moverse de en el programa, incluyendo un mapa general.
- Versatilidad: que pueda incorporar nuevas funciones específicas.
- Adaptabilidad: deberá ofrecer modalidades de navegación de acuerdo al contenido, los destinatarios y el nivel de profundidad.
- Multimodalidad: con integración de modalidades de comunicación necesaria para cada concepto.
- Multidimensionalidad: para los diseños hipermediales.
- Agilidad: para que la interacción sea dinámica.
- Transparencia: cuanto más natural sea, será más fácil para el usuario acceder a los contenidos.
- Interactividad: darle al usuario un papel protagónico.
- Conectividad: para utilizar redes.

Respecto de las funciones, la interface debe tener una triple funcionalidad: utilidades, navegación e información.

En su artículo sobre los agentes de interface, Brenda Laurel (1990) señala como principales características de las mismas: son dar respuestas, actuar como agente, competencia y accesibilidad.

¹² Escala de Maslow A. H. (1943): “A theory of human motivation”, Psychological Review, July, págs. 370-396.

La metáfora navegacional a aplicar estará condicionada por el tipo de contenido, las características de los destinatarios y el lenguaje o herramienta de autor usado para desarrollar el software. Las metáforas más utilizadas son las de los menús: cerrados, abiertos o mixtos y las de los iconos; en este caso su utilización es mucho más intuitiva. La metáfora espacial, es aquella que usa la realidad como modelo, con escenarios que simulan la realidad misma. Un modelo de interface espacial son los paisajes de información, este modelo incluye conjuntos de datos, documentos interactivos, recorridos guiados, películas y actividades.

Como no hay una metáfora ideal de menú principal de usuario, se trata de brindarle una combinación de todas ellas dando al mismo la posibilidad de realizar su elección.

Las metáforas navegacionales están asociadas a las diferentes estrategias de aprendizaje. Cuando se preparan programas totalmente interactivos, ramificados, con caminos de aprendizaje múltiples a elección del alumno, los estilos de aprendizaje pueden convertirse en un elemento más a tener en cuenta en el diseño didáctico (Alonso, 1992).

Las funciones de navegación permiten saber al usuario dónde está en cada momento, de dónde viene y a dónde puede ir. Los modelos de organización de la información para estructurar los contenidos de las aplicaciones educativas son muy diversos. Florín (1990) plantea una estructura multidimensional que permite al usuario acceder a la información sobre la base de distintos intereses.

La metodología recomendada por Gallego y Alonso (1997), para aplicar la interface al ámbito educativo y la formación, se basa en los siguiente principios:

- Ofrecer al usuario la posibilidad de que se sienta protagonista.
- Presentar los contenidos de forma atractiva y de fácil manejo.
- Combinar diferentes metáforas de navegación interactivas.
- Prever diversas funcionalidades de la interface de navegación en función del tipo de contenido, del destinatario y de los niveles de profundidad previstos.
- Considerar las normas de calidad en el diseño.

Las principales especificaciones de una interface de aprendizaje son:

- Facilidad de manejo.
- Ayudas alternativas.
- Sistema de seguimiento del alumno que permita el diagnóstico de progreso realizado en función del grado de logro de los objetivos.

En la Tabla 2.7 se pueden observar una clasificación de los diferentes tipos de pantallas a utilizar de acuerdo a los objetivos didácticos perseguidos.

Tipos de pantallas	Objetivos didácticos
<i>Presentación del programa</i>	<ul style="list-style-type: none"> – Captar la atención – Generar, dirigir, motivar y/o aumentar la motivación
<i>Pantallas de antesala o de anticipación</i>	<ul style="list-style-type: none"> – Anticipar los conceptos a aprender
<i>Pantallas de presentación de información simple</i>	<ul style="list-style-type: none"> – Presentar información: – Nueva y relevante – Relacionada con algún concepto posterior
<i>Pantallas de presentación de información compleja: relación de información simple</i>	<ul style="list-style-type: none"> – Integrar los conceptos en conceptos complejos
<i>Pantallas de integración y síntesis de la Información</i>	<ul style="list-style-type: none"> – Integrar los conceptos en categorías
<i>Pantallas de actividades y resolución de problemas</i>	<ul style="list-style-type: none"> – Autoevaluar gradualmente el aprendizaje – Reorganizar y aplicar la nueva información – Transferir el aprendizaje a situaciones nuevas
<i>Pantallas de presentación de información de control</i>	<ul style="list-style-type: none"> – Informar acerca de la marcha del aprendizaje
<i>Interface de acceso a otras fuentes de Información</i>	<ul style="list-style-type: none"> – Acceder a fuentes complementarias de información – Realizar consultas a tutores – Relacionarse virtualmente con otros compañeros de estudios.

Tabla 2.7: Objetivos de los diferentes tipos de pantallas (Zangara, 1998)

2.13. La planificación didáctica.

Finalmente, una buena planificación didáctica para aplicación de un programa de computadora debe considerar aspectos tales como:

- *La inserción del programa en el currículum:* se deberá indicar para qué nivel educativo está dirigido el software y si está de acuerdo a un determinado currículum.
- *Los objetivos perseguidos:* constituyen el “para qué” de la propuesta educativa y la dirección de toda la acción educadora. César Coll (1994) dice que es la conducta esperable y que depende de la teoría del aprendizaje. Coll lo plantea como estrategias de pensamiento que se desea que el alumno realice, puntualizando las aspiraciones a corto y a largo plazo Ausubel (psicólogo cognitivo) habla de predisposición sin referirse a los procedimientos, usando estrategias cognitivas. Ampliando el esquema propuesto por Romiszowski (1981) en Coll (1994), quien estableció que a la concreción de las intenciones educativas puede accederse desde los contenidos, desde los resultados o desde las actividades, se debe agregar la posibilidad de acceder al conocimiento desde los medios, que atraviesan la realidad desde una visión tecnológica. Esta visión consiste en abordar la educación desde el paradigma *teleinformático*. Cuando se plantean los objetivos tanto para una asignatura, como en este caso de un software de un determinado tema en particular, el objetivo es el estado final logrado a partir de un estado inicial definido, este estado final real no siempre coincide con el valor teórico o probable a alcanzar en un tiempo definido. Existe un grado de apartamiento que es cuantificable y minimizar este apartamiento sería deseable.
- *Las características de los destinatarios:* hay que realizar una descripción en términos de edad, prerrequisitos de contenidos y habilidades, nivel educativo formal o informal.
- *Los contenidos desarrollados:* los contenidos se pueden abordar de distintas maneras. Desde el punto de vista cognitivo los contenidos son casi más importantes que los objetivos, consiste en una delimitación de qué. Un ejemplo son las estructuras de mapas conceptuales como un representación gráfica de las relaciones de conceptos y el aprendizaje significativo. La estrategia de trabajo de Novak (1988) es el armado de mapas conceptuales para la toma de decisiones.
- *Metodología y actividades a desarrollar:* aquí el docente debe determinar de acuerdo a su metodología de aplicación del programa, cuáles son las actividades que va a desarrollar con sus alumnos, indicando si usará el software como material de apoyo, por ejemplo, si utilizará proyecciones como complementos y una sola computadora, o si los alumnos trabajarán en grupos o en forma individual. También debe quedar claro cuáles son los procesos de pensamiento que se pretende desarrollar en los alumnos a partir de la interacción como por ejemplo: comparar, discriminar, resumir, globalizar, analizar, concatenar, experimentar, construir, negociar, discutir, investigar, evaluar, etc.
- *Recursos necesarios, medios y tiempo de interacción:* en la planificación didáctica deben quedar especificados los recursos necesarios, los medios indispensables y el tiempo que durará la interacción con el software. En el caso particular de un software realizado por encargo y para apoyo del docente, no se puede cuantificar en forma precisa este tiempo, como para arribar a un resultado óptimo. Cuando se habla de software de apoyo, el tiempo de interacción del alumno con el programa mediado en términos absolutos no sirve, porque el programa fue diseñado para usarlo de soporte, con complementos por parte del docente, que no forman parte del programa mismo. Por este motivo, para un alumno principiante en el tema, el software se potencia con las explicaciones adicionales del docente, pero si luego queda a disposición de los alumnos que pueden usarlo y verlo cuantas veces deseen, hasta lograr dominio del tema la estimación del tiempo aquí, carece de sentido.
- *Evaluación de los aprendizajes:* la instancia de evaluación del proceso de enseñanza y aprendizaje, para este tipo de producto, es quizás la más difícil, ya que evaluar un software significa basarse en los resultados alcanzados por los alumnos en las pruebas diseñadas de acuerdo a la teoría educativa aplicada. Ya sea mediante acercamiento a los objetivos, o por desarrollo y estimulación de procesos mentales y significatividad de aprendizajes.

3. La ingeniería de software

3.1. Resumen

En esta sección se desea presentar los fundamentos en los que se basa el software educativo (sección 3.2): los métodos, las herramientas y los procedimientos que provee la ingeniería de software a fin de considerarlos para el desarrollo de los programas didácticos. Se describen y analizan los paradigmas principales del ciclo de vida (sección 3.2) a la luz de la visión de Mario Piattini, desde la cascada tradicional hasta los actuales orientados a objetos (sección 3.3).

Se destaca la necesidad de una metodología para el desarrollo de productos lógicos y se describen las más importantes (sección 3.4). A fin de seleccionar el ciclo de vida adecuado para cada desarrollo, se analizan las actividades de cada uno de los procesos del mismo (sección 3.5). Por último se define calidad del software y la normativa vigente (sección 3.6) para un proyecto de software y se hace una revisión de las métricas de calidad comúnmente usadas.

3.2. Fundamentos

Uno de los problemas más importantes con los que se enfrentan los ingenieros en software y los programadores en el momento de desarrollar un software de aplicación, es la falta de marcos teóricos comunes que puedan ser usados por todas las personas que participan en el desarrollo del proyecto informático.

El problema se agrava cuando el desarrollo corresponde al ámbito educativo debido a la inexistencia de marcos teóricos interdisciplinarios entre las áreas de trabajo.

Si bien, algunos autores como Galvis (1996) reconocen la necesidad de un marco de referencia, teniendo en cuenta que se debe lograr la satisfacción de los requisitos en las diversas fases del desarrollo, de lo que constituye un material didáctico informatizado; esta necesidad sigue vigente, aunque en la mayoría de los casos analizados, se trata de software hipermedial diseñado a partir de herramientas de autor.

Marquès (1995), es otro de los autores que plantean un ciclo de desarrollo para software educativo de programas en diez etapas, con una descripción detallada de las actividades y recursos necesarios para cada una de ellas. El inconveniente principal de esta metodología es que centra el eje de la construcción de los programas educativos en el equipo pedagógico, otorgándole el rol protagónico.

Es por este motivo, que en este capítulo se sintetizan las metodologías, métodos, herramientas y procedimientos de la ingeniería de software, que deben ser utilizados para lograr un producto de mejor calidad desde el punto de vista técnico. Su conocimiento y aplicación conjuntamente con las teorías: educativa, epistemológica y comunicacional permitirán el logro de un producto óptimo desde el punto de vista educativo.

Cabe recordar una de las primeras definiciones de ingeniería de software propuesta por Fritz Bauer en la primera conferencia importante dedicada al tema (Naur, 1969) como: *"El establecimiento y uso de principios de ingeniería robustos, orientados a obtener software económico y que funcione de manera eficiente sobre máquinas reales"*.

Posteriormente se han propuesto muchas definiciones destacando la importancia de base teórica ingenieril para el desarrollo del software.

"La ingeniería del software surge a partir de las ingenierías de sistemas y de hardware, y considera tres elementos clave: que son los métodos, las herramientas y los procedimientos que facilitan el control del proceso de desarrollo de software y brinda a los desarrolladores las bases de la calidad de una forma productiva". (Pressman, 1993).

La ingeniería de software está compuesta por una serie de modelos que abarcan los métodos, las herramientas y los procedimientos. Estos modelos se denominan frecuentemente *paradigmas de la ingeniería del software* y la elección de un paradigma se realiza básicamente de acuerdo a la naturaleza del proyecto y de la aplicación, los controles y las entregas a realizar.

Debido a las características particulares de los desarrollos educativos, ya que se deben tener en cuenta los aspectos pedagógicos y de la comunicación con el usuario, en cada caso en particular, la respuesta a la problemática debe basarse en una adaptación de los actuales paradigmas de desarrollo a las teorías de aprendizaje que permitan satisfacer una demanda en especial.

Para la construcción de un sistema de software, el proceso puede describirse sintéticamente como: la obtención de los requisitos del software, el diseño del sistema de software (diseño preliminar y diseño detallado), la implementación, las pruebas, la instalación, el mantenimiento y la ampliación o actualización del sistema.

El proceso de construcción está formado por etapas que son: la obtención de los requisitos, el diseño del sistema, la codificación y las pruebas del sistema. Desde la perspectiva del producto, se parte de una necesidad, se especifican los requisitos, se obtiene el diseño del mismo, el código respectivo y por último el sistema de software. Algunos autores sostienen que el nombre *ciclo de vida* ha sido relegado en los últimos años, utilizando en su lugar *proceso de software*, cambiando la perspectiva de producto a proceso. (J. Juzgado, 1996)

El software o producto, en su desarrollo pasa por una serie de etapas que se denominan ciclo de vida, siendo necesario, definir en todas las etapas del ciclo de vida del producto, los procesos, las actividades y las tareas a desarrollar.

Por lo tanto, se puede decir que: *"se denomina ciclo de vida a toda la vida del software, comenzando con su concepción y finalizando en el momento de la desinstalación del mismo"*. (Sigwart et al., 1990), aunque a veces, se habla de ciclo de desarrollo, para denominar al subconjunto del ciclo de vida que empieza en el análisis y finaliza la entrega del producto.

Un ciclo de vida establece el orden de las etapas del proceso de software y los criterios a tener en cuenta para poder pasar de una etapa a la siguiente.

El tema del ciclo de vida ha sido tratado por algunas organizaciones profesionales y organismos internacionales como la IEEE (Institute of of Electrical and Electronics Engineers) y la ISO/IEC (International Standards Organization/International Electrochemical Commission), que han publicado normas tituladas *"Standard for Developing Software Life Cycle Processes"* (Estándar IEEE para el desarrollo de procesos del ciclo de vida del software) (IEEE, 1991) y *"Software life-cycle process"* (Proceso de ciclo de vida del software) (ISO, 1994).

Según la norma 1074 IEEE se define al ciclo de vida del software como *"una aproximación lógica a la adquisición, el suministro, el desarrollo, la explotación y el mantenimiento del software"* y la norma ISO 12207

define como modelo de ciclo de vida al “marco de referencia, que contiene los procesos, las actividades y las tareas involucradas en el desarrollo, la explotación y el mantenimiento de un producto de software, abarcando la vida del sistema desde la definición de requisitos hasta la finalización de su uso”. Ambas consideran una actividad como un subconjunto de tareas y una tarea como una acción que transforma las entradas en salidas. (Piattini, 1996).

3.3. Los procesos del ciclo de vida del software

Según la norma ISO 12207-1, las actividades que se pueden realizar durante el ciclo de vida se pueden agrupar en cinco procesos principales, ocho de soporte y cuatro procesos generales de la organización, así como un proceso que permite adaptar el ciclo de vida a cada caso concreto.

En la Tabla 3.1 se describen los grupos de procesos citados.

Procesos Principales (Aquellos que resultan útiles a las personas que inician o realizan el desarrollo, explotación o mantenimiento durante el ciclo de vida).	Adquisición	Contiene las actividades y tareas que el usuario realiza para comprar un producto
	Suministro	Contiene las actividades y tareas que el suministrador realiza
	Desarrollo	Contiene las actividades de análisis de requisitos, diseño, codificación, integración, pruebas, instalación y aceptación.
	Explotación	También se denomina operación del software.
	Mantenimiento	Tiene como objetivo modificar el software manteniendo su consistencia.
Procesos de soporte (se aplican en cualquier punto del ciclo de vida)	Documentación	Registra la información producida en cada proceso o actividad del ciclo de vida
	Gestión de la configuración	Aplica procedimientos para controlar las modificaciones
	Aseguramiento de la calidad	Para asegurar que todo el software cumple con los requisitos especificados de calidad.
	Verificación	Para determinar si los requisitos están completos y son correctos.
	Validación	Para determinar si cumple con los requisitos previstos para su uso.
	Revisión	Para evaluar el estado del software en cada etapa del ciclo de vida
	Auditoría	Para determinar si se han cumplido los requisitos, planes y el contrato.
	Resolución de los problemas	Para asegurar el análisis y la eliminación de problemas encontrados durante el desarrollo.
Procesos de la Organización (Ayudan a la organización en general).	Gestión	Contiene actividades genéricas de la organización como planificación, seguimiento, control, revisión y evaluación.
	Mejora	Sirve para establecer, valorar, medir, controlar y mejorar los procesos del ciclo de vida del software.
	Infraestructura	Incluye la infraestructura necesaria: hardware, software, herramientas, técnicas, normas e instalaciones para el desarrollo, la explotación o el mantenimiento.
	Formación	Para mantener al personal formado: incluyendo el material de formación y el plan de formación.

Tabla 3.1: Los procesos del ciclo de vida del software según ISO 12207-1

3.3.1. El modelo en cascada

La versión original del modelo en cascada, fue presentada por Royce en 1970, aunque son más conocidos los refinamientos realizados por Boehm (1981), Sommerville (1985) y Sigwart et al. (1990). En este modelo, el producto evoluciona a través de una secuencia de fases ordenadas en forma lineal, permitiendo iteraciones al estado anterior. El número de etapas suele variar, pero en general suelen ser:

- Análisis de requisitos del sistema.
- Análisis de requisitos del software.
- Diseño preliminar.
- Diseño detallado.
- Codificación y pruebas.

- Explotación (u operación) y mantenimiento.

Las características de este modelo son:

- Cada fase empieza cuando se ha terminado la anterior.
- Para pasar a la fase posterior es necesario haber logrado los objetivos de la previa.
- Es útil como control de fechas de entregas.
- Al final de cada fase el personal técnico y los usuarios tienen la oportunidad de revisar el progreso del proyecto.

Mc Cracken y Jackson (1982) han realizado algunas críticas al modelo:

- Sostienen que los proyectos reales rara vez siguen una linealidad tal, y que casi siempre hay iteraciones que van más allá de la etapa anterior.
- Además, como el sistema no estará en funcionamiento hasta finalizar el proyecto, el usuario, recibe el primer producto al haber consumido casi la totalidad de los recursos.

Otra limitación que se argumenta es que el modelo supone que los requisitos pueden ser “congelados” antes de comenzar el diseño y esto significa un hardware asociado durante el tiempo que dure el proyecto.

3.3.2. El modelo incremental, de refinamiento sucesivo o mejora iterativa.

Las etapas son las mismas que en el ciclo de vida en cascada y su realización sigue el mismo orden, pero corrige la problemática de la linealidad del modelo en cascada. Este modelo incremental fue desarrollado por Lehman (1984), y en cada paso sucesivo agrega al sistema nuevas funcionalidades o requisitos que permiten el refinado a partir de una versión previa. El modelo es útil cuando la definición de los requisitos es ambigua y poco precisa, porque permite el refinamiento, o sea se pueden ampliar los requisitos y las especificaciones derivadas de la etapa anterior.

Uno de los problemas que puede presentar es detección de requisitos tardíamente, siendo su corrección tan costosa como en el caso de la cascada.

3.3.3. Prototipado evolutivo

El uso de prototipos se centra en la idea de ayudar a comprender los requisitos que plantea el usuario, sobre todo si este no tiene una idea muy acabada de lo que desea. También pueden utilizarse cuando el ingeniero de software tiene dudas acerca de la viabilidad de la solución pensada. Esta versión temprana de lo que será el producto, con una funcionalidad reducida, en principio, podrá incrementarse paulatinamente a través de refinamientos sucesivos de las especificaciones del sistema, evolucionando hasta llegar al sistema final.

Al usar prototipos, las etapas del ciclo de vida clásico quedan modificadas de la siguiente manera:

- Análisis de requisitos del sistema.
- Análisis de requisitos del software.
- Diseño, desarrollo e implementación del prototipo
- Prueba del prototipo.
- Refinamiento iterativo del prototipo.
- Refinamiento de las especificaciones del prototipo.
- Diseño e implementación del sistema final.
- Explotación (u operación) y mantenimiento.

Si bien el modelo de prototipos evolutivos, fácilmente modificables y ampliables es muy usado, en muchos casos pueden usarse prototipos descartables para esclarecer aquellos aspectos del sistema que no se comprenden bien. (J. Juzgado, 1996).

3.3.4. El modelo en espiral de Boehm

En 1988 Boehm propone el modelo en espiral, para superar algunas de las limitaciones del modelo en cascada. La espiral se forma a partir de una serie de ciclos de desarrollo y va evolucionando. Los ciclos internos de la espiral denotan análisis y prototipado y los externos el modelo clásico. En la dimensión radial están los costos acumulativos y la dimensión angular representa el progreso realizado en cada etapa.

En cada ciclo se empieza identificando los objetivos, las alternativas y las restricciones del mismo. Se deben evaluar las alternativas de solución respecto de los objetivos, considerando las restricciones en cada caso. Es en este momento en que se puede llevar a cabo el siguiente ciclo.

Una vez finalizado, comienza el planteo de un nuevo ciclo. Durante cada ciclo de la espiral, aparece el análisis de riesgos, identificando situaciones que pueden hacer fracasar el proyecto, demorarlo o incrementar su costo. El análisis de riesgo representa la misma cantidad de desplazamiento angular en cada etapa y el volumen barrido denota el incremento de los niveles de esfuerzo requeridos para el análisis de riesgo. (Boehm, 1988)

Pueden resumirse las siguientes ventajas respecto de los modelos anteriores:

- Se explicitan las diferentes alternativas posibles para lograr los objetivos.

- El modelo tiene en cuenta la identificación de los riesgos para cada alternativa y los modos de controlarlos.
- Este modelo es adaptable a desarrollos de todo tipo y no establece una diferencia entre desarrollo de software y mantenimiento del sistema

El modelo en espiral se adapta bien en la mayoría de los casos. En el caso de proyectos de riesgo, se hace necesaria la presencia de un experto en evaluación de riesgos para identificar y manejar las fuentes de riesgos potenciales del mismo.

3.3.5. Los modelos orientados al objeto

La tecnología de objetos permite acelerar el desarrollo de sistemas de manera iterativa e incremental, permitiendo la generalización de los componentes para que sean reutilizables. Piattini (1996) presenta algunos de los modelos propuestos desde esta perspectiva.

Los modelos a tener en cuenta son:

- El modelo de agrupamiento o de clúster:** según Meyer (1990), los modelos usuales de ciclo de vida se basan en una cultura del proyecto, mientras que los desarrollos orientados al objeto están basados en el producto, entendido como elementos software reutilizables, cuyo beneficio económico aparece a largo plazo. En el modelo de *agrupamiento* se tiene en cuenta esta nueva fase de generalización que aparece combinada con la fase de validación. El concepto clave de este modelo es el de agrupamiento, que es un conjunto de clases relacionadas con un objetivo común. Se crean así diferentes sub-ciclos de vida que se pueden solapar en el tiempo y cada agrupamiento depende de los desarrollados con anterioridad, ya que se utiliza un enfoque ascendente: se empieza por las clases más básicas que pueden estar incluidas en bibliotecas.
- El modelo fuente:** Fue desarrollado por Henderson-Sellers y Edwards (1990), y representa gráficamente el alto grado de iteración y solapamiento que hace posible la tecnología de objetos. En la base está el análisis de requisitos, a partir del cual va creciendo el ciclo de vida, cayendo sólo para el mantenimiento necesario a la piscina, que sería el repositorio de clases.
- El modelo de remolino:** Rumbaugh (1992), analiza las cuestiones que afectan al ciclo de vida en el desarrollo orientado al objeto. Rumbaugh considera que el modelo en cascada supone una sola dimensión de iteración, consistente en la fase del proceso.

Dimensiones	Descripción
Amplitud	Tamaño del desarrollo
Profundidad	Nivel de abstracción o detalle
Madurez	Grado de completitud, corrección y elegancia
Alternativas	Diferentes soluciones de un problema
Alcance	En cuanto a cambio en los requisitos

Tabla 3.2: Las diferentes dimensiones según Rumbaugh (1990).

Debido a la identificación de otras dimensiones como amplitud, profundidad, madurez, alternativas y alcance, este proceso sería un desarrollo multicíclico, fractal más que lineal, en forma de remolino (ver Tabla 3.2).

- Modelo pinball¹³:** es un modelo propuesto por Amler (1994), quien señala que el pinball es el que refleja realmente la forma en la que se desarrolla el software. En este modelo el proyecto completo o un subproyecto está representado por la pelota y el jugador es el equipo de desarrollo. En forma iterativa se procede a encontrar clases, atributos, métodos e interrelaciones (en la fase de análisis) y definir colaboraciones, herencia, agregación y subsistema (que se incluyen en el diseño). Un último paso es la programación, prueba e implementación, y como en el pinball, los pasos se pueden tomar en cualquier orden y de forma simultánea. Amler destaca que se puede jugar a lo seguro, con tecnologías y métodos probados, o al límite, con mayor riesgo, pero con probabilidades de conseguir buenos beneficios. Destaca que la habilidad y la experiencia son los factores más importantes.

Llorca et al. (1991) sostienen que estos modelos se caracterizan por el desarrollo orientado al objeto, ya que:

- Eliminan los límites entre fases, tornándose cada vez más difusos debido a la naturaleza interactiva del desarrollo orientado al objeto.
- Permiten una nueva forma de concebir los lenguajes de programación y su uso e incorporan bibliotecas de clases y otros componentes reutilizables.
- La forma de trabajo es muy dinámica, debido al alto grado de iteración y solapamiento.

¹³ En relación al juego.

Los expertos en tecnologías de objetos, proponen un desarrollo interactivo e incremental, existiendo un ciclo evolutivo del sistema en el sentido análisis-diseño-instrumentación-análisis, que se lleva a cabo en forma iterativa. Algunas metodologías hablan de diseños o metodologías recursivos pero como incrementales. (Piattini, 1996)

Goldberg (1993), dice que *“la idea de la integración incremental es la diferencia clave de cómo debe ser gestionado un proyecto que utiliza tecnología orientada al objeto.”* Así las actividades de validación, verificación y aseguramiento de la calidad se pueden realizar para cada iteración de cada fase de cada incremento en el desarrollo del sistema, o sea en forma continuada.

Existen otros modelos de ciclo de vida, que no se han detallado en esta selección ya que aunque presenten ciertas potencialidades, no están muy extendidos. (J. Juzgado, 1996).

En el estándar IEEE 1074-1991 (IEEE, 1991) se detallan las fases del proceso base de construcción de software. Este estándar determina el *“conjunto de actividades esenciales que deben ser incorporadas dentro de un modelo de ciclo de vida del software y la documentación involucrada”*, pero estas actividades no están ordenadas en el tiempo.

3.4. La necesidad de una metodología de desarrollo

Para desarrollar un proyecto de software es necesario establecer un enfoque disciplinado y sistemático. Las metodologías de desarrollo influyen directamente en el proceso de construcción y se elaboran a partir del marco definido por uno o más ciclos de vida. (Piattini, 1996)

Según Piattini (1996), no hay un consenso entre los autores sobre el concepto de metodología, y por lo tanto no existe una definición universalmente aceptada. Sí hay un acuerdo en considerar a la metodología como *“un conjunto de pasos y procedimientos que deben seguirse para el desarrollo del software”*.

Maddison (1983) define metodología como un conjunto de filosofías, fases, procedimientos, reglas, técnicas, herramientas, documentación y aspectos de formación para los desarrolladores de sistemas de información. Por lo tanto, una metodología es un conjunto de componentes que especifican:

- Cómo se debe dividir un proyecto en etapas.
- Qué tareas se llevan a cabo en cada etapa.
- Qué salidas se producen y cuándo se deben producir.
- Qué restricciones se aplican.
- Qué herramientas se van a utilizar.
- Cómo se gestiona y controla un proyecto.

Generalizando, Piattini llega a la definición de metodología de desarrollo como *“un conjunto de procedimientos, técnicas, herramientas, y un soporte documental que ayuda a los desarrolladores a realizar nuevo software”*. Normalmente consistirá en fases o etapas descompuestas en subfases, módulos, etapas, pasos, etc. Esta descomposición ayuda a los desarrolladores en la elección de las técnicas a utilizar en cada estado del proyecto, facilitando la planificación, gestión, control y evaluación de los proyectos.

Sintetizando lo anterior, el autor dice que: *“una metodología representa el camino para desarrollar software de una manera sistemática”*.

Las metodologías persiguen tres necesidades principales:

- Mejores aplicaciones, tendientes a una mejor calidad, aunque a veces no es suficiente.
- Un proceso de desarrollo controlado, que asegure uso de recursos apropiados y costo adecuado.
- Un proceso estándar en la organización, que no sienta los cambios del personal.

Las metodologías a veces tienen diferentes objetivos, pero los más representativos pueden ser:

- Brindar un método sistemático, de modo de controlar el progreso del desarrollo.
- Especificar los requerimientos de un software en forma apropiada.
- Construir productos bien documentados y de fácil mantenimiento.
- Ayudar a identificar las necesidades de cambio lo más pronto posible.
- Proporcionar un sistema ágil que satisfaga a todas las personas involucradas.

Los procesos se descomponen hasta el nivel de tareas o actividades elementales, donde cada tarea está identificada por un procedimiento que define la forma de llevarla a cabo. Para aplicar un procedimiento se pueden usar una o más técnicas. Estas pueden ser gráficas con apoyos textuales, formales y determinan el formato de los productos resultantes en la tarea.

Para llevar a cabo las tareas se pueden usar herramientas software que automatizan la aplicación en determinado grado.

3.4.1. Evolución de las metodologías de desarrollo

Los primeros desarrollos no tuvieron una metodología definida, fueron totalmente artesanales y se los llamó desarrollos convencionales. Se caracterizaron por el aspecto monolítico de los programas y por una falta de control de lo que sucede en un proyecto. Estas limitaciones condujeron a una serie de problemas y por lo tanto a una búsqueda más sistemática en el desarrollo.

Como una posibilidad de "*nuevo orden*", surge el desarrollo estructurado, sobre la base de la programación estructurada, los métodos de análisis y el diseño estructurado. Esta nueva etapa es la piedra fundamental para la construcción de programas con métodos ingenieriles.

La programación estructurada, aparece en los sesenta en el ámbito científico y en los setenta pasa al ámbito empresarial. Tiene como punto de partida el establecimiento y uso de normas para la aplicación de estructuras de datos y control.

En los setenta, el enfoque estructurado, se extiende de la *fase de análisis* a la *fase de diseño* y las técnicas estructuradas se dirigen tanto a los aspectos técnicos como los relacionados con la gestión en la construcción de software.

Myers (1975), Yourdon y Constantine (1975) y Page-Jones (1980), en sus publicaciones, definen al módulo del programa como el componente básico de la construcción software, pasando luego a la normalización de la estructura los módulos de programa y al refinamiento posterior. Es este período comienzan a aplicarse medidas de calidad de los programas.

Según Gane y Sarsons (1977), y DeMarco (1979), consideraron que uno de los problemas de los programas desarrollados monolíticamente era que se necesitaba una total comprensión total de las especificaciones, por parte de los analistas, que en muchos casos eran ambiguas, y en otros eran obsoletas al llegar al final del proyecto.

La base de la programación y el diseño estructurados, es un análisis del problema usando el diseño *top-down* o *descendente*, con énfasis en las especificaciones funcionales. Se compone de diagramas, con textos de referencia de los mismos, y con independencia para que se puedan leer en forma parcial, con una redundancia mínima, de modo que los cambios no lo afecten en forma notable.

También, se puede destacar, que ha habido una evolución en cuanto al modelado de sistemas en tiempo real, modelado de datos y estudio de eventos. (Piattini, 1996).

El *paradigma orientado objetos*, que aparece más tarde, trata a los procesos y los datos en forma conjunta, modularizando la información y el procesamiento.

Aparece el Smalltalk, con énfasis en la abstracción de datos, considerando a los problemas como un conjunto de objetos de datos a los que se les adicionaba un conjunto de operaciones, pero se fundamenta en la abstracción, la modularidad y el ocultamiento de la información, derivados del diseño estructurado.

En los ochenta, aparecen el C++ y el object C y más tarde el Lenguaje ADA, del Departamento de Defensa (DoD) de los Estados Unidos para la definición y mejora de tecnologías basada en este lenguaje.

En general para los desarrollos de las metodologías orientadas al objeto se tomaron de los conceptos de técnicas estructuradas.

3.4.2. Características y clasificación de las metodologías

Se pueden enumerar una serie de características que debe tener la metodología y que influirán en el entorno de desarrollo:

- Reglas predefinidas.
- Determinación de los pasos del ciclo de vida.
- Verificaciones en cada etapa.
- Planificación y control.
- Comunicación efectiva entre desarrolladores y usuarios.
- Flexibilidad: aplicación en un amplio espectro de casos.
- De fácil comprensión.
- Soporte de herramientas automatizadas.
- Que permita definir mediciones que indiquen mejoras.
- Que permita modificaciones.
- Que soporte reusabilidad del software.

Las metodologías se pueden clasificar considerando tres dimensiones de acuerdo a la Tabla 3.3.

3.4.2.1. Metodologías estructuradas

Estas metodologías definen los modelos del sistema que representan los procesos, los flujos y la estructura de datos de un modo descendente, pasando de la una visión general del problema a un nivel de abstracción más sencillo, pudiendo centrarse en las funciones o procesos del sistema, en la estructura de datos o en ambas, dando lugar a los tipos de metodologías indicados en la Tabla 3.3.

Las metodologías orientadas a procesos como las de Gane y Sarsons (1979), DeMarco (1979) y Yourdon (1989), tienen como base la utilización de un método descendente para la descomposición funcional del problema y se apoyan en técnicas gráficas de especificación estructurada.

Enfoque	Tipo de sistema	Formalidad
Estructuradas	Gestión	No formal

<ul style="list-style-type: none"> – Orientadas a procesos – Orientadas a datos <ul style="list-style-type: none"> – Jerárquicos – No jerárquicos – Mixtas 		
Orientadas a objetos	Tiempo real	Formal

Tabla 3.3: Clasificación de las metodologías (Piattini, 1996)

Estos modelos gráficos, jerárquicos, descendentes y particionados son los diagramas de flujo de datos (DFD), los diccionarios de datos (DD) donde se definen los datos y los detalles de especificaciones de los procesos. En la bibliografía, se pueden consultar los detalles de las metodologías de DeMarco (1979) y Gane y Sarsons (1977), las cuales se fueron refinando a través del tiempo y se expandieron a las fases de diseño e implementación. En la Tabla 3.4 se presenta las diferencias entre las tres metodologías citadas:

Fases del análisis estructurado		
Método de Gane y Sarsons	Método de DeMarco	Método de Jourdon
<ul style="list-style-type: none"> – Construir un modelo lógico actual – Construir un modelo lógico del nuevo sistema – Seleccionar un modelo lógico – Crear un nuevo modelo físico del sistema – Empaquetar la especificación 	<ul style="list-style-type: none"> – Construir un modelo físico actual – Construir un modelo lógico actual – Crear un conjunto de modelos físicos alternativos – Examinar los costos y tiempos de cada opción – Empaquetar la especificación 	<ul style="list-style-type: none"> – Realizar los diagramas de flujo del sistema – Realizar el diagrama de estructuras evaluar el diseño, midiendo la calidad cohesión y el acoplamiento. – Preparar el diseño para la implantación

Tabla 3.4: Diferencias entre las metodologías de Gane y Sarsons, DeMarco y Yourdon (Piattini, 1996)

La metodología de orientación a objetos, cambia el modo de ver al sistema, como un modelado de objetos que interactúan entre sí y no desde el punto de vista de la funcionalidad y la descomposición en tareas y módulos, pasando de las funciones de los programas y datos almacenados a un enfoque integrador y unificado.

Las metodologías orientadas al objeto, se pueden clasificar en: puras, que cambian radicalmente la perspectiva estructurada como sostiene Booch (1991) y evolutivas como la de Rumbaugh (1991) y Martin y Odell (1997), que toman al diseño estructurado como base para el desarrollo orientado al objeto.

El proceso y la notación de diseño de Booch (1991) y los escenarios de Jacobson (J. Juzgado, 1996), están siendo utilizados por otras metodologías más recientes y sistemáticas, conjuntamente con las métricas y los modelos de mejora del software como el CMM¹⁴ (Modelo de Calidad y Madurez) o SPICE de ISO (Konrad y Paulk, 1995).

3.5. El ciclo de vida y los procesos

Todo proyecto tiene asociado, por más pequeño que éste sea, pasos que se deben seguir tales como: planificación, estimación de recursos, seguimiento y control, y evaluación del proyecto. La selección de un modelo de ciclo de vida está asociada a un orden en la realización de las actividades a desarrollar.

La red de actividades, es la que permitirá establecer a partir de la matriz de precedencia el camino crítico, como la secuencia de tareas más larga de principio al fin.

El diagrama de Gantt, o los diagramas calendario permitirán establecer el estado del proyecto en un determinado momento a partir de su inicio, en cuanto a recursos se refiere.

Para estimar el tamaño del producto o del programa a desarrollar, definido como la cantidad de código fuente, especificaciones, casos de prueba, documentación del usuario y otros productos, que han de ser desarrollados, se debe recurrir a datos estadísticos propios o no. La estimación consiste en la predicción del personal, el esfuerzo y el costo asociado para llevar a cabo todas las actividades del mismo.

3.5.1. La planificación de la gestión proyecto

Se la puede describir en términos de, las actividades a realizar, los documentos de salida y de las técnicas a utilizar como se observa en la Tabla 3.5.

¹⁴ CMM: Capability Maturity Model

3.5.2. La identificación de la necesidad

La identificación de una necesidad, enunciada en términos concretos, es el punto de partida para la puesta en marcha de un proyecto y la evaluación de las posibles soluciones darán la viabilidad del mismo.

Planificación de la gestión del proyecto	
Actividades a realizar	Confeccionar el mapa de actividades para el modelo elegido del ciclo de vida, asignar de los recursos, definir el proyecto, planificar la gestión.
Documentos de salida	Plan de gestión, plan de retiro.
Técnicas a utilizar	CPM, PERT, diagrama de Gantt, estadísticas, simulación (Montecarlo), puntos funcionales, Modelos de estimación (COCOMO), Técnicas de descomposición para estimación.

Tabla 3.5: Planificación de la gestión del proyecto.

Por lo tanto, es deseable, confeccionar un informe de necesidades basados en los ítems de la Tabla 3.6:

Identificación de la necesidad	
Actividades a realizar	Identificar necesidades, formular posibles soluciones y estudiar su viabilidad.
Documentos de salida	Informe de necesidades. Alternativas de solución. Soluciones factibles.
Técnicas a usar	De adquisición de conocimientos, análisis costo-beneficio, modelización, diagramas de flujos de datos, prototipado.

Tabla 3.6: Identificación de la necesidad

3.5.3. El proceso de especificación de los requisitos

Consiste en establecer de un modo conciso, claro y preciso el conjunto de requisitos que deben ser satisfechos por el software a desarrollar. El objetivo es determinar en forma total y consistente los requisitos de software. El análisis se realiza sobre la salida resultante, la descomposición de los datos, el procesamiento de los mismos, las bases de datos y las interfaces de usuario. (J. Juzgado, 1996). (ver Tabla 3.7).

Se debe considerar que un requisito es una condición o característica que debe tener el programa para satisfacer un documento formal. Estos requisitos pueden ser funcionales, de rendimiento o de interfaces. Los primeros especifican la función que el programa debe realizar, los segundos especifican una característica numérica y los últimos determinan las características de las interfaces, usuario-software, software-hardware y software-software. (Juzgado, 1996).

Especificación de requisitos	
Actividades a realizar	Definir y desarrollar los requisitos del software y de las interfaces.
Documentos de salida	Especificación de los requisitos del software, requisitos de interfaces de usuario, de interfaces con otro software y con hardware. Requisitos de interfaces con el medio.
Técnicas a usar	Técnicas orientadas a los procesos: Análisis estructurado: diagramas de flujo de datos (DFD), diccionario de datos (DD), especificación de procesos. Diagramas de actividades. Técnicas orientadas a los datos: Diagramas entidad relación y diagramas de datos. Técnicas orientadas a los objetos. Diagramas de clases/objetos Jerarquía de clases/objetos Técnicas formales de especificación: Técnicas relacionales: ecuaciones implícitas, relaciones recurrentes, axiomas algebraicos. Técnicas orientadas al estado: tablas de decisión, de eventos, de transición, mecanismos de estado finitos, etc. Técnicas de prototipación

Tabla 3.7: La especificación de los requisitos

3.5.4. El proceso de diseño

El proceso de diseño es la piedra angular para la obtención de un producto coherente que satisfaga los requisitos de software. El diseño desde el punto de vista técnico comprende cuatro tipos de actividades: diseño de datos, arquitectónico, procedimental y diseño de interfaces y desde el punto de vista del proyecto evoluciona desde un diseño preliminar al diseño detallado.

El diseño de datos, modela las estructuras de datos necesarias para el desarrollo, el arquitectónico define las relaciones entre las estructuras del programa, considerando el desarrollo de módulos que se relacionan, mezcla la estructura de programas y de datos, y define las interfaces. El diseño procedimental transforma estructuras en descripción procedimental del software y por último el diseño de interface establece los mecanismos de interacción humano-computadora.

Desde el punto de vista del proyecto, el diseño preliminar se centra en las funciones y estructuras de los componentes que forman el sistema y el detallado se ocupa de refinar el anterior en algoritmos, para cada módulo.

Una actividad importante a realizar es el diseño conceptual, lógico y físico de la base de datos, si la hubiera. Este proceso de diseño, es la correcta traducción de los requisitos de software en un producto. (ver Tabla 3.8).

Se deben aplicar algunos principios conducentes a un software de calidad, tales como:

- Abstracción.
- Refinamiento sucesivo.
- Modularidad (consiste en la división en forma lógica de elementos en funciones y subfunciones).
- Estructura jerárquica en módulos con control entre componentes.
- Estructura de los datos.
- Procedimientos por capas funcionales.
- Ocultamiento de la información, etc., aplicación de métodos sistemáticos y una revisión constante.

Para evaluar la calidad de un diseño se deben tener en cuenta criterios tales como:

- División en módulos con funciones independientes.
- Organización jerárquica de los módulos.
- Representaciones de datos y procedimientos distintas.
- Minimización de la complejidad de las conexiones entre las interfaces.
- Reproducibilidad del método de diseño con los datos de los requisitos.

Los diseños modulares, reducen la problemática de los cambios, permitiendo desarrollos en paralelo. Para la definición de los módulos se usan conceptos tales como la abstracción y el ocultamiento de la información derivados de la independencia funcional de los mismos.

Los módulos tienen una función específica y definida o sea cohesión máxima y mínima interacción con los otros módulos o acoplamiento mínimo. La cohesión es una medida de la fortaleza funcional del módulo y la el acoplamiento es una medida de interdependencia de los módulos de un programa.

Existen herramientas de tipo CASE (Computer Aided Software Engineerig) que permiten automatizar el proceso traducción a código.

3.5.5. El proceso de implementación

Proceso de diseño	
Actividades a realizar	Realizar el diseño arquitectónico, analizar el flujo de información, diseñar la base de datos, diseñar las interfaces, desarrollar los algoritmos, realizar el diseño detallado.
Documentos de salida	Descripción del diseño del software de la arquitectura del software, del flujo de información, descripción de la base de datos, de las interfaces, de los algoritmos.
Técnicas a usar	<p>Técnicas orientadas a los procesos: diseño estructurado, diálogo de las interfaces, diseño lógico, HIPO (Hierarchy Input Process Output).</p> <p>Técnicas orientadas a los datos. Modelo lógico y físico de datos. Jackson, etc.</p> <p>Técnicas orientada a los objetos: Modelo clase/objeto, diagrama de módulos.</p> <p>Técnicas de bajo nivel:</p> <ul style="list-style-type: none"> Programación estructurada: diagramas de árbol Programación orientada a objetos: diagrama de procesos Técnicas de prototipación, Técnicas de refinamiento, Jackson, etc.

Tabla 3.8: El proceso de diseño

Proceso de implementación	
Actividades a realizar	Crear los datos de prueba, crear código fuente, generar el código fuente, crear la documentación, planificar y realizar la integración de módulos.
Documentos de salida	Datos de prueba, documentación del sistema y del usuario. Plan de integración.
Técnicas a usar	Lenguajes de programación. Jackson

Tabla 3.9: El proceso de implementación.

Este proceso (ver Tabla 3.9), produce código fuente, código de la base de datos y documentación, de base de acuerdo a los estándares utilizados. La salida de este proceso conduce a las pruebas de validación y verificación.

3.5.6. El proceso de instalación

Este proceso se centra en la verificación de la implementación adecuada del software y en la conformidad del cliente, previa prueba de aceptación (Tabla 3.10).

Proceso de instalación	
Actividades a realizar	Planificar la instalación, instalar el software, cargar la base de datos, realizar las prueba de aceptación.
Documentos de salida	Plan de instalación del software e informe de instalación

Tabla 3.10: El proceso de instalación

3.5.7. Los procesos de mantenimiento y retiro

El proceso de mantenimiento se centra en el cambio asociado a los errores detectados, fallas, mejoras solicitadas y cambios. Se lo considera como una vuelta a la aplicación del ciclo de vida pero con un software existente como iteraciones de desarrollo.

Los tipos de mantenimiento pueden ser: correctivos, ante defectos encontrados, adaptativos, o sea, cambios del software de acuerdo al cambio en el entorno y de mejoras, con agregado de funciones adicionales.

Procesos de:	Mantenimiento	Retiro
Actividades a realizar	Reaplicar el ciclo de vida.	Notificar al usuario, realizar las operaciones en paralelo y retirar el sistema.
Documentos de salida	Orden de mantenimiento Y recomendaciones de mantenimiento.	Plan de retiro

Tabla 3.11: Los procesos de mantenimiento y retiro.

El retiro es la baja de un sistema existente. Muchas veces, se lo reemplaza por una nueva versión, y otras por un nuevo sistema, siendo otras veces reemplazado por un nuevo sistema que opera temporalmente en paralelo durante un cierto tiempo de preparación del nuevo sistema. (ver Tabla 3.11).

3.5.8. El proceso de verificación y validación

Las tareas que abarca son las siguientes: pruebas de verificación, revisiones y auditoría e incluye las tareas de validación y pruebas de validación que se realizan durante el ciclo de vida del software para asegurar la satisfacción con los requisitos.

Para la verificación y validación del software, cuando ya exista código ejecutable, se pueden realizar las pruebas del mismo que consisten en ejecutar el software con determinados datos de entrada y producir resultados que luego serán comparados con los teóricos.

Un proceso asociado a las pruebas, es la depuración que consiste en tratar de deducir dónde están localizados los defectos en el software que hacen que este no funcione correctamente. (ver Tabla 3.12).

Procesos de verificación y de validación	
Actividades a realizar	Planificar y ejecutar las tareas de verificación y validación. Recoger y analizar los datos de las métricas, planificar las pruebas, desarrollar las especificaciones de las pruebas y ejecutarlas.
Documentos de salida	Plan de verificación y validación. Informes de evaluación. Plan de pruebas. Especificación de las pruebas. Resultados de las pruebas.

Técnicas a usar	Técnicas de prueba de caja blanca: Técnicas de prueba de caja negra: Revisiones formales. Auditorías.
-----------------	--

Tabla 3.12: Los procesos de verificación y validación.

3.5.9. El proceso de la gestión de la configuración

El proceso llamado gestión de la configuración, involucra la gestión de los cambios durante el ciclo de vida que a partir de la configuración del sistema en un dado momento, tiene como objetivo un control de los cambios producidos y la coherencia del mismo.

Proceso de gestión de la configuración	
Actividades a realizar	Planificar la gestión de la configuración, identificar la configuración, realizar el control de la configuración y la información de estado de la misma.
Documentos de salida	Plan de gestión de la configuración, orden de cambio, cambio de estado, informe de estado.

Tabla 3.13: El proceso de gestión de la configuración.

Para ello es necesario la documentación del sistema en un momento determinado, el establecimiento de una configuración inicial y control de los cambios. (ver Tabla 3.13).

3.5.10. Los procesos de desarrollo de la documentación y de formación

Este proceso permite planificar, diseñar, implementar, editar, producir, distribuir y mantener los documentos para los desarrolladores y los usuarios.

Para una utilización efectiva del sistema se debe proporcionar al usuario las instrucciones y guías necesarias acerca del uso del software y de sus limitaciones. Es un punto fundamental la formación del usuario en el sistema. También es importante la formación de los desarrolladores y soporte técnico. (ver Tabla 3.14).

Proceso de desarrollo de:	La documentación	Formación
Actividades a realizar	Planificar e implementar la documentación, producir y distribuir la documentación.	Planificar el programa de formación. Desarrollar materiales de formación. Validar e implementar el programa.
Documentos de salida	Plan de documentación.	Plan de formación.

Tabla 3.14: Los procesos de documentación y de formación.

3.5.11. La selección de un ciclo de vida

La elección de un ciclo de vida adecuado para cada desarrollo está relacionada con las características del producto a obtener, a partir de los requisitos del desarrollo especificados.

De acuerdo al tipo de desarrollo, es conveniente realizar una adaptación del proceso descrito anteriormente en forma general y realizar, la matriz de actividades, partiendo del mapa de actividades-etapa del ciclo de vida elegido, la que se confecciona como una tabla de doble entrada, colocándose una cruz en la actividad a realizar en cada etapa. El mapa completo se denomina entonces, matriz de actividades para un determinado ciclo de vida elegido.

A partir de esta matriz se puede pasar a una estimación del tiempo y costo de cada actividad y para el proyecto global. También se pueden estimar los recursos necesarios por actividad. Algunas actividades se realizan por única vez y otras se repiten en cada etapa

3.6. El concepto de la calidad

Habría que comenzar con una revisión de algunas definiciones acerca de lo que significa calidad.

Deming (1982) propuso la idea de la calidad como conformidad con requisitos y confiabilidad en el funcionamiento. Juran (1995) dice brevemente: *"Quality is fitness for use"*, o sea es la adecuación del producto al uso, suponiendo un producto libre de deficiencias, cuyas características permiten la satisfacción del usuario. Crosby (1979) pone énfasis en la prevención y dice *"con defectos cero"*. La norma ISO 8402 define la calidad como:

"Totalidad de características de un producto o servicio que le confieren su aptitud para satisfacer unas necesidades expresadas o implícitas".

Estas necesidades especificadas, bien pueden estar en un contrato o se deben definir explícitamente.

El logro de la calidad puede tener tres orígenes: *calidad realizada*, *calidad programada* y *calidad necesaria*. La primera es la que es capaz de obtener la persona que realiza el trabajo, la segunda es la que ha pretendido

obtener y la tercera la que exige el cliente y que le gustaría recibir. La gestión de la calidad pretenderá que estas coincidan.

3.6.1. La calidad en ingeniería de software

El software es un producto con características muy especiales, hay que tener en cuenta que es un producto que se desarrolla y se centra su diseño, con una existencia lógica de instrucciones sobre un soporte, siendo un producto que no se gasta con el uso como otros y repararlo no significa restaurarlo al estado original, sino corregir algún defecto de origen lo que significa que el producto entregado posee defectos, que podrán ser solucionados en la etapa de mantenimiento (Piattini, 1996).

El diccionario IEEE estándar de ingeniería del software (IEEE, 1990) dice que son software: *“los programas de ordenador, los procedimientos y, posiblemente la documentación asociada y los datos relativos a la operación del sistema informática”*, no limitándose al código.

El estándar IEEE 6.10 -1990 (IEEE, 1990) da la definición de calidad como *“el grado con el que un sistema, componente o proceso cumple con los requisitos especificados y las necesidades o expectativas del cliente o usuario”*.

Pressman (1993) la define como *“concordancia del software con los requisitos explícitamente establecidos, con los estándares de desarrollo expresamente fijados y con los requisitos implícitos, no establecidos formalmente que desea el usuario”*.

La aplicación de estándares de desarrollo y de normas para el software permitirá lograr calidad técnica del mismo. La calidad del software se puede ver a nivel empresa como implantación de un sistema de calidad y a nivel de proyecto aplicando las técnicas de evaluación y control de la calidad del software a lo largo del ciclo de vida.

El interrogante que surge es: *¿Por dónde empezar el proyecto de mejora de la calidad de los programas en general?*. Como respuesta a ello, se analizarán brevemente en la sección siguiente los aportes de algunas de las instituciones están trabajando al respecto y que elaboraron algunos estándares como: IEEE, ISO y SEI.¹⁵

Los requisitos explícitos, ya sean funcionales, de seguridad, de rendimiento, de interface, son la culminación de la etapa de análisis y quedan establecidos en el documento de especificación de requisitos del software y es en la etapa de análisis donde muchos de los requisitos implícitos no expresados formalmente por el usuario quedarán declarados en el documento de especificación.

3.6.2. La calidad desde el aspecto organizacional. La familia ISO 9000

La familia ISO 9000, es un conjunto de normas en las que se apoya el sistema de calidad de una empresa.

La norma ISO 9000 define al sistema de calidad desde la perspectiva de la organización como: *“La estructura de organización, de responsabilidades, de actividades, de recursos y de procedimientos que se establecen para llevar a cabo la gestión de calidad”* (ISO-9001: 1994). En la Tabla 3.15 se define la terminología básica empleada.

Desde esta posición, se debe fijar la estructura organizativa ligada al sistema de gestión de la calidad a través de líneas jerárquicas y de comunicación.

Entre las normas con la que las organizaciones cuentan para el logro del aseguramiento de la calidad de los productos software, están las de la familia ISO 9000. (ISO 9000, 1991). Si bien esta familia en un principio se la diseñó para aplicaciones industriales en general, existe una versión la 9000-3, específica para productos lógicos. (ISO 9000-3, 1991).

Bajo la denominación de ISO SPICE (Konrad, Paulk y Graydon, 1995) se desarrollaron para ISO e IEC¹⁶ un conjunto de estándares en un intento de armonizar los diferentes esfuerzos en el mundo para conducir a la obtención de un proceso de software confiable.

Término empleado	Definición
Gestión de la calidad	Es un aspecto de la función general de la gestión que determina y aplica la política de la calidad (objetivos y directrices generales de calidad) y normalmente se aplica a nivel empresa. (Aenor).
Aseguramiento de la Calidad	“Es el conjunto de actividades planificadas y sistemáticas necesarias para aportar la confianza de que el producto (software) satisfará los requisitos dados de calidad” (Aenor, 1992). El estándar IEEE (1990) dice que se puede referir al “conjunto de actividades para evaluar el proceso mediante el cual se desarrolla el producto”
Control de la	Son las técnicas y actividades de tipo operativo destinadas a controlar un proceso y elimi-

¹⁵ IEEE. Institute of Electrician and Electronics Engineering.

ISO: International Organization for standarization.

SEI: Software Engineering Institute.

¹⁶ IEC: International Electrotechnical Comission.

calidad del software	nar las causas de defectos durante las etapas del ciclo de vida. (Aenor, 1992).
Verificación y Validación	La IEEE (1990) dice que es una actividad ligada al control de la calidad en el ámbito del software. La verificación trata de comprobar si el producto construido en una etapa del ciclo de vida satisface los requisitos establecidos en la etapa anterior, respondiendo a la pregunta: ¿está el producto construido correctamente? La validación : consiste en comprobar si el software construido satisface los requisitos del usuario, respondiendo a la pregunta. ¿El producto es el correcto?

Tabla 3.15: Terminología empleada.

Uno de los productos del proyecto es una guía para buenas prácticas de dirección e ingeniería de software, similar a CMM¹⁷ del SEI y al Trillium de Northern Telecom.

La idea central es crear un modo para medir la capacidad mientras se permite una aproximación a la mejora como los niveles de madurez del CMM, siendo estas aproximaciones para medir la implementación e institucionalización de procesos específicos.

El estándar IEEE 1074-1991 define el conjunto de actividades requeridas para llevar a cabo el desarrollo y el mantenimiento de un producto software, cuya calidad sea confiable. La definición de los procesos y de las actividades para cada una de las etapas del ciclo de vida elegido permiten obtener un producto que se ajuste a los requerimientos. Desde esta perspectiva, cada actividad se la define a partir de una descripción de la misma y de las respectivas entradas y salidas.

Por otra parte, cuantificar la calidad significa tener que medirla. Para cuantificarla se requiere de mediciones indirectas de algunas manifestaciones de la misma, que se denominan métricas.

El estándar IEEE 1061-1992, establece el propósito de las métricas para calidad del software, provee un sistema de métricas y provee de una metodología para el establecimiento de los requerimientos de calidad. No prescribe métricas específicas, pero si ejemplos de aplicación del mismo.

Existen algunos modelos como el CMM (Capability Maturity Model) desarrollado por Victor Basili en el SEI (Software Engineering Institute) de la Universidad Carnegie Mellon (CMU), pensado para el Departamento de Defensa (DoD) de los Estados Unidos, el cual se extendió rápidamente al ámbito empresarial.

Este modelo considera el *grado de madurez del producto*, teniendo en cuenta cinco etapas bien diferenciadas. Estas etapas van desde un proceso inmaduro e improvisado, con falta de rigurosidad metodológica, hasta llegar a un proceso totalmente maduro, que evidencia rigurosidad y consistencia en la administración de los recursos y adecuación de los requerimientos con el proceso.

Quizás la clave consiste en la mejora evolutiva, a partir de establecimiento y claridad de los puntos de partida y de arribo en cada una de las etapas.

La evolución a través de los cinco niveles: inicial, repetible, definido, administrado y optimizado, permiten establecer los controles requeridos para gestionar los proyectos: planificar, administrar, controlar, supervisar (nivel 2), para permitir (en el nivel 3) integrar los aspectos organizacionales con el proyecto en si. Recién, en el nivel 4, se establece una administración cuantitativa del proceso y de la calidad del producto software y el último, define los aspectos a tener en cuenta para la prevención de los defectos, implementación de mejoras y cambios.

Es un modelo aplicable a organizaciones que desarrollan proyectos, y que permite luego comparar resultados, pero su aplicación no garantiza el éxito del proyecto, tornando a la empresa en una estructura más rígida.

Este modelo de mejora gradual del proceso de software y calidad del producto, no es el único y su aplicabilidad depende de las necesidades y características de la organización.

Si bien a través de estos estándares (ISO 9000-3 y CMM) se intenta obtener una mejora continua, se centran exclusivamente en los procesos y no en los procesos de pruebas, quedando fuera de su alcance el plan de pruebas y la selección de las mismas, aunque en la ISO se menciona la documentación de las pruebas y no el proceso de prueba específicamente, habría que ver también el estándar 820 de la IEEE (IEEE, 1991).

Bender (1996) sugiere agregar al CMM, una “*área clave de proceso*” (KPA, Key Process Area), para la prueba y evaluación del software. Burnstein, et al. (1996) desarrollan un modelo que denominan TMM (Testing Maturity Model) a modo de complemento al CMM, como un indicador de la madurez del proceso de prueba, a fin de implementar las mejoras pertinentes. Se basa en niveles de madurez y metas para cada uno, con cuestionarios de valoración de cumplimiento en cada nivel, como también entrenamiento del personal de prueba y valoración mediante de cuestionarios y entrevistas.

3.6.3. El concepto de calidad del software

¹⁷ CMM: Capability Maturity Model.

Boehm (1978) y McCall (1977) descomponen el concepto de calidad en propiedades más sencillas de medir y de evaluar. El modelo de McCall se basa en la descomposición del concepto de calidad en tres usos importantes de un producto de software desde el punto de vista del usuario:

- Características de operación.
- Capacidad para soportar cambios (ser modificado).
- Adaptabilidad a nuevos entornos.

Cada capacidad se descompone en una serie de factores a saber: facilidad de uso, integridad, fiabilidad, corrección, flexibilidad, facilidad de prueba, facilidad de mantenimiento, transportabilidad, reusabilidad y interoperabilidad.

Cada factor se descompone en criterios o propiedades internas del software que determinan su calidad: facilidad de operación, facilidad de comunicación, facilidad de formación o aprendizaje, control de accesos, facilidad de auditoría, eficiencia de ejecución, eficiencia de almacenamiento, exactitud o precisión, consistencia, tolerancia a fallas, modularidad, simplicidad, completitud, facilidad de traza, autodescripción, capacidad de expansión, generalidad, instrumentación independencia entre sistema y software, independencia del hardware, compatibilidad de comunicaciones y compatibilidad de datos.

McCall define el factor de calidad como FC, según:

$$FC = c_1 \times m_1 + c_2 \times m_2 + \dots + c_n \times m_n$$

Donde los c_i son los coeficientes de regresión y los m_i son las métricas que afectan al factor de la calidad.

Estos criterios pueden ser evaluados mediante un conjunto de métricas, las que se pueden calcular observando directamente el software. Para cada criterio McCall propuso una serie de métricas, aunque, muchas de ellas sólo pueden ser medidas en forma subjetiva. Las métricas pueden estar en forma de listas de comprobaciones, para obtener el grado de los atributos específicos del software. McCall propuso un esquema de graduación mediante una escala que va de cero (bajo) a 10 (alto) y utiliza como métricas los criterios o propiedades internas del software citados anteriormente.

La norma IEEE 1061 propone un modelo de medición muy parecido al de McCall y la norma ISO 9126 (ISO, 1991) establece un modelo propio, similar al de McCall.

En la década del ochenta, se comenzó a usar modelos particulares de evaluación para cada empresa o proyecto, implantándose el concepto de calidad relativa. Gilb (1988) propone la creación de una especificación de requisitos de calidad a redactar conjuntamente el usuario y los analistas, determinando así la lista de características que definan la calidad de cada aplicación. Este enfoque se ha asociado a la filosofía QFD (Quality Function Deployment), o el despliegue de la función de la calidad que se aplica al ámbito de la gestión de la calidad industrial y en el que se han basado modelos posteriores. Otros modelos son los de Basili y Rombach (1988) proponen el paradigma GQM, (objetivo–pregunta–métrica o goal-question-metric) para evaluar la calidad de cada proyecto.

Grady y Caswell (1987) presentan un enfoque de medición inspirado en el control estadístico de procesos aplicado a la industria convencional de fabricación, considerando a la calidad como la ausencia de defectos, que en este caso pueden ser fallas, defectos o errores.

3.6.4. Métricas de calidad del software

Para la evaluación de la calidad es más habitual referirse a medidas del producto que en medidas del proceso. Una métrica (Fenton, 1997) es “una asignación de un valor a un atributo de una entidad de software, ya sea un producto o un proceso”. En todos los casos las métricas representan medidas indirectas de la calidad, ya que sólo se miden las manifestaciones de ella.

Se pueden tener métricas basadas en el texto del código y métricas basadas en la estructura de control del código.

Métricas basadas en el texto del código: En general, se pueden tomar la cantidad de líneas de código, como un indicador de tamaño, el número de líneas de comentarios como un indicador de la documentación interna, el número de instrucciones, el porcentaje de líneas de código o densidad de documentación, etc. Halstead (1975), propone sus métricas dentro de este tipo, denominadas: “*Ciencia del software*”.

Métricas basadas en la estructura de control del código: Pueden tomarse dos tipos de medidas: unas relacionadas con el control intramodular, basada en el grafo de control y otras relacionadas con la arquitectura en módulos, basada en el grafo de llamadas o en el diagrama de estructuras. Las métricas de McCabe (McCabe, 1976) son del primer tipo y constituyen un indicador del número de caminos independientes linealmente basándose en conceptos matemáticos que existen en un grafo.

Yin y Winchester (1978) definen el *fan-in* y el *fan-out* de cada módulo. Henry y Kafura (1984) definen la complejidad del módulo. Piattini (1996) sostiene que los resultados parecen indicar que mejores valores de métricas implican un menor mantenimiento posterior debido a un menor número de defectos.

3.6.5. Las diferentes aproximaciones

Siguiendo a Fenton (1997), la medición es un proceso por el cual, se debe asignar números o símbolos a atributos y entidades en el mundo real, de tal modo de describirlas de acuerdo a reglas definidas claramente. Recordando a DeMarco (1982) *“no se puede controlar lo que no se puede medir”*, esto daría una idea acerca de cuál es la necesidad primordial de efectuar mediciones sobre un proceso, en un proyecto. Cada acción de medición de algo, debe estar motivada por un objetivo particular o necesidad definida claramente. Si este razonamiento, se suma al principio de Gilb (1988), citado por Fenton: *“los proyectos que no tienen objetivos claros, no arriban a metas claras”*, queda explícita la necesidad de una metodología y la adecuación de lo que se deba medir al tipo de software a desarrollar y su uso particular.

¿Para qué medir? Bien, para ayudarnos a entender qué está sucediendo durante un desarrollo, analizando los desvíos respecto de las líneas de base, o para controlar lo que está sucediendo en el proyecto o programa respecto de una línea de base, o, por último, para mejorar los procesos.

Si bien en los proyectos software empresarial, se miden esfuerzo, costo, productividad, (COCOMO y COCOMO II),¹⁸ capacidad y madurez, calidad (Mc Call) confiabilidad, entre otras métricas, no todos los modelos desarrollados para este tipo de software, serían adecuados para los proyectos de programas educativos.

Ya se describió sintéticamente el modelo de tres niveles de Mc Call (1977), llamado comúnmente FCM (Factor Criteria Metric). Cada factor de calidad está compuesto por un criterio, que es lo que realmente se mide, ya que son más fáciles de entender. En un tercer nivel se describe el grado de pertinencia de las relaciones entre factores, aclarando que hay que *“dividir para conquistar”*.

De acuerdo a Boehm (1978) y a Mc Call (1977) ambos definen atributos externos: confiabilidad, usabilidad (utilidad) y mantenibilidad y eficiencia y testeabilidad como internos. Dentro de los internos, también estarían la estructurabilidad y la modularidad y estos se reflejan sobre los externos. A cada uno se le asignan criterios y métricas, posteriormente.

La duda que surge es: *¿Utilizar un modelo predefinido o definir un modelo propio?*. Ya que los anteriores son típicos para modelos fijos, esta podría ser una salida al problema, como sostiene Gilb (1977) o el COQUAMO (Constructive Quality Model) de Kitchenham y Walter (1989).

La ISO 9126 (1991) define calidad en términos de atributos de interés para el usuario del producto de software, algunos internos y otros externos al mismo, define seis factores y atributos. Se lo puede considerar la piedra angular en la definición de un proceso para evaluación de la calidad del software.

Otros investigadores, miden portabilidad, integridad, densidad de defectos, no habiendo consenso de qué es el defecto que se mide, por lo cual habría que definirlo en cada caso.

Llegando al concepto de *“utilidad (usability) del software”* como la extensión para la cual, el producto es conveniente y práctico, de un modo intuitivo se la podría considerar como amigabilidad teniendo en cuenta su practicidad o valor práctico.¹⁹

Fenton dice que se podría definir en términos de *“la probabilidad de que un operador de un sistema no experimente un problema en la interface de usuario durante un período dado de operación en condiciones normales de operación o cómo el usuario interactúa con la interface”*.

Por ahí, sería conveniente remitirse a características internas más simples que conduzcan a una buena utilidad, como:

- Buen uso de menú y gráficos.
- Mensajes de error e informativos.
- Funciones de ayuda.
- Manuales bien estructurados.

Este podría ser el *concepto clave*, buscado, especialmente para los programas educativos y en él se hará hincapié.

3.6.6. La verificación y la validación del software

Otros aspectos a tener en cuenta, son las revisiones y las pruebas del software como parte del ciclo de vida, que se utilizan para detectar fallas en los requisitos, en el diseño y en la implantación y son procesos orientados a la detección de defectos en el producto, dándole mucha importancia a las revisiones.

La verificación y la validación del software (VyV) incluyen un conjunto de procedimientos, actividades, técnicas y herramientas que se utilizan paralelamente al desarrollo del software, para asegurar que el producto resuelve correctamente el problema para el que fuera diseñado. El objetivo es prevenir las fallas desde los requerimientos hasta su implementación.

La VyV actúa sobre los productos intermedios intentando detectar y corregir cuanto antes sus defectos y desviaciones del objetivo primigenio, si las hubiera.

¹⁸ COCOMO: COst COnstructive MOdel

¹⁹ Simon & Schuster's. International Dictionary. McMillan, USA.

Respecto de las pruebas a realizar en el software, ellas pueden ser dinámicas o estáticas, de acuerdo a si se realizan o no sobre el código. Entre las pruebas dinámicas, están las llamadas de caja blanca y las de caja negra, "testeando" los procedimientos estructurales o las salidas. Y entre las estáticas están las revisiones, las verificaciones, a fin de detectar problemas durante el proceso de desarrollo.

3.6.7. Las revisiones del software

El estándar IEEE (1989), indica que para conseguir los objetivos de aseguramiento de la calidad se disponen de los siguientes métodos como se puede ver en la Tabla 3.18.

Las revisiones y auditorías se pueden utilizar para revisar procedimientos e gestión y productos. Un tipo de revisiones a considerar son las *revisiones técnicas*, cuyo objetivo es evaluar un producto intermedio de desarrollo para comprobar que el producto se ajuste a las especificaciones y que se está llevando a cabo de acuerdo a los planes y estándares.

Objetivos de calidad	Método principal
Evaluación	Revisión de gestión, revisión técnica
Verificación	Inspección, walkthrough
Validación	Pruebas
Confirmación de cumplimiento	Auditoría

Tabla 3.18: Métodos principales para conseguir objetivos del aseguramiento de la calidad.

Las *inspecciones* tienen por objetivo detectar y registrar los defectos del producto intermedio, en forma rigurosa y formal, buscando que el producto satisfaga las especificaciones y estándares. Están pensadas como una medida de ayuda al gestor.

Los *walkthroughs* se realizan para evaluar un producto en busca de defectos u omisiones, como una medida de ayuda al desarrollador, considerando las posibles alternativas de solución cuyo objetivo es comprender bien el objeto.

Los métodos citados anteriormente constituyen el análisis estático ya que no se necesita ejecutar el software. Una mención especial dentro de los análisis dinámicos del software corresponde a las pruebas modulares y de integración, según se detalla en el estándar IEEE-1012 (1986) y las pruebas pueden ser funcionales o estructurales. En la figura 3.1 se puede ver, la relación entre de algunos procesos de aseguramiento de la calidad sobre productos y proyectos:

Las auditorías se realizan para determinar en forma objetiva que los productos desarrollados se ajustan a los estándares. En el estándar IEEE (1984), se recomienda realizar una cantidad determinada de revisiones y auditorías sobre todo en software críticos.

4. La evaluación de software educativo.

4.1. Resumen

En esta sección se definen los tipos de evaluación a las que se deben someter los programas educativos (sección 4.2). Se describen las evaluaciones interna (sección 4.3) y externa (sección 4.4), y se analizan los instrumentos de evaluación utilizados (sección 4.5), presentándose una síntesis de los criterios de selección y de evaluación de los programas didácticos, a modo diacrónico durante las últimas décadas y sincrónico en el fin de siglo, considerando las pautas fijadas en el ámbito didáctico para lograr una efectividad del producto (sección 4.6).

Para este relevamiento y análisis documental, se han tenido en cuenta las líneas didácticas de investigadores de diferentes escuelas como la norteamericana y la española, y también algunos otros trabajos relevantes en evaluación de programas educativos, aunque consideren casi exclusivamente el aspecto técnico y muy poco el pedagógico.

4.2. La evaluación

La evaluación de los programas educativos es un proceso que consiste en la determinación del grado de adecuación de dichos programas al contexto educativo. Cuando el programa llega al docente, es de suponer que ha sido analizado y evaluado tanto en sus aspectos pedagógicos y didácticos, como en los técnicos que hacen a la calidad del producto desarrollado según ciertas pautas de garantía de calidad.

Básicamente, se realizan las evaluaciones interna y externa del software, a fin de detectar los problemas que generarán cambios en el producto, lo antes posible, a fin de reducir costos y esfuerzos posteriores. Estas evaluaciones consideran las eventuales modificaciones sugeridas por el equipo de desarrollo y por los usuarios finales, teniéndose en cuenta a docentes y alumnos en el contexto de aprendizaje.

Cuando un producto del tipo comercial educativo, llega al docente, significa que ha superado las etapas de evaluaciones interna y externa y además se espera obtener el grado de eficacia y de eficiencia del producto para lo que se deberá realizar una evaluación en el contexto de uso particular.

Es preciso definir ciertos “*criterios*” para seleccionar un programa que esté “*de acuerdo a las necesidades del docente*”. Además se debe considerar el uso de los vocablos “*evaluación*” y “*valoración*”²⁰, que en muchos de los trabajos consultados se usan indistintamente para determinar si un programa dado cumple con los objetivos tanto técnicos como pedagógicos y didácticos para lo que fue pensado.

4.3. La evaluación interna

De acuerdo a lo expuesto anteriormente, se deberá llevar a cabo una evaluación interna del software, (Marquès, 1995) que estará a cargo de los miembros del equipo de desarrollo y otra evaluación, la externa en la que participan profesores y alumnos destinatarios del programa, cuando se haya terminado el mismo, o esté casi listo.

La versión preliminar a evaluar poseerá todas las funcionalidades del programa, pero algunos aspectos como los mensajes, las imágenes y los gráficos serán provisorios e incompletos y las bases de datos muchas veces estarán cargadas parcialmente.

Luego de la evaluación realizada por el equipo de desarrollo, se confeccionará una lista con los cambios necesarios y las modificaciones, las mejoras estructurales y todos aquellos aspectos que se crean convenientes antes de utilizar el programa o de comercializarlo, buscando adecuación a las especificaciones de requerimientos y un aseguramiento de los aspectos funcionales y pedagógico–didácticos.

Algunos autores como Marquès (1995) consideran que se pueden contemplar tres aspectos fundamentales en la evaluación en general: aspectos técnicos, pedagógicos y funcionales.

Los primeros permitirán asegurar la calidad del producto desde el punto de vista técnico específicamente, pudiéndose realizar un análisis estructural de elementos tales como el diseño de pantallas y la interface de comunicación.

Los aspectos pedagógicos, son aquellos que se refieren al fin con el que el software será utilizado. Por ello hay que analizar elementos como: los objetivos educativos, los contenidos y los caminos pedagógicos, que se deben considerar en toda buena programación didáctica. Respecto de los funcionales, habría que considerar cuáles son las ventajas que le da al profesor como material didáctico, cómo facilita los aprendizajes de los alumnos y cuáles de las funciones del pensamiento favorece.

Bork (1986), denomina a esta evaluación interna como formativa, o sea la evaluación del proceso, como aquella realizada generalmente por los desarrolladores.

Para realizar las evaluaciones generalmente se utilizan listas de control o *checklists*, mediante planillas o plantillas de *checklists* y casillas de verificación, incluyendo no sólo preguntas cerradas, sino preguntas abiertas sobre diversos aspectos del programa. Estos resultados son los que necesita el equipo desarrollador para hacer todos los cambios necesarios y convenientes.

Luego de producidos los cambios, es cuando se agregan los efectos faltantes (como sonido, animaciones, imágenes y gráficos) y se carga de la base de datos final (si la hubiere), para proceder a emitir una versión de prueba externa (Marquès, 1995).

La documentación, además de ser un proceso que se realiza paralelamente durante todo el desarrollo del programa, será también evaluada externamente, junto con el programa. Este punto es muy importante, ya que la mayor parte de los programas carecen de ambas documentaciones: interna y externa, y en otros casos esta es escasa como para conocer su estructura interna.

4.4. La evaluación externa

La evaluación externa permite obtener las sugerencias de los alumnos potenciales, quienes serán en definitiva los usuarios del software y de los docentes que lo utilizarán como material didáctico.

Durante este tipo de pruebas, se encuentran a menudo errores imprevistos no detectados y se verifica el cumplimiento de los programas con los objetivos educativos que se han considerado en el diseño.

Alfred Bork (1986) la denomina evaluación sumativa y es la evaluación del producto final que generalmente la realizan equipos distintos a los desarrolladores. La información se recoge mediante *checklists* y preguntas cerradas y abiertas a contestar luego de interactuar con el programa, durante un tiempo predeterminado.

²⁰ *Valorar*: en un sentido amplio, según Squires y Mc Dougall (1994), tiene en cuenta la evaluación, la revisión y la selección. Tanto la revisión (resumen de las características para permitir la selección) y la selección (es la valoración de los profesores antes del uso en el aula) se las puede considerar como parte de la valoración misma en un sentido más restringido.

Evaluar: es determinar el grado de adecuación, ya sea durante el desarrollo para hacer las modificaciones (formativa) o posterior a él mediante experiencias de uso (sumativa).

En casi todas las investigaciones consideradas se denota la falta de herramientas de evaluación sencillas y de documentación de los programas educativos.

Como resultado de ambas evaluaciones, se obtiene la llamada primera versión del programa con su respectivo manual de usuario, que contiene todos los aspectos que se consideren indispensables para el uso docente, con detalles técnicos, y del entorno pedagógico y didáctico para el que se desarrolló el programa.

A la etapa de adquisición de un programa le sobreviene, una etapa fundamental: la de mantenimiento y de actualización, en la cual la documentación técnica tanto externa como interna juega un papel importantísimo, ya que es la base para cambios posteriores, y que en la mayoría de los casos no se desarrolla o es casi inexistente.

Tanto la etapa de actualización como la de mantenimiento, a la hora de hablar de programas educativos, no se tienen en cuenta al adquirir el producto y éste puede quedar fuera de mercado, en muy poco tiempo, debido a los avances tecnológicos, sin posibilidad de agregarle nuevas funcionalidades.

4.5. Los instrumentos de evaluación

En general, los instrumentos más usados, son los cuestionarios de valoración, donde las respuestas a estos cuestionarios son valoradas entre 0 y 5, por ejemplo, siendo el resultado el grado de conformidad del usuario con las afirmaciones propuestas.

Los instrumentos de evaluación, en forma de planillas se deben confeccionar con inclusión no sólo de preguntas del tipo cerradas, sino también de preguntas abiertas, y casillas de verificación, permitiendo al usuario final la descripción de aspectos problemáticos y particulares del programa que no hayan sido tenidos en cuenta durante la confección del instrumento.

Se deberá tener en cuenta al redactar los cuestionarios, la utilización de un vocabulario adecuado, sin ambigüedades y claro para los destinatarios previstos en cada caso en particular.

En la mayor parte de los cuestionarios relevados se consideran algunos aspectos claves o sobresalientes: como el logro de los objetivos, los aspectos técnicos, el desarrollo de contenidos, actividades y la documentación. Estos aspectos se categorizan en ítems, según cada propuesta.

Como cada propuesta de evaluación está relacionada a un software particular o a paquetes de programas que se relacionan, se deben analizar con cuidado los diferentes criterios para evaluación de estos medios didácticos, debidos a las particularidades del software educativo, considerándolas sólo como una “guía” que luego se deberá “readaptar” a cada contexto educativo particular. Es esta adaptación a cada contexto en particular la que nos permite afirmar que no hay un instrumento de evaluación único, sino que el mismo será función del contexto de aplicación.

4.6. Las propuestas de selección y evaluación de software educativo

En las últimas décadas se han elaborado muchas propuestas con listas de criterios para seleccionar y evaluar el software educativo, algunas forma individual y otras en el ámbito institucional. Si bien varían en cuanto a su contenido y estilo, todas ellas tienen un objetivo común: ayudar al docente a elegir y valorar un programa adecuado para sus necesidades.

Cabero (1993) sostiene que las propuestas para la evaluación de los programas informáticos han sido muy variadas. Sancho (1994) después de revisar distintas escalas propone una como representativa de las dimensiones de análisis, que se describe a continuación.

Pretendía recoger información acerca de: contenidos (puntos 1 y 25), aspecto técnico del programa (3, 6, 17, y 22), motivación para el alumno (4), valoración didáctica general del programa (5 y 11), claridad del programa (7, 18, 23 y 2), duración del programa (9), facilidad de manejo (10 y 12), adecuación a los receptores (14, 15, 20 y 21) y objetivos (13, 16 y 24) en unas 25 preguntas. Cada ítem debía ser valorado de 1 a 5 de acuerdo con la siguiente escala (5) muy adecuado (4) bastante adecuado (3) adecuado (2) poco adecuado (1) nada adecuado, pudiéndose utilizar la contestación NA: no aplicable. Como se puede ver en los números dentro de los paréntesis, no había un agrupamiento a priori en categorías, sino que las preguntas relacionadas a un mismo ítem estaban intercaladas con otras.

Marquès (1995), investigador de la Universidad Autónoma de Barcelona propone una ficha para catalogación y evaluación de programas didácticos. Considera rasgos generales del programa, objetivos que se persiguen, tipología, contenidos que se tratan, valoración técnica, valoración pedagógica, aspectos negativos, etc., concientizado de que al evaluar un programa, hay que considerar sus características y su adecuación al contexto en el que se quiere utilizar.

En su ficha separa la catalogación, cuyo objetivo es proporcionar una idea de las prestaciones que ofrece, de la evaluación que recoge la información del profesor acerca de diferentes aspectos del programa.

Los aspectos técnicos a considerar son: las pantallas, el algoritmo principal, el entorno de comunicación y las bases de datos; los aspectos pedagógicos: los objetivos educativos, los contenidos, las actividades interactivas, la integración curricular, la documentación del programa, los aspectos funcionales: utilidad del programa en cuanto a motivación y facilitación de aprendizajes.

Considera a la evaluación contextual de un programa como a la forma en que ha sido utilizado en clase un determinado programa independientemente de su calidad técnica y pedagógica. Esta evaluación tiene en cuenta el grado de logro de los objetivos educativos respecto de los planificados. Insiste en que la metodología utilizada por el profesor constituye el principal elemento determinante del éxito de la intervención didáctica, por lo tanto debe tenerse en cuenta la motivación previa que ha realizado el profesor antes de la sesión, la distribución de los alumnos en clase, la autonomía para interactuar con el programa. Aquí juega un rol importante las características de los alumnos, el grado de motivación, los estilos cognitivos, los intereses, el conocimiento previo y las capacidades.

Osuna, Bermejo y Berroso (1997), del Departamento de Ciencias de la Educación de la Universidad de Extremadura, proponen una escala de evaluación para software educativo. Sostienen la necesidad de una evaluación sistemática debida a la creciente cantidad de productos informáticos generados por la industria informática. La misma debería facilitar la toma de decisiones de los profesores y administradores educativos para su adquisición y uso. La escala de evaluación que articulan contiene: identificación del programa, valoración de elementos y valoración de relaciones contexto-entrada-proceso. Valoran los elementos en muy adecuados, adecuados, poco adecuados y nada adecuados.

Cabero Almenara (1992), de la Universidad de Sevilla, sostiene que uno de los grandes problemas para el uso de la informática en el terreno educativo radica en la existencia y calidad del software. La calidad del software educativo es un concepto complejo y muy difícil de precisar, ya que es el resultado de una serie de interacciones: el contenido, el docente, la tecnología que condicionarán los resultados que de él se obtengan. Un problema recurrente de los medios de enseñanza, consiste en determinar de qué manera pueden diseñarse para que la comunicación de los mensajes sea más eficaz y la interacción con el usuario sea lo más útil posible. En síntesis, para que facilite el aprendizaje y el recuerdo de la información por ellos transmitida y propicie entorno de aprendizaje más variados.

Los principios de diseño variarán de acuerdo a la función que tengan y al papel que desempeñe en el proceso de enseñanza y aprendizaje, bien sea transmisión de la información, evaluación de los estudiantes, presentación de ejemplos, motivación, simulación de fenómenos, etc.

La evaluación del software educativo puede ser realizada desde diversas perspectivas y por diversas personas y especialistas; respecto de este aspecto Olivares et al. (1990) llaman la atención en que éstos pueden ser especialistas de comunicación informática, especialistas en comunicación audiovisual, evaluadores externos, profesores, alumnos. Cada grupo tendrá preocupaciones diferentes, que llevan a observar aspectos desiguales. En su investigación Cabero encuentra que tanto alumnos como docentes, sugieren que la forma de diseñar el software educativo cae dentro de la concepción de libro interactivo, forma que se destaca por el uso de organizadores previos, resúmenes, división de pantallas, buen uso de animación, aparición progresiva de la información, utilización de parpadeo, ejercicios prácticos para los alumnos. Enfatiza respecto de la cautela a tener en cuenta en los resultados ya que psicológicamente y actitudinalmente podrían estar influenciados por el factor novedad y la difícil ruptura del paradigma tradicional del libro.

Otras observaciones hacen referencia a la rigidez en cuanto que no deja pasar de la segunda a la tercera etapa, por ejemplo hasta que no se aciertan muchas respuestas bien, por lo que el programa se hace monótono.

Como conclusiones: el programa debería tener un ítem NS/NC (no sabe/no contesta), haría falta un ítem que contemple el uso del hipermedia, que puede ser la clave para la rigidez encontrada. También se encontró que la referencia explícita a valores, temas transversales, contenidos actitudinales, apenas se reflejan o se dan en forma poco explícita Bunes et al. (1993) y Bolívar (1995), pero aunque sujeto a modificaciones, lo puede considerar como válido como una primera aproximación.

Por último, se ha dejado a Bartolomé Pina (1998), de Departamento de Didáctica de la Universidad de Barcelona, quien considera que la larga lista de preguntas en los instrumentos de evaluación, tiene dos objeciones principales: una que proviene de la relevancia de los parámetros observables y la otra de la relatividad de estos parámetros.

Normalmente los parámetros relacionados con la adecuación para la realización de un aprendizaje concreto, capacidad de estímulo resultan difícilmente observables, y su medida suele adolecer de una gran subjetividad. Cabe señalar que hay algunos parámetros observables que pueden ser relevantes, como la explicitación de los objetivos, pero a veces, un programa puede no explicitarlos ya que es parte de un diseño curricular modular y sus objetivos estarán definidos en ese marco. El autor señala que inclusive aquellos parámetros relevantes que hacen referencia a los beneficios en términos de aprendizaje, se relacionan al diseño curricular y al modo de uso de los medios por el docente.

Quizás, sostiene Pina (1998), debería considerarse el *“uso didáctico de los medios”*, ya que este es el aspecto clave. Por consiguiente, habría que evaluar el software en función del uso que se hiciera de él. De este modo siempre habría una forma original para aplicar un programa en los aprendizajes. Sería deseable, definir criterios que ayuden a los docentes a la selección, ya que no existe a información acerca de la evaluación median-

te el uso controlado de un programa determinado. En estos casos, cabe recurrir a la experiencia o a la consulta de los grandes distribuidores como Anaya Interactiva²¹ y Zeta Multimedia²².

Conclusiones del Estado del Arte

En estas primeras cuatro secciones, se presentó la evolución de los programas educativos de computadora, como un estudio diacrónico, paralelamente a los desarrollos de las teorías de aprendizaje y líneas más difundidas desde los años sesenta.

Se observa, la base conductista de Skinner²³ que permeó los primeros desarrollos y se puede decir que, en sus inicios, toda una época hasta aproximadamente los setenta, fue educada de acuerdo a los principios de la instrucción programada, aunque no exclusivamente sobre la base de software.

Estos primeros diseños, están muy lejos de los actuales hipermedias didácticos, desarrollados mediante aplicación de estrategias cognitivas y metacognitivas con una fundamentación muy fuerte basada en estudios psicología cognitiva.

Mediante este gran paso hacia el conocimiento de los procesos mentales, la psicología cognitiva, intenta la aplicación de estrategias específicas de aprendizaje y del propio conocimiento. Este conocimiento de los propios aprendizajes, es el que permite hacer hincapié en determinadas técnicas específicas para desarrollar ciertas funciones del pensamiento. La interpretación de estas técnicas por parte de los programadores, es de fundamental importancia, como así, la superación de la etapa de construcción de programas monolíticos de software y poco documentados.

La aparición de la programación estructurada y la diversidad de lenguajes que se suceden a partir de los primeros: Basic, Fortran, Cobol, conducen al Smalltalk, C, C+, Visual Basic, etc., que diversifican el panorama actual, pero a la vez permiten la aplicación específica y el uso de la programación orientada a objetos y a eventos, para facilitar los desarrollos de los programas actuales con una gran gama de recursos disponibles, impensables, diez años atrás, por ejemplo.

Uno de los factores, a tener en cuenta, es el vertiginoso desarrollo tecnológico de los últimos años, que facilita las comunicaciones. Las NTIC (Nuevas Tecnología de Comunicación e Información) han sufrido durante los últimos años ritmo de avance muy notorio, cada vez más, se hace necesaria una habilidad mayor en el manejo y procesamiento de volúmenes de información cada vez más grandes. Esto repercute sobre la población estudiantil, ya que se necesitan estudiantes con nuevas habilidades, no sólo de gestión, sino de selección y de acceso a la información que permitan generar nuevo conocimiento. Los programas educativos deben cambiarse y actualizarse en función de estas necesidades, pero es el docente el que debe considerar cuál es la magnitud de dicha necesidad y cuál será su grado de incorporación al aula.

Siempre se vuelve al triángulo irreductible: docente-alumno-contenido, como el punto de partida de toda práctica educativa. Si bien, algunos plantean que los aprendizajes mediados, son en sí, el uso de los medios didácticos, siendo éstos los que determinan en definitiva la eficiencia del proceso de aprendizaje. Aquí se debe señalar la necesidad de hacer buen uso de tales medios y para ello habría que redefinir qué es ser un buen docente, como aquel que hace buen uso didáctico de los medios y no sólo un buen uso de los medios didácticos.

Habría que señalar entonces que hay una relación directa entre el estilo docente y la eficiencia del medio: *¿Un docente en cuestión o un docente en rutina?* Como señala Fernández Pérez (1995), esta es una de las variables más importantes, a tener en cuenta en el acto didáctico, ya que es el estilo docente el que condiciona el uso didáctico de los medios.

Es por ello, que se debe remarcar que existe una gran cantidad de variables a tener en cuenta en las prácticas educativas y que difícilmente se puedan comparar experiencias, a menos que estas se realicen de un modo muy controlado. Pero, trabajar en un ámbito de “laboratorio”, diseñando experiencias que sean “comparables”, muchas veces implica cambiar el ambiente natural donde se desarrolla el aprendizaje. Otras veces, significa dejar variables de lado ponderando a priori que su influencia en los resultados es insignificante.

Quedaría por señalar que las largas listas de criterios a desarrollar para evaluar los programas educativos, son datos relativos a la hora de hacer uso del recurso educativo. El rol docente, es el que condiciona, el uso de los programas, siendo la creatividad y la originalidad de las propuestas las que permiten incrementar el valor de los medios y no el medio mismo.

De acuerdo a las necesidades en cada caso, y a la teoría educativa y/o el curriculum, se deberán adaptar algunos de los paradigmas del ciclo de vida, discriminando en cada etapa las actividades a realizar con la documentación, las técnicas y herramientas a utilizar. Este ciclo de vida elegido, será acorde al tipo de lenguaje de programación, particulares como los orientados a objetos, de acuerdo al tipo de proyecto o programa, se realizarán las estimaciones de tiempo, personal y costo, de algún modo convencional.

²¹ www.anayainteractiva.com

²² www.zetamultimedia.es

²³ Citado en Fernández Pérez (1995).

En la mayor parte de los diseños realizados, los usuarios, en general solicitan uno o más prototipos, para ver como evoluciona el desarrollo durante el tiempo previsto y si éstos coinciden con sus expectativas. Aunque se han descrito varios de los modelos de ciclo de vida más usados, es recurrente inclusive en orientación a objetos utilizar modelos con prototipos evolutivos, aunque no se excluyen otras posibilidades.

Se prevé plantear de este modo una posible solución a la problemática de los desarrollos de los programas educativos actuales, mediante la definición de los procesos de construcción de los programas educativos y las tareas involucradas en cada uno de ellos, considerando las herramientas y técnicas a usar en cada una de ellas. Esto podría traducirse en la adopción de una gran matriz de procesos, actividades y tareas, asociadas a un ciclo de vida.

Quedaría por considerar la buena consistencia en la simbiosis de la programación estructurada y la teoría del aprendizaje significativo de Ausubel (1997) y los mapas conceptuales de Novak (1988), de acuerdo a la línea de la Escuela de Harvard de David Perkins (1995) y Howard Gardner (1997). También habría que considerar los desarrollos de Coll (1994) y de Rogers (1984).

El cuanto a las metodologías de desarrollo de programas educativos: se ha detectado que gran parte de los docentes que intentan hacer un uso de las computadoras lo hacen como una necesidad de estar acorde a los avances tecnológicos, sin hacer un uso racional del recurso. No se detectó el uso de una metodología para el desarrollo de los programas, sino la necesidad de usar programas que faciliten la visualización e interpretación de algunos temas, para los cuales se realizan “programas” sin base metodológica.

Por otra parte, existe también una gran cantidad de programadores, sin base didáctica, que intenta aplicar su propia racionalidad y criterio al diseño de los programas educativos. Muchos de ellos son egresados de institutos terciarios, que sólo conocen fundamentos de programación en algún lenguaje, y están muy lejos de las normativas vigentes y de los criterios de modularización, programación estructurada y más aún de la documentación interna y externa de los programas.

De más está decir, que la metodología en la investigación tecnológica, ya sea esta tecnología dura o blanda, es una de las asignaturas ausentes en la mayoría de las carreras de grado, esta ausencia es la que condiciona y limita la confección de buenas propuestas para el desarrollo de aplicaciones tecnológicas. Habría que pensar además, que a esto aún le falta el ingrediente educativo: la enseñanza orientada hacia la pedagogía de la diversidad y la educación reflexiva, dinámica e informada, sobre la que nos hace reflexionar David Perkins (1995) en sus trabajos.

Quizás también, habría que considerar que gran parte de los desarrollos de software se realizan en forma intuitiva incursionando en las dos áreas y el resultado, a la hora de las aplicaciones es bueno, y hasta se logran los objetivos educativos: el alumno aprende. Quizás, llegado a este punto habría que señalar, que: para un especialista novel, viéndolo desde la posición de etnógrafo, le sería más fácil aplicar una metodología con sólidos fundamentos teóricos sumados a la base de conocimientos adquirida sólo por prueba y error y mucho esfuerzo, pero adquirir estos conocimientos requiere de mucho tiempo.

La base epistemológica es fundamental en ambos campos del saber: el infomático y el educativo, y más aún si los mismos se potencian y las metodologías se suman, a la luz de las semejanzas y diferencias, como dice Morín (1985) desde el paradigma de la complejidad en este fin de siglo y Ander Egg (1995) recrea en sus obras.

Se ha presentado el marco conceptual de los desarrollos de software actuales, para poder incorporar los diseños de programas educativos, siendo a partir aquí que se marca la necesidad de una trabajo interdisciplinario muy fuerte.

Los pilares de las aplicaciones tecnológicas están condicionados, es época de trabajo interdisciplinario, en este caso concibiendo la diversidad y multidimensionalidad de las actividades que se requieren hoy día y su interacción. El desarrollo tecnológico y la práctica docente, deben avanzar juntos; aunque en la realidad no sea así, al menos hasta ahora.

Quizás, habría que capacitar a los profesionales de ambas áreas insistiendo en un nuevo modo de ver las cosas, dejando los reduccionismos y la unidimensionalidad de la realidad por una visión poliocular. (Morín, 1995).

En síntesis: habría que reiterar como solución a las deficiencias detectadas que a la programación estructurada puede sumarse la teoría del aprendizaje significativo con todas sus variantes para el desarrollo de los programas. En cuanto a la evaluación deben considerarse con mucho detenimiento las variables involucradas a la hora de hacer comparaciones.

Descripción de la Problemática

5. Presentación de la problemática

5.1. Resumen

Se desea presentar la problemática que responde a la siguiente pregunta: ¿Qué normativas provee la ingeniería de software y cuáles son las teorías de aprendizaje a aplicar para el diseño del software educativo? (sección 5.2).

Se desglosa la problemática, ya que gran parte de los programas educativos desarrollados en la actualidad no consideran las metodologías provistas por la ingeniería de software para su diseño (sección 5.3). A este defecto se le agrega que en otros casos se desconocen los principios de las teorías de aprendizaje que les dan marco, no se evalúan lo suficiente (sección 5.4) y no se aplican las pautas de calidad (sección 5.5).

5.2. La problemática

Se ha detectado una serie de problemas relacionados con el diseño y el desarrollo del software educativo. Por una parte *existe una falta de capacitación de los docentes de áreas no informáticas, que conduce a intentos de construcción de programas educativos sin fundamentos metodológicos y una falta de capacitación en el área educativa de los programadores que intentan desarrollar programas educativos y que en la mayoría de las veces caen en desarrollos meramente conductistas.* Esto no significa que la base conductista no sea apropiada en determinados contextos y favorecedora de determinadas situaciones de aprendizaje.

Por estos motivos, realizar un producto con características tan particulares, requiere de una sólida base en ingeniería de software sumada a un conocimiento de las teorías de aprendizaje y a los ámbitos de aplicación, es decir a los entornos socio-educativos actuales.

El problema se puede dividir, para su mejor estudio, en dos subproblemas relacionados: por un lado el problema de cómo diseñar y desarrollar programas educativos de calidad y por otro lado cómo evaluar este tipo de desarrollos.

5.3. El diseño y desarrollo del software educativo

Los programas educativos constituyen un producto que posee algunas características, como las técnicas, que pueden ser evaluadas mediante el uso de métricas adecuadas y otras que están relacionadas con la significatividad de los aprendizajes, en un ámbito donde la cantidad de variables a tener en cuenta son muchas: el estilo docente, el modo de uso de la herramienta informática por el docente, el estilo de aprendizaje de los alumnos, el contexto áulico, el estilo institucional, el tipo de currículum, etc.

Algunas otras características las define el usuario final, es decir, son los alumnos quienes en última instancia deciden si el producto es eficiente o no, después de haberse finalizado el programa y realizado la evaluación contextualizada.

El uso de metodologías adecuadas de desarrollo en principio, puede solucionar el problema de la calidad, para obtener un programa educativo, libre de errores, o con una tasa de errores no encontrados ni detectados, tolerable, esto requiere a su vez de un equipo de desarrollo con experiencia. Pero, como en cada producto desarrollado debe desencadenar un aprendizaje de cierto tipo o desarrollar una habilidad o capacidad en particular, sumada a la calidad técnica se necesita una evaluación desde el punto de vista de los aprendizajes que se pretenden lograr.

5.4. La evaluación del software educativo

La evaluación es para los programas educativos, la etapa más importante del todo el proceso de construcción, evaluando desde el diseño del producto y la producción del mismo, hasta el modo de uso, el tiempo y el momento de uso. La evaluación es una tarea constante a lo largo de todo el desarrollo y aún después, en el contexto de aplicación, ya que requiere también de evaluación de las estrategias cognitivas propuestas.

Como la cantidad de software educativo ha crecido muy rápidamente, el docente se encuentra con la necesidad cada vez mayor de evaluarlo para determinar el grado de adecuación de un programa su propio entorno. Ellos necesitan saber cómo utilizar un programa determinado y cuándo deberían utilizarlo para mejorar su enseñanza. Por otra parte los alumnos también deben saber cómo mediante tal o cual programa podrían mejorar sus aprendizajes.

Es cierto, que confeccionar un instrumento de evaluación y realizar la evaluación de un programa en particular con un grupo de alumnos específico no brindará resultados generalizables a todos los ámbitos de aplicación, pero esta puede ser una guía como punto de partida de selección del programa para el docente.

Los proveedores de programas educativos deberían informar y aconsejar al docente acerca de la conveniencia de usar tal o cual programa adecuándose a las necesidades de éste, pero son muy pocas las empresas que realmente brindan el asesoramiento pedagógico.

En la actualidad son muy pocos los programas que tienen documentación interna y aún externa. En muchos casos los manuales de usuario se remiten a especificaciones técnicas y requerimientos del programa.

Las herramientas de evaluación que se han relevado son largas y tediosas listas de preguntas o listas de verificación que en la mayoría de los casos no pueden ser aplicadas más que a una situación de aprendizaje en particular: un programa educativo, con un docente con un estilo propio, con alumnos en una situación áulica, en una realidad diferente a las demás. Por este motivo, una lista de evaluación significa personalizarla a las condiciones de trabajo en particular. De este modo sólo se pueden usar sus resultados como orientativos y no cómo comparativos entre dos situaciones de aprendizaje diferentes.

Es en este momento donde se debe recordar que aprender es el fin y las computadoras es el medio mediante el cual esos aprendizajes se pueden facilitar y que no es el “*único medio*”: si no con sólo recordar que “*Sócrates no tenía computadoras para enseñar*”, bastaría para darse cuenta que si bien esta herramienta nos permite gestionar los grandes volúmenes de información que hoy día se manejan, es tan sólo uno de los medios a través de los cuales se puede aprender. Un muy buen medio mal empleado puede ser más nocivo que el más discreto de los medios empleado en el momento preciso, en el lugar adecuado y en la cantidad necesaria.

Cuando se evalúan los programas educativos se consideran largas listas de criterios aplicadas en situaciones que distan mucho del contexto áulico. Estas listas carecen en algunos casos de sentido práctico por su falta de dinámica y de adaptación al cambio tecnológico constante.

Algunas evaluaciones experimentales, utilizan grupos de alumnos (de igual edad, distribución de género, conocimientos previos) con comparación de resultados. Pensar en comparar los resultados obtenidos, mediante pruebas tradicionales entre un grupo que utiliza la herramienta computacional y el software educativo con otro que utiliza medios convencionales como libros, significa caer en el reduccionismo de pensar que aprender es reproducir contenidos, repetir o memorizar contenidos.

Este es uno de los problemas que a menudo se presenta en las evaluaciones del software educativo: realizar una evaluación mediante una lista de comprobación, es útil, pero faltan los alumnos y evaluar logros de dos grupos comparando resultados, significa considerar al alumno sin la influencia del entorno social, la motivación inherente de esta contextualización y perder de vista la expectativa social que lo conduce a adquirir ese aprendizaje.

5.5. La calidad en el software educativo

El problema de la calidad se debe situar en el contexto de diseño y desarrollo como un producto de software más y en el contexto de calidad educativa como respuesta a un modelo de aprender o un modelo curricular, según las características de un programa que puede ser de refuerzo a una clase explicativa o de apoyo para trabajo colaborativo.

El problema de la determinación de la calidad en el uso de medios de comunicación es uno de los problemas educativos recurrentes. Si bien se pueden utilizar los catálogos de listas de verificación, pero la idea es dar a estas listas un marco de uso práctico.

Cuando se habla de calidad de la enseñanza que se ofrece a los estudiantes como uno de los objetivos educativos a lograr mediante aplicación de programas bien diseñados, se piensa en el logro de los estudiantes con un perfil lo más cercano posible al ideal.

La calidad educativa de los programas se puede ver como la potenciación de habilidades cognitivas y de adquisición de conocimientos, mediante el uso de programas específicos que desencadenan las funciones superiores del pensamiento.

Por otra parte se quiere estudiantes, capaces de controlar el contenido de sus aprendizajes y de su trabajo, promoviendo el autoaprendizaje²⁴, y el aprender a aprender, mediante el uso y conocimiento de estrategias metacognitivas.

Por estas razones, en las secciones siguientes se planteará, una de las posibles soluciones a esta problemática. Para ello, se tendrán en cuenta las herramientas y recursos a utilizar descriptos en el Estado del Arte, debiéndose destacar que existe además una necesidad imperiosa de rediseño de las prácticas educativas²⁵ y una concientización de un perfeccionamiento docente constante a lo largo de la trayectoria del mismo.

Esto permitirá considerar, analizar, incorporar y por último poner en práctica no sólo el uso de los programas educativos, sino aspectos concernientes a su diseño, desarrollo y evaluación, utilizando los criterios científicos y tecnológicos apropiados y acordes a las necesidades particulares.

²⁴ o aprendizaje autónomo.

²⁵ en referencia a que: “*La cultura del aprendizaje dirigida a la reproducción de saberes previamente establecidos, debe dar paso a una cultura de la comprensión, del análisis crítico, sobre lo que hacemos y creemos. ...*” (Pozo Municio, 1998).

Solución Propuesta

6. Propuesta de metodología de diseño y desarrollo

6.1. Resumen

Se propone y se justifica el ciclo de vida elegido para el diseño del software educativo (sección 6.1). Se consideran las etapas del ciclo y se define la matriz de actividades (sección 6.4). A partir de ella se describen las herramientas y las técnicas que se utilizarán en cada uno de los procesos considerados.

En la propuesta se ha elegido un ciclo de vida de prototipos evolutivos con refinamientos sucesivos como punto de partida. En la etapa metodológica de diseño se integra a los instrumentos de representación clásicos los que provee el enfoque cognitivista-constructivista.

La propuesta metodológica considera la construcción de programas educativos desde un aspecto integral, teniendo en cuenta los aspectos pedagógicos en el ciclo de vida.

Se pone un interés especial en la configuración de los perfiles de los diferentes profesionales del equipo de desarrollo. (sección 6.4), en el diseño del programa (secciones 6.5 y 6.6) y en la confección de la documentación de los procesos (sección 6.7).

6.2. La elección del ciclo de vida

El modelo de ciclo de vida elegido es el de *prototipos evolutivos con refinamientos sucesivos*, por varios motivos a saber:

- Cuando el software a desarrollar es por encargo, es interesante tener una idea de cómo será el programa lo antes posible, y a fin de disminuir las expectativas del cliente o usuario, se le irán entregando prototipos con funcionalidades en forma incremental, para que se los pruebe durante un período de tiempo a convenir y haga las sugerencias y los cambios en etapas lo más tempranas posibles del ciclo de vida.
- Por otra parte, es importante cuando se desea que el usuario sepa cuanto antes si el producto tal como se lo interpretó está de acuerdo a sus necesidades y consideraciones.
- En muchos casos, el usuario no puede dar una idea detallada de lo que desea, y debido a ello, el desarrollador no termina de saber qué es lo que éste quiere exactamente, por lo que cada prototipo realizado, significa una revisión de los requerimientos y un refinamiento de dichos requerimientos a fin de acercarse al producto final.

En el ciclo de vida de prototipo incremental se definen las siguientes etapas:

1. *Factibilidad (FAC)*
2. *Definición de requisitos del sistema (RES)*
3. *Especificación de los requisitos del prototipo (REP)*
4. *Diseño del prototipo (DPR)*
5. *Diseño detallado del prototipo (DDP)*
6. *Desarrollo del prototipo (codificación) (DEP)*
7. *Implementación y prueba del prototipo (IPP)*
8. *Refinamiento iterativo de las especificaciones del prototipo (aumentando el objetivo y/o el alcance). Luego, se puede volver a la etapa 2 o continuar si se logró el objetivo y alcance deseados. (RIT)*
9. *Diseño del sistema final (DSF)*
10. *Implementación del sistema final (ISF)*
11. *Operación y mantenimiento (OPM)*
12. *Retiro (si corresponde) (RET)*

A continuación se describen cada una de las etapas del ciclo de vida elegido que formarán parte de la matriz de actividades²⁶:

Factibilidad (FAC): En esta etapa se define el producto software y se determina su factibilidad en el ciclo de vida desde la perspectiva de la relación costo –beneficio, como así las ventajas y desventajas respecto de otros productos.

Requisitos del sistema (RES): En esta etapa se deben definir las funcionalidades requeridas para el desarrollo del sistema (o programa), las interfaces y el tipo de diseño.

Especificación de requisitos del prototipo (REP): Consiste en especificar las funciones requeridas, las interfaces y el rendimiento para el prototipo. Aquí se considerarán incrementos en porcentajes de la funcionalidad total del sistema.

Diseño del prototipo (DPR): Es poner en ejecución del plan del prototipo, ya que una vez fijadas las restricciones con el usuario, hay que mostrar el mismo funcionando, aunque sean sólo algunas fun-

²⁶ La matriz de actividades base para el diseño de software, se adaptó de Juristo Juzgado N. (1996): *Proceso de Construcción del software y ciclos de vida en módulos de Proyectos de Software*. Universidad Politécnica de Madrid.

cionalidades restringidas. Aquí, hay que hacer un análisis de cómo se va a trabajar, qué módulos se van a hacer, con qué lógica y qué funciones se van a usar.

Diseño detallado del prototipo (DDP): Esta etapa es una especificación verificada de la estructura de control, la estructura de los datos, las relaciones de interfaces, el tamaño, los algoritmos básicos y las suposiciones de cada componente del programa. En esta etapa no sólo se definen y sino que se documentan los algoritmos que llevarán a cabo la función a realizar por cada uno de los módulos. El diseño de software, es un proceso que se centra en cuatro atributos distintos del programa: la estructura de datos, la arquitectura del software, el detalle procedimental y la caracterización de la interface. En este proceso deben traducirse los requisitos a una representación del software que pueda ser establecida de forma que se obtenga la calidad requerida antes de que comience la codificación.

Desarrollo del prototipo (codificación) (DEP): Consiste en realizar la codificación o diseño detallado, en forma legible para la máquina.

Implementación y prueba del prototipo (IPP): Consiste en lograr un funcionamiento adecuado del producto software en el sistema informático, funcionando operacionalmente, incluyendo objetivos tales como la conversión del programa y datos (si la hubiere), la instalación y el entrenamiento. La prueba debe asegurar que se han probado todas las sentencias del mismo, y que en las funciones externas se han realizado pruebas que aseguren que la entrada definida produce los resultados que se esperan realmente.

Refinamiento iterativo de las especificaciones del prototipo (RIT): Es un aumento de la funcionalidad del sistema, para luego volver REP a fin de aumentar la funcionalidad del prototipo o continuar, si se logró el objetivo y alcance deseados.

Diseño del sistema final (DSF): Consiste en ajustar las restricciones o condiciones finales e integrar los últimos módulos.

Implementación del sistema final (ISF): Es el sistema de informático funcionando operativamente, incluyendo tales objetivos como conversión del programa y datos, (si la hubiere), la instalación y La capacitación del personal..

Operación y mantenimiento (OPM): Es la puesta en funcionamiento del sistema informático, objetivo que se repite para cada actualización.

Retiro (RET): Es una transición adecuada de las funciones realizadas para el producto y sus sucesores. Luego, se definen los procesos básicos para este ciclo de vida, y las actividades para cada uno de ellos. Los procesos incluyen aquellos concernientes al desarrollo de software y los específicos teniendo en cuenta los aspectos educativos, aunque en la propuesta, no se define una teoría educativa en particular, sino que las actividades permiten ver un enfoque cognitivista-constructivista.

En la Tabla 6.1, se puede ver la matriz de actividades, con el desglose de actividades para cada uno de los procesos para el ciclo de vida elegido.

En la matriz de actividades, se puede observar en color gris, las etapas del ciclo de vida involucradas al incrementar las funcionalidades a fin de obtener los diferentes prototipos, que han de desarrollar. En *arial normal* se destacan las actividades relativas al desarrollo de software propiamente dicho y en *arial cursiva* a aquellas actividades concernientes a definir y considerar los aspectos educativos didáctico-pedagógicos para desarrollar el software.

Una vez definidas cada una de las actividades, quedaría por considerar qué herramientas y técnicas se utilizarán para cada una de ellas, teniendo en cuenta el modelo de ciclo de vida elegido (prototipado evolutivo) y la teoría educativa que sustente al desarrollo.

6.3. La matriz de actividades (Tabla 6.1)

ACTIVIDADES DE LOS PROCESOS	FAC	RES	REP	DPR	DDP	DEP	IPP	RIT	DSF	ISF	OPM	RET
Proceso de identificación de la necesidad educativa												
<i>Identificar necesidad del programa educativo</i>	✓											
<i>Identificar o seleccionar la teoría educativa a utilizar para el desarrollo de acuerdo a la necesidad</i>	✓											
Proceso de selección del modelo de ciclo de vida												
Identificar de los posibles modelos ciclos de vida el que más se adapta a las necesidades.	✓											
<i>Seleccionar un modelo para el programa de acuerdo a la teoría educativa elegida.</i>	✓											
Proceso de iniciación, planificación y estimación del proyecto												
<i>Establecer la matriz de actividades para el ciclo de vida considerando la teoría educativa a usar</i>	✓											
Asignar los recursos del proyecto/programa	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Definir el entorno del proyecto/programa	✓	✓	✓		✓			✓				
Planificar la gestión del proyecto/programa	✓	✓	✓					✓				
Proceso de seguimiento y control del proyecto												
Analizar los riesgos	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
Realizar la planificación de contingencias		✓	✓	✓	✓	✓	✓	✓	✓	✓		
Gestionar el proyecto/programa	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Implementar el sistema/programa	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Archivar registros			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Proceso de gestión de calidad del software												
Planificar la garantía de calidad del software		✓	✓	✓				✓				
Desarrollar métricas de calidad		✓	✓	✓	✓	✓		✓				
Gestionar la calidad del software	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Identificar a los responsables de cada tarea	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Implementar el sistema de informes de problemas			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Archivar registros			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Proceso de exploración de conceptos												

<i>Identificar las necesidades educativas</i>	✓											
<i>Formular las posibles soluciones potenciales</i>	✓	✓	✓									
Identificar las necesidades del soporte lógico	✓	✓	✓						✓			
<i>Formular soluciones potenciales compatibles</i>	✓	✓	✓									
Realizar los estudios de viabilidad	✓	✓							✓			
Refinar y concretar la idea o necesidad		✓	✓						✓			
Proceso de asignación del sistema												
Analizar las funcionalidades del sistema/programa		✓	✓	✓					✓			
<i>Definir las funcionalidades del programa</i>		✓	✓	✓								
<i>Desarrollar la arquitectura del sistema/programa basándose en la teoría educativa elegida.</i>		✓	✓	✓								
Descomponer los requisitos del sistema/programa		✓	✓									
Proceso de análisis de requisitos educativos												
<i>Definir los objetivos educativos</i>		✓	✓	✓								
<i>Definir las características del grupo destinatario</i>		✓	✓	✓								
<i>Definir los contenidos y el recorte de contenidos</i>		✓	✓	✓								
<i>Definir las estrategias didácticas</i>		✓	✓	✓								
<i>Definir las actividades mentales a desarrollar</i>		✓	✓	✓								
<i>Definir el nivel de integración curricular</i>		✓	✓	✓								
<i>Definir el tipo de uso del programa y nivel de interactividad</i>		✓	✓	✓								
<i>Definir los efectos motivantes</i>		✓	✓	✓								
<i>Definir los posibles caminos pedagógicos</i>		✓	✓	✓								
<i>Definir el tiempo y modo de uso</i>		✓	✓	✓								
<i>Definir el hardware necesario</i>		✓	✓	✓								
Proceso de análisis de requisitos del software												
<i>Definir el tipo de programa a desarrollar</i>		✓	✓	✓								
Definir los requerimientos de las interfaces		✓	✓	✓					✓			
<i>Definir el tipo de interactividad</i>		✓	✓	✓								
Definir y desarrollar los requisitos del software		✓	✓	✓					✓			
<i>Priorizar e integrar los requisitos educativos con los del software</i>		✓	✓	✓					✓			
Proceso de diseño												
<i>Definir la organización de los menús</i>				✓	✓							
<i>Definir el tipo de íconos a usar</i>				✓	✓							
<i>Seleccionar los efectos a usar: sonidos, música, animaciones, videos, etc.</i>				✓								

<i>Seleccionar los textos a usar</i>				✓								
<i>Asegurar la facilidad de lectura</i>				✓								
<i>Realizar el diseño de las pantallas</i>				✓	✓							
<i>Realizar el diseño de los menús</i>				✓								
<i>Realizar los storyboards (si corresponde)</i>				✓	✓							
Realizar el diseño preliminar				✓								
Analizar el flujo de información				✓	✓							
Diseñar la base de datos (si la hubiere)				✓	✓							
Diseñar las interfaces				✓	✓							
<i>Definir los criterios de navegación</i>				✓								
<i>Definir las actividades: Información, preguntas, búsqueda, resolución de ejercicios, etc.</i>				✓								
<i>Definir los tipos de módulos a usar: de evaluación, de problemas, de preguntas, etc.</i>				✓								
<i>Definir los tipos de ayudas didácticas: errores, mensajes de ayuda, etc.</i>				✓								
Desarrollar los algoritmos		✓	✓	✓	✓							
Realizar el diseño detallado												
Confeccionar la documentación				✓	✓							

Proceso de implementación e integración de módulos

Crear los datos para las pruebas				✓	✓	✓			✓			
Crear el código fuente				✓	✓	✓			✓			
Generar el código objeto				✓	✓	✓			✓			
Crear la documentación de operación				✓		✓			✓			
Planificar la integración de los módulos				✓	✓	✓			✓			

Proceso de instalación y aceptación

Planificar la instalación							✓		✓			
Distribuir el software							✓			✓		
Instalar el software							✓			✓		
Cargar la base de datos si la hubiere							✓			✓		

ACTIVIDADES DE LOS PROCESOS	FAC	RES	REP	DPR	DDP	DEP	IPP	RIT	DSF	ISF	OPM	RET
Aceptar el software en el entorno de operación							✓			✓		
Realizar las actualizaciones pertinentes							✓				✓	
Proceso de operación y soporte												
Operar el sistema/programa											✓	

Proveer asistencia técnica y consultas on line.											✓	
Mantener el historial de pedidos de soporte											✓	
Proceso de mantenimiento												
Realizar el mantenimiento correctivo											✓	
Reaplicar el ciclo de vida del software											✓	
Proceso de retiro												
Notificar al usuario											✓	✓
Conducir operaciones en paralelo												✓
Retirar el sistema												✓
Proceso de verificación y validación												
Planificar la verificación y validación del software		✓	✓	✓								
Ejecutar las tareas de verificación y validación		✓	✓	✓	✓							
Recoger y analizar los datos de las métricas		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓
Planificar las pruebas de V y V					✓	✓	✓		✓	✓	✓	✓
Desarrollar las especificaciones de las pruebas					✓	✓						
Ejecutar las pruebas							✓		✓			
Archivar los resultados							✓		✓	✓		
Proceso de evaluación de los prototipos del software												
<i>Confeccionar el instrumento de evaluación</i>							✓					
<i>Evaluar los prototipos del programa</i>							✓					
<i>Elaborar los resultados</i>							✓					
<i>Identificar los cambios y ajustes a realizar</i>							✓					
<i>Llevar a cabo las modificaciones pertinentes</i>							✓					
<i>Archivar los resultados</i>							✓					
Proceso de evaluación interna y externa del software												
<i>Confeccionar los instrumentos de evaluación interna y externa</i>									✓			
<i>Realizar las evaluaciones interna y externa.</i>									✓			
<i>Elaborar los resultados</i>									✓			
<i>Llevar a cabo las modificaciones pertinentes</i>									✓			
<i>Obtener la versión final del programa.</i>									✓			
<i>Archivar los resultados</i>									✓			
Proceso de evaluación contextualizada												
<i>Diseñar la evaluación: definir grupo/s (caso de control y experimen-</i>									✓			

<i>tal), tiempo, docente, materiales a usar y modo.</i>												
<i>Aplicar de la prueba</i>									✓			
<i>Identificar posibles problemas.</i>									✓			
<i>Realizar las modificaciones y ajustes a la versión</i>									✓			
<i>Archivar los resultados</i>									✓			
Proceso de configuración												
Planificar la gestión de la configuración		✓	✓	✓								
Realizar la gestión de la configuración		✓	✓	✓	✓	✓	✓		✓			
Realizar el control de la configuración				✓	✓	✓	✓		✓	✓	✓	✓
Realizar la información del estado de la configuración				✓	✓	✓	✓		✓	✓	✓	✓
Proceso de documentación técnica												
Planificar la documentación interna			✓	✓	✓							
Planificar la documentación externa			✓	✓	✓							
Implementar las documentaciones						✓	✓					
<i>Incluir los resultados de las evaluaciones</i>							✓		✓			
Elaborar el manual de usuario con información técnica.			✓	✓	✓	✓	✓	✓	✓			
Producir y distribuir la documentación			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Proceso de documentación didáctica												
<i>Planificar la documentación didáctica</i>			✓	✓	✓							
<i>Elaborar una guía didáctica, con ejemplos de uso.</i>								✓	✓			
<i>Adjuntar la información didáctica pertinente, los caminos pedagógicos, las teorías de aprendizaje y la programación didáctica.</i>									✓			
<i>Producir la documentación y adjuntarla al programa.</i>						✓	✓		✓	✓		
Proceso de formación del personal												
Planificar el programa de formación		✓	✓	✓								
Desarrollar los materiales de formación			✓	✓	✓	✓	✓		✓			
Validar el programa de formación						✓	✓		✓	✓		
Implementar el programa de formación							✓			✓		

Tabla 6.1: Matriz de actividades básica para un programa educativo, con ciclo de vida de prototipado evolutivo.

En la Tabla 6.2, se pueden observar los métodos, las técnicas y/o herramientas a emplear en cada uno de los diferentes procesos

Procesos		Documento de salida	Métodos/Técnicas/Herramientas a emplear
<i>Proceso de identificación de la necesidad educativa</i>		Definición del marco educativo y comunicacional.	Encuesta, entrevista
Proceso de selección del modelo de ciclo de vida		Ciclo de vida adoptado	
Proceso de iniciación, planificación y estimación del proyecto		Plan de gestión del proyecto	Diagrama de Gantt o Pert (CPM). Modelos empíricos de estimación
Proceso de seguimiento y control del proyecto (programa),		Análisis de riesgos y plan de contingencias. Registro histórico del proyecto	Modelizado. Prototipado. Revisiones. Auditorías. Análisis CPM.
Proceso de gestión de calidad del software		Plan de garantía de calidad. Recomendaciones de mejora de calidad.	Técnicas de planificación. Métricas de calidad del software
Proceso de exploración de conceptos		Informe de necesidades. Posibles soluciones factibles	Análisis Costo Beneficio. DFD. Prototipado
Proceso de asignación del programa (sistema).		Especificación de requisitos funcionales de hardware y software. Especificación de interfaces del sistema o programa. Descripción funcional. Arquitectura.	DFD Módulos
<i>Proceso de análisis de requisitos educativos</i>		Especificación de los objetivos y estructuración de conceptos. Selección de contenidos y pertinencia.	Enfoques cognitivistas. Enfoques constructivistas. Estrategias cognitivas.
Proceso de análisis de requisitos del software		Especificación de requisitos del software, de interfaces de usuario y otros software. Interface de hardware y con el sistema físico.	Análisis estructurado. DFD. DD. Diagramas E/R. Técnicas de Prototipación.
Proceso de diseño	de los contenidos	Identificación de los procesos mentales a estimular. Definición de las actividades a realizar por los alumnos. Jerarquización de los conceptos.	Uso de estrategias cognitivas. Teoría de Ausubel y Novak. ²⁷ Uso de mapas conceptuales.
	del software	Descripción del diseño del software y de la arquitectura. Descripción del flujo de información, bases, interfaces y algoritmos.	Programación estructurada. Programación Orientada a objetos. Técnicas de prototipado.
Proceso de implementación e integración de módulos		Datos para las pruebas. Documentación del sistema o programa y del usuario. Plan de integración.	Lenguajes de Programación
Proceso de instalación		Plan de instalación. Informe de Instalación.	Lenguajes de Programación
Proceso de operación y soporte		Histórico de pedidos de soporte	Análisis estadístico.
Proceso de mantenimiento		Recomendaciones de mantenimiento	Reaplicar el ciclo de vida
Proceso de retiro		Plan de retiro	
Proceso de verificación y validación		Plan de verificación y validación Plan de pruebas. Especificación y resumen de la prueba. Software probado.	Pruebas de caja negra y pruebas de caja blanca
<i>Proceso de evaluación de los prototipos</i>		Diseño del instrumento de evaluación.	Cuestionario estructurado, semi y

²⁷Se ha señalado esta teoría como una guía simplemente, a modo de referencia, puede utilizarse otra, de acuerdo a las necesidades y/o tipo de propuesta educativa. Ver referencias: Ausubel (1983) y Novak (1988).

<i>pos del software</i>	Resumen de la prueba. Selección de la muestra.	abierto.	
<i>Proceso de evaluación interna y externa del software</i>	Diseño del instrumento de evaluación. Resumen de la prueba. Selección de la muestra.	Cuestionario estructurado, semi y abierto	
<i>Proceso de evaluación contextualizada</i>	Diseño de la experiencia. Definición de los grupos de control y experimental.	Técnicas de análisis pre-post. Test de Raven. Prueba de Wilcoxon.	
Proceso de configuración	Plan de gestión de la configuración	Base de datos	Diagramas Pert y Gantt
Proceso de documentación técnica	Plan de documentación técnica.		
<i>Proceso de documentación didáctica</i>	Plan de confección de la documentación didáctica.		
Proceso de formación y capacitación del personal	Plan de formación y capacitación		

Tabla 6.2: Definición de los métodos, técnicas y/o herramientas a utilizar en cada proceso

6.4. El equipo de trabajo

La creación de los programas educativos, es una tarea que compete a diferentes áreas del saber, y, en este tipo de proyectos educativos es fundamental la formación y la conformación de los equipos de desarrollo.

Para ello, es necesario recurrir a especialistas en desarrollos de software, planificadores eficaces y docentes conocedores del área de experticia en que se desarrollará el programa ya que en cada caso habrá que generar un modelo pedagógico de acuerdo a las necesidades.

Si bien, la conformación de los equipos de desarrollo para software comercial, es una instancia que requiere de la habilidad del líder del proyecto y de las buenas relaciones entre los integrantes, en este caso en particular, la situación se vuelve mucho más crítica, debido a interdisciplinariedad del equipo, centrado en la tarea de integrar y coordinar profesionales de campos disciplinares diferentes. Básicamente; se requieren de profesionales con los perfiles que se señalan en la Tabla 6.3:

<i>Profesionales del área en la que se quiere desarrollar el software</i>	Profesores y especialistas en pedagogía para determinar los contenidos a incluir y expertos en el área de desarrollo
<i>Profesionales desarrolladores de software</i>	Analistas y programadores. Para el análisis del proyecto y la codificación.
<i>Coordinador del proyecto</i>	Como en todo proyecto soportado por una ingeniería de base, recaerá en el ingeniero de software.
<i>Personal técnico de apoyo (diseño gráfico y sonido)</i>	De acuerdo a las dimensiones del desarrollo habrá operadores, diseñadores gráficos, especialistas en sonido, vídeo.

Tabla 6.3: Profesionales que se requieren para construir software educativo.

Algunos autores como Marquès (1995) consideran al proceso de desarrollo del software en un eje centrado en el equipo pedagógico. Se deberá tener en cuenta que el programa a desarrollar tiene dos aspectos: uno que es el logro de la calidad técnica que es responsabilidad del equipo técnico y éste debe ser el eje central, ya que diseñar y desarrollar software es responsabilidad de profesionales del área. La determinación de las tareas a asignar de cada uno integrantes del equipo de desarrollo, en cada una de las etapas juega un rol crucial a la hora de optimizar los tiempos de desarrollo.

Cabe señalar que en algunos casos que así lo requieran se recurrirá a asesores en tecnologías informáticas muy específicas y de redes aplicadas a la educación, y psicopedagogos que orientarán al equipo cuando se requiera un producto con alguna característica particular.

Uno de los aspectos a tener en cuenta en todo equipo de trabajo, es una perfecta definición de las actividades de cada uno de los miembros. El “*quién hace qué*”, facilita la identificación de los responsables en cada una de las etapas de trabajo, y permite un seguimiento evolutivo de cada uno de los componentes del software a lo largo de su desarrollo.

Por otra parte, de acuerdo al modelo de desarrollo propuesto, habrá evaluaciones del producto, donde tendrán que intervenir todos los integrantes del equipo.

6.5. El primer diseño del programa

Para diseñar el entorno de comunicación, cuando el equipo de desarrollo tiene escaso conocimiento en informática, comúnmente se utiliza la técnica de guiones o de *"storyboard"*. Estas series de bosquejos, se los debe considerar como una puesta en común de las necesidades, y se llevan a cabo a partir del acuerdo de las partes del equipo.

Los desarrolladores de software expertos y especialistas en el diseño de interfaces humanas elaborarán algunos prototipos de las pantallas a tal efecto. Es por ello, que se descartará el uso de los guiones en este sentido, no así desde la perspectiva de un punto de partida común entre docentes y programadores para interpretar los requerimientos de trabajo.

6.6. Acerca del diseño

Al análisis de requisitos educativos, le sobreviene el análisis de requisitos de software. Luego, la etapa de diseño del software educativo es una de las más importantes en el ciclo de vida de este producto, donde deben interactuar los especialistas en pedagogía y en contenidos con los de diseño de software desde la perspectiva computacional.

Se puede dividir la etapa de diseño, en dos grandes sub-etapas: una es el proceso de diseño de los contenidos y la otra el proceso de diseño del software.

Posteriormente, el diseño de los contenidos se integrará con el del software, siendo este un punto crucial, ya que aquí es donde hay que poner a prueba las estrategias cognitivas, para que el alumno tenga la posibilidad de acceder al conocimiento a través de las diferentes actividades propuestas.

Aquí, resulta fundamental la buena estructuración de los conceptos, considerando en definir correctamente el tema, los objetivos propuestos y las actividades para que el alumno pueda acceder al conocimiento.

Quizás, se podría afirmar que hay que definir las tareas y las actividades, siendo éstas las diferentes situaciones problemáticas referidas a la temática en cuestión que se le presenten al alumno.

En algunos casos, las tareas a ejecutar por los estudiantes pueden estar definidas en niveles de complejidad y de dificultad incrementales. Aunque este puede ser un punto cuestionable, ya que si el software pretende ser de soporte al docente, las tareas y actividades las podría proponer el mismo enseñante, a través de consignas claras y precisas.

Algunos autores, plantean la realización de un análisis subjetivo o sea, un panorama total de la tarea, desde la perspectiva de la tarea misma y del sujeto que la resuelve, de su nivel de dificultad, de la estructura, y de la forma como se organiza y articula con los objetivos y posibles modos de solución. (Valencia, 1998). Este autor plantea el análisis desde la descripción objetiva de la consigna y el análisis desde los factores que desde el punto de vista cognitivo problematizan y/o hacen significativo el conocimiento.

6.7. La documentación

Un aspecto a considerar que es de especial relevancia es el desarrollo de la documentación técnica a lo largo del ciclo de vida del producto: ya sea esta documentación interna y/o externa. La primera considera a todos aquellos comentarios del programa que serán útiles a la hora de realizar modificaciones posteriores, ya sea por el mismo equipo de desarrollo o por otro. Aunque también hay que prever las ayudas on-line a los usuarios, para lo cual se deberá desarrollar un módulo especial a tal efecto, si se la piensa implementar.

La documentación externa, es la que se refiere a todo el material confeccionado desde la etapa inicial de análisis, con los diagramas de entidades y relaciones, estructuras de datos, diagramas de flujos de procesos, diseño modular descendente, etc., y toda aquella que se considere pertinente para interpretar el desarrollo del programa.

Por último hay que señalar la necesidad de un manual del usuario claro y didáctico, para que el docente pueda recurrir a él como elemento de ayuda. En este manual, se considerarán todos los aspectos técnicos requeridos para el funcionamiento del programa y se podría incluir una guía de soluciones, para aquellos problemas más frecuentes.

Se puede considerar también la confección de una *"guía o manual didáctico"*, para brindarle al docente todo aquello que se considere necesario para su aplicación. En esta guía o manual de aplicaciones didácticas se debe dar referencia acerca de: objetivos, contenidos, destinatarios, actividades propuestas, y sería interesante incluir qué teoría o teorías de aprendizaje sustentan el desarrollo y cuál es el tratamiento de los errores de los estudiantes en sus procesos de aprendizajes, que permite el programa.

También se deberían almacenar los resultados de las evaluaciones realizadas, considerando los aspectos técnicos, hayan sido éstos señalados como positivos o negativos y las funcionalidades, detallando los resultados estadísticos y teniendo en cuenta el tipo de instrumento elaborado para la toma de dichos resultados y la selección y el tipo de las muestras, de quiénes evaluaron el producto.

6.8. Otras cuestiones

Hay que señalar que tanto para proyectos grandes o pequeños, se podrá realizar la estimación del esfuerzo, del tiempo y los recursos mediante un método convencional, del tipo COCOMO, puntos funcionales u otras.

De este modo se obtendrá el tiempo estimado de duración del proyecto, el personal PSF (personal software full-time) requerido, y el costo del proyecto. Hay que señalar además, que se debe tener en cuenta la estimación de las LDC (líneas de código) para realizar el cálculo y esto presupone alguna experiencia en proyectos similares.

Recordando el principio de Gilb (1988) de los objetivos difusos: *"los proyectos que no tienen objetivos claros, no lograrán sus metas claramente"*, lo que significa que para poder llevar a cabo un proyecto o simplemente un programa, es necesario una buena planificación. Una buena planificación es una distribución eficiente de los recursos en el tiempo. Para ello se plantean metas y submetas, las que se deberán cumplir, de acuerdo a los diagramas de Gantt, por ejemplo.

Para una buena distribución de los recursos es necesario, partir de la definición clara de los requerimientos, y así continuar a lo largo del ciclo de vida del software. Esto conlleva a un conocimiento acabado de todas las actividades involucradas con previsión de los recursos necesarios en cada una de ellas. Se debe además considerar la existencia del camino crítico y su repercusión sobre la duración total de proyecto (o programa) y los posibles retardos, si los hubiere.

Desde el punto de vista metodológico, definir cada uno de los pasos a seguir, con las herramientas involucradas en cada uno de ellos, significa disponer de una manera clara y precisa de una *"hoja de ruta"* para continuar con el proyecto. En la sección 3, ya se ha visto acerca de la necesidad y la importancia de las metodologías, para los desarrollos de software, tal como lo señala Piattini (1996). La realización de un trabajo interdisciplinario hace que tal necesidad sea mucho más importante, ya que se deben realizar tareas en conjunto y en paralelo.

Otra cuestión, es la definición del ciclo de vida más apropiado para cada necesidad. La solución propuesta, es sólo una de tantas posibles, pero las tareas involucradas, a la hora de realizar la matriz de actividades, son las mismas.

7. Propuesta de Evaluación

7.1. Resumen

En el presente capítulo, se desea considerar una propuesta de evaluación de un software que ha sido desarrollado con la metodología expuesta en la sección anterior.

Se destacan aspectos del desarrollo del software educativo (sección 7.2). Se presentan y discuten los resultados de una evaluación incremental realizada a prototipo v1 (sección 7.2.1.), prototipo v2 (sección 7.2.2.) y prototipo vfinal (sección 7.2.3.), adecuando las sucesivas versiones a la evaluación formulada por los potenciales usuarios.

Se describe el desarrollo de la evaluación interna formulada por el equipo de desarrollo (sección 7.3) y la evaluación externa (sección 7.4) realizada por los docentes de los alumnos (potenciales usuarios).

Se presenta finalmente una propuesta de criterios tabulados para medir la calidad del software educativo (sección 7.5).

7.2. Desarrollo

Se tomó un caso: un programa que fuera solicitado por los docentes y alumnos de una carrera de postgrado universitaria no Informática.

Se solicitó para la asignatura "Computación" un software donde pudieran apreciar las partes de la computadora y en especial el funcionamiento interno de la misma, ya que se consideraba que una clase expositiva no era suficiente para las expectativas de los alumnos. Se consideró oportuno realizar un programa en Delphi 3, y para la evaluación del mismo desarrollar varios prototipos con incrementos en las funcionalidades. Para ello, se le solicitó a un programador realizar el programa de acuerdo a la metodología propuesta en la sección anterior, con la ayuda de un especialista en contenidos del área temática. De este modo, se partió de un mapa de conceptos de este tema, como los desarrollados por Novak (1988), en orden conceptual jerárquico o del tipo árbol.

Se presentó la propuesta de evaluación de cada prototipo a los alumnos de una carrera de postgrado en Tecnología, y luego fue seguida de un plan de incorporación de las modificaciones sugeridas.

Las preguntas consideraban aspectos de la interface de comunicación y de los contenidos desarrollados, debiendo ser valoradas con una escala de calificaciones entre 1 y 5 (siendo 5: excelente 4: muy bueno 3: bueno 2: regular 1: malo o 5: muy adecuado 4: bastante 3: poco 2: muy poco 1: nada, de acuerdo al tipo de pregunta), pudiéndose obtener un valor promedio de la calificación. Este valor permite obtener una puntuación de los aspectos tenidos en cuenta, para poder reformular o modificar aquellos que hayan tenido una puntuación menor que 2.5.

En todos los casos de evaluación había un espacio abierto para las sugerencias al cambio o reflexión acerca del programa o de la situación de interacción.

Se evaluaron los dos prototipos (v1 y v2) y el prototipo final (vfinal), el que también se evaluó en forma interna, externa y contextualizada.

7.2.1. Prototipo V1 (versión 1)

Para la evaluación del mismo, se tomó un grupo de 20 alumnos, recogiendo los resultados cuantitativos en la Tabla que se adjunta en el Anexo I.

Para el primer prototipo, se consideró pertinente presentar un pre-diseño de las pantallas, el menú desplegable y el árbol de contenidos. Las imágenes y los vídeos y el sonido no estaban cargados aún. Se pensó en un diseño de interface del tipo Windows estándar, pero el programa en sí mismo mostraba muy poco. Las preguntas realizadas se pueden ver en la Tabla 7.1.

1. ¿Considera adecuado el diseño general de la pantalla?
2. ¿Considera adecuado el uso de las
 - Ventanas
 - Botones
 - Colores
 - Tipos de letras?
3. ¿Considera que el programa es interactivo?
4. ¿Considera la interface como amigable?
5. ¿Le da buena información acerca del recorrido?
6. ¿Considera criteriosa la secuenciación de las pantallas?
7. ¿Es de fácil manejo?
8. ¿Considera que el uso de los íconos es correcto?
9. ¿Le resulta útil el uso de teclas rápidas?
10. ¿Ha despertado interés en usted?
 - Sugerencias de cambio Si- No

Tabla 7.1: Esquema de Evaluación del prototipo v1.

De acuerdo a los resultados, que se pueden resumir cualitativamente: el diseño de la pantalla pareció adecuado, como las ventanas y los botones, pero no así con los colores utilizados y los tipos de letras. La interface pareció fácil de navegar y la secuenciación de las pantallas en general fue considerada como muy buena y de fácil manejo.

No hubo problemas en cuanto a la interactividad y despertó el interés y curiosidad en saber como sería el segundo prototipo del programa, ya con más funcionalidades incorporadas. Otra cuestión a señalar, fue que muchos desconocían la existencia de las teclas rápidas, lo que realmente no les interesaba.

Hubo una pregunta no ponderada y abierta, donde los alumnos que lo creían conveniente debían realizar sugerencias de cambio antes de pasar a una etapa posterior del desarrollo.

Las sugerencias fueron básicamente :

- Usar un tamaño de letra más grande de modo que fuera bien legible en una notebook.
- Cambiar los colores para que hubiera más contraste.
- Cambiar el puntero del mouse cuando se activaba un objeto de la pantalla.

7.2.2. Prototipo V2 (versión 2)

De acuerdo a las preguntas ponderadas y las sugerencias realizadas se pasó al siguiente prototipo incremental con las mejoras pertinentes. Se cargó el glosario, las imágenes, algunos vídeos y la información acerca de cada una de las partes de la computadora. Ahora se podría tener una idea mucho más cercana a lo que sería el programa finalizado.

En este caso las preguntas acerca de la interface comunicación fueron pocas, se precisaba en aspectos relacionados a los contenidos y su pertinencia, se hacía mucho hincapié en la presentación de los mismos, la estructuración y la adecuación a las necesidades del grupo.

1. ¿Considera adecuada la selección de los contenidos?
2. ¿Consideraría adecuado el uso del programa terminado en otros niveles?
3. ¿Los cambios realizados fueron pertinentes?
4. ¿Quisiera que el programa fuera un tutorial?
5. ¿Le facilita la comprensión acerca del tema?
6. ¿Quisiera sonido en los vídeos?
 - Sugerencias de cambio Si- No

Tabla 7.2: Esquema de Evaluación del prototipo v2.

En el Anexo II se adjuntan los resultados obtenidos con las ponderaciones y las sugerencias de los alumnos. Cabe destacar que a la mayor parte de los alumnos no les interesó en demasía considerar la realización de un programa del tipo tutorial, por lo que se aprecia que no le interesaba al grupo reemplazar totalmente al docente en sus explicaciones, sino usar el software como material de apoyo al docente y orientado a la ejercitación. Respecto de las sugerencias, quizás la más relevante es que el programa finalizado les permitirá “*ver cosas que no hubieran imaginado*”.

7.2.3. Evaluación del prototipo versión final (vfinal)

Se confeccionó una planilla con preguntas pertinentes a diferentes criterios, tomando como base la utilidad, aspectos pedagógicos y didácticos y técnicos. Las preguntas que se observan en la Tabla 7.2, se deben ponderar como en los casos anteriores.

Utilidad²⁸	1.Facilidad de Uso
	2.Grado de adaptación a otros niveles de usuarios.
Pedagógicos y didácticos	3.Clareza de contenidos
	4.Nivel de actualización
	5.Interface de navegación
	6.Nivel de Motivación
	7.¿Es adecuado para la comprensión del tema?
	8.¿Es adecuado para el aprendizaje del tema?
Técnicos	9. ¿Hay documentación y ayudas?
	10.¿Son adecuados los recursos que necesita?

Tabla 7.3: Esquema de Evaluación del Producto Final

Por último, se solicitaron algunas sugerencias a los usuarios, mediante un ítem abierto, sean éstas para el uso del programa o para realizar algún cambio que se considere pertinente. Los resultados obtenidos se adjuntan en el Anexo III. En general, no se solicitaron cambios, y a partir de la evaluación del programa, se deriva que hubo aceptación y acuerdo respecto de los cambios producidos en las etapas anteriores.

7.3. Evaluación interna

El grupo que trabajó en el desarrollo del programa, estuvo de acuerdo con los cambios propuestos por los alumnos. Además consideró la pertinencia de las sugerencias. Es importante destacar que un programa de esta naturaleza debe ser actualizado permanentemente, lo que implica un gran tiempo insumido en actualizar los contenidos.

Además se consideró la propuesta de los alumnos, de usarlo paralelamente a las explicaciones del docente o de usarlo como apoyo a las clases de práctica y entrenamiento.

7.4. Evaluación externa

Se presentó el programa a docentes de una carrera de Ciencias no Informáticas, quienes lo consideraron como una herramienta interesante a la hora de tener que profundizar los conocimientos acerca del tema. Se les proporcionó una planilla similar a las anteriores, con preguntas cerradas y abiertas y se les proveyó del producto terminado.

Los resultados cuantitativos, se pueden ver en el Anexo IV.

Cualitativamente consideraron interesante la propuesta y remarcaron que a veces el grado de dificultad que tienen los usuarios no informáticos para entender cómo funciona la máquina, es muy grande. Otra de las consideraciones realizadas es que desde la escuela primaria se debería alfabetizar en informática, y que habría que pensarlo en lo sucesivo.

7.5. La calidad de los programas de software: un problema interdisciplinario

7.5.1. La propuesta: ¿qué medir en el software educativo?

Considerando que el producto software educativo, de acuerdo a las necesidades de aplicación y a los objetivos educativos perseguidos, requiere de características especiales, y además que éstas sean apropiadas en cuanto a calidad y pertinencia, se elaboró la Tabla 7.4, partiendo del criterio de usabilidad (en el sentido de amigabilidad) y analizando algunos subcriterios. A éstos se los puede calificar de acuerdo a tres niveles pro-

²⁸ En sentido práctico.

puestos: muy bueno, bueno o malo. Cada nivel tiene una puntuación. Al final de la evaluación el puntaje obtenido, estará entre los que se pueden observar en la Tabla 7.5, donde obviamente, se ve que un programas con una puntuación entre 21 y 30 puntos estará dentro de un nivel de calidad aceptable.

Se ha evaluado el programa de acuerdo a la Tabla 7.4, obteniéndose un promedio de 22.1 para 20 docentes evaluadores como se aprecia en el Anexo V.

7.5.2. La calidad desde la perspectiva pedagógica

Rivera Quijano (1999) considera que el estudiante es el centro del nuevo paradigma educativo denominado "Learner Centered Approach" (LCA) o "aprendizaje centrado en el estudiante", en muchos casos considerando a éste como un consumidor o cliente inclusive, por duro que parezca.

Como Tecnólogo Educativo, afirma que "la calidad de los proyectos tecnológicos se mide en términos del comportamiento observado al final de la formación". Señala que "el programa de formación debe denotar atributos mensurables y observables en el estudiante, de lo contrario, es imposible determinar si el programa logra los objetivos o no".

Considera que para la "Dimanche Scientifique" o método científico en educación, es esencial la humildad y el espíritu de equipo, siendo la humildad para cuestionar nuestras prácticas y nuestras experiencias. (Romiszowski, 1981), proposiciones a las que adhiero.

Criterio: Utilidad (amigabilidad)	Subcriterio	Calificación	Puntaje	Puntaje obtenido
Utilidad Externa	Velocidad a de aprendizaje (learnability)	Muy buena	3	
		Buena	2	
		Mala	1	
	Facilidad de uso (operabilidad)	Muy buena	3	
		Buena	2	
		Mala	1	
	Nivel de adicción	Muy buena	3	
		Buena	2	
		Mala	1	
Utilidad Interna	Nivel de legibilidad (Lecturability)	Muy buena	3	
		Buena	2	
		Mala	1	
	Grado de comprensión	Muy bueno	3	
		Bueno	2	
		Malo	1	
	Estructuras de los manuales	Muy buena	3	
		Buena	2	
		Mala	1	
	Uso de menús, gráficos e imágenes	Muy bueno	3	
		Bueno	2	
		Malo	1	
	Mensajes de errores e información	Muy bueno	3	
		Bueno	2	
		Malo	1	
	Ayudas on-line	Muy buena	3	
		Buena	2	
		Mala	1	
	Definición de adecuación de la interface	Muy buena	3	
		Buena	2	
		Mala	1	
	Puntaje obtenido			

Tabla 7.4: Criterios y subcriterios

Puntaje	Evaluación de la propuesta	Calidad
1-10	Mala	Inaceptable
11-20	Regular	Dudosa
21-30	Buena	Aceptable

Tabla 7.5: Tabla de puntuación.

7.5.3. Algo más acerca de la evaluación de los programas educativos

La evaluación de los programas educativos es un proceso que consiste en la determinación del grado de adecuación de dichos programas al contexto educativo. Cuando el programa llega al docente, se supone que ha sido analizado y evaluado tanto en sus aspectos pedagógicos y didácticos, como en los técnicos que hacen a la calidad del producto desarrollado según ciertas pautas de garantía de calidad.

Básicamente, se realizan las evaluaciones interna y externa del software, a fin de detectar los problemas que generarán cambios en el producto, lo antes posible, a fin de reducir costos y esfuerzos posteriores. Estas evaluaciones consideran las eventuales modificaciones sugeridas por el equipo de desarrollo y por los usuarios finales, teniéndose en cuenta a docentes y alumnos en el contexto de aprendizaje.

Cuando un producto del tipo comercial educativo, llega al docente, significa que ha superado las etapas de evaluaciones interna y externa. Además para obtener el grado de eficacia y de eficiencia del producto se deberá realizar una evaluación en el contexto de uso.

Es preciso definir ciertos “criterios” para seleccionar un programa como “de acuerdo a las necesidades del docente”, y se debe considerar el uso de los vocablos evaluación y valoración que en muchos de los trabajos consultados se usan indistintamente para determinar si un programa dado cumple con los objetivos tanto técnicos como pedagógicos y didácticos para lo que fue pensado

7.5.4. La integración de perspectivas

Rivera Quijano (1999) considera que el punto de partida de las propuestas debe ser el análisis de necesidades entre la situación ideal a la cual arribar y la actual, definida en términos claros y precisos, poniendo énfasis en el diseño pedagógico de las actividades de aprendizaje o sea, partiendo del proyecto mismo y no de las necesidades, para su elaboración.

Considera que el punto más importante de la planificación y el diseño de los productos educativos está en la definición correcta de los objetivos generales y particulares, como la determinación del tipo de los contenidos, la metodología a usar y la evaluación a hacer.

Mediante un diseño didáctico, o programación didáctica, rigurosa y detallada, se puede decidir, cuál es la mejor tecnología que se adapta a cada situación problemática particular. pero los modelos de creación pedagógica, son múltiples, y dependen de las características de cada institución.

“La evaluación es aquella cuyos logros pueden ser observados y medidos. En la medida que un nuevo proyecto permite la incorporación de conocimientos nuevos o de habilidades por parte de los estudiantes, y esos conocimientos pueden ser observados y medidos, esto da la seguridad que el proyecto responde a la necesidad. Este diseño pedagógico es el que permite saber si lo que se está haciendo es educativo o no. (Rivera Quijano, 1999)

Por último, Guitert (1999) afirma que “La existencia de prácticas innovadoras pueden resultar amplificadas por la utilización de tecnologías, pero no suelen ser provocadas por la tecnología misma, ya que no están tan relacionadas con su introducción en el aula, como con la concepción previa que el profesor tenga sobre su propia práctica pedagógica”.

Las evaluaciones realizadas dejan entrever varias cuestiones a saber:

- A los alumnos les interesa una interface estandarizada como las comerciales.
- Los productos donde se pueda navegar sin perderse brindan más confianza, la de saber dónde está parados en cada momento.
- La estructuración de los contenidos es un punto fundamental para saber hacia dónde van en los aprendizajes.
- Los alumnos se sienten motivados por participar de los cambios en las etapas de construcción del programa.
- La evaluación de los prototipos sucesivos, permite hacer cambios rápidos, de acuerdo a las sugerencias del usuario o potencial usuario.
- Es importante la buena documentación del programa a fin de hacer dichos cambios y las posteriores actualizaciones lo más eficientemente posible.
- Los usuarios no informáticos necesitan adicionalmente al software, un docente o guía, que les ayude a interpretar los procesos, que vistos en los vídeos e imágenes, y no quieren perder el diálogo fluido, que no tendrían con la máquina, aunque el programa fuera muy interactivo.
- Las evaluaciones son una guía para tener en cuenta, considerando un determinado conjunto de variables.
- Cuando se diseñan aplicaciones tecnológicas, la metodología de desarrollo es uno de los pilares que permitirán lograr acercarse a los objetivos propuestos.

Por otra parte, la calidad en los programas educativos, sigue siendo muy difícil de cuantificar, y las medidas indirectas son resultados parciales y puntuales, pero se pueden tomar como un “referente parcial”, a partir del cual elaborar la propuesta particular.

Es por ello, que considero de poca utilidad las largas listas de preguntas acerca de una propuesta en particular, para un docente particular, un alumno y un contexto determinado.

Lo que sí es interesante y práctico es la ponderación de algunos indicadores que permitan establecer una calidad “aceptable” a partir de criterios como el de utilidad, vista como “amigabilidad” y tomada desde un aspecto interno y externo al programa mismo.

Actualmente, hay una gran necesidad de optimizar los procesos, y particularmente el educativo viendo al alumno como el “cliente”, y desde este aspecto no sólo se debe lograr la satisfacción, sino un acercamiento a un perfil ideal que el estudiante deberá tener luego de un cierto “entrenamiento”.

La experiencia dice, que se puede realizar el programa educativo, con las herramientas más novedosas y mejores, pero, en realidad, el indicador más preciso se obtiene cuando el estudiante finaliza el ciclo o período y adquiere el conjunto de habilidades y conocimientos previstos, o cuando se incorpora al sistema productivo. Estos son finalmente los indicadores válidos que permitirán dar una idea de la calidad o bondad de la propuesta.

8. Evaluación contextualizada

8.1. Resumen

Se presenta la evaluación de un software educativo en un contexto similar a aquel para el cual fuera creado el programa. Los resultados de este tipo de evaluación se consideran como los más representativos ya que dan cuenta de las reacciones de los potenciales usuarios ante el programa y dan cuenta de la eficacia del producto. (Fainholc, 1994).

Para ello se tienen en cuenta las variables involucradas en el proceso de enseñanza y de aprendizaje tales como el docente y estilo docente, tipo de alumnos destinatarios, el tiempo y modo de uso del software, el currículum, entre otras.

Se formulan y se describen las etapas preparatorias (sección 8.2) y, posteriormente se presentan las experiencias realizadas a fin de establecer las diferencias en cuanto a logro de aprendizajes significativos entre un software desarrollado con una metodología extendida para cautelar los aspectos pedagógicos, partiendo de un paradigma clásico de la ingeniería de software tal como se describe en la sección 6 y uno de idéntica funcionalidad desarrollado con una metodología estándar (sección 8.3).

Para ello, se formaron dos grupos equilibrados²⁹ mediante la definición de pares homólogos: uno de control, llamado A y otro experimental ó B. Para definir grupos equilibrados, se partió de la aplicación del test de matrices progresivas de Raven³⁰ a los sujetos.

Ambos grupos, en conjunto recibieron la misma instrucción acerca de los aspectos teóricos, mediante clases expositivas, siendo el tema desarrollado: el funcionamiento interno de una computadora personal. Luego, al grupo de control "A" se le mostró aspectos inherentes a la lógica de funcionamiento mediante un software desarrollado con una metodología que no contempla los aspectos pedagógicos y al grupo experimental "B", mediante un software desarrollado con la metodología propuesta extendida. Los software señalados fueron desarrollados por equipos de implementación diferentes.

Una vez realizadas las experiencias, se verificó el rendimiento de los alumnos mediante la misma evaluación para los dos grupos, se aplicó un test estadístico de comparación para muestras pequeñas de Wilcoxon, obteniéndose las conclusiones que se enuncian.

8.2. Formulación de la tesis y etapas preparatorias

La tesis a demostrar experimentalmente es:

“El software educativo desarrollado con una metodología que contempla aspectos psicopedagógicos en el modelo de ciclo de vida del software permite un mejor aprendizaje de los conceptos que un software que ha sido desarrollado con una metodología que no los contempla”.

Para operacionalizar esta tesis se desarrollaron las siguientes etapas preparatorias:

ETAPA I: Se tomó un curso de Computación de un postgrado, no informático, donde a los alumnos se les debía instruir acerca del funcionamiento de una computadora personal. Para la experiencia, se tomó el tema específico de la estructura de una computadora, sus partes principales y el funcionamiento las unidades componentes.

²⁹ Se parte del supuesto que los pares homólogos requeridos para el apareamiento de Wilcoxon, tendrán una respuesta similar ante los nuevos aprendizajes. Se poría haber usado en lugar de la puntuación del Test de una prueba de evaluación diagnóstica diseñada específicamente.

³⁰ Raven J. C. (1970), *Test de Matrices Progresivas*. Escala General. Vol.3. Paidós.

ETAPA II: Mediante la aplicación del Test de Raven de Matrices Progresivas, se formaron pares de homólogos con igual puntuación en dicho test³¹, como se observa en la Tabla 8.1. Se formaron dos grupos: uno de control "A" y otro experimental "B".

ETAPA III: A ambos grupos en conjunto se les explicó el tema en sus aspectos teóricos, de modo tradicional, mediante una clase expositiva. Luego, el grupo A se ejercitó con un software desarrollado sin un metodología de que cautelara los aspectos pedagógicos y el grupo B utilizó un software desarrollado con la metodología propuesta. Las actividades desarrolladas por los grupos se resumen en la Tabla 8.2, resaltándose las diferencias.

Grupo A		Grupo B	
Alumno	Puntuación (%)	Alumno	Puntuación (%)
Paloma	9.83	Enrique	9.83
Javier	9.79	Carlos	9.79
Susana	9.72	Silvia	9.72
Carola	9.66	Cristina	9.66
Miguel	9.61	Guillermo	9.61
Mónica	9.58	Luis	9.58
Favio	9.5	Gustavo	9.5
Alejandro	9.33	Marcos	9.33
José	9	Ada	9

Tabla 8.1: Pares homólogos formados de acuerdo al Test de Raven

Actividades	Grupo A		Grupo B
Aspectos Teóricos Clase Tradicional Magistral Expositiva	Explicación del funcionamiento de una PC y de sus partes		
	Se usaron dibujos en el pizarrón		
	Se usaron transparencias ilustrativas.		
Aspectos Prácticos Ejercitación	Explicación del funcionamiento de una PC y de sus partes		
	Se usó un software multimedia del tema, desarrollado con metodología que no contemplaba los aspectos pedagógicos.		Se usó un programa confeccionado con la metodología propuesta extendida.
	Ambos programas tenían el mismo conjunto de imágenes y vídeos. Partían del mismo árbol de conceptos, pero diferían en el diseño y secuenciación.		
Las clases fueron dictadas por el mismo docente			
Los dos grupos fueron evaluados con el mismo conjuinto de preguntas.			

Tabla 8.2: Actividades de los dos grupos.

Con estas tres etapas concluidas se estuvo en condiciones de operacionalizar la tesis inicial reformulándola de la siguiente manera:

"Siendo los valores de las variables independientes los mismos para ambos grupos de alumnos, salvo la variable: 'programa utilizado para la ejercitación'; el grupo de alumnos que trabaje con el programa desarrollado con la metodología propuesta (grupo B), debe tener un mejor rendimiento que el grupo de alumnos que utilice el programa desarrollado con la metodología convencional (grupo A)"

8.3. Desarrollo de la experiencia y valuación estadística

Al finalizar la ejercitación ambos grupos fueron sometidos a la misma prueba, confeccionada específicamente, siendo los resultados obtenidos los que se presentan en la Tabla 8.3:

Grupo A		Grupo B	
Alumno	Nota	Alumno	Nota
Paloma	6	Enrique	10
Javier	7	Carlos	9
Susana	6	Silvia	10
Carola	8	Cristina	7

³¹ De la totalidad de los alumnos del curso se tomaron los nueve pares homólogos, que se presentan en la tabla 8.1 seleccionados con igual puntuación.

Miguel	7	Guillermo	8
Mónica	7	Luis	7
Favio	8	Gustavo	8
Alejandro	7	Marcos	8
José	8	Ada	10

Tabla 8.3: Tabla comparativa del rendimiento obtenido en la prueba.

Aplicado el test de Wilcoxon (Ledesma, 1980), a los grupos A y B, habiendo tomado al A como grupo de control, *se espera que el grupo B tenga un mejor rendimiento. Como la única diferencia, sería el software con el cual se desarrolló la ejercitación, el hecho de estar trabajando con pares homólogos, permitiría concluir que de existir una diferencia esta se deba al software, en particular a la metodología aplicada para su desarrollo.*

Grupo A		Grupo B		D _{A-B}
Alumno	Nota	Alumno	Nota	
Paloma	6	Enrique	10	-4
Javier	7	Carlos	9	-2
Susana	6	Silvia	10	-4
Carola	8	Cristina	7	1
Miguel	7	Guillermo	8	-1
Mónica	7	Luis	7	0
Favio	8	Gustavo	8	0
Alejandro	7	Marcos	8	-1
José	8	Ada	10	-2

Tabla 8.4: cálculo de las diferencias D_{A-B}.

El primer paso del test de Wilcoxon (Ledesma, 1980), consiste en realizar la diferencia de calificaciones entre ambos grupos: En la Tabla 8.4 se puede observar en la última columna la diferencia D_{A-B}. Como indica el método de Wilcoxon se procede al ordenamiento por valor absoluto de las diferencias como se ve en la Tabla 8.5.

1
-1
-1
-2
-2
-4
-4

Tabla 8.5: Ordenamiento de las diferencias.

Luego, se le asignan los números de orden a cada valor y en el caso de valores con valor absoluto igual se promedian las posiciones, tal como se observa en la Tabla 8.6.

1	1	2
-1	2	2
-1	3	2
-2	4	4.5
-2	5	4.5
-4	6	6.5
-4	7	6.5

Tabla 8.6: Obtención de los números de orden.

Finalmente, se suman los números de orden de las diferencias negativas tal como se aprecia en la Tabla 8.7.

-1	2
-1	2
-2	4.5
-2	4.5
-4	6.5

-4	6.5
SUMA	26

Tabla 8.7: Suma de las diferencias negativas.

Según la Tabla 10 (ver Tabla 8.9) del apéndice del libro de Ledesma de Estadística Médica (1980) y el Manual de la Universidad de Málaga de Bioestadística (1999) para los niveles significación de 5% donde $2\alpha \leq 0,05$ (siendo α la probabilidad de error de primer orden) y para un número de muestras $n = 7$ (en este caso el número de pares homólogos cuyas diferencias D_{A-B} sean diferentes a cero) se puede observar que:

n: Número de pares	$2\alpha \leq 0,05$
6	0-21
7	3-26

Tabla 8.9: de Wilcoxon para muestras continuas de pares homólogos

la suma de los números de orden de las seis observaciones negativas cae fuera de los límites tabulados. Y, como si: “o bien coincide con uno de los límites del intervalo de significatividad o está fuera de dichos límites, la diferencia es significativa”, (descartándose entonces la hipótesis nula de contraste), se puede decir que **la diferencia** entre el método aplicado al grupo B y al grupo A **es significativa a favor de B**, con lo que experimentalmente se confirma la tesis:

"Los alumnos que trabajaron con el programa desarrollado con la metodología extendida (grupo B), tienen un mejor rendimiento que los alumnos que utilizaron el programa desarrollado con la metodología que no considera los aspectos pedagógicos en su diseño (grupo A)".

Desde esta perspectiva queda demostrada experimentalmente la tesis central:

"El software educativo desarrollado con una metodología que contempla aspectos psicopedagógicos en el modelo de ciclo de vida permite un mejor aprendizaje de los conceptos que un software que ha sido desarrollado con una metodología que no los contempla".

Los resultados de la experiencia apoyan la metodología propuesta en esta tesis, con independencia de los resultados satisfactorios creemos que la misma está sujeta a replicaciones posteriores, debido a que fue restringida.

Desde la perspectiva tecnológica, se ha insistido en el hecho, de que el nivel de estructuración y el grado de adecuación de los contenidos es función directa del rendimiento esperado por los alumnos, la comprensión del material educativo depende fundamentalmente de la organización y estructuración de los contenidos del mismo. (Pozo Municio, 1998; Fainholc, 1998)

Esta coherencia interna, se logra mediante un desarrollo metódico, que permite realizar las conexiones lógicas y conceptuales entre los elementos. Esta información organizada, dice Pozo Municio (1998), se parece a un árbol de conocimientos, en el que se pueden establecer relaciones diversas entre ellos y recorrer diferentes rutas para recuperar el conocimiento y mediante la comprensión de la misma se podrá “reconstruir” o “traer a la luz el material” a las palabras propias del aprendizaje.

Por este motivo, la producción de programas educativos, no es una tarea sencilla, sino que se debe aplicar la secuencia metodológica que provee básicamente la ingeniería de software, sea en la elección de un ciclo de vida para el desarrollo de los productos, por un lado o en el uso de las técnicas, y herramientas por el otro, pero deben estar articuladas dinámicamente con una teoría acerca de los aprendizajes y de los niveles de representación de los alumnos cuando estos necesiten interpretar los fenómenos físicos.

Desde la perspectiva pedagógica, a fin de mejorar las prácticas educativas, los docentes investigadores (Gutiérrez, 1997) han basado sus estudios en las teorías del aprendizaje, pero en muchos casos las teorías solamente, si bien dan una respuesta válida para llegar a una eventual solución, no conducen al cierre definitivo del problema.

Es por ello, que se ha tomado como eje central para el análisis de la experiencia, algunos de los campos disciplinares³² de la “Ciencia Cognitiva”³³.

³² Los campos disciplinares que confluyen y contribuyen son: la psicología cognitiva, la neurociencia, la lingüística, la ciencia de la computación y la filosofía.

³³ Gardner (1988) dice: “defino la ciencia cognitiva como un empeño contemporáneo de base empírica por responder a interrogantes epistemológicos de antigua data, en particular los vinculados a la naturaleza del conocimiento, sus elementos componentes, sus fuentes, evolución y difusión. Aunque a veces la expresión ciencia cognitiva se hace ex-

A fin de acotar el problema se ha tomado la teoría de Jonhson-Laird (1998) dentro de la Psicología Cognitiva, para fundamentar los resultados experimentales, desde la perspectiva de las representaciones mentales.

Para los alumnos es importante entender los “fenómenos”, saber qué los causa y sus consecuencias, cómo replicarlos y darles fin; esto significa tener un “modelo de funcionamiento” o “de trabajo”.

Para intentar explicar los altos rendimientos, en general de los alumnos de ambos grupos, habría que considerar que “han podido formar modelos mentales”, sea proposicionales o por analogía³⁴, y sus imágenes mentales fueron las que les permitieron, interpretar los procesos o fenómenos. Los alumnos de estos grupos, no sólo entienden claramente los conceptos, y los pueden transferir de y hacia otras asignaturas, aunque algunos sólo utilicen las fórmulas y/o las definiciones y otros adapten los modelos del material de estudio. El rendimiento del grupo que “forma imágenes mentales”, es superior a aquel que no lo hace.

Por otra parte, Perkins (1995) habla acerca de la conexión importante que existe entre la *pedagogía de la comprensión* (o el arte de enseñar a comprender) y las *imágenes mentales*, por lo que se puede decir que la relación es bilateral.

Esta relación recíproca existente es la que puede ayudar al alumno a adquirir y elaborar imágenes mentales, con lo cual desarrolla su capacidad de comprensión y al exigirles actividades de comprensión (como por ejemplo: predecir, explicar, resolver, ejemplificar, generalizar) se hará que construyan imágenes mentales, para lo que afirma Perkins que: “se alimentan unas a otras como si fueran el Yin y el Yan de la comprensión”.

La detección de las “representaciones mentales” de los alumnos, no es el tema central de este estudio, pero la construcción de “modelos” implica un aprendizaje significativo de conceptos. Esta construcción se vería ampliamente favorecida con el uso de materiales de estudio multimediales, especialmente con vídeos y con una secuenciación adecuada y lógica, como se deriva de los resultados obtenidos por el grupo experimental.

tensiva a todas las formas del conocimiento, (...) yo la aplicaré principalmente a los esfuerzos por explicar el conocimiento humano”.

³⁴ Jonhson-Laird (1998) plantea que existen al menos tres formas en la que se puede codificar mentalmente para representar información; las representaciones proposicionales, los modelos mentales y las imágenes, sean estas auditivas, visuales o táctiles.

Conclusiones

Como conclusiones finales del trabajo realizado, se puede puntualizar, que el software educativo, es uno de los pilares en que se sostiene, del sistema educativo a distancia y, como material de aprendizaje, su comprensión depende fundamentalmente de la organización y estructuración de los contenidos del mismo.

Esta coherencia interna, se logra mediante un desarrollo metódico, que permite realizar las conexiones lógicas y conceptuales entre los elementos. Esta información organizada, dice Pozo Muncio (1998), se parece a un árbol de conocimientos, en el que se pueden establecer relaciones diversas entre ellos y recorrer diferentes rutas para recuperar el conocimiento y mediante la comprensión de la misma se podrá *"reconstruir"* o *"traducir el material"* a las palabras propias del aprendiz.

Recordando a Freire (1997), en *"Pedagogía de la autonomía"*, quien dice que *"No temo decir que carece de validez la enseñanza que no resulta en un aprendizaje, en que el aprendiz no se volvió capaz de recrear o de rehacer lo enseñado, en que lo enseñado que no fue aprehendido no pudo ser realmente aprendido por el aprendiz"*, se puede pensar en el desarrollo de los materiales para formar sujetos autónomos, más que autodidactas.

1. Aportes del presente trabajo

1. Se ha presentado un estudio crítico del estado de los desarrollos de los programas de software educativos de modo diacrónico, en paralelo con las diferentes teorías y líneas educativas, desde su aparición hasta la actualidad.
2. Se ha visto, que la situación actual se complejiza, en tanto existe una gran cantidad de lenguajes de programación que posibilitan diferentes alternativas de desarrollo, como así también, los avances en cuanto a la tecnología informática, que permiten utilizar recursos impensados una década atrás.
3. Se han detectado las problemáticas concernientes al diseño, desarrollo y evaluación de los programas educativos y se ha propuesto una de las posibles metodologías, como solución a dichas necesidades.
4. Se ha tomado un software desarrollado con las características propuestas y se mostrado experimentalmente, que los alumnos que lo usaron, obtiene un rendimiento notable respecto de otros productos que existen en el mercado.
5. Se ha tenido en cuenta las opiniones del grupo de alumnos "evaluadores" del programa, que fueron los que probaron los prototipos e hicieron las sugerencias que como se ve en las Tablas que se adjuntan en los Anexos I a IV.
6. Durante el proceso de investigación se detectaron líneas de investigación derivadas del problema central tratado que serán enumeradas en el epígrafe siguiente.

2. Líneas de trabajo futuras.

A partir de las problemáticas detectadas y de los resultados obtenidos se pueden derivar las siguientes líneas de trabajo:

1. Desarrollo de otras metodologías basadas en diferentes combinaciones de ciclos de vida del software y de teorías de aprendizaje, estableciendo así comparaciones con la solución propuesta en este trabajo y con ponderación de resultados.
2. Es en este punto, en el que se ha pensado, partiendo de la necesidad detectada en alumnos y docentes, en desarrollar un buscador de Internet para este sector particular, que es el EGB, y por cierto, en castellano y para necesidades particulares.
3. Otra de las posibles líneas de trabajo, que se deriva al respecto de los programas educativos, es la adaptación de los programas de mercado y/o desarrollo de nuevos programas a las necesidades específicas de algunos grupos de estudiantes, como los hipoacúsicos, en sus distintos niveles, ya que se ha detectado, que el software desarrollado para estos grupos con necesidades especiales, es escaso y caro.
4. Un línea de trabajo de fondo en los desarrollos con base informática sería, el nivel de infoalbetización de los docentes que desean aplicar y/o diseñar programas didácticos, como así también el caso inverso de los programadores que desean hacerlos.
5. Por último, se ha dejado un tema central, de las investigaciones didácticas actuales, respecto de la influencia del estilo docente en los procesos educativos y, en este caso en particular en que deben aplicar la herramienta informática y los productos lógicos.

Anexos

Anexo I: Planilla de evaluación de la interface de comunicación. Prototipo v1

Calificación de 1 a 5 (5: excelente 4: muy bueno 3: bueno 2: regular 1: malo o 5: muy adecuado 4: bastante 3: poco 2: muy poco 1: nada)																					
Número de orden	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Prom
1. Considera adecuado el diseño general de la pantalla?	4	4	3	3	3	3	3	3	3	3	4	4	3	4	4	3	3	4	4	3	3.4
2. Considera adecuado el uso de las	Ventanas	5	4	4	4	4	4	4	4	4	4	4	4	4	4	3	4	3	4	4	3.95
	Botones	4	4	3	4	4	4	3	4					3	3	4	3	4	4	4	2.33
	Colores	4	1	2	2	2	2	2	2	3	3	2	2	3	3	2	2	2	2	2	2.25
	Tipos de letras?	3	4	2	2	4	4	3	4	3	2	3	3	3	3	3	2	2	2	2	2.65
3. Considera que el programa es interactivo?	5	4	4	4	2	3	2	3	4	3	4	4	4	3	3	3	4	4	4	4	3.55
4. Considera la interface como amigable?	4	4	4	3	4	4	4	4	4	4	4	4	4	4	4	3	4	3	3	3	3.6
5. Le da buena información acerca del recorrido?	5	5	4	5	4	4	4	5	4	4	4	4	4	4	4	4	4	5	5	5	4.15
6. Considera criteriosa la secuenciación de las pantallas?	4	4	4	3	4	4	4	4	5	4	5	5	5	4	4	4	3	3	3	4	4.0
7. Es de fácil manejo?	4	5	5	4	5	5	5	4	5	4	5	5	4	5	4	5	5	5	5	5	4.7
8. Considera que el uso de los íconos es correcto?	4	5	4	5	5	5	5	4	4	3	4	5	4	4	4	4	4	4	4	4	4.25
9. Le resulta útil el uso de teclas rápidas?	3	3	3	5	-	-	-	4	-	-	-	-	-	-	-	-	3	-	4	4	3.62
10. Ha despertado interés en usted?	4	4	3	5	3	4	3	4	4	4	4	4	4	5	5	4	4	3	4	4	3.95
Sugerencias de cambio Si- No	N	S	S	S	S	S	S	N			S	S		S	S	S	S	S	S	S	-

Número de orden	Sugerencias de cambio
1	Cambiar los colores en pantalla para que resulte más atractivo
2	Para evaluar mejor tendría que estar más completo, sugiero mayor colorido y que el indicador sea distinto. Es decir no flecha sino mano.
3	Cuando se posiciona en algo que se expande que aparezca otro apuntador como la mano, y que a la vez haya cambio de relieve. Ponerle sonido al momento de activar algo. Más vistoso, más atractivo.
4	Cambio de colores - más contraste
5	Interconectar los elementos por medio de dibujos que representen cables en la pantalla de presentación. Tipo diagrama de flujo
6	Cambiar la flecha de indicación por la manito
10	Tamaño de letra más grande para usar en la notebook.
15	Cambiar los colores de pantalla para que resulte más contraste.
16	Hace falta un icono de retorno en la pantalla. Tamaño y tipo de letra
18	Sería bueno agregar algún tapiz de fondo por que parece que los elementos en la pantalla están flotando?
	Las letras de los botones son muy débiles, les falta fuerza.
19	Colocar el icono de “volver” fijo en la pantalla para retornar a ventanas anteriores.
20	Mejor color de pantalla, letra más gruesa, crear subventanas u opciones de algunos temas

Anexo II: Evaluación de contenidos y pertinencia. Prototipo v2

Calificación de 1 a 5 (5: excelente 4: muy bueno 3: bueno 2: regular 1: malo o 5: muy adecuado 4: bastante 3: poco 2: muy poco 1: nada)																					
Número de orden	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Prom
1.¿Considera adecuada la selección de los contenidos?	4	5	3	5	4	5	4	3	5	3	4	4	4	5	4	5	3	4	4	4	4.1
2.¿Consideraría adecuado el uso del programa terminado en otros niveles?	4		5	5	5	5	4	4	4	4	4	5	5	4	4	4	4	4	5	5	4.42
3.¿Los cambios desde realizados fueron pertinentes?	4	4	4	4	4	4	4	5	5	5	5	5	4	4	4	3	3	4	4	4	4.15
4.¿Quisiera que el programa fuera un tutorial?	2	3	4	3	3	3	2	2	2	3	3	3	3	3	3	3	3	3	4	3	2.9
5. ¿Le facilita la comprensión acerca del tema?	4	4	4	4	5	5	4	4	5	5	4	4	4	4	4	4	4	5	5	5	4.35
6. ¿Quisiera sonido en los vídeos?	3	3	3	4	4	3	3	3	3	4	4	4	3	3	3	3	3	3	3	3	3.75
Sugerencias de cambio Si- No	-	S	-	S	S	-	S	-	S	-	S	-	S	-	-	-	-	-	-	-	-

Número de orden	Sugerencias de cambio
1	Poner las direcciones web y la bibliografía en las pantallas explicativas.
2	Ahora puedo evaluar mejor la capacidad del programa y me lo imagino terminado
3	Sugiero que no se le ponga audio, ya que me gusta tener al docente que me explique lo que pasa en el video y así puedo hacerle preguntas. Sólo lo usaría para los eventos.
6	Considero que los cambios estuvieron bien y que respeten en los programas el formato de Windows
10	Los colores siguen siendo muy pálidos
16	El problema es que hay que actualizarlo constantemente.
19	Me parece bueno el programa porque me permite ver cosas que no me hubiera imaginado y que desconocía

Anexo III: Evaluación Prototipo versión final (vfinal)

Calificación de 1 a 5 (5: excelente 4: muy bueno/a 3: bueno/a 2: regular 1: malo/a o 5: muy adecuado 4: bastante 3: poco 2: muy poco 1: nada)																							
Número de orden		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Prom	
Aspecto	Utilidad	1. Facilidad de Uso	5	5	5	5	4	5	4	5	5	5	5	4	4	4	4	4	4	4	4	4.75	
		2. Grado de adaptación a otros niveles de usuarios.	4	5	4	5	4	5	5	5	5	5	4	5	5	4	4	4	5	5	5	5	4.65
Pedagógicos y didácticos	3. Claridad de contenidos	4	4	4	3	3	3	4	4	3	4	3	3	3	3	3	4	4	4	4	4	3.55	
	4. Nivel de actualización	5	5	5	5	5	5	4	4	5	5	5	4	4	4	4	4	4	4	4	4	4.45	
	5. Interface de navegación	4	4	4	4	3	3	3	3	4	4	3	3	3	3	3	3	3	3	3	3	3.3	
	6. Nivel de Motivación	4	4	4	3	3	3	3	4	4	4	4	4	4	4	4	4	3	3	3	3	3.8	
	7. ¿Es adecuado para la comprensión del tema?	3	5	5	5	4	3	3	3	4	4	5	5	5	3	4	4	3	3	3	3	3.85	
	8. ¿Es adecuado para el aprendizaje del tema?	3	5	5	5	3	3	3	4	4	4	4	3	5	5	5	4	4	4	3	3	3	3.9
	9. Documentación y ayudas	4	4	4	4	5	5	5	4	4	4	3	5	5	5	4	4	4	3	3	3	4.05	
	10. ¿Son adecuados los recursos que necesita?	5	5	5	5	5	3	3	4	5	5	5	5	5	5	5	5	3	3	3	3	4	4.1
Sugerencias																							

Anexo IV: Evaluación Externa de Producto Final

Calificación de 1 a 5 (5: excelente 4: muy bueno 3: bueno 2: regular 1: malo o 5: muy adecuado 4: bastante 3: poco 2: muy poco 1: nada)																							
Número de orden		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Prom	
1. ¿Considera adecuado el diseño general de la pantalla?		4	4	4	4	4	3	4	4	5	5	5	5	5	5	5	5	5	5	5	5	4.55	
2. ¿Considera adecuado el uso de	Ventanas	3	3	4	4	4	4	4	4	5	5	5	5	5	5	4	5	4	5	4	5	4	4.3
	Botones	4	4	4	4	4	4	4	5	5	5	5	5	5	5	4	4	4	4	4	4	4	4.3
	Colores	3	3	3	3	4	4	4	4	4	3	3	3	3	4	4	4	4	4	4	4	4	3.6
	Tipos de letras?	4	4	4	4	4	3	3	3	4	3	3	4	4	4	4	4	4	4	4	4	4	3.75
3. ¿Considera que el programa es interactivo?		4	4	4	4	4	4	4	4	5	5	4	4	4	4	4	4	4	4	4	4	4	4.1
4. ¿Considera la interface como amigable?		4	4	4	4	4	4	5	5	5	4	4	5	5	4	4	4	4	4	4	4	4	4.25
5. ¿Le da buena información acerca del recorrido?		5	5	5	5	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5	5	5	4.8
6. ¿Considera criteriosa la secuenciación de las pantallas?		4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
7. ¿Es de fácil manejo?		5	5	5	5	4	4	5	5	5	4	4	4	4	4	5	5	5	5	5	5	5	4.25
8. ¿Considera que el uso de los íconos es correcto?		5	5	5	5	5	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	4.7
9. ¿Le resulta útil el uso de teclas rápidas?		3	3	3	3	2			2	2	2				2	2		2	2	2		2.3	
10. ¿Considera adecuada la selección de los contenidos?		4	4	4	4	4	4	5	5	5	4	4	4	4	4	5	4	5	4	5	4	4	4.05
11. ¿Consideraría adecuado el uso del programa terminado en otros niveles?		4	4	4	4	4	4	4	4	4	4	4	5	5	4	5	4	5	5	4	4	4	4
12. ¿Quisiera que el programa fuera un tutorial?		4	4	4	-	-	-	3	3	3	-	-	-	-	-	3	3	3	-	-	-	3.33	
13. ¿Le facilita la comprensión acerca del tema?		4	4	4	4	3	3	3	3	5	5	5	5	5	5	5	5	5	5	5	5	5	4.15
14. ¿Quisiera sonido en los vídeos?		5	2	3	4	4	3	3	2	2	2	2	3	3	-	-	3	-	-	3	3	2.37	
15. ¿Ha despertado interés en usted?		5	5	5	5	5	4	4	4	4	3	3	3	-	5	5	5	-	5	-	5	4.41	
Sugerencias de cambio Si- No		S	-	S	-	-	-	S	-	-	-	S	-	-	S	-	-	S	-	-	S	-	

Número de orden	Sugerencias de Cambio	Número de orden	Sugerencias de Cambio
1	Los colores más fuertes.	12	No me interesan las teclas rápidas.
5	Las letras más grandes	14	Habría que poner más ventanas.
10	Habría que considerar usarlo en otros cursos	15	Me vino muy bien por que yo no sabía nada
		16	Quizás sería bueno, proyectarlo mientras el docente explica

Anexo V: Aplicación de Criterios y Subcriterios de Calidad (ver Tabla 7.5)

Número de orden	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Prom.
Puntaje obtenido	20	21	23	20	23	20	24	25	20	21	23	22	22	22	23	21	17	24	26	25	22.1

Referencias Bibliográficas

- AACE. Society for Information Technology and Teacher Education. www.AACE.org
- Aenor (1992): *Normas para la gestión y el aseguramiento de la calidad*. Madrid.
- Akahori Kenji (1985): *Evaluation of Educational Computers Software in Japan (I y II): Methods and results*. PLET, vol. 25, 46-66.
- Alcalde E., García M. y Peñuelas S. (1988): *Informática Básica*. Mc Graw Hill.
- Alessi S. M. y Trollip S. R. (1985): *Computer-based instruction. Methods and Development*. Prentice Hall. Nueva Jersey.
- Alonso, C. M. (1992): *Estilos de aprendizaje y tecnologías de la información*. Conferencia europea sobre tecnología de la Información. Barcelona. 3-6 noviembre.
- Amler (1994): citado por Piattini M.(1996): *Análisis y Diseño Detallado de Aplicaciones Informáticas de Gestión*. Rama. Madrid.
- Ander Egg, Ezequiel (1986): *Acerca del pensar científico*. Humanitas. Buenos Aires.
- Aspillaga M. (1991): *Para un diseño efectivo de presentación de la información en la computadora*, Revista de Tecnología Educativa, XI, 4, 307-323.
- Ausubel D., Novak J. y Hanesian H.(1978): *Psicología educativa. Un punto de vista cognitivo*. Trillas. Ediciones 1978, 1997.
- Basili V. y Rombach H. (1988): *The TAME project: towards improvement -orientated software enviroments*. IEEE Transactions on Software Engineering, vol. 14, número 6, págs. 758-773.
- Baumgartner P. y Payr S. (1996): *Learning as action: A social science approach to the evaluation of interactive media*. Universities of Innsbruck and Klagenfurt. Educational Multimedia and Hipermedia, AACE, Charlottesville, V.A. www.webcom.com/journal/baumgart.html
- Bender, Richard, (1996): *Proposed software evaluation and test KPA*. Bender and Associated Inc. Position Papers, White Paper, April.
- Benett (1996): *Computers as tutors: Solving a crisis in education*. www.cris.com/~faben1/
- Blease (1986): *Evaluating Educational Software*. Londres. Croom Helm, citado en Squires y Mc Dougall (1994).
- Bloom B, et al. (1956): *Taxonomy of educational objectives. The classification of educational goals*. David McKay. N. Y.
- Boehm B. (1978): *Characteristics of Software Quality*. Nueva York. North Holland.
- Boehm B. (1981): *Software Engineering Economics*, Englewood Cliffs, Nueva Jersey.
- Boehm B. (1988): *A spiral model of software development and enhancement*. Computer 1988 IEEE págs. 61-72.
- Bolívar A. (1995): *La evaluación de valores y actitudes*. Madrid, Anaya, citado en Castillo Segurado (1997)
- Booch G. (1991): *Object Oriented Design with applications*. Redwood City. Benjamin Cummings Publishing
- Bork A. (1986): *El ordenador en la enseñanza. Análisis y perspectivas de futuro*, Barcelona, Gustavo Gili.
- Bruner J. (1988): *Desarrollo cognitivo y educación*. Morata. Madrid.
- Bruner J. (1991): *Actos de significado. Más allá de la revolución cognitiva*. Madrid. Alianza.
- Bunes y otros (1993): *Los valores del LOGSE*. Un análisis de documentos a través de la metodología de Hall-Tonna, Bilbao. ICE, Universidad de Deusto, citado en Castillo Segurado (1997)
- Burnstein I. et al. (1996): *Developing a Testing Maturity Model. Part I*, Crosstalk, STSC, Hill Air Force Base, Utah, Agosto.
- Cabero J. (1993): citado en Sancho J. (coord.) (1994): *Para una Tecnología Educativa*. Editorial Horsori. Barcelona. España. pág. 255.
- Cabero Almenara J. (1992): *Diseño de Software Informático*. Universidad de Sevilla. Bordón, 44,4 383-391 ISSN: 0210-5934
- Caftori N. y Paprzycki M. (1997): *The design, evaluation and usage of educational software* en Price J. D., Rosa K, Mc Neil S. Y Willis J. Editores (1997): *Technology and Teacher Education Annual*. Association for el Advancement of Computing Education, Charlottesville, V.A. Campos F. et al. (1996): *Dez etapas o desenvolvimiento de software educacional do tipo hipermedia*: Memorias del Tercer Congreso Iberoamericano de Informática Educativa de Barranquilla, Colombia citado en Sanchez J. y Alonso O.
- Castillo Segurado (1997): *Un ejemplo de evaluación de software educativo multimedia*. Edutec 97. Comunicaciones: Formación y recursos.
- Castorina J. A. (1989): *La posición del objeto en el desarrollo del conocimiento*, en Castorina et al. *Problemas de la psicología genética*. Buenos Aires. Miño y Dávila Eds.
- Cedipro, (1998): *Actividades para el logro de la Comprensión*. Material de trabajo.
- Clarke P. y Peté M. (1996): *The KwaZulu concept burger: A hypertext concept map of educational software evaluation*. AACE Site '97. Consultado el 20/06/99 a las 22 horas. www.AACE.org/pubs/elec_pub/th_char.htm

- Coburn P., Kelman P. et al. (1985): *Practical guide to computers in Education*, Reading Massachusetts. Addison Wesley, citado en Squires y Mc Dougall (1994).
- Coll César (1994): *Psicología y Curriculum*. Paidós.
- Crosby P. (1979): *La calidad no cuesta*. Mc Graw Hill. México
- Cruz Feliú, Jaime (1986): *Teorías del Aprendizaje y Tecnología de la Enseñanza*, Trillas.
- Del Moral M. E. (1998): citada por Marquès (1998b): *Programas didácticos: diseño y evaluación*. Universidad Autónoma de Barcelona. Consultado en octubre de 1998. www.doe.d5.ub.es/te
- DeMarco T. (1979): *Structured analysis and systems specifications*. Prentice Hall.
- Deming W. E. (1982): *Out of the crisis*. Cambridge University Press.
- Deterline W. A. (1969): *Introducción a la Enseñanza Programada*, Buenos Aires Troquel.
- Doll C. A. (1987): *Evaluating Educational software*. Chicago-London: American Library Association.
- Dorado Carlos (1998): Citado por Marquès (1998) y comunicación vía e-mail del 26 mayo de 1999, cdorado@pie.xtec.es
- EDUCOM (1989): *Software snapshots: Where are you in the picture?*, Washington D. C. EDUCOM, citado en Squires y Mc Dougall (1994).
- Eisner E. (1992): *Procesos cognitivos y curriculum*. Ed. Martínez Roca. Barcelona.
- England E. (1988): *Case study: iterative screen design-errors as the basic of learning*. Educational & Training Technology International, 26,2, 149-155, citado en Cabero Almenara (1992).
- Fainholc Beatriz: (1994): *La tecnología educativa propia y apropiada*. Humanitas. Bs. As.
- Feire Paulo (1997): *Pedagogía de la Autonomía*. Ediciones.Siglo XXI
- Fenton N. (1991): *Software Metrics. A rigorous and practical approach*. PWS Publishing Company. Boston.
- Fernández Pérez M. (1995): *Las tareas de la Profesión de Enseñar*. Siglo veintiuno Editores.
- Flagg B. (1990): *Formative evaluation for educational technologies*. Hillsdale, New Jersey: Lawrence Erlbaum Associates, Publishers.
- Flavell J. H. (1993): *El desarrollo cognitivo*. Madrid. Ed. Visor.
- Florín F. (1990): *Information Landscape*, Ambroa S y Hooper K.: *Learning with interactive multimedia*. Nueva York: Apple Computer, Inc. And Microsoft Press.
- Gagné R. M. (1970): *The conditions of Learning*. Nueva York, Holt Rinehart & Winston.
- Gallego D, Alonso C.: (1997): *Los Sistemas Multimediales desde una Perspectiva Pedagógica en Multimedia*, UNED, Madrid.
- Gallego D. y Alonso C. (1997): *Multimedia*. UNED. España.
- Gallego M. J.: (1997): *La tecnología Educativa en acción*. Granada, Force. Universidad de Granada, p. 191-208.
- Galvis A. (1996): *Software educativo multimedia aspectos críticos no seu ciclo de vida*. Revista Brasileira de Informática no Educação. Sociedad Brasileira de Computação. Consultado el 22/06/99 a las 23 horas. www.janus.ufse.br:1085/revista/nr1/galvis_p.htm
- Gane C. y Sarsons T. (1977): *Análisis Estructurado de Sistemas*. Quinta reimpresión. El Ateneo.
- García López M. y Ruiz del Olmo F. (1997): *Nuevas Tecnologías, "El relato hipermedia"*. Universidad de Málaga, 1997.
- Gardner H. (1995): *La mente no escolarizada*. Paidós
- Gardner H. (1987-8): *La nueva ciencia de la mente: Historia de la psicología cognitiva*. Barcelona. Paidós.
- Gardner H. (1993): *Las Inteligencias Múltiples: La teoría en la práctica*. Barcelona. Paidós.
- Gardner H. (1997): *Inteligencias Múltiples*. Paidós
- Garrido M. (1991): *Diseño y creación de software educativo*, Infodidac, 14-15, 31-34.
- Garrido P. y Geisser C. (1996): *A methodology for software evaluation*.
- Gayan J. y Segarra D. (1985): *Propuesta de evaluación de programas de enseñanza asistida por ordenador*, en Pfeiffer, A. y Galván, J. (ed): *Informática y escuela*, Madrid, FUNDESCO, 379-382.
- Gilb T.(1988): *Principles of software project management*. Nueva York. Addison Wesley.
- Goldberg Mark F. (1991): *Portrait de Seymour Papert*, vol. 48. Nº 7 (1990-1991): Pág. 68-70, citado en *Seymour Papert 1965-1996* por Paula Holder. (1996)
- Goldberg Mark F. (1993): *Wishful Thinking*. Object Magazine, vol. 3, número 1, mayo-junio, págs. 87-88, citado en Piattini (1996)
- Grady R. y Caswell (1987): *Software metrics establishing a company wide program*. Nueva Jersey. Prentice Hall.
- Gros Begoña (1997): *Diseños y programas educativos*. Barcelona Ariel, citado en Marquès Graells Pedro: (1999): *Programas didácticos: diseño y evaluación*. Consultado el 22/05/99 www.xtec.es/~marquès/edusoft
- Guiraudó, María Teresa (1997): *Seminario de Psicosociología de los Aprendizajes*. UTN-FRBA.

- Guitert Catasús, Montserrat (1999): *Principios a tener en cuenta para una buena práctica pedagógica en Tecnología educativa y en educación a distancia*. III Curso Internacional de Tecnología Educativa Apropiaada. 8 y 9 de mayo.
- Gutiérrez, M. C. (1997) *Transferencia de masa; un problema a resolver*. Seminario de Psicosociología de los Aprendizajes. Maestría en Docencia Universitaria. UTN-FRBA.
- Halstead M. (1975): *Elements of software science*. Nueva York. Elsevier.
- Hammond N., Trapp A. et al. (1996): *Evaluating educational software: a suitable case for analysis*. AACE. www.york.ac.uk/inst/ctipsych/ewb/CTI/WebCip/Hammond.html
- Hartley J. (1972): *Strategies for Programmed Instruction: An Educational Technology*. Londres, Butterworths, citado por Cruz Feliú, Jaime (1986) en *Teorías del Aprendizaje y Tecnología de la Enseñanza*, Trillas.
- Hativa W. y Reingold A. (1987): *Effects of audiovisual stimuli on learning through microcomputer-based class presentation*, Instructional Science, 16, 287-306, citado en Cabero Almenara (1992).
- Heller R. (1991): *Evaluating Software: A review of the options*, Computers and education, 17, (4) págs. 285-291.
- Henderson-Sellers B. y Edwards J. M. (1990): *Book Two of Object-Oriented Knowledge: The Working Object*; Prentice Hall.
- Henry S. y Kafura D. (1984): *The evaluation of software systems: software practice an experience*. Vol. 14, número 6, págs. 561-573.
- Hernández Rojas G. (1998): *Paradigmas en psicología de la educación*. Paidós Educador.
- Holder Paula (1996): Seymour Papert 1965-1996. Consultado el 24/05/99 a las 22:40. www.ezinfo.ucs.indiana.edu/~pjholder/page3.html
- Hopper y Hannafin, (1991) *Psychological Perspectives on emerging instructional Technology: A Critical Analysis*. Educational Psychologist, 26, 69-95, citado en Schunk Dale H.: (1997): *Teorías de la Educación*, Prentice Hall.
- IEEE (1984a): IEEE 730. *Standard for software quality assurance Plans*. N. Y.
- IEEE (1986): a. Standard 1008. *Standard for Software Unit Testing*. N. Y. B. Standard 1012. *Standard for software verification and validation Plans*.
- IEEE (1989) Normas para el Aseguramiento de la Calidad
- IEEE (1990): Standard 610, *Computer Dictionary*. Nueva York .
- IEEE (1991): *Standard for Developing Software Life Cycle Process*. IEEE Std. 1074-1991 Nueva York. IEEE Computer Society.
- IEEE (1991b): *IEEE Standard for software Test and Documentation*. Std. 820-1983.
- IEEE (1992): *Standard for a software quality metrics methodology*. Std.1061
- ISO (1991): *Information Technology Software Quality Evaluation Characteristics*. ISO 9126. Ginebra, Suiza.
- ISO (1994): ISO/IEC 12701-1, *Software Life-cycle Process*.
- ISO (1995) 12207-1: *Information Technology-Software Life Cycle Processes*. International Standard Organization. Suiza.
- ISO 8402 (1994): *Gestión de la calidad y aseguramiento de la calidad*. Vocabulario.
- ISO 9000 (1994): *Normas para la gestión de la calidad y el aseguramiento de la calidad*. Guía para su selección y uso.
- ISO 9001 (1994): *Sistemas de la calidad*. Modelo para el aseguramiento de la calidad en el diseño, suministro y mantenimiento de soportes lógicos.
- J. Juzgado, N. (1996): *Procesos de construcción del software y ciclos de vida*. Universidad Politécnica de Madrid.
- Jackson M. A. (1975): *Principles of Program Design*. Nueva York. Academic Press.
- Johnson-Laird, P.N. (1998): *El ordenador y la mente: introducción a la ciencia cognitiva*. Paidós.
- Johnston V. M. (1987a): *Attitudes towards microcomputers in learning: 2. Teachers and software for language development*, Educational Research, 29, 2, 137-145, citado en Cabero Almenara (1992).
- Johnston V. M. (1987b): *The evaluation of Microcomputer Programas: An area of debate*, Journal of Computer Assisted Learning, 3, (1): págs. 40-50.
- Juran J. M. (1995): *Análisis y Planeación de la calidad*. Mc Graw Hill
- Kemmis S. (1976): *The educational Potential of Computer Assisted Learning: Qualitative Evidence About Student Learning*. U. K. University of East Anglia.
- Kemmis S., Atkin R. y Wright E. (1973-1975): *How do students learn?: Working papers on CAL*. Documento de trabajo número 5. Centre for applied Research in education. University of East Anglia. Gran Bretaña, citado en Squires y Mc Dougall (1994).
- Komosky P. et al. (1995): *Seven steps to responsible software selection*. Eric Digest. Clearinghouse on information and Technology, Syracuse. N. Y.

- Konrad M., Paulk M. y Graydon A. (1995): *An overview of Spice's model for Process Management*. SEI. Proceedings of the Fifth International Conference on Software Quality, Austin, TX. 1995 Págs. 291-301
- Ktchewan y Walter (1989): Citado en Fenton (1991): *Software Metrics. A rigorous and practical approach*. PWS Publishing Company. Boston.
- Lachman R. et al. (1979): *Cognitive psychology and information processing: An introduction*. Hillsdale, N. J. Erlbaum.
- Laurel B. (1990): *The art of human computer interface design*. Nueva York. Addison Wesley.
- Ledesma D. A. (1980): *Estadística Médica*. Eudeba
- Lehman M. (1984): *A Further Model of Coherent Programming Processes. Workshop*, Egham, UK, febrero, págs. 27-33, citado en Piattini (1996).
- Lepper (1985): *Microcomputer in education: Motivational and social Issues*. American Psychologist, 40, 1-18, citado en Schunk Dale H.: (1997): *Teorías de la Educación*, Prentice Hall.
- Libedinsky, M. (1995): *La utilización del correo electrónico en la escuela*, en Litwin (1995): *Tecnología educativa. Políticas, historias, propuestas*, Paidós.
- Liguori, L. (1995): *Las nuevas tecnologías de información y comunicación*, en Litwin (1995): *Tecnología educativa. Políticas, historias, propuestas*, Paidós.
- Llorca J. et al. (1991): *Desarrollo de software dirigido a objetos. (DDO)*: En Novática, vol. XVIII, número 47, citado en Piattini (1996).
- Logo, (1994) *Logo: Educational applications of*. (1994): *In the International Encyclopedia of Education*, vol. 15, pág. 3508-3512, citado en Seymour Papert 1965-1996 por Paula Holder (1996)
- MacDonald B., Atkin R., Jenkins D. y Kemmis S. (1977): *Computed Assisted Learning: its educational potential*, en Hooper R. (Ed.): *Final Report of the Director National Development program in Computer Assisted Learning*. Londres. Council for Educational Technology, citado en Squires y Mc Dougall (1994).
- Maddison R. N. (1983): *Information System methodologies*. Wiley Henden.
- Mager R. F. (1967): *Formulación operativa de objetivos didácticos*, Madrid, Marova.
- Manual de la Universidad de Málaga. Bioestadística: Métodos y Aplicaciones. ISBN 847496-653-1. Facultad de Medicina. consultado el 28/9/99 a las 10 hs. www.ftp.medprev.uma.es/libro/node148.htm
- Markle S. M. (1967): *Empirical Testing of Programs en P. C. Lange Ed. Programmed Instruction*, Chicago University of Chicago Press, págs. 104-138, citado por Cruz Feliú, Jaime (1986) en *Teorías del Aprendizaje y Tecnología de la Enseñanza*, Trillas.
- Marquès P. (1995): *Metodología para la elaboración de software educativo en Software Educativo. Guía de uso y metodología de diseño*. Barcelona Estel. Consultado el 24/05/99 a las 23 horas. www.xtec.es/~pmarques, www.doe.d5.ub.es
- Marquès, Pere: (1998a): *La evaluación de programas didácticos*. Comunicación y Pedagogía, N° 149, p. 53-58. Barcelona.
- Marquès, Pere: (1998b): *Programas didácticos: diseño y evaluación*. Universidad Autónoma de Barcelona. Consultado el 15/10/98 a las 22:30 horas www.doe.d5.ub.es/te
- Martin J. y Odell J. (1997): *Métodos orientados a objetos*. Prentice Hall.
- Mc Cracken D. y Jackson A. (1982): *Lifecycle concepts considered harmful*: ACM, Sigsoft Software Engineering Notes, vol. 7, número 2, abril, págs. 29-32, citado en Piattini (1996).
- McCabe T. (1976): *A complexity measure*. IEEE Transactions on software Engineering, vol.2, número 4, págs. 308-320.
- McCall J. (1977): *Factors in software quality*, vols. I, II y III. NTIS; Roma, citado en Piattini (1996)
- Meritxell Estebanell (1996): *Ficha de Evaluación de Programas Educativos*, Universidad de Girona.
- Meyer B. (1990): *La nueva cultura del desarrollo de software*. En System, setiembre, págs. 12-13, citado en Piattini (1996).
- MicroSIFT (1982): *Evaluation guide for Microcomputer-Based Instructional Packages. Microcomputer Software Information for Teachers*. (MicroSIFT): Northwest Regional Laboratory, Oregon, citado en Squires y Mc Dougall (1994).
- Morín Edgard (1995): *Ciencia con conciencia*. Anthropolos.
- Murrilo F.J. y Fernández M. J. (1992): *Software educativo. Algunos criterios para su evaluación*, Infodidac, 18, 8-12.
- Myers G. (1975): *Reliable Systems through Composite design*, 1º Ed. Petrocelli Charter., citado en Piattini (1996).
- Naur P. Y Randell B. (1969): Editores. *Software engineering: A report on a Conference sponsored by the NATO Science Committee*, citado en Pressman (1993).
- Newell y Simon (1975): *Procesamiento de la información en la computadora y en el hombre*, en Crosson F. J. (comp.): *Inteligencia humana e Inteligencia Artificial*. Fondo de Cultura Económica: México.

- Nielsen Jacob: (1995): *Multimedia and Hypertext, The Internet and Beyond*, - AP Professional.
- Norman D. (1988): *The psychology of everyday things*. New York. Basic Books.
- Norman D. y Drapper S. (1988): *User centered system design*. Hillsdale. N.J: Lawrence Erlbaum.
- Novak J. y Gowin D. B. (1988): *Aprendiendo a aprender*, Barcelona. Martínez Roca.
- OCDE (Organización para la Cooperación y el Desarrollo Económico) (1989): *Information Technologies in Education: The Quest for Quality Software*, Paris, Organisation for the Economic Cooperation and Development.
- Olivares M. A. et al. (1990): *A proposal to answer the necessity to evaluate computer software*, en McDougall, A. y Dowling, G. (editors): *Computers in Education*, Elsevier Science Publishers, North-Holland, 171-174,
- Osuna J., Bermejo J. L. y Berroso J. (1997): *Evaluación de medios informáticos: una escala de evaluación para software educativo*. EDUTEC 97. Comunicaciones: Formación y recursos.
- OTA (Office Technology Assessment de E.E.U.U.) (1988): *Power on! New tools for Teaching and Learning*. Washington D. C, U.S. Government Printing Office, citado en Squires y Mc Dougall (1994).
- Page-Jones M. (1980): *The Practical Guide to Structured Systems Design*. 1º Ed. Yourdon Press.
- Papert Seymour: (1981): *Desafío a la mente*, Ediciones Galápagos.
- Pelgrum J. y Plomp T. (1991): *The use of computers Worldwide*. Oxford, Pergamon Press, citado en Squires y Mc Dougall (1994).
- Perkins D. (1995): *La Escuela Inteligente*. Gedisa
- Pessacq R., Iglesias O. et al. (1997): *Evaluation of University Educational Software*. John Wiley & Sons. Apl. Eng. Educ. 5: 181.185.
- Piaget J.(1989): *La construcción de lo real en el niño*. Crítica. Grijalbo.
- Piattini M. (1996): *Análisis y Diseño Detallado de Aplicaciones Informáticas de Gestión*. Rama. Madrid.
- Pina, Bartolomé (1998): *Sistemas multimedia en educación*. Consulta on line www.doe.d5.es.te/WEBNTES/t, 20 de abril de 1999 a las 21 horas.
- Pozo Municio, I: (1998): *Aprendices y Maestros*. Alianza.
- Preece J. y Jones A. (1985): *Training teachers to select educational software: results of a formative evaluation of an Open University pack*, British Journal of Educational Technology, 16, 1, 9-20, citado en Cabero Almenara (1992).
- Prendres Espinosa M. P. (1996): *El multimedia en entornos educativos*, en *II Jornadas sobre medios de comunicación, recursos y materiales para la mejora educativa*, Sevilla, Centro Municipal de Investigación y Dinamización Educativa y Secretariado de Recursos Audiovisuales y Nuevas Tecnologías, citado en Castillo Segurado (1997).
- Pressman R.(1993): *Ingeniería de Software. Un enfoque práctico*. Mc Graw Hill.
- Raven J. C. (1979): *Test de Matrices Progresivas. Escala General*. Vol. 3b. Paidós. Buenos Aires.
- Raven J. C. (1979): *Test de Matrices Progresivas. Manual para la Aplicación*. Paidós. Buenos Aires (con notas de Jaime Bernstein).
- Reay D. G. (1985): *Evaluating Educational software for the classroom*, en Reid I, y Rushton J. (Eds.): *Teachers, computers and the classroom*. Manchester. Manchester University Press, págs. 79-87, citado en Squires y Mc Dougall (1994).
- Reeves T. C. (1993): *Evaluating technology based learning*, in Piskurich. ASTD. Handbook of Information Technology, citado en Reeves T. C. (1997).
- Reeves T. C. (1997): *Evaluation tools*, consulta on line en diciembre de 1998. www.mime1.marc.gatech.edu/MM_tools/evaluation.html
- Requena A. y Romero F. (1983): *¿Cómo seleccionar el software educativo?*, *El ordenador personal*, 13, 47-51, citado en Cabero Almenara (1992).
- Rivera Quijano M. (1999): *Nuevos caminos para evaluar proyectos y materiales educativos tecnológicos y para educación a distancia*. III Curso Internacional de Tecnología Educativa Apropiaada. 8 y 9 de mayo de 1999.
- Rivière A. (1987): *El sujeto de la psicología cognitiva*. Madrid. Alianza.
- Rogers C. (1984): *Libertad y creatividad en la educación*. Paidós
- Romiszowski (1981): Universidad de Syracuse. E.E. U: U: *Designing Instructional System*. London: Nichols Kogan Page.
- Romiszowski A. D. (1981): citado en *Psicología y Curriculum* por César Coll (1994): Paidós.
- Rowntree D. (1982): *Educational Technology in curriculum development*. Londres. Harper and Row, citado en Squires y Mc Dougall (1994).
- Royce W. (1970): *Managing the development of Large software systems: Concepts and techniques*. Proceedings, Wescon, agosto, 1970, citado en Piattini (1996).
- Rumbaugh J. (1991): *Object Oriented modeling and design*: Prentice Hall, Englewood Cliffs. Nueva Jersey.
- Rumbaugh J. (1992): *Over waterfall and into the whirlpool*. En JOOP, mayo, págs. 23-26, citado en Piattini (1996).

- Salvas A. D. y Thomas G. J. (1964): *Evaluation of software*, Melbourne, Department of Victoria, citado en Squires y Mc Dougall (1994).
- Sánchez J. y Alonso O. (1997-8): *Evaluación distribuida de software educativo a través de Web*. www.dcc.uchile.cl/~oalonso/educacion/, consultado el 23/11/98 a las 23:30 horas.
- Sancho J. (1994): *Para una Tecnología Educativa*, Editorial Horsori. Barcelona. España.
- Schunk Dale H. (1997): *Teorías de la Educación*, Prentice Hall.
- Self J. (1985): *Microcomputers in Education: a critical appraisal of educational software*. Brighton, Harvester Press, citado en Squires y Mc Dougall (1994).
- Shall W. E., Leake L. y Whitacker W. (1986): *Computer education: Literacy and beyond*. Monterrey, California. Brooks-Cole, citado en Squires y Mc Dougall (1994).
- Sigwart C. et al. (1990): *Software Engineering: a project oriented approach*. Franklin, Beedle y Associates, Inc., Irvine, California, citado en Piattini (1996).
- Skinner B. F., (1958, 1963): Teaching Machines, Science, publicado en 1958; Reflection on a decade of teaching Machines, publicado en 1963, citados por Cruz Feliú, Jaime (1986) en *Teorías del Aprendizaje y Tecnología de la Enseñanza*, Trillas.
- Smith D. y Keep R. (1986): *Children's opinions of educational software*, Educational Research, 28, 2, 83-88, citado en Cabero Almenara (1992).
- Solomon, C. (1987): *Entornos de aprendizaje con ordenadores. Una reflexión sobre las teorías del aprendizaje y la educación*. Temas de Educación. Paidós. M.E.C.
- Sommerville Y. (1985): *Software Engineering*. Addison Wesley.
- Squires D. y Mc Dougall A. (1994): *Cómo elegir y utilizar software educativo*. Morata. Barcelona.
- Stufflebeam D. y Shinkfield (1987): *Evaluación Sistemática*. Paidós.
- Taylor R. P. (1980): *The computer in the School: tutor, tool, tutee*. Nueva York. Teachers College Press, citado en Squires y Mc Dougall (1994).
- Templeton R. (1985): *Be careful but Don't worry: a guide to buying educational software*, en Tagg W, (Ed.): *A Parent's Guide to Educational Software*, Londres, Telegraphs Publications, págs. 54-64, citado en Squires y Mc Dougall (1994).
- Truett A. (1984): *Field testing educational software: are publishers making the effort?*, Educational Technology, mayo, 7-12, citado en Cabero Almenara (1992).
- Underwood J. D. y Underwood G. (1990): *Computers and Learning*, Oxford, Basil Blackwell, citado en Cabero Almenara (1992).
- Valencia M. E., Toro I. y Donneys C. (1998): *Desarrollo de aplicaciones hipemedia: propuesta para el diseño educativo*. TISE'98. Consultado el 28/9/99 a las 10 hs. en www.sofia.univalle.edu.co/gidse
- Vigotskii L. (1978): *Mind in Society. The development of higher psychological process*. Cambridge. M. A. Harvard University Press.
- Villar, M.; Mínguez, E. (1998): *Guía de evaluación de software educativo*. Grupo ORIXE. Euskadi.
- Wellington J. J. (1985): *Children, computers and the curriculum*, Cambridge, Harper & Row, Publishers, citado en Cabero Almenara (1992).
- Winograd T. (1996): *Bringing design to software*. New York. ACM Press.
- Wishart J. (1989): *Cognitive factors related to the user involvement with computers and their effects upon learning an educational computer game*. Paper read at the Cal'89, Conference University of Surrey, citado en Cabero Almenara (1992).
- Yin B. y Winchester J. (1978): *The establishment and use of measures to evaluate the quality of software design*. Performance Evaluation Review, vol. 7, número 3-4, págs 45-52, citado en Piattini (1996).
- Yourdon E. y Constantine L. (1975): *Structured design*, 2º Ed. Englewood Cliffs, Prentice Hall.
- Zangara A. (1998): *Seminario de Sistemas Multimediales Aplicados a la Educación*. UTN.

Criterios para seleccionar y evaluar Un software de mantenimiento

(Revista Mantenimiento - Chile - N° 26 - AÑO 1996 – ISS 0716 – 8616)

Lourival Augusto Tavares

Presidente del Comité UPADI

De ingeniería de mantenimiento

Hasta la década de los 80, las industrias de la mayoría de los países occidentales tenían sus objetivos basados en obtener el máximo de rentabilidad para una inversión efectuada. Posteriormente, con la penetración de la industria oriental en el mercado occidental, el consumidor pasó a considerar un complemento importante en los productos a adquirir, o sea, la calidad de los productos o servicios provistos y esta exigencia hizo que las empresas considerasen este factor, calidad, como una necesidad para mantenerse competitivas, particularmente en el mercado internacional, transfiriendo a los gerentes de operación y mantenimiento, el mejor resultado posible en sus funciones para lograr obtener contabilidad, disponibilidad y reducción de plazos de fabricación con bajos costos. Además, la exigencia de la confiabilidad y disponibilidad es de tal orden que se impone al gerente de mantenimiento, responsabilidades que sólo pueden ser ejecutadas con herramientas adecuadas de gestión.

En consecuencia, las empresas buscan cada vez más, sistemas informatizados adecuados para auxiliar a esos gerentes en sus funciones. Esta búsqueda llevó a la comercialización, en apenas uno de los países europeos, de más de 3,300 sistemas de gestión de mantenimiento (1) de los cuales 2,470 están en operación. Algunos de esos sistemas son comercializados junto con un Análisis y Diagnóstico y, prácticamente todos, de forma modular e integrada.

En los países americanos también existe una gran cantidad de sistemas de gestión de mantenimiento ofrecidos como la solución final de los problemas de los gerentes de mantenimiento (2), sin embargo, después de su adquisición la realidad muestra que, en vez de obtener soluciones para sus problemas, los gerentes en la realidad adquirirán más problemas para administrar.

Como ejemplo de esas adquisiciones inadecuadas, citamos los casos de dos empresas que adquirieron sistemas de gestión que ocasionaron incendios por falta de ejecución de mantenimiento (que era hecho cuando había sistema manual) con grandes perjuicios; acarreando, en la primera, la necesidad de reforma total de las instalaciones y, en la segunda, la necesidad de venta de la fábrica debido a pérdida total. En los dos casos los gerentes de mantenimiento fueron despedidos aunque no tuvieron participación en la compra del sistema. En otra empresa de transporte un Director compró un sistema "especialista" en un valor que fue totalmente rehecho por la gente de informática y de mantenimiento de la empresa con una pérdida de más de 2 millones de dólares.

Estamos seguros que más del 50% de los sistemas comercializados no llegan a atender adecuadamente a las empresas y lamentablemente no son divulgadas esas experiencias negativas, con raras excepciones.

De esta forma, los gerentes deben preocuparse en la selección de un sistema que realmente atienda a sus necesidades, no sólo basados en las demostraciones hechas por los proveedores y sí con una investigación consciente de las consecuencias que vendrán con la adquisición del sistema.

Como sugerencia indicamos, en el listado presentado a continuación, algunas características que deben ser observadas en la selección de softwares de mantenimiento:

1.- Que el proveedor tenga los programas "fuente" para venderlos, en caso de interés del cliente (naturalmente bajo criterios que eviten la comercialización del sistema por el cliente o por cualquiera de sus funcionarios);

2.- Que el sistema opere en el ambiente o plataforma utilizado por la empresa así como tenga las características de un mono multiusuario, de acuerdo con la necesidad;

3.- Que el proyectista sea un experto en mantenimiento y que continúe produciendo nuevas versiones;

4.- Que el sistema sea de fácil operación no exigiendo, en consecuencia, la participación de ingenieros o técnicos especializados para la ejecución de sus tareas cotidianas.

5.- Que el sistema pueda ser comercializado de forma modular, pero sin exigir ninguna adecuación a medida que sean adquiridos nuevos módulos y que sea de fácil navegabilidad entre las pantallas, ventanas y módulos.

6.- Que los códigos sean compuestos por células para permitir selecciones o filtros en los reportes y listados y además que el contenido de esas células sean establecidas por el propio usuario, a partir de las tablas patrones para sus necesidades;

7.- Que la recolección de datos de mano de obra sea independiente de las órdenes de trabajo de forma que permita su implementación en cualquier momento;

8.- Que exista la posibilidad de integrar los sistemas de gestión de material de forma que el sistema de mantenimiento informe al sistema de material las necesidades para los servicios programables y hasta inicie el proceso de reposición de stocks y el sistema de material provea al sistema de mantenimiento, los costos de repuestos y material de uso común;

9.- Que sea posible monitorear servicios de terceros, tanto a través de contratos permanentes y globales como a través de servicios eventuales.

10.- Que existan niveles de acceso para restringir algunas operaciones sólo a usuarios acreditados como, por ejemplo, recuperación de datos de back-up, operación con sueldos, acceso a reportes confidenciales, exclusión de informaciones de los archivos, etc;

11.- Que la capacidad de memoria (RAM) necesaria para el procesamiento del sistema, sea compatible con la disponible en los equipos de la empresa así como la capacidad del almacenaje de datos por períodos de consulta definidos por el usuario y la creación de archivos “muertos” a partir de plazos también definidos por el usuario;

12.- Contestación rápida a consultas cuando los archivos están muy cargados de informaciones. En este caso es recomendable analizar el tiempo de procesamiento cuando los archivos más usuales llegan a ocupar más de 1Mbyte de capacidad.

Al acompañar una demostración, procure llevar con usted e implemente ejemplos reales ocurridos en su unidad;

13.- Garantía de ejecución de back-up automáticamente, de forma eficiente, rápida y compactada;

14.- Que sea permitido cambiar títulos y leyendas para personalizar las informaciones de la empresa(así como cambios de idioma);

15.- Que sea permitido crear nuevos reportes de acuerdo con la necesidad del usuario a partir de los datos existentes en los archivos.

16.- Atender la gestión de costos, de material (en el nivel de mantenimiento) y de mano de obra de acuerdo con las reales necesidades del usuario;

17.- Posibilidad de implementación de recursos de sistema experto con módulo de mantenimiento predictivo, alertas a la gerencia de mantenimiento y nivelación de recursos de mano de obra;

18.- Que los costos sean adecuados y los pagos puedan ser hechos de forma parcial, o sea de acuerdo con la implementación de cada módulo, así como los costos sean para toda la empresa y no sólo para cada copia del sistema provista.

Un hecho importante es que los gerentes deben estar conscientes de que la selección del software no cierra la tarea de informatizar el proceso de planificación y control del mantenimiento, una vez que la formación de los archivos iniciales (inventarios de equipos y correlación con repuestos, programación, instrucciones, recomendaciones y valores estándares de medición) vayan a necesitar de gran inversión de tiempo de personal técnico para lograr que el sistema esté en condiciones de operar.

Se puede considerar como promedio que cada técnico de mantenimiento puede hacer alrededor de 6 inventarios completos de equipos por día y los ingenieros podrán hacer 30 programaciones promedios por día. En términos de costo esto significa que la obtención y digitación de datos de inventario y programación en el sistema para que pueda iniciar su operación es algo como 8 veces el costo de adquisición de un paquete de sistema monousuario.

Finalmente cabe destacar las dificultades que serán encontradas para iniciar la operación del sistema después de instalar y procesar los archivos básicos. Esas dificultades están muy relacionadas a reacciones del personal en llenar correctamente los documentos para realimentación del sistema (historia de ocurrencias, consumo de hombres-hora y material, cambios de localización, etc.) así como de los solicitantes de servicios en la solicitud a través del órgano competente, la atribución correcta del grado de prioridad y en la evaluación de los servicios (calidad del mantenimiento). Normalmente estas dificultades (o cambio de actitudes) son superadas entre el 1 y 2 años a partir del momento de implementación del sistema.

Referencias

1 Progiciels de maintenance- les grandes tendances

Edmond Kloeckner

Maintenance & enterprise

“Best of the best” study

A.T. Kearney in conjunction with Industry Week magazine 1990

Software de Manutencao: uma panacea e seus males

Revista Manutecdo
ABRAMAN – Associacao
Brasileira de manutencao
Nº 36 – maio/junho/92
Pp16 a 21

APRUEBAN DOCUMENTO “GUÍA TÉCNICA SOBRE EVALUACIÓN DE SOFTWARE PARA LA ADMINISTRACIÓN PÚBLICA”

RESOLUCIÓN MINISTERIAL N° 139-2004-PCM

Lima, 27 de mayo de 2004

CONSIDERANDO:

Que, mediante el Decreto Supremo N° 066-2003-PCM se fusionó la Subjefatura de Informática del Instituto Nacional de Estadística e Informática - INEI y la Presidencia del Consejo de Ministros, en virtud a lo cual el numeral 3.10 del artículo 3° del Reglamento de Organización y Funciones de la Presidencia del Consejo de Ministros, aprobado por Decreto Supremo N° 067-2003-PCM, ha establecido que es función de la Presidencia del Consejo de Ministros actuar como ente rector del Sistema Nacional de Informática;

Que, a efectos de implementar la infraestructura de Gobierno Electrónico, el mismo que comienza con la identificación y evaluación de los componentes funcionales requeridos, adopción de estándares abiertos y aceptados internacionalmente, la planificación y seguridad, en el marco de sus funciones la Oficina Nacional de Gobierno Electrónico e Informática – ONGEI, en coordinación con el Instituto Nacional de Defensa de la Competencia y de la Protección de la Propiedad Intelectual – INDECOPI, ha propuesto la “Guía Técnica Sobre Evaluación de Software para la Administración Pública”, por ser éste el que procesa datos y produce información, que es considerada actualmente un activo importante y estratégico de las organizaciones y países;

De conformidad con lo dispuesto por el Decreto Legislativo N° 560 – Ley del Poder Ejecutivo y el Reglamento de Organización y Funciones de la Presidencia del Consejo de Ministros, aprobado por Decreto Supremo N° 067-2003-PCM;

SE RESUELVE:

Artículo 1º.- Aprobar el documento “Guía Técnica Sobre Evaluación de Software para la Administración Pública”, documento que será publicado en el portal de la Presidencia del Consejo de Ministros (www.pcm.gob.pe).

Artículo 2º.- Las entidades de la Administración Pública, integrantes del Sistema Nacional de Informática, deberán aplicar lo establecido en la “Guía Técnica Sobre Evaluación de Software para la Administración Pública” en los productos de software que desarrollen o adquieran a partir de la fecha de publicación de la presente Resolución.

Si no fuera posible parcial o totalmente su aplicación, el área de informática o la que haga sus veces de la entidad respectiva, comunicará esta situación a la Oficina Nacional de Gobierno Electrónico e Informática – ONGEI, adjuntando el informe técnico que sustente la justificación para la no aplicación de la citada Guía.

Regístrese, comuníquese y publíquese.

CARLOS FERRERO
Presidente del Consejo de Ministros

Publicado en el Diario Oficial “El Peruano” el 28/05/04



**Oficina Nacional de Gobierno Electrónico e Informática
Presidencia del Consejo de Ministros**

Guía Técnica sobre Evaluación de Software en la Administración Pública

Presidencia del Consejo de Ministros – Gobierno del Perú – ONGEI
formatos@pcm.gob.pe

Ref: Guía Técnica de Evaluación
Versión: 01

Nombre del Proyecto: *“Guía Técnica sobre Evaluación de Software
en la Administración Pública”*



HOJA DE INFORMACION GENERAL

CONTROL DOCUMENTAL:

PROYECTO:	Guía Técnica sobre Evaluación de Software en la Administración Pública
ENTIDAD:	Presidencia del Consejo de Ministros
VERSIÓN:	1.0
FECHA EDICIÓN:	05/05/2004
NOMBRE DE ARCHIVO:	P01-PCM-GUIAEVALUACIONSOFTWARE
RESUMEN:	Guía que permite evaluar eficientemente los desarrollos y software en el Estado.

DERECHOS DE USO:

La presente documentación es de uso para la Administración Pública del Estado Peruano.

ESTADO FORMAL:

Preparado por: ONGEI

Entidad: ONGEI – PCM

Fecha: Mayo 2004

Presidencia del Consejo de Ministros – Gobierno del Perú – ONGEI

formatos@pcm.gob.pe

Ref: Guía Técnica de Evaluación
Versión: 01
Fecha: 05/05/04

Nombre del Proyecto: “Guía Técnica sobre Evaluación de Software en la Administración Pública”



CONTROL DE VERSIONES

FUENTE DE CAMBIO	FECHA DE SOLICITUD DEL CAMBIO	VERSIÓN	PARTES QUE CAMBIAN	DESCRIPCIÓN DEL CAMBIO	FECHA DE CAMBIO
P01-PCM-GUIAEVALUACIONSOFTWARE.doc		1.00	N/A		



Índice

INTRODUCCIÓN	04
APLICACIÓN	04
ESTRUCTURA	04
 PARTE 1: MODELO DE LA CALIDAD	 05
1.1 Alcance	05
1.2 Conformidad	06
1.3 Marco de trabajo del modelo de la calidad	06
1.4 Modelo de calidad para la calidad externa e interna	11
1.5 Modelo de calidad para la calidad en uso	18
 PARTE 2: MÉTRICAS	 20
2.1 Atributos Internos y Externos	20
2.2 Métrica interna	21
2.3 Métrica externa	21
2.4 Relación entre las métricas internas y externas	21
2.5 Calidad en el uso de métricas	22
2.6 Opción de métrica y criterio de medidas	22
2.7 Métricas usadas para la comparación	23
 PARTE 3: PROCESO DE EVALUACIÓN DE SOFTWARE	 24
3.1 Establecer el propósito de la evaluación	24
3.2 Identificar el tipo de producto	24
3.3 Especificar el Modelo de Calidad	24
3.4 Seleccionar métricas	24
3.5 Establecer niveles, escalas para las métricas	25
3.6 Establecer criterios de valoración	25
3.7 Tomar medidas	25
3.8 Comparar con los criterios	26
3.9 Valorar resultados	26
3.10 Documentación	26
 GLOSARIO DE TÉRMINOS	 27
BIBLIOGRAFÍA	32



GUÍA TÉCNICA SOBRE EVALUACIÓN DE SOFTWARE PARA LA ADMINISTRACIÓN PÚBLICA

INTRODUCCIÓN

La presente guía esta basada sobre la norma ISO/IEC 9126 de la ISO (Organización Internacional de Normalización) y la IEC (Comisión Electrotécnica Internacional) que forman el sistema especializado para la normalización internacional.

El desarrollo o selección de productos de software con calidad es muy importante en la actualidad en las instituciones públicas, ya que éstas procesan información, que es considerada como un activo importante de sus organizaciones.

Una especificación y evaluación integral y detallada de la calidad de los productos de software es un factor clave para asegurar que la calidad sea la adecuada. Esto se puede lograr definiendo de manera apropiada las características de calidad, teniendo en cuenta el propósito del uso del producto de software en la institución.

Es importante especificar y evaluar cada característica relevante de la calidad de los productos de software, cuando esto sea posible, utilizando mediciones validadas o de amplia aceptación, que hagan técnicamente transparente esta actividad.

Agradecemos la colaboración del Comité Técnico de Normalización de Ingeniería de Software y Sistemas de Información - INDECOPI, por su apoyo técnico en la elaboración de la presente guía.

APLICACIÓN

- La presente guía es aplicable al **software propietario y software libre o de código abierto** utilizado en la Administración Pública.
- Esta guía debe aplicarse en toda evaluación de software propietario considerando esquemas comparativos con el software libre o de código abierto y viceversa, evidenciando ventajas y desventajas.
- Será utilizada para evaluar un solo software o un conjunto de softwares de naturaleza o funciones similares, tipo y/o categoría.

ESTRUCTURA

La presente guía consta de las siguientes partes:

- 1: Modelo de la calidad
- 2: Métricas



– 3. Proceso de evaluación de software

PARTE 1: MODELO DE LA CALIDAD

1.1 ALCANCE

Se describe un modelo de calidad para los productos de software, dividido en dos partes:

- a) Calidad interna y externa, y
- b) Calidad en uso.

La primera parte del modelo especifica seis características para calidad interna y externa, las cuales, a su vez, están subdivididas en sub características. Estas sub características se manifiestan externamente cuando el software es usado como parte de un sistema de computadora, y son el resultado de atributos internos de software.

La segunda parte del modelo, especifica cuatro características para la calidad en uso. Calidad en uso es el efecto combinado para el usuario de las seis características de la calidad interna y externa de productos de software.

Las características definidas son aplicables a todo software, incluyendo programas de computadora y datos contenidos en *firmware*. Las características y sub características proveen terminología consistente para la calidad de productos de software. Ellas también proveen un marco de trabajo para especificar los requerimientos de la calidad para productos de software, y para hacer análisis y evaluaciones entre capacidades de productos de software.

Esta parte de la norma permite especificar y evaluar la calidad de productos de software desde diferentes perspectivas asociadas con adquisición, requerimientos, desarrollo, uso, evaluación, soporte, mantenimiento, aseguramiento de la calidad y auditoría de software.

Esta norma será usada por personal de informática que cumple funciones de desarrolladores, adquirientes, personal de aseguramiento de la calidad y aquellos responsables de especificar y evaluar la calidad de productos de software.

Como ejemplo del uso del modelo de la calidad, tenemos:

- Validar que la definición de un requerimiento esté completa;
- Identificar requerimientos de software;
- Identificar objetivos del diseño de software;



- Identificar objetivos de prueba de software;
- Identificar criterios de aseguramiento de la calidad;
- Identificar criterios de aceptación para un producto de software completo.

1.2 CONFORMIDAD

Cualquier requerimiento, especificación o evaluación de la calidad sobre cualquier producto de software que cumpla esta parte de la guía, debe usar las características y sub características de los ítems 1.4 y 1.5, dando las razones por cualquier exclusión, o describiendo su propia categorización de los atributos de la calidad de productos de software, explicando la equivalencia respectiva.

1.3 MARCO DE TRABAJO DEL MODELO DE LA CALIDAD

En esta parte se describe el marco de trabajo de un modelo de calidad, el cual explica la relación entre los diferentes enfoques de la calidad.

1.3.1 Perspectivas de calidad

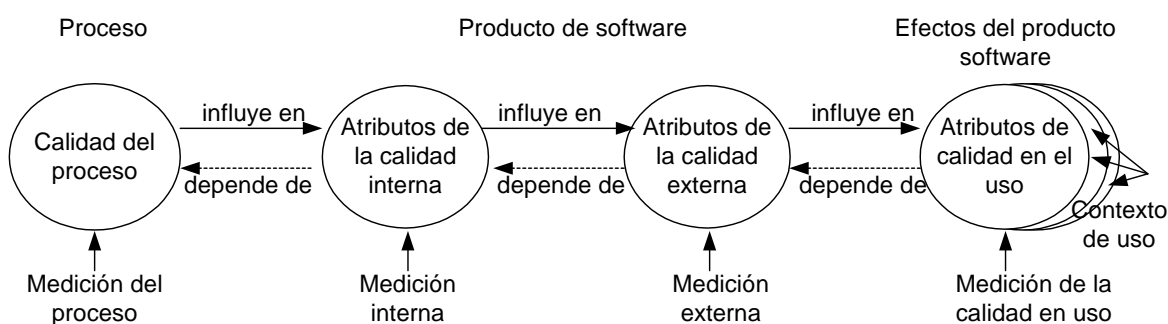


FIGURA 1 – Ciclo de vida de la calidad

Las necesidades de calidad del usuario incluyen requerimientos de calidad en uso, en contextos específicos. Estas necesidades identificadas pueden ser usadas cuando se especifiquen la calidad externa e interna, utilizando características y sub características de la calidad del producto de software.

La evaluación de los productos de software para satisfacer las necesidades de calidad es uno de los procesos en el ciclo de vida del desarrollo del software. La calidad del



producto de software puede ser evaluada midiendo atributos internos (medidas típicamente estáticas de productos intermedios), o midiendo atributos externos (midiendo típicamente el comportamiento del código cuando es ejecutado), o bien midiendo los atributos de aplicación de calidad en uso. El objetivo para que este producto tenga el efecto requerido en un contexto particular de uso se diagrama en la Figura 2.

La calidad del proceso contribuye a mejorar la calidad del producto, y la calidad del producto contribuye a mejorar la calidad en uso. Por lo tanto, evaluar y mejorar un proceso es una manera de mejorar la calidad del producto, y evaluar y mejorar la calidad del producto es una manera de mejorar la calidad en uso. De igual manera, evaluar la calidad en uso proporciona una retroalimentación para mejorar el producto, y evaluar un producto puede proporcionar una respuesta para mejorar un proceso.

Atributos internos apropiados en el software son pre requisitos para alcanzar el comportamiento externo requerido, y un apropiado comportamiento externo es un pre requisito para alcanzar la calidad en uso (Figura 1).

Los requisitos para la calidad del producto de software incluirán criterios de evaluación para calidad interna, calidad externa y calidad en uso, para cumplir las necesidades de los desarrolladores, responsables de mantenimiento, adquirientes y usuarios finales.

1.3.2 Calidad de producto y el ciclo de vida

Las vistas de calidad interna, calidad externa y calidad en uso cambian durante el ciclo de vida del software. Por ejemplo, la calidad especificada, como requisito de calidad al comienzo de un ciclo de vida, es mayormente observada desde el punto de vista externo y de usuario, y se diferencia de la calidad del producto intermedio, como la calidad del diseño, la cual es mayormente observada desde el punto de vista interno del desarrollador. Las tecnologías usadas para alcanzar el nivel de calidad necesario, así como la especificación y evaluación de calidad, necesitan soportar estos diversos puntos de vista. Es necesario definir estas perspectivas y las tecnologías asociadas a la calidad, para manejarla apropiadamente en cada etapa del ciclo de vida.

La meta es alcanzar la calidad necesaria y suficiente para cumplir con las necesidades reales de los usuarios. La norma ISO 8402 define calidad en términos de la habilidad de satisfacer necesidades explícitas (declaradas/descritas/especificadas) e implícitas. Sin embargo, las necesidades descritas por un usuario no siempre reflejan las verdaderas necesidades del mismo, porque:

- Un usuario normalmente no está consciente de sus necesidades reales.
- Las necesidades podrían cambiar después de ser especificadas.
- Diferentes usuarios pueden tener diferentes ambientes de operación.
- Podría ser imposible consultar a todos los posibles tipos de usuario, particularmente para un tipo de software (que no está en el mostrador/ producto preelaborado).



Por lo tanto, los requisitos de calidad no pueden ser completamente definidos antes de empezar con el diseño. Sin embargo, es necesario entender las necesidades reales del usuario tan al detalle como sea posible, y representarlas en los requerimientos. La meta no es obtener la calidad perfecta, pero sí la calidad necesaria y suficiente para cada contexto específico de uso, cuando el producto sea entregado y utilizado por los usuarios.

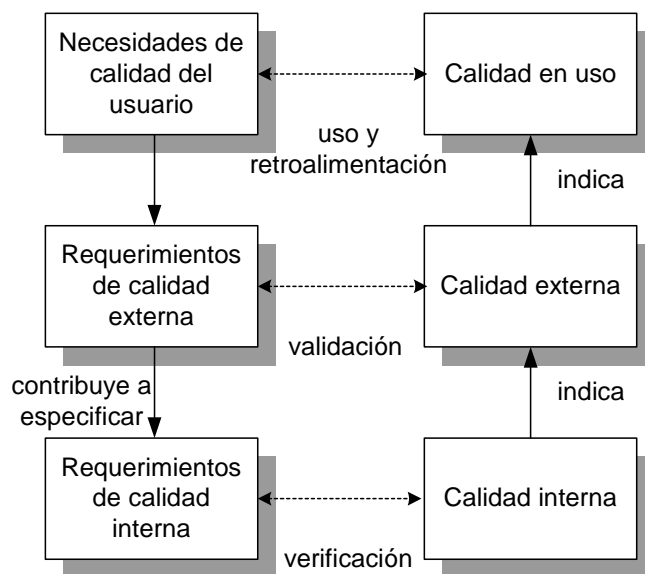


FIGURA 2 – Calidad en el ciclo de vida del software

Las escalas de medidas para las métricas usadas en los requerimientos de calidad pueden ser divididas en categorías correspondientes a diferentes grados de satisfacción de los requerimientos. Por ejemplo, la escala podría estar dividida en dos categorías: no satisfactoria y satisfactoria, o en cuatro categorías: excede los requerimientos, cumple los objetivos, mínimamente aceptable e inaceptable. Las categorías deberían ser especificadas para que ambos, el usuario y el desarrollador, puedan evitar costos innecesarios e incumplimiento de cronogramas. Existen diferentes perspectivas de la calidad del producto y sus métricas asociadas a las diferentes etapas del ciclo de vida del software. (Ver Figura 3)

Las **Necesidades de Calidad del Usuario** pueden ser especificadas como requerimientos de calidad por las métricas de calidad en uso, por métricas externas y a veces por métricas internas. Estos requerimientos especificados por las métricas, deberían ser usados como criterios cuando un producto es validado. Lograr un producto que satisfaga las necesidades del usuario, normalmente requiere de un enfoque interactivo en el desarrollo de software, con una continua retroalimentación desde la perspectiva del usuario.

Los **Requerimientos de Calidad Externos** especifican el nivel de calidad requerido desde una perspectiva externa. Estos incluyen requerimientos derivados de las necesidades de calidad de usuarios, incluyendo calidad en requerimientos de uso. Los



requerimientos de calidad externos son usados como los objetivos para la validación en varias etapas de desarrollo. Los requerimientos de calidad externos para todas las características de calidad definidas en esta parte, deben ser establecidos en la especificación de requerimientos de calidad usando métricas externas, deben ser transformados en requerimientos de calidad internos y deben ser usados como criterios cuando un producto es evaluado.

Los **Requerimientos de Calidad Internos** especifican el nivel de calidad requerido desde la perspectiva interna del producto. Los requerimientos de calidad internos son usados para especificar propiedades internas de productos. Estos pueden incluir modelos estáticos y dinámicos, otros documentos y código fuente. Los requerimientos de calidad internos pueden ser usados como objetivos para la validación en varias etapas de desarrollo. Ellos también pueden ser usados para definir estrategias de desarrollo y criterios de evaluación y verificación durante el desarrollo. Esto puede incluir el uso de métricas adicionales (por ejemplo: reusabilidad). Los requerimientos específicos de calidad interna deben ser especificados cuantitativamente usando métricas internas.

La **Calidad Interna** es la totalidad de características del producto de software desde una perspectiva interna. La calidad interna es medida y evaluada en base a los requerimientos internos de calidad. Los detalles de la calidad del producto de software pueden ser mejorados durante la implementación, revisión y prueba del código fuente del software, pero la naturaleza fundamental de la calidad del producto de software representada por la calidad interna, permanece sin cambios a menos que sea rediseñado.

La **Calidad Externa Estimada (o Predicha)** es la calidad que es estimada o predicha para el producto de software final, en cada etapa de desarrollo para cada característica de calidad, basada en el conocimiento de la calidad interna.

La **Calidad Externa** es la totalidad de las características del producto de software desde una perspectiva externa. Es la calidad cuando el software es ejecutado, la cual es típicamente medida y evaluada en un ambiente simulado, con datos simulados y usando métricas externas. Durante las pruebas, muchas fallas serán descubiertas y eliminadas. Sin embargo, algunas fallas todavía pueden permanecer después de las pruebas. Como es difícil corregir la arquitectura del software u otros aspectos fundamentales del diseño del software, el diseño fundamental permanece sin cambios a través de las pruebas.

La **Calidad en Uso Estimada (o Predicha)** es la calidad que es estimada o predicha para el producto de software final, en cada etapa de desarrollo para cada característica de calidad en uso, y se basa en el conocimiento de la calidad externa e interna.

La calidad externa y la calidad en uso pueden ser estimadas y predichas durante el desarrollo de cada característica de calidad cuando las tecnologías apropiadas son desarrolladas. Sin embargo, como actualmente no se proporciona todo el soporte necesario para el propósito de predicción, se debe desarrollar más tecnología para mostrar la correlación entre la calidad interna, la calidad externa y la calidad en uso.



La **Calidad en Uso** es la perspectiva del usuario de la calidad del producto de software cuando éste es usado en un ambiente específico y en un contexto de uso específico. Esta mide la extensión en la cual los usuarios pueden conseguir sus metas en un ambiente particular, en vez de medir las propiedades del software en si mismo.

El término 'Usuario' se refiere a cualquier tipo de posible usuario, incluyendo operadores y personal de mantenimiento, y sus requerimientos pueden ser diferentes.

El nivel de calidad en el ambiente del usuario puede ser diferente del ambiente de desarrollo, debido a diferencias entre las necesidades y capacidades de diversos usuarios y diferencias entre hardware y ambientes de soporte. El usuario evalúa sólo aquellos atributos de software que son usados para sus tareas. Algunas veces, los atributos de software especificados por un usuario final durante la fase de análisis de requerimientos, ya no cumplen los requerimientos del usuario cuando el producto está en uso, debido a cambiantes requerimientos del usuario y a la dificultad de especificar necesidades implícitas.

1.3.3 Ítems a ser evaluados

Los ítems pueden ser evaluados por medición directa, o de manera indirecta, midiendo sus consecuencias. Por ejemplo, un proceso puede ser medido indirectamente por la medición y evaluación de sus productos, y un producto puede ser evaluado indirectamente por la medición del desempeño de un usuario en sus tareas (usando métricas de calidad en uso).

El software nunca corre solo sino que siempre es parte de un sistema mayor, típicamente consistente de otros productos de software con los cuales él tiene interfaces: hardware, operadores humanos, y flujos de trabajo. El producto de software completado puede ser evaluado por los niveles de las métricas externas elegidas. Estas métricas describen su interacción con su entorno, y son medidas al observar el software en operación. La calidad en uso puede ser medida por la extensión por la cual un producto empleado por usuarios específicos cumple las necesidades de alcanzar metas específicas con efectividad, productividad, seguridad y satisfacción. Esto normalmente será complementado con mediciones de características de calidad más específicas del producto de software, lo cual también es posible en el proceso inicial de desarrollo.

En etapas más tempranas de desarrollo, sólo pueden ser medidos los recursos y procesos. Cuando los productos intermedios (especificaciones, código fuente, etc.) se tornan disponibles, estos pueden ser evaluados por los niveles de las métricas internas elegidas. Estas métricas pueden ser usadas para predecir los valores de las métricas externas. Ellas también pueden ser medidas por derecho propio, al ser pre requisitos esenciales para la calidad externa.

Se puede hacer una distinción adicional entre la evaluación del producto de software y la evaluación del sistema en el cual es ejecutado.

Por ejemplo, la confiabilidad de un sistema es medida al observar todas las fallas originadas por cualquier causa (hardware, software, errores humanos, etc.), mientras que la confiabilidad del producto de software es medida al extraer de las fallas



observadas sólo aquellas que son debidas a faltas en el software (originadas en requerimientos, diseño o implementación).

Además, depende del propósito de la evaluación y de quienes son los usuarios, el juzgar dónde están los límites del sistema.

En ese sentido, por ejemplo, si se supone que los pasajeros son los usuarios de un avión con un sistema de control de vuelo basado en computadora, entonces el sistema del cual ellos dependen incluye la tripulación, el fuselaje, el hardware y software del sistema de control de vuelo, mientras que si se toma a la tripulación como los usuarios, entonces el sistema del cual ellos dependen consiste sólo del fuselaje y el sistema de control de vuelo.

1.3.4 Usando un modelo de calidad

La calidad de un producto de software se debe evaluar usando un modelo definido. El modelo de calidad debe ser utilizado al fijar las metas de la calidad para los productos de software y los productos intermedios. La calidad del producto de software debería ser jerárquicamente descompuesta en un modelo de calidad constituido por características y sub características, las cuales se pueden utilizar como lista de comprobación de las ediciones relacionadas con la calidad. Más adelante se define un modelo jerárquico de calidad (aunque otras maneras de categorizar la calidad pueden ser más apropiadas en circunstancias particulares y justificadas).

No es prácticamente posible medir todas las sub características internas y externas para todas las partes de un gran producto de software. De modo similar, no es generalmente práctico medir la calidad en el uso para todos los escenarios posibles de las tareas del usuario. Los recursos para la evaluación necesitan ser asignados entre los diversos tipos de medida, dependiente de los objetivos de la institución y de la naturaleza del producto y diseño de procesos.

1.4 Modelo de calidad para la calidad externa e interna

En esta sección se define el Modelo de Calidad para la calidad externa e interna a ser usado en las instituciones públicas. Se han establecido categorías para las cualidades de la calidad del software, basadas en seis características (funcionalidad, confiabilidad, utilidad, eficiencia, capacidad de mantenimiento y portabilidad), que se subdividen a su vez en sub características (Figura 3). Las sub características se pueden medir por métrica interna o externa.

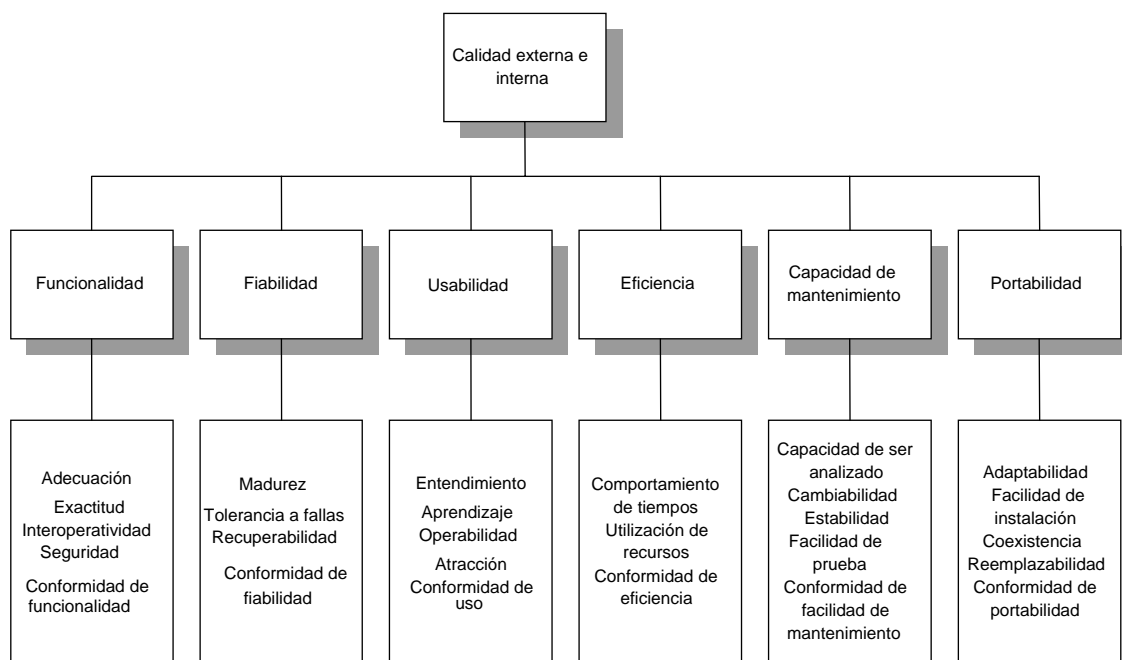


FIGURA 3 – Modelo de calidad para la calidad externa e interna

Las definiciones se dan para cada característica y sub característica de calidad del software que influye en la calidad. Para cada característica y sub característica, la capacidad del software es determinada por un conjunto de atributos internos que pueden ser medidos. Las características y sub características se pueden medir externamente por la capacidad provista por el sistema que contiene el software.

1.4.1 Funcionalidad

La capacidad del producto de software para proveer las funciones que satisfacen las necesidades explícitas e implícitas cuando el software se utiliza bajo condiciones específicas.

Esta característica se refiere a lo que hace el software para satisfacer necesidades, mientras que las otras características se refieren principalmente a cuándo y a cómo satisfacen las necesidades.

Para un sistema que es operado por un usuario, la combinación de la funcionalidad, fiabilidad, usabilidad y eficiencia puede ser medida externamente por su calidad en uso.

1.4.1.1 Adecuación

La capacidad del producto de software para proveer un adecuado conjunto de funciones para las tareas y objetivos especificados por el usuario.

Ejemplos de adecuación son la composición orientada a tareas de funciones a partir de sub funciones que las constituyen, y las capacidades de las tablas.



1.4.1.2 Exactitud

La capacidad del producto de software para proveer los resultados o efectos acordados con un grado necesario de precisión.

1.4.1.3 Interoperabilidad

La capacidad del producto de software de interactuar con uno o más sistemas especificados. La interoperabilidad se utiliza en lugar de compatibilidad para evitar una posible ambigüedad con la reemplazabilidad.

1.4.1.4 Seguridad

La capacidad del producto de software para proteger la información y los datos de modo que las personas o los sistemas no autorizados no puedan leerlos o modificarlos, y a las personas o sistemas autorizados no se les niegue el acceso a ellos.

La seguridad en un sentido amplio se define como característica de la calidad en uso, pues no se relaciona con el software solamente, sino con todo un sistema.

1.4.1.5 Conformidad de la funcionalidad

La capacidad del producto de software de adherirse a los estándares, convenciones o regulaciones legales y prescripciones similares referentes a la funcionalidad.

1.4.2 Fiabilidad

La capacidad del producto de software para mantener un nivel específico de funcionamiento cuando se está utilizando bajo condiciones especificadas.

El desgaste o envejecimiento no ocurre en el software. Las limitaciones en fiabilidad son debido a fallas en los requerimientos, diseño, e implementación. Las fallas debido a estos errores dependen de la manera en que se utiliza el producto de software y de las opciones del programa seleccionadas, más que del tiempo transcurrido.

La definición de fiabilidad en la ISO/IEC 2382-14:1997 es "la habilidad de la unidad funcional de realizar una función requerida...". En este documento, la funcionalidad es solamente una de las características de la calidad del software. Por lo tanto, la definición de la fiabilidad se ha ampliado a "mantener un nivel especificado del funcionamiento..." en vez de "...realizar una función requerida".

1.4.2.1 Madurez

La capacidad del producto de software para evitar fallas como resultado de errores en el software.



1.4.2.2 Tolerancia a errores

La capacidad del producto de software para mantener un nivel especificado de funcionamiento en caso de errores del software o de incumplimiento de su interfaz especificada.

El nivel especificado de funcionamiento puede incluir la falta de capacidad de seguridad.

1.4.2.3 Recuperabilidad

La capacidad del producto de software para restablecer un nivel especificado de funcionamiento y recuperar los datos afectados directamente en el caso de una falla.

Después de una falla, un producto de software a veces estará no disponible por cierto período del tiempo, intervalo en el cual se evaluará su recuperabilidad.

La disponibilidad es la capacidad del producto de software para poder realizar una función requerida en un punto dado en el tiempo, bajo condiciones indicadas de uso. En extremo, la disponibilidad se puede determinar por la proporción de tiempo total, durante la cual, el producto de software está en un estado ascendente. La disponibilidad, por lo tanto, es una combinación de madurez (con control de frecuencias de fallas), de la tolerancia de errores y de la recuperabilidad (que gobierna el intervalo de tiempo en cada falla). Por esta razón es que no ha sido incluida como una sub característica separada.

1.4.2.4 Conformidad de la fiabilidad

La capacidad del producto de software para adherirse a las normas, convenciones o regulaciones relativas a la fiabilidad.

1.4.3 Usabilidad

La capacidad del producto de software de ser entendido, aprendido, usado y atractivo al usuario, cuando es utilizado bajo las condiciones especificadas.

Algunos aspectos de funcionalidad, fiabilidad y eficiencia también afectarán la usabilidad, pero para los propósitos de la ISO/IEC 9126 ellos no son clasificados como usabilidad.

Los usuarios pueden ser operadores, usuarios finales y usuarios indirectos que están bajo la influencia o dependencia del uso del software. La usabilidad debe dirigirse a todo los diferentes ambientes de usuarios que el software puede afectar, o estar relacionado con la preparación del uso y evaluación de los resultados.



1.4.3.1 Entendimiento

La capacidad del producto de software para permitir al usuario entender si el software es adecuado, y cómo puede ser utilizado para las tareas y las condiciones particulares de la aplicación.

Esto dependerá de la documentación y de las impresiones iniciales dadas por el software.

1.4.3.2 Aprendizaje

La capacidad del producto de software para permitir al usuario aprender su aplicación. Un aspecto importante a considerar aquí es la documentación del software.

1.4.3.3 Operabilidad

La capacidad del producto de software para permitir al usuario operarlo y controlarlo.

Los aspectos de propiedad, de cambio, de adaptabilidad y de instalación pueden afectar la operabilidad.

La operabilidad corresponde a la controlabilidad, a la tolerancia a errores y a la conformidad con las expectativas del usuario.

Para un sistema que es operado por un usuario, la combinación de la funcionalidad, confiabilidad, usabilidad y eficacia puede ser una medida considerada por la calidad en uso.

1.4.3.4 Atracción

La capacidad del producto de software de ser atractivo al usuario.

Esto se refiere a las cualidades del software para hacer el software más atractivo al usuario, tal como el uso del color y la naturaleza del diseño gráfico.

1.4.3.5 Conformidad de uso

La capacidad del producto de software para adherirse a los estándares, convenciones, guías de estilo o regulaciones relacionadas a su usabilidad.

1.4.4 Eficiencia

La capacidad del producto de software para proveer un desempeño adecuado, de acuerdo a la cantidad de recursos utilizados y bajo las condiciones planteadas.

Los recursos pueden incluir otros productos de software, la configuración de hardware y software del sistema, y materiales (Ej: Papel de impresión o diskettes).



Para un sistema operado por usuarios, la combinación de funcionalidad, fiabilidad, usabilidad y eficiencia pueden ser medidas externamente por medio de la calidad en uso.

1.4.4.1 Comportamiento de tiempos

La capacidad del producto de software para proveer tiempos adecuados de respuesta y procesamiento, y ratios de rendimiento cuando realiza su función bajo las condiciones establecidas.

1.4.4.2 Utilización de recursos

La capacidad del producto de software para utilizar cantidades y tipos adecuados de recursos cuando este funciona bajo las condiciones establecidas.

Los recursos humanos están incluidos dentro del concepto de productividad.

1.4.4.3 Conformidad de eficiencia

La capacidad del producto de software para adherirse a estándares o convenciones relacionados a la eficiencia.

1.4.5 Capacidad de mantenimiento

Capacidad del producto de software para ser modificado. Las modificaciones pueden incluir correcciones, mejoras o adaptación del software a cambios en el entorno, y especificaciones de requerimientos funcionales.

1.4.5.1 Capacidad de ser analizado

La capacidad del producto de software para atenerse a diagnósticos de deficiencias o causas de fallas en el software o la identificación de las partes a ser modificadas.

1.4.5.2 Cambiabilidad

La capacidad del software para permitir que una determinada modificación sea implementada.

Implementación incluye codificación, diseño y documentación de cambios.

Si el software va a ser modificado por el usuario final, la cambiabilidad podría afectar la operabilidad.

1.4.5.3 Estabilidad

La capacidad del producto de software para evitar efectos inesperados debido a modificaciones del software.



1.4.5.4 Facilidad de prueba

La capacidad del software para permitir que las modificaciones sean validadas.

1.4.5.5 Conformidad de facilidad de mantenimiento

La capacidad del software para adherirse a estándares o convenciones relativas a la facilidad de mantenimiento.

1.4.6 Portabilidad

La capacidad del software para ser trasladado de un entorno a otro. El entorno puede incluir entornos organizacionales, de hardware o de software.

1.4.6.1 Adaptabilidad

La capacidad del producto de software para ser adaptado a diferentes entornos especificados sin aplicar acciones o medios diferentes de los previstos para el propósito del software considerado.

Adaptabilidad incluye la escalabilidad de capacidad interna (Ejemplo: Campos en pantalla, tablas, volúmenes de transacciones, formatos de reporte, etc.).

Si el software va a ser adaptado por el usuario final, la adaptabilidad corresponde a la conveniencia de la individualización, y podría afectar la operabilidad.

1.4.6.2 Facilidad de instalación

La capacidad del producto de software para ser instalado en un ambiente especificado.

Si el software va a ser instalado por el usuario final, puede afectar la propiedad y operatividad resultantes.

1.4.6.3 Coexistencia

La capacidad del producto de software para coexistir con otros productos de software independientes dentro de un mismo entorno, compartiendo recursos comunes.

1.4.6.4 Reemplazabilidad

La capacidad del producto de software para ser utilizado en lugar de otro producto de software, para el mismo propósito y en el mismo entorno.

Por ejemplo, la reemplazabilidad de una nueva versión de un producto de software es importante para el usuario cuando dicho producto de software es actualizado (actualizaciones, upgrades).



Reemplazabilidad se utiliza en lugar de compatibilidad de manera que se evitan posibles ambigüedades con la interoperabilidad.

La reemplazabilidad puede incluir atributos de ambos, inestabilidad y adaptabilidad. El concepto ha sido introducido como una sub característica por sí misma, dada su importancia.

1.4.6.5 Conformidad de portabilidad

La capacidad del software para adherirse a estándares o convenciones relacionados a la portabilidad.

1.5 MODELO DE CALIDAD PARA LA CALIDAD EN USO

En esta parte se define el modelo de calidad para la calidad en uso. Los atributos de la calidad en uso están categorizados en cuatro características: eficacia, productividad, seguridad y satisfacción (Figura 4).

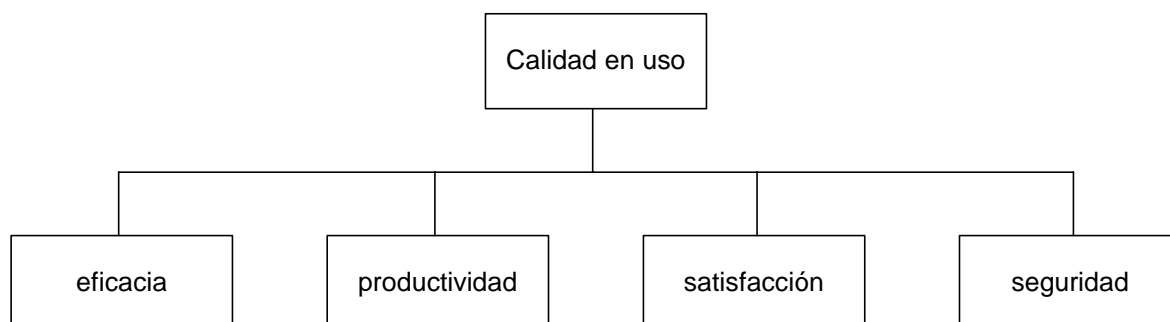


FIGURA 4 – Modelo de calidad para la calidad en uso

La calidad en uso es la visión de calidad del usuario. Alcanzar la calidad en uso depende de alcanzar la calidad externa necesaria que a su vez depende de alcanzar la calidad interna necesaria (Figura 1)

Las medidas son normalmente requeridas en tres niveles: interno, externo y de uso. Encontrar criterios para las medidas internas, no es normalmente suficiente para asegurar el logro de criterios para las medidas externas, y encontrar criterios para las medidas externas, no es normalmente suficiente para asegurar el logro de criterios para la calidad en uso.



1.5.1 Calidad en uso

La capacidad del producto de software para permitirles a usuarios específicos lograr las metas propuestas con eficacia, productividad, seguridad y satisfacción, en contextos especificados de uso.

Calidad en uso es la visión de calidad del usuario de un entorno que contiene el software, y es medida a partir de los resultados de usar el software en el entorno, más que por las propiedades del software mismo.

1.5.1.1 Eficacia

La capacidad del producto de software para permitir a los usuarios lograr las metas especificadas con exactitud e integridad, en un contexto especificado de uso.

1.5.1.2 Productividad

La capacidad del producto de software para permitir a los usuarios emplear cantidades apropiadas de recursos, en relación a la eficacia lograda en un contexto especificado de uso.

Los recursos relevantes pueden incluir: tiempo para completar la tarea, esfuerzo del usuario, materiales o costo financiero.

1.5.1.3 Seguridad

La capacidad del producto de software para lograr niveles aceptables de riesgo de daño a las personas, institución, software, propiedad (licencias, contratos de uso de software) o entorno, en un contexto especificado de uso.

Los riesgos son normalmente el resultado de deficiencias en la funcionalidad (incluyendo seguridad), fiabilidad, usabilidad o facilidad de mantenimiento.

1.5.1.4 Satisfacción

La capacidad del producto de software para satisfacer a los usuarios en un contexto especificado de uso.

La satisfacción es la respuesta del usuario a la interacción con el producto, e incluye las actitudes hacia el uso del producto.

PARTE 2: MÉTRICAS

2.1 Atributos Internos y Externos

Los niveles de ciertos atributos internos se han encontrado para influir en los niveles de algunos atributos externos, de modo que haya un aspecto externo y un aspecto interno en la mayoría de las características. Por ejemplo, la confiabilidad puede ser medida externamente observando el número de fallas en un período dado del tiempo de ejecución durante un ensayo del software, e internamente examinando las especificaciones detalladas y el código fuente para determinar el nivel de la tolerancia de falla. Los atributos internos serían los indicadores de los atributos externos.

Un atributo interno puede influenciar a una o más características, y una característica puede estar influenciada por más de un atributo (ver Figura 5). En este modelo la totalidad de atributos de la calidad del producto de software se clasifica en una estructura arborescente jerárquica de características y de sub características. El nivel más alto de esta estructura consiste en características de calidad y el nivel más bajo consiste en atributos de calidad de software. La jerarquía no es perfecta cuando algunos atributos pueden contribuir a más de una sub característica.

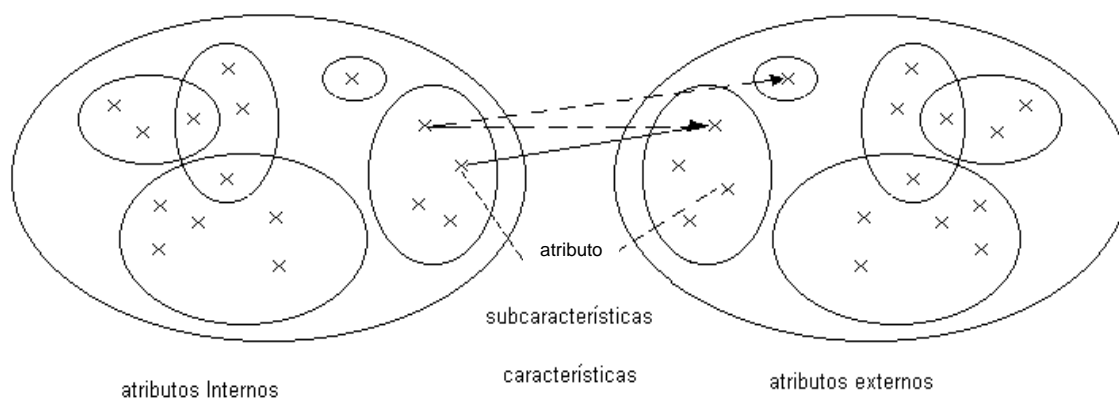


Figura 5 – Características de calidad, sub características y atributos

La sub característica puede medirse por la métrica interna o por la métrica externa.

La correlación entre los atributos internos y las medidas externas nunca es perfecta, y el efecto que un atributo interno dado tiene en una medida externa asociada, será determinado por la experiencia, y dependerá del contexto particular en que el software es usado.

De la misma manera, las propiedades externas (como la conveniencia, exactitud, tolerancia a fallas o tiempos de ejecución) influirán en la calidad observada. Una falla en la calidad del uso (por ejemplo: el usuario no puede completar la tarea) puede



remontarse a los atributos de calidad externa (por ejemplo: conveniencia u operabilidad) y los atributos internos asociados tienen que ser cambiados.

2.2 Métrica interna

La métrica interna puede ser aplicada a un producto de software no-ejecutable (como una especificación o código fuente) durante el diseño y la codificación. En el desarrollo de un producto de software, los productos intermedios deben ser evaluados usando métricas internas que permitan medir las propiedades intrínsecas, incluyendo aquellas que pueden derivarse de comportamientos simulados. El propósito primario de esta métrica interna es asegurar que se logre la calidad externa y la calidad de uso requerida. La métrica interna proporciona a los usuarios, evaluadores, verificadores y desarrolladores el beneficio de que puedan evaluar la calidad del producto de software y lo referido a problemas de calidad antes que el producto de software sea puesto en ejecución.

Las métricas internas miden atributos internos o indican los atributos externos, a través del análisis de las propiedades estáticas de productos intermedios o entregables del software. Las medidas de las métricas internas usan números o frecuencias de elementos de composición de software, los cuales aparecen, por ejemplo, en las sentencias de código de fuente, control de gráficos, flujo de datos y estados de representación de procesos.

2.3 Métrica externa

Las métricas externas usan medidas de un producto de software, derivadas del comportamiento del mismo, a través de la prueba, operación y observación del software. Antes de adquirir o usar un producto de software, éste debe ser evaluado usando las métricas basadas en los objetivos del área usuaria de la institución relacionados al uso, explotación y dirección del producto, considerando la organización y el ambiente técnico. La métrica externa proporciona a los usuarios, evaluadores, verificadores y desarrolladores, el beneficio de que puedan evaluar la calidad del producto de software durante las pruebas o el funcionamiento.

2.4 Relación entre las métricas internas y externas

Cuando los requisitos de calidad del producto de software son definidos, se listan las características o sub características de calidad del producto de software que contribuyen a dichos requisitos. Entonces, las métricas externas apropiadas y los rangos aceptables son especificados para cuantificar el criterio de calidad que valida que el software satisface las necesidades del usuario. Luego, los atributos de calidad interna del software se definen y especifican para planear y finalmente lograr la calidad externa y calidad en el uso requeridas, para construirlos durante el desarrollo del producto.

Apropiadas métricas internas y rangos aceptables son especificados para cuantificar los atributos de calidad interna, así ellos pueden usarse para verificar que el software intermedio reúne las especificaciones de calidad interna durante el desarrollo.



Se recomienda que las métricas internas que se usen tengan en lo posible una fuerte relación con la métrica externa diseñada, para que ellas puedan ser usadas para predecir los valores de las métricas externas. Sin embargo, es generalmente difícil diseñar un modelo teórico riguroso que proporcione una relación fuerte entre la métrica interna y la externa.

2.5 Calidad en el uso de métricas

La calidad en el uso de métricas mide la extensión de un producto que reúne las necesidades especificadas por los usuarios para lograr las metas propuestas, con la efectividad, productividad, seguridad y satisfacción en un contexto de uso específico. La evaluación de la calidad en uso valida la calidad del producto de software en los escenarios específicos de tareas de usuario.

La calidad en el uso es la vista del usuario sobre la calidad que el sistema de software contiene y es medida en términos de resultados de uso del software, en lugar de las propiedades del propio software. La calidad en el uso es el efecto combinado de calidad interna y externa para el usuario.

La relación de calidad en el uso con otras características de calidad del producto de software depende del tipo de usuario:

- El usuario final para quien la calidad en el uso es principalmente un resultado de funcionalidad, fiabilidad, utilidad y eficacia.
- La persona que mantiene el software para quien la calidad en el uso es un resultado del mantenimiento.
- La persona que hace portable el software para quien la calidad en el uso es un resultado de portabilidad.

2.6 Opción de métrica y criterio de medidas

La base en que las métricas son seleccionadas dependerá de las metas de la institución para el producto y las necesidades del evaluador. Las necesidades son especificadas por un criterio de medidas. El modelo en esta parte soporta una variedad de requisitos de evaluación, por ejemplo:

- Un usuario o un área de la institución, podría evaluar la conveniencia de un producto de software usando las métricas de calidad en el uso.
- Una institución que adquiere, podría evaluar un producto de software contra un criterio de valores de medidas externas de funcionalidad, fiabilidad, utilidad y eficacia, o de calidad en el uso.
- Un responsable de mantenimiento, podría evaluar un producto de software usando métricas para mantenimiento.
- Una persona responsable de la implementación del software en diferentes ambientes, podría evaluar un producto de software usando métricas de portabilidad.
- Un desarrollador, podría evaluar un producto de software contra criterios de



valores usando medidas internas de cualquiera de las características de calidad.

2.7 Métricas usadas para la comparación

Al informar los resultados del uso de métricas cuantitativas para hacer las comparaciones entre los productos, el informe mostrará si las métricas son objetivas o empíricas, usando valores conocidos y reproducibles.

Las comparaciones fiables entre los productos sólo se pueden hacer cuando se usan métricas rigurosas. Los procedimientos de medición deben medir las características (o sub características) de calidad del producto de software. Estos exigen ser medidos con suficiente exactitud para permitir asignar los criterios y hacer las comparaciones.

La concesión debe hacerse para posibles errores de medición causados por herramientas de medida o errores humanos.

La métrica usada para las comparaciones debe ser válida y suficientemente exacta para permitir hacer comparaciones fiables. Esto significa que las medidas deben ser objetivas, empíricas, usando una escala válida, y reproducibles.

- Para ser objetivo, habrá un procedimiento escrito y convenido para asignar el número o categoría al atributo del producto.
- Para ser empírico, los datos serán obtenidos de la observación o de un cuestionario psicométricamente válido.
- Para utilizar una escala válida, los datos deberán estar basados en ítems de igual valor o de un valor conocido. Si una lista de comprobación se utiliza para proporcionar datos, los ítems deben, si es necesario, ser ponderados.
- Para ser reproducible, el proceso para medir debería producir las mismas medidas (dentro de las tolerancias apropiadas) que son obtenidas por diferentes personas haciendo la misma medición del producto de software en diferentes ocasiones.

Las métricas internas deberían también tener valor predictivo, esto es, ellas deben correlacionarse con algunas medidas externas deseadas. Por ejemplo, una medida interna de un atributo particular del software debería tener correlación con cierto aspecto medible de calidad cuando se utiliza el software. Es importante que los valores asignados a las mediciones coincidan con las expectativas normales. Por ejemplo, si la medición sugiere que el producto es de alta calidad, entonces ésta debería ser consistente con el producto, satisfaciendo las necesidades de un usuario.



PARTE 3: PROCESO DE EVALUACIÓN DE SOFTWARE

Todo proceso de evaluación de software deberá partir de una evaluación cualitativa y derivar en una evaluación cuantitativa, siendo todo el proceso documentado, cumpliendo los siguientes pasos:

3.1 Establecer el propósito de la evaluación:

Productos intermedios:

- Decidir sobre la aceptación de un producto intermedio de un subcontratista o proveedor.
- Decidir cuándo un proceso está completo y cuando remitir los productos al siguiente proceso.
- Predecir o estimar la calidad del producto final.
- Recoger información con objeto de controlar y gestionar el proceso.
- Otros con justificación.

Producto final:

- Decidir sobre la aceptación del producto.
- Decidir cuando publicar el producto.
- Comparar el producto con otros productos competitivos.
- Seleccionar un producto entre productos alternativos.
- Valorar tanto el aspecto positivo, como el negativo, cuando está en uso.
- Decidir cuando mejorar o reemplazar un producto.
- Otros con justificación.

3.2 Identificar el tipo de producto

Especificar el tipo de producto a evaluar, si es un sistema operativo, software de seguridad, software de ofimática, lenguaje de programación, base de datos, aplicativo desarrollado, ERP, entre otros. Asimismo, se deberá establecer su relación con Estándares de Tecnologías de Información y Comunicaciones que utiliza la Institución; y asegurar la legalidad del producto.

3.3 Especificar el Modelo de Calidad

Se elaborará de acuerdo a lo establecido en la Parte I, y deberá ser aprobado por el Jefe de Informática o quien haga sus veces.

3.4 Seleccionar métricas

La selección de métricas se obtiene a partir de los atributos especificados en el Modelo de Calidad. Se agruparán en:



- Métricas internas.
- Métricas externas.
- Métricas de uso.

3.5 Establecer niveles, escalas para las métricas

- El área de informática aplicará el tipo de escala de proporción.
- A cada métrica seleccionada le asignará un puntaje máximo de referencia.
- La suma de los puntajes máximos de todas las métricas deberá ser igual a 100 puntos.
- El área de informática podrá establecer niveles de calificación cualitativa en base a los puntajes como por ejemplo:
 - Puntaje mínimo de aprobación.
 - Inaceptable, mínimo aceptable, rango objeto, excede los requisitos.
 - Insatisfactorio, satisfactorio.
- Se pueden usar números hasta con un decimal de aproximación. (Ejemplos: 4.1, 3.8, 11.7).
- El área de informática podrá establecer, por cada métrica, un puntaje mínimo de aprobación. En caso no se alcance ese puntaje, se considerará que el producto de software no cumple con las necesidades de información de la institución y será rechazado.

3.6 Establecer criterios de valoración

El área de informática elaborará sus procedimientos, con criterios distintos para diferentes características de calidad, cada uno puede estar expresado en términos de sub características individuales, o una combinación ponderada de ellas. El procedimiento puede incluir otros aspectos como el tiempo y costo que contribuyen a la estimación de la calidad de un producto de software en un entorno concreto.

3.7 Tomar medidas

Para la medición, las métricas seleccionadas se aplican al producto de software. Los resultados son valores expresados en las escalas de las métricas, definidos previamente.



3.8 Comparar con los criterios

En el paso de puntuación, el valor medido se compara con los criterios predeterminados.

Se debe elaborar un cuadro de resultados, como el que se aprecia a continuación.

	PUNTAJE MAX.	SOFT. 1	SOFT. 2	SOFT.n
Atributos internos (Ai) <ul style="list-style-type: none"> • Ai1 • Ai2 • . • . • Ain 	PMax. Ai1 PMax. Ai2 . . PMax Ain				
Atributos externos (Ae) <ul style="list-style-type: none"> • Ae1 • Ae2 • . • . • Aen 	PMax Ae1 PMax Ae2 . . PMax Aen				
Atributos de uso (Au) <ul style="list-style-type: none"> • Au1 • Au2 • . • . • Aun 	PMax Au1 PMax Au2 . . PMax Aun				
PUNTAJE TOTAL	100.0				

3.9 Valorar resultados

La valoración, que resume un conjunto de niveles calificados, es el paso final del proceso de evaluación del software.

3.10 Documentación

Todo el proceso de evaluación debe estar documentado, indicando nombres y apellidos, cargos, procedencia de las personas que participaron en el proceso de evaluación, especificando las etapas en las que participaron, si es necesario. Este documento deberá ser aprobado por el Jefe de Informática o quien haga sus veces.



GLOSARIO DE TÉRMINOS

Adquiriente

Una organización que adquiere u obtiene un sistema, producto de software o servicio software de un proveedor.

Atributo

Una característica física o abstracta mensurable de una entidad. Los atributos pueden ser internos o externos.

Calidad

Son todas las características de una entidad que forman parte de su habilidad para satisfacer las necesidades propias e implícitas.

Calidad en el empleo

Es la medida en que un producto empleado por usuarios específicos satisface sus necesidades con efectividad, productividad y entera satisfacción para alcanzar objetivos o metas en contextos específicos de su empleo.

Calidad externa

La extensión para la cual un producto satisface necesidades explícitas e implícitas cuando es usado bajo condiciones específicas.

Calidad interna

Es la totalidad de atributos del producto que determinan su habilidad para satisfacer las necesidades propias e implícitas bajo condiciones específicas.

Calificación

La acción de evaluar el valor medido al nivel de calificación adecuado. Utilizado para determinar el nivel de calificación asociado con el software para una característica específica de calidad.

Defecto

Un paso, proceso o definición de dato incorrecto en un programa de computadora.

Desarrollador

Una organización que realiza actividades de desarrollo (incluyendo análisis de los requisitos, diseño y pruebas de aceptación) durante el proceso del ciclo de vida del software.

Escala

Un conjunto de valores con propiedades definidas

Ejemplos de tipos de escalas son: una escala nominal que corresponda a un conjunto de categorías; una escala ordinal que corresponda a un conjunto ordenado de puntos; una escala de intervalo que corresponda a una escala ordenada con puntos equidistantes; y una escala de ratios que no sólo tiene puntos equidistantes sino que



posee el cero absoluto. Las métricas utilizando escalas nominales u ordinales producen datos cualitativos, y las métricas utilizando escalas de intervalos o ratios producen datos cuantitativos.

Falla

La terminación de la capacidad de un producto de realizar una función requerida o su incapacidad para realizarla dentro de límites previamente especificados.

Firmware

El firmware contiene las instrucciones e información acerca del funcionamiento de un dispositivo o hardware, generalmente grabado en un chip. Es el código que rige el comportamiento del mismo.

Indicador

Una medida que se puede utilizar para estimar o para predecir otra medida. Los indicadores pueden emplearse para evaluar los atributos cualitativos del software y para calcular los atributos del proceso de desarrollo. Ambos son valores indirectos e imprecisos de los atributos.

Medición

Actividad que usa la definición de la métrica para producir el valor de una medida.

Medida

Número o categoría asignada a un atributo de una entidad mediante una medición.

Medida directa

Una medida de un atributo que no depende de la medida de ningún otro atributo.

Métrica

Es un método definido de valoración y su escala de valoración.

Las métricas pueden ser internas o externas, directas o indirectas.

Las métricas incluyen métodos para clasificar la data o información cualitativa en diferentes categorías.

Medida externa

Una medida indirecta de un producto derivada de las medidas del comportamiento del sistema del que es parte.

El sistema incluye cualquier hardware, software (ya sea software a medida o software tipo paquete) y usuarios.

El número de fallas encontradas durante las pruebas es una medida externa del número de fallas en el programa, porque el número de fallas es contado durante la operación del programa corriendo en un sistema de cómputo.

Las medidas externas pueden ser usadas para evaluar los atributos de calidad cercanos a los objetivos finales de diseño.

Modelo cualitativo

Es una serie de características y la relación entre las mismas, que conforman la base de los requerimientos cualitativos específicos y la valoración cualitativa.



Módulo de evaluación

Un paquete de tecnología de evaluación para una característica o sub característica de calidad de un software específico. El paquete incluye métodos y técnicas de evaluación, entradas a ser evaluadas, datos a ser medidos y recopilados y procedimientos y herramientas de soporte.

Necesidades implícitas

Necesidades que pueden no haber sido especificadas pero que son necesidades reales cuando la entidad es usada en condiciones particulares.

Necesidades implícitas son necesidades reales, las cuales pueden no haber sido documentadas.

Nivel de calificación

Un punto en la escala ordinal que es utilizado para categorizar una escala de medida. El nivel de calificación habilita al software para ser clasificado de acuerdo con las necesidades explícitas o implícitas. Los niveles de clasificación adecuados pueden ser asociados con las vistas diferentes de calidad, por ejemplo, usuarios, gerentes o desarrolladores.

Producto de software

El conjunto de programas de cómputo, procedimientos, y posible documentación y datos asociados.

Los productos incluyen productos intermedios y productos para los usuarios, como los desarrolladores y personal de soporte.

Producto de software intermedio

Es un producto del proceso de desarrollo del software que se emplea para alimentar una etapa diferente del proceso de desarrollo.

En algunos casos, un producto intermedio puede ser también un producto final.

Proveedor

Una organización que entra a un contrato con el adquiriente para el suministro de un sistema, producto de software o servicio de software bajo los términos de dicho contrato.

Servicio

Es una organización que presta servicios de mantenimiento.

Sistema

Una composición integrada que consiste en uno o más procesos, hardware, software, instalaciones y personas, que proveen una capacidad para satisfacer una necesidad establecida o un objetivo.

Software

Todo o parte de los programas, procedimientos, reglas y documentación asociada a un sistema de procesamiento de información.

El software es una creación intelectual que es independiente del medio en el cual fue grabado.



Usuario

Un individuo que utiliza el producto de software para realizar una función específica. Los usuarios pueden incluir operadores, receptores de los resultados del software, desarrolladores o personal de soporte de software.

Valoración

Emplear una métrica para asignar uno de los valores de una escala (el mismo que puede ser un número o categoría) al atributo de una entidad.

La valoración puede ser cualitativa cuando se emplean categorías. Por ejemplo, algunos de los atributos importantes de los productos de software, tales como el lenguaje del programa base (ADA, C, COBOL, etc.) son categorías cualitativas.

Valoración indirecta

Es la valoración de un atributo derivada del valor de uno o más atributos diferentes. La valoración externa de un atributo de un sistema de cómputo (tal como el tiempo de respuesta a la información alimentada por el usuario) es una valoración indirecta de los atributos del software, dado que esta medida se verá influenciada por los atributos del entorno de cómputo, así como por los atributos propios del software.

Valoración interna

Es una valoración del producto en sí, ya sea directa o indirecta.

El número de líneas del código, las valoraciones de complejidad, el número de fallas encontradas durante el proceso y el índice de señales o alertas, son todas las valoraciones internas propias del producto en sí.

Valorar (verbo)

Realizar una valoración o estimación.

Valor (sustantivo)

Es el número o categoría que una entidad le asigna a un atributo al efectuar la valoración.

Valoración Cualitativa

Es una evaluación sistemática del grado o capacidad de una entidad para satisfacer necesidades o requerimientos específicos.

Dichos requerimientos pueden ser formalmente especificados, por ejemplo, por el área de desarrollo de sistemas, cuando el producto se diseña por contrato para un usuario específico, cuando el producto es desarrollado sin un usuario específico, o bien que se trate de necesidades más generales, como cuando un usuario evalúa los productos con propósitos de comparación y selección.

Validación

Confirmación por inspección y provisión de evidencia objetiva de que los requerimientos particulares para un uso específico son alcanzados.

En diseño y desarrollo, la validación está relacionada con el proceso de reexaminación de un producto para determinar la conformidad con las necesidades del usuario.

La validación es realizada normalmente sobre el producto final bajo condiciones operacionales definidas. Puede ser necesaria en las fases iniciales.

“Validado” es utilizado para designar el estado correspondiente.



Verificación

Confirmación por examen y provisión de evidencia objetiva que los requerimientos específicos han sido alcanzados.

En diseño y desarrollo, la verificación está relacionada con el proceso de examinar el resultado de una actividad dada para determinar su conformidad con los requerimientos definidos para dicha actividad.

“Verificado” es utilizado para designar el estado correspondiente.



BIBLIOGRAFÍA

- Norma ISO/IEC 9126-1: 2001 - Software engineering -- Product quality -- Part 1: Quality model.
- Norma ISO/IEC TR 9126-2: 2003 - Software engineering -- Product quality -- Part 2: External metrics.
- Norma ISO/IEC TR 9126-3: 2003 - Software engineering -- Product quality -- Part 3: Internal metrics.
- Norma ISO/IEC 14598-1:1999 - Part 1: General overview.
- Norma ISO/IEC 14598-2:2000 - Part 2: Planning and management.
- Norma ISO/IEC 14598-3:2000 - Part 3: Process for developers.
- Norma ISO/IEC 14598-5:1998 - Part 5: Process for evaluators.

Intercambio de modelos UML y OO-Method¹

Beatriz Marín¹, Giovanni Giachetti¹, Oscar Pastor¹

¹ Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n
46022 Valencia, España
{bmarin, ggiachetti, opastor}@dsic.upv.es

Resumen. Los modelos objetuales se han convertido en los actores principales en el proceso de desarrollo de software, provocando un auge en la aparición de herramientas CASE que permiten realizar este tipo de modelos, e incorporando distintas facilidades que diferencian a unas y otras. Este auge conlleva cada vez más la necesidad de intercambiar modelos entre las distintas herramientas para poder aprovechar estas facilidades. En particular, se quiere aprovechar las prestaciones de herramientas avanzadas, que posibilitan la compilación de modelos objetuales, generando de esta manera la aplicación final descrita en dichos modelos. En este contexto, este artículo presenta la correspondencia entre modelos objetuales UML y OO-Method –un método de producción automática de software a partir de modelos objetuales compatible con MDA- y una implementación que transforma automáticamente dichos modelos según la correspondencia establecida, que posibilita el intercambio de modelos, aprovechando principalmente la generación automática de aplicaciones mediante OO-Method a partir de modelos UML.

1 Introducción

Hoy en día, las investigaciones enfocadas al análisis y diseño de sistemas se centran cada vez menos en los objetos vistos como artefactos de programación, y más en los modelos conceptuales conformados por dichos objetos. No porque los objetos hayan perdido importancia, sino todo lo contrario. La orientación a objetos es una tecnología que se encuentra madura, lo que queda demostrado por la unificación de los lenguajes de modelado orientado a objetos en el estándar UML[7], que está globalmente aceptado y usado en los procesos de desarrollo de software, y que ha pasado del ámbito de la programación (espacio de la solución) al ámbito del modelado (espacio del problema).

Desde hace algunos años, los modelos son los que se encuentran en la cima de las tecnologías, puesto que además de la gran ventaja de reutilización de código de la orientación a objetos, permiten obtener ventajas de portabilidad de los sistemas, interoperabilidad con otros sistemas y facilidad de cambio del sistema. Debido a que

¹ Este trabajo ha sido desarrollado con el soporte del MEC bajo el proyecto DESTINO TIN2004-03534 y cofinanciado por FEDER.

los modelos son una representación del conocimiento que se tiene de alguna realidad, desde un punto de vista en particular, se pretende conseguir el sistema de software final a partir de las transformaciones automáticas de los modelos realizados.

En este sentido, la OMG ha dictado un estándar para el desarrollo de software llamado MDA [1] (Model Driven Architecture), el cual es un paradigma de desarrollo basado en separar la lógica de la arquitectura de la aplicación; representando cada parte con modelos, que mediante transformaciones dirigidas por metamodelos, se puede obtener el sistema final.

Por esta razón, actualmente en el mercado se tiene una serie de herramientas CASE que apoyan los procesos de desarrollo que utilizan el paradigma MDA, unas con más éxito que otras, ya que algunas permiten obtener a partir de los modelos el código de las clases, con sus atributos y cabeceras de servicios; y por otra parte, existe un conjunto más reducido de herramientas, que permite obtener el sistema final mediante transformaciones de los modelos. Dentro de este conjunto se encuentra Olivanova Modeler, que pertenece a la empresa CARE-Technologies [2] [3] [13], y que operacionaliza la conversión de un modelo objetual conceptual en el “código” de la aplicación final descrita en dicho modelo.

Además, dado que existen varias herramientas CASE que permiten realizar modelos objetuales, la OMG ha dictado un estándar para el intercambio de modelos entre las diferentes herramientas, repositorios, bases de datos y productos de software en general. Este estándar es XMI [4], XML Metadata Interchange, con lo cual los modelos pueden ser exportados desde una herramienta, importados en otra, almacenados en algún repositorio, tras pasados a algún formato, permiten generar código a partir de ellos, etc.

Cabe destacar que, para poder intercambiar cualquier tipo de modelos mediante XMI, sus meta-modelos deben tener elementos que se encuentren en la especificación MOF (Meta Object Facility), estándar dictado también por la OMG [5] [6]. MOF define los elementos que pueden componer los distintos meta-modelos; por lo que, una vez que se han identificado los elementos de los modelos en una herramienta origen y destino, se pueden intercambiar los modelos entre las herramientas según sean las relaciones de sus elementos.

En este contexto tecnológico, el problema que aborda este trabajo es la elaboración de un proceso que transforma los modelos UML construidos con herramientas CASE de propósito general, en los modelos conceptuales concretos requeridos por un compilador de modelos OO-Method y viceversa. La contribución principal aportada es demostrar que estas estrategias basadas en estándares tienen una aplicación práctica concreta y una utilidad evidente cuando las componentes tecnológicas se acoplan adecuadamente.

Particularmente, se considerará el diagrama de clases como el modelo conductor del análisis y diseño de un sistema, enfocando la propuesta de este trabajo al intercambio de modelos de clases entre herramientas que soporten UML [7] y OO-Method [8][9], por ejemplo, entre Poseidon y Olivanova Modeler.

Este artículo está organizado en 5 secciones. Después de esta introducción, la sección 2 describe las correspondencias entre los elementos UML y OO-Method, la sección 3 muestra una la estrategia de implementación para el intercambio de modelos, la sección 4 describe un ejemplo de aplicación de dicha estrategia, y finalmente, la sección 5 relata las conclusiones y los trabajos futuros.

2 Modelos UML y OO-Method

Antes de identificar la correspondencia entre los elementos de los modelos de clases UML y OO-Method, primero se debe identificar cada uno de los elementos, relevantes para el intercambio, que pueden componer dichos modelos.

2.1 Elementos UML

El diagrama de clases UML refleja básicamente clases y relaciones entre clases [12]. A continuación se lista un conjunto de elementos de modelado relevantes para el intercambio de modelos, asociados al diagrama de clases: Clases, que contienen Atributos y Operaciones; Asociaciones de Clases, que pueden ser Asociación, Agregación, Composición o Generalización; Actores; Interfaces y Paquetes.

2.2 Elementos OO-Method

Al igual que el diagrama de clases UML, el diagrama de clases OO-Method también refleja clases y relaciones entre clases. Los elementos relevantes para el intercambio que se pueden utilizar en este diagrama son los siguientes: Clases, que contienen Atributos, Eventos, Transacciones y Operaciones; Asociaciones de Clases, que pueden ser Asociación, Agregación, Composición o Generalización; Agentes y Clusters.

2.3 Correspondencia entre elementos UML y OO-Method

Se debe tener en cuenta que, OO-Method utiliza un subconjunto de los elementos pertenecientes a UML, los cuales permiten especificar completamente un sistema organizacional mediante un lenguaje formal de especificación OASIS [10], que no presenta heterogeneidad de interpretación semántica, a diferencia de lo que ocurre con las interpretaciones más informales asociadas a la especificación de modelos UML.

Por medio de este lenguaje formal, es posible obtener el sistema final listo para compilar y ejecutar. Por esta razón, cada elemento de OO-Method se ha analizado detalladamente para establecer la correspondencia con los elementos del diagrama UML, reduciendo al máximo la brecha señalada anteriormente, dada por el uso del lenguaje natural en UML y OASIS en OO-Method.

- Correspondencia de clases:

Una clase describe un conjunto de objetos que comparten las mismas especificaciones de características, restricciones y semántica. Tanto las clases UML como las clases OO-Method conciernen a esta descripción, por lo que la correspondencia entre ellas es directa, es decir, el nombre de una clase UML corresponderá al nombre de una clase OO-Method y los comentarios de una clase UML corresponderán a los comentarios de una

clase OO-Method. En esta línea, los elementos contenidos en una clase UML serán también los elementos contenidos en una clase OO-Method, pero las correspondencias entre estos elementos serán detalladas más adelante.

Existen algunas clases de OO-Method que se comportan de manera especial, ya que pueden ejecutar servicios de otras clases. Estas clases son conocidas como agentes, que además de ser clases del modelo de clases UML, serán actores del diagrama de casos de uso, conservando el nombre de la clase e interactuando con los servicios que ejecuta.

En el sentido contrario, los actores de UML que se encuentren en el modelo de clases, serán clases agentes en OO-Method; y si están relacionados con clases, entonces podrán ejecutar todos los servicios de la clase que tiene relacionada.

- Correspondencia de atributos:

Los atributos de una clase, ya sea UML u OO-Method, representan características de ella, por lo que su correspondencia es directa, es decir, el nombre de un atributo UML será el nombre del atributo OO-Method y la documentación del atributo UML será el comentario del atributo OO-Method.

En cuanto al tipo de dato de los atributos, UML permite definir los tipos de datos que se utilizarán en un proyecto, pudiendo ser simples, complejos, objetuales, etc.; en cambio OO-Method tiene un conjunto predefinido de tipos de datos simples para los atributos y los tipos de datos objetuales son representados mediante la correspondiente asociación entre clases. Cada herramienta CASE que permite realizar modelos UML, tiene un conjunto predefinido de tipos de datos, además de entregar la posibilidad de crear otros tipos de datos, por lo que, teniendo en cuenta la herramienta desde la que se quiere transformar un modelo y su conjunto predefinido de tipos de datos, se realizará la equivalencia uno a uno por cada tipo de dato; y para el caso en que no exista equivalencia, el tipo de dato en OO-Method se asumirá como *String*. A pesar de que UML y OO-Method acepten tipos de datos abiertos, las herramientas que los soportan fijan un subconjunto predefinido de tipos de datos para las diferentes características de los modelos. A modo de ejemplificar las equivalencias entre los subconjuntos predefinidos de los tipos de datos de las herramientas que soportan UML y OO-Method, a continuación se muestra una tabla que detalla los tipos de datos predefinidos en la herramienta UML Poseidon y la herramienta OO-Method Olivanova Modeler.

Tabla 1. Ejemplo de correspondencia entre tipos de datos UML y OO-Method

UML	OO-Method
Boolean	Bool
Byte	Int
Short	Int
Float	Real
Decimal	Real
Money	Real
Char	String
Time	DateTime
Currency	Real
Date	Date
Double	Real
Integer	Int
Otro tipo	String

Cada atributo tiene una propiedad que indica si el valor que toma puede cambiar en el futuro. Esta propiedad de los atributos es conocida como *changeability* en UML. Para expresar la misma semántica, en OO-Method se clasifican los atributos en constantes, variables y derivados. En este sentido, los atributos que no pueden cambiar su valor en el tiempo, corresponderán a los atributos constantes y los que sí pueden cambiar su valor, serán los atributos variables. UML no representa si un atributo es derivado en la propiedad *changeability*, sino que tiene una propiedad booleana para los atributos que indica si son derivados (*isDerived*); por lo que los atributos que tengan esta propiedad como verdadera corresponderán a atributos derivados en OO-Method.

Para el caso de los atributos derivados, OO-Method permite especificar su fórmula de derivación mediante el lenguaje formal OASIS; en UML podría usarse OCL [11] para especificar las fórmulas de derivación, pero en términos reales, la mayoría de los modelos UML no presentan las especificaciones OCL. Por esta razón, las fórmulas de derivación serán traspasadas como parte de la documentación de los atributos cuando se transforme un modelo OO-Method a uno UML.

El valor inicial de un atributo UML corresponde al valor que tomará el atributo cuando se crea un objeto de una clase. Esto es equivalente al valor por defecto de un atributo en OO-Method.

Para los atributos que pueden no tener valores asignados en un instante del tiempo, OO-Method permite asignarles la propiedad de que pueden ser nulos; en cambio en UML se especifica el número mínimo y máximo de valores que un atributo puede tener en un instante del tiempo. Esta especificación es la multiplicidad del atributo, y sólo cuando su mínimo es 0 significa que el atributo acepta nulos.

- Correspondencia de servicios:

Los servicios definen el comportamiento de los objetos de una clase. OO-Method distingue los eventos, las transacciones y las operaciones. Los eventos son servicios atómicos, indivisibles, que ocurren en un instante temporal y que pueden asignar valores a los atributos de las clases. Las transacciones son servicios que contemplan una secuencia de eventos y otras transacciones que sólo tienen dos formas de terminar: se ejecuta de manera completa o se deshace toda la ejecución. Las operaciones son servicios que detallan una secuencia de eventos, transacciones y otras operaciones; y que se ejecuta paso a paso desde el inicio al final, independientemente de si ha fallado en algún paso.

En UML, las operaciones no especifican la forma en que se consumirá el servicio, por lo que fue necesario decidir la correspondencia más cercana a los servicios en OO-Method. Para esto, se tiene en cuenta que el modelo UML que se quiere transformar es correcto y que no tiene servicios inventados por el diseñador, sino que los servicios especificados describen fehacientemente el comportamiento de los objetos.

Dado que las operaciones UML generalmente especifican el comportamiento de los objetos sin detallar qué atributos del objeto serán afectados por la realización del servicio, las operaciones UML no corresponderán a eventos de OO-method. Las operaciones modeladas en UML podrían ser más semejantes a las transacciones u operaciones de OO-Method; y como la semántica de las transacciones es más común que la semántica de las operaciones en el comportamiento de un sistema, ya que asegura que la base de datos del sistema es consistente; entonces las operaciones UML corresponderán a las transacciones OO-Method. En el sentido contrario, los eventos, transacciones y operaciones de OO-Method corresponderán a operaciones del diagrama de clases UML.

Cada servicio, ya sea de UML u OO-Method, conservará el nombre, documentación, argumentos de entrada y salida cuando sea transformado en un sentido y en otro.

Para el caso de los servicios OO-Method, se pueden especificar las fórmulas de evaluación de los atributos y las fórmulas de las transacciones y las operaciones mediante el lenguaje formal OASIS; sin embargo, tal como para los atributos derivados, la especificación OCL para los servicios no es muy utilizada en los modelos UML; por lo que serán concatenadas con la documentación de cada operación UML, según sea el caso.

Los servicios tienen una propiedad para indicar si pueden ser vistos desde los demás elementos del modelo: la visibilidad. En UML, esta propiedad

puede tomar los valores *public*, *private*, *protected* y *package*; y se puede interpretar como:

- Un elemento *public* es visible a todos los elementos que tienen acceso al espacio de nombres que lo posee.
- Un elemento *private* es sólo visible dentro del espacio de nombres que lo posee.
- Un elemento *protected* es visible a los elementos que tienen una relación de generalización con el espacio de nombres que lo posee.
- Un elemento *package* es visible desde los paquetes que se encuentran en el mismo espacio de nombres que lo posee; y sólo puede ser otorgado a elementos que no se encuentren dentro de paquetes.

Los servicios OO-Method permiten indicar de forma precisa su visibilidad, es decir, las clases que lo pueden ver, o más bien, sus agentes. Además, OO-Method permite indicar si un servicio no debe ser visto por ninguna clase agente mediante la propiedad *internal use*, con lo cual sólo será visto por la clase que lo contiene y sus relacionadas. Para realizar la correspondencia entre la visibilidad de los servicios UML y OO-Method, se ha tenido en cuenta esta propiedad, por lo que los servicios que tengan visibilidad *private* o *protected* en UML corresponderán a servicios con la propiedad *internal use* en OO-Method. Para los servicios que tengan visibilidad *public* o *package* en UML, una vez que se tenga el modelo en OO-Method, será necesario indicar detalladamente qué clase puede ver cada servicio. Si la transformación se quiere realizar en sentido inverso, es decir desde OO-Method a UML, todos los servicios OO-Method tendrán visibilidad *public*, a excepción de los que tengan la propiedad *internal use* que corresponderán a visibilidad *private*.

Si una operación UML tiene el estereotipo <<constructor>>, significa que al realizarla se creará una nueva instancia de la clase que contiene la operación. Esto es equivalente a que la transacción en OO-Method tenga el flag *New*. Lo mismo sucede con el estereotipo <<destructor>> en las operaciones UML, ya que corresponderán a transacciones OO-Method con el flag *Destructor*.

- Correspondencia de asociaciones:

Las asociaciones representan relaciones entre dos clasificadores, que pueden ser clases, actores, interfaces, etc. Las asociaciones entre dos clases UML corresponderán a asociaciones de clases OO-Method, conservando el nombre y la documentación.

Cada asociación tiene dos extremos, en cada uno de los cuales se conecta la asociación con las clases participantes en ella. El nombre de cada extremo de la asociación UML corresponderá al nombre de cada rol participante en la asociación OO-Method.

Además, OO-Method distingue las clases participantes en la asociación como componente y compuesta, según la independencia que tenga una clase de la otra, es decir, una clase será componente si en cualquier momento del tiempo pueden crearse o destruirse enlaces de las instancias de esa clase con las instancias de la clase que se encuentra en el otro extremo de la asociación; y una clase será compuesta cuando una vez creados los enlaces de las instancias de esa clase con las de la clase que se encuentra en el otro extremo, no pueden ser destruidos o no se puede crear un nuevo enlace cuando los objetos del otro extremo de la asociación ya han sido inicializados. Cada extremo de la asociación UML tiene una propiedad que indica su cambio en el tiempo, por lo que si la propiedad indica que no se puede cambiar (*frozen*), entonces ese extremo corresponderá a un extremo compuesto; y cuando dicha propiedad indique que si puede cambiar (*changeable*), entonces ese extremo de la asociación será componente en OO-Method.

Siguiendo con las propiedades de los extremos de las asociaciones, UML permite que ambos extremos sean *frozen*, lo cual implicaría que los objetos de las clases participantes en la asociación deberían ser creados en el mismo instante de tiempo; lo cual podría significar que se trata del mismo objeto y por lo tanto sería necesario revisar el modelo. Para el caso en que un extremo de la asociación sea *frozen* y el otro *changeable*, corresponderá a asociaciones estático-dinámicas en OO-Method. Para el caso en que ambos extremos de la asociación sean *changeable*, corresponderá a una asociación dinámica-dinámica de OO-Method.

Cuando se tienen asociaciones del tipo dinámica-dinámica, es necesario especificar los servicios que crearán y destruirán los enlaces entre las clases participantes de la asociación. Generalmente, esta especificación no se realiza en los modelos UML, que luego son codificados para obtener el sistema final, en cambio en OO-Method es necesario especificar estos servicios, ya que a partir de las especificaciones se puede generar el sistema de software por completo.

Tanto en UML como en OO-Method, cada extremo de la asociación posee una propiedad que indica la multiplicidad, es decir, cuántas instancias como mínimo y cuántas instancias como máximo pueden ser asociadas al otro extremo. Por esta razón, la multiplicidad de cada extremo de una asociación tiene una correspondencia directa entre asociaciones UML y OO-Method.

Dado que los actores presentes en el diagrama de clases UML corresponderán a clases agentes en OO-Method, las asociaciones entre actores o entre actores y clases serán tratadas de la misma manera que se ha detallado para las asociaciones entre clases.

Las asociaciones entre clases UML pueden especializarse en agregaciones o composiciones, las cuales son relaciones del tipo es-parte-de que se diferencian en que la composición incorpora dependencia entre las instancias de las clases relacionadas. En OO-Method esta especialización se realiza mediante el manejo de cardinalidades y dependencia de identificación. A modo de ejemplo, para representar una relación de agregación entre la clase A y la clase B, donde A es agregada a B, bastaría con definir una relación que posea cardinalidad máxima uno desde A hacia B. Mientras que una asociación de composición de A en B se representaría mediante una relación con cardinalidad mínima y máxima de uno desde A hacia B, incorporando adicionalmente la dependencia de identificación de la clase A; con lo cual la duración de cualquier instancia de A quedará determinada por la instancia de B relacionada. Ya que la semántica de las asociaciones de agregación y composición es equivalente entre UML y OO-Method, es posible establecer una correspondencia directa entre ellas para el intercambio de modelos.

- Correspondencia de generalización:

La generalización corresponde a una asociación entre varios clasificadores generales (hijos) y un clasificador específico (padre). En UML, los hijos heredan todo el comportamiento del padre y agregan información adicional.

Sin embargo, en OO-Method la herencia es semánticamente más precisa, pues permite especificar tanto la generalización permanente como la generalización temporal. La generalización permanente ocurre cuando la clase padre posee una condición de especialización sobre los atributos constantes de las clases hijas. En cambio, la generalización temporal ocurre cuando el padre tiene una condición de especialización sobre los atributos variables de sus hijos; o a través de eventos que asocian y liberan a padres con hijos (*carrier* y *liberator* respectivamente). En la generalización temporal también se debe indicar la clase donde se encuentran los eventos *carrier* y *liberator*, pues los hijos pueden liberarse por sí mismos o pueden ser liberados desde el padre.

Dado que en el contexto general, la generalización de UML es similar a la de OO-Method, éstas serán correspondidas. Se debe tener en cuenta que, cuando se quiera transformar un modelo UML a uno OO-Method, será necesario especificar en el modelo OO-Method los eventos *carrier* y *liberator* de la generalización de manera detallada.

- Correspondencia de paquetes:

Los paquetes son utilizados para agrupar elementos y proveer un espacio de nombres común a los elementos agrupados. Tanto en la especificación

OO-Method como en UML los clusters y paquetes son coherentes a esta descripción, por lo que su correspondencia será directa.

A modo de resumen, a continuación se presenta una tabla de equivalencias entre elementos de OO-Method con elementos de UML.

Tabla 2. Correspondencia entre elementos OO-Method y UML

OO-Method	UML
Proyecto	Modelo
Clase	Clase
Agente	Actor
Atributo	Atributo
Evento	Operación
Transacción	Operación
Operación	Operación
Argumento Entrada	Parámetro
Argumento Salida	Parámetro
Asociación	Asociación
Generalización	Generalización
Cluster	Paquete

3 Estrategia de Implementación

La estrategia de implementación, tiene por objetivo definir los pasos y mecanismos necesarios para poder llevar a cabo la transformación de elementos desde UML a OO-Method y viceversa, según las correspondencias presentadas en el punto 2.

3.1 Proceso de transformación desde UML a OO-Method

Para iniciar el proceso de transformación de un modelo UML a uno OO-Method, es necesario volcar el modelo UML en un archivo XMI, según el estándar establecido por la OMG. El volcado al archivo XMI se realiza con las utilidades que presente la propia herramienta UML en donde se ha modelado; por este motivo, es importante que la herramienta UML utilizada cumpla con el estándar XMI.

A continuación, con el archivo XMI que contiene el modelo UML, se efectúa la transformación automática del modelo UML a un modelo OO-Method, aplicando las correspondencias detalladas en el punto 2. Al término del proceso de transformación, se obtiene como resultado un archivo XML que contiene la definición del modelo equivalente para OO-Method.

Es importante señalar que las transformaciones aplicadas sobre el archivo XMI varían según la versión de UML con la cual se generó el modelo y la versión del

estándar XMI utilizada para volcarlo; por lo que será necesario especificar dicha información al momento de iniciar la transformación.

Finalmente, el archivo XML generado como producto de la transformación del archivo XMI, es importado en la herramienta OO-Method.

3.2 Proceso de transformación desde OO-Method a UML

Al igual que en la transformación desde UML a OO-Method, el primer paso en el proceso de transformación es volcar el modelo OO-Method en un archivo XML.

Una vez que se tiene el archivo XML que contiene el modelo OO-Method, se realiza su transformación automática a un modelo UML, utilizando las correspondencias descritas en el punto 2. Se obtendrá un archivo XMI como resultado de la transformación automática, construido según el estándar establecido por la OMG y que contiene la definición del modelo equivalente para UML.

Cabe destacar que el archivo XMI resultante debe estar en una versión del estándar XMI soportada por la herramienta UML. Lo mismo ocurre con el modelo UML que está definido en dicho archivo; razón por la que es necesario especificar la versión del archivo XMI y de UML soportados por la herramienta, antes de realizar la transformación del archivo XML que contiene el modelo OO-Method.

En último lugar, el archivo XMI generado es importado en la herramienta UML.

3.3 Consideraciones

Actualmente las herramientas UML no cumplen íntegramente con el estándar XMI establecido por la OMG, por lo que es necesario especificar además de las versiones de XMI y UML, la herramienta utilizada para generar el archivo XMI; tanto para el caso de la transformación UML a OO-Method, como para la transformación de OO-Method a UML.

Por otro lado, debido a que existen elementos OO-Method que no tienen representación directa en UML; la exportación de un modelo desde OO-Method a UML y luego la importación a OO-Method, puede dar como resultado un modelo OO-Method distinto al modelo original. Esta diferencia radica principalmente en las definiciones formales que se realizan mediante OASIS en el modelo OO-Method original y que pierden su formalidad al realizar la exportación a UML, ya que se convierten en documentación del modelo UML, y al transformar nuevamente el modelo UML a OO-Method, corresponden a documentación y no a definiciones formales en OASIS. Por esta razón, no existen garantías para recuperar el modelo original a partir de un modelo que se ha exportado e importado. Ejemplos de estas definiciones formales son la especificación de los servicios, de las derivaciones para los atributos derivados y de las precondiciones.

4 Ejemplo de intercambio de modelos

La estrategia de intercambio planteada en el punto 3 será aplicada en un proceso de intercambio real, que transforma un modelo de ejemplo generado en la herramienta UML Poseidon v4.1.2 (que se muestra en la Fig. 1) a un modelo de la herramienta OO-Method OLIVANOVA Modeler v6.9.a (que se muestra en la Fig. 2).

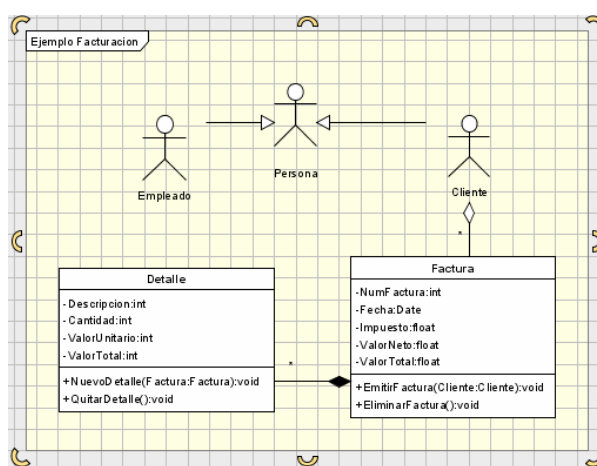


Fig. 1. Ejemplo de modelo realizado en Poseidon.

El primer paso contempla el volcado del modelo UML generado en Poseidon a un archivo XMI que cumpla con el estándar definido por la OMG. La versión utilizada de Poseidon cumple con la versión 1.2 del estándar XMI definido por la OMG para modelos UML versión 2.0. Para realizar el volcado del modelo, se utiliza la función de exportación a XMI que posee Poseidon.

El segundo paso consiste en ejecutar la aplicación de importación XMI construida para OLIVANOVA Modeler, que utiliza una plantilla de transformación XSL, donde son definidos los procedimientos necesarios para la identificación de los elementos del modelo volcado y su transformación, según lo establecido en el punto 2 de este documento.

La aplicación de importación solicita el archivo XMI que será importado, la herramienta utilizada para su generación, versión del estándar XMI utilizada para generar dicho archivo y versión de UML del modelo descrito. En este punto se debe especificar la herramienta Poseidon, versión 1.2 de XMI y versión 2.0 de UML.

Una vez que se ha ingresado toda información requerida por la aplicación, se identifican los elementos y se genera un reporte que muestra al usuario los elementos que han sido transformados desde el modelo UML.

Finalmente, el resultado del proceso de transformación es un archivo XML, que describe un modelo OO-Method según la especificación definida en el DTD 3.3 de la herramienta OLIVANOVA Modeler.

El modelo desplegado en la herramienta es el que se muestra en la Fig. 2, donde se puede apreciar claramente que los actores **Persona**, **Empleado** y **Cliente** del modelo original han sido traspasados como clases agentes en el modelo destino, las clases

factura y detalle del modelo de origen se conservan como clases en el modelo destino, todas ellas conservando sus nombres, atributos y operaciones. Además se puede visualizar la relación de generalización entre Persona, Empleado y Cliente; y la asociación entre Cliente y Factura; y también la asociación entre Factura y Detalle, cada una con sus respectivas cardinalidades.

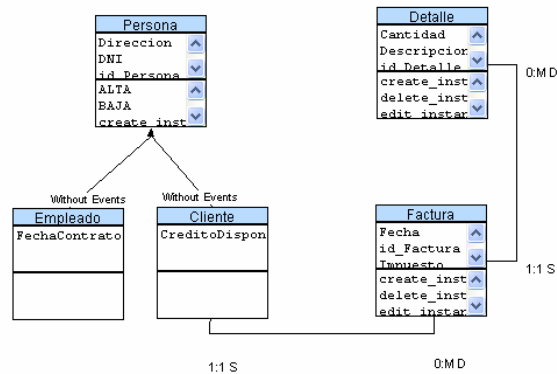


Fig. 2. Ejemplo de modelo traspasado a Olivanova Modeler.

El proceso de exportación funciona de manera inversa al proceso de importación, con las consideraciones especificadas el punto 3.

En este momento, se encuentran implementadas dos aplicaciones con la estrategia presentada en este documento: XMIImporter y XMIIExporter.

XMIImporter, permite importar los modelos realizados en herramientas UML que cumplan con el estándar XMI a la herramienta OO-Method OLIVANOVA Modeler. Las herramientas UML soportadas por XMIImporter son:

- Poseidon, para XMI v1.2 y UML v2.0
- MagicDraw, para XMI v1.2 y UML v1.4
- Rational Rose, para XMI v1.1 y UML v1.3

XMIIExporter, permite exportar los modelos realizados en la herramienta OLIVANOVA Modeler a herramientas UML que cumplan con el estándar XMI. Actualmente soporta la herramienta Poseidon, para XMI v1.2 y UML v2.0.

4 Conclusiones y trabajo futuro

Actualmente en el mercado existe un sinfín de herramientas CASE que apoyan el proceso de desarrollo de software. Algunas tienen más éxito que otras, dadas las facilidades que presentan para su uso y los productos que se pueden conseguir con ellas. En este sentido, el intercambio de modelos entre las distintas herramientas juega un papel principal, ya que permitiría utilizar las facilidades presentadas por las distintas herramientas y en un plano más global facilitar el intercambio de conocimiento entre organizaciones que desarrollan software y sus proyectos.

Hoy por hoy, la falta de estandarización por parte de las distintas herramientas de modelado presenta un obstáculo importante para alcanzar las metas señaladas; motivo por el cual es muy relevante la apertura de las herramientas al intercambio de modelos y que se adapten cada vez más a los estándares dictados, como lo es XMI. Por esta razón, se han desarrollado dos herramientas que permiten facilitar el intercambio de modelos UML y OO-Method que cumplen con el estándar XMI: XMIImporter y XMIExporter, que como su nombre lo indica, permiten importar modelos a la herramienta OO-Method y exportar modelos OO-Method respectivamente. Estas herramientas presentan una contribución real al intercambio de modelos bajo los estándares dictados por la OMG.

XMIImporter presta especial ayuda cuando se tiene un modelo realizado en UML y se quiere generar de manera automática el sistema para ese modelo, ya que al importar el modelo a OLIVANOVA Modeler y especificar en detalle los servicios nombrados en el diagrama de clases mediante OASIS y las diferentes utilidades presentadas por la herramienta, se puede generar y probar el sistema final en muy poco tiempo, permitiendo que los desarrollos de software sean más rápidos y con una menor probabilidad de fallos producto de la codificación.

Aún queda bastante trabajo por realizar en los procesos de intercambio de modelos, por lo que el trabajo futuro estará centrado en la definición de nuevas correspondencias, tanto a nivel de elementos como en la definición formal del comportamiento de dichos elementos. Ya se ha comenzado con el análisis de correspondencias entre los elementos de OO-Method y UML para los diagramas de estados y de colaboración, luego se continuará con los diagramas de secuencia y de actividades; y posteriormente se trabajará sobre la correspondencia entre definiciones OASIS y OCL que permitan intercambiar de la manera más completa posible los modelos entre herramientas UML y OO-Method.

Referencias

1. <http://www.omg.org/mda/>
2. <http://www.care-t.com>
3. <http://www.care-t.com/technology/mda.asp>
4. OMG: XML Metadata Interchange (XMI) Specification, version 1.2, 2002.
5. <http://www.omg.org/mof/>
6. OMG: MOF, versión 1.3, 2000.
7. <http://www.uml.org/>
8. <http://oomethod.dsic.upv.es>
9. Pastor O., Insfrán E.: OO-Method, the methodological support for Oliva Nova Model Execution System. <http://www.care-t.com/downloads/whitepapers/WP-OOMethod.pdf>
10. Pastor O., Ramos I., Cuevas J., Devesa J.: OASIS versión 2.0, An Object Definition Language for Object Oriented Databases, 1994.
11. OMG: Object Constraint Language, versión 2.0, 2006.
12. OMG: UML Superstructure Specification, versión 2.0, 2005.
13. Molina J., Pastor O.: MDA, OO-Method y la tecnología OLIVANOVA Model Execution, DSDM - 2004.

TÉCNICAS DE EVALUACIÓN DE SOFTWARE

Versión: 12.0

Fecha: 17 de octubre de 2005

Autoras: Natalia Juristo, Ana M. Moreno, Sira Vegas

ÍNDICE

1. INTRODUCCIÓN A LA EVALUACIÓN DE SOFTWARE	4
2. EVALUACIÓN DE SOFTWARE Y PROCESO DE DESARROLLO	8
3. TÉCNICAS DE EVALUACIÓN ESTÁTICA	11
3.1 BENEFICIOS DE LAS REVISIONES	11
3.2 OBJETIVOS DE LA EVALUACIÓN ESTÁTICA.....	11
3.3 TÉCNICAS DE EVALUACIÓN ESTÁTICA	13
3.4 INSPECCIONES	14
3.4.1 ¿Qué son las Inspecciones?.....	14
3.4.2 El Proceso de Inspección	14
3.4.3 Estimación de los Defectos Remanentes	17
3.4.4 Técnicas de Lectura.....	18
4. TÉCNICAS DE EVALUACIÓN DINÁMICA	27
4.1 CARACTERÍSTICAS Y FASES DE LA PRUEBA.....	27
4.2 TÉCNICAS DE PRUEBA	28
4.2.1 Pruebas de Caja Blanca o Estructurales	29
4.2.2 Pruebas de Caja Negra o Funcionales	35
4.3 ESTRATEGIA DE PRUEBAS	38
4.3.1 Pruebas Unitarias.....	38
4.3.2 Pruebas de Integración	38
4.3.3 Pruebas del Sistema.....	40
4.3.4 Pruebas de Aceptación	40
4.3.5 Pruebas de Regresión.....	41
5. PRUEBAS ORIENTADAS A OBJETOS.....	42
5.1 PRUEBA DE UNIDAD	42
5.2 PRUEBA DE INTEGRACIÓN	42
5.3 PRUEBA DE SISTEMA	43
5.4 PRUEBA DE ACEPTACIÓN.....	43
6. HERRAMIENTAS DE PRUEBA.....	44
6.1 HERRAMIENTA PARA EL ANÁLISIS ESTÁTICO DE CÓDIGO FUENTE.....	44
6.2 HERRAMIENTAS PARA PRUEBAS DE CARGA Y STRESS	44
6.3 HERRAMIENTA PARA LA AUTOMATIZACIÓN DE LAS PRUEBAS FUNCIONALES	45
6.4 HERRAMIENTAS DE DIAGNÓSTICO	45
6.5 HERRAMIENTA DE RESOLUCIÓN Y AFINADO	46
ANEXO A. DOCUMENTO DE REQUISITOS PARA EL SISTEMA DE VIDEO ABC	47
ANEXO B. LISTAS DE COMPROBACIÓN	57
ANEXO C. LISTAS DE COMPROBACIÓN PARA CÓDIGO	61
ANEXO D. PROGRAMA PARA EJERCICIO DE CÓDIGO	68
ANEXO E. SOLUCIÓN PARA EL EJERCICIO PRÁCTICO “COUNT”	72
ANEXO F. PROGRAMA “TOKENS” PARA PRACTICAR LA TÉCNICA DE ABSTRACCIÓN SUCESIVA	79
ANEXO G. PROGRAMA “SERIES” PARA PRACTICAR CON LA TÉCNICA DE ABSTRACCIÓN SUCESIVA	85
ANEXO H. EJERCICIO DE LECTURA BASADA EN DISEÑO.....	90
ANEXO I. EJERCICIO DE LECTURA BASADA EN PRUEBAS.....	93
ANEXO J. EJERCICIO DE LECTURA BASADA EN USO	96
ANEXO K. LISTA DE DEFECTOS — SISTEMA DE VÍDEO ABC	99
ANEXO L. EJERCICIO DE PRUEBA DE CAJA BLANCA	101
ANEXO M. EJERCICIO DE PRUEBA DE CAJA NEGRA	105
ANEXO N. PROGRAM “TOKENS” PARA PRACTICAR CON LA TÉCNICA DE CAJA BLANCA ..	108

ANEXO O. PROGRAMA “SERIES” PARA PRACTICAR CON LA TÉCNICA DE CAJA NEGRA....112

1. INTRODUCCIÓN A LA EVALUACIÓN DE SOFTWARE

La construcción de un sistema software tiene como objetivo satisfacer una necesidad planteada por un cliente. Pero ¿cómo puede saber un desarrollador si el producto construido se corresponde exactamente con lo que el cliente les pidió? y ¿cómo puede un desarrollador estar seguro de que el producto que ha construido va a funcionar correctamente?

Una de las posibles soluciones a este problema podría ser que el cliente evaluase el sistema que se ha construido una vez terminado. Sin embargo, esto tiene varias implicaciones:

1. Por una parte, puede que el cliente descubra que el producto desarrollado no cumple con sus expectativas, esto es, que el producto no hace lo que él esperaba que hiciera. Sin embargo, el producto ha sido terminado, por lo que ¿habría que comenzar de nuevo el desarrollo?
2. Por otra parte, la comprobación que el cliente podría realizar no sirve para comprobar que no hay errores en el software, puesto que ello depende de la porción del programa que se esté ejecutando en el momento de esta comprobación, pudiendo existir errores en otras partes del programa que no se ejecuten en ese momento.

Por lo tanto, lo recomendable es que el producto software vaya siendo evaluado a medida que se va construyendo. Por lo tanto, como veremos posteriormente, se hace necesario llevar a cabo, en paralelo al proceso de desarrollo, un proceso de evaluación o comprobación de los distintos productos o modelos que se van generando, en el que participarán desarrolladores y clientes.

Con el fin de entregar a los clientes productos satisfactorios, el software debe alcanzar ciertos niveles de calidad. Para alcanzar buenos niveles de calidad el número de defectos necesita mantenerse bajo mínimos.

El término calidad es difícil de definir. Esta dificultad se ha atacado elaborando este término en seis atributos que permiten una definición más sencilla. Estos seis atributos son:

- **Funcionalidad** – Habilidad del software para realizar el trabajo deseado.
- **Fiabilidad** - Habilidad del software para mantenerse operativo (funcionando).
- **Eficiencia** - Habilidad del software para responder a una petición de usuario con la velocidad apropiada.
- **Usabilidad** – Habilidad del software para satisfacer al usuario.
- **Mantenibilidad** – Habilidad del software para poder realizar cambios en él rápidamente y con una adecuada proporción cambio/costo.
- **Portabilidad** - Habilidad del software para correr en diferentes entornos informáticos.

A su vez, cada una de estas características del software se han subdividido en atributos aún más concretos. La Tabla 1 muestra una posible subdivisión. Aunque existen otras muchas otras descomposiciones de la calidad del software, ésta es una de las más aceptadas.

Independientemente de la descomposición de calidad que se elija, el nivel de propensión de faltas de un sistema software afecta siempre a varios de los atributos de calidad. En particular, fiabilidad y funcionalidad son siempre los más afectados. No obstante, no existe una relación bien establecida entre las faltas y la fiabilidad y funcionalidad. O dicho de otro modo, entre las faltas y los fallos. Todas las faltas de un producto software no se manifiestan como fallos. Las faltas se convierten en fallos cuando el usuario de un sistema software nota un comportamiento erróneo. Para que un sistema software alcance un nivel alto de calidad se requiere que el número de *fallos* sea bajo. Pero para mantener los fallos a niveles mínimos las faltas necesariamente deben también estar en niveles mínimos.

CHARACTERISTICS AND SUBCHARACTERISTICS		DESCRIPTION
Functionality		Characteristics relating to achievement of the basic purpose for which the software is being engineered
	Suitability	The presence and appropriateness of a set of functions for specified tasks
	Accuracy	The provision of right or agreed results or effects
	Interoperability	Software's ability to interact with specified systems
	Security	Ability to prevent unauthorized access, whether accidental or deliberate, to programs and data
	Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols.
Reliability		Characteristics relating to capability of software to maintain its level of performance under stated conditions for a stated period of time
	Maturity	Attributes of software that bear on the frequency of failure by faults in software
	Fault tolerance	Ability to maintain a specified level of performance in cases of software faults or unexpected inputs
	Recoverability	Capability and effort needed to re-establish level of performance and recover affected data after possible failure
	Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols
Usability		Characteristics relating to the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users
	Understandability	The effort required for a user to recognize the logical concept and its applicability
	Learnability	The effort required for a user to learn its application, operation, input and output
	Operability	The ease of operation and control by users
	Attractiveness	The capability of the software to be attractive to the user
	Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols
Efficiency		Characteristics related to the relationship between the level of performance of the software and the amount of resources used, under stated conditions
	Time behavior	The speed of response and processing times and throughput rates in performing its function
	Resource utilization	The amount of resources used and the duration of such use in performing its function
	Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols
Maintainability		Characteristics related to the effort needed to make modifications, including corrections, improvements or adaptation of software to changes in environment, requirements and functional specifications
	Analyzability	The effort needed for diagnosis of deficiencies or causes of failures, or for identification parts to be modified
	Changeability	The effort needed for modification fault removal or for environmental change
	Stability	The risk of unexpected effect of modifications
	Testability	The effort needed for validating the modified software
	Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols
Portability		Characteristics related to the ability to transfer the software from one organization or hardware or software environment to another
	Adaptability	The opportunity for its adaptation to different specified environments
	Installability	The effort needed to install the software in a specified environment
	Co-existence	The capability of a software product to co-exist with other independent software in common environment
	Replaceability	The opportunity and effort of using it in the place of other software in a particular environment
	Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols

Tabla 1. Descomposición de la calidad del software por ISO 9126-1998

Resumiendo, la calidad es difícil de definirse. Para facilitar su comprensión la calidad se ha descompuesto en atributos. Controlar y corregir las faltas existentes en un producto software afecta positivamente algunos atributos de calidad. En particular, si se trabaja en detectar y eliminar las faltas y los fallos la funcionalidad y la fiabilidad mejoran.

A la hora de abordar estos atributos es importante distinguir entre cuatro conceptos muy relacionados pero distintos, algunos de ellos ya mencionados:

- *Error*: acción humana que produce una falta.
- *Falta*: algo que está mal en un producto (modelo, código, documento, etc.).
- *Fallo*: manifestación de una falta.
- *Defecto*: error, falta o fallo.

Durante el desarrollo de software, las distintas técnicas de evaluación son las principales estrategias para detectar faltas y fallos. Por tanto, son métodos de control de la calidad.

En términos generales, se pueden distinguir dos tipos de evaluaciones durante el proceso de desarrollo: Verificaciones y Validaciones. Según el IEEE Std 729-1983 éstas se definen como:

- *Verificación*: Proceso de determinar si los productos de una determinada fase del desarrollo de software cumplen o no los requisitos establecidos durante la fase anterior.
- *Validación*: Proceso de evaluación del software al final del proceso de desarrollo para asegurar el cumplimiento de las necesidades del cliente.

Así, la verificación ayuda a comprobar si se ha construido el producto correctamente, mientras que la validación ayuda a comprobar si se ha construido el producto correcto. En otras palabras, la verificación tiene que ver típicamente con errores de los desarrolladores que no han transformado bien un producto del desarrollo en otro. Mientras que la validación tiene que ver con errores de los desarrolladores al malinterpretar las necesidades del cliente. Así la única persona que puede validar el software, ya sea durante su desarrollo como una vez finalizado, es el cliente, ya que será quien pueda detectar si hubo o no errores en la interpretación de sus necesidades.

Tanto para la realización de verificaciones como de validaciones se pueden utilizar distintos tipos de técnicas. En general, estas técnicas se agrupan en dos categorías:

- *Técnicas de Evaluación Estáticas*: Buscan faltas sobre el sistema en reposo. Esto es, estudian los distintos modelos que componen el sistema software buscando posibles faltas en los mismos. Así pues, estas técnicas se pueden aplicar, tanto a requisitos como a modelos de análisis, diseño y código.
- *Técnicas de Evaluación Dinámicas*: Generan entradas al sistema con el objetivo de detectar fallos, al ejecutar dicho sistema sobre esas entradas. Esto es, se pone el sistema a funcionar buscando posibles incongruencias entre la salida esperada y la salida real. La aplicación de técnica dinámicas es también conocido como *pruebas* del software o *testing* y se aplican generalmente sobre código que es, hoy por hoy, el único producto ejecutable del desarrollo.

Veamos en la siguiente sección cómo estas técnicas de evaluación han de aplicarse durante todo el proceso de desarrollo. Pero antes recordemos que todo proceso de evaluación además de la posible detección de defectos conlleva un proceso de depuración, esto es la corrección de los mismos. Ambas tareas (detección y corrección) pueden realizarse por una misma persona o por personas distintas según la organización y el modelo de desarrollo sobre el que se esté aplicando la técnica.

Nosotros nos centraremos en el proceso de detección de defectos. Las técnicas de evaluación, tanto las estáticas como las dinámicas se centran en detectar, la primera faltas y la segunda fallos, pero no aporta ayuda en la corrección de los mismos

2. EVALUACIÓN DE SOFTWARE Y PROCESO DE DESARROLLO

Tal como se ha indicado anteriormente, es necesario evaluar el sistema software a medida que se va avanzando en el proceso de desarrollo de dicho sistema. De esta forma se intenta que la detección de defectos se haga lo antes posible y tenga menor impacto en el tiempo y esfuerzo de desarrollo. Ahora bien ¿cómo se realiza esta evaluación?

Las técnicas de evaluación estática se aplican en el mismo orden en que se van generando los distintos productos del desarrollo siguiendo una filosofía *top-down*. Esto es, la evaluación estática acompaña a las actividades de desarrollo, a diferencia de la evaluación dinámica que únicamente puede dar comienzo cuando finaliza la actividad de codificación, siguiendo así una estrategia *bottom-up*. La evaluación estática es el único modo disponible de evaluación de artefactos para las primeras fases del proceso de desarrollo (análisis y diseño), cuando no existe código. Esta idea se muestra en la Figura 1 en la que como se observa la evaluación estática se realiza en el mismo sentido en que se van generando los productos del desarrollo de software, mientras que la dinámica se realiza en sentido inverso.

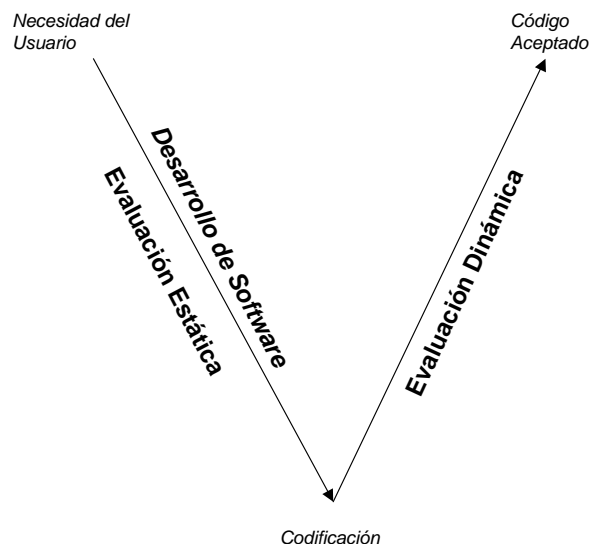


Figura 1. Abstracción de la Relación entre Evaluación y Proceso Software

Más concretamente, la Figura 2 muestra en detalle la aplicación de las técnicas estáticas y dinámicas para evaluar software. La evaluación estática (conocida con el nombre genérico de Revisiones) se realiza en paralelo al proceso de construcción, constando de una actividad de evaluación emparejada con cada actividad de desarrollo. Es decir, la actividad de Definición de Requisitos de Usuario va acompañada de una actividad de Revisión de Requisitos de Usuario, la actividad de Definición de Requisitos Software va emparejada con su correspondiente actividad de revisión y así, sucesivamente.

Las actividades de revisión marcan el punto de decisión para el paso a la siguiente actividad de desarrollo. Es decir, la actividad de requisitos interactúa con la actividad de revisión de requisitos en un bucle de mejora iterativa hasta el momento en que la calidad de los requisitos permite abordar la subsiguiente fase de desarrollo. Lo mismo ocurre con el diseño arquitectónico: sufrirá una mejora iterativa hasta que su nivel de calidad permita pasar al diseño detallado y así, sucesivamente. Nótese que esto también ocurre en la fase de codificación. La actividad siguiente a la de implementación es la fase de pruebas unitarias. No obstante, antes de pasar a ella, los

programas deberán evaluarse estáticamente. Del mismo modo que se ha hecho con los otros productos.

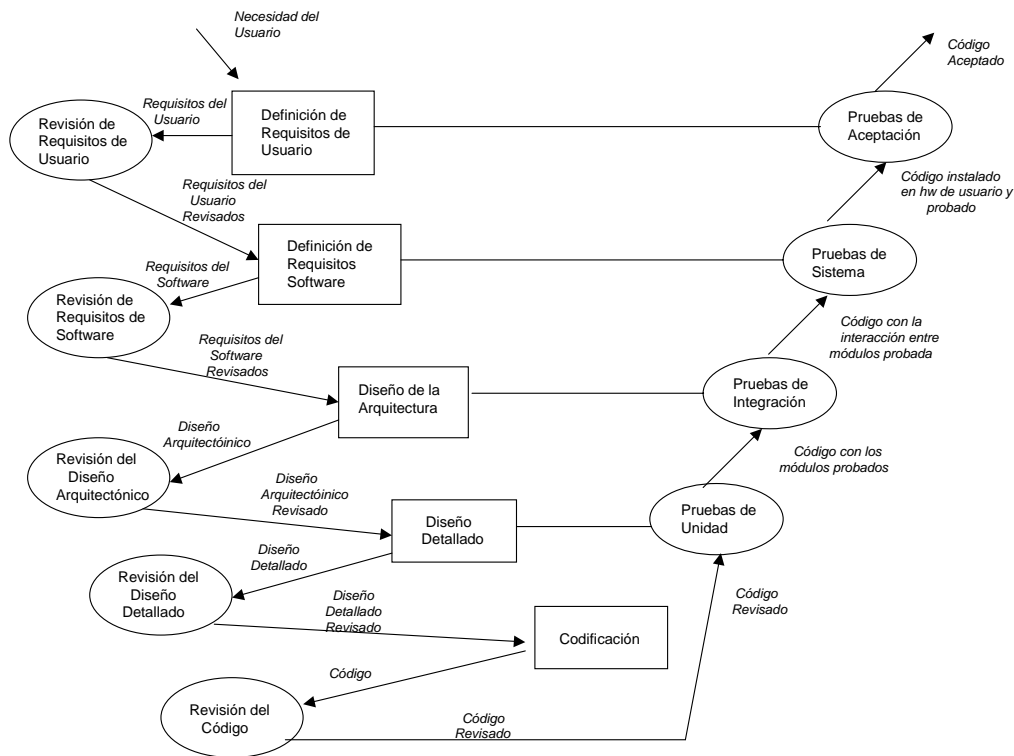


Figura 2. Modelo en V de Evaluación de Software

Por tanto, las actividades de evaluación estática constituyen los puntos de control o revisión utilizados por los gestores de proyectos y las organizaciones para evaluar tanto la calidad de los productos como el progreso del proyecto. Es decir, las actividades de revisión son una herramienta de control para el producto software.

Una vez realizadas estas revisiones se procede con la evaluación dinámica, que como ya se ha indicado se realiza sobre el código. Aunque más adelante se estudiarán en detalle los distintos tipos de pruebas dinámicas, se puede indicar que la primera prueba a realizar es la denominada Prueba de Unidad en la que se buscan errores en los componentes más pequeños del programa (módulos). Estos errores se detectan cuando dichos componentes no actúan como se ha especificado en el diseño detallado. Seguidamente, se prueban los distintos componentes que constituyen el software en la denominada Prueba de Integración. Esta prueba está orientada a detectar fallos provocados por una incorrecta (no acorde con la especificación de diseño de alto nivel) comunicación entre módulos. El software se puede ejecutar en un contexto hardware concreto, por lo que la Prueba de Sistema es la que se encarga de buscar errores en este ensamblaje software/hardware. Finalmente, el usuario ha de realizar la Prueba de Aceptación final sobre el sistema completo.

Nótese cómo la evaluación de los productos software mediante revisiones permite contar con una estimación temprana de la calidad con que se está llevando a cabo el desarrollo. Esto es así porque las revisiones encuentran faltas, pero la cantidad de faltas encontradas en un producto dan una idea de las faltas que aún pueden quedar así como de la calidad del trabajo de desarrollo de dicho producto. La experiencia parece indicar que donde hay un defecto hay otros. Es decir, la probabilidad de descubrir nuevos defectos en una parte del software es proporcional al número de defectos ya descubiertos. Es en este principio sobre el que se basan los métodos de estimación de los defectos que quedan en un software; ya sean los modelos de fiabilidad (que utilizan como entrada los fallos encontrados durante las pruebas) ya sean los métodos de estimación del contenido de faltas (que utilizan como entrada las faltas encontradas mediante revisiones). No obstante, es gracias a la evaluación estática que se puede realizar esta estimación de la calidad del software de manera temprana, puesto que los modelos de fiabilidad requieren el código ya desarrollado para dar una indicación de los posibles fallos que quedan remanentes en dicho código.

Así pues, la importancia de las técnicas estáticas de evaluación a la hora de controlar el nivel de calidad con el que se está llevando a cabo el desarrollo es crucial. Los modelos que utilizan los datos de las técnicas de testing, ayudan a predecir la fiabilidad del software que se está entregando (cuántas fallos quedan en el sistema sin encontrar), pero poco se puede hacer ya, excepto seguir probando el sistema hasta elevar el nivel de fiabilidad del mismo. Sin embargo, la estimación de faltas que aún quedan en un producto utilizando datos de las revisiones permiten dos acciones que ayudan a prevenir futuros defectos en el proyecto:

- Seguir revisando el producto para disminuir el número de faltas remanentes. Por tanto esta detección temprana previene encontrar estas faltas en estadios más avanzados del desarrollo. Es decir, la falta que detectemos en los requisitos estaremos evitando contagiarla al diseño y al código.
- Tomar medidas correctivas del desarrollo si las estimaciones indican que se está llevando a cabo un trabajo pobre. Es decir, si las estimaciones de faltas remanentes indican que un determinado producto contiene más faltas de las habituales, algo se está haciendo mal (hay problemas en el equipo de desarrollo, algún miembro del equipo tiene problemas que está afectando a su trabajo, hay problemas con las técnicas que se están utilizando, quizás el equipo no las conoce bien, etc.) y deben tomarse acciones que remedien o palien estos problemas antes de que afecten al resultado final del proyecto.

En las secciones 3 y 4 se detallan las técnicas estáticas y dinámicas respectivamente.

3. TÉCNICAS DE EVALUACIÓN ESTÁTICA

3.1 Beneficios de las Revisiones

La razón para buscar defectos en productos tempranos es porque éstos se traducen en defectos en el producto final. Es decir, defectos en los requisitos se traducirán en defectos en el sistema final. Veamos una analogía con la arquitectura de edificios. Si en un plano el color de una línea indica su significado, una confusión en el color se traducirá en un error en el edificio. Por ejemplo, si el azul indica tuberías de agua y el amarillo cables eléctricos y el arquitecto comete un error usando el azul en una conducción eléctrica, los electricistas que usen el plano como guía para su trabajo no colocarán cables eléctricos mientras que los fontaneros colocarán tuberías de agua donde no debían ir. El plano de un edificio es el artefacto equivalente al diseño de un producto software. Si un diseño contiene defectos, seguramente estos defectos se transmitirán al código cuando los programadores usen ese diseño como guía para su trabajo.

La detección temprana de errores acarrea grandes beneficios. Si las revisiones únicamente se aplican al código mejoran la calidad y producen ahorros en los costos del proyecto. Pero los ahorros son mayores si se inspeccionan artefactos tempranos del desarrollo. Estudiando los resultados publicados sobre ahorros con las revisiones, puede afirmarse que la utilización de inspecciones de código produce un ahorro del 39% sobre el coste de detectar y corregir defectos, frente a únicamente utilizar la evaluación dinámica. Sin embargo, el ahorro es del 44% si se inspecciona también el diseño.

La experiencia demuestra que entre el 30% y el 70% de los defectos, de diseño y código son detectados por las técnicas estáticas. Esto supone un gran ahorro, pues la corrección es más fácil y menos costosa durante la evaluación estática que durante la dinámica. Nótese que cuando durante la evaluación dinámica del sistema se detecta un fallo en un programa, lo que se detecta es el *fallo*, no la falta que lo provoca. Es decir, tras la detección del fallo, se requiere una labor de localización en el programa de la falta que provocó el fallo. Sin embargo, con las técnicas estáticas, lo que se detecta son directamente *faltas*. Por tanto, una vez detectada, se puede pasar a la fase de corrección. Es decir, desaparece la tarea de localización de la falta. Esto significa, que las técnicas estáticas son más baratas por falta que las dinámicas.

Las revisiones también proporcionan beneficios más generales. Entre éstos se pueden citar están:

- Evaluación del progreso del proyecto
- Potencia las capacidades de los participantes
- Mejoran la comunicación entre el equipo de desarrollo, aumentando su motivación, pues los productos pasan a ser documentos públicos.
- Proporciona aprendizaje, retroalimentación y prevención
- Forma y educa a los participantes

En el caso concreto de las revisiones de código, éstas, además, permiten localizar secciones críticas, lo que permitirá dedicar un mayor esfuerzo a ellas en la fase de pruebas.

3.2 Objetivos de la Evaluación Estática

La evaluación estática de los distintos artefactos o productos que se generan en el desarrollo de software (especificación de requisitos, modelos conceptuales, diseño, código, etc.) pretende comprobar su calidad.

La calidad significa una cosa distinta para cada producto, precisamente porque son artefactos distintos. Del mismo modo que la calidad de un plano y la calidad de una casa significa cosas distintas. En un plano de un futuro edificio se desea que sea claro (se entienda suficientemente bien como para servir de guía a la construcción del edificio), que sea correcto (por ejemplo, que las líneas que identifican paredes indiquen, a escala, efectivamente el lugar donde se desea que vayan las paredes), que no tenga inconsistencias (por ejemplo, entre las distintas hojas que forman el plano; si una página se focaliza, digamos, en una habitación que en otra página aparecía sólo sus cuatro paredes, que las medidas de las líneas en ambas páginas se correspondan con la misma medida de la realidad), etc.. Sin embargo, de una casa se espera que sea robusta (por ejemplo, que no se caiga), usable (por ejemplo, que los peldaños de las escaleras no sean tan estrechos que provoquen caídas) etc. Por tanto, cuando se esté evaluando estáticamente un producto software, es importante que el evaluador tenga en mente qué tipo de defectos está buscando y cuál sería un producto de ese tipo de calidad adecuada. Digamos que si uno no sabe lo que busca (por ejemplo, inconsistencias al revisar la calidad de un plano) es difícil que lo encuentre, aunque lo tenga delante.

Los defectos que se buscan al evaluar estáticamente los productos software son:

- Para los **requisitos**:
 - *Corrección*. Los requisitos especifican correctamente lo que el sistema debe hacer. Es decir, un requisito incorrecto es un requisito que no cumple bien su función. Puesto que la función de un requisito es indicar qué debe hacer el sistema, un requisito incorrecto será aquel que, o bien pide que el sistema haga algo que no es lo que debe hacer, o pide que haga algo que no deba hacer.
 - *Compleción*. Especificación completamente el problema. Está especificado todo lo que tiene que hacer el sistema y no incluye nada que el sistema no deba hacer. En dos palabras: no falta nada; no sobra nada.
 - *Consistencia*. No hay requisitos contradictorios.
 - *Ambigüedad*. Los requisitos no pueden estar sujetos a interpretación. Si fuese así, un mismo requisito puede ser interpretado de modo distinto por dos personas diferentes y, por tanto, crear dos sistemas distintos. Si esto es así, los requisitos pierden su valor pues dejan de cumplir su función (indicar qué debe hacer el sistema). Las ambigüedades provocan interpretación por parte de la persona que use o lea los requisitos. Por tanto, una especificación debe carecer de ambigüedades.
 - *Claridad*. Se entiende claramente lo que está especificado.
- Para el **diseño**:
 - *Corrección*. El diseño no debe contener errores. Los errores de corrección se pueden referir a dos aspectos. Defectos de “escritura”, es decir, defectos en el uso de la notación de diseño empleada (el diseño contiene detalles prohibidos por la notación). Defectos con respecto a los requisitos: el diseño no realiza lo que el requisito establece. Hablando apropiadamente, los primeros son los puros defectos de corrección, mientras que los segundos son defectos de validez.

- *Compleción.* El diseño debe estar completo. Ya sea que diseña todo el sistema marcado por los requisitos; ya sea no diseñando ninguna parte no indicada en los requisitos. De nuevo, nada falta, nada sobra.
- *Consistencia.* Al igual que en los requisitos, el diseño debe ser consistente entre todas sus partes. No puede indicarse algo en una parte del diseño, y lo contrario en otra.
- *Factibilidad.* El diseño debe ser realizable. Debe poderse implementar.
- *Trazabilidad.* Se debe poder navegar desde un requisito hasta el fragmento de diseño donde éste se encuentra representado.
- **Código Fuente:**
 - *Corrección.* El código no debe contener errores. Los errores de corrección se pueden referir a dos aspectos. Defectos de “escritura”, es decir, lo que habitualmente se conoce por “programa que no funciona”. Por ejemplo, bucles infinitos, variable definida de un tipo pero utilizada de otro, contador que se sale de las dimensiones de un array, etc. Defectos con respecto al diseño: el diseño no realiza lo que el diseño establece.

De nuevo, hablando apropiadamente, los primeros son los puros defectos de corrección, mientras que los segundos son defectos de validez. Un defecto de corrección es un código que está mal para cualquier dominio. Un defecto de validez es un código que, en este dominio particular (el marcado por esta necesidad de usuario, estos requisitos, y este diseño) hace algo inapropiado. Por ejemplo, define una variable de un tipo (y se usa en el programa con ese tipo, es decir, “a primera vista” no hay nada incorrecto en la definición del tipo y su uso) que no es la que corresponde con el problema; o define un array de un tamaño que no es el que se corresponde con el problema. Nótese que para detectar los errores de validez (en cualquier producto) debe entenderse el problema que se pretende resolver, mientras que los defectos de corrección son errores siempre, aún sin conocer el problema que se pretende resolver.

- *Compleción.* El código debe estar completo. Una vez más, nada falta ni nada sobra (con respecto, en este caso, al diseño)
- *Consistencia.* Al igual que en los requisitos y diseño, el código debe ser consistente entre todas sus partes. No puede hacerse algo en una parte del código, y lo contrario en otra.
- *Trazabilidad.* Se debe poder navegar desde un requisito hasta el fragmento de código donde éste se ejecute, pasando por el fragmento de diseño.

3.3 Técnicas de Evaluación Estática

Las técnicas de Evaluación estática de artefactos del desarrollo se las conoce de modo genérico por **Revisiones**. Las revisiones pretenden detectar *manualmente* defectos en cualquier producto del

desarrollo. Por manualmente queremos decir que el producto en cuestión (sea requisito, diseño, código, etc.) está impreso en papel y los revisores están analizando ese producto mediante la lectura del mismo, *sin ejecutarlo*.

Existen varios tipos de revisiones, dependiendo de qué se busca y cómo se analiza ese producto. Podemos distinguir entre:

- *Revisiones informales*, también llamadas inadecuadamente sólo Revisiones (lo cual genera confusión con el nombre genérico de todas estas técnicas). Las Revisiones Informales no dejan de ser un intercambio de opiniones entre los participantes.
- Revisiones formales o *Inspecciones*. En las Revisiones Formales, los participantes son responsables de la fiabilidad de la evaluación, y generan un informe que refleja el acto de la revisión. Por tanto, sólo consideramos aquí como técnica de evaluación las revisiones formales, puesto que las informales podemos considerarlas un antepasado poco evolucionado de esta misma técnica.
- *Walkthrough*. Es una revisión que consiste en simular la ejecución de casos de prueba para el programa que se está evaluando. No existe traducción exacta en español y a menudo se usa el término en inglés. Quizás la mejor traducción porque ilustra muy bien la idea es *Recorrido*. De hecho, con los walkthrough se recorre el programa imitando lo que haría la computadora.
- *Auditorias*. Las auditorias contrastan los artefactos generados durante el desarrollo con estándares, generales o de la organización. Típicamente pretenden comprobar formatos de documentos, inclusión de toda la información necesaria, etc. Es decir, no se tratan de comprobaciones técnicas, sino de gestión o administración del proyecto.

3.4 Inspecciones

3.4.1 ¿Qué son las Inspecciones?

Las inspecciones de software son un método de análisis estático para verificar y validar un producto software manualmente. Los términos Inspecciones y Revisiones se emplean a menudo como sinónimos. Sin embargo, como ya se ha visto, este uso intercambiable no es correcto.

Las Inspecciones son un proceso bien definido y disciplinado donde *un equipo de personas* cualificadas analizan un producto software usando una *técnica de lectura* con el propósito de detectar defectos. El objetivo principal de una inspección es detectar faltas antes de que la fase de prueba comience. Cualquier desviación de una propiedad de calidad predefinida es considerada un defecto.

Para aprender a realizar inspecciones vamos a estudiar primero el proceso que debe seguirse y luego las técnicas de lectura.

3.4.2 El Proceso de Inspección

Las Inspecciones constan de dos partes: Primero, la *comprensión* del artefacto que se inspecciona; Y en segundo lugar, la *búsqueda de faltas* en dicho artefacto. Más concretamente, una inspección tiene cuatro fases principales:

1. **Inicio** – El objetivo es preparar la inspección y proporcionar la información que se necesita sobre el artefacto para realizar la inspección.
2. **Detección de defectos** – Cada miembro del equipo realiza *individualmente* la lectura del materia, comprensión del artefacto a revisar y la detección de faltas. Las Técnicas de Lectura ayudan en esta etapa al inspector tanto a comprender el artefacto como a detectar faltas. Basándose en las faltas detectadas cada miembro debe realizar una estimación subjetiva del número de faltas remanentes en el artefacto.
3. **Colección de defectos** – El registro de las faltas encontrada por cada miembro del equipo es compilado en un solo documento que servirá de basa a la discusión sobre faltas que se realizará *en grupo*. Utilizando como base las faltas comunes encontradas por los distintos inspectores se puede realizar una estimación objetiva del número de faltas remanentes. En la reunión se discutirá si las faltas detectadas son falsos positivos (faltas que algún inspector cree que son defectos pero que en realidad no lo son) y se intentará encontrar más faltas ayudados por la sinergia del grupo.
4. **Corrección y seguimiento** – El autor del artefacto inspeccionado debe corregir las faltas encontradas e informar de las correcciones realizadas a modo de seguimiento.

Estas fases se subdividen además en varias subfases:

1. Inicio
 - 1.1 Planificación
 - 1.2 Lanzamiento
2. Detección de defectos
3. Colección de defectos
 - 3.1 Compilación
 - 3.2 Inspección en grupo
4. Corrección y seguimiento
 - 4.1 Corrección
 - 4.2 Seguimiento

Veamos cada una de estas fases;

Durante **La Planificación** se deben seleccionar los participantes, asignarles roles, preparar un calendario para la reunión y distribuir el material a inspeccionar. Típicamente suele haber una persona en la organización o en el proyecto que es responsable de planificar todas las actividades de inspección, aunque luego juegue además otros papeles. Los papeles que existen en una inspección son:

- *Organizador*. El organizador planifica las actividades de inspección en un proyecto, o incluso en varios proyectos (o entre proyectos, porque se intercambian participantes: los desarrollados de uno son inspectores de otro).

- *Moderador.* El moderador debe garantizar que se sigan los procedimientos de la inspección así como que los miembros del equipo cumplan sus responsabilidades. Además, modera las reuniones, lo que significa que el éxito de la reunión depende de esta persona y, por tanto, debe actuar como líder de la inspección. Es aconsejable que la persona que juegue este rol haya seguido cursos de manejo de reuniones y liderazgo.
- *Inspector.* Los inspectores son los responsables de detectar defectos en el producto software bajo inspección. Habitualmente, todos los participantes en una inspección actúan también como inspectores, independientemente de que, además, jueguen algún otro papel.
- *Lector/Presentador.* Durante la reunión para la inspección en grupo, el lector dirigirá al equipo a través del material de modo completo y lógico. El material debe ser parafraseado a una velocidad que permita su examen detallado al resto de los participantes. Parafrasear el material significa que el lector debe explicar e interpretar el producto en lugar de leerlo literalmente.
- *Autor.* El autor es la persona que ha desarrollado el producto que se está inspeccionando y es el responsable de la corrección de los defectos durante la fase de corrección. Durante la reunión, el autor contesta a las preguntas que el lector no es capaz de responder. El autor no debe actuar al mismo tiempo ni de moderador, ni de lector, ni de escriba.
- *Escriba.* El secretario o escriba es responsable de incorporar todos los defectos en una lista de defectos durante la reunión.
- *Recolector.* El recolector recoge los defectos encontrados por los inspectores en caso de no haber una reunión de inspección.

Es necesario hacer ciertas consideraciones sobre *el número de participantes*. Un equipo de inspección nunca debería contar con más de cinco miembros. Por otro lado, el número mínimo de participantes son dos: el autor (que actúa también de inspector) y un inspector. Lo recomendable es comenzar con un equipo de tres o cuatro personas: el autor, uno o dos inspectores y el moderador (que actúa también como lector y escriba). Tras unas cuantas inspecciones la organización puede experimentar incorporando un inspector más al grupo y evaluar si resulta rentable en términos de defectos encontrados.

Sobre el tema de cómo *seleccionar las personas adecuadas* para una inspección, los candidatos principales para actuar como inspectores es el personal involucrado en el desarrollo del producto. Se pueden incorporar inspectores externos si poseen algún tipo de experiencia específica que enriquezca la inspección. Los inspectores deben tener un alto grado de experiencia y conocimiento.

La fase de **Lanzamiento** consiste en una primera reunión donde el autor explica el producto a inspeccionar a los otros participantes. El objetivo principal de esta reunión de lanzamiento, es facilitar la comprensión e inspección a los participantes. No obstante, este meeting no es completamente necesario, pues en ocasiones puede consumir más tiempo y recursos de los beneficios que reporte. Sin embargo, existen un par de condiciones bajo las cuales es recomendable realizar esta reunión. En primer lugar, cuando el artefacto a inspeccionar es complejo y difícil de comprender. En este caso una explicación por parte del autor sobre el producto a inspeccionar facilita la comprensión al resto de participantes. En segundo lugar, cuando el artefacto a inspeccionar pertenece a un sistema software de gran tamaño. En este caso, se hace necesario que el autor explique las relaciones entre el artefacto inspeccionado y el sistema software en su globalidad.

La fase de **Detección de Defectos** es el corazón de la inspección. El objetivo de esta fase es escudriñar un artefacto software para obtener defectos. Localizar defectos es una actividad en parte individual y en parte de grupo. Si se olvida la parte individual de la inspección, se corre el riesgo de que los participantes sean más pasivos durante la reunión y se escuden en el grupo para evitar hacer su contribución. Así pues, es deseable que exista una fase de detección individual de defectos con el objetivo explícito de que cada participante examine y entienda en solitario el producto y busque defectos. Este esfuerzo individual garantiza que los participantes irán bien preparados a la puesta en común.

Los defectos detectados por cada participante en la inspección debe ser reunido y documentado. Es más, debe decidirse si un defecto es realmente un defecto. Esta recolección de defectos y la discusión sobre falsos positivos se realizan, respectivamente, en la fase de **Compilación y Reunión**. La recolección de defectos debe ayudar a tomar la decisión sobre si es necesaria una reinspección del artefacto o no. Esta decisión dependerá de la cantidad de defectos encontrados y sobre todo de la coincidencia de los defectos encontrados por distintos participantes. Una coincidencia alta de los defectos encontrados por unos y por otros (y un numero bajo de defectos encontrados) hace pensar que la cantidad de defectos que permanecen ocultos sea baja. Una coincidencia pobre (y un numero relativamente alto de defectos encontrados) hace pensar que aun quedan muchos defectos por detectar y que, por tanto, es necesaria una reinspección (una vez acabada ésta y corregidas las faltas encontradas).

Dado que una reunión consume bastantes recursos (y más cuantos más participantes involucre) se ha pensado en una alternativa para hacer las reuniones más ligeras. Las llamadas reuniones de deposición, donde sólo asisten tres participantes: moderador, autor, y un representante de los inspectores. Este representante suele ser el inspector de más experiencia, el cual recibe los defectos detectados por los otros inspectores y decide él, unilateralmente, sobre los falsos positivos. Algunos autores incluso han dudado del efecto sinergia de las reuniones y han aconsejado su no realización. Parece que lo más recomendable es que las organizaciones comiencen con un proceso de inspección tradicional, donde la reunión sirve para compilar defectos y discutir falsos positivos, y con el tiempo y la experiencia prueben a variar el proceso eliminando la reunión y estudiando si se obtienen beneficios equivalentes en términos de defectos encontrados.

Es importante resaltar, que la reunión de una inspección no es una sesión para resolver los defectos u otros problemas. No se deben discutir en estas reuniones ni soluciones digamos radicales (otras alternativas de diseño o implementación, que el autor no ha utilizado pero podría haberlo hecho), ni cómo resolver los defectos detectados, y, mucho menos, discutir sobre conflictos personales o departamentales.

Finalmente, el autor corrige su artefacto para resolver los defectos encontrados o bien proporciona una explicación razonada sobre porqué cierto defecto detectado en realidad no lo es. Para esto, el autor repasa la lista de defectos recopilada y discute o corrige cada defecto. El autor deberá enviar al moderador un informe sobre los defectos corregidos o, en caso de no haber corregido alguno, porqué no debe corregirse. Este informe sirve de seguimiento y cierre de la inspección, o, en caso de haberse decidido en la fase de recopilación que el artefacto necesitaba reinspección, se iniciará de nuevo el proceso.

3.4.3 Estimación de los Defectos Remanentes

A pesar de que el propósito principal de las Inspecciones es detectar y reducir el número de defectos, un efecto colateral pero importante es que permiten realizar desde momentos muy

iniciales del desarrollo predicciones de la calidad del producto. Concretamente, las estimaciones de las faltas remanentes tras una inspección debe utilizarse como control de la calidad del proceso de desarrollo.

Hay varios momentos de estimación de faltas remanentes en una inspección. Al realizar la búsqueda individual de faltas, el inspector puede tener una idea de las faltas remanentes en base a las siguientes dos heurísticas:

- Encontrar muchas faltas es sospechoso. Muchas faltas detectadas hacen pensar que debe haber muchas más, puesto que la creencia de que queden pocas faltas sólo se puede apoyar en la confianza en el proceso de inspección y no en la calidad del artefacto (que parece bastante baja puesto que hemos encontrado muchas faltas)
- Encontrar muy pocas faltas también resulta sospechoso, especialmente si es la primera vez que se inspecciona este artefacto. Pocas faltas hacen pensar que deben quedar muchas más, puesto que esta situación hace dudar sobre la calidad del proceso de inspección: No puede saberse si se han encontrado pocas debido a la alta calidad del artefacto o a la baja calidad de la inspección.

La estimación más fiable sobre el número de faltas remanentes que se puede obtener en una Inspección es la coincidencia de faltas entre los distintos inspectores. Los métodos que explotan coincidencias para estimar se llaman Estimaciones Captura-Recaptura y no son originarias de la Ingeniería del Software. En concreto lo usó por primera vez Laplace para estimar la población de Francia en 1786, y se utilizan a menudo en Biología para estimar el tamaño de poblaciones de animales. En el caso de las Inspecciones, el nivel de coincidencia de las faltas detectadas por los distintos revisores es usado como estimador¹.

La idea sobre la que se basa el uso de las coincidencias es la siguiente: Pocas coincidencias entre los revisores, significa un número alto de faltas remanentes; Muchas coincidencias, significará pocas faltas remanentes. Los casos extremos son los siguientes. Supongamos que todos los revisores han encontrado exactamente el mismo conjunto de faltas. Esto debe significar que deben quedar muy pocas faltas, puesto que el proceso de inspección parece haber sido bueno (los inspectores han encontrado las mismas faltas) parece poco probable que queden más faltas por ahí que ningún revisor ha encontrado. Supongamos ahora que ningún revisor ha encontrado la misma falta que otro revisor. Esto debe querer decir que quedan muchas más por encontrar, puesto que el proceso de revisión parece haber sido pobre (a cada revisor se le han quedado ocultas n faltas –todas las encontradas por los otros revisores–) vaya usted a saber cuántas faltas más han quedado ocultas a todos los revisores. Esta situación debería implicar una nueva inspección del artefacto (tanto para mejorar la calidad del mismo, como para hacer un intento de mejorar la propia inspección).

3.4.4 Técnicas de Lectura

Las técnicas de lectura son guías que ayudan a detectar defectos en los productos software. Típicamente, una técnica de lectura consiste en una serie de pasos o procedimientos cuyo propósito es que el inspector adquiere un conocimiento profundo del producto software que inspecciona. La comprensión de un producto software bajo inspección es un prerequisite para detectar defectos sutiles y, o, complejos. En cierto sentido, una técnica de lectura puede verse como un mecanismo para que los inspectores detecten defectos en el producto inspeccionado.

¹ Un estimador en una fórmula usada para predecir el número de faltas que quedan. Un modelo de estimación es el paraguas para denominar a un conjunto de estimadores basados en los mismos prerequisites.

Las técnicas de lectura más populares son la lectura Ad-hoc y la lectura basada en listas de comprobación. Ambas técnicas pueden aplicarse sobre cualquier artefacto software, no solo sobre código. Además de estas dos técnicas existen otras que, aunque menos utilizadas en la industria, intentan abordar los problemas de la lectura con listas y la lectura ad-hoc: Lectura por Abstracción, Revisión Activa de Diseño y Lectura Basada en Escenarios. Esta última se trata de una familia de técnicas a la que pertenecen: Lectura Basada en Defectos y Lectura Basada en Perspectivas. Veamos en qué consisten cada una.

3.4.4.1 Lectura sin Checklists y con Checklists

En la técnica de **lectura Ad-hoc**, el producto software se entrega a los inspectores sin ninguna indicación o guía sobre cómo proceder con el producto ni qué buscar. Por eso la denominamos también como Lectura sin Checklists.

Sin embargo, que los participantes no cuenten con guías de qué buscar no significa que no escudriñen sistemáticamente el producto inspeccionado, ni tampoco que no tengan en mente el tipo de defecto que están buscando. Como ya hemos dicho antes, si no se sabe lo que se busca, es imposible encontrarlo. El término "ad-hoc" sólo se refiere al hecho de no proporcionar apoyo a los inspectores. En este caso la detección de los defectos depende completamente de las habilidades, conocimientos y experiencia del inspector.

Típicamente, el inspector deberá buscar secuencialmente los defectos típicos del producto que esté leyendo (y que hemos indicado más arriba). Por ejemplo, si se está inspeccionando unos requisitos, el inspector, buscará sistemática y secuencialmente defectos de corrección, de completud, de ambigüedad, etc...

Para practicar esta técnica, en el Anexo A aparece unos requisitos con defectos. Intenta buscarlos de acuerdo a lo indicado en el párrafo anterior. Sin guía alguna, simplemente utilizando la lista de criterios de calidad que debe cumplir unos requisitos que hemos indicado anteriormente.

La **lectura basada en Listas de Comprobación** (checklists, en inglés) proporciona un apoyo mayor mediante preguntas que los inspectores deben de responder mientras leen el artefacto. Es decir, esta técnica proporciona listas que ayudan al inspector a saber qué tipo de faltas buscar. Aunque una lista supone más ayuda que nada, esta técnica presenta la debilidad de que las preguntas proporcionan poco apoyo para ayudar a un inspector a entender el artefacto inspeccionado. Además, las preguntas son a menudo generales y no suficientemente adaptadas a un entorno de desarrollo particular. Finalmente, las preguntas en una lista de comprobación están limitadas a la detección de defectos de un tipo determinado, típicamente de corrección, puesto que las listas establecen errores universales (independientes del contexto y el problema).

3.4.4.1.1 Checklists para Requisitos y Diseño

Las listas de comprobación para requisitos contiene preguntas sobre los defectos típicos que suelen aparecer en los requisitos. Preguntas típicas que aparecen en las checklists de requisitos son:

- ✓ ¿Existen contradicciones en la especificación de los requisitos?
- ✓ ¿Resulta comprensible la especificación?
- ✓ ¿Está especificado el rendimiento?
- ✓ ¿Puede ser eliminado algún requisito? ¿Pueden juntarse dos requisitos?
- ✓ ¿Son redundantes o contradictorios?

- ✓ ¿Se han especificado todos los recursos hardware necesarios?
- ✓ ¿Se han especificado las interfaces externas necesarias?
- ✓ ¿Se han definido los criterios de aceptación para cada una de las funciones especificadas?

Nótese que las cinco primeras preguntas, corresponden simplemente a los criterios de calidad de los requisitos. Mientras que las tres últimas tratan olvidos típicos al especificar requisitos. Las dos que aparecen en primer lugar son comprobaciones sobre la especificación del hardware sobre el que correrá el futuro sistema y sobre cómo deberá interactuar con otros sistemas. La última comprueba que los requisitos contienen criterios de aceptación para cuando se realicen las pruebas de aceptación.

Los requisitos necesitan ser evaluados de forma crítica para prevenir errores. En esta fase radica la calidad del producto software desde la perspectiva del usuario. Si la evaluación en general es difícil, la de los requisitos en particular lo es más, debido a que lo que se evalúa es la definición del problema.

Con respecto al diseño, los objetivos principales de su evaluación estática son:

- Determinar si la solución elegida es la mejor de todas las opciones; es decir, si la opción es la más simple y la forma más fácil de realizar el trabajo.
- Determinar si la solución abarca todos los requisitos descritos en la especificación; es decir, si la solución elegida realizará la función encomendada al software.

Al igual que la evaluación de requisitos, la evaluación de diseño es crucial, ya que los defectos de diseño que queden y sean transmitidos al código, cuando sean detectados en fases más avanzadas del desarrollo, o incluso durante el uso, implicará un rediseño del sistema, con la subsiguiente re-codificación. Es decir, existirá una pérdida real de trabajo.

Veamos un ejemplo de preguntas para el diseño:

- ✓ ¿Cubre el diseño todos los requisitos funcionales?
- ✓ ¿Resulta ambigua la documentación del diseño?
- ✓ ¿Se ha aplicado la notación de diseño correctamente?
- ✓ ¿Se han definido correctamente las interfaces entre elementos del diseño?
- ✓ ¿Es el diseño suficientemente detallado como para que sea posible implementarlo en el lenguaje de programación elegido?

En el Anexo B aparecen listas de comprobación para diferentes productos del desarrollo proporcionadas por la empresa Construx. Intenta revisar ahora de nuevo los requisitos del Anexo A, esta vez usando las listas de comprobación del Anexo B. Esta misma especificación de requisitos se usa más tarde con otra técnica de lectura. Será entonces cuando aportemos la solución sobre qué defectos contienen estos requisitos.

3.4.4.1.2 Checklists para Código

Las listas para código han sido mucho más estudiadas que para otros artefactos. Así hay listas para distintos lenguajes de programación, para distintas partes de código, etc.

Una típica lista de comprobación para código contendrá varias partes (una por los distintos tipos de defecto que se buscan) cada una con preguntas sobre defectos universales y típicos. Por ejemplo:

- Lógica del programa:

- ✓ ¿Es correcta la lógica del programa?
- ✓ ¿Está completa la lógica del programa?, es decir, ¿está todo correctamente especificado sin faltar ninguna función?
- Interfaces Internas:
 - ✓ ¿Es igual el número de parámetros recibidos por el módulo a probar al número de argumentos enviados?, además, ¿el orden es correcto?
 - ✓ ¿Los atributos (por ejemplo, tipo y tamaño) de cada parámetro recibido por el módulo a probar coinciden con los atributos del argumento correspondiente?
 - ✓ ¿Coinciden las unidades en las que se expresan parámetros y argumentos?. Por ejemplo, argumentos en grados y parámetros en radianes. ¿Altera el módulo un parámetro de sólo lectura? ¿Son consistentes las definiciones de variables globales entre los módulos?
- Interfaces Externas:
 - ✓ ¿Se declaran los ficheros con todos sus atributos de forma correcta?
 - ✓ ¿Se abren todos los ficheros antes de usarlos?
 - ✓ ¿Coincide el formato del fichero con el formato especificado en la lectura? ¿Se manejan correctamente las condiciones de fin de fichero?. ¿Se los libera de memoria?
 - ✓ ¿Se manejan correctamente los errores de entrada/salida?
- Datos:
 - Referencias de datos. Se refieren a los accesos que se realizan a los mismos. Ejemplos típicos son: Utilizar variables no inicializadas
 - ✓ Salirse del límite de las matrices y vectores
 - ✓ Superar el límite de tamaño de una cadena
 - Declaración de datos. El propósito es comprobar que todas las definiciones de los datos locales son correctas. Por ejemplo:
 - ✓ Comprobar que no hay dos variables con el mismo nombre
 - ✓ Comprobar que todas las variables estén declaradas
 - ✓ Comprobar que las longitudes y tipos de las variables sean correctos.
 - Cálculo. Intenta localizar errores derivados del uso de las variables. Por ejemplo:
 - ✓ Comprobar que no se producen overflow o underflow (valores fuera de rango, por encima o por debajo) en los cálculos o divisiones por cero.
 - Comparación. Intenta localizar errores en las comparaciones realizadas en instrucciones tipo *If-Then-Else*, *While*, etc. Por ejemplo:
 - ✓ Comprobar que no existen comparaciones entre variables con diferentes tipos de datos o si las variables tienen diferente longitud.
 - ✓ Comprobar si los operadores utilizados en la comparación son correctos, si utilizan operadores booleanos comprobar si los operandos usados son booleanos, etc.

En el Anexo C se proporcionan distintas listas de comprobación para diversas partes y características del código. En el Anexo D tienes un programa y una pequeña lista de comprobación de código para que te ejercites buscando defectos. Deberías detectar al menos un par de defectos, al menos. Más adelante usaremos este mismo programa para practicar con otra técnica y será entonces cuando proporcionaremos la lista de defectos de este programa.

3.4.4.2 Lectura por Abstracción Sucesiva

La **Lectura por Abstracción Sucesiva** sirve para inspeccionar código, y no otro tipo de artefacto como requisitos o diseño. La idea sobre la que se basa esta técnica de lectura para detectar defectos es en la comparación entre la especificación del programa (es decir, el texto que describe lo que el programa debería hacer) y lo que el programa hace realmente. Naturalmente, todos aquellos puntos donde no coincida lo que el programa debería hacer con lo que el programa hace es un defecto.

Dado que comparar código con texto (la especificación) es inapropiado pues se estarían comparando unidades distintas (peras con manzanas), se hace necesario convertir ambos artefactos a las mismas “unidades”. Lo que se hace, entonces, es convertir el programa en una especificación en forma de texto. De modo que podamos comparar especificación (texto) con especificación (texto).

Obtener una especificación a partir de un código significa recorrer el camino de la programación en sentido inverso. El sentido directo es obtener un código a partir de una especificación. Este camino se recorre en una serie de pasos (que a menudo quedan ocultos en la mente del programador y no se hacen explícitos, pero que no por eso dejan de existir). El recorrido directo del camino de la especificación al código consta de los siguientes pasos:

1. Leer la especificación varias veces hasta que el programador ha entendido lo que el código debe hacer.
2. Descomponer la tarea que el programa debe hacer en subtareas, que típicamente se corresponderán con las funciones o módulos que compondrán el programa. Esta descomposición muestra la relación entre funciones, que no siempre es secuencial, sino típicamente en forma de árbol: Funciones alternativas que se ejecutarán dependiendo de alguna condición; Funciones suplementarias que se ejecutarán siempre una si se ejecuta la otra; etc.
3. Para cada uno de estas subtareas (funciones):
 - 3.1. Hacer una descripción sistemática (típicamente en pseudocódigo) de cómo realizar la tarea. En esta descripción se pueden ya apreciar las principales estructuras de las que constará la función (bucles, condicionales, etc.)
 - 3.2. Programar cada línea de código que compone la función

Como puede observarse en este proceso, el programador trabaja desde la especificación por descomposiciones sucesivas. Es decir, dividiendo una cosa compleja (la especificación) en cosas cada vez mas sencillas (primero las funciones, luego las estructuras elementales de los programas, finalmente las líneas de código). Este tipo de tarea se realiza de arriba hacia abajo, partiendo de la especificación (arriba o nodo raíz) y llegando a n líneas de código (abajo o nodos hojas).

Pues bien, si queremos obtener una especificación a partir de un código deberemos hacer este mismo recorrido pero en sentido contrario: de abajo hacia arriba. Por tanto, deberemos comenzar con las líneas de código, agruparlas en estructuras elementales, y éstas en funciones y éstas en un todo que será la descripción del comportamiento del programa. En este caso no estamos trabajando descomponiendo, sino componiendo. La tarea que se realiza es de abstracción. De ahí el nombre de la técnica: abstracción sucesiva (ir sucesivamente –ascendiendo en niveles cada vez superiores– abstrayendo qué hace cada elemento –primero las líneas, luego las estructuras, finalmente las funciones).

Esta técnica requiere que el inspector lea una serie de líneas de código y que abstraiga la función que estas líneas computan. El inspector debe repetir este procedimiento hasta que la función final del código que se está inspeccionando se haya abstraído y pueda compararse con la especificación original del programa.

Más concretamente, el proceso que se debe seguir para realizar la abstracción sucesiva es el siguiente:

1. Leer por encima el código para tener una idea general del programa.
2. Determinar las dependencias entre las funciones individuales del código fuente (quién llama a quién). Para esto puede usarse un árbol de llamadas para representar tales dependencias comenzando por las hojas (funciones que no llaman a nadie) y acabando por la raíz.(función principal).
3. Comprender qué hace cada función. Para ello se deberá: Entender la estructura de cada función individual identificando las estructuras elementales (secuencias, condicionales, bucles, etc.) y marcándolas; Combinar las estructuras elementales para formar estructuras más grandes hasta que se haya entendido la función entera. Es decir, para cada función y comenzando desde las funciones hoja y acabando por la raíz:
 - 3.1. Identificar las estructuras elementales de cada función y marcarlas de la más interna a la más externa.
 - 3.2. Determinar el significado de cada estructura comenzando con la más interna. Para ello pueden seguirse las siguientes recomendaciones:
 - Usar los números de línea (líneas x-y) para identificar las líneas que abarca una estructura e indicar a su lado qué hace.
 - Evitar utilizar conocimiento implícito que no resida en la estructura (valores iniciales, entradas o valores de parámetros).
 - Usar principios generalmente aceptados del dominio de aplicación para mantener la descripción breve y entendible (“búsqueda en profundidad” en lugar de describir lo que hace la búsqueda en profundidad).
 - 3.3. Especificar el comportamiento de la función entera. Es decir, utilizar la información de los pasos 3.1 y 3.2 para entender qué hace la función y describirlo en texto.
4. Combinar el comportamiento de cada función y las relaciones entre ellas para entender el comportamiento del programa entero. El comportamiento deberá describirse en texto. Esta descripción es la especificación del programa obtenida a partir del código.
5. Comparar la especificación obtenida con la especificación original del programa. Cada punto donde especificación original y especificación generada a partir del código no coincidan es un defecto. Anotar los defectos en una lista.

Veamos un breve ejemplo para entender mejor la técnica. El programa “count” ya lo conoces pues lo has usado para practicar con la técnica de lectura con listas de comprobación. Centrémonos en el siguiente trozo de especificación original:

Si alguno de los ficheros que se le pasa como argumento no existe, aparece por la salida de error el mensaje de error correspondiente y se continúa procesando el resto de los ficheros.

Que se implementa mediante las siguientes líneas de código entresacadas del programa “count”.

```
if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {  
    fprintf (stdout, "can't open %s\n", argv[i]);  
    exit(1)  
}
```

La abstracción de estas líneas sería:

Línea 1 Si no hay ficheros que coincidan con el argumento (fp=fopen(argv[i], "r")) == NULL)

Línea 2 Se imprime por la salida estándar (stdout) el mensaje de que no se puede abrir el fichero (indicando el nombre del fichero)

Línea 3 Se sale del programa

Que se corresponde con la siguiente descripción:

Se proporcionan argumentos, pero no hay ficheros con nombres correspondientes a los argumentos. En este caso, el programa se para con un mensaje de error que sale por la salida estándar.

Nótese que las especificaciones no coinciden en algunos puntos. En la Tabla 2 se ve la comparación y aparecen en negrita señaladas las diferencias.

ESPECIFICACIÓN ORIGINAL	ESPECIFICACIÓN OBTENIDA POR ABSTRACCIÓN
Si alguno de los ficheros que se le pasa como argumento no existe, aparece <i>por la salida de error</i> el mensaje de error correspondiente y <u>se continúa</u> procesando el resto de los ficheros.	Se proporcionan argumentos, pero no hay ficheros con nombres correspondientes a los argumentos. En este caso, <u>el programa se para</u> con un mensaje de error que sale por la <i>salida estándar</i> .

Tabla 2. Diferencias entre especificación original y especificación abstraída

Por tanto, se detecta la siguiente falta en el código:

Falta en línea 2: La llamada a “fprintf” usa “stdout” en lugar de “stderr”.

Causa fallo: Los mensajes de error aparecen en la salida estándar (stdout) en lugar de la salida estándar de errores (stderr).

En el Anexo E se muestra la aplicación de esta técnica al programa completo “count”. Intenta practicar con este ejercicio realizando tú mismo la abstracción y detección de faltas antes de mirar la solución proporcionada. Además, el Anexo F y el Anexo G contienen dos programas más y su lista de defectos para que el lector se ejercite hasta dominar esta técnica de lectura.

3.4.4.3 Lectura Activa de Diseño

La **Revisión Activa de Diseño** sólo es aplicable sobre el diseño, y no sobre código o requisitos. Esta técnica propone una cierta variación metodológica al proceso básico de inspección. En concreto requiere que los inspectores tomen un papel más activo del habitual, solicitando que hagan aseveraciones sobre determinadas partes del diseño, en lugar de simplemente señalar defectos. La Revisión Activa de Diseño considera que la inspección debe explotar más la interacción entre autor e inspector, y que la inspección tradicional limita demasiado esta interacción. En esta técnica sólo se definen dos roles: un inspector que tiene la responsabilidad de encontrar defectos, y un diseñador que es el autor del diseño que se está examinando.

El proceso de la Inspección Activa de Diseño consta de tres pasos. Comienza con una fase de inicio donde el diseñador presenta una visión general del diseño que se pretende inspeccionar y también se establece el calendario. La segunda fase es la de detección, para la cual el autor proporciona un cuestionario para guiar a los inspectores. Las preguntas sólo deben poderse responder tras un estudio detallado y cuidadoso del documento de diseño. Esto es, los inspectores deben elaborar una respuesta, en lugar de simplemente responder sí o no. Las preguntas refuerzan un papel activo de inspección puesto que deben realizar afirmaciones sobre decisiones de diseño. Por ejemplo, se le puede pedir al inspector que escriba un segmento de programa que implemente un diseño particular al inspeccionar un diseño de bajo nivel. El último paso es la recolección de defectos que se realiza en una reunión de inspección. Sin embargo, el meeting se subdivide en pequeñas reuniones especializadas, cada una se concentra en una propiedad de calidad del artefacto. Por ejemplo, comprobar la consistencia entre las asunciones y las funciones, es decir, determinar si las asunciones son consistentes y detalladas lo suficiente para asegurar que las funciones puedan ser correctamente implementadas y usadas.

3.4.4.4 Lectura Basada en Escenarios

La Lectura Basada en Escenarios proporciona guías al inspector (escenarios que pueden ser preguntas pero también alguna descripción más detallada) sobre cómo realizar el examen del artefacto. Principalmente, un escenario limita la atención de un inspector en la detección de defectos particulares definidos por la guía. Dado que inspectores diferentes pueden usar escenarios distintos, y como cada escenario se centra en diferentes tipos de defectos, se espera que el equipo de inspección resulte más efectivo en su globalidad. Existen dos técnicas de lectura basada en

escenarios: Lectura Basada en Defectos y Lectura Basada en Perspectivas. Ambas técnicas examinan documentos de requisitos.

La **Lectura Basada en Defectos** focaliza cada inspector en una clase distinta de defecto mientras inspecciona un documento de requisitos. Contestar a las preguntas planteadas en el escenario ayuda al inspector a encontrar defectos de determinado tipo.

La **Lectura Basada en Perspectiva** establece que un producto software debería inspeccionarse bajo las perspectivas de los distintos participantes en un proyecto de desarrollo. Las perspectivas dependen del papel que los distintos participantes tienen en el proyecto. Para cada perspectiva se definen uno o varios escenarios consistentes en actividades repetibles que un inspector debe realizar y preguntas que el inspector debe responder.

Por ejemplo, diseñar los casos de prueba es una actividad típicamente realizada por el validador. Así pues, un inspector leyendo desde la perspectiva de un validador debe pensar en la obtención de los casos de prueba. Mientras que un inspector ejerciendo la perspectiva del diseñador, deberá leer ese mismo artefacto pensando en que va a tener que realizar el diseño.

En el Anexo H, Anexo I y Anexo J se muestran respectivamente las perspectivas del diseñador, validador y usuario respectivamente para que las uses con los requisitos del Anexo A. Cada perspectiva descubre defectos distintos. En dichos anexos te proporcionamos también la solución de qué defectos son encontrados desde cada perspectiva. Finalmente, y a modo de resumen el Anexo K puedes encontrar una tabla con todos los defectos de los requisitos del Anexo A.

4. TÉCNICAS DE EVALUACIÓN DINÁMICA

4.1 Características y Fases de la Prueba

Como se ha indicado anteriormente, a la aplicación de técnicas de evaluación dinámicas se le denomina también *prueba* del software.

La Figura 2 muestra el contexto en el que se realiza la prueba de software. Concretamente la Prueba de software se puede definir como una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas (configuración de la prueba), registrándose los resultados obtenidos. Seguidamente se realiza un proceso de Evaluación en el que los resultados obtenidos se comparan con los resultados esperados para localizar fallos en el software. Estos fallos conducen a un proceso de Depuración en el que es necesario identificar la falta asociada con cada fallo y corregirla, pudiendo dar lugar a una nueva prueba. Como resultado final se puede obtener una determinada Predicción de Fiabilidad, tal como se indicó anteriormente, o un cierto nivel de confianza en el software probado.

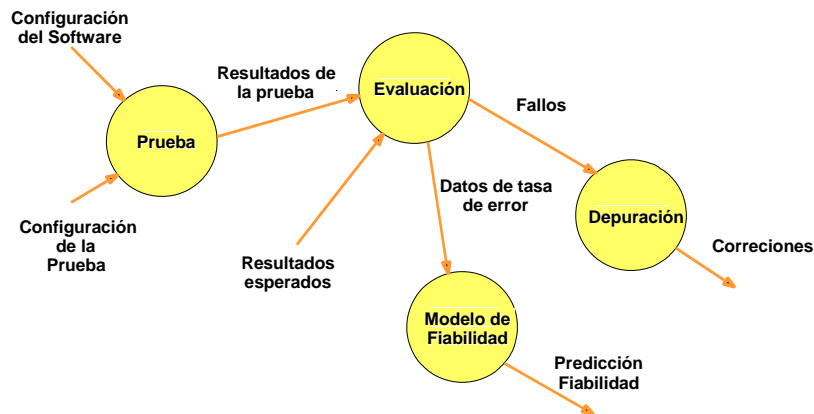


Figura 2. Contexto de la Prueba de Software

El objetivo de las pruebas no es asegurar la ausencia de defectos en un software, únicamente puede demostrar que existen defectos en el software. Nuestro objetivo es pues, diseñar pruebas que sistemáticamente saquen a la luz diferentes clases de errores, haciendolo con la menor cantidad de tiempo y esfuerzo.

Para ser más eficaces (es decir, con más alta probabilidad de encontrar errores), las pruebas deberían ser realizadas por un equipo independiente al que realizó el software. El ingeniero de software que creó el sistema no es el más adecuado para llevar a cabo las pruebas de dicho software, ya que inconscientemente tratará de demostrar que el software funciona, y no que no lo hace, por lo que la prueba puede tener menos éxito.

Una prueba de software, comparando los resultados obtenidos con los esperados. A continuación se presentan algunas características de una buena prueba:

- Una buena prueba ha de tener una alta probabilidad de encontrar un fallo. Para alcanzar este objetivo el responsable de la prueba debe entender el software e intentar desarrollar una *imagen mental* de cómo podría fallar.
- Una buena prueba debe centrarse en dos objetivos: 1) *probar si el software no hace lo que debe hacer*, y 2) *probar si el software hace lo que no debe hacer*.

- Una buena prueba no debe ser redundante. El tiempo y los recursos son limitados, así que *todas las pruebas deberían tener un propósito diferente.*
- Una buena prueba debería ser la “mejor de la cosecha”. Esto es, se debería emplear la prueba que tenga la *más alta probabilidad de descubrir una clase entera de errores.*
- Una buena prueba no debería ser ni demasiado sencilla ni demasiado compleja, pero si se quieren combinar varias pruebas a la vez se pueden enmascarar errores, por lo que en general, *cada prueba debería realizarse separadamente.*

Veamos ahora cuáles son las tareas a realizar para probar un software:

1. *Diseño de las pruebas.* Esto es, identificación de la técnica o técnicas de pruebas que se utilizarán para probar el software. Distintas técnicas de prueba ejercitan diferentes criterios como guía para realizar las pruebas. Seguidamente veremos algunas de estas técnicas.
2. *Generación de los casos de prueba.* Los casos de prueba representan los datos que se utilizarán como entrada para ejecutar el software a probar. Más concretamente los casos de prueba determinan un conjunto de entradas, condiciones de ejecución y resultados esperados para un objetivo particular. Como veremos posteriormente, cada técnica de pruebas proporciona unos criterios distintos para generar estos casos o datos de prueba. Por lo tanto, durante la tarea de generación de casos de prueba, se han de confeccionar los distintos casos de prueba según la técnica o técnicas identificadas previamente. La generación de cada caso de prueba debe ir acompañada del resultado que ha de producir el software al ejecutar dicho caso (como se verá más adelante, esto es necesario para detectar un posible fallo en el programa).
3. *Definición de los procedimientos de la prueba.* Esto es, especificación de cómo se va a llevar a cabo el proceso, quién lo va a realizar, cuándo, ...
4. *Ejecución de la prueba,* aplicando los casos de prueba generados previamente e identificando los posibles fallos producidos al comparar los resultados esperados con los resultados obtenidos.
5. *Realización de un informe de la prueba,* con el resultado de la ejecución de las pruebas, qué casos de prueba pasaron satisfactoriamente, cuáles no, y qué fallos se detectaron.

Tras estas tareas es necesario realizar un proceso de depuración de las faltas asociadas a los fallos identificados. Nosotros nos centraremos en el segundo paso, explicando cómo distintas técnicas de pruebas pueden proporcionar criterios para generar distintos datos de prueba.

4.2 Técnicas de Prueba

Como se ha indicado anteriormente, las técnicas de evaluación dinámica o prueba proporcionan distintos criterios para generar casos de prueba que provoquen fallos en los programas. Estas técnicas se agrupan en:

- *Técnicas de caja blanca o estructurales,* que se basan en un minucioso examen de los detalles procedimentales del código a evaluar, por lo que es necesario conocer la lógica del programa.
- *Técnicas de caja negra o funcionales,* que realizan pruebas sobre la interfaz del programa a probar, entendiendo por interfaz las entradas y salidas de dicho programa. No es necesario conocer la lógica del programa, únicamente la funcionalidad que debe realizar.

La Figura 3 representa gráficamente la filosofía de las pruebas de caja blanca y caja negra. Como se puede observar las pruebas de caja blanca necesitan conocer los detalles procedimentales del código, mientras que las de caja negra únicamente necesitan saber el objetivo o funcionalidad que el código ha de proporcionar.

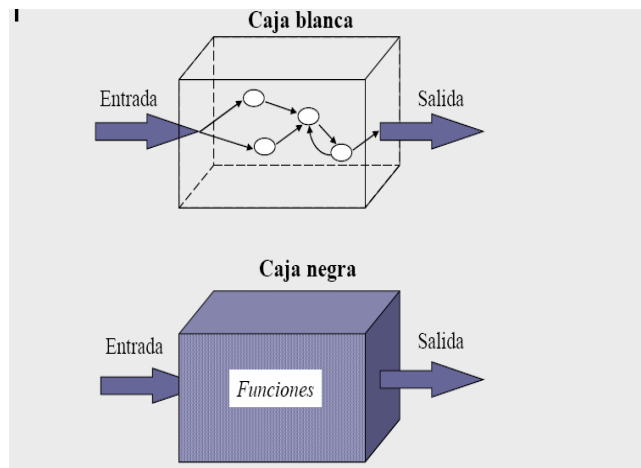


Figura 3. Representación de pruebas de Caja Blanca y Caja Negra

A primera vista parecería que una prueba de caja blanca completa nos llevaría a disponer de un código perfectamente correcto. De hecho esto ocurriría si se han probado todos los posibles caminos por los que puede pasar el flujo de control de un programa. Sin embargo, para programas de cierta envergadura, el número de casos de prueba que habría que generar sería excesivo, nótese que el número de caminos incrementa exponencialmente a medida que el número de sentencias condicionales y bucles aumenta. Sin embargo, este tipo de prueba no se desecha como impracticable. Se pueden elegir y ejercitar ciertos caminos representativos de un programa.

Por su parte, tampoco sería factible en una prueba de caja negra probar todas y cada una de las posibles entradas a un programa, por lo que análogamente a como ocurría con las técnicas de caja blanca, se seleccionan un conjunto representativo de entradas y se generan los correspondientes casos de prueba, con el fin de provocar fallos en los programas.

En realidad estos dos tipos de técnicas son técnicas complementarias que han de aplicarse al realizar una prueba dinámica, ya que pueden ayudar a identificar distintos tipos de faltas en un programa.

A continuación, se describen en detalle los procedimientos propuestos por ambos tipos de técnicas para generar casos de prueba.

4.2.1 Pruebas de Caja Blanca o Estructurales

A este tipo de técnicas se le conoce también como Técnicas de Caja Transparente o de Cristal. Este método se centra en cómo diseñar los casos de prueba atendiendo al *comportamiento interno* y la *estructura del programa*. Se examina así la lógica interna del programa sin considerar los aspectos de rendimiento.

El objetivo de la técnica es diseñar casos de prueba para que se ejecuten, al menos una vez, todas las sentencias del programa, y todas las condiciones tanto en su vertiente verdadera como falsa.

Como se ha indicado ya, puede ser impracticable realizar una prueba exhaustiva de todos los caminos de un programa. Por ello se han definido distintos criterios de cobertura lógica, que permiten decidir qué sentencias o caminos se deben examinar con los casos de prueba. Estos criterios son:

- *Cobertura de Sentencias*: Se escriben casos de prueba suficientes para que cada sentencia en el programa se ejecute, al menos, una vez.
- *Cobertura de Decisión*: Se escriben casos de prueba suficientes para que cada decisión en el programa se ejecute una vez con resultado verdadero y otra con el falso.
- *Cobertura de Condiciones*: Se escriben casos de prueba suficientes para que cada condición en una decisión tenga una vez resultado verdadero y otra falso.
- *Cobertura Decisión/Condición*: Se escriben casos de prueba suficientes para que cada condición en una decisión tome todas las posibles salidas, al menos una vez, y cada decisión tome todas las posibles salidas, al menos una vez.
- *Cobertura de Condición Múltiple*: Se escriben casos de prueba suficientes para que todas las combinaciones posibles de resultados de cada condición se invoquen al menos una vez.
- *Cobertura de Caminos*: Se escriben casos de prueba suficientes para que se ejecuten todos los caminos de un programa. Entendiendo camino como una secuencia de sentencias encadenadas desde la entrada del programa hasta su salida.

Este último criterio es el que se va a estudiar.

4.2.1.1 Cobertura de Caminos

La aplicación de este criterio de cobertura asegura que los casos de prueba diseñados permiten que todas las sentencias del programa sean ejecutadas al menos una vez y que las condiciones sean probadas tanto para su valor verdadero como falso.

Una de las técnicas empleadas para aplicar este criterio de cobertura es la **Prueba del Camino Básico**. Esta técnica se basa en obtener una medida de la complejidad del diseño procedimental de un programa (o de la lógica del programa). Esta medida es la complejidad ciclomática de McCabe, y representa un límite superior para el número de casos de prueba que se deben realizar para asegurar que se ejecuta cada camino del programa.

Los pasos a realizar para aplicar esta técnica son:

- Representar el programa en un grafo de flujo
- Calcular la complejidad ciclomática
- Determinar el conjunto básico de caminos independientes
- Derivar los casos de prueba que fuerzan la ejecución de cada camino.

A continuación, se detallan cada uno de estos pasos.

4.2.1.1.1 Representar el programa en un grafo de flujo

El grafo de flujo se utiliza para representar flujo de control lógico de un programa. Para ello se utilizan los tres elementos siguientes:

- *Nodos*: representan cero, una o varias sentencias en secuencia. Cada nodo comprende como máximo una sentencia de decisión (bifurcación).
- *Aristas*: líneas que unen dos nodos.

- *Regiones*: áreas delimitadas por aristas y nodos. Cuando se contabilizan las regiones de un programa debe incluirse el área externa como una región más.
- *Nodos predicado*: cuando en una condición aparecen uno o más operadores lógicos (AND, OR, XOR, ...) se crea un nodo distinto por cada una de las condiciones simples. Cada nodo generado de esta forma se denomina nodo predicado. La Figura 4 muestra un ejemplo de condición múltiple.

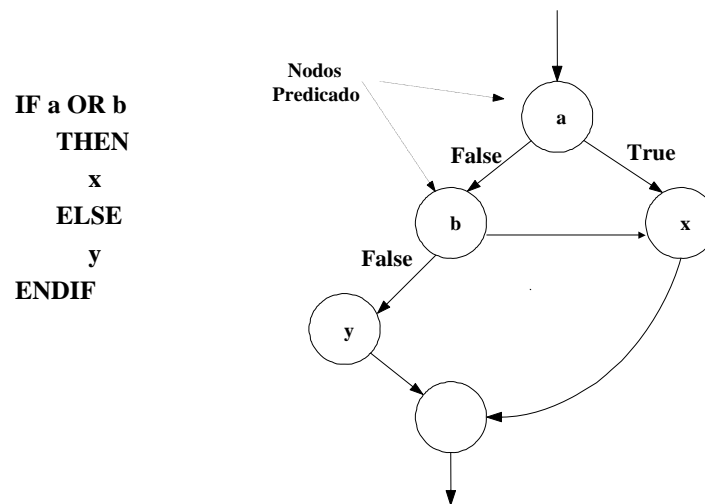


Figura 4. Representación de condición múltiple

Así, cada construcción lógica de un programa tiene una representación. La Figura 5 muestra dichas representaciones.

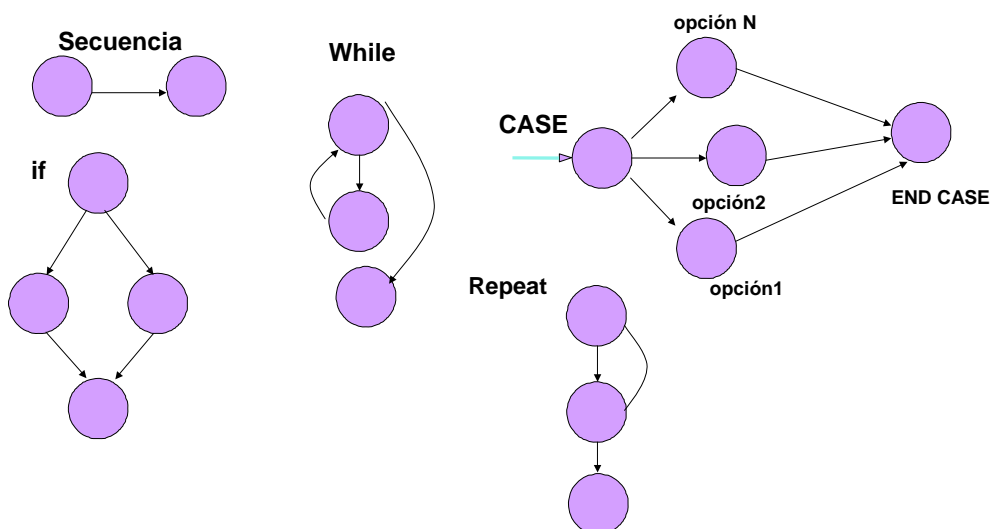


Figura 5. Representación en grafo de flujo de las estructuras lógicas de un programa

La Figura 6 muestra un grafo de flujo del diagrama de módulos correspondiente. Nótese cómo la estructura principal corresponde a un *while*, y dentro del bucle se encuentran anidados dos constructores *if*.

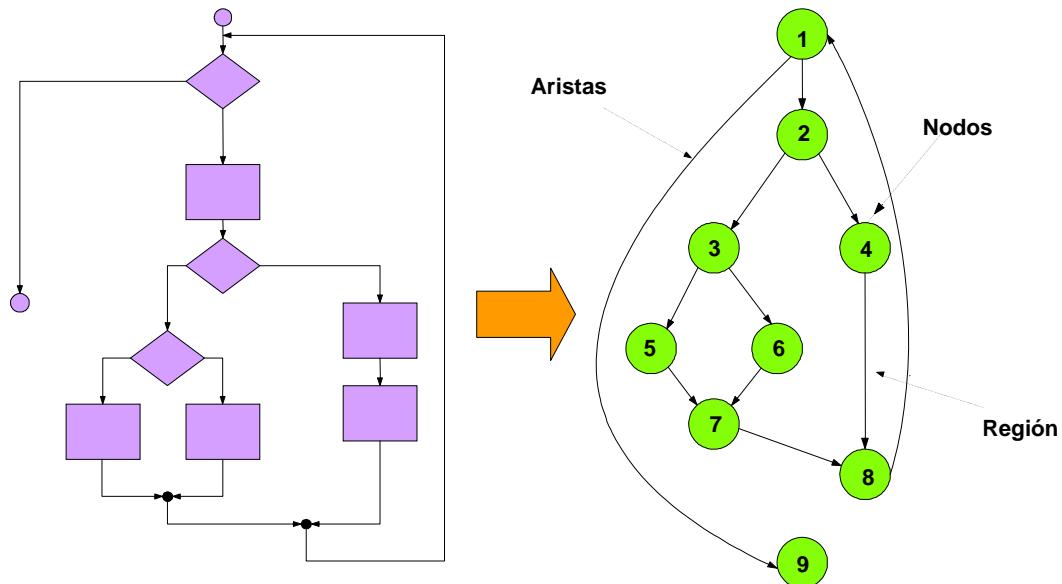


Figura 6. Ejemplo de grafo de flujo correspondiente a un diagrama de módulos

4.2.1.1.2 Calcular la complejidad ciclomática

La complejidad ciclomática es una métrica del software que proporciona una medida cuantitativa de la complejidad lógica de un programa. En el contexto del método de prueba del camino básico, el valor de la complejidad ciclomática define el número de caminos independientes de dicho programa, y por lo tanto, el número de casos de prueba a realizar. Posteriormente veremos cómo se identifican esos caminos, pero primero veamos cómo se puede calcular la complejidad ciclomática a partir de un grafo de flujo, para obtener el número de caminos a identificar.

Existen varias formas de calcular la complejidad ciclomática de un programa a partir de un grafo de flujo:

1. El número de regiones del grafo coincide con la complejidad ciclomática, $V(G)$.
2. La complejidad ciclomática, $V(G)$, de un grafo de flujo G se define como

$$V(G) = \text{Aristas} - \text{Nodos} + 2$$

3. La complejidad ciclomática, $V(G)$, de un grafo de flujo G se define como

$$V(G) = \text{Nodos Predicado} + 1$$

La Figura 7 representa, por ejemplo, las cuatro regiones del grafo de flujo, obteniéndose así la complejidad ciclomática de 4. Análogamente se puede calcular el número de aristas y nodos predicados para confirmar la complejidad ciclomática. Así:

$$V(G) = \text{Número de regiones} = 4$$

$$V(G) = \text{Aristas} - \text{Nodos} + 2 = 11 - 9 + 2 = 4$$

$$V(G) = \text{Nodos Predicado} + 1 = 3 + 1 = 4$$

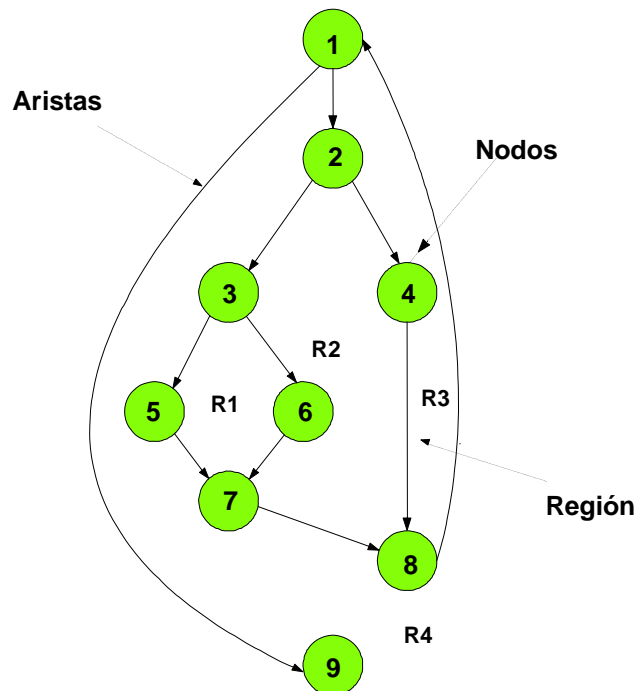


Figura 7. Número de regiones del grafo de flujo

Esta complejidad ciclomática determina el número de casos de prueba que deben ejecutarse para garantizar que todas las sentencias de un programa se han ejecutado al menos una vez, y que cada condición se habrá ejecutado en sus vertientes verdadera y falsa. Veamos ahora, cómo se identifican estos caminos.

4.2.1.1.3 Determinar el conjunto básico de caminos independientes

Un camino independiente es cualquier camino del programa que introduce, por lo menos, un nuevo conjunto de sentencias de proceso o una condición, respecto a los caminos existentes. En términos del diagrama de flujo, un camino independiente está constituido por lo menos por una arista que no haya sido recorrida anteriormente a la definición del camino. En la identificación de los distintos caminos de un programa para probar se debe tener en cuenta que cada nuevo camino debe tener el mínimo número de sentencias nuevas o condiciones nuevas respecto a los que ya existen. De esta manera se intenta que el proceso de depuración sea más sencillo.

El conjunto de caminos independientes de un grafo no es único. No obstante, a continuación, se muestran algunas heurísticas para identificar dichos caminos:

- (a) Elegir un *camino principal* que represente una función válida que no sea un tratamiento de error. Debe intentar elegirse el camino que atraviese el máximo número de decisiones en el grafo.
- (b) Identificar el segundo camino mediante la localización de la *primera decisión* en el camino de la línea básica alternando su resultado mientras se mantiene el máximo número de decisiones originales del camino inicial.
- (c) Identificar un tercer camino, colocando la primera decisión en su valor original a la vez que se altera la *segunda decisión* del camino básico, mientras se intenta mantener el resto de decisiones originales.
- (d) Continuar el proceso hasta haber conseguido *tratar todas las decisiones*, intentando mantener como en su origen el resto de ellas.

Este método permite obtener $V(G)$ caminos independientes cubriendo el criterio de cobertura de decisión y sentencia.

Así por ejemplo, para la el grafo de la Figura 7 los cuatro posibles caminos independientes generados serían:

Camino 1: 1-10

Camino 2: 1-2-4-8-1-9

Camino 3: 1-2-3-5-7-8-1-9

Camino 4: 1-2-5-6-7-8-1-9

Estos cuatro caminos constituyen el *camino básico* para el grafo de flujo correspondiente.

4.2.1.1.4 Derivar los casos de prueba que fuerzan la ejecución de cada camino.

El último paso es construir los casos de prueba que fuerzan la ejecución de cada camino. Una forma de representar el conjunto de casos de prueba es como se muestra en la Tabla 3.

Número del Camino	Caso de Prueba	Resultado Esperado

Tabla 3. Posible representación de casos de prueba para pruebas estructurales

En el Anexo L se encuentra un posible ejemplo de pruebas de Caja Blanca para que los alumnos trabajen con él junto con su solución. En el Anexo N se muestra un ejercicio propuesto para que los alumnos se ejerciten en esta técnica de pruebas. El código correspondiente ha sido ya utilizado para la evaluación con técnicas estáticas.

4.2.2 Pruebas de Caja Negra o Funcionales

También conocidas como Pruebas de Comportamiento, estas pruebas se basan en la *especificación* del programa o componente a ser probado para elaborar los casos de prueba. El componente se ve como una “Caja Negra” cuyo comportamiento sólo puede ser determinado estudiando sus entradas y las salidas obtenidas a partir de ellas. No obstante, como el estudio de todas las posibles entradas y salidas de un programa sería impracticable se selecciona un conjunto de ellas sobre las que se realizan las pruebas. Para seleccionar el conjunto de entradas y salidas sobre las que trabajar, hay que tener en cuenta que en todo programa existe un conjunto de entradas que causan un comportamiento erróneo en nuestro sistema, y como consecuencia producen una serie de salidas que revelan la presencia de defectos. Entonces, dado que la prueba exhaustiva es imposible, el objetivo final es pues, encontrar una serie de datos de entrada cuya probabilidad de pertenecer al conjunto de entradas que causan dicho comportamiento erróneo sea lo más alto posible.

Al igual que ocurriría con las técnicas de Caja Blanca, para confeccionar los casos de prueba de Caja Negra existen distintos criterios. Algunos de ellos son:

- Particiones de Equivalencia.
- Análisis de Valores Límite.
- Métodos Basados en Grafos.
- Pruebas de Comparación.
- Análisis Causa-Efecto.

De ellas, las técnicas que estudiaremos son las dos primeras, esto es, Particiones de Equivalencia y Análisis de Valores Límite.

4.2.2.1 Particiones de Equivalencia

La partición de equivalencia es un método de prueba de Caja Negra que divide el campo de entrada de un programa en *clases de datos* de los que se pueden derivar casos de prueba. La partición equivalente se dirige a una definición de casos de prueba que descubran *clases de errores*, reduciendo así el número total de casos de prueba que hay que desarrollar.

En otras palabras, este método intenta dividir el dominio de entrada de un programa en un número finito de *clases de equivalencia*. De tal modo que se pueda asumir razonablemente que una prueba realizada con un valor representativo de cada clase es equivalente a una prueba realizada con cualquier otro valor de dicha clase. Esto quiere decir que si el caso de prueba correspondiente a una clase de equivalencia detecta un error, el resto de los casos de prueba de dicha clase de equivalencia deben detectar el mismo error. Y viceversa, si un caso de prueba no ha detectado ningún error, es de esperar que ninguno de los casos de prueba correspondientes a la misma clase de equivalencia encuentre ningún error.

El diseño de casos de prueba según esta técnica consta de dos pasos:

1. Identificar las clases de equivalencia.
2. Identificar los casos de prueba.

4.2.2.1.1 Identificar las clases de equivalencia

Una clase de equivalencia representa un conjunto de estados válidos y no válidos para las condiciones de entrada de un programa. Las clases de equivalencia se identifican examinando cada condición de entrada (normalmente una frase en la especificación) y dividiéndola en dos o más grupos. Se definen dos tipos de clases de equivalencia, las *clases de equivalencia válidas*, que representan entradas válidas al programa, y las *clases de equivalencia no válidas*, que representan valores de entrada erróneos. Estas clases se pueden representar en una tabla como la Tabla 4.

Condición Externa	Clases de Equivalencia Válidas	Clases de Equivalencia No Válidas

Tabla 4. Tabla para la identificación de clases de equivalencia

En función de cuál sea la condición de entrada se pueden seguir las siguientes pautas identificar las clases de equivalencia correspondientes:

- Si una condición de entrada especifica un *rango de valores*, identificar una clase de equivalencia válida y dos clases no válidas. Por ejemplo, si un contador puede ir de 1 a 999, la clase válida sería “ $1 \leq \text{contador} \leq 999$ ”. Mientras que las clases no válidas serían “ $\text{contador} < 1$ ” y “ $\text{contador} > 999$ ”
- Si una condición de entrada especifica un *valor o número de valores*, identificar una clase válida y dos clases no válidas. Por ejemplo, si tenemos que puede haber desde uno hasta seis propietarios en la vida de un coche. Habrá una clase válida y dos no válidas: “no hay propietarios” y “más de seis propietarios”.
- Si una condición de entrada especifica un conjunto de valores de entrada, identificar una clase de equivalencia válida y una no válida. Sin embargo, si hay razones para creer que cada uno de los miembros del conjunto será tratado de distinto modo por el programa, identificar una clase válida por cada miembro y una clase no válida. Por ejemplo, el tipo de un vehículo puede ser: autobús, camión, taxi, coche o moto. Habrá una clase válida por cada tipo de vehículo admitido, y la clase no válida estará formada por otro tipo de vehículo.
- Si una condición de entrada especifica una situación que debe ocurrir, esto es, es lógica, identificar una clase válida y una no válida. Por ejemplo, el primer carácter del identificador debe ser una letra. La clase válida sería “identificador que comienza con letra”, y la clase inválida sería “identificador que no comienza con letra”.
- En general, si hay alguna razón para creer que los elementos de una clase de equivalencia no se tratan de igual modo por el programa, dividir la clase de equivalencia entre clases de equivalencia más pequeñas para cada tipo de elementos.

4.2.2.1.2 Identificar los casos de prueba

El objetivo es minimizar el número de casos de prueba, así cada caso de prueba debe considerar tantas condiciones de entrada como sea posible. No obstante, es necesario realizar con cierto

cuidado los casos de prueba de manera que no se enmascaren faltas. Así, para crear los casos de prueba a partir de las clases de equivalencia se han de seguir los siguientes pasos:

1. Asignar a cada clase de equivalencia un número único.
2. Hasta que todas las clases de equivalencia hayan sido cubiertas por los casos de prueba, se tratará de escribir un caso que cubra tantas clases válidas no incorporadas como sea posible.
3. Hasta que todas las clases de equivalencia no válidas hayan sido cubiertas por casos de prueba, escribir un caso para cubrir una única clase no válida no cubierta.

La razón de cubrir con casos individuales las clases no válidas es que ciertos controles de entrada pueden enmascarar o invalidar otros controles similares. Por ejemplo, si tenemos dos clases válidas: “introducir cantidad entre 1 y 99” y “seguir con letra entre A y Z”, el caso *105 I* (dos errores) puede dar como resultado *105 fuera de rango de cantidad*, y no examinar el resto de la entrada no comprobando así la respuesta del sistema ante una posible entrada no válida.

4.2.2.2 Análisis de Valores Límite

La experiencia muestra que los casos de prueba que exploran las condiciones límite producen mejor resultado que aquellos que no lo hacen. Las condiciones límite son aquellas que se hayan en los márgenes de la clase de equivalencia, tanto de entrada como de salida. Por ello, se ha desarrollado el *análisis de valores límite* como técnica de prueba. Esta técnica nos lleva a elegir los casos de prueba que ejerciten los valores límite.

Por lo tanto, el análisis de valores límite complementa la técnica de partición de equivalencia de manera que:

- En lugar de seleccionar cualquier caso de prueba de las clases válidas e inválidas, se eligen los casos de prueba en los extremos.
- En lugar de centrarse sólo en el dominio de entrada, los casos de prueba se diseñan también considerando el dominio de salida.

Las pautas para desarrollar casos de prueba con esta técnica son:

- Si una condición de entrada especifica un rango de valores, se diseñarán casos de prueba para los dos límites del rango, y otros dos casos para situaciones justo por debajo y por encima de los extremos.
- Si una condición de entrada especifica un número de valores, se diseñan dos casos de prueba para los valores mínimo y máximo, además de otros dos casos de prueba para valores justo por encima del máximo y justo por debajo del mínimo.
- Aplicar las reglas anteriores a los datos de salida.
- Si la entrada o salida de un programa es un conjunto ordenado, habrá que prestar atención a los elementos primero y último del conjunto.

El Anexo M. presenta un ejemplo de prueba de caja negra con Particiones de Equivalencia y Análisis de Valores Límite para que los alumnos practiquen con la técnica. En el Anexo O se muestra un ejercicio propuesto para que los alumnos ejerciten. El código correspondiente ha sido ya utilizado para la evaluación con técnicas estáticas.

4.3 Estrategia de Pruebas

La estrategia que se ha de seguir a la hora de evaluar dinámicamente un sistema software debe permitir comenzar por los componente más simples y más pequeños e ir avanzando progresivamente hasta probar todo el software en su conjunto. Más concretamente, los pasos a seguir son:

1. Pruebas Unitarias. Comienzan con la prueba de cada módulo.
2. Pruebas de Integración. A partir del esquema del diseño, los módulos probados se vuelven a probar combinados para probar sus interfaces.
3. Prueba del Sistema. El software ensamblado totalmente con cualquier componente hardware que requiere se prueba para comprobar que se cumplen los requisitos funcionales.
4. Pruebas de Aceptación. El cliente comprueba que el software funciona según sus expectativas.

4.3.1 Pruebas Unitarias

La prueba de unidad es la primera fase de las pruebas dinámicas y se realizan sobre cada módulo del software de manera independiente. El objetivo es comprobar que el módulo, entendido como una unidad funcional de un programa independiente, está correctamente codificado. En estas pruebas cada módulo será probado por separado y lo hará, generalmente, la persona que lo creo. En general, un módulo se entiende como un componente software que cumple las siguiente características:

- Debe ser un bloque básico de construcción de programas.
- Debe implementar una función independiente simple.
- Podrá ser probado al cien por cien por separado.
- No deberá tener más de 500 líneas de código.

Tanto pruebas de caja blanca como de caja negra han de aplicar para probar de la manera más completa posible un módulo. Nótese que las pruebas de caja negra (los casos de prueba) se pueden especificar antes de que módulo sea programado, no así las pruebas de caja blanca.

4.3.2 Pruebas de Integración

Aún cuando los módulos de un programa funcionen bien por separado es necesario probarlos conjuntamente: un módulo puede tener un efecto adverso o inadvertido sobre otro módulo; las subfunciones, cuando se combinan, pueden no producir la función principal deseada; la imprecisión aceptada individualmente puede crecer hasta niveles inaceptables al combinar los módulos; los datos pueden perderse o malinterpretarse entre interfaces, etc.

Por lo tanto, es necesario probar el software ensamblando todos los módulos probados previamente. Ésta es el objetivo de la pruebas de integración.

A menudo hay una tendencia a intentar una integración *no incremental*; es decir, a combinar todos los módulos y probar todo el programa en su conjunto. El resultado puede ser un poco caótico con

un gran conjunto de fallos y la consiguiente dificultad para identificar el módulo (o módulos) que los provocó.

En contra, se puede aplicar la integración *incremental* en la que el programa se prueba en pequeñas porciones en las que los fallos son más fáciles de detectar. Existen dos tipos de integración incremental, la denominada *ascendente* y *descendente*. Veamos los pasos a seguir para cada caso:

Integración incremental ascendente:

1. Se combinan los módulos de bajo nivel en grupos que realicen una subfunción específica
2. Se escribe un *controlador* (un programa de control de la prueba) para coordinar la entrada y salida de los casos de prueba.
3. Se prueba el grupo
4. Se eliminan los controladores y se combinan los grupos moviéndose hacia arriba por la estructura del programa.

La Figura 8 muestra este proceso. Concretamente, se forman los grupos 1, 2 y 3 de módulos relacionados, y cada uno de estos grupos se prueba con el controlador C1, C2 y C3 respectivamente. Seguidamente, los grupos 1 y 2 son subordinados de Ma, luego se eliminan los controladores correspondientes y se prueban los grupos directamente con Ma. Análogamente se procede con el grupo 3 eliminando el controlador C3 y probando el grupo directamente con Mb. Tanto Ma y Mb se integran finalmente con el módulo Mc y así sucesivamente.

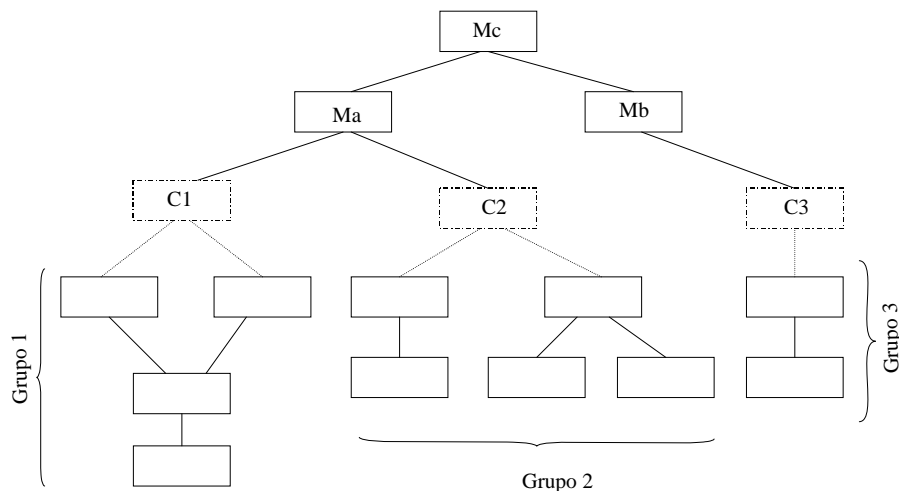


Figura 8. Integración ascendente

Integración incremental descendente:

1. Se usa el módulo de control principal como controlador de la prueba, creando *resguardos* (módulos que simulan el funcionamiento de los módulos que utiliza el que está probando) para todos los módulos directamente subordinados al módulo de control principal.
2. Dependiendo del enfoque e integración elegido (es decir, primero-en-profundidad, o primero-en-anchura) se van sustituyendo uno a uno los resguardos subordinados por los módulos reales.
3. Se llevan a cabo pruebas cada vez que se integra un nuevo módulo.

4. Tras terminar cada conjunto de pruebas, se reemplaza otro resguardo con el módulo real.

La Figura 9 muestra un ejemplo de integración descendiente. Supongamos que se selecciona una integración descendiente por profundidad, y que por ejemplo se prueba M1, M2 y M4. Sería entonces necesario preparar resguardos para M5 y M6, y para M7 y M3. Estos resguardos se ha representado en la figura como R5, R6, R7 y R4 respectivamente. Una vez realizada esta primera prueba se sustituiría R5 por M5, seguidamente R6 por M6, y así sucesivamente hasta probar todos los módulos.

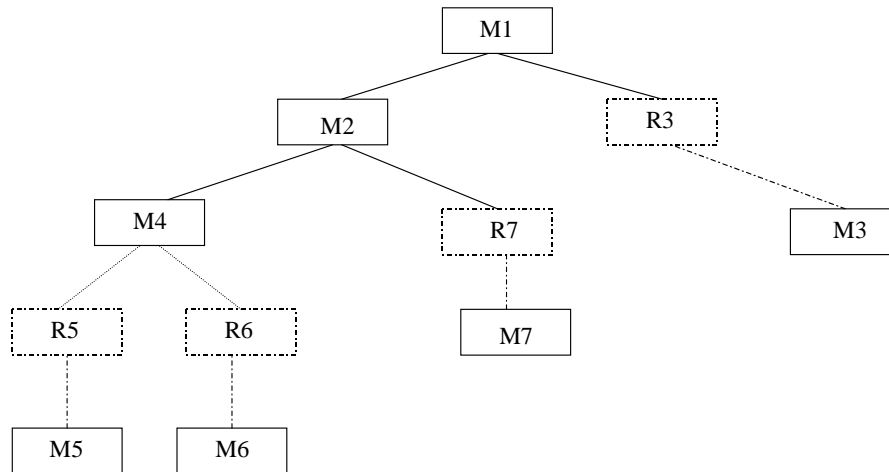


Figura 9. Integración descendiente

Para la generación de casos de prueba de integración, ya sea descendente o ascendente se utilizan técnicas de caja negra.

4.3.3 Pruebas del Sistema

Este tipo de pruebas tiene como propósito ejercitar profundamente el sistema para verificar que se han integrado adecuadamente todos los elementos del sistema (hardware, otro software, etc.) y que realizan las funciones adecuadas. Concretamente se debe comprobar que:

- Se cumplen los requisitos funcionales establecidos.
- El funcionamiento y rendimiento de las interfaces hardware, software y de usuario.
- La adecuación de la documentación de usuario.
- Rendimiento y respuesta en condiciones límite y de sobrecarga.

Para la generación de casos de prueba de sistema se utilizan técnicas de caja negra.

Este tipo de pruebas se suelen hacer inicialmente en el entorno del desarrollador, denominadas *Pruebas Alfa*, y seguidamente en el entorno del cliente denominadas *Pruebas Beta*.

4.3.4 Pruebas de Aceptación

A la hora de realizar estas pruebas, el producto está listo para implantarse en el entorno del cliente. El usuario debe ser el que realice las pruebas, ayudado por personas del equipo de pruebas, siendo deseable, que sea el mismo usuario quien aporte los casos de prueba.

Estas pruebas se caracterizan por:

- Participación activa del usuario, que debe ejecutar los casos de prueba ayudado por miembros del equipo de pruebas.
- Están enfocadas a probar los requisitos de usuario, o mejor dicho a demostrar que no se cumplen los requisitos, los criterios de aceptación o el contrato. Si no se consigue demostrar esto el cliente deberá aceptar el producto
- Corresponden a la fase final del proceso de desarrollo de software.

Es muy recomendable que las pruebas de aceptación se realicen en el entorno en que se va a explotar el sistema (incluido el personal que lo maneje). En caso de un producto de interés general, se realizan pruebas con varios usuarios que reportarán sus valoraciones sobre el producto.

Para la generación de casos de prueba de aceptación se utilizan técnicas de caja negra.

4.3.5 Pruebas de Regresión

La regresión consiste en la repetición selectiva de pruebas para detectar fallos introducidos durante la modificación de un sistema o componente de un sistema. Se efectuarán para comprobar que los cambios no han originado efectos adversos no intencionados o que se siguen cumpliendo los requisitos especificados.

En las pruebas de regresión hay que:

- Probar íntegramente los módulos que se han cambiado.
- Decidir las pruebas a efectuar para los módulos que no han cambiado y que han sido afectados por los cambios producidos.

Este tipo de pruebas ha de realizarse, tanto durante el desarrollo cuando se produzcan cambios en el software, como durante el mantenimiento.

5. PRUEBAS ORIENTADAS A OBJETOS

En las secciones anteriores se ha presentado el proceso de pruebas orientado al concepto general de módulo. Sin embargo, en el caso de la orientación a objetos (OO) es el concepto de clase y objeto el que se utiliza. Veamos a continuación, algunas particularidades de las pruebas para el caso de la OO.

5.1 Prueba de Unidad

Al tratar software OO cambia el concepto de unidad. El encapsulamiento dirige la definición de clases y objetos. Esto significa que cada clase e instancia de clase (objeto) empaqueta los atributos (datos) y las operaciones (también conocidas como métodos o servicios) que manipulan estos datos. Por lo tanto, en vez de módulos individuales, la menor unidad a probar es la clase u objeto encapsulado. Una clase puede contener un cierto número de operaciones, y una operación particular puede existir como parte de un número de clases diferentes. Por tanto, el significado de prueba de unidad cambia ampliamente frente al concepto general visto antes.

De esta manera, la *prueba de clases* para el software OO es el equivalente a la prueba de unidad para software convencional. A diferencia de la prueba de unidad del software convencional, la cual tiende a centrarse en el detalle algorítmico de un módulo y los datos que fluyen a lo largo de la interfaz de éste, la prueba de clases para software OO está dirigida por las operaciones encapsuladas en la clase y el estado del comportamiento de la clase. Así, la prueba de una clase debe haber probado mediante las correspondientes técnicas de caja blanca y caja negra el funcionamiento de cada uno de los métodos de dicha clase. Además, se deben haber generado casos de prueba para probar valores representativos de los atributos de dicha clase (esto puede realizarse aplicando la técnica de clases de equivalencia y análisis de valores límite).

5.2 Prueba de Integración

Debido a que el software OO no tiene una estructura de control jerárquica, las estrategias convencionales de integración ascendente y descendente poseen un significado poco relevante en este contexto.

Generalmente se pueden encontrar dos estrategias diferentes de pruebas de integración en sistemas OO. La primera, *prueba basada en hilos (o threads)*, integra el conjunto de clases necesario para responder a una entrada o evento del sistema. Cada hilo se integra y prueba individualmente. El segundo enfoque para la integración, *prueba basada en el uso*. Esta prueba comienza la construcción del sistema integrando y probando aquellas clases (llamadas clases independientes) que usan muy pocas de las clases. Después de probar las clases independientes, se comprueba la próxima capa de clases, llamadas clases dependientes, que usan las clases independientes. Esta secuencia de capas de pruebas de clases dependientes continúa hasta construir el sistema por completo.

Nótese cómo la prueba basada en hilos proporciona una estrategia más ordenada para realizar la prueba que la prueba basada en el uso. Esta prueba basada en hilos, suele aplicarse utilizando los diagramas de secuencia de objetos que diseñan cada evento de entrada al sistema.

Concretamente, se pueden realizar los siguientes pasos para generar casos de prueba a partir de un diagrama de secuencias:

1. Definir el conjunto de secuencias de mensajes a partir del diagrama de secuencia. Cada secuencia ha de comenzar con un mensaje m sin predecesor (habitualmente, un mensaje enviado al sistema por un actor), y estará formada por el conjunto de mensajes cuya ejecución dispara m .
2. Analizar sub-secuencias de mensajes a partir de posibles caminos condicionales en los diagramas de secuencia.
3. Identificar los casos de prueba que se han de introducir al sistema para que se ejecuten las secuencias de mensajes anteriores, en función de los métodos y las clases afectadas por la secuencia. Tanto valores válidos como inválidos deberían considerarse.

Nótese cómo el conjunto de casos de prueba puede aumentar exponencialmente si se trabaja sobre un sistema OO con un número elevado de interacciones. Por lo tanto, es necesario tener en cuenta este factor a la hora de realizar el diseño.

5.3 Prueba de Sistema

En el nivel de prueba del sistema, los detalles de las conexiones entre clases no afectan. El software debe integrarse con los componentes hardware correspondientes y se ha de comprobar el funcionamiento del sistema completo acorde a los requisitos. Como en el caso del software convencional, la validación del software OO se centra en las acciones visibles del usuario y las salidas del sistema reconocibles por éste. Para asistir en la determinación de casos de prueba de sistema, el ejecutor de la prueba debe basarse en los casos de uso que forman parte del modelo de análisis. El caso de uso brinda un escenario que posee una alta probabilidad con errores encubiertos en los requisitos de interacción del cliente. Los métodos convencionales de prueba de caja negra, pueden usarse para dirigir estas pruebas.

5.4 Prueba de Aceptación

La prueba de aceptación en un sistema OO es semejante a la prueba de aceptación en un software tradicional. El motivo es que el objetivo de este tipo de prueba es comprobar si el cliente está satisfecho con el producto desarrollado y si este producto cumple con sus expectativas, en términos de los errores que genera y de la funcionalidad que suministra. Al igual que las pruebas convencionales serán los clientes quienes realicen estas pruebas y suministren los casos de prueba correspondientes.

6. HERRAMIENTAS DE PRUEBA

A continuación se muestran algunas herramientas que permiten automatizar en cierta medida el proceso de prueba.

6.1 Herramienta para el Análisis Estático de Código Fuente

Jtest



Las características de esta herramienta son las siguientes:

- Jtest comprueba automáticamente la construcción del código fuente ("pruebas de caja blanca"), la funcionalidad del código ("pruebas de caja negra"), y mantiene la integridad del código (pruebas de regresión).
- Se aplica sobre clases Herra Java y JSP.

6.2 Herramientas para Pruebas de Carga y Stress



OpenLoad Tester

Las características de esta herramienta son las siguientes:

- OpenLoad Tester es una herramienta de optimización de rendimiento basada en navegador para pruebas de carga y stress de sitios web dinámicos.
- Permite elaborar escenarios de ejecución y ejecutarlos de forma repetida, simulando la carga de un entorno de producción real de nuestra aplicación como si múltiples usuarios estuvieran usándola



Benchmark Factory

Las características de esta herramienta son las siguientes:

- Benchmark Factory es una herramienta de prueba de carga y capacity planning, capaz de simular el acceso de miles de usuarios a sus servidores de bases de datos, archivos, internet y correo, localizando los posibles cuellos de botella y aislar los problemas relacionados con sobrecargas del sistema

6.3 Herramienta para la automatización de las Pruebas Funcionales



DataFactory

Las características de esta herramienta son las siguientes:

- DataFactory, ayuda a la creación automática de juegos de ensayo o casos de prueba basados en la funcionalidad de las aplicaciones (casos de uso), que facilitan la labor de tener que crearlos manualmente y típicamente, se utilizan junto a las herramientas de pruebas de carga.

6.4 Herramientas de Diagnóstico



PerformaSure

Las características de esta herramienta son las siguientes:

- PerformaSure, es una herramienta de diagnóstico de rendimiento para el análisis en entornos distribuidos J2EE, que permite el seguimiento de los problemas de rendimiento detectados en tiempo real, desde la transacción del usuario final en fase de producción, hasta la línea de código fuente que genera el problema.



Spotlight

Las características de esta herramienta son las siguientes:

- Spotlight, en sus versiones para Servidores de Aplicaciones, Servidores Web, Bases de datos, Sistemas Operativos, etc. es la herramienta visual para la detección en tiempo real, de los cuellos de botella en estos componentes. Una vez que identifica la causa de estos problemas, proporciona la información y consejos necesarios, para su resolución.

6.5 Herramienta de Resolución y Afinado



JProbe

- Esta herramienta, permite detectar los 'puntos calientes' de los componentes de una aplicación JAVA, tales como el uso de la memoria, el uso del CPU, los hilos de ejecución,... y a partir de ellos, bajar al nivel del código fuente que los provoca ofreciendo una serie de consejos o buenas prácticas de codificación para la resolución del problema.

Anexo A. DOCUMENTO DE REQUISITOS PARA EL SISTEMA DE VIDEO ABC

Este Anexo consta de la siguiente estructura en secciones:

- A.1 INTRODUCCIÓN
 - A.1.1 PROPÓSITO
 - A.1.2 ÁMBITO DEL SISTEMA
 - A.1.3 DEFINICIONES, ACRÓNIMOS, ABREVIATURAS
 - A.1.4 VISIÓN GENERAL DEL DOCUMENTO
- A.2 DESCRIPCIÓN GENERAL
 - A.2.1 VISIÓN GENERAL
 - A.2.2 PERSPECTIVA DEL PRODUCTO
 - A.2.3 FUNCIONES DEL PRODUCTO
 - A.2.4 CARACTERÍSTICAS DE LOS USUARIOS
 - A.2.5 RESTRICCIONES Y SUPOSICIONES
- A.3 REQUISITOS ESPECÍFICOS
 - A.3.1 REQUISITOS FUNCIONALES
 - A.3.1.1 Requisitos Generales
 - A.3.1.2 Requisitos para alquilar una cinta
 - A.3.1.3 Requisitos para la devolución de la cintas de vídeo
 - A.3.1.4 Requisitos para dar de alta a un cliente
 - A.3.1.5 Requisitos para dar de alta una cinta de vídeo
 - A.3.1.6 Requisitos de función para la dirección
 - A.3.2 REQUISITOS DE LAS INTERFACES EXTERNAS
 - A.3.2.1 Interfaces de usuario
 - A.3.2.2 Interfaces de hardware
 - A.3.3 REQUISITOS DE RENDIMIENTO
 - A.3.4 OTROS REQUISITOS

A.1. INTRODUCCIÓN

A.1.1. Propósito

Este documento presenta la especificación requisitos de un sistema de gestión de vídeos para ABC vídeo. Estos requisitos representan el comportamiento del sistema, y se ha obtenido mediante entrevistas con los directivos de ABC vídeo y la documentación proporcionada por éstos. La finalidad de este documento es, por tanto, triple:

1. Demostrar a la dirección de ABC Vídeo que el ingeniero de requisitos ha entendido perfectamente los requisitos del sistema.
2. Para obtener comentarios de la dirección de ABC Vídeo con respecto al conocimiento actual del sistema, y en particular, para servir como base de acuerdo para el problema que ha de ser resuelto.
3. Y finalmente, para servir de guía durante el diseño del sistema.

A.1.2. Ámbito del Sistema

Este documento se centra principalmente en los requisitos para un sistema de gestión de vídeos. Por tanto, los requisitos para otro tipo de operaciones de ABC Vídeo están fuera del alcance de este documento. Sin embargo, este documento también especifica los requisitos para cada entidad externa al sistema, las cuales se comunicarán mediante una interfaz con el sistema.

A.1.3. Definiciones, acrónimos, abreviaturas

- Cliente.
Una persona que tiene una cuenta y alquilará cintas de vídeo a ABC Vídeo.
- Cuenta.
Registro de información de un cliente, que permite transacciones, tales como alquileres.
- Productos.
Cintas de vídeo que pueden ser alquiladas. En el futuro este concepto podrá albergar juegos, CDs, etc.
- Recibo.
Recibo de cada transacción.
- PC.
Hace referencia al ordenador que procesará todas las transacciones.
- Tarjeta ABC.
Esta tarjeta la obtiene cada cliente después de ser registrado en el sistema, es decir, cuando se abre una cuenta. En esta tarjeta hay un código de barras con el número de cuenta que puede ser leído por un lector de códigos de barras.

A.1.4. Visión general del documento

El resto del documento consta de en dos secciones. La Sección 2 realiza una descripción informal del sistema basándose en la información obtenida de la directiva de ABC Vídeo. La Sección 3 describe los requisitos funcionales y los requisitos de rendimiento.

A.2. Descripción general

A.2.1. Visión general

Los clientes eligen al menos un vídeo para alquilar. El número máximo de cintas que un cliente puede tener alquiladas es 20. El número de cuenta del cliente se introduce para obtener información del cliente y realizar un pedido. Cada cliente recibe una tarjeta identificativa de ABC. Esta tarjeta identificativa tiene un código de barras que se lee con un lector de códigos de barras. El código de barras de cada cinta que se va alquilar, se

introduce y aparecerá por pantalla la información de inventario de cada vídeo. El fichero del inventario de videos se actualiza. Cuando se introducen todos los códigos de las cintas, el sistema calcula el total a pagar. Se recoge el dinero y se introduce el importe en el sistema. El sistema calcula el cambio y se muestra por pantalla. La transacción de alquiler se crea, imprime y almacena. El cliente firma el recibo de alquiler, recoge las cinta(s) y se marcha.

Para devolver una cinta, se proporciona el código de barras dentro del sistema. La transacción se muestra por pantalla y se anota la fecha de devolución la cinta. Si existen recargos por retraso éstos pueden ser pagados en este momento; o el dependiente puede seleccionar una opción la cual actualizará el alquiler con la fecha de devolución y que calculará el recargo. Aparecen en pantalla todos las películas que el cliente posee en alquiler, así como la cantidad a abonar por cada cinta y la cantidad total. Cualquier recargo deber ser abonado antes de poder alquilar nuevas cintas.

A.2.2. Perspectiva del producto

El sistema funcionará sobre un PentiumIII que actualmente posee ABC Vídeo. El sistema se comunicará con lectores de códigos de barras para simplificar los procesos de alquiler y devolución, y usará impresoras para emitir los recibos para los clientes y para ABC Vídeo. Toda la información concerniente a los clientes, dependientes, videos, tarifas, y históricas de alquileres será almacenada en ese ordenador.

A.2.3. Funciones del producto

El sistema automatizará las operaciones diarias de ABC Vídeo. El sistema permitirá realizar alquileres de forma rápida y conveniente, así como el procesamiento de los pagos, llevando un control automático del estado de cada objeto registrado en el inventario de ABC Vídeo. Las mejoras futuras incluirán:

- Se podrá responder fácilmente a las preguntas de los clientes acerca de las disponibilidades de películas.
- Se avisará de forma automática a los clientes que vuelvan a alquilar un vídeo.
- Se realizaran automáticamente estadísticas diarias acerca del flujo de caja total y el total de cintas alquiladas.

Además, para facilitar las operaciones usuales de ABC Vídeo, el sistema gestionará una variedad de información útil para la dirección. El sistema almacenará información sobre cada cliente de ABC Vídeo además de históricos de transacciones de alquileres.

A.2.4. Características de los usuarios

El sistema será usado por la directiva de ABC Vídeo, los dependientes e indirectamente por los clientes. Desde el punto de vista del sistema, los dependientes y la dirección son idénticos. Algunas operaciones del sistema solo son accesibles para la dirección (tales como imprimir los informes diarios) y están protegidas por un clave.

Los dependientes no necesitan tener conocimientos informáticos, pero necesitarán estar familiarizados con los procedimientos del software para alquileres y devoluciones. La dirección deberá familiarizarse con las capacidades del sistema, incluidas la programación de tarifas, realización de informes y mantenimiento de información sobre clientes e información sobre videos. Los clientes no necesitan ninguna experiencia con el sistema, pero deberán llevar su tarjeta de ABC.

A.2.5. Restricciones y suposiciones

Suposiciones iniciales.

- ABC Vídeo actualizará el hardware cuando sea necesario para el sistema, incluyendo la compra de escáneres, cajas registradoras, una unidad de cinta, y la actualización del disco duro si es necesario.
- El contenido inicial del inventario de videos esta fuera del alcance de este documento.

Otras restricciones y suposiciones

- Todos los clientes y los productos deben estar registrados en el sistema.
- Cada producto debe tener una tarifa asociada.
- Los clientes deben disponer de una tarjeta de crédito válida o realizar un deposito en efectivo o por cheque para poder ser socios.
- El cliente debe dar un número de teléfono y una dirección.
- El cliente debe pagar cualquier factura completamente, el pago parcial no esta permitido.
- Los productos tienen asociados un único periodo de alquiler. Por ejemplo, no se tiene la opción de alquilar a la vez el mismo vídeo para uno o dos días.
- Las devoluciones deben realizarse antes de que se elabore el informe diario para que se puedan incluirse en éste.
- El ordenador mantiene de forma correcta la fecha.
- Los medios de pagos son efectivo, cheque y tarjeta de crédito.

Si el procesamiento falla, se informará que ha habido un error.

A.3. Requisitos Específicos

A.3.1. Requisitos funcionales

Esta es una lista de los requisitos funcionales que el sistema debe satisfacer. Se presentan como sigue:

Descripción: una descripción del requisito.

Entrada: una descripción de las entradas que el sistema necesita.

Procesamiento: una descripción de lo que el sistema debe hacer con las entradas que recibe.

Salida: una descripción de las respuestas/nuevo estado del sistema.

La entrada, procesamiento y salida son especificados cuando son necesarias.

A.3.1.1. Requisitos Generales

Requisito funcional 1

- *Descripción.*
En el estado inicial del sistema se muestra por pantalla el menú principal. Desde el menú principal, el dependiente puede elegir una de las siguientes opciones:

1. Alquilar cintas.
2. Devolver cintas.

3. Dar de alta a un nuevo cliente.
4. Dar de alta una nueva cinta de vídeo
5. Modificar la información de un cliente.
6. Modificar la información de una cinta de vídeo.
7. Borrar un cinta de vídeo.
8. Borrar un cliente
9. Salir.

- *Entrada.*
El empleado elige una opción.
- *Procesamiento.*
Procesar la opción.
- *Salida.*
Mostrar los menús para la opción elegida.

Requisito funcional 2

- *Descripción.*
El sistema mantiene un inventario de cintas de videos almacenando información descriptiva y el estado actual del vídeo.

Requisito funcional 3

- *Descripción.*
El sistema mantiene un registro de transacciones para cada cliente proporcionando información sobre él y las cintas que tiene alquiladas.

A.3.1.2. Requisitos para alquilar una cinta

Requisito funcional 4

- *Descripción.*
El cliente lleva la tarjeta ABC consigo. El número de cuenta del cliente se lee con el lector de código de barras para que el dependiente recupere el registro de transacciones.
- *Entrada.*
Se introduce el número de cuenta mediante el código de barras de la tarjeta.
- *Procesamiento.*
Búsqueda en el registro de transacciones.
- *Salida.*
Mostrar el registro de transacciones.

Requisito funcional 5

- *Descripción.*
El cliente no tiene la tarjeta consigo. El número de cuenta del cliente lo introduce el dependiente a través del teclado para obtener el registro de transacciones.

- *Entrada.*
El número de cuenta se introduce por el teclado.
- *Procesamiento.*
Búsqueda del registro de transacciones.
- *Salida.*
Mostrar el registro de transacciones.

Requisito funcional 6

- *Descripción.*
Se introduce el código de barras de cada cinta que el cliente va a alquilar.
- *Entrada.*
Se introduce mediante el lector el código de barras de cada cinta que va a ser alquilada.
- *Procesamiento.*
Obtener la información sobre el vídeo del inventario.
- *Salida.*
Mostrar el nombre de la cinta de vídeo y el precio de alquiler.

Requisito funcional 7

- *Descripción.*
El número máximo de cintas que pueden alquilarse en una transacción es 20.
- *Entrada.*
Se introduce el identificador de las cintas mediante el lector el código de barras de cada cinta que va a ser alquilada.
- *Procesamiento.*
Si el código introducido corresponde a la cinta número 21, el alquiler se rechaza.
- *Salida.*
Se muestra un mensaje de error.

Requisito funcional 8

- *Descripción.*
Cuando se han introducido todas las cintas el sistema calcula el total.
- *Entrada.*
La tecla Intro se pulsa después de haber introducido la última cinta.
- *Procesamiento.*
Calcular el total a pagar. El total es la suma de los recargos, otras tarifas y los precios de los videos a alquilar.
- *Salida.*
El total a pagar.

Requisito funcional 9

- *Descripción.*
El dependiente recoge el dinero del cliente e introduce la cantidad recibida en el sistema.
- *Entrada.*
Cantidad recibida por el dependiente, introducida por teclado.
- *Procesamiento.*
Calcular el cambio.
- *Salida.*
Mostrar el cambio total.

Requisito funcional 10

- *Descripción.*
Cuando el dependiente presiona la tecla de la opción “orden completa” (definida por el sistema) el alquiler se completa y el fichero de inventario de videos es actualizado.
- *Entrada.*
El dependiente pulsa la tecla de opción “orden completa”.
- *Procesamiento.*
Actualizar el fichero de inventario de videos. Cerrar la transacción de alquiler.
- *Salida.*
El fichero de inventario se actualiza. El fichero de transacciones de alquiler se actualiza.

Requisito funcional 11

- *Descripción.*
Al finalizar el alquiler, la transacción se almacena e imprime.
- *Entrada.*
Cerrar el alquiler actual.
- *Procesamiento.*
Almacenar el alquiler e imprimir el recibo que el cliente tiene que firmar. Volver al estado inicial. Los recibos serán mantenidos en un fichero se que mantendrá durante un mes después de que las cintas se hayan devuelto.
- *Salida.*
Imprimir el recibo. Mostrar el menú inicial.

A.3.1.3. Requisitos para la devolución de las cintas de vídeo

Requisito funcional 12

- *Descripción.*
El código de barras del vídeo se introduce en el sistema.
- *Entrada.*
El código de barras del vídeo.
- *Procesamiento.*

Obtener el registro de transacción de alquiler.

- *Salida.*
Mostrar el registro de transacción de alquiler.

Requisito funcional 13

- *Descripción.*
Cuando se recupera el registro de transacciones, se anota en el registro del vídeo la fecha de devolución.
- *Entrada.*
El código de barras del vídeo a alquilar.
- *Procesamiento.*
Se obtienen el registro de transacciones y el del inventario del video. Se anota el día de devolución en el registro.
- *Salida.*
Actualizar el registro de inventario y las transacciones de alquiler.

Requisito funcional 14

- *Descripción.*
Si existe un recargo por retraso, se puede pagar en ese momento; o el dependiente puede pulsar la tecla de “orden completa” con lo cual se actualiza el alquiler con la nueva fecha de devolución y se calculan los recargos.
- *Entrada.*
Abono o cálculo del recargo
- *Procesamiento.*
Actualización del registro de transacciones. Ir al estado inicial.
- *Salida.*
Actualizar el fichero de transacciones.

A.3.1.4. Requisitos para dar de alta a un cliente

Requisito funcional 15

- *Descripción.*
Un nuevo cliente quiere alquilar cintas de video. El dependiente introduce toda la información necesaria, imprime el código de barras para la tarjeta ABC y lo pega una tarjeta nueva. La tarjeta se entrega al cliente.
- *Entrada.*
El dependiente introduce la siguiente información: Nombre, dirección e información sobre la tarjeta de crédito del cliente.
- *Procesamiento.*
Crea un nuevo registro de transacciones de alquiler para el cliente. El sistema asigna un número de cuenta al cliente e imprime el código de barras. Ir al estado inicial.
- *Salida.*

Imprime el código de barras. El cliente puede alquilar cintas de vídeo.

A.3.1.5. Requisitos para dar de alta una cinta de vídeo

Requisito funcional 16

- *Descripción.*
Antes que se pueda alquilar una cinta nueva se debe introducir toda la información necesaria. El código de barras es impreso y el dependiente tiene que pegarlo en el vídeo.
- *Entrada.*
Una cinta de vídeo se caracteriza por los siguientes atributos: nombre del vídeo, precio de alquiler e identificador de la cinta.
- *Procesamiento.*
Crea un nuevo registro en el inventario de cintas de vídeo para la cinta.
- *Salida.*
Se genera un registro en el inventario de videos. La cinta puede ser alquilada. Se imprime el código de barras para la cinta.

Requisito funcional 17

- *Descripción.*
El dependiente puede modificar la información de un vídeo o un cliente.
- *Entrada.*
El dependiente introduce nueva información bien sobre un vídeo o sobre cliente.
- *Procesamiento.*
Actualizar los datos del fichero de inventario de vídeos.
- *Salida.*
Mostrar el cambio si éste tiene lugar.

A.3.1.6 Requisitos de función para la dirección

Requisito funcional 18

- *Descripción.*
Solo la dirección pueden borrar un cliente o un vídeo.
- *Entrada.*
El directivo introduce el número de cuenta de un vídeo o cliente.
- *Procesamiento.*
Borrar el vídeo o cliente.
- *Salida.*
Mostrar el cambio si este tiene lugar.

Requisito funcional 19

- *Descripción.*
El directivo puede imprimir diariamente informes o algunas estadísticas.
- *Entrada.*
El directivo selecciona que clase de información quieren tener. Pueden elegir cualquier elemento de la siguiente lista:
 - Informes diarios
 - Lista de clientes registrados durante algún periodo de tiempos
 - Lista de clientes morosos
 - Lista de clientes con alquileres retrasados
 - Lista de cintas organizadas por estados
 - Lista de cintas no alquiladas durante un cierto número de días.
 - Número de alquileres (por copia, título, tipo) durante un cierto periodo de tiempo.
 - Número de días alquilados (por mes, año, copia y título)
 - Históricos de alquileres de los clientes.
- *Procesamiento.*
Recoger toda la información necesaria para la petición realizada e imprimirla.
- *Salida.*
El informe impreso.

A.3.2. Requisitos de las interfaces externas

A.3.2.1. Interfaces de usuario

No aplicable.

A.3.2.2. Interfaces de hardware

No aplicable

A.3.3. Requisitos de rendimiento

Requisito de rendimiento 1

El sistema software completo debe ser amigable.

Requisito de rendimiento 2

El sistema tendrá que responder rápidamente. El tiempo típico de respuesta para la lectura del código de barras de un vídeo que va a ser alquilado debe ser inferior a 15 seg. En casi todos los casos el tiempo de respuesta deberá estar por debajo de 5 minutos, con la posible excepción de la recuperación de información desde una cinta de backup de 8mm o de un disco.

A.3.4. Otros requisitos

Requisito 1.

El sistema debe funcionar en cualquier ordenador corriente.

Anexo B. LISTAS DE COMPROBACIÓN

La estructura de este anexo es la siguiente:

- B.1. Lista de comprobación para requisitos
- B.2. Lista de comprobación para diseño arquitectónico
- B.3. Lista de comprobación para diseño de alto nivel

B.1.- Lista de Comprobación para Requisitos

Contenido de los requisitos

- ¿Están especificadas todas las entradas al sistema, incluyendo su origen, precisión, rango de valores y frecuencia?
- ¿Están especificadas todas las salidas del sistema, incluyendo su destino, precisión, rango de valores, frecuencia y formato?
- ¿Están todos los formatos de los informes especificados?
- ¿Están especificados los interfaces con otros sistemas software y hardware externos?
- ¿Están especificados todos los interfaces de comunicación, incluyendo *handshaking*, chequeo de errores y protocolos de comunicación?
- ¿Se ha especificado, para todas aquellas operaciones que sea necesario, el tiempo esperado de respuesta desde el punto de vista del usuario?
- ¿Se han especificado otras consideraciones de temporización, como el tiempo de procesamiento, transferencia de datos y rendimiento del sistema?
- ¿Se han especificado todas las tareas que el usuario desea realizar?
- ¿Especifica cada tarea los datos que se usan en la tarea y los datos resultantes de la tarea?
- ¿Se ha especificado el nivel de seguridad?
- ¿Se ha especificado la fiabilidad incluyendo las consecuencias de los fallos del software, información vital que ha de protegerse de fallos, detección de errores y recuperación de los mismos?
- ¿Se ha especificado el equilibrio aceptable entre atributos de calidad contradictorios, por ejemplo entre robustez y corrección?
- ¿Se ha especificado el límite de memoria?
- ¿Se ha especificado el límite de almacenamiento?
- ¿Se ha incluido la definición de éxito? ¿Y de fallo?
- ¿Se ha especificado la mantenibilidad del sistema, incluyendo la habilidad para responder a cambios en el entorno operativo, interfaces con otro software, precisión, rendimiento y otras capacidades adicionales predichas?

Complejidad de los requisitos

- En aquellos puntos en los que no hay información disponible antes de que comience el desarrollo, ¿se han especificado las áreas de incomplejidad?

- ¿Son los requisitos completos en el sentido de que si un producto satisface todos los requisitos, será aceptable?
- ¿Surgen dudas acerca de alguna parte de los requisitos? ¿Son algunas partes del sistema imposibles de implementar y se han incluido simplemente para satisfacer al cliente?

Calidad de los requisitos

- ¿Están todos los requisitos escritos en el lenguaje del usuario? ¿Piensan eso los usuarios?
- ¿Evitan todos los requisitos conflictos con otros requisitos?
- ¿Evitan los requisitos especificar el diseño?
- ¿Están los requisitos a un nivel bastante consistente? ¿Debería especificarse algún requisito con más detalle? ¿Debería especificarse algún requisito con menos detalles?
- ¿Son los requisitos lo suficientemente claros para poderse enviar a un grupo independiente para su implementación y que lo entiendan?
- ¿Es cada requisito relevante al problema y a su solución? ¿Se puede trazar cada uno al origen en el entorno del problema?
- ¿Es cada requisito testeable? ¿Sería posible para un grupo independiente de pruebas determinar si cada requisito se ha satisfecho?
- ¿Se han especificado todos los cambios posibles a los requisitos, incluyendo la probabilidad de cada cambio?

B.2.- Lista de Comprobación para el Diseño Arquitectónico

- ¿Es la organización del sistema clara, incluyendo una buena visión general de la arquitectura y su justificación?
- ¿Están todos los módulos bien definidos, incluyendo su funcionalidad e interfaces con otros módulos?
- ¿Se cubren todas las funciones que aparecen en los requisitos, ni por demasiados ni por pocos módulos?
- ¿Se han descrito y justificado todas las estructuras de datos más importantes?
- ¿Se han ocultado todas las estructuras de datos con funciones de acceso?
- ¿Se ha especificado la organización y contenidos de la base de datos?
- ¿Se han descrito y justificado todos los algoritmos principales?
- ¿Se han descrito y justificado los objetos principales?
- ¿Se ha modularizado la interfaz de usuario de tal forma que los cambios en ella no afectarán al resto del programa?
- ¿Se ha descrito alguna estrategia para gestionar la entrada del usuario?
- ¿Se han definido los aspectos principales del interfaz de usuario?

- ¿Se han descrito y justificado las estimaciones de uso de la memoria, así como una estrategia para la gestión de la misma?
- ¿Asigna la arquitectura espacio y velocidad para cada módulo?
- ¿Se describe una estrategia para manejar cadenas, y se incluyen estimaciones sobre el almacenamiento de las cadenas de caracteres?
- ¿Se describe y justifica una estrategia para el manejo de entradas y salidas?
- ¿Se incluye una estrategia de manejo de errores coherente?
- ¿Se especifica un nivel de robustez?
- ¿Se han incluido decisiones necesarias acerca de compra o construcción de software?
- ¿Se ha diseñado la arquitectura para acomodar cambios probables?
- ¿Está alguna parte demasiado o poco trabajada?
- ¿Se han enunciado los principales objetivos del sistema?
- ¿Encaja conceptualmente cada parte de la arquitectura para formar un todo?
- ¿Es el diseño a alto nivel independiente de la máquina y del lenguaje que se usará para implementarlo?
- ¿Se han dado motivaciones para todas las decisiones de diseño principales?

B.3.- Lista de comprobación para Diseño de Alto Nivel

- ¿Se han considerado varias opciones de diseño seleccionando la mejor de varias opciones, o simplemente se ha elegido la primera que se ha pensado?
- ¿Es el diseño del sistema actual consistente con el diseño de sistemas relacionados?
- ¿Gestiona adecuadamente el diseño asuntos que fueron identificados y postergados al nivel de la arquitectura?
- ¿Es satisfactoria la forma en la que el programa se ha descompuesto en módulos u objetos?
- ¿Es satisfactoria la forma en que los módulos se han descompuesto en rutinas?
- ¿Se han definido bien todas las fronteras de los subsistemas?
- ¿Se han diseñado los subsistemas para la interacción mínima de unos con otros?
- ¿Tiene sentido el diseño recorriéndolo tanto de arriba abajo como de abajo a arriba?
- ¿Distingue el diseño entre componentes pertenecientes al dominio del problema, componentes de interfaz de usuario, componentes de gestión de tareas y componentes de gestión de datos?
- ¿Es el diseño manejable intelectualmente?
- ¿Tiene el diseño complejidad baja?
- ¿Será el sistema fácil de mantener?
- ¿Reduce el diseño las conexiones entre subsistemas a la mínima cantidad?
- ¿Permite el diseño extensiones futuras al sistema?
- ¿Están diseñados los subsistemas de tal modo que se pueden usar en otros sistemas?
- ¿Tienen las rutinas de bajo nivel un *fan-in* alto?
- ¿Tienen la mayoría de las rutinas un *fan-out* entre bajo y medio?
- ¿Será sencillo portar el diseño a otro entorno?
- ¿Es el diseño balanceado? ¿Son todas sus partes estrictamente necesarias?
- ¿Está estratificado el diseño en niveles?

- ¿Usa el diseño técnicas estándar y evita elementos exóticos, difíciles de entender?

Anexo C. LISTAS DE COMPROBACIÓN PARA CÓDIGO

La estructura de este anexo es la siguiente:

- C.1. Lista de comprobación para estructuras de control
- C.2. Lista de comprobación para estructuras de control inusuales
- C.3. Lista de comprobación para sentencias condicionales
- C.4. Lista de comprobación para bucles
- C.5. Lista de comprobación para el formato del código
- C.6. Lista de comprobación para que el código sea inteligible
- C.7. Lista de comprobación para comentarios

C.1.- Lista de Comprobación para Estructuras de Control

- ¿Se usan las expresiones verdadero y falso en lugar de 1 y 0?
- ¿Se comparan los valores booleanos con falso implícitamente?
- ¿Se han simplificado las expresiones booleanas usando variables booleanas adicionales, funciones booleanas y tablas de decisión?
- ¿Se muestran las expresiones de forma positiva?
- En C, ¿se comparan con 0 explícitamente los números, caracteres y punteros?
- ¿Están balanceados los pares *begin/end*?
- ¿Se usan los pares *begin/end* en todos aquellos sitios en que son necesarios por claridad?
- ¿Son las sentencias nulas obvias?
- ¿Se han simplificado las sentencias anidadas re-probando parte de la condición, convirtiéndolas a *if-then-else* o moviendo código anidado a una rutina propia?
- Si la rutina tiene más de 10 decisiones, ¿hay una buena razón para no rediseñarla?

C.2.- Lista de Comprobación para Estructuras de Control Inusuales

Goto

- ¿Se usan los *gotos* únicamente como último recurso, y únicamente para hacer el código más legible y mantenible?
- Si se usa un *goto* por eficiencia, ¿se ha medido y documentado la ganancia en eficiencia?
- ¿Están limitados los *gotos* a una etiqueta por rutina?
- ¿Van todos los *gotos* hacia delante, no hacia atrás?
- ¿Se usan todas las etiquetas de los *gotos*?

Return

- ¿Hace cada rutina uso del número mínimo posible de *returns*?

- ¿Mejoran los *return* la legibilidad?

Recursividad

- ¿Incluye la rutina recursiva código para parar la recursividad?
- ¿Usa la rutina un contador de seguridad para garantizar que la rutina para?
- ¿Está la recursividad limitada a una rutina?
- ¿Está la profundidad de la recursividad de la rutina dentro de los límites impuestos por el tamaño del programa?
- ¿Es la recursividad la mejor forma de implementar la rutina? ¿Es mejor que una simple iteración?

C.3.- Lista de Comprobación para Sentencias Condicionales

Sentencias if-then

- ¿Está claro el camino nominal a través del código?
- ¿Se comportan las comprobaciones *if-then* correctamente con la igualdad?
- ¿Está la cláusula *else* presente o documentada?
- ¿Es la cláusula *else* correcta?
- ¿Se usan las cláusulas *if* y *else* correctamente, no cambiadas?
- ¿Siguel el caso normal el *if* más que el *else*?

Cadenas if-then-else-if

- ¿Están encapsulados las comprobaciones complicadas en llamadas a funciones booleanas?
- ¿Se comprueban primero los casos más comunes?
- ¿Están cubiertos todos los casos?
- ¿Es la cadena *if-then-else-if* la mejor implementación? ¿Mejor que una sentencia *case*?

Sentencias case

- ¿Están los casos ordenados significativamente?
- ¿Son sencillas las acciones para cada caso—llamando a otras rutinas si es necesario?
- ¿Chequea el caso una variable real, no una *phony* que está hecha únicamente para usar y abusar de la sentencia *case*?
- ¿Es legítimo el uso de la cláusula *default*?
- ¿Se usa la cláusula *default* para detectar y reportar casos no esperados?
- En C, ¿Hay al final de cada *case* un *break*?

C.4.- Lista de Comprobación para Bucles

- ¿Se entra al bucle desde arriba?

- ¿Está la inicialización del código directamente antes del bucle?
- Si el bucle es un bucle de evento, ¿Está construido limpiamente más que usando un *kludge* como *for i:=1 to 9999*?
- Si el bucle es un bucle para C, ¿Está la cabecera del bucle reservada para el código de control del bucle?
- ¿Usa el bucle *begin* y *end* o sus equivalentes para evitar problemas que surgen de modificaciones inadecuadas?
- ¿Tiene el bucle algo dentro de él? ¿Está vacío?
- ¿Lleva a cabo el bucle una, y sólo una, función, como una rutina bien definida?
- ¿Termina el bucle bajo todas las posibles condiciones?
- ¿Es la condición de terminación del bucle obvia?
- Si el bucle es un bucle *for*, ¿Evita el código dentro de él *monkeying* con el índice del bucle?
- ¿Se usa una variable para guardar valores importantes de índice del bucle, en lugar de usar el índice del bucle fuera de él?
- ¿Usa el bucle contadores de seguridad?
- Si el bucle está anidado, ¿Se usan nombres de bucle claros?
- ¿Es el índice del bucle de tipo ordinal o enumerado?
- ¿Tiene el índice del bucle un nombre significativo?
- ¿Es el bucle lo suficientemente corto para verlo de una sola vez?
- ¿Está el bucle anidado a tres niveles o menos?
- Si el bucle es largo, ¿Es especialmente claro?

C.5.- Lista de Comprobación para el Formato del Código

General

- ¿Se ha hecho el formateado principalmente para iluminar la estructura lógica del código?
- ¿Se puede usar el esquema de formateado consistentemente?
- ¿Facilita el esquema de formateado del código su mantenimiento?
- ¿Mejora el esquema de formateado la legibilidad del código?

Estructuras de Control

- ¿Evita el código indentar doblemente los pares *begin/end*?
- ¿Están los bloques secuenciales separados por líneas en blanco?
- ¿Se han formateado las expresiones booleanas complicadas por legibilidad?
- ¿Están los bloques con una única sentencia formateados consistentemente?
- ¿Están las sentencias *case* formateadas de tal forma que sean consistentes con el formateado de otras estructuras de control?
- ¿Se han formateado los *gotos* de tal forma que se hace obvio su uso?

Sentencias Individuales

- ¿Finalizan la línea las sentencias incompletas de tal forma que es obviamente incorrecto?
- ¿Están las líneas de continuación indentadas sensiblemente?
- ¿Están alineados los grupos de sentencias relacionadas?
- ¿Están los grupos de sentencias no relacionadas no alineadas?

C.6.- Lista de Comprobación para que el Código sea Inteligible

Rutinas

- ¿Describe el nombre de cada rutina exactamente lo que hace?
- ¿Lleva a cabo cada rutina una tarea bien definida?
- ¿Es el interfaz de cada rutina obvio y claro?

Nombre de Datos

- ¿Son los nombre de los tipos de datos lo suficientemente descriptivos para ayudar a documentar las declaraciones de los datos? ¿Se usan específicamente para ese propósito?
- ¿Se han nombrado bien las variables?
- ¿Se han usado las variables únicamente para el propósito por el cual se han nombrado?
- ¿Tienen los contadores de bucle nombres más informativos que i, j, y k?
- ¿Se usan tipos enumerados bien nombrados en lugar de *flags* o variables booleanas?
- ¿Se usan constantes declaradas en lugar de números mágicos o cadenas mágicas?
- ¿Distinguen las convenciones de nombrado entre nombres de tipos, tipos enumerados, constantes declaradas, variables locales, variables de módulo y variables globales?

Organización de los datos

- ¿Se usan variables extra por claridad cuando es necesario?
- ¿Están las referencias a variables cercanas la una a la otra?
- ¿Son las estructuras de datos simples de tal forma que minimizan la complejidad?
- ¿Es complicado el acceso a los datos a través de rutinas de acceso abstractas (tipos abstractos de datos)?

Control

- ¿Está claro el camino nominal a través del código?
- ¿Están agrupadas juntas sentencias relacionadas?
- ¿Se han empaquetado en sus propias rutinas grupos de sentencias relativamente independientes?

- ¿Son las estructuras de control simples de tal modo que minimizan la complejidad?
- ¿Se ha minimizado el número de anidamientos?

Diseño

- ¿Es el código directo y evita “virguerías”?
- ¿Están escondidos en la medida de lo posible los detalles de implementación?
- ¿Está escrito el programa en términos del dominio del programa en la medida de lo posible más que en términos de estructuras informáticas o de lenguaje de programación?

C.7.- Lista de Comprobación para Comentarios

General

- ¿Contiene el código fuente la mayor parte de la información sobre el programa?
- ¿Puede alguien coger el código y comenzar a entenderlo inmediatamente?
- ¿Explican los comentarios la intención del código o lo resumen, más que repetirlo?
- ¿Están actualizados los comentarios?
- ¿Son los comentarios claros y correctos?
- ¿Permite el estilo de comentarios que se modifiquen los mismos fácilmente?

Sentencias y párrafos

- ¿Se centran los comentarios en el por qué más que en el cómo?
- ¿Preparan los comentarios la mente del lector para lo que sigue a continuación?
- ¿Sirve para algo cada comentario? ¿Se han eliminado o mejorado comentarios redundantes, extraños o indulgentes?
- ¿Se documentan las sorpresas?
- ¿Se han sustituido las abreviaciones?
- ¿Está clara la distinción entre comentarios mayores y menores?
- ¿Se ha comentado el código que está relacionado con un error o una característica no documentada?

Declaraciones de Datos

- ¿Se han comentado las unidades de declaraciones de datos?
- ¿Se ha comentado el rango de valores de datos numéricos?
- ¿Están comentados los significados codificados?
- ¿Están comentadas las limitaciones sobre los datos de entrada?
- ¿Están documentados los *flags* a nivel de bit?
- ¿Se ha comentado cada variable global en el lugar donde se ha declarado?
- ¿Se ha documentado cada variable global cada vez que se usa, bien con una convención de nombrado o con un comentario?
- ¿Se ha comentado cada sentencia de control?
- ¿Se han comentado los finales de estructuras de control largas o complejas?

Rutinas

- ¿Se ha comentado el propósito de cada rutina?
- ¿Se han comentado otros hechos de cada rutina, cuando es relevante, incluyendo datos de entrada y salida, asunciones de interfaz, limitaciones, correcciones de errores, efectos globales y fuentes de algoritmos?

Ficheros, módulos y programas

- ¿Tiene el programa un documento de 4 a 5 páginas que da una vista general de cómo se ha organizado el programa?
- ¿Se ha descrito el propósito de cada fichero?
- ¿Está en el listado el nombre del autor y el modo de localizarlo?

Anexo D. PROGRAMA PARA EJERCICIO DE CÓDIGO

D.1.- Especificación del programa “count”

Nombre

count – cuenta líneas, palabras y caracteres.

Uso

count Fichero [Fichero...]

Descripción

count cuenta el número de líneas, palabras y caracteres que hay en cada fichero que se le pasa como argumento. Las palabras son secuencias de caracteres que están separadas por uno o más espacios, tabuladores o saltos de línea.

Si alguno de los ficheros que se le pasa como argumento no existe, aparece por la salida de error el mensaje de error correspondiente y se continúa procesando el resto de ficheros. Si no se indica ningún fichero, **count** lee de la entrada estándar.

Se muestra por pantalla los valores computados para cada fichero (junto con el nombre del fichero) así como la suma total de cada valor para todos los ficheros. Si se procesa un único fichero o si se procesan datos provenientes de la entrada estándar, entonces no se imprime ninguna suma. La salida se imprime en el orden siguiente: primero líneas, luego palabras, luego caracteres y finalmente bien el nombre del fichero o la palabra “total” para la suma. Si se ha leído de la entrada estándar, el cuarto valor (nombre) no aparece.

Opciones

Ninguna.

Ejemplo

C:\count fichero1

84 462 3621 fichero1

Descripción breve de las funciones de librería usadas

- FILE *fopen(“Nombre”, “r”)

Abre el fichero indicado para lectura y devuelve un puntero al mismo, en caso de error, devuelve NULL.

- int fclose (fp)

Cierra un fichero que estaba previamente abierto.

- `int printf ("Texto", ...)`
Imprime el texto en la salida estándar.
- `int fprintf (stderr, "Texto" ...)`
Imprime el texto en la salida error estándar.

D.2.- Lista de Comprobación

1. ¿Se han inicializado todas las variables del programa antes de usarlas?
2. ¿Están inicializadas con el valor adecuado?
3. ¿Se han definido todas las constantes?
4. El límite inferior de los arrays, ¿es 0, 1 u otra cosa?
5. El límite superior de los arrays, ¿es igual al tamaño del array o el tamaño menos uno?
6. Si se usan delimitadores de cadenas de caracteres, ¿hay un delimitador específicamente asignado?
7. Para cada sentencia condicional, ¿es correcta la condición?
8. ¿Terminan todos los bucles?
9. ¿Están correctamente puestos los paréntesis y/o llaves en las sentencias?
10. ¿Se han tenido en cuenta todos los posibles casos de las sentencias CASE?
11. ¿Se usan todas las variables definidas?
12. ¿Se asigna algún valor a todas las variables de salida?
13. Si se usa memoria dinámica, ¿se ha reservado espacio adecuadamente antes de usarla?
14. ¿Se libera la memoria dinámica cuando ya no se necesita más?
15. ¿Se han tenido en cuenta todas las posibles condiciones de error?

D.3.- Código fuente para el programa “count”. Fichero “count.c”

```
#include <stdio.h>

main (argc, argv)
    int argc;
    char *argv[];
{
    int      c, i, inword;
    FILE     *fp;
    long     linect, wordct, charct;
    long     tlinect = 1, twordct = 1, tcharct = 1;

    i = 1;
    do {
        if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
            fprintf (stdout, "can't open %s\n", argv[i]);
            exit (1);
        }
        linect = wordct = charct = 0;
        inword = 0;
        while ((c = getc(fp)) != EOF) {
            ++charct;
            if (c == '\n')
                ++linect;
            if (c == ' ' || c == '\t' || c == '\n')
                inword = 0;
            else if (inword == 0) {
                inword = 1;
                ++wordct;
            }
        }
    }
    printf("%7ld %7ld %7ld", linect, wordct, charct);
    if (argc > 1)
        printf(" %s\n", *argv);
    else
        printf("\n");
}
```

```
fclose(fp);
tlinect += linect;
twordct += wordct;
tcharct += charct;
} while (++i <= argc);
if (argc > 1)
    printf("%7ld %7ld %7ld total\n", linect, twordct, tcharct);
exit(0);
```


Anexo E. SOLUCIÓN PARA EL EJERCICIO PRÁCTICO “COUNT”

La estructura de este Anexo es la siguiente:

- E.1. Abstracción del programa “count” cuyo código aparecía en el Anexo D
- E.2. Especificación obtenida a partir del código
- E.3. Especificación original
- E.4. Comparación de ambas especificaciones
- E.5. Faltas del programa “count”

E.1. Abstracción del programa “count” cuyo código aparecía en el Anexo D

A continuación se listan los números de línea y las abstracciones (especificaciones) del código fuente de estas líneas.

Línea(s)	Abstracción
----------	-------------

21-29	Secuencia
-------	-----------

Significado de los componentes:

21	$\text{charct} := \text{charct} + 1$
----	--------------------------------------

22-23	$c = \text{otralínea} \rightarrow \text{linect} := \text{linect} + 1 \mid ()$
-------	---

24-29	$c = \text{espacio} \rightarrow \text{inword} := 0 \mid$
-------	--

24-30	$(\text{inword} = 0 \rightarrow \text{inword}, \text{wordct} := 1, \text{wordct} + 1 \mid ())$
-------	--

. Como hay 4 posibles caminos en esta secuencia, resulta lo siguiente:

$c = \text{otralínea} \rightarrow \text{charct}, \text{linect} := \text{charct} + 1, \text{linect} + 1 \mid$

$c \neq \text{otralínea} \wedge c = \text{espacio} \rightarrow \text{charct}, \text{inword} := \text{charct} + 1, 0 \mid$

$c \neq \text{otralínea} \wedge c \neq \text{espacio} \wedge \text{inword} = 0 \rightarrow$

$\text{charct}, \text{wordct}, \text{inword} := \text{charct} + 1, \text{wordct} + 1, 1 \mid$

$c \neq \text{otralínea} \wedge c \neq \text{espacio} \wedge \text{inword} \neq 0 \rightarrow \text{charct} := \text{charct} + 1.$

14-39	Secuencia
-------	-----------

De nuevo, el significado de los componentes es:

14-17	$\text{argc} > 1 \rightarrow (\text{file-open}(\text{argv}[i]) = \text{fallo} \rightarrow \text{msgerr y parar} \mid \text{fp} := \text{stream}) \mid ()$
-------	---

18-30	Para determinar el significado de este trozo, se incluyen las líneas 18-19 y se miran las tareas de las variables:
-------	--

1. La variable “charct” se incrementa en los cuatro casos; es decir, cuenta cada carácter.
2. La variable “linect” sólo se incrementa si se lee *otralínea*; es decir, cuenta líneas.
3. La variable “inword” es un switch que toma los valores 0 y 1.

Si se encuentra un espacio en blanco, el switch se pone a 0. Si se encuentra otro carácter, se pone a 1 y al mismo tiempo se incrementa “wordct”; es decir, cuenta palabras.

$c \neq \text{EOF} \rightarrow \text{charct}, \text{wordct}, \text{linect} :=$
 $\text{character-count}(\text{stdin}), \text{word-count}(\text{stdin}), \text{line-count}(\text{stdin}).$

31 $\text{stdout} := \text{“linect, wordct, charct”}$
32-35 $\text{argc} > 1 \rightarrow \text{stdout} := *argv$ (*es decir, nombre del programa*) y *otralinea* |
 $\text{stdout} := \text{otralinea}$
36 *cerrar entrada*
37-39 $\text{tlinect}, \text{twordct}, \text{tcharct} := \text{tlinect} + \text{linect}, \text{twordct} + \text{wordct}, \text{tcharct} + \text{charct}$

Como hay 3 posibles caminos en esta secuencia, resulta lo siguiente:

$\text{argc} > 1 \wedge \text{open-file}(\text{argv}[i]) = \text{failure} \rightarrow \text{stdout} := \text{err_msg}$ y parar |

$\text{argc} > 1 \vee \text{open-file}(\text{argv}[i]) = \text{success} \rightarrow$

$\text{tcharct}, \text{tlinect}, \text{twordct}, \text{stdout} := \text{tcharct} + \text{character-count}(\text{stream}),$
 $\text{twordct} + \text{word-count}(\text{stream}), \text{tlinect} + \text{line-count}(\text{stream}),$
 $\text{“line-count}(\text{stream}), \text{word-count}(\text{stream}), \text{character-count}(\text{stream}),$
 pgr-name”

$\text{argc} \leq 1 \rightarrow$

$\text{tcharct}, \text{tlinect}, \text{twordct}, \text{stdout} := \text{tcharct} + \text{character-count}(\text{<nil>}),$

$\text{twordct} + \text{word-count}(\text{<nil>}), \text{tlinect} + \text{character-count}(\text{<nil>}),$

$\text{“line count}(\text{<nil>}), \text{word-count}(\text{<nil>}), \text{character-count}(\text{<nil>})”$

Nota: si argc es ≤ 1 , fp no se inicializa; es decir, el programa lee de una entrada indefinida (etiqueta <nil> de arriba).

3-44

Secuencia

De nuevo, primero el significado de los componentes:

10 $\text{tlinect}, \text{twordct}, \text{tcharct} := 1, 1, 1$
12-40 para todos los índices de los argumentos de la línea de comandos de 1 hasta argc hacer [14-39].

Nota: Siempre se intenta leer un fichero más de los existentes en la línea de comandos.

41-42 $\text{argc} > 1 \rightarrow \text{stdout} := \text{“linect, twordct, tcharct”} | ()$

43 parar.

E.2. Especificación del programa “count” obtenida a partir del código

El comportamiento del programa obtenido a partir del código es:

Todos los argumentos de la línea de comandos se tratan como nombres de ficheros, aunque siempre se intenta leer uno de más que no existe. Se distinguen tres casos:

1. Se proporcionan argumentos, pero no hay ficheros con nombres correspondientes a los argumentos. En este caso, el programa se para con un mensaje de error que sale por la salida estándar.
2. Se proporcionan argumentos, y corresponden a ficheros que existen. Para cada fichero, el número de líneas, palabras y caracteres se cuenta y se imprime la suma junto con el nombre del programa.
3. No se proporcionan argumentos. En este caso, el programa intenta leer de algo que no está inicializado y procede como en (2), a pesar de que no se imprime el nombre del programa.

Si el número total de argumentos es al menos 1, se imprime la suma total más uno de todos los caracteres y palabras en todos los ficheros y el número de líneas del último fichero.

E.3. Especificación original del programa “count

count cuenta el número de líneas, palabras y caracteres que hay en cada fichero que se le pasa como argumento. Las palabras son secuencias de caracteres que están separadas por uno o más espacios, tabuladores o saltos de línea.

Si alguno de los ficheros que se le pasa como argumento no existe, aparece por la salida de error el mensaje de error correspondiente y se continúa procesando el resto de ficheros. Si no se indica ningún fichero, **count** lee de la entrada estándar.

Se muestra por pantalla los valores computados para cada fichero (junto con el nombre del fichero) así como la suma total de cada valor para todos los ficheros. Si se procesa un único fichero o si se procesan datos provenientes de la entrada estándar, entonces no se imprime ninguna suma. La salida se imprime en el orden siguiente: primero líneas, luego palabras, luego caracteres y finalmente bien el nombre del fichero o la palabra “total” para la suma. Si se ha leído de la entrada estándar, el cuarto valor (nombre) no aparece.

E.4. Comparación de ambas especificaciones y detección de defectos.

Se señalan en negrita los puntos donde la especificación obtenida por abstracción diverge de la especificación original:

Todos los argumentos de la línea de comandos se tratan como nombres de ficheros, aunque siempre se intenta leer uno de más que no existe. Se distinguen tres casos:

1. Se proporcionan argumentos, pero no hay ficheros con nombres correspondientes a los argumentos. En este caso, ***el programa se para*** con un mensaje de error que sale por la ***salida estándar***.
2. Se proporcionan argumentos, y corresponden a ficheros que existen. Para cada fichero, el número de líneas, palabras y caracteres se cuenta y se imprime la suma junto con el ***nombre del programa***.

3. No se proporcionan argumentos. En este caso, el programa *intenta leer de algo que no está inicializado* y procede como en (2), a pesar de que no se imprime el nombre del programa.

Si el número total de argumentos es al menos 1, *se imprime la suma total más uno* de todos los caracteres y palabras en todos los ficheros y el *número de líneas del último fichero*.

Ahora se señalan en negrita los puntos donde la especificación original que no coinciden con la especificación obtenida por abstracción:

count cuenta el *número de líneas, palabras y caracteres* que hay en cada fichero que se le pasa como argumento. Las palabras son secuencias de caracteres que están separadas por uno o más espacios, tabuladores o saltos de línea.

Si alguno de los ficheros que se le pasa como argumento no existe, aparece *por la salida de error* el mensaje de error correspondiente y *se continúa procesando el resto de ficheros*. Si no se indica ningún fichero, **count lee de la entrada estándar**.

Se muestra por pantalla los valores computados para cada fichero (junto con el *nombre del fichero*) así como la *suma total de cada valor para todos los ficheros*. Si se procesa un único fichero o si se procesan datos provenientes de la entrada estándar, entonces *no se imprime ninguna suma*. La salida se imprime en el orden siguiente: primero líneas, luego palabras, luego caracteres y finalmente bien el nombre del fichero o la palabra “total” para la suma. Si se ha leído de la entrada estándar, el cuarto valor (nombre) no aparece.

E.5 Faltas del programa “count”

Por tanto, los defectos del programa “count” son (se señala entre llaves el tipo de falta):

1. Falta en línea 10: Las variables están inicializadas a 1 y deberían estar a 0.

{Comisión, Inicialización}

Causa fallo: Las sumas son incorrectas (por una unidad).

2. Falta en línea 14: La variable “fp” no está inicializada en caso de que la entrada se tome de “stdin” (entrada estándar).

{Omisión, Inicialización}

Causa fallo: El programa no puede leer de la entrada estándar

.

3. Falta en línea 15: La llamada a “fprintf” usa “stdout” en lugar de “stderr”.

{Comisión, Interfaz}

Causa fallo: Los mensajes de error aparecen en la salida estándar (stdout) en lugar de la salida estándar de errores (stderr).

4. Falta en línea 16: El componente termina con “exit(1)”, donde debería haberse usado un “continue”.

{Comisión, Control}

Causa fallo: Si no se encuentra un fichero, el programa para en lugar de continuar con los otros ficheros; tampoco se imprime una suma

.

5. Falta en línea 33: Se usa *argv en lugar de argv[i].

{Comisión, Datos}

Causa fallo: El programa imprime su propio nombre en lugar del nombre del fichero cuando muestra las cuentas.

6. Falta en línea 40: La comparación debe ser un menor estricto.

{Comisión, Computación}

Causa fallo: Siempre se intenta leer un fichero de más, que no existe.

7. Falta en línea 41: Argc se compara con 1, y debería ser comparado con 2.

{Comisión, Computación}

Causa fallo: El programa imprime sumas incluso cuando sólo se procesa un único fichero.

8. Falta en línea 42: En lugar de “tlinec”, se usa la variable “linec”.

{Comisión, Datos}

Causa fallo: Las sumas no se computan correctamente. Por ejemplo:

C:\count fichero2 fichero2 fichero2

1	2	14	count
1	2	14	count
1	2	14	count
1	7	43	total

A continuación aparecen señaladas en el código las faltas enumeradas anteriormente:

```
#include <stdio.h>
```

```
main (argc, argv)
```

```
int argc;
```

```
char *argv[];
```

```

{
    int      c, i, inword;
    FILE     *fp;
    long     linect, wordct, charct;
    long     tlinect = 1, twordct = 1, tcharct = 1;

    i = 1;
    do {
        if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
            fprintf (stdout, "can't open %s\n", argv[i]);
            exit (1);
        }
        linect = wordct = charct = 0;
        inword = 0;
        while ((c =getc(fp)) != EOF) {
            ++charct;
            if (c == '\n')
                ++linect;
            if (c == ' ' || c == '\t' || c == '\n')
                inword = 0;
            else if (inword == 0) {
                inword = 1;
                ++wordct;
            }
        }
        printf("%7ld %7ld %7ld", linect, wordct, charct);
        if (argc > 1)
            printf(" %s\n", *argv);
        else
            printf("\n");
        fclose(fp);
        tlinect += linect;
        twordct += wordct;
        tcharct += charct;
    } while (++i <= argc);
    if (argc > 1)
        printf("%7ld %7ld %7ld total\n", linect, twordct, tcharct);

```

```
    exit(0);  
}
```

Anexo F. PROGRAMA “TOKENS” PARA PRACTICAR LA TÉCNICA DE ABSTRACCIÓN SUCESIVA

Este Anexo se estructura del siguiente modo:

- F.1. Especificación del programa “tokens”
- F.2. Programa “tokens”
- F.3. Faltas del programa “tokens”

F.1. Especificación del programa “tokens”

Especificación del programa “tokens”

Nombre

tokens – ordena/cuenta tokens alfanuméricos

Uso

tokens [-ai] [-c chars] [-m cuenta]

Descripción

tokens lee de la entrada estándar, cuenta todos los tokens alfanuméricos que aparecen e imprime el número de veces que aparecen, siguiendo un orden lexicográfico ascendente.

Opciones

- “-a”: Sólo tiene en cuenta tokens alfabéticos (sin dígitos).
- “-c chars”: Permite que los caracteres sean parte de los tokens.
- “-i”: Causa que el programa ignore la diferencia entre mayúsculas y minúsculas pasando toda la entrada a minúsculas.
- “-m cuenta”: Indica el número mínimo de veces que han de aparecer los tokens en la entrada para que aparezca en la salida.

Ver también

wc(1), sort(1), uniq(1).


```

/*COPYRIGHT (c) Copyright 1992 Gary Perlman */

#include <stdio.h>
#include <string.h>
#include <assert.h>

int Ignore = 0;
int Mincount = 0;
int Alpha = 0;
char MapAllowed[256];

typedef struct tnode
{
    char    *contents;
    int     count;
    struct tnode *left;
    struct tnode *right;
} TNODE;

void treeprint(tree) TNODE *tree;
{
    if (tree != NULL)
    {
        treeprint(tree->left);
        if (tree->count > Mincount)
            printf("%7d\t%s\n", tree->count, tree->contents);
        treeprint(tree->right);
    }
}

TNODE *
install(string, tree) char *string; TNODE * tree;
{
    int cond;
    assert(string != NULL);
    if (tree == NULL)
    {

```

```

        if (tree = (TNODE *) calloc(1, sizeof(TNODE)))
        {
            tree->contents = strdup(string);
            tree->count = 1;
        }
    }
else
    {
        cond = strcmp(string, tree->contents);
        if (cond < 0)
            tree->left = install(string, tree->left);
        else if (cond == 0)
            tree->count++;
        else
            tree->right = install(string, tree->right);
    }
return(tree);
}

```

```

char *
getword(ioptr) FILE *ioptr;
{
    static char string[1024];
    char *ptr = string;
    register int c;
    assert(ioptr != NULL);
    for (;;)
    {
        c = getc(ioptr);
        if (c == EOF)
            if (ptr == string)
                return(NULL);
            else
                break;
        if (!MapAllowed[c])
            if (ptr == string)
                continue;
    }
}

```

```

        else
            break;
        *ptr++ = MapAllowed[c];
    }
    *ptr = '\0';
    return(string);
}

void tokens(ioptr) FILE *ioptr;
{
    TNODE *root = NULL;
    char *s;
    assert(ioptr != NULL);
    while (s = getword(ioptr))
        root = install(s, root);
    treeprint(root);
}

int main(argc, argv) int argc; char ** argv;
{
    int c, errcnt = 0;
    extern char *optarg;

    while ((c = getopt(argc, argv, "ac:im:")) != EOF)
        switch(c)
        {
            {
                case 'a': Alpha = 0; break;
                case 'c':
                    while (*optarg)
                    {
                        MapAllowed[*optarg] = *optarg;
                        optarg++;
                    }
                break;
                case 'i': Ignore = 1; break;
                case 'm': Mincount = atoi(optarg); break;
                default: errcnt++;
            }
        }
}

```

```

    }
    if (errcnt)
    {
        fprintf(stderr, "Usage: %s [-i] [-c chars] [-m count]\n", *argv);
        return(1);
    }
    for (c = 'a'; c <= 'z'; c++)
        MapAllowed[c] = c;
    for (c = 'A'; c <= 'Z'; c++)
        MapAllowed[c] = Ignore ? c - 'A' + 'a' : c;
    if (!Alpha)
        for (c = '0'; c <= '9'; c++)
            MapAllowed[c] = c;
    tokens(stdin);
    return(0);
}

```

F.3. Faltas del programa “tokens”

La clasificación de cada falta se da entre llaves.

1. Falta en la línea 25: El símbolo “>” debería ser “>=”.

{Comisión, Control}

Causa fallo: Si se proporciona un valor límite n con el argumento “-m”, se usa el valor n+1 en lugar de n.

2. Falta en la línea 64/77: La longitud de la entrada no se comprueba.

{Omisión, Datos}

Causa fallo: El programa hace un “core dumped” si se lee una palabra muy larga.

3. Falta en línea 97: El vector “MapAllowed” no se inicializa nunca.

{Comisión, Inicialización}

Causa fallo: Es dependiente del compilador; se puede aceptar por error símbolos no alfanuméricos.

4. Falta en línea 101: La variable “Alpha” se inicializa a 0 en lugar de a 1.

{Comisión, Inicialización}

Causa fallo: El argumento -a no tiene efecto.

5. Falta en línea 115: El argumento -a no aparece en el mensaje de uso.

{Omisión, Cosmético}

Causa fallo: El mensaje de ayuda no dice nada acerca del argumento -a.

Anexo G. PROGRAMA “SERIES” PARA PRACTICAR CON LA TÉCNICA DE ABSTRACCIÓN SUCESIVA

El contenido de este Anexo es el siguiente:

- G.1. Especificación del programa “series”
- G.2 Código del programa “series”
- G.3 Faltas del programa “series”

G.1. Especificación del programa “series”

Nombre

`series` – genera series numéricas aditivas.

Uso

`series` comienzo fin [cadencia]

Descripción

`series` imprime los números reales comprendidos entre **comienzo** y **fin**, con el formato de uno por línea. `series` empieza por **comienzo**, al que suma o resta sucesivamente, según corresponda, la **cadencia**, para acercarse, posiblemente llegar a, pero nunca pasar el **fin**.

Si todos los argumentos son números enteros, sólo se generan números enteros en la salida. La **cadencia** debe ser un número distinto de cero; si no se especifica, se asume que es de tamaño unitario (1). En el resto de los casos, `series` imprime el mensaje de error correspondiente.

Ejemplo

Para contar de 1 a 100:

```
series 1 100
```

Para hacer lo mismo pero al revés:

```
series 100 1
```

Limitaciones

El número de dígitos significativos reportados está limitado. Si el ratio del rango de la series entre la cadencia es demasiado grande, aparecerán varios números iguales.

La longitud máxima de una serie está limitada al tamaño del máximo entero largo que pueda ser representado en la máquina que está siendo usada. Si se excediese este valor, el resultado es impredecible.

G.2 Código del programa “series”

```
#include<stdio.h>
#include<ctype.h>
#include<fcntl.h>

#define startstr  argv[1]
#define endingstr argv[2]
#define stepstr   argv[3]

#define GFORMAT "%g\n"
#define IFORMAT "%.0f\n"

char    *Format = GFORMAT;
int      Onlyint;
double  Start;
double  Ending;

#define RANGE (Ending-Start)
#define FZERO  10e-10
#define fzero(x) (fabs (x) < FZERO)

double  Step = 1.0;
extern double fabs();
extern double atof();

int isinteger (char *cadena)
{
    int res;
    char cadena2[30];

    res = atoi (cadena);
    sprintf (cadena2, "%d", res);
    return (!strcmp (cadena, cadena2));
}
```

```

int number (char *cadena)
{
    int numero;
    float res;
    char cadena2[30];

    numero = sscanf (cadena, "%f%s", &res, cadena2);
    return (numero == 1);
}

```

```

int main (int argc,
          char **argv)
{
    long    nitems;
    long    item;
    double value;
    int     nargs = argc - 1;

    switch (nargs)
    {
        case 3:
            if (! number(stepstr))
            {
                printf("El argumento #3 no es un numero: %s\n", stepstr);
                exit(1);
            }
        case 2:
            if (! number(startstr))
            {
                printf("El argumento #1 no es un numero: %s\n", endingstr);
                exit(1);
            }
            if (! number(startstr))
            {
                printf("El argumento #2 no es un numero: %s\n", endingstr);
                exit(1);
            }
            break;
    }
}

```



```

default:
    printf("USO comienzo fin cadencia\n");
    exit(1);
}

Start    = atof(startstr);
Ending   = atof(endingstr);
Onlyint  = isinteger(startstr) && isinteger(endingstr);
if (nargs == 3)
{
    Step    = fabs(atof(stepstr));
    if (! fzero(RANGE) && fzero(Step))
    {
        printf("cadencia no puede ser cero\n");
        exit(0);
    }
    Onlyint &= isinteger(stepstr);
}

if (Onlyint)
    Format = IFORMAT;

if (fzero(RANGE))
    nitems = 2;
else
    nitems = RANGE / Step + 1.0 + FZERO;

for (item = 0; item < nitems; item++)
{
    value = Start + Step * item;
    if (fzero(value))
        printf(Format, 0.0);
    else
        printf(Format, value);
}

exit(0);
}

```

G.3 Faltas del programa “series”

La clasificación de cada falta se da entre llaves.

1. Falta en línea 66-67: En “printf()”, se usa la variable “endingstr” en lugar de la variable “startstr”.

{Comisión, Datos}

Causa fallo: Aunque se reconoce como error el que el primer argumento no sea un número, el mensaje de error correspondiente muestra el segundo argumento.

2. Falta en línea 70: En la condición del if, se usa la variable “startstr” en lugar de “endingstr”.

{Comisión, Datos}

Causa fallo: El programa no reconoce como error el proporcionar un segundo argumento no numérico.

3. Falta en línea 100: A la variable “nitems” se le asigna 2 en lugar de a 1.

{Comisión, Inicialización}

Causa fallo: Si la distancia entre el primer y el segundo parámetro se evalúa como cero, entonces se generan dos líneas de salida a pesar de que sólo debería aparecer una.

4. Falta en línea 94-95: El tratamiento del caso “fin < comienzo” se ha olvidado.

{Omisión, Computación}

Causa fallo: No se genera salida en caso de que el valor de inicio sea mayor que el de fin.

Anexo H. EJERCICIO DE LECTURA BASADA EN DISEÑO

Este Anexo tiene la siguiente estructura:

- H.1. Preguntas desde la perspectiva del diseñador
- H.2. Faltas encontradas en los requisitos

H.1. Preguntas desde la perspectiva del diseñador

Imagina que debes generar un diseño del sistema a partir del cual se pueda implementar el mismo. Usa el enfoque y técnica que prefieras para ello, e imagina que debes incorporar todos los objetos de datos necesarios, las estructuras de datos y las funciones. A la hora de hacer eso, hazte a ti mismo las siguientes preguntas:

1. ¿Se han definido todos los objetos necesarios? (datos, estructuras de datos y funciones)
2. ¿Se han especificado todos los interfaces? ¿Son consistentes?
3. ¿Se pueden definir todos los tipos de datos? (es decir, se han especificado las unidades correspondientes así como la precisión requerida?)
4. ¿Está disponible toda la información necesaria para hacer el diseño? ¿Se han especificado todas las condiciones para todos los objetos?
5. ¿Hay puntos en los cuales no estás seguro de lo que deberías hacer porque el requisito/especificación funcional no está claro o no es consistente?
6. ¿Tiene sentido el requisito/especificación funcional con respecto a lo que sabes de la aplicación o de lo que se especifica en la descripción general/introducción?

H.2. Faltas encontradas en los requisitos

1. ¿Se han definido todos los objetos necesarios? (datos, estructuras de datos y funciones)
Requisito funcional 10: No se especifica el significado de “actualizar” los distintos ficheros.
2. ¿Se han especificado todos los interfaces? ¿Son consistentes?
Requisito funcional 6: No se especifica que deba haberse introducido el número de cuenta antes de poder introducir el número de una cinta.

3. ¿Se pueden definir todos los tipos de datos? (es decir, se han especificado las unidades correspondientes así como la precisión requerida?)

Requisito funcional 3: ¿Qué es un “registro de transacción”?

4. ¿Está disponible toda la información necesaria para hacer el diseño? ¿Se han especificado todas las condiciones para todos los objetos?

Requisitos funcionales 18/19: ¿Cómo se restringen estas operaciones sólo a la dirección?

5. ¿Hay puntos en los cuales no estás seguro de lo que deberías hacer porque el requisito/especificación funcional no está claro o no es consistente?

Requisito funcional 2: Hay distintas formas potencialmente de interpretar “estado actual” para clasificar las cintas: en tienda/alquilada, dañada/buen estado, rebajada/precio normal, ...

6. ¿Tiene sentido el requisito/especificación funcional con respecto a lo que sabes de la aplicación o de lo que se especifica en la descripción general/introducción?

Página 13: Otro requisito: Este requisito no tiene sentido porque el sistema debería ejecutarse sólo en PCs

Anexo I. EJERCICIO DE LECTURA BASADA EN PRUEBAS

Este Anexo tiene la siguiente estructura:

- I.1. Preguntas desde la perspectiva del validador
- I.2. Faltas encontradas en los requisitos

I.1. Preguntas desde la perspectiva del validador

Para cada requisito/especificación funcional, intenta generar mentalmente un caso de prueba o conjunto de casos de prueba que te permitan asegurar que una determinada implementación del sistema satisface el requisito/especificación funcional. Usa el enfoque y técnica de pruebas que prefieras, e incorpora el criterio de la prueba en el caso de prueba. Al hacer eso, pregúntate a ti mismo las siguientes cosas para cada caso de prueba:

1. ¿Tienes toda la información necesaria para identificar el ítem que va a ser probado y el criterio de la prueba? ¿Puedes generar un caso de prueba razonable para cada ítem basándote en el criterio?
2. ¿Puedes estar seguro de que los casos de prueba generados conducirán a los valores correctos en las unidades correctas?
3. ¿Hay otras interpretaciones de este requisito que el implementador pueda hacer basándose en la forma en que está definido el requisito/especificación funcional? ¿Afectará esto a los casos de prueba que tú generes?
4. ¿Hay otro requisito/especificación funcional para el que generarías un caso de prueba similar pero que te conduciría a un resultado contradictorio?
5. ¿Tiene sentido el requisito/especificación funcional a partir de lo que sabes acerca de la aplicación o de lo que se especifica en la descripción general?

I.2. Defectos de los Requisitos

1. ¿Tienes toda la información necesaria para identificar el ítem que va a ser probado y el criterio de la prueba? ¿Puedes generar un caso de prueba razonable para cada ítem basándote en el criterio?

Requisito funcional 10: No se especifica el significado de “actualizar” los distintos ficheros.

2. ¿Puedes estar seguro de que los casos de prueba generados conducirán a los valores correctos en las unidades correctas?

Requisito funcional 3: No se especifica lo que es “información”. Hay varias interpretaciones posibles para esto.

3. ¿Hay otras interpretaciones de este requisito que el implementador pueda hacer basándose en la forma en que está definido el requisito/especificación funcional? ¿Afectará esto a los casos de prueba que tú generes?

Requisito funcional 2: Hay distintas formas potencialmente de interpretar “estado actual” para clasificar las cintas: en tienda/alquilada, dañada/buen estado, rebajada/precio normal, ...

4. ¿Hay otro requisito/especificación funcional para el que generarías un caso de prueba similar pero que te conduciría a un resultado contradictorio?

Requisito funcional 7: La descripción general especifica el número máximo de cintas que pueden estar alquiladas de una vez. El requisito funcional pone una restricción al número máximo de cintas por transacción. En cada caso, la prueba se hace para el mismo número, aunque se están probando dos cosas diferentes.

5. ¿Tiene sentido el requisito/especificación funcional a partir de lo que saber acerca de la aplicación o de lo que se especifica en la descripción general?

Requisito funcional 15: La descripción general especifica que se necesita un número de teléfono para cada cliente, pero el requisito funcional para añadir un nuevo cliente no requiere que se introduzca un número de teléfono.

Anexo J. EJERCICIO DE LECTURA BASADA EN USO

Este Anexo tiene la siguiente estructura:

- J.1. Preguntas desde la perspectiva del usuario
- J.2. Faltas encontradas en los requisitos

J.1. Preguntas desde la perspectiva del usuario

Imagina que debes definir el conjunto de funciones que un usuario del sistema debería ser capaz de llevar a cabo. Para el ejemplo vamos a trabajar sólo sobre dos tipos de usuarios: un directivo (d) y un cajero (c)).

Imagina que debes definir el conjunto de objetos de entrada necesarios para llevar a cabo cada función, y el conjunto de objetos de salida que genera cada función. Esto puede verse como escribir todos los escenarios operacionales o subconjuntos de escenarios operacionales que el sistema debería llevar a cabo. Comienza con los escenarios más obvios o nominales, y continúa hasta llegar a los escenarios menos comunes o condiciones especiales/contingencia. Al hacerlo, pregúntate a ti mismo las siguientes cosas para cada escenario:

1. ¿Están especificadas en los requisitos o especificaciones funcionales todas las funciones necesarias para escribir el escenario operacional? (es decir, ¿están especificadas todas las capacidades listadas en la descripción general?)
2. ¿Están claras y son correctas las condiciones iniciales para comenzar el escenario operacional?
3. ¿Están bien definidas las interfaces entre funciones? ¿Son compatibles? (es decir, ¿están linkadas las entradas de una función a las salidas de la función previa?)
4. ¿Puedes llegar a un estado del sistema que debe ser evitado? (por ejemplo por razones de seguridad)
5. ¿Puede dar alguna parte del escenario operacional un resultado distinto dependiendo de cómo se interprete un requisito/especificación funcional?
6. ¿Tiene sentido el requisito/especificación funcional teniendo en cuenta lo que sabes acerca de la aplicación de lo que se especifica en la descripción general?
- 7.

J.2. Defectos de los Requisitos

8. ¿Están especificadas en los requisitos o especificaciones funcionales todas las funciones necesarias para escribir el escenario operacional? (es decir, ¿están especificadas todas las capacidades listadas en la descripción general?)

No hay un requisito funcional que describa cómo acceder al estado de directivo.

9. ¿Están claras y son correctas las condiciones iniciales para comenzar el escenario operacional?

Requisito funcional 1: El menú principal especificado por el estado inicial enumera únicamente las funciones del cajero. No está claro cómo acceder a las funciones del directivo.

10. ¿Están bien definidas las interfaces entre funciones? ¿Son compatibles? (es decir, ¿están linkadas las entradas de una función a las salidas de la función previa?)

Requisito funcional 6: No se ha especificado que el número de cuenta debe introducirse antes de los códigos de barras de las cintas.

11. ¿Puedes llegar a un estado del sistema que debe ser evitado? (por ejemplo por razones de seguridad)

Falta un requisito funcional para cancelar transacciones, por tanto, el sistema puede llegar a un estado en el cual se ha introducido información incorrecta y ésta no se puede borrar.

12. ¿Puede dar alguna parte del escenario operacional un resultado distinto dependiendo de cómo se interprete un requisito/especificación funcional?

Requisito funcional 10: No se especifica el significado de “actualizar” los distintos ficheros.

13. ¿Tiene sentido el requisito/especificación funcional teniendo en cuenta lo que sabes acerca de la aplicación de lo que se especifica en la descripción general?

Requisito funcional 15: La descripción general especifica que se necesita un número de teléfono para cada cliente, pero el requisito funcional para añadir un nuevo cliente no requiere que se introduzca un número de teléfono.

Anexo K. LISTA DE DEFECTOS — SISTEMA DE VÍDEO ABC

6.5.1.1 Defectos				
Def.	Pág.	Req.	Clase	6.5.1.1.1 <u>Descripción</u>

1	7	RF1	O	Sólo se enumeran las funciones del cajero. No está claro cómo se accede a las funciones del directivo.
2	7	RF2	O	Se ha omitido la definición de “estado actual”.
3	8	RF3	O	No se ha especificado la información requerida para el registro de transacción.
4	8	-	O	Falta un requisito funcional: para manejar casos de error en búsquedas en la cuenta.
5	8	RF6	O	Falta la secuencia de la información: se debe haber introducido la cuenta del cliente antes de los identificadores de las cintas
6	9	RF7	I	Inconsistencia con requisitos generales: la página 4 especifica el número máximo de cintas permitidas en alquiler, no el número máximo de cintas que se puedan alquilar en una transacción concreta. Falta un requisito funcional para cancelar transacciones.
7	9	-	O	Se ha omitido la definición: “tecla de opción de orden completa”
8	9	RF10	O	Se ha omitido la definición: “actualizar”
9	9	RF10	O/A	El rellenado de formularios no es parte del sistema y no necesita especificarse aquí.
10	10	RF11	E	No se ha especificado la información necesaria para imprimir. Se ha omitido la definición: “actualizar el alquiler”.
11	10	RF11	O	Se ha omitido la entrada: el número de teléfono del cliente se necesita para ser registrado.
12	11	RF14	O/A	
13	11	RF15	O	Falta un requisito funcional: para restringir estas operaciones a directivos.
14	12	RF18, RF19	O	No se ha mencionado anteriormente el archivo de datos. Requisito sin sentido: “ejecutarse en todos los ordenadores comunes”
15	13	RR2	M	
16	13	OR1	A	

Anexo L. EJERCICIO DE PRUEBA DE CAJA BLANCA

A continuación se muestra un procedimiento en pseudocódigo para calcular la media de varios números. Se han numerado las sentencias con objeto de crear el correspondiente grafo de flujo.

PROCEDURE Media;

* Este procedimiento calcula la media de 100 o menos números que se encuentran entre unos límites; también calcula el total de entradas y el total de números válidos.

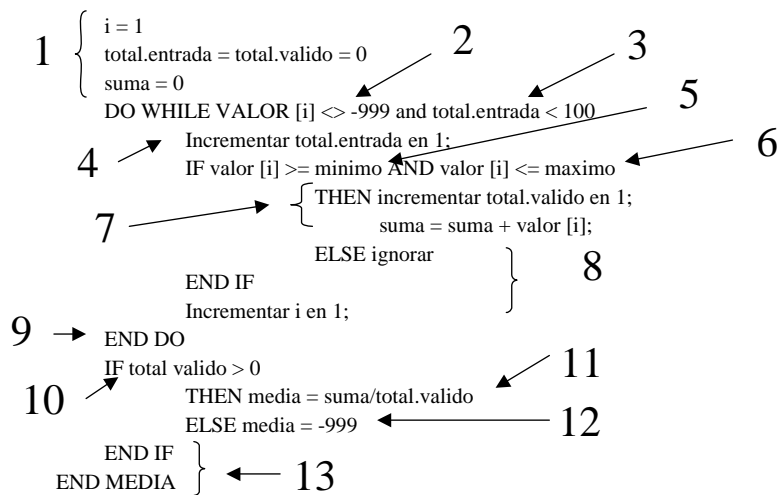
INTERFACE RETURNS media, total.entrada, total.valido;

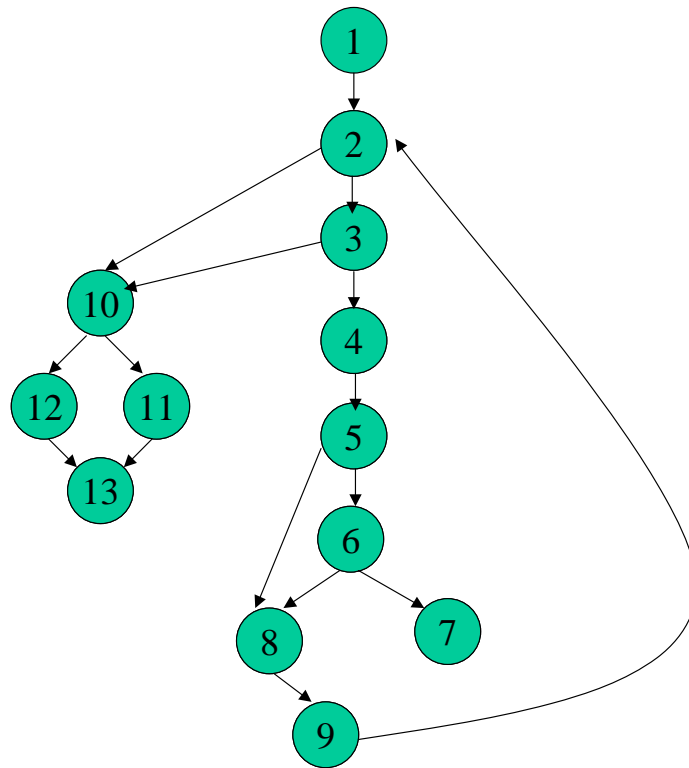
INTERFACE ACCEPTS valor, minimo, maximo;

TYPE valor [1:100] IS INTEGER ARRAY;

TYPE media, total.entrada, total.valido, minimo, maximo, suma IS INTEGER;

TYPE i IS INTEGER;





El grafo tiene seis regiones por lo que el camino básico estará formado por seis caminos en el programa. Estos caminos pueden ser:

Camino 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 2, 10, 11, 13

Camino 2: 1, 2, 3, 4, 5, 6, 7, 8, 9, 2, 10, 12, 13

Camino 3: 1, 2, 3, 10, 11, 13

Camino 4: 1, 2, 3, 4, 5, 8, 9, 2, 10, 12, 13

Camino 5: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 13

Camino 6: 1, 2, 10, 12, 13

Veamos los posibles casos de prueba que se pueden generar para probar estos caminos.

Número de Camino	Caso de Prueba	Objetivo	Resultado Esperado
1	valor = [0, -999] mínimo = 0 máximo = cualquier entero	Probar el cálculo de la media pasando una vez por el ciclo	media = 0 total.entrada= 1 total.valido = 1
	valor = 101 números o más válidos (menores o iguales que el máximo y mayores o iguales que el mínimo)	Probar el cálculo de la media pasando 100 veces por el ciclo y proporcionando 101 valores de entrada o más	media de los 100 primeros valores total.entrada=100 total.valido =100
2	No es posible realizar este camino con la bifurcación 12, ya que si se ejecuta el paso 7 esta bifurcación no es posible. Probar la bifurcación en otro caso de prueba		
3	No es posible probarlo sin pasar 100 veces por el ciclo. Probar el camino en combinación con otro que pase 100 veces por el ciclo, por ejemplo en el Camino 1	Probar el cálculo de más de 100 valores de entrada	
4	valor = [19, -999] mínimo = 20 máximo = cualquier entero	Probar el cálculo de media con valor menor que el mínimo	media = -999 total.entrada = 1 total.valido = 0
5	valor = [21, -999] mínimo = 1 máximo = 20	Probar el cálculo de media con valor mayor que el máximo	media = -999 total.entrada = 1 total.valido = 0
6	valor = -999 mínimo = cualquier entero máximo = cualquier entero	Probar el cálculo de la media con un número inválido	media = -999 total.entrada = 0 total.valido = 0

Inicialmente el camino 1 podría conducir a un caso de prueba sencillo en el que se pasara una sola vez por el bucle (este es el primero que se ha indicado). Sin embargo, dada la restricción del camino 3, este caso de prueba se puede ampliar para pasar 100 veces por el bucle y observar la reacción del sistema ante el dato 101.

Los casos de prueba 4 y 5 podrían modificarse para que se pasara un número n de veces por el bucle siendo la vez $n+1$ la que presentara el problema del valor menor que el mínimo o mayor que el máximo respectivamente.

Anexo M. EJERCICIO DE PRUEBA DE CAJA NEGRA

Considérese una aplicación bancaria, donde el usuario puede conectarse al banco por Internet y realizar una serie de operaciones bancarias. Una vez accedido al banco con las consiguientes medidas de seguridad (clave de acceso y demás), la información de entrada del procedimiento que gestiona las operaciones concretas a realizar por el usuario requiere la siguiente entrada:

- **Código del banco.** En blanco o número de tres dígitos. En este último caso, el primero de los tiene que ser mayor que 1.
- **Código de sucursal.** Un número de cuatro dígitos. El primero de ellos mayor de 0.
- **Número de cuenta.** Número de cinco dígitos.
- **Clave personal.** Valor alfanumérico de cinco posiciones. Este valor se introducirá según la orden que se desee realizar.
- **Orden.** Puede estar en blanco o ser una de las dos cadenas siguientes:
 - o “Talonario”
 - o “Movimientos”

En el primer caso el usuario recibirá un talonario de cheques, mientras que en el segundo recibirá los movimientos del mes en curso. Si este código está en blanco, el usuario recibirá los dos documentos.

A continuación, se describen las clases de equivalencia derivadas para este programa. Cada una de las clases ha sido numerada para facilitar después la realización de los casos de prueba.

Condición de Entrada	Tipo	Clase Equivalencia Válida	Clase Equivalencia No Válida
Código banco	Lógica (puede estar o no) Si está es Rango	1: En blanco 2: $100 \leq \text{Código banco} \leq 999$	3: Un valor no numérico 4: Código banco < 100 5: Código banco > 999
Código sucursal	Rango	6: $1000 \leq \text{Código sucursal} \leq 9999$	7: Código sucursal < 1000 8: Código sucursal ≥ 9999
Nº Cuenta	Valor	9: Cualquier número de cinco dígitos	10: Número de menos de cinco dígitos 11: Número de menos de cuatro dígitos
Clave	Valor	12: Cualquier cadena de caracteres alfanuméricos de 5 posiciones	13: Cadena de menos de cinco posiciones 14: Cadena de más de cinco posiciones

Orden	Conjunto, con comportamiento distinto	15: "" 16: "Talónario" 17: "Movimientos"	18: Cadena distinto de blanco y de las válidas 19: "Talónarios" 20: "Movimiento"
-------	---------------------------------------	--	--

Para generar los casos de prueba, consideremos la técnica de Análisis de Valores Límite. Esta técnica conduce a que para determinadas clases de equivalencia se genere más de un caso de prueba. Este es el caso por ejemplo, de la clases de equivalencia 2 y 6 que representan un rango de valores y para los que la técnica de Análisis de Valores Límite indica que se generen dos casos de prueba con el límite inferior y el superior del rango respectivamente (para identificar estos casos de prueba se ha añadido el sufijo a y b a las clases de equivalencia correspondientes).

Los casos de prueba resultantes se muestran a continuación.

Nº Caso	Clase de equivalencia	Banco	Sucursal	Cuenta	Clave	Orden	Resultado
1	1, 6a, 9a, 12a, 15	-	1000	00000	00000	""	Todos los movimientos y talonario
2	2a, 6b, 9b, 12b, 16	100	9999	99999	zzzzz	"Talónario"	Envío de talonario
3	2b, 6, 9, 12, 17	999	1001	12345	Hyu56	"Movimientos"	Envío de movimientos
4	3, 6, 9, 12, 15	30A	1989	12347	Kuh98	""	Código banco erróneo
5	4, 6, 9, 12, 15	99	1989	12347	Kuh98	""	Código banco erróneo
6	5, 6, 9, 12, 15	1000	1989	12347	Kuh98	""	Código banco erróneo
7	1, 6, 7, 9, 12, 16	-	999	12347	Kuh98	""	Código sucursal erróneo
8	1, 6, 8, 9, 12, 16	-	10000	12345	Hyu56	"Movimientos"	Código sucursal erróneo
9	1, 6, 10, 12, 16	-	2345	9999	Jkgy5	"Movimientos"	Número cuenta erróneo

10	1, 6, 11, 12, 16	-	7863	100000	Jkgy5	“Movimientos”	Número cuenta erróneo
11	1, 6, 9, 13, 16	-	6754	89765	Jut8	“Movimientos”	Clave errónea
12	1, 6, 9, 14, 16	-	9998	89765	Jut890	“Movimientos”	Clave errónea
13	1, 6, 9, 12, 18	-	8765	89765	Ghy78	988	Orden errónea
14	1, 6, 9, 12, 19	-	7654	89765	Ghy78	“Talones”	Orden errónea
15	1, 6, 9, 12, 20	-	8769	89765	Ghy78	“Movimiento”	Orden errónea

Anexo N. PROGRAM “TOKENS” PARA PRACTICAR CON LA TÉCNICA DE CAJA BLANCA

Nótese que este es el mismo programa que se utilizó en la parte de evaluación estática.

Especificación del programa “tokens”

Nombre

tokens – ordena/cuenta tokens alfanuméricos

Uso

tokens [-ai] [-c chars] [-m cuenta]

Descripción

tokens lee de la entrada estándar, cuenta todos los tokens alfanuméricos que aparecen e imprime el número de veces que aparecen, siguiendo un orden lexicográfico ascendente.

Opciones

- “-a”: Sólo tiene en cuenta tokens alfabéticos (sin dígitos).
- “-c chars”: Permite que los caracteres sean parte de los tokens.
- “-i”: Causa que el programa ignore la diferencia entre mayúsculas y minúsculas pasando toda la entrada a minúsculas.
- “-m cuenta”: Indica el número mínimo de veces que han de aparecer los tokens en la entrada para que aparezca en la salida.

Ver también

wc(1), sort(1), uniq(1).

Código fuente para el programa “tokens”. Fichero “tokens.c”

```
/*COPYRIGHT (c) Copyright 1992 Gary Perlman */

#include <stdio.h>
#include <string.h>
#include <assert.h>

int Ignore = 0;
int Mincount = 0;
int Alpha = 0;
char MapAllowed[256];

typedef struct tnode
{
    char    *contents;
    int     count;
    struct tnode *left;
    struct tnode *right;
} TNODE;

void treeprint(tree) TNODE *tree;
{
    if (tree != NULL)
    {
        treeprint(tree->left);
        if (tree->count > Mincount)
            printf("%7d\t%s\n", tree->count, tree->contents);
        treeprint(tree->right);
    }
}

TNODE *
install(string, tree) char *string; TNODE * tree;
{
    int cond;
    assert(string != NULL);
    if (tree == NULL)
    {
        if (tree = (TNODE *) calloc(1, sizeof(TNODE)))
        {
            tree->contents = strdup(string);
            tree->count = 1;
        }
    }
    else
    {
        cond = strcmp(string, tree->contents);
        if (cond < 0)
            tree->left = install(string, tree->left);
        else if (cond == 0)
            tree->count++;
        else
            tree->right = install(string, tree->right);
    }
    return(tree);
}
```

```

}

char *
getword(ioptr) FILE *ioptr;
{
    static char string[1024];
    char *ptr = string;
    register int c;
    assert(ioptr != NULL);
    for (;;)
    {
        c = getc(ioptr);
        if (c == EOF)
            if (ptr == string)
                return(NULL);
            else
                break;
        if (!MapAllowed[c])
            if (ptr == string)
                continue;
            else
                break;
        *ptr++ = MapAllowed[c];
    }
    *ptr = '\0';
    return(string);
}

void tokens(ioptr) FILE *ioptr;
{
    TNODE *root = NULL;
    char *s;
    assert(ioptr != NULL);
    while (s = getword(ioptr))
        root = install(s, root);
    treeprint(root);
}

int main(argc, argv) int argc; char ** argv;
{
    int c, errcnt = 0;
    extern char *optarg;

    while ((c = getopt(argc, argv, "ac:im:")) != EOF)
        switch(c)
        {
            {
                case 'a': Alpha = 0; break;
                case 'c':
                    while (*optarg)
                    {
                        MapAllowed[*optarg] = *optarg;
                        optarg++;
                    }
                    break;
                case 'i': Ignore = 1; break;
                case 'm': Mincount = atoi(optarg); break;
                default: errcnt++;
            }
        }
}

```

```

    }
    if (errcnt)
    {
        fprintf(stderr, "Usage: %s [-i] [-c chars] [-m count]\n", *argv);
        return(1);
    }
    for (c = 'a'; c <= 'z'; c++)
        MapAllowed[c] = c;
    for (c = 'A'; c <= 'Z'; c++)
        MapAllowed[c] = Ignore ? c - 'A' + 'a' : c;
    if (!Alpha)
        for (c = '0'; c <= '9'; c++)
            MapAllowed[c] = c;
    tokens(stdin);
    return(0);
}

```


Anexo O. PROGRAMA “SERIES” PARA PRACTICAR CON LA TÉCNICA DE CAJA NEGRA

Nótese que este ejercicio se utilizó también para las técnicas estáticas.

Especificación del programa “series”

Nombre

series – genera series numéricas aditivas.

Uso

series comienzo fin [cadencia]

Descripción

series imprime los números reales comprendidos entre **comienzo** y **fin**, con el formato de uno por línea. **series** empieza por **comienzo**, al que suma o resta sucesivamente, según corresponda, la **cadencia**, para acercarse, posiblemente llegar a, pero nunca pasar el **fin**.

Si todos los argumentos son números enteros, sólo se generan números enteros en la salida. La **cadencia** debe ser un número distinto de cero; si no se especifica, se asume que es de tamaño unitario (1). En el resto de los casos, **series** imprime el mensaje de error correspondiente.

Ejemplo

Para contar de 1 a 100:

```
series 1 100
```

Para hacer lo mismo pero al revés:

```
series 100 1
```

Limitaciones

El número de dígitos significativos reportados está limitado. Si el ratio del rango de la series entre la cadencia es demasiado grande, aparecerán varios números iguales.

La longitud máxima de una serie está limitada al tamaño del máximo entero largo que pueda ser representado en la máquina que está siendo usada. Si se excediese este valor, el resultado es impredecible.

Anexo P. MATERIAL PARA LA APLICACIÓN DE LECTURA DE CÓDIGO

E10: Instrucciones para aplicar la técnica “Revisión de Código”

Consultar la hoja suplementaria para aplicar la técnica “Revisión de Código” con el Programa X a medida que se leen las instrucciones

E10: Instrucciones para aplicar la técnica “Revisión de Código”

Consultar la hoja suplementaria para aplicar la técnica “Revisión de Código” con el Programa X a medida que se leen las instrucciones

Lectura del código

2. Pone el nombre en el Formulario de abstracciones (E12) y en el de recogida de datos (E11).
3. Lee por encima el código para tener una idea general del componente. Si te parece descubrir faltas mientras haces este paso a alguno de los dos siguientes, márcalas. No obstante, no pierdas demasiado tiempo en hacer un análisis preciso de ellas.
4. Determina las dependencias entre las funciones individuales del código fuente, usando para ello un árbol de llamadas. Normalmente las funciones ya estarán ordenadas en el código, de tal modo que las funciones de bajo nivel (hojas) están al comienzo y las funciones de más alto nivel (raíz) aparecen al final. Comienza aplicando la técnica de revisión de código con las funciones hoja y sigue hacia la raíz.
5. Intenta entender la estructura de cada función individual identificando las estructuras elementales (secuencias condicionales, bucles) y marcándolas. Combina las estructuras elementales para formar estructuras más grandes hasta que hayas entendido la función entera.
6. Trata de determinar el significado de cada estructura comenzando con la más interna y anota el resultado en el formulario para tal propósito (E12). Usa números de línea (línea x-y) cuando hagas eso. Evita utilizar conocimiento implícito que no resida en la estructura, por ejemplo valores iniciales, entradas o valores de parámetros. Describe las partes lo más funcionalmente posible. Mientras hagas eso, usa principios generalmente aceptados del dominio de aplicación para mantener la descripción breve y entendible. Por ejemplo, en el caso de búsqueda en un árbol, menciona “búsqueda en profundidad” en lugar de describir lo que hace la búsqueda en profundidad.

7. Cuando hayas completado las abstracciones, incluyendo tu versión de la especificación del programa, deberás “congelarlas”. Pasa al siguiente paso.

Búsqueda de inconsistencias

8. Recoge el Formulario para inconsistencias (E13) y la especificación del código (E01). Pon el nombre al Formulario para inconsistencias.
9. Lee con detenimiento la especificación que se te ha entregado. Dicha especificación describe el componente como un todo, no las funciones individuales. Detecta las posibles inconsistencias comparando la especificación con tu descripción del componente. Escribe las inconsistencias que hayas detectado en el formulario E13.

Para ello, numera las inconsistencias que hayas encontrado desde 1 hasta n en la columna denominada “Nº Inc” (número de inconsistencia) del formulario para inconsistencias. Describe la inconsistencia en las columnas a la derecha del número de inconsistencia.

10. Una vez creas que has detectado todas las inconsistencias, entrega todo el material a la persona a cargo del experimento. Ya has terminado.

E11: Formulario de Recogida de datos para la Revisión de Código

Identificador:

Nombre Alumno

Antes de comenzar...

1. ¿Cuál es tu experiencia con el lenguaje de programación C?

Experiencia (relativa). Marca la escala apropiadamente. Las marcas entre cajas son válidas.

Estimación	0	1	2	3	4	5
Comparación	ninguna	conozco la teoría	pequeños ejercicios	usado en prácticas	usado en un desarrollo	experto

2. Experiencia (absoluta). Número de años:

Resultados

Para cada entrada referida al tiempo que ha transcurrido, deduce el tiempo que hayas tomado para descansos, etc.

3. ¿A qué hora comenzaste el ejercicio de “revisión de código”? (hora:minutos)
4. ¿Cuánto tiempo necesitaste para construir las abstracciones? (horas:minutos)
5. ¿Cuántos niveles de abstracciones has generado?
6. ¿A qué hora comenzaste a buscar las inconsistencias? (hora:minutos)
7. ¿Cuánto tiempo necesitaste para encontrar las inconsistencias? (horas:minutos)
8. ¿A qué hora terminaste el experimento? (hora:minutos)
9. ¿Podrías asegurar que encontraste todas los fallos? Estima el porcentaje de fallos que has encontrado (en %)
10. ¿Cómo de bien crees que has efectuado la revisión de código? Marca la escala apropiadamente.

Estimación	0	1	2	3	4	5
Comparación	fatal	bastante mal	regular	bien	muy bien	perfectamente

E12: Formulario de abstracciones para la Revisión de Código

Identificador:

Nombre Alumno 1

Abstracciones	
Líneas	Abstracción

E13: Formulario de inconsistencias para la Revisión de Código

Identificador:

Nombre Alumno 1

Inconsistencias detectadas		
Nº Inc.	Comportamiento esperado	Número de líneas (comienzo, fin) de la abstracción y explicación breve de la inconsistencia

Anexo Q. MATERIAL PARA LA APLICACIÓN DE LAS PRUEBAS ESTRUCTURALES

E20: Instrucciones para aplicar la técnica “Pruebas Estructurales”

Consultar la hoja suplementaria para aplicar la técnica “Pruebas Estructurales” al Programa X a medida que se leen las instrucciones

Generación de casos de prueba

1. Pon el nombre en el formulario de datos de prueba (E22) y en el de recogida de datos (E21)
2. Lee de pasada el código para tener una idea general del componente. Si te parece descubrir faltas mientras haces este paso a alguno de los dos siguientes, márcalas. No obstante, no pierdas demasiado tiempo en hacer un análisis preciso de ellas.
3. Comienza a generar los casos de prueba para cumplir el criterio de cobertura (que aparece explicado al final de estas instrucciones). En la hoja suplementaria aparecen propiedades especiales del componente.
4. Anota el propósito del caso de prueba (una breve descripción de la prueba) y el caso de prueba en el formulario E22.
5. Una vez que hayas alcanzado la cobertura deseada o de que no puedes conseguir una mejor, has terminado la primera parte de este ejercicio. Por favor, no generes más casos de prueba a partir de éste momento.

Criterio de Cobertura

Obtener cobertura de decisiones (y de sentencias) significa que cada rama en un componente se ha ejecutado al menos una vez. Las ramas en los programas escritos en C las crean las sentencias: **if**, **¿**, **for**, **while** y **do-while**. Cada una de estas sentencias genera dos decisiones (ramas); la evaluación de la expresión de la decisión debe tomar valores verdadero y falso al menos una vez.

Las ramas también las puede crear la construcción **switch-case**. Para probar todas las ramas, se deben ejecutar todas las etiquetas **case** al menos una vez. Esto incluye la etiqueta **default**, incluso si no está escrita explícitamente en el código fuente. Cada etiqueta **case** genera una decisión única.

E21: Formulario de Recogida de datos para las Pruebas Estructurales

Identificador:

Nombre Alumno

Antes de comenzar...

1. ¿Cuál es tu experiencia con el lenguaje de programación C?

Experiencia (relativa). Marca la escala apropiadamente. Las marcas entre cajas son válidas.

Estimación	0	1	2	3	4	5
Comparación	ninguna	conozco la teoría	pequeños ejercicios	usado en prácticas	usado en un desarrollo	experto

2. Experiencia (absoluta). Número de años:

Resultados

Para cada entrada referida al tiempo que ha transcurrido, deduce el tiempo que hayas tomado para descansos, etc.

1. ¿A qué hora comenzaste el ejercicio de prueba estructural? (hora:minutos)
2. ¿Cuánto tiempo necesitaste para elaborar los casos de prueba? (horas:minutos)
3. ¿Cuántos casos de prueba generaste?
4. ¿Cómo de bien crees que has efectuado la prueba estructural? Marca la escala apropiadamente.

Estimación	0	1	2	3	4	5
Comparación	fatal	bastante mal	regular	bien	muy bien	perfectamente

E22: Formulario de Datos de Prueba para las Pruebas Estructurales

Identificador:

Nombre Alumno

N ^a de caso	Propósito del caso	Datos de prueba

Anexo R. ANEXO G. MATERIAL PARA LA APLICACIÓN DE PRUEBAS FUNCIONALES

E30: Instrucciones para aplicar la técnica de “Pruebas Funcionales”

Consultar la hoja suplementaria para aplicar la técnica “Pruebas Funcionales” al Programa X a medida que se leen las instrucciones

Generación de casos de prueba

1. Pon el nombre al formulario para las clases de equivalencia (E32), al de datos de prueba para las pruebas funcionales (E33), y al de recogida de datos (E31).
2. Lee cuidadosamente la especificación del código. Úsala para obtener las clases de equivalencia y escríbelas en el formulario E32.
3. Genera casos de prueba eligiendo valores de las clases de equivalencia y anota los casos de prueba en el formulario E33. La columna nombrada “Nº Clase Equival.” Debería contener el/los número/s de las clases de equivalencia del formulario E32 que el caso de prueba está ejercitando. Mientras hagas esto, ignora cualquier dependencia del sistema (no del programa).
4. En este momento se congelarán los casos de prueba generados. No puedes generar ningún caso de prueba más.

E31: Formulario de Recogida de datos para las Pruebas Funcionales

Identificador:

Nombre Alumno

Antes de comenzar...

1. ¿Cuál es tu experiencia con el lenguaje de programación C?

Experiencia (relativa). Marca la escala apropiadamente. Las marcas entre cajas son válidas.

Estimación	0	1	2	3	4	5
Comparación	Ninguna	conozco la teoría	pequeños ejercicios	usado en prácticas	usado en un desarrollo	experto

2. Experiencia (absoluta). Número de años:

3. Procedencia (escuela/Facultad):

Resultados

Para cada entrada referida al tiempo que ha transcurrido, deduce el tiempo que hayas tomado para descansos, etc.

1. ¿A qué hora comenzaste el ejercicio de pruebas funcionales? (hora:minutos)
2. ¿Cuántas clases de equivalencia generaste?
3. ¿Cuántos casos de prueba generaste?
4. ¿Cuánto tiempo necesitaste para generar los casos de prueba? (horas:minutos)
5. ¿Cómo de bien crees que has efectuado las pruebas funcionales? Marca la escala apropiadamente.

Estimación	0	1	2	3	4	5
Comparación	fatal	bastante mal	regular	bien	muy bien	perfectamente

E32: Formulario de Clases de Equivalencia para las Pruebas Funcionales

Identificador:

Nombre Alumno

Clases de Equivalencia	
Número	Descripción

E33: Formulario de Datos de Prueba para las Pruebas Funcionales

Identificador:

Nombre Alumno

N ^a de caso	Nº Clase Equival.	Datos de prueba

--	--