UNIVERSIDAD ESTATAL A DISTANCIA
VICERRECTORÍA ACADÉMICA
ESCUELA CIENCIAS EXACTAS Y NATURALES

Análisis de Sistemas I

Código: 827

Guía de estudio

Preparada por:
Alonso Marín Sánchez

San José, Costa Rica
2009

Edición académica:
Ana María Sandoval Poveda

Encargado de cátedra:
Roberto Morales

Revisión filológica:
Óscar Alvarado Vega

Motivación

Este trabajo nace con la intención de favorecer el proceso de aprendizaje de los estudiantes que han optado por el diplomado en Informática de la Universidad Estatal a Distancia (UNED), puntualmente en el curso de Análisis de Sistemas I.

Para resolver un problema de cualquier índole se necesita realizar un proceso de obtención y análisis de información que permita entender lo que se enfrenta y elaborar la o las estrategias para resolverlo.

Con el tema del análisis de los sistemas de información ocurre lo mismo. Es preciso, e inevitable para los analistas, entender la naturaleza del problema al que están por enfrentarse, desde una perspectiva inicialmente tecnológica (cuando se están delimitando los inicios del proyecto) y, al avanzar en su desarrollo, establecer una más robusta y alimentada por el aporte gradual del tema que se está explorando.

Conocer cómo está constituido el problema es un paso muy importante para el objetivo final esperado por el cliente, mas si los analistas no tienen los mecanismos necesarios para traducir un problema en términos computacionales y presentarlo como un conjunto de procesos, métodos y herramientas, realmente se ha avanzado muy poco en pro del futuro sistema.

Para esto existe la ingeniería de sistemas, y en particular la ingeniería de *software*, rama que entra a proveer una serie de actividades asociadas a la calidad, que va a facilitar la creación de *software* robusto y consistente.

La guía que está leyendo está arraigada de manera directa con el libro de texto del curso ***Análisis de Sistemas I***, llamado **Ingeniería del *software.* Un enfoque práctico**, de Roger S. Pressman. Cada uno de los temas definidos acá, contempla una serie de capítulos agrupados de manera consciente. Esta organización expresa una perspectiva más orientada a los intereses particulares del curso y menos dentro del libro de texto, de manera generalizada.

Se espera que el esfuerzo de su parte por leer y analizar esta guía de estudio le permita ampliar su perspectiva a la hora de enfrentar, como futuro profesional, el desarrollo de aplicaciones informáticas en donde solo el análisis robusto y confiable permita garantizar la continuidad y mejoramiento del negocio, motivos que son los que dan origen a los nuevos desarrollos.

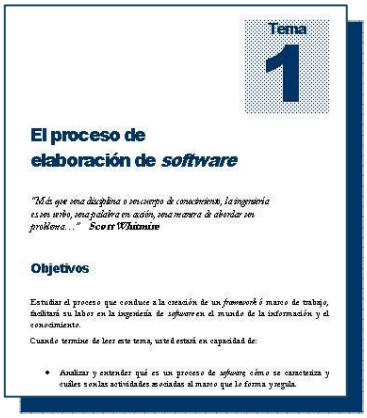
De parte de este servidor y de la UNED en general, le damos la bienvenida a este curso, esperando que sea de su agrado tanto en la narrativa como en su contenido, y que realmente ayude y facilite el proceso de enseñanza-aprendizaje del que será sujeto este período lectivo.

Cómo estudiar con esta guía

Esta guía de estudio está separada en cuatro temas, cada uno tiene asociados varios capítulos relacionados directamente con la materia del libro **Ingeniería del *Software*. Un enfoque práctico** de Roger S. Pressman, texto base del curso Análisis de Sistemas I.

La estructura de cada una de los temas es la siguiente:

Inicio del tema

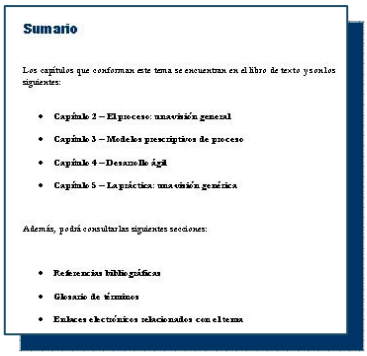

Tema
1
El proceso de elaboración de *software*
"Más que una disciplina o un cuerpo de conocimiento, la ingeniería es un verbo, una palabra en acción, una manera de abordar un problema…" **Scott Whitmire**
Objetivos
Estudiar el proceso que conduce a la creación de un *framework* o marco de trabajo, facilitará su labor en la ingeniería de *software* en el mundo de la información y el conocimiento.
Cuando termine de leer este tema, usted estará en capacidad de:
- Analizar y entender qué es un proceso de *software*, cómo se caracteriza y cuáles son las actividades asociadas al marco que lo forma y regula.

Corresponde a la presentación de cada tema, incluye el título y algún mensaje sobre la materia que está por estudiar. Se espera que analice el texto y lo compare con el tema después de haberlo estudiado.

Objetivos

En esta sección se presenta una lista de objetivos que se pretenden alcanzar con la lectura y comprensión de cada capítulo del tema.

Sumario del tema

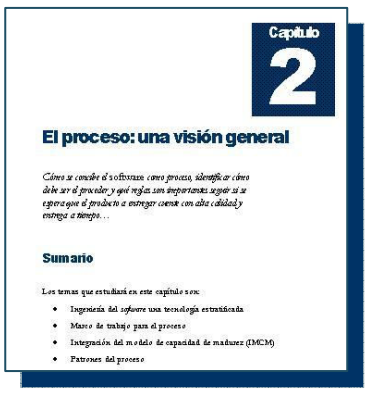

Sumario
Los capítulos que conforman este tema se encuentran en el libro de texto y son los siguientes:
- **Capítulo 2 – El proceso: una visión general**
- **Capítulo 3 – Modelos prescriptivos de proceso**
- **Capítulo 4 – Desarrollo ágil**
- **Capítulo 5 – La práctica: una visión genérica**

Además, podrá consultar las siguientes secciones:
- **Referencias bibliográficas**
- **Glosario de términos**
- **Enlaces electrónicos relacionados con el tema**

En esta parte se consigna un listado de los capítulos del libro de texto que se estudiarán en este tema. Se mantiene la numeración para ubicarlos más fácilmente y se agregaron hipervínculos para que pueda ingresar directamente a cada uno de ellos.

Además, en esta página encontrará los hipervínculos para acceder a las referencias de la bibliografía, al glosario de términos del tema y a los enlaces electrónicos de interés que se relacionan con la temática tratada en los diferentes capítulos estudiados.

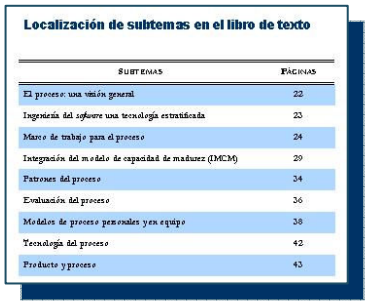
Inicio de los capítulos


Capítulo
2
El proceso: una visión general
Cómo se concibe el software como proceso, identificar cómo debe ser el procedimiento y qué reglas son importantes seguir si se espera que el producto a entregar cuente con alta calidad y entrega a tiempo…
Sumario
Los temas que estudiará en este capítulo son:
- Ingeniería del *software*: una tecnología estratificada
- Marco de trabajo para el proceso
- Integración del modelo de capacidad de madurez (IMCM)
- Patrones del proceso

La presentación de cada capítulo consiste en el título y un mensaje sobre la materia que está por estudiar. En algunas ocasiones las ideas expuestas en esta parte son de reconocidos especialistas en el tema del capítulo. Cuando es de esa manera, se indica a quién se le atribuye ese pensamiento. En otros casos, las ideas son del autor de esta guía de estudio.

Además, se despliega el sumario para ese capítulo en particular. Note que corresponde a los aspectos tratados en el libro de texto.

Localización de subtemas en el libro de texto

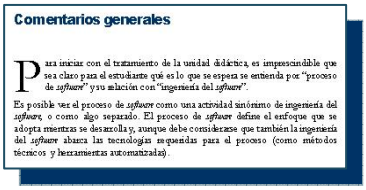

Localización de subtemas en el libro de texto

SUBTEMAS	PÁGINAS
El proceso: una visión general	22
Ingeniería del *software* una tecnología estratificada	23
Marco de trabajo para el proceso	24
Integración del modelo de capacidad de madurez (IMCM)	29
Patrones del proceso	34
Evaluación del proceso	36
Modelos de proceso personales y en equipo	38
Tecnología del proceso	42
Producto y proceso	43

Se ofrece al lector una tabla que establece la relación entre la materia de la guía de estudio y dónde está localizada en el libro de texto.

Para claridad del lector se ofrece el número de página, del libro de texto, en la cuál encontrará la información.

Comentarios generales

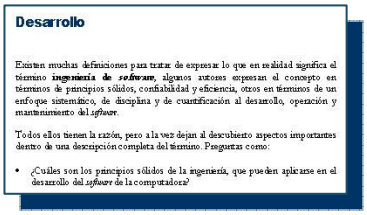

Comentarios generales

Para iniciar con el tratamiento de la unidad didáctica, es imprescindible que sea claro para el estudiante qué es lo que se espera se entienda por "proceso de *software*" y su relación con "ingeniería del *software*".

Es posible ver el proceso de *software* como una actividad sinónimo de ingeniería del *software*, o como algo separado. El proceso de *software* define el enfoque que se adopta mientras se desarrolla y, aunque debe considerarse que también la ingeniería del *software* abarca las tecnologías requeridas para el proceso (como métodos técnicos y herramientas automatizadas).

Es una introducción al capítulo que esta por estudiar. Le ayudará a hacer la transición entre el capítulo anterior y el que está por leer.

Desarrollo


Desarrollo

Existen muchas definiciones para tratar de expresar lo que en realidad significa el término **ingeniería de *software***, algunos autores expresan el concepto en términos de principios sólidos, confiabilidad y eficiencia, otros en términos de un enfoque sistemático, de disciplina y de cuantificación al desarrollo, operación y mantenimiento del *software*.

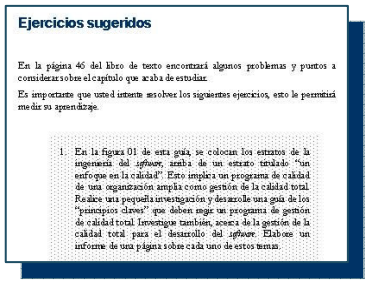
Todos ellos tienen la razón, pero a la vez dejan al descubierto aspectos importantes dentro de una descripción completa del término. Preguntas como:

- ¿Cuáles son los principios sólidos de la ingeniería, que pueden aplicarse en el desarrollo del *software* de la computadora?

Después de este título comienza la materia que se ha de estudiar para cada capítulo. El desarrollo de esta guía fue pensado y escrito para que el estudiante obtenga una perspectiva diferente de lo que está por aprender directamente del libro de curso. Desde un enfoque liviano y simple se busca explicar los argumentos más importantes expuestos por el autor del texto base, sin pretender adueñarse de los conocimientos que el mismo ofrece, sino más bien complementar esta información con una narrativa diferente y con una serie de **actividades**.

En el desarrollo de los temas, encontrará una serie de figuras elaboradas por el autor para ilustrar los temas tratados.

Ejercicios sugeridos


Ejercicios sugeridos

En la página 46 del libro de texto encontrará algunos problemas y puntos a considerar sobre el capítulo que acaba de estudiar.

Es importante que usted intente resolver los siguientes ejercicios, esto le permitirá medir su aprendizaje.

1. En la figura 01 de esta guía, se colocan los estratos de la ingeniería del *software*, arriba de un estrato titulado "un enfoque en la calidad". Esto implica un programa de calidad de una organización amplia como gestión de la calidad total. Realice una pequeña investigación y desarrolle una guía de los "principios claves" que deben regir un programa de gestión de calidad total. Investigue también, acerca de la gestión de la calidad total para el desarrollo del *software*. Elabore un informe de una página sobre cada uno de estos temas.

Ejercicios sugeridos

Referencias


Referencias

Larman, Craig (2004) **UML y Patrones**. México D.F.: Prentice Hall, segunda edición.

McConnell, Steve (1996) **Desarrollo y gestión de proyectos informáticos**. México D.F.: Prentice Hall, segunda edición.

En cada uno de los temas encontrará una lista de referencias. Contiene los materiales utilizados para la elaboración de la guía.

Encontrará los datos acomodados en orden alfabético.

Glosario de análisis de *software*

Cada sección, correspondiente a un tema de estudio de esta guía, tiene una sección de glosario. En ella encontrará un conjunto de términos complejos y su correspondiente significado. Esto le ayudará a interpretar su utilización en el contexto de cada temática.

Enlaces electrónicos

Para cada tema hay una lista de enlaces electrónicos que contienen información relevante con respecto a la existente en cada capítulo y que sirve como material de apoyo en su estudio.

En cada caso se indica el capítulo del libro que utiliza el concepto o tema y la dirección electrónica de cada referencia.

Tema

1

El proceso de elaboración de *software*

"Más que una disciplina o un cuerpo de conocimiento, la ingeniería es un verbo, una palabra en acción, una manera de abordar un problema…" ***Scott Whitmire***

Objetivos

Estudiar el proceso que conduce a la creación de un *framework* ó marco de trabajo, facilitará su labor en la ingeniería de *software* en el mundo de la información y el conocimiento.

Cuando termine de leer este tema, estará en capacidad de:

- Analizar y entender qué es un proceso de *software*, cómo se caracteriza y cuáles son las actividades asociadas al marco que lo forma y regula.
- Explicar la importancia de utilizar un modelo prescriptivo idóneo, que permita especificar, realizar prototipos, diseñar, implementar, revisar y probar las actividades definidas en un proyecto de desarrollo de *software* particular.
- Definir qué es el desarrollo rápido dentro de la jerga del desarrollo de *software* y sus diferencias y ventajas con otras metodologías existentes.
- Conocer cuáles son los conceptos que guían la ejecución de la ingeniería de *software* en la práctica, mediante principios bien establecidos, que provoquen calidad en el producto de *software* generado.

Sumario

Los capítulos que conforman este tema se encuentran en el libro de texto y son los siguientes:

- **Capítulo 2 – El proceso: una visión general**
- **Capítulo 3 – Modelos prescriptivos de proceso**
- **Capítulo 4 – Desarrollo ágil**
- **Capítulo 5 – La práctica: una visión genérica**

Además, podrá consultar las siguientes secciones:

- **Referencias bibliográficas**
- **Glosario de términos**
- **Enlaces electrónicos relacionados con el tema**

Capítulo

2

El proceso: una visión general

Cómo se concibe el software *como proceso, identificar cómo debe ser el proceder y qué reglas son importantes seguir si se espera que el producto por entregar cuente con alta calidad y suministre a tiempo…*

Sumario

Los temas que estudiará en este capítulo son:

- Ingeniería del *software*: una tecnología estratificada
- Marco de trabajo para el proceso
- Integración del modelo de capacidad de madurez (IMCM)
- Patrones del proceso
- Evaluación del proceso
- Modelos de proceso personales y en equipo
- Tecnología del proceso
- Producto y proceso

Localización de subtemas en el libro de texto

SUBTEMAS	PÁGINAS
El proceso: una visión general	22
Ingeniería del *software*: una tecnología estratificada	23
Marco de trabajo para el proceso	24
Integración del modelo de capacidad de madurez (IMCM)	29
Patrones del proceso	34
Evaluación del proceso	36
Modelos de proceso personales y en equipo	38
Tecnología del proceso	42
Producto y proceso	43

Comentarios generales

Para iniciar con el tratamiento de la unidad didáctica, es imprescindible que sea claro para el estudiante qué es lo que se espera se entienda por "proceso de *software*" y su relación con "ingeniería del *software*".

Es posible ver el proceso de *software* como una actividad sinónimo de ingeniería del *software*, o como algo separado. El proceso de *software* define el enfoque que se adopta mientras se desarrolla, aunque debe considerarse que también la ingeniería del *software* abarca las tecnologías requeridas para el proceso (como métodos técnicos y herramientas automatizadas).

El proceso del *software*, desde un punto de vista técnico, se define como un marco de trabajo para las tareas que se requieren en la construcción de *software* de alta calidad. Esta explicación deja claro que la ingeniería de *software* corresponde a las reglas dentro de un enfoque sistemático y el proceso es visto como la instancia de estas reglas aplicadas.

En resumen, el proceso del *software* se puede ver como un conjunto de actividades que permiten crear un producto de *software*. Estas actividades se definen como: **especificación**, **desarrollo**, **validación** y **evaluación**.

Es de interés en este capítulo describir ampliamente ¿qué es?, ¿cómo se aplica? y ¿para qué sirve? la ingeniería del *software*, con la finalidad de entender como reproducir los procesos, de manera tal que se logre alcanzar desarrollo de productos con calidad.

Desarrollo

Existen muchas definiciones para tratar de expresar lo que en realidad significa el término **ingeniería de *software***. Algunos autores expresan el concepto en términos de principios sólidos, confiabilidad y eficiencia, otros en términos de un enfoque sistemático, de disciplina y de cuantificación al desarrollo, operación y mantenimiento del *software*.

Todos ellos tienen la razón, pero a la vez dejan al descubierto aspectos importantes dentro de una descripción completa del término. Preguntas como:

- ¿Cuáles son los principios sólidos de la ingeniería, que pueden aplicarse en el desarrollo del *software* de la computadora?
- ¿De qué manera se construye, económicamente, un *software* confiable?
- ¿Qué se necesita para crear programas que funcionen de manera eficiente en varias máquinas reales diferentes?

Estas son preguntas no solucionadas todavía por la comunidad informática. Al no tener esas respuestas es necesario enfocar la ingeniería de *software* como una tecnología estratificada, donde se depende completamente de la interacción entre las capas en cada nivel de esta estructura, compuesta por el **proceso**, los **métodos** y las **herramientas**.

Como se comentaba al inicio, es vital comprender cuál es el alcance que tiene la ingeniería de *software* dentro del contexto de desarrollo de sistemas de información.

El libro indica que la ingeniería de *software* debe reconocerse como una tecnología organizada por estratos, donde todas las capas se confabulan para garantizar el marco que sustenta la calidad del *software*, calidad esperada por los clientes, como la esperarían al comprar cualquier otro producto tangible, como un vehículo o un par de zapatos.

En la página 24 del libro de texto encontrará una valiosa explicación de los diferentes estratos que soportan la ingeniería del *software*, envueltos en una gran cortina permeable de calidad, que le da alcance a cada uno de los estratos indicados acá.

> *Importante*:
>
> **Proceso**: es el elemento que mantiene juntos los estratos de la tecnología y que permite el desarrollo racional y a tiempo del *software*.
>
> **Métodos**: son los que proporcionan el "cómo" de la parte técnica para construir el *software*.
>
> **Herramientas**: son las que proporcionan el soporte automatizado de forma parcial o completa para el proceso y los métodos.

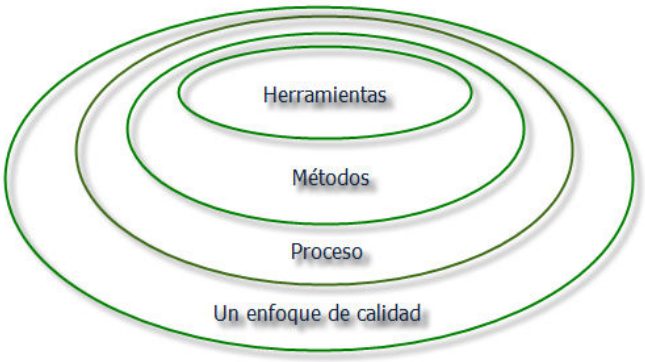


Figura 01 – Estrato de la ingeniería del software.

Es necesario ubicar la ingeniería de *software* dentro de un **marco de trabajo** para el proceso, que permite identificar un número de actividades que son comunes y se aplican a todos los proyectos de *software* por realizar.

Cada actividad va a tener una lista de **acciones** o **tareas** relacionadas que permiten devolver un material entregable al finalizar la misma.

La unidad didáctica propone un marco genérico para el proceso del *software*. Este procedimiento es enfático respecto a la **comunicación**, la **planeación**, el **modelado**, la **construcción** y el **despliegue**.

> 🕮 **Actividad**
>
> Se recomienda al estudiante analizar la descripción propuesta para cada uno de estos términos, en la unidad didáctica, y apoyarse en el cuadro informativo en la página 27, que describe fácilmente a qué se refiere el autor con acciones asociadas a una actividad puntual dentro de un marco de trabajo.

Por encima de las actividades del modelo propuesto, se encuentran otras de **tipo sombrilla**, que permiten principalmente **medir**, **gestionar** y **controlar** las acciones descritas anteriormente. Llamarlas de "tipo sombrilla" hace referencia a su papel como marco protector para la adaptación y para el buen desarrollo de las actividades a lo interno del desarrollo.

Es necesario mencionar que hay una fuerte corriente sobre el tema de los modelos **ágiles**. Estos modelos son un poco más informales y sus procesos se basan en la **agilidad**, **manejabilidad** y **adaptabilidad**, para garantizar la efectividad de los modelos tradicionales. Este aspecto se retomará en capítulos posteriores.

> 🕮 **Actividad**
>
> Se le sugiere pensar, a lo largo de los temas, en semejanzas y diferencias entre los modelos descritos en este tema y los que se mostrarán posteriormente.

Respecto a los modelos de procesos, el libro incorpora uno imprescindible para el tema. Se trata del CMM, o CMMI como se le conoce recientemente.

El Instituto de Ingeniería de *Software*, (SEI, por sus siglas en Ingles) con sede en Estados Unidos, desarrolló un modelo complejo basado en un conjunto de capacidades de *software* que deben estar presentes, forzosamente, conforme las organizaciones logran alcanzar diferentes grados de madurez y capacidad del proceso.

Como el libro de texto se editó en junio de 2005, y este modelo es apoyado por diferentes organizaciones de la industria del *software*, es evidente que el modelo ha evolucionado hasta la fecha.

Por este motivo, se le guiará para que se refiera a la literatura más actualizada sobre este tema tan importante dentro del desarrollo del *software*. Al final de este capítulo, encontrara referencias web sobre el SEI, y los nuevos lineamientos que se han madurado en torno al CMMI.

CMMI (*Capability Maturity Model® Integration*)

CMMI es el sucesor de CMM. El objetivo del proyecto CMMI es mejorar la utilización de modelos de madurez, por medio de la integración de varios modelos diferentes en un solo marco (*framework*). Fue creado por los miembros de la industria, el gobierno y el SEI. Entre los principales patrocinadores se incluyen la Oficina del Secretario de Defensa (OSD) y la *National Defense Industrial Association*.

> *Importante*:
>
> **CMMI**: el IMCM. Son siglas que, para efectos de la guía de estudio, significan lo mismo.

La SEI define el CMMI de la siguiente manera: La Integración del Modelo de Capacidad de Madurez (CMMI, por sus siglas en inglés), es un proceso de mejora en el enfoque, que proporciona a las organizaciones los elementos esenciales para los procesos efectivos. Se puede utilizar para guiar la mejora del proceso a través de un proyecto, una división, o toda una organización. CMMI ayuda a integrar las funciones de organización (tradicionalmente separadas), mejora el proceso establecido para las metas y prioridades y proporciona orientación para los procesos de calidad y un punto de referencia para evaluar los procesos actuales.

Dos representaciones: continua y escalonada

El modelo para *software* CMM-SW establece 5 niveles de madurez para clasificar a las organizaciones, en función de qué áreas de procesos consiguen sus objetivos y se gestionan con principios de ingeniería. Esto se denomina un modelo **escalonado**, o centrado en la madurez de la organización.

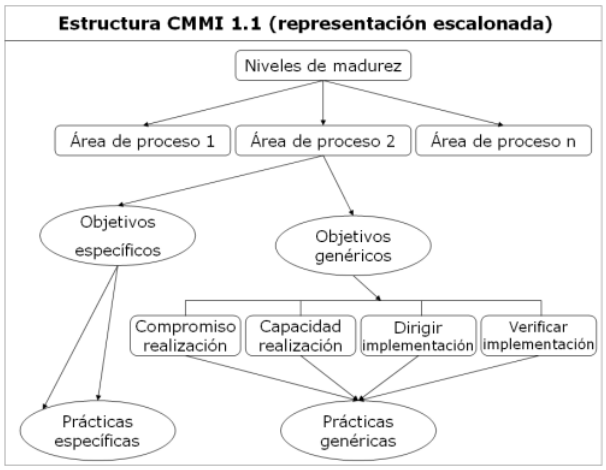


Figura 02 – Modelo escalonado CMMI.

El modelo para ingeniería de sistemas SE-CMM establece seis niveles posibles de capacidad para una de las 22 áreas de proceso implicadas en la ingeniería de sistemas (ver referencia web a la izquierda). No agrupa los procesos en cinco tramos para definir el nivel de madurez de la organización, sino que directamente analiza la capacidad de cada proceso por separado. Es lo que se denomina un modelo **continuo**.

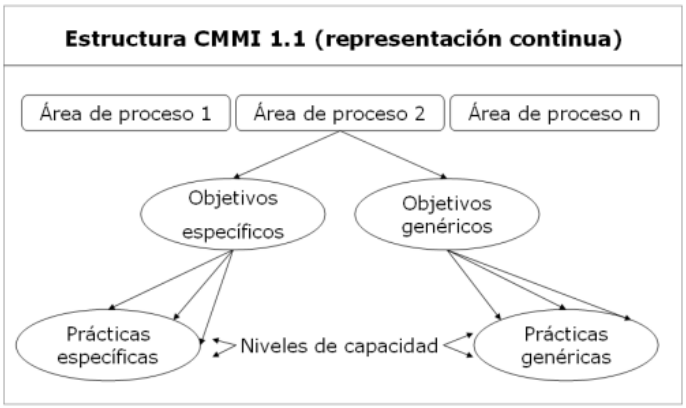


Figura 03 – Modelo continuo CMMI.

En el equipo de desarrollo de CMMI había defensores de ambos tipos de representaciones. El resultado fue la publicación del modelo con dos representaciones: continua y escalonada. Son equivalentes, y cada organización puede optar por adoptar la que se adapte a sus características y prioridades de mejora.

La visión continua de una organización mostrará la representación de nivel de capacidad de cada una de las áreas de proceso del modelo.

La visión escalonada definirá a la organización dándole en su conjunto un nivel de madurez del 1 al 5.

Los 6 niveles definidos en CMMI para medir la capacidad de los procesos son:

0) **Incompleto**: el proceso no se realiza, o no se consiguen sus objetivos.

1) **Ejecutado**: el proceso se ejecuta y se logra su objetivo.

2) **Gestionado**: además de ejecutarse, el proceso se planifica, se revisa y se evalúa para comprobar que cumple los requisitos.

3) **Definido**: además de ser un proceso gestionado se ajusta a la política de procesos que existe en la organización, alineada con las directivas de la empresa.

4) **Cuantitativamente gestionado**: además de ser un proceso definido se controla utilizando técnicas cuantitativas.

5) **Optimizante**: además de ser un proceso cuantitativamente gestionado, de forma sistemática se revisa y modifica o cambia para adaptarlo a los objetivos del negocio. Implica mejora continua.

Patrones de proceso y evaluación

Los patrones son un bastión dentro del proceso de desarrollo de *software*. Permiten crear guías con grados de abstracción bastante amplios para darle solución a problemas cotidianos en el desarrollo de *software*, de una manera robusta y ordenada.

> *Importante*:
>
> Los **patrones de proceso** se definen como un método consistente para describir una característica importante del proceso y darle una posible solución satisfactoria.

El libro presenta un posible patrón por seguir (puede ubicarlo a partir de la página 34), sin embargo, es claro que no es el único existente. El uso de un patrón no garantiza, por sí solo, que el resultado de la ejecución sea satisfactorio. Se necesita meter, dentro de una actividad evaluativa, la aplicación de dicho patrón para validar su correcta ejecución.

El libro presenta y sugiere varios enfoques interesantes sobre la *evaluación de procesos* de *software*, algunos de ellos son: CMM para mejoramiento de proceso interno, ISO 9001:2000 para *software*, entre otros.

> 🕮 **Actividad**
>
> Lea la información de las páginas 37 y 38, del libro de texto, referida al ISO 9001:2000.
>
> Consulte el enlace señalado en la página 38 y describa con sus propias palabras la importancia y la necesidad de contar con este instrumento.

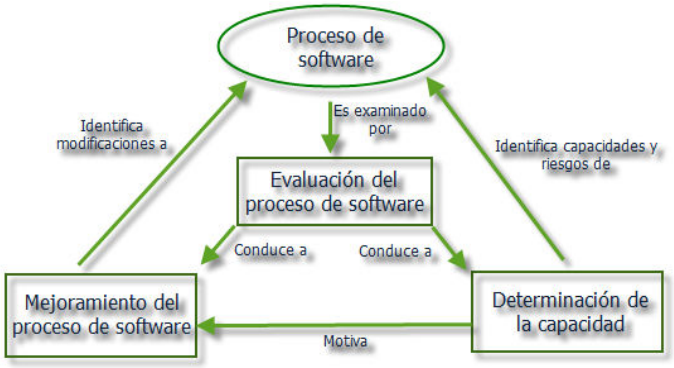


Figura 04 – Relación entre proceso del **software** *y métodos para la evaluación.*

Debido al reconocimiento internacional del ISO 9001:2000 es necesario conocer un poco más a fondo del tema. En el capítulo 26 del libro de texto, se habla sobre la gestión de calidad. La lectura de este capítulo le permitiría ampliar el tema.

Procesos de *software* personal y en equipo

Tanto los patrones como los procesos mencionados anteriormente son llevados a la práctica por diferentes personas.

La efectividad de los patrones, en las diferentes organizaciones o corporaciones, depende exclusivamente de la afinidad de ellos con la forma interna de trabajo de las personas que los utilizan.

Por lo tanto, es necesario orquestar dichos **procesos individuales**, que nacen del comportamiento de cada quien, con el **comportamiento grupal** del equipo en el proyecto asignado. Qué tan de la mano se presente esta relación, será una medida para entender qué tan bien o qué tan mal será la entrega en cada proceso terminado.

La postura en la unidad didáctica promueve que dicha relación sí existe y que se requiere de un arduo trabajo, capacitación y coordinación para lograr avanzar en este tema. Ambos procesos destacan la medición, planeación y autodirección como partes importantes de un proceso exitoso.

El detalle que justifica la existencia de estas proposiciones, tanto de proceso de ***software* personal** como de **equipo**, lo puede detallar en la página 39 a la 41 del libro de texto.

Tecnología a favor del proceso y del producto

La ejecución de los procesos de *software* se beneficia ampliamente de las **herramientas de tecnologías de procesos**. Estas herramientas permiten automatizar los marcos de trabajo de los modelos de procesos de la organización, y esto aumenta la eficiencia de las actividades de análisis de procesos, organización de tareas, control y monitoreo de progreso y administración de calidad técnica, entre otras.

La aplicación de estas herramientas disminuye los costos y aumenta la productividad de cada etapa. Esto se debe a que, normalmente, estas labores son muy tediosas de documentar de manera manual.

La relación dual existente entre el proceso y el producto es vital para que los ingenieros de *software* depuren los mecanismos para fabricar productos de alta calidad basados siempre en ingeniería del *software*, y a la vez garantizar la continuidad del proceso y del producto.

Ejercicios sugeridos

En la página 46 del libro de texto encontrará algunos problemas y puntos por considerar sobre el capítulo que acaba de estudiar.

Es importante que intente resolver los siguientes ejercicios. Esto le permitirá medir su aprendizaje.

1. En la figura 01 de esta guía, se colocan los estratos de la ingeniería del *software*, arriba de un estrato titulado "un enfoque en la calidad". Esto implica un programa de calidad de una organización amplia como gestión de la calidad total. Realice una pequeña investigación y desarrolle una guía de los "principios claves" que deben regir un programa de gestión de calidad total. Investigue también, acerca de la gestión de la calidad total para el desarrollo del *software*. Elabore un informe de una página sobre cada uno de estos temas.

2. Las actividades sombrilla ocurren a lo largo de todo proceso de *software*. ¿Se aplican de modo uniforme a través del proceso o algunas están concentradas en una o más actividades del marco de trabajo? Explique ampliamente su respuesta.

3. Describa con sus propias palabras un **marco de trabajo del proceso**. Cuando se dice que las actividades del marco de trabajo son aplicables a todos los proyectos, ¿esto significa que las mismas tareas de trabajo se aplican a todos los proyectos, sin importar el tamaño y la complejidad? Explique detalladamente.

4. Investigue acerca de CMMI, y su evolución desde CMM. Elabore un resumen acerca de este avance y sobre la aceptación en la comunidad de desarrollo de *software*.

5. ¿Cuál es el propósito de la evaluación del proceso? Indique porqué existe y de dónde proviene su importancia.

Capítulo

3

Modelos prescriptivos de proceso

Cómo ordenar el caos en el desarrollo del software*, mediante el llenado de un marco de trabajo definido, con un conjunto de tareas explícitas para cada una de las acciones de la ingeniería de sistemas.*

Sumario

Los temas que estudiará en este capítulo son:

- Modelos prescriptivos
- El modelo en cascada
- Modelos de procesos incrementales
- Modelos de proceso evolutivos
- Modelos especializados de procesos
- El proceso unificado

Localización de subtemas en el libro de texto

SUBTEMAS	PÁGINAS
Modelos prescriptivos	49
El modelo en cascada	50
Modelos de procesos incrementales	51
Modelos de proceso evolutivos	54
Modelos especializados de procesos	63
El proceso unificado	67

Comentarios generales

Sea cual sea la manera definida en una organización de llevar a cabo el desarrollo de un proyecto de *software*, debe recurrir a un marco de trabajo lleno de actividades compuestas por un grupo importante de acciones asociadas a la ingeniería de *software*.

A la vez, deberá orquestar dicho marco de trabajo al ambiente en que se desarrollará y a las personas que lo pondrán en práctica. Bajo esta premisa, los estudiosos de la materia han observado las mejores prácticas en el desarrollo de *software* a través del tiempo, y han extraído algunos **modelos prescriptivos** que permiten, para diferentes tipos de proyectos, soluciones ajustables según cada caso.

Se dice que son prescriptivos porque precisamente señalan una serie de elementos del proceso, tales como: **actividades del marco de trabajo**, **acciones de ingeniería de *software***, **tareas**, **productos de trabajo**, **aseguramiento de la calidad y mecanismos de control de cambio para cada proyecto**.

Las actividades genéricas del marco de trabajo están identificadas como: **comunicación**, **planeación**, **modelado**, **construcción** y **desarrollo**. Las características en los modelos que se describirán seguidamente, le permitirán evaluar cuál se ajusta mejor a la necesidad existente.

Desarrollo

Los modelos prescriptivos se pueden agrupar en las siguientes categorías: **lineales o en cascada**, **incrementales**, **evolutivos**, **especializados de proceso** y un caso particular de agrupamiento, conocido como el **proceso unificado**.

Para el caso de los modelos **en cascada** o **lineales**, se dirá que fue el primer modelo de proceso creado y es visto como el **modelo clásico** por excelencia. Se puede decir que nace por el sentido común aplicado al desarrollo normal de un producto de *software*.

Tiene las etapas básicas necesarias para garantizar un desarrollo consistente del producto. Una vez ejecutada y terminada cada fase, ésta sirve como insumo para la siguiente. Esto permite la continuidad del proceso y la entrega al final de un producto terminado.

Ahora bien, la existencia de los demás modelos se debe, principalmente, a que este primer paradigma presenta algunos problemas difíciles de subsanar en cuando a desarrollo se refiere.

En detalle, se presentan problemas como los siguientes:

Comportamiento no lineal: los proyectos de *software* en la actualidad tienen esta particularidad.

Requerimientos implícitos: el cliente normalmente no tiene toda la información necesaria para aclarar el dominio del problema a los analistas.

Impaciencia: la naturaleza del modelo lleva a observar un producto tangible hasta que casi todas las etapas se concretaron.

La existencia del modelo de cascada, en la actualidad, se da porque concurren situaciones donde el listado de requerimientos es muy estático (por aspectos muy propios del negocio), lo que hace importante y eficiente este modelo, mas no es lo común en estos días. Al contrario, en la gran mayoría de los casos para las organizaciones, tanto públicas como privadas, los requerimientos son completamente cambiantes día a día.

Actividad

Puede complementar la explicación para este modelo, en la página 50 del libro de texto.

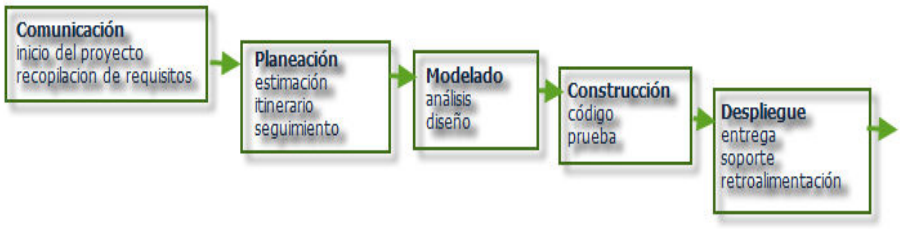


Figura 05 – Modelo de cascada.

El modelo **incremental** solventa algunas de las deficiencias encontradas en el modelo anterior. Permite separar los **requerimientos funcionales**, conocidos como las condiciones mínimas de funcionamiento que debe tener el *software*, de los **requerimientos no funcionales**, que son los que se trabajan de una manera iterativa sobre el producto de inicio y que no forman parte de la línea base del mismo.

Lo que hacen estos requerimientos no funcionales es complementar a los funcionales para darle elegancia, robustez y eficiencia al producto; sin embargo el *software*, en el peor de los escenarios, puede iniciar sin ellos.

Cada iteración o incremento del proceso, nos acerca más al producto final por desarrollar. Lo esperado en cada iteración es que devuelva un *producto funcional* con el contenido de las iteraciones hasta el momento de su creación.

> 🕮 **Actividad**
>
> Puede complementar la explicación para este modelo, en la página 52 del libro de texto.

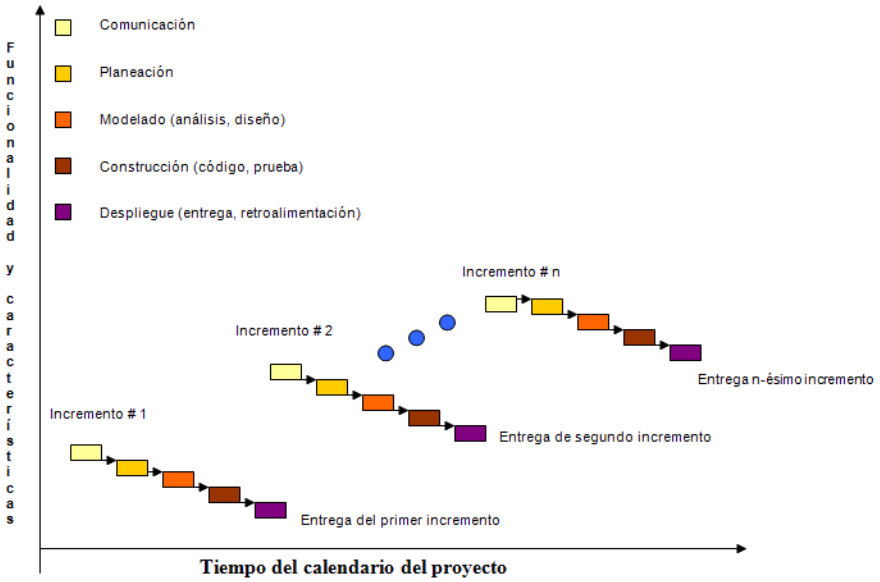


Figura 06 – Modelo incremental.

Como otra opción al hablar de modelos incrementales se encuentra al **desarrollo rápido de aplicaciones** (DRA), el cual refiere siempre a proyectos con problemas de calendario ajustado. Cuando se tienen los requerimientos claros (cosa que sucede muy pocas veces) y se logra delimitar el ámbito de proyecto, este modelo es fuerte candidato a dar solución a dicho desarrollo.

La separación por equipos de trabajo, de todo el conjunto de personas que van a trabajar en el proyecto, permite modularlo en grandes funciones para las cuales se van a aplicar todas las actividades del marco de trabajo definidas en el capítulo 1, como lo son la **comunicación**, **planeación**, **modelado**, **construcción** y **despliegue**.

Existen inconvenientes para este modelo, el principal corresponde a no lograr la cantidad necesaria de recurso humano cuando el proyecto es de grandes dimensiones (muchos equipos DRA). Por otro lado, si el proyecto no se modula de una manera adecuada, la construcción de los componentes será problemática, sin excepción.

> 🕮 **Actividad**
>
> Puede complementar la explicación para este modelo, en la página 53 del libro de texto.

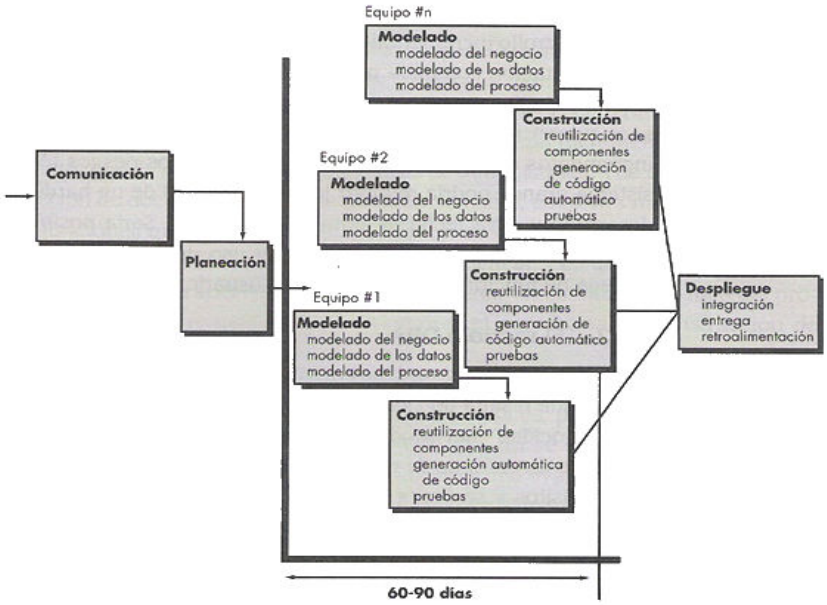


Figura 07 – Modelo desarrollo rápido de aplicaciones.

Aparte de los aspectos mencionados hasta el momento, que describen los problemas a considerar en el desarrollo del *software*, se tienen que tomar en cuenta aspectos propios del mercado como la competitividad, que propone una fecha meta de salida para el producto.

Se necesitan mecanismos que permitan desarrollar versiones del producto en los tiempos que marque la **competencia**, y que a la vez permitan ser la base directa para iniciar, sobre este, el nuevo esfuerzo por cumplir para la siguiente meta propuesta en el tiempo.

Por este motivo, se crean los **modelos evolutivos**, que a su vez son iterativos y que permiten obtener versiones cada vez más completas del *software*, ajustado a tiempos específicos.

Como primer ejemplo de estos modelos evolutivos, se señala la **construcción de prototipos**. Este paradigma de desarrollo permite que el cliente y el ingeniero de *software* identifiquen los requerimientos conocidos, tengan claros los objetivos globales a cumplir con el desarrollo y las áreas en donde haga falta más información.

Con este insumo, algo disminuido para lo que normalmente se ocupa, se inicia la creación de un **prototipo visual** (sin contenido o código desarrollado), en donde se plasma lo que el cliente o usuario final esperaría ver del producto terminado al final del proceso.

El cliente evalúa este prototipo y se retroalimenta al equipo de trabajo para que perfeccione la propuesta de este primer prototipo base. El objetivo es conseguir, a partir de la alimentación visual y la interacción con el prototipo, la versión que cumpla con las expectativas.

Este es un punto crucial, porque al encontrar el producto a nivel de prototipo, que agrupe las características necesarias y esperadas por el cliente, el equipo de trabajo deberá, en la buena teoría, extraer de este trabajo la lista de requerimientos y **eliminarlo** casi por completo, para no provocar la sensación, ni en el cliente ni en el equipo, de que la tarea está casi por terminar.

Recuerde que para este modelo en particular, se inicia de alguna forma de atrás hacia adelante, obteniendo primero el caparazón del sistema, y luego el contenido de operación.

Con la lista anterior deberá iniciar la creación del nuevo proyecto que ahora sí será el producto final que se ha de entregar al cliente.

📖 **Actividad**

Puede complementar la explicación para este modelo, en la página 55 del libro de texto.

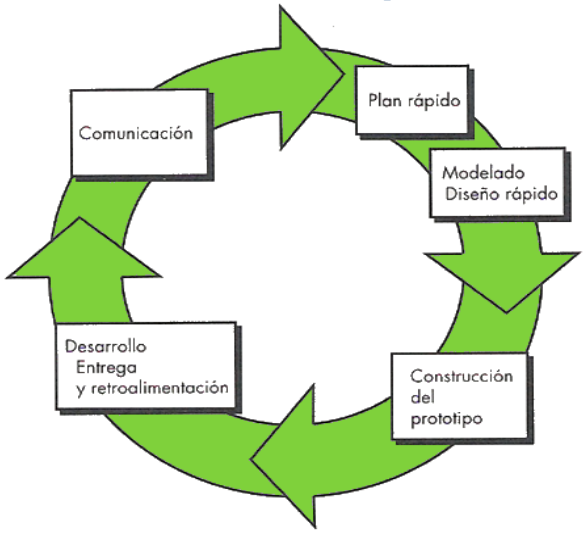


Figura 08 – Modelo por prototipo.

El modelo por prototipo también tiene sus deficiencias a la hora de interactuar con el cliente, debido a que le crea una **falsa expectativa** de que hace falta muy poco para pasar del prototipo a la versión utilizable, desconociendo que la esencia está en el código que va dentro del prototipo que observa y que es precisamente lo que está por iniciar.

El **modelo en espiral** es otro de los modelos evolutivos. Es muy famoso entre los modelos de *software* debido a la importancia que le otorga al manejo y administración del **riesgo** para el proyecto.

Tal y como lo describe el nombre del modelo, se trata de una espiral que en cada uno de los circuitos o iteraciones aplica cada una de las actividades genéricas del marco de trabajo explicado anteriormente

(capítulo 1). Lo hace de manera tal que el factor **riesgo** se considera ampliamente en cada revolución, con el fin de mitigarlo o eliminarlo según sea el caso.

La primera vez puede ser que se trate tan solo de la especificación del producto, pero posiblemente el paso siguiente sea generar un prototipo, y así avanzar hasta obtener el producto final ampliamente refinado por la espiral.

Se necesita mucho conocimiento por parte de las personas que quieren implementar este modelo, debido a la habilidad necesaria para abstraer el riesgo en cada iteración y ponerlo en beneficio del proyecto. De hecho, este es el motivo por el cual no siempre se escoge como modelo, aunque en el papel sea tan conveniente.

> 🕮 **Actividad**
>
> Puede complementar la explicación para este modelo, en la página 59 del libro de texto.

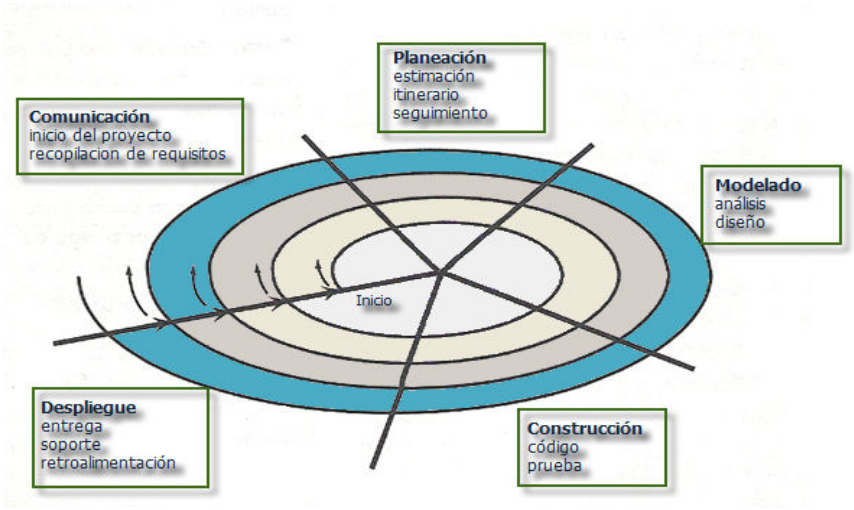


Figura 09 – Modelo en espiral.

Como último modelo propuesto para los tipos evolutivos, tenemos el **de desarrollo concurrente**, que se representa en forma esquemática como una serie de actividades del marco de trabajo, acciones y tareas de la ingeniería de *software* y sus **estados asociados**.

En un momento del proceso, todas las actividades del marco de trabajo existen de manera concurrente, posiblemente en estados diferentes. Este proceso se aplica a todos los desarrollos de *software* y proporciona una visión exacta del estado del proyecto. En lugar de confinar las actividades, acciones y tareas a una secuencia de eventos, define una red de actividades concurrente.

> 🕮 **Actividad**
>
> Puede complementar la explicación para este modelo, en la página 60 del libro de texto.

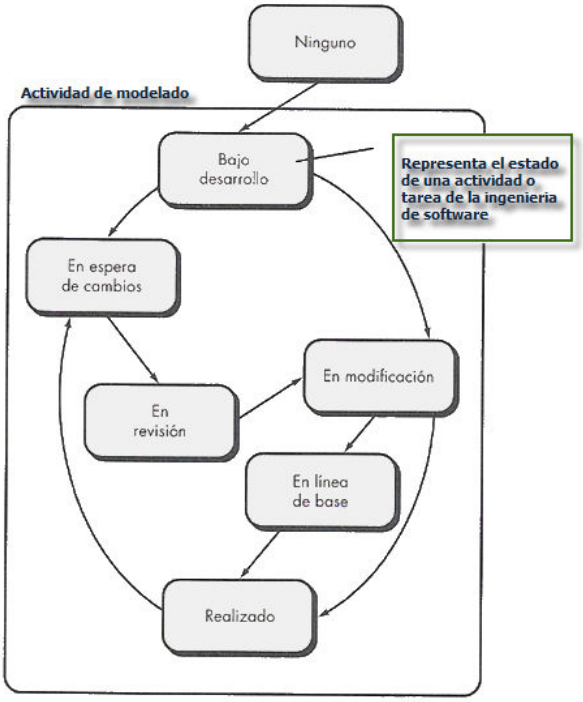


Figura 10 – Modelo desarrollo concurrente.

Los modelos **especializados de procesos** son otra variante. Su fortaleza es la reutilización de los modelos convencionales descritos en párrafos anteriores. Entran en vigencia cuando ya se ha tomado un enfoque de ingeniería de *software* de manera robusta.

Dentro de esta gama tenemos al **desarrollo basado por componentes**, que busca reducir bruscamente los tiempos de la etapa de construcción del *software*, mediante la **componentización**.

> *Importante*:
>
> La **componentización** se describe como la separación o empaquetado de módulos de sistema, clases o paquetes de clases orientadas a objetos, que se logran orquestar dentro de la etapa de construcción y que permiten, mediante su adquisición, reducir ampliamente los tiempos, gracias a que están completamente desarrollados.

Su funcionalidad está dirigida a la presentación mediante interfaces amigables, que permiten al *software* integrarse fácilmente al resto de *software* desarrollado o incorporado de la misma manera.

> 🕮 **Actividad**
>
> Puede complementar la explicación para este modelo, en la página 63 del libro de texto y en todo el capítulo 30.

También, dentro de los modelos especializados en los procesos, se encuentra el modelo de **métodos formales**. Este modelo está sustentado en la verificación matemática rigurosa y en el análisis estadístico de las pruebas arrojadas en diferentes etapas del desarrollo, principalmente en la parte de pruebas unitarias e integrales del *software*.

> 🕮 **Actividad**
>
> Puede complementar la explicación para este modelo, en la página 64 del libro de texto y en los capítulos 28 y 29.

Como último ejemplo dentro de los modelos especializados al proceso, se tiene al **desarrollo del *software* orientado a aspectos**. Se refiere, principalmente, a las áreas de interés para el cliente, que permite identificar qué aspectos del *software* deben ser fuertemente tratados para que, a partir de un enfoque metodológico, se defina, especifique, diseñe y construyan aspectos del *software*, vistos como mecanismos más allá de subrutinas y ligados para localizar la expresión de un interés general.

Las técnicas orientadas a aspectos extienden las técnicas tradicionales como la orientación a objetos, permitiendo a los desarrolladores encapsular en módulos separados **los aspectos**, propiedades que normalmente atraviesan o están presentes en varios componentes del sistema.

Esta tecnología propugna la identificación, separación y modularización de los distintos **aspectos** que intervienen en una aplicación, para componerlos luego y construir la aplicación final.

El beneficio principal de esta tecnología es la mejora dada en la modularización de los sistemas; con esto se obtiene un código menos "enmarañado", evitándose la mezcla entre funcionalidad y aspectos extra-funcionales, facilitándose el mantenimiento y la evolución del código.

> 🕮 **Actividad**
>
> Puede complementar la explicación para este modelo en la página 65 del libro de texto y en los capítulos 28 y 29.

Proceso unificado (PU)

El tema del proceso unificado del *software*, debido a su masiva utilización en la actualidad, necesita un espacio aparte para su descripción como modelo de desarrollo de *software*.

El PU es un proceso de *software* guiado por los **casos de uso**, de arquitectura céntrica, iterativo e incremental diseñado como un marco que permite a todos los métodos y herramientas del UML (*Unified Modeling Language*) convivir.

> *Importante*:
>
> Un caso de uso es una "colección de escenarios con éxito y fallos relacionados, que describen a los actores utilizando un sistema para satisfacer un objetivo" (Larman 2004).

En el proceso unificado se encuentran claramente representadas, mediante fases, las cinco grandes actividades genéricas del marco de trabajo para cualquier modelo de procesos.

Importante:

Fase de inicio: es la comunicación con el cliente para identificar requerimientos de negocios (casos de uso) y la planeación de los mismos.

Fase de elaboración: es la continuación de comunicación y además actividades de modelado del negocio genérico del proceso. Algunas veces se alcanza un sistema ejecutable en su primera versión o iteración.

Fase de construcción: es la parte constructiva de un proceso genérico de *software*. Se da la codificación de cada incremento de análisis dado por la fase de elaboración anterior, para la iteración actual.

Fase de transición: es la acción que abarca las últimas actividades de la fase de construcción y la primera parte de la actividad de despliegue. Se dan las primeras pruebas al *software* y la retroalimentación del usuario para la siguiente iteración.

Fase de producción: es la monitorización de las fases subsecuentes del *software*, paralelo a la fase de transición. Permite retomar informes por desperfectos y cambios nuevos en el *software*.

Es importante indicar que al ser un proceso **iterativo incremental**, al realizarse alguna de las etapas de construcción, transición y producción para un caso específico, ya se ha iniciado una nueva iteración para el siguiente incremento del proceso, lo que nos indica que existe concurrencia en las etapas.

El **proceso unificado de *rational*** (*Rational Unified Process* en inglés, habitualmente resumido como **RUP**) es un proceso de desarrollo de *software* y junto con el Lenguaje Unificado de Modelado UML, constituye la metodología estándar más utilizada para el análisis, implementación y documentación de sistemas orientados a objetos para el PU.

Actividad

A su izquierda está la referencia de la página de IBM, empresa de computación que compró recientemente los derechos de *Rational Software*. Investigue los productos relacionados con la ingeniería de *software* y la forma en cómo facilitan cada vez más el proceso operativo a los equipos de trabajo.

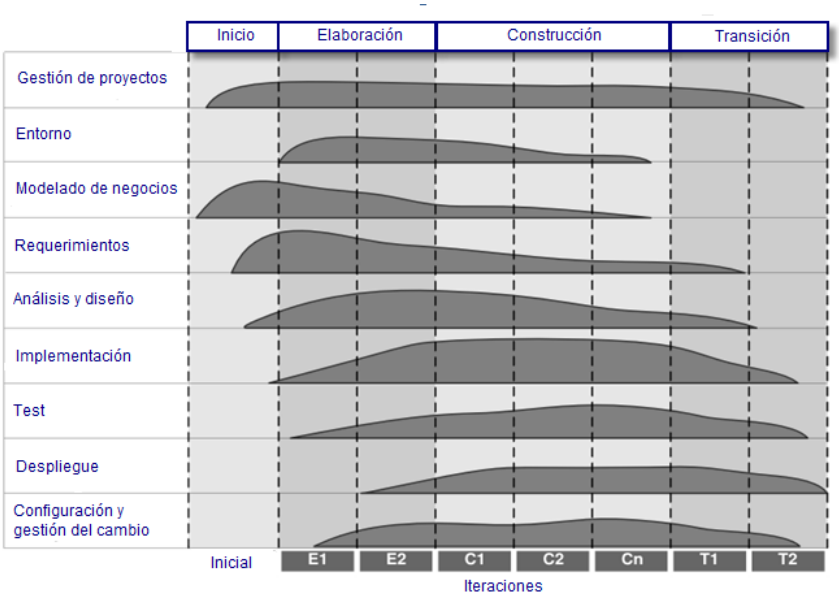


Figura 11. Diagrama fases y actividades del proceso unificado.

Actividad

Puede complementar la explicación para este modelo en la página 67 del libro de texto. Tendrá acceso a una breve historia del PU. Además, en la parte 2 de su material, verá los métodos que pueblan el proceso y las técnicas del UML.

Cuadro de capacidades de algunos modelos de ciclo de vida del **software**. ***Fuente: McConnell, Steve (1996)*** *Desarrollo y gestión de proyectos informáticos*. ***México D.F.: Prentice Hall, segunda edición.***

CAPACIDADES DEL MODELO DEL CICLO DE VIDA	CASCADA PURA	ESPIRAL	PROTOTIPO EVOLUTIVO	ENTREGA EVOLUTIVA
Trabaja con poca identificación de los requerimientos	Malo	Excelente	Excelente	Medio a excelente
Trabaja con poca comprensión sobre la arquitectura	Malo	Excelente	Malo a medio	Malo
Genera un sistema altamente fiable	Excelente	Excelente	Medio	Medio a excelente
Genera un sistema con amplio desarrollo	Excelente	Excelente	Excelente	Excelente
Gestiona riesgos	Malo	Excelente	Medio	Medio
Esta sometido a una planificación predefinida	Medio	Medio	Malo	Medio
Requiere poco tiempo de gestión	Malo	Medio	Medio	Medio
Permite modificaciones a medio camino	Malo	Medio	Excelente	Medio a excelente
Ofrece a los clientes signos visibles de progreso	Medio	Excelente	Medio	Excelente
Ofrece a la directiva signos visibles de progreso	Medio	Excelente	Medio	Excelente
Requiere poca sofisticación para los directivos y desarrolladores	Medio	Malo	Malo	Medio

El cuadro anterior le presenta una comparación entre algunos de los modelos descritos en este capítulo. En la primera columna contiene varias características importantes entre los modelos eficientes.

En fila inicial, están indicados algunos de los modelos que son de interés y que son utilizados comúnmente.

Actividad

Analice detenidamente el cuadro de capacidades de los modelos de ciclo de vida del *software*. Observe los valores asignados en cada una de las celdas. Indique en cuáles concuerda con el valor asignado y en cuáles no. Si su respuesta es negativa, describa los motivos que lo llevan a razonar de esa manera.

Ejercicios sugeridos

En la página 74 del libro de texto encontrará algunos problemas y puntos por considerar sobre el capítulo que acaba de estudiar.

Es importante que intente resolver los siguientes ejercicios. Esto le permitirá medir su aprendizaje.

6. Dé tres ejemplos de proyectos de *software* adaptables al modelo en cascada. Sea específico.

7. Para lograr un desarrollo rápido, el modelo DRA asume la existencia de una cosa. ¿Cuál es y por qué dicha suposición no siempre es verdadera?

8. Proporcione tres ejemplos de proyectos de *software* adaptables al modelo incremental. Sea específico.

9. ¿Cuáles son las ventajas y desventajas de desarrollar *software* con calidad "lo suficientemente buena"? Esto es, ¿qué pasa cuando se resalta la velocidad de desarrollo sobre la calidad del proyecto?

10. Es posible probar que un componente de *software* o incluso un programa completo está correcto. Entonces, ¿por qué no todos lo hacen?

11. Explique por qué los programas que se desarrollan utilizando el desarrollo evolutivo tienden a ser difíciles de mantener.

Capítulo

4

Desarrollo ágil

Cómo ordenar el caos en el desarrollo del software*, mediante el llenado de un marco de trabajo definido, con un conjunto de tareas explícitas para cada una de las acciones de la ingeniería de sistemas.*

Sumario

Los temas que estudiará en este capítulo son:

- Agilidad en el desarrollo
- ¿Qué es un proceso ágil?
- Modelos ágiles de proceso

Localización de subtemas en el libro de texto

SUBTEMAS	PÁGINAS
Agilidad en el desarrollo	79
¿Qué es un proceso ágil?	81
Modelos ágiles de proceso	84

Comentarios generales

La aparición del **desarrollo ágil** obedece a la necesidad del ser humano de provocar flexibilidad y dinamismo en las actividades que ejecuta. Esto lo hace para buscar siempre agilidad en los procesos y con esto, la obtención del producto en la menor cantidad de tiempo y sin perder ningún atributo importante en su confección.

En la actualidad, tal como se ha planteado repetidamente, las necesidades de mercado son muy cambiantes en el mundo del desarrollo de *software*. Esto obedece, muchas veces, a las situaciones que establece la competencia.

Por lo tanto, los equipos de desarrollo deben proveer soluciones cada vez más rápidas y robustas en contenido, pero que a la vez no disminuya su calidad a cambio de esa velocidad.

El desarrollo ágil busca precisamente eso: cómo sacarle el mayor provecho a la ingeniería de *software* pero con un nivel de desarrollo mucho más ligero o dinámico que el visto hasta este momento. Se buscará, en este capítulo, exponer la manera de conseguirlo y aplicarlo en el mundo real. Siempre habrá que tener presente que esta producción acelerada no es viable para todo tipo de proyecto, aspecto que también se explicará.

Desarrollo

Es posible encontrar la agilidad dentro del contexto del desarrollo del *software*, un ejemplo son aquellos esfuerzos en donde se asumen los lineamientos indicados por la ingeniería del *software*, en donde se le da mucha seriedad al tema del **cambio**, como eje conductor para alcanzar dicha agilidad.

> 🕮 **Actividad**
>
> Analice la descripción que ofrece Ivar Jacobson, en la página 79 del libro, para el tema de **agilidad**. Considere las apreciaciones dadas por él y cómo deben ser incluidas en un desarrollo de *software*. ¿Qué tan sencillo es lograrlo?

Además del tema del cambio, para el desarrollo ágil encontramos elementos que deben ser considerados tales como la **estimulación en la comunicación de los interesados** en el proyecto, la **entrega pronta** del *software* operativo y la **flexibilidad** en la planificación del material.

> *Importante*:
>
> El desarrollo ágil basa su existencia en la incertidumbre existente en cualquiera de las etapas del proyecto. Este titubeo se presenta al no poder controlar las modificaciones (aumento – disminución) a los requerimientos ya establecidos o que están por venir. Ante esto, solo una buena administración de soporte para el control de cambio nos ayudará a salir con los tiempos establecidos y con la calidad esperada para el desarrollo.

> 🕮 **Actividad**
>
> En la página 90 del libro de texto, se encuentran 12 principios para alcanzar la agilidad en el desarrollo, léalas y analícelas a la luz de lo entendido hasta ahora para desarrollo ágil de *software*.

Un proceso ágil de *software* debe tener fuerte **adaptabilidad** para que sea un éxito, en gran medida porque estos enfoques son incrementales, lo que indica claramente que avanzan en cuerpo para cada nueva iteración.

Esto lleva a pensar que esta **adaptación incremental** requiere aspectos como la retroalimentación del cliente, incrementos de *software* que permitan entregar soluciones o prototipos ejecutables que sirvan de catalizador al cliente, para medir el grado de avance con respecto a lo indicado en el análisis previo a la etapa de construcción.

Existen tres suposiciones claves acerca de los proyectos de *software*, que son aptos para un proceso de desarrollo ágil de *software*. Se encuentran en el siguiente recuadro:

Importante:

- Resulta difícil predecir cuáles requisitos del *software* persistirán y cuáles cambiarán.
- Para muchos tipos de *software*, el diseño y la construcción están intercalados; por lo tanto, el diseño se prueba casi cuando se crea.
- El análisis del diseño y la construcción no son predecibles.

Con estas tres suposiciones, se genera una incógnita importante por resolver: ¿se puede desarrollar un proceso manipulable de manera impredecible?

La respuesta a esto se encuentra en la **adaptabilidad** del proceso, tal y como se mencionó en párrafos anteriores, para con el proyecto y sus condiciones técnicas y de ambiente.

Tal como lo indica el libro, el **factor humano** es un elemento vital para la consecución exitosa de un proyecto. Principalmente porque el proceso **se debe** ajustar a personas y equipos de trabajo específicos, cada uno con sus talentos y habilidades puestos al servicio del proceso.

📖 **Actividad**

En las páginas 83 a 84 del libro de texto, se encuentra un grupo de rasgos clave entre la gente y el equipo de desarrollo. Es importante que las lea, las analice y las ubique en el comportamiento cotidiano del ser humano, a la luz de lo entendido hasta ahora para desarrollo ágil de *software*.

Modelos ágiles de procesos

Para los modelos ágiles de procesos de desarrollo del *software*, también hay diferentes propuestas. Todas buscan encontrar un lugar y posicionarse en la mente de los ingenieros y desarrolladores de *software*, como la propuesta más completa para dar agilidad a los proyectos de desarrollo.

Como primer exponente, está al modelo de la **programación extrema** (PE), basado en un conjunto de reglas y prácticas que ocurren en el contexto de las cuatro actividades del marco de trabajo: **planeación**, **diseño**, **codificación** y **pruebas**.

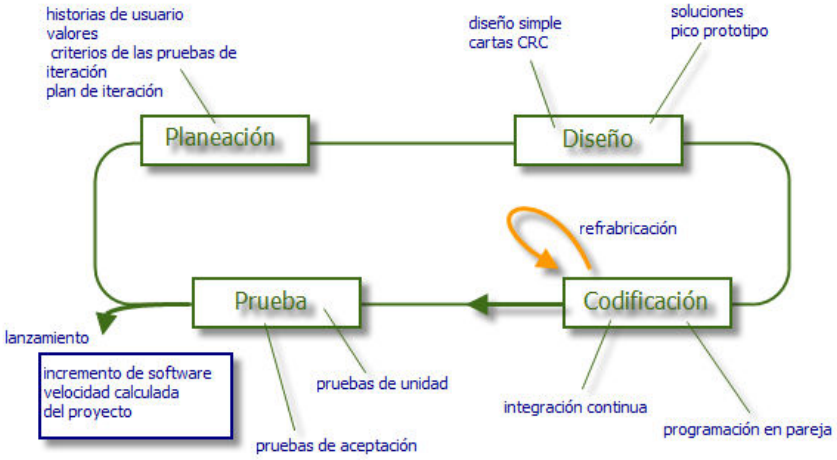


Figura 12 – Proceso de la programación extrema.

La **planeación** está basada en una serie de **historias** similares a lo que sería un **caso de uso**, que describen las características y funciones requeridas por el sistema. Estas historias se analizan y clasifican según **el peso otorgado al valor y al costo** dado, para iniciar el desarrollo según sea ese peso.

En el **diseño** se intenta respetar al máximo la simplicidad de lo recolectado hasta el momento. En otras palabras, se trata de modelar la historia tal y como quedó definida en la etapa anterior.

El diseño se apoya en las tarjetas de CRC (Colaborador-Responsabilidad-Clase, capítulo 8 del libro de texto), que llevan a este modelo a un contexto orientado a objetos.

Se apoya en el concepto de **refabricación**, que permite sugerir modificaciones al diseño de forma que lo mejoren de manera radical. Este proceso se realiza tanto antes como después de la codificación.

Se espera que antes del inicio de la **codificación** se presente una etapa de pruebas de unidad, que evalúen lo indicado en las **historias** que van incluidas en la iteración de ese momento en particular. Esto se hace con el fin de favorecer el proceso de verificación de código por parte de los desarrolladores, mediante dicha retroalimentación.

En la etapa de **pruebas**, se crean las **pruebas de unidad**, que se enfocan en cada uno de los componentes desarrollados de manera individual. Dichas pruebas tratan de automatizarse dentro del marco de trabajo del proceso, para que sean exactas y repetibles cuanto se quiera.

> 🕮 **Actividad**
>
> En las páginas 84 a 88 del texto, se encuentra cada una de las etapas descritas con amplitud. Léalas y analícelas para ahondar en el tema de los procesos ágiles de desarrollo. Elabore un resumen que complemente su estudio.

Otro de los métodos descrito en el libro es el **desarrollo adaptativo del *software*** (DSA), basado en la colaboración humana y la organización propia del equipo. Está pensado en 3 fases: **especulación**, **colaboración** y **aprendizaje**. Además, utiliza un proceso iterativo que incorpora inteligentemente la planeación del ciclo adaptativo, métodos de recopilación de requisitos y un ciclo iterativo de desarrollo que incorpora grupos enfocados al cliente.

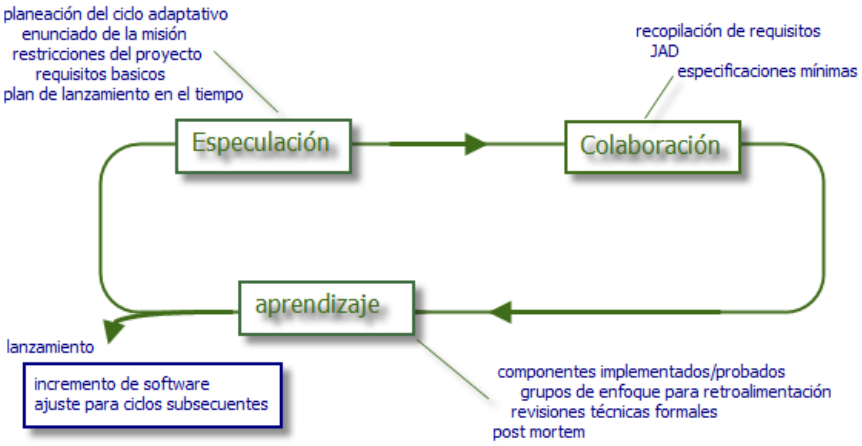


Figura 13 – Desarrollo adaptativo del software

El desarrollo de sistemas dinámicos MDSD es otro método explicación que da el texto sobre los modelos ágiles de proceso. Éste basa su enfoque en proporcionar un marco de trabajo para construir y mantener sistemas con **restricciones de tiempo** mediante prototipos incrementales en un ambiente de proyecto controlado.

Al igual que el PE y el SDA es un proceso iterativo del *software*, pero con la característica que sigue la regla del 80/20 (80% de la aplicación en el 20% del tiempo total). Esto es que solo se necesita el trabajo suficiente para cada incremento y para facilitar el pase al siguiente movimiento en ruta hacia el nuevo incremento.

También el libro nos describe el modelo **melé**, que incorpora las siguientes actividades del marco de trabajo: **requisitos**, **análisis**, **diseño**, **evolución** y **entrega**. En cada actividad del marco de trabajo, las tareas se suceden dentro del "patrón de proceso" llamado ***sprint***. El trabajo dentro de cada *sprint* se adapta al problema y, con frecuencia, lo modifica en tiempo real por medio del equipo de trabajo.

Los patrones de proceso mencionados ya han probado su uso y efectividad en proyectos con tiempos reducidos de entrega, requerimientos cambiantes y condiciones adversas para el negocio.

Cada patrón de proceso contiene un conjunto de actividades de desarrollo: **lista de retrasos**, ***sprint***, **reuniones de melé**, y **demostración**.

> 🕮 **Actividad**
>
> En las páginas 89 a 95 del texto se encuentra cada uno de estos métodos descritos con mayor amplitud. Léalos para complementar su estudio del tema.

La lista de métodos ágiles de proceso es más larga aún, contemplando a métodos como el de la familia Cristal, o los de desarrollo conducido por características (DCC), o modelado ágil (MA).

Todos estos métodos tratan, a su manera y con sus elementos distintivos, de ayudar al ingeniero de *software* a alcanzar su máxima meta: la de entregar cada desarrollo solicitado en los tiempos establecidos, con la calidad esperada y con la satisfacción del cliente de por medio.

🕮 **Actividad**

En las páginas 95 a 98 del libro de texto, se encuentra cada uno de estos métodos descritos completamente. Es necesario que realice la lectura para comprender su existencia como métodos ágiles.

Ejercicios sugeridos

En la página 101 del libro de texto encontrará algunos problemas y puntos por considerar sobre el capítulo que acaba de estudiar.

Es importante que intente resolver los siguientes ejercicios. Esto le permitirá medir su aprendizaje.

12. Describa, con palabras propias, la palabra agilidad (dentro del ámbito de proyectos de *software*).

13. ¿Podría cada uno de los procesos ágiles describirse recurriendo a las actividades genéricas del marco de trabajo mencionado en el capítulo 3 de esta guía? Construya una tabla que coloque las actividades genéricas dentro de las actividades definidas para cada proceso ágil.

14. Seleccione un principio de agilidad de los enunciados en la sección 4.1 (segunda actividad mencionada en este capítulo de la guía) y trate de determinar si cada uno de los modelos de procesos presentados en este capítulo muestran ese principio.

15. ¿Por qué cambian tanto los requisitos? Después de todo ¿la gente no sabe lo que quiere? Amplíe su respuesta.

16. Considere los siete rasgos enunciados en la sección 4.2.2 (tercera actividad mencionada en este capítulo de la guía). Ordene los rasgos con base en su percepción desde lo que es más importante a lo menos.

Capítulo

5

La práctica: una visión general

…en un principio genérico, la práctica es una colección de conceptos, principios, métodos y herramientas a las que un ingeniero de software *recurre a diario…*

Sumario

Los temas tratados en este capítulo son:

- La práctica de ingeniería de *software*
- Prácticas de comunicación
- Prácticas de planeación
- Prácticas del modelado
- Prácticas de la construcción
- Despliegue

Localización de subtemas en el libro de texto

SUBTEMAS	PÁGINAS
La práctica de ingeniería de *software*	105
Prácticas de comunicación	109
Prácticas de planeación	113
Prácticas del modelado	116
Prácticas de la construcción	122
Despliegue	126

Comentarios generales

El arte de la ingeniería de *software* es una colección de conceptos, principios, métodos y herramientas a las que los ingenieros encargados del *software* recurren todos los días para resolver problemas en la creación de programas para la computadora.

Todo aspecto que nos presente un **cómo** resolver, en cuestión de código, remite directamente a alguna de las actividades asociadas con ingeniería de *software*.

La visión genérica que se describirá permite ver todo el proceso de desarrollo, desde que se concibe la idea hasta que tenemos algo tangible en las manos.

Permite ver con claridad cómo se atienden cada una de las situaciones que aumentan o disminuyen el ritmo marcado en cada actividad realizada para la construcción del *software*.

Desarrollo

Tal y como se mencionó en los comentarios generales, la ingeniería soporta de manera metódica las actividades del quehacer informático, en cuanto a desarrollo de *software* se refiere.

La guía de estudio, en el capítulo siguiente, lo orienta para adentrarse en el mundo real del ingeniero y en la práctica de la ingeniería de *software*, en cuanto a la dimensión de las actividades y del entorno que encontrará en las diferentes organizaciones.

Todo esto va muy de la mano con las **actividades genéricas** dentro del marco de trabajo descrito en capítulos anteriores.

> *Importante*:
>
> Las actividades genéricas dentro del marco de trabajo son: **comunicación**, **planeación**, **modelado**, **construcción** y **despliegue**.

En la práctica se debe seguir una serie de **principios genéricos** y **conceptos** que rigen cada una de estas actividades.

El libro de texto hace hincapié en la importancia del tema de resolución de problemas; de hecho puntualiza los pasos necesarios que representan la **esencia** en la resolución de problemas.

Los pasos necesarios para dar solución a un problema puntual son: entender el problema, planear la solución, llevar a cabo el plan y examinar los resultados obtenidos. Estos pasos se pueden ajustar para un problema de *software*.

> *Importante*:
>
> Ajuste para el desarrollo de *software*:
>
> **Entender el problema**: comunicación y análisis.
>
> **Planear la solución**: modelado y diseño de *software*.
>
> **Llevar a cabo el plan**: generación de código.
>
> **Examinar los resultados obtenidos para probar la precisión**: realización de pruebas y aseguramiento de la calidad.

> 🕮 **Actividad**
>
> En la página 106 del libro de texto se encuentran los cuatro puntos para alcanzar la esencia de la práctica. Lea las preguntas asociadas y analícelas según su experiencia personal. Contraste estos principios con los problemas que vive en su vida diaria, y la forma en cómo los resuelve.

Los principios expuestos son, en general, **leyes** que gobiernan ciertas acciones en el pensamiento humano. Su alcance puede tener diferentes niveles de abstracción, desde estar asociados a una actividad genérica de la ingeniería del *software*, hasta estar presente en alguna de sus actividades.

En el libro de Roger S. Pressman se proponen **siete principios básicos** que se reproducen rápidamente en las actividades sobre la ingeniería de *software*, a saber:

Importante:

Principios esenciales de la ingeniería del *software*:

I Razón por la que todo existe: ofrecer un valor al usuario.

II Mantenerlo simple: todo diseño debe ser tan simple como sea posible, pero no más simple.

III Mantener la visión: una visión clara es esencial para el proyecto de software.

IV Lo que uno produzca otros lo consumirán: pensar en consumidores de la reutilización.

V Estar abierto al futuro: resolver problemas de forma general, no específica.

VI Planear para la reutilización: la reutilización ahorra tiempo y esfuerzo y su planeación, de esa manera, otorga valor agregado.

VII Pensar: cuando se tiene un pensamiento claro y completo antes de cada acción, se producen los mejores resultados.

Actividad

En las páginas 107 – 109 del libro de texto se encuentran los siete principios esenciales de la ingeniería de *software*. Lea el desarrollo de cada uno de estos principios y analícelos según su experiencia particular.

¿Por qué se consideran principios esenciales?, ¿agregaría usted algunos?

Tal y como se describió en capítulos anteriores, la etapa de la **comunicación** (permite la obtención de los requerimientos) forma parte del marco de trabajo de la ingeniería de *software*. Al igual que las demás etapas, está regida por principios que permiten llevar a cabo dicha etapa de la manera más completa y eficiente.

Importante:

Principios esenciales de la comunicación:

I Escuchar.

II Prepararse antes de comunicar.

III Alguien debe facilitar la actividad.

IV La comunicación cara a cara es lo mejor.

V Tomar notas y documentar las decisiones.

VI Buscar colaboración.

VII Conservar el enfoque, examinar un módulo a la vez.

VIII Si algo no está claro, se hace un dibujo.

IX Una vez que se llega a un acuerdo sobre algo, se debe continuar. Este estado no debe cambiar; no importa si se llega a un acuerdo posterior, o no.

X La negociación no es un concurso o un juego. Funciona mejor cuando ambas partes ganan.

📖 **Actividad**

En las páginas 110 – 111 del libro de texto, se encuentran los diez principios esenciales de la comunicación. Lea el desarrollo de cada uno de estos principios y analícelos según su experiencia particular.

¿Por qué es tan importante el levantado de requerimientos?, ¿agregaría usted algunos principios?, ¿cuáles?

En la **planeación** encontramos el camino ordenado mediante las pautas por seguir, para aplicar los objetivos tácticos definidos en la comunicación, mediante un conjunto de prácticas técnicas. Se encuentra también en esta etapa una serie de principios que guían de manera ordenada y metódica su realización:

Importante:

Principios esenciales de la **planeación**:

I Entender los alcances del proyecto.

II Involucrar al cliente en la actividad de planeación.

III Reconocer que la planeación es iterativa.

IV Estimar con base en el conocimiento disponible.

V Considerar el riesgo cuando se define el plan.

VI Ser realista.

VII Ajustar la granularidad mientras se define el plan.

VIII Definir cómo se intentará asegurar la calidad.

IX Describir cómo se pretende incluir el cambio.

X Adaptar el plan a menudo y hacer ajustes cuando se requieran.

📖 **Actividad**

En las páginas 113 – 115 del libro de texto, se encuentran los diez principios esenciales de la planeación. Lea el desarrollo de cada uno de estos principios y analícelos según su experiencia propia.

Prácticas del modelado

Los modelos de análisis y diseño permiten entender porqué y cómo se desarrollará la solución, tomando en cuenta los dominios de **información**, **funcional** y del **comportamiento** (análisis), y de características que ayudan a su construcción, como lo son la **arquitectura**, la **interfaz de usuario** y el detalle a **nivel de componentes** (diseño).

Existen una variedad de principios operativos, que permiten ejecutar estas prácticas de modelado de manera segura y ordenada, a pesar de la existencia de diferentes métodos de modelado en el mercado.

Importante:

Principios esenciales del modelado del **análisis**:

I El dominio de información de un problema debe representarse y entenderse.

II Se deben definir las funciones que ejecuta el *software*.

III Se debe representar el comportamiento del *software* (como consecuencia de eventos externos).

IV Los modelos que presentan información, función y comportamiento deben partirse de forma que descubran el detalle de una manera estratificada o jerárquica.

V La tarea del análisis debe moverse de la información esencial hacia el detalle de la implementación.

Principios esenciales del modelado del **diseño**:

I El diseño debe ser rastreable hasta el modelo de análisis.

II Siempre se debe considerar la arquitectura del sistema que se va a construir.

III El diseño de datos es tan importante como el diseño de funciones de procesamiento.

IV Las interfaces (internas y externas) deben diseñarse con cuidado.

V El diseño de interfaz del usuario debe ajustarse a las necesidades del usuario final.

VI El diseño al nivel de componentes debe ser independiente del modo funcional.

VII Los componentes deben estar apareados entre sí en formas mínimas y vinculadas con el ambiente externo.

VIII Las representaciones del diseño (modelos) deben ser fácilmente comprensibles.

IX El diseño debe desarrollarse de manera interactiva. En cada iteración el desarrollador debe buscar la mayor simplicidad.

Prácticas de la construcción

La práctica de la **construcción** establece el momento donde se toman todos los lineamientos encontrados y se desarrolla el código que va a soportar la lógica definida para la aplicación.

Posterior a esta etapa, vendrá una serie de **pruebas** de diferentes tipos, que permiten asegurar, tanto a lo interno como a lo externo, que la interacción entre los componentes está correcta y, por lo tanto, se van a obtener los resultados deseados por el cliente.

Los tipos diferentes de pruebas son: **unitarias**, de **validación** y de **aceptación**.

Importante:

Tanto las pruebas unitarias, como las de validación y las de aceptación, serán retomadas en los capítulos 13 y 14, que nos hablan sobre estrategias y técnicas para el tema de las pruebas de un sistema de información, que es tan importante y amplio en la búsqueda de *software* de calidad.

En las páginas 123 a la 125 del libro de texto encontrará detalles de algunos principios asociados, tanto a la codificación como a las pruebas en sí.

🕮 **Actividad**

A la luz de lo leído hasta ahora, analice y describa por qué es tan importante esta etapa dentro de todo el ciclo de desarrollo de *software*.

Una vez terminada la etapa de construcción, solo queda el **despliegue** del programa desarrollado y ya probado, que se coloca en las máquinas del cliente. Se debe recordar, tal y como se señaló en capítulos anteriores, que el paradigma de desarrollo de sistemas que más ha avanzado es el **evolutivo**. Este provee en todo su ciclo de acción, diferentes entregas del producto evolucionado, para tener al final una versión remozada y robusta que acapara cada avance, pero mejorado en cada iteración del ciclo.

Cada una de estas entregas, debe respetar una serie de principios claves para su correcto ingreso y despliegue:

Importante:

Principios para el ***despliegue****:*

I Se deben administrar las expectativas que el cliente tiene del software.

II Se debe ensamblar y probar un paquete de entrega completo.

III Se debe establecer un régimen de soporte antes de entregar el software.

IV Se debe proporcionar material instructivo asociado a los usuarios finales.

V El software *con errores se debe arreglar primero y entregar después.*

🕮 **Actividad**

En las páginas 126 y 127 del libro de texto encontrará los principios asociados al despliegue. Analice y describa la importancia de un buen proceso de **despliegue**, como etapa directa de interacción con el usuario o cliente final.

Ejercicios sugeridos

En la página 131 del libro de texto encontrará algunos problemas y puntos que debe considerar acerca del capítulo que acaba de estudiar.

Es importante que intente resolver los siguientes ejercicios, lo cual le permitirá medir su aprendizaje.

17. Investigue acerca de la "**facilitación**" para la actividad de comunicarse. Prepare un conjunto de directrices que se enfoquen solo en este punto.

18. ¿En que difieren la comunicación ágil y la comunicación de la ingeniería de *software* tradicional? ¿En que se asemejan?

19. Investigue acerca de la "**negociación**" para la actividad de comunicarse. Prepare un conjunto de directrices que se enfoquen solo en este aspecto.

20. Describir lo que significa **granularidad** en el contexto de un calendario de proyecto.

21. ¿Cuáles son los **tres dominios** que se consideran durante el modelado de análisis.

22. En sus propias palabras, ¿qué es una prueba exitosa? ¿Cómo se mide?

Referencias

Larman, Craig (2004) **UML y Patrones**. México D.F.: Prentice Hall, segunda edición.

McConnell, Steve (1996) **Desarrollo y gestión de proyectos informáticos**. México D.F.: Prentice Hall, segunda edición.

Glosario de análisis de *software*

CMM: *Capability Maturity Model* o Modelo de Capacidad y Madurez es un modelo de evaluación de los procesos de una organización. Fue desarrollado inicialmente para los procesos relativos al *software* por la Universidad Carnegie-Mellon para el SEI (*Software Engineering Institute*) en los Estados Unidos.

CMMI: *Capability Maturity Model Integration* es un modelo para la mejora de procesos que proporciona a las organizaciones los elementos esenciales para procesos eficaces. Las mejores prácticas CMMI se publican en los documentos llamados modelos. En la actualidad hay dos áreas de interés cubiertas por los modelos de CMMI: **desarrollo** (se tratan procesos de desarrollo de productos y servicios) y **adquisición** (se tratan: la gestión de la cadena de suministro, adquisición y contratación externa en los procesos del gobierno y la industria).

Framework: estructura de soporte definida en la cual otro proyecto de *software* puede ser organizado y desarrollado. Típicamente, un *framework* puede incluir soporte de programas, bibliotecas y un lenguaje interpretado entre otros *software* para ayudar a desarrollar y unir los diferentes componentes de un proyecto. Un *framework* representa una arquitectura de *software* que modela las relaciones generales de las entidades del dominio. Provee una estructura y una metodología de trabajo la cual extiende o utiliza las aplicaciones del dominio.

ISO: *International Organization for Standardization* u Organización Internacional para la Estandarización es el organismo encargado de promover el desarrollo de normas internacionales de fabricación, comercio y comunicación para todas las ramas industriales a excepción de la eléctrica y la electrónica. Su función principal es la de buscar la estandarización de normas de productos y seguridad para las empresas u organizaciones a nivel internacional.

OMG: (*Object Management Group*; por sus siglas en inglés) **Grupo de Gestión de Objetos** es un consorcio dedicado al cuidado y establecimiento de diversos estándares de tecnologías orientadas a objetos. Es una organización sin ánimo de lucro que promueve el uso de tecnología orientada a objetos mediante guías y especificaciones para las mismas.

UML: (*Unified Modeling Language*; por sus siglas en inglés) **Lenguaje Unificado de Modelado** es el lenguaje de **modelado** de sistemas de *software* más conocido y utilizado en la actualidad; está respaldado por el **OMG** (*Object Management Group*). Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de *software*. UML ofrece un estándar para

describir un "plano" del sistema (modelo), incluyendo aspectos conceptuales tales como procesos de negocios y funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes de *software* reutilizables.

Enlaces electrónicos

CAPÍTULO	CONCEPTO	PÁGINA
2	Página oficial CMMI	http://www.sei.cmu.edu/cmmi/
2	Última revisión del CMMI	http://www.sei.cmu.edu/cmmi/adoption/pdf/cmmi-overview07.pdf
2	Sitio en español de CMMI	http://es.wikipedia.org/wiki/CMMI
2	Imagen 1 de estructura CMMI 1.1	http://upload.wikimedia.org/wikipedia/commons/3/3c/ Estructuracmmi11cont.png
2	Página oficial de ISO	http://www.iso.org/iso/home.htm
3	Descripción del proceso unificado	http://es.wikipedia.org/wiki/Proceso_Unificado
3	Descripción del proceso unificado de Rational	http://es.wikipedia.org/wiki/RUP
3	Página de IBM acerca del RUP	http://www-306.ibm.com/software/mx/rational/
4	Descripción de programación extrema	http://es.wikipedia.org/wiki/Programaci%C3%B3n_Extrema
4	Descripción del manifiesto ágil	http://www.extremeprogramming.org/index.html
4	Descripción del manifiesto ágil, indicado en el libro de texto	http://agilemanifesto.org/
4	Descripción metodologías ágiles en desarrollo de *software*	http://www.willydev.net/descargas/prev/TodoAgil.pdf

Tema

2

La práctica de la ingeniería de *software*

"…la práctica es un amplio arreglo de conceptos, principios, métodos, y herramientas que deben considerarse cuando se planea el software. *Representa los detalles – las consideraciones técnicas y los cómo – que están bajo la superficie del proceso de* software…" ***Roger Pressman***

Objetivos

Este tema pretende llevar a la práctica todos los conceptos, métodos y técnicas mencionadas en el tema I, con la esperanza de que el estudiante encuentre la conexión y el sentido existente, entre la teoría relatada y las aplicaciones en el mundo real, para el desarrollo del *software.*

Cuando termine de estudiar este tema, estará en capacidad de:

- Analizar cuáles son los conceptos y principios que deben seguirse para una ingeniería del *software* cada vez más efectiva.
- Entender qué aspectos de la ingeniería de sistemas corresponden específicamente al área de ingeniería de requisitos, además de identificar cuáles conceptos son necesarios para su buena aplicación.
- Describir cómo se obtiene el modelo de análisis y cuáles son los elementos necesarios para soportarlo.
- Explicar en qué consiste la ingeniería de diseño, y cuáles son los conceptos que permiten darle un ambiente robusto en su realización.

Sumario

Los capítulos que conforman este tema se encuentran en el libro de texto y son los siguientes:

- **Capítulo 6 – Ingeniería de sistemas**
- **Capítulo 7 – Ingeniería de requisitos**
- **Capítulo 8 – Modelado de análisis**

Además, podrá consultar las siguientes secciones:

- **Referencias bibliográficas**
- **Glosario de términos**
- **Enlaces electrónicos relacionados con el tema**

Capítulo

6

Ingeniería de sistemas

"…no hay nada más difícil de llevar a cabo, más peligroso de realizar o de éxito más incierto que encabezar la introducción de un nuevo orden de cosas…"
Maquiavelo

Sumario

Los temas tratados en este capítulo son:

- Sistemas basados en computadora
- La jerarquía de la ingeniería de sistemas
- Ingeniería de procesos de negocios
- Ingeniería del producto
- Modelado del sistema

Localización de subtemas en el libro de texto

SUBTEMAS	PÁGINAS
Sistemas basados en computadora	134
La jerarquía de la ingeniería de sistemas	136
Ingeniería de procesos de negocios: una visión general	140
Ingeniería del producto: una visión general	142
Modelado del sistema	144

Comentarios generales

La **ingeniería de sistemas** comprende una serie de elementos que se relacionan con el desarrollo del sistema como producto final. Además, esta ingeniería se considera una serie de "partes de un todo", que permiten al sistema desarrollado terminar del modo esperado.

La **ingeniería del *software*** tan solo es un eslabón dentro de este proceso de ingeniería de sistemas. Antes de ella, se trabajó oportunamente en aspectos de análisis, diseño y organización que permitieran crear el ambiente óptimo que envuelve a la ingeniería del *software,* para que ésta trabaje bien.

En este capítulo se dará la introducción a la rama de las ingenierías que interesa en este curso: la ingeniería de *software*. Se usará la óptica de la ingeniería, vista desde un enfoque de proceso de negocios y también desde un enfoque orientado al producto como tal, para facilitar su comprensión.

Desarrollo

Sistemas basados en computadora

Para empezar es importante tener claro qué se entiende cuando se habla de la palabra **sistema**. Tanto en diccionario como en libros, encontrará definiciones de dónde escoger.

El libro de Pressman propone, como la versión más acertada para el mundo de la computación, la siguiente definición:

Importante:

Concepto de **sistema**: conjunto o disposición de elementos que están organizados para cumplir una meta predefinida al procesar información.

📖 **Actividad**

Investigue sobre otras definiciones, tanto en el libro de texto como en diferentes fuentes, para la palabra sistema. Contraste los significados encontrados con el propuesto acá y determine semejanzas y diferencias.

Los elementos que se menciona en la definición dada, es posible enlistarlos en el siguiente cuadro:

Importante:

Elementos del **sistema**.

Software: programa de computadora.

Hardware: diferentes tipos de dispositivos electrónicos tangibles que proporcionan una función externa del mundo real.

Personas: usuario y operadores de *software* y *hardware*.

Bases de datos: recopilación de información acezada mediante el *software*.

Documentación: información descriptiva que detalle el uso y operación del sistema.

Procedimientos: pasos que define el uso específico de cada elemento del sistema.

Jerarquía de la ingeniería de sistemas

La ingeniería de sistemas se puede acomodar de manera jerárquica. Esto permite navegar siempre de lo general a lo específico, y pasar por todos los estratos de esta cadena de elementos. Se puede describir dicha jerarquía de la siguiente forma:

Importante:

Jerarquía de **sistema**.

Visión global: conjunto de dominios.

VG = [D1, D2, D3,… DN]

Visión dominio: conjunto de elementos.

VD = [E1, E2, E3,… EN]

Visión elementos: conjunto de componentes.

VE = [C1, C2, C3,… CN]

Todo este enfoque le permitirá al ingeniero de sistemas tener una perspectiva amplia de la localización del elemento que resuelve una necesidad puntual. Le ayudará a entender de manera intuitiva cuándo debe bajar para atacar algún problema, según corresponda, dentro de la jerarquía presentada.

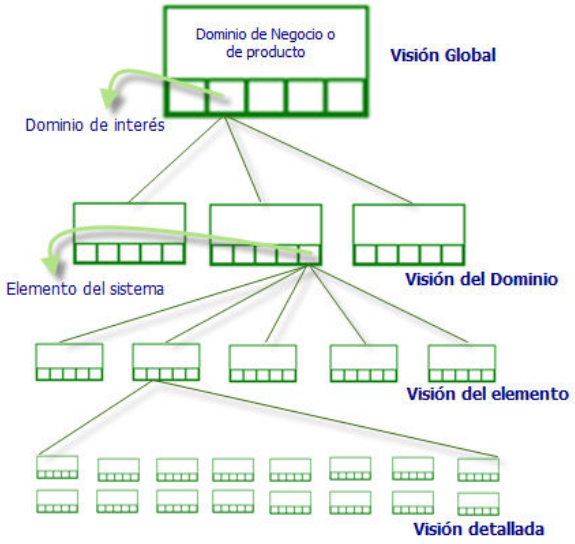


Figura 01 - Jerarquía de ingeniería de sistemas.

📖 **Actividad:**

Analice la figura anterior. Considere los siguientes aspectos:

- ¿Es posible determinar qué representan las subdivisiones de cada visión?
- ¿Qué tipo de ayuda le brinda este esquema?

Investigue sobre algún sistema ya desarrollado en su área de trabajo. Ubique cada uno de los estratos de esta jerarquía en el sistema escogido.

El **modelado de sistemas** permite a los ingenieros crear su visión del problema por resolver, de manera tal que:

- Logren definir los procesos que satisfacen las necesidades de la visión que se considera.
- Representen el comportamiento del proceso y sus supuestos.
- Definan de modo explícito las entradas exógenas y endógenas de información al modelo.
- Representen todas las uniones que permitan al ingeniero entender mejor la visión.

Además, se deben considerar algunas **restricciones** que se presentan en el modelado de sistemas, debido a que existe todo un ambiente muy complejo que modifica el estado de las cosas en cada momento para el modelado.

Se debe tener claro cuál es este estado actual, en el momento de iniciado el proceso, para contemplar tipos de restricciones como las siguientes:

Importante:

Tipos de **restricciones** para el modelado.

Supuestos: reducen el número de permutaciones y variaciones posibles, lo que permite al modelo reflejar el problema de una manera razonable.

Simplificaciones: permiten la creación del modelo a tiempo.

Limitaciones: ayudan a delimitar el sistema.

Restricciones: guían la manera de crear el modelo y tomar el enfoque al implementarlo.

Preferencias: indican la arquitectura preferida para todos los datos, funciones y tecnología.

Por otro lado, el modelado de sistemas y las herramientas de simulación, se utiliza ampliamente en los sistemas clasificados como **reactivos**, y que están basados en computadoras. Esto porque dichos sistemas normalmente son de tiempo real y su ejecución correcta puede salvar vidas o ahorrar grandes cantidades de dinero.

De ahí que la capacidad de simulación de la operación del sistema, antes de que el mismo se ponga a producción, es vital para el éxito integral del proyecto.

Ingeniería de proceso de negocios

El libro describe la ingeniería de procesos de negocios, como un enfoque dentro de la ingeniería de sistemas que crea un plan general para implementar la arquitectura de cómputo. Se divide en tres arquitecturas diferentes:

Importante:

Arquitecturas dentro de la ingeniería de proceso.

Arquitectura de datos: proporciona un marco de trabajo para las necesidades de información de un negocio o de una función de negocio.

Arquitectura de aplicaciones: abarca elementos de un sistema que transforma objetos dentro de la arquitectura de datos por algún propósito de negocio.

Arquitectura de la tecnología: proporciona el fundamento para las estructuras de datos y aplicación.

📖 **Actividad:**

Analice la figura que se encuentra en la página 141 del libro de texto, llamada *La jerarquía de la ingeniería de procesos de negocios*.

Considere la forma, la estructura, los niveles que se presentan y la relevancia de cada uno. Evalúe la jerarquía y dé su opinión al respecto.

Ingeniería de producto

Llevar una serie de capacidades definidas por el cliente a un programa desarrollado es la finalidad de la ingeniería de producto. Para lograrlo, debe tener una arquitectura y una estructura que la soporte.

La arquitectura mencionada abarca cuatro componentes, a saber: ***software***, ***hardware***, **datos** y **personas**.

Una visión general del dominio se consigue al determinar los requerimientos del cliente (ingeniería de requisitos). Esta ingeniería tiene como trabajo asignar función y comportamiento a cada uno de los componentes descritos anteriormente.

Terminada esta asignación, sigue la ingeniería de componentes, que reúne un conjunto de actividades para cada uno de los componentes de ingeniería de sistema: ingeniería de ***software***, de ***hardware***, de **datos** y la **humana**.

Modelado de sistema

Los modelos de sistemas tienden a ser jerárquicos por naturaleza. En su parte más alta se representa un modelo del sistema **completo**. Cuando se empieza a refinar aparece el nivel de detalle de los **componentes**, y culminan con la aparición de los modelos de **ingeniería** que son específicos para cada una de las disciplinas de ingeniería que está asociada.

Como los modelos más emblemáticos para el modelado de sistemas, se encuentra el modelado de sistemas de **Hatley-Pirbhai** y el modelado de sistemas con **UML**.

El libro de texto expone cada uno de estos modelos de manera amplia y trata de exponer la actividad y óptica del modelado de sistemas, a partir de estos dos métodos tan utilizados hoy día.

El modelo de Hatley-Pirbhai permite representar la **entrada**, el **procesamiento** y la **salida** junto con la **interfaz de usuario** y su correspondiente **mantenimiento**. Además, le suma una característica importante al modelado tradicional: el procesamiento de **autocomprobación**.

Por otro lado, en el modelado con **UML**, lo que se obtiene como diferenciador, dentro de los modelos actuales, es la gran cantidad de diagramas que pueden utilizarse para el análisis y diseño a nivel del *software* y del sistema.

Estos diagramas aumentan la óptica sobre el sistema y su contexto, ampliando la cantidad de vistas estáticas y dinámicas existentes sobre el mismo problema en común.

Dentro del desarrollo de los sistemas nos enfocamos en tres diferentes modelos: el **funcional**, el de **objetos** y el **dinámico**. Para el modelo funcional, el UML propone el diagrama de **casos de uso**, para el de objetos propone el diagrama de **clases**, y para el dinámico se propone los modelos de **secuencia**, **actividad** y de **estados**.

📖 **Actividad:**

El libro de texto, en las páginas 144 a 150, describe ampliamente ambos modelos y hace uso de un caso práctico para su explicación.

Lea y evalúe ambos modelos. Compare las ventajas ofrecidas para cada caso. Analice si ambos modelos sirven para los mismos tipos de problema que se puedan presentar a la hora de desarrollar una solución de *software*.

Ejercicios sugeridos

En la página 152 del libro de texto encontrará algunos problemas y puntos por considerar sobre el capítulo que acaba de estudiar.

Es importante que intente resolver los siguientes ejercicios. Esto le permitirá medir su aprendizaje.

23. Seleccione algún sistema con el que esté familiarizado y haga lo siguiente:
    * Defina el conjunto de dominios que definan la visión global del sistema.
    * Describa el conjunto de elementos que componen dos de los dominios del sistema.
    * Para un elemento, identifique los componentes técnicos por desarrollar.

24. Seleccione algún sistema grande con el cual esté familiarizado, y establezca suposiciones, simplificaciones, limitaciones, restricciones y preferencias que se deberían hacer para construir un modelado de sistemas eficaz.

25. La ingeniería de procesos del negocio requiere definir **datos**, **arquitectura de aplicaciones**, además de una **infraestructura de aplicaciones**. Describa cada uno de estos términos mediante un ejemplo.

26. Un ingeniero de sistemas puede recibir encargos de tres procedencias: el desarrollo de sistemas, el cliente o una organización externa. Discuta los **pros** y los **contras** de cada procedencia.

27. Describa al ingeniero de sistemas "ideal", considerando las lecturas, sobre la ingeniería de sistemas, realizadas hasta ahora. Qué características serían las más propicias para que su labor nunca sea tropiezo para el desarrollo de un sistema en particular.

28. Averigüe sobre el UML. A parte de los ya mencionados, ¿qué diagramas existen, y para qué se utilizan?

Capítulo

7

Ingeniería de requisitos

"…un cliente entra a su oficina y dice: Yo sé que usted piensa que entiende lo que digo, pero lo que usted no entiende es que lo que digo no es realmente lo que quiero decir…"

Sumario

Los temas que se tratan en este capítulo son:

- Tareas de la ingeniería de requisitos
- Inicio del proceso de la ingeniería de requisitos
- Obtención de requisitos
- Desarrollo de casos de uso
- Construcción del modelo de análisis
- Negociación y validación de requisitos

Localización de subtemas en el libro de texto

SUBTEMAS	PÁGINAS
Un puente hacia el diseño y la construcción	156
Tareas de la ingeniería de requisitos	157
Inicio del proceso de la ingeniería de requisitos	163
Obtención de requisitos	166
Desarrollo de casos de uso	173
Construcción del modelo de análisis	179
Negociación de requisitos	184
Validación de requisitos	186

Comentarios generales

La ingeniería de requisitos ocupa un espacio vital dentro de todo el proceso de desarrollo de *software*. Permite al ingeniero o grupo de ingenieros, mediante diferentes técnicas de obtención de requerimientos, alcanzar la primera visión del cliente acerca del producto por construir.

Aunque casi ningún requerimiento es perpetuo en el tiempo, estos plantean un escenario bastante complicado a los analistas y desarrolladores del producto, ya que estarán en un constante reproceso entre las etapas que soportan el desarrollo de *software*.

Debido a este detalle, el desarrollo normal de los productos de *software* adopta la **evolución iterativa** como el modelo de desarrollo más común en la actualidad.

Este capítulo dará una perspectiva más clara de cómo se debe tratar el tema del levantamiento y manejo de requisitos, dentro del proceso de desarrollo de sistemas. Con esto se trata de aclarar al lector la importancia de que el equipo de desarrollo haga un verdadero esfuerzo por entender los requisitos del problema **antes** de intentar resolverlo.

Desarrollo

La ingeniería de requisitos representa el primer esfuerzo real por tratar de alcanzar el sentimiento del cliente, con respecto a la necesidad que se quiere cubrir.

Esta disciplina proporciona a los ingenieros lo necesario para lograr **entender** qué es lo que realmente quiere, **analizar sus necesidades**, **evaluar la factibilidad** de lo que pide, **negociar una solución** razonable, **eliminar ambigüedades**, **validar la especificación** y **administrar los requerimientos** conforme cambian en el tiempo.

El proceso de la ingeniería de requisitos se compone de siete distintas funciones, a saber: **inicio**, **obtención**, **elaboración**, **negociación**, **especificación, validación** y **gestión**.

Inicio del proceso de la ingeniería de requisitos

Es importante entender que la localización del cliente no necesariamente estará siempre al alcance del encargado de obtener los requerimientos. La unidad didáctica sugiere que el primer enfoque por seguir es el de **entender la diversidad de localizaciones**, **opiniones** y **calidades** de los clientes, que se pueden dar al inicio de un proceso de levantamiento de requerimientos.

De esta situación se obtienen algunos pasos requeridos para atender esta realidad y sacar el mayor provecho, con la intención de crear un proyecto exitoso.

Se debe recordar que el objetivo principal es conseguir una **visión global** del problema que se intenta atacar. Esto se irá depurando a como se avanza en las distintas funciones de la ingeniería de requisitos. Observe:

Importante:

Pasos requeridos para **iniciar la ingeniería de requisitos**:

- **Identificación de los interesados**: todos aquellos que se benefician de forma directa o indirecta del sistema que está en desarrollo.
- **Reconocimiento de múltiples puntos de vista**: clientes diferentes, múltiples puntos de vista sobre los requerimientos.
- **Trabajo con respecto a la colaboración**: clientes colaborando entre sí. Buscar aéreas en común y áreas en desacuerdo para tomar acciones.
- **Formulación de las primeras preguntas**: deben ser de libre contexto, enfocadas en el cliente, metas generales y beneficio directo.

📖 **Actividad:**

En las páginas 165 y 166 del libro de texto, encontrará algunas preguntas de "libre contexto", asociadas con el tema del inicio para la ingeniería de requisitos.

Lea y estudie dichas preguntas. Analice si considera que estas preguntas son suficientes para obtener la información necesaria para continuar con el proceso de la ingeniería de requisitos. Explique su respuesta en forma detallada.

Obtención de requisitos

Hasta aquí, se tiene la visión genérica del problema visto desde algunos metros de altura. Ahora se profundizará en la obtención de los requerimientos pero de una manera más formal.

Se han propuesto muchos enfoques diferentes para la recopilación conjunta de requisitos; la mayoría concuerdan con las siguientes directrices básicas:

- Reunión dirigida por algún asistente, no debe quedar a la libre.
- Establecer reglas para la preparación y participación.
- Agenda formal, pero con libertad para el flujo de ideas.
- Siempre debe tener un moderador.
- Predefinir mecanismos para aterrizar lo encontrado.
- Obtener como meta la identificación del problema, la solución propuesta y garantizar el ambiente para que se cumpla.

📖 **Actividad:**

En las páginas 167 y 170 del libro de texto se presenta un flujo de eventos mediante un ejemplo puntual asociado a la lectura anterior de las directrices.

Lea y estudie el caso con el fin de entender para qué existen dichas directrices y la importancia de que se cumplan. Justifique sus ideas.

El **despliegue de la función de calidad** (QFD por sus siglas en inglés) se maneja para cada una de las etapas del proceso de ingeniería. En particular para la obtención de requisitos, esta técnica se presenta con el objeto de traducir las necesidades del cliente en requisitos técnicos del *software*.

Para lograrlo, el QFD resalta una comprensión de qué es valioso y lo acomoda en tres tipos de requisitos:

Importante:

Tipos de requisitos para la **QFD**:

- **Normales**: obligatorios para el cliente, responden a los objetivos y metas.

- **Esperados**: están implícitos en el producto o sistema. No se piden de manera explícita, tan solo se esperan como obvios.
- **Estimulantes**: características que van más allá de lo esperado por el cliente.

En esta etapa de obtención de requisitos, se obtiene como producto de trabajo en la mayoría de sistemas lo siguiente:

- Un enunciado de necesidad y factibilidad.
- Un enunciado limitado del ámbito del sistema o producto.
- Una lista de clientes, usuarios y otros interesados que participaron en la obtención de requisitos.
- Una descripción del ambiente técnico del sistema.
- Una lista de requisitos ordenados por función, y las restricciones de dominio aplicables para cada uno.
- Un conjunto de escenarios de uso para discernir la utilización del sistema en diferentes condiciones de operación.
- Algún prototipo desarrollado que ejemplifique mejor los requerimientos.

Desarrollo de casos de uso

En esencia, un caso de uso (**CU**) cuenta una "historia estilizada de la manera en que un usuario final interactúa con el sistema en un conjunto específico de circunstancias". La idea de este desarrollo es mostrar siempre el sistema desde el punto de vista del **usuario final**.

El CU contiene **actores**, que son las diferentes personas (o dispositivos) que utilizan el sistema dentro del contexto de la función y el comportamiento que se describirá.

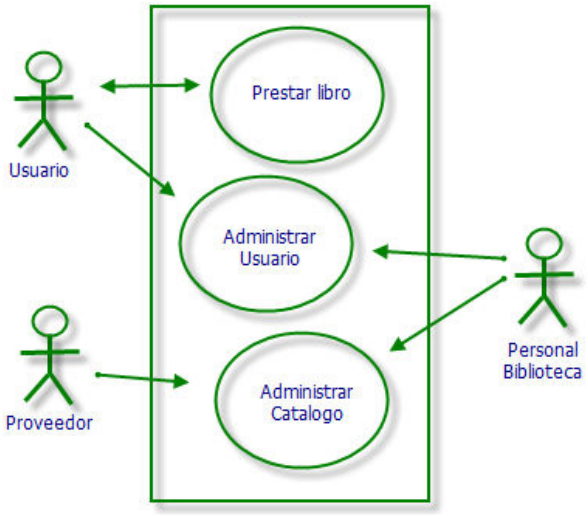


Figura 02 – Ejemplo de diagrama de CU para una biblioteca.

En los CU que se desarrollan están claros los actores del futuro sistema. Una vez identificados, se puede llevar a cabo una serie de preguntas que tienen que ser contestadas por el CU para darlo como satisfactorio.

- ¿Quién(es) es (son) el(los) actor(es) primario(s)?
- ¿Cuáles son las metas de cada actor?
- ¿Cuáles son las condiciones previas que deben existir antes de comenzar la historia?
- ¿Cuáles son las tareas o funciones principales que realiza cada actor?
- ¿Cuáles excepciones podrían considerarse mientras se describe la historia?
- ¿Cuáles son las variaciones posibles en la interacción de cada autor?
- ¿Cuál es la información del sistema que cada actor adquirirá, producirá o cambiará?
- ¿Cada actor tendrá que informar al sistema acerca de cambios en el ambiente externo?
- ¿Cuál es la información que cada actor desea del sistema?
- ¿Cada actor quiere ser informado de cambios inesperados?

📖 **Actividad:**

En las páginas 175 y 178 del libro de texto, se presentan varios ejemplos de cómo documentar los CU.

Lea y estudie cada caso con el fin de entender cómo se describen los CU, tanto de manera en prosa y cómo mediante el uso de plantilla.

El **modelo de análisis** es una representación de los requisitos en un momento determinado, por lo tanto es normal que algunos de éstos se modifiquen en el siguiente momento de evaluación.

El objetivo del modelo es describir los dominios requeridos de información, funcionamiento y comportamiento para un sistema basado en computadoras. En el capítulo siguiente se hablará ampliamente sobre este tema.

> 🕮 **Actividad:**
>
> En las páginas 179 a 184 del libro de texto se describe la construcción del modelo de análisis y sus elementos. Lea y estudie el enfoque dado desde la ingeniería de requerimientos.
>
> Elabore un resumen, para que pueda compararlo con el enfoque que encontrará en el siguiente capítulo del libro de texto.

Negociación y validación de requisitos

En su deseo por lograr todo lo que se propone, el ser humano necesita aterrizar sus ideas y defenderlas a capa y espada ante los demás, siempre con el afán de quedar ganancioso en la toma de decisiones acordada por todos los participantes.

Esta etapa es la que se conoce como **negociación** dentro del marco de ingeniería de requisitos, y busca encontrar un balance de la funcionalidad, el rendimiento y otras características del producto frente al costo y el tiempo de colocación en el mercado.

Por otro lado, una vez alcanzados los acuerdos correspondientes a lo que se va a hacer y a lo que no se va a hacer, se comienza la etapa de **validación** de requerimientos. En esta etapa, se examina cada elemento del modelo de análisis para conocer su **consistencia**, sus **omisiones** y sus **ambigüedades**. El cliente les otorga una jerarquía y se agrupan en paquetes que se implementarán como incrementos de *software*.

Se tienen identificadas preguntas para validar que el modelo de requisitos es un reflejo exacto de las necesidades del cliente, y que se está preparado para pasar a la etapa de diseño.

> 🕮 **Actividad:**
>
> En la página 186 del libro de texto se enlistan algunas de las preguntas que validan el modelo de requisitos.
>
> Lea cada una de las preguntas. Analice las posibles respuestas y determine cuáles otras deberían agregarse para fortalecer la sensación de entrega de un producto robusto como documento de requisitos.

Ejercicios sugeridos

En la página 188 del libro de texto encontrará algunos problemas y puntos por considerar sobre el capítulo que acaba de estudiar.

Es importante que intente resolver los siguientes ejercicios. Esto le permitirá medir su aprendizaje.

29. ¿Por qué varios desarrolladores de *software* no prestan mucha atención a la ingeniería de **requisitos**?

30. ¿Qué implica el "**análisis de factibilidad**" cuando se examina dentro del contexto de la función inicio?

31. A usted se le ha dado la responsabilidad de obtener requisitos de un cliente que dice estar demasiado ocupado para reunirse con usted. ¿Qué cosas debería hacer?

32. Exponga algunos de los problemas que pueden surgir cuando los requisitos deben obtenerse de tres o cuatro personas diferentes.

33. ¿Por qué se dice que el **modelo de análisis** representa una foto instantánea de un sistema en el tiempo?

34. Desarrolle al menos tres preguntas de "**contexto libre**" adicionales que pueda hacerle a algún interesado durante la fase de inicio.

35. Desarrolle un **CU** (utilizando la plantilla) para los siguientes problemas:
    * Busque libros para un tema específico en una librería en línea
    * Pague el recibo telefónico, desde el servicio que le presta su entidad bancaria en línea.

Capítulo

8

Modelado de análisis

"…¿Por qué debemos construir modelos? ¿Porqué no construimos el sistema y ya? La respuesta es que podemos construir modelos de tal forma que resaltemos o enfaticemos ciertas características críticas del sistema, al mismo tiempo que quitamos énfasis a otros aspectos del sistema…"

Ed Yourdon

Sumario

Los temas que se tratan en este capítulo son:

- Análisis de requisitos
- Enfoques y conceptos del modelado del análisis
- Análisis orientado a objetos
- Modelado basado en escenarios, orientado al flujo y basado en clases
- Creación de un modelo de comportamiento

Localización de subtemas en el libro de texto

SUBTEMAS	PÁGINAS
Análisis de requisitos	192
Enfoques del modelado del análisis	196
Conceptos del modelado del análisis	197
Análisis orientado a objetos	201
Modelado basado en escenarios	202
Modelado orientado al flujo	211
Modelado basado en clases	219
Creación de un modelo de comportamiento	235

Comentarios generales

La ingeniería de sistemas persigue, mediante el modelado del análisis, encontrar diferentes vías que permitan a las partes involucradas, obtener una visión tan clara como sea posible (sin importar cuantas diferentes vistas se ocupen) de las dimensiones en tamaño y alcance del sistema que se desea construir. Esto permite también determinar qué requerimientos deberán ser atendidos.

Aspectos asociados a la información contenida, a las funciones por realizar y al comportamiento de los diferentes objetos que componen el sistema, deberán atenderse de manera integral, para garantizar con esto que la solución alcanzará con éxito la utilización y consumo de parte del usuario final.

Este capítulo describirá y analizará dichos modelos; además, intentará dar una visión clara del orden de actividades y de los posibles insumos necesarios para llevar a cabo un buen modelado del análisis, lo suficientemente robusto como para soportar las etapas venideras y obtener un producto tangible, como lo sería el sistema desarrollado ya puesto en producción.

Desarrollo

Como se estudió en el capítulo anterior, la ingeniería de requisitos produce un documento entregable, que contiene todos los aspectos operacionales esperados en la creación del nuevo producto de *software*. Con este documento se inicia el **análisis de requisitos**, mediante el cual se indican las características operacionales, restricciones y una posible interfaz sugerida del *software*, con otros elementos del sistema.

El análisis de requisitos le proporciona al diseñador una representación de la información, la función y el comportamiento que logrará trasladar a diseños arquitectónicos, de interfaz y en el nivel de componentes.

El **modelo de análisis** junto a la especificación de requisitos, le permiten también tanto al cliente como al desarrollador, saber cómo evaluar la calidad una vez construido el *software*.

El modelo de análisis busca cumplir tres grandes objetivos:

1) Describir qué requiere el cliente.

2) Establecer una base para la creación de un diseño de *software*.

3) Definir un conjunto de requisitos que puedan validarse una vez construido el producto.

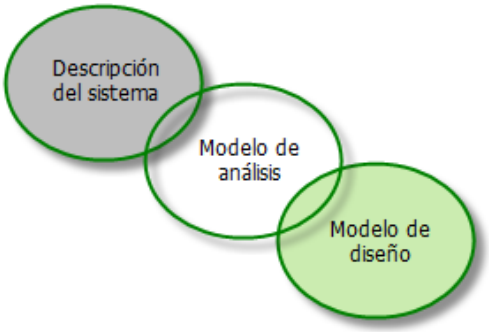


Figura 03 – Modelo de análisis como puente entre la descripción del sistema y el modelo de diseño.

Este modelo llena el vacío entre la descripción dada a **nivel de sistema** que detalla la funcionalidad total del sistema y del **diseño de *software*** (figura anterior), que detalla la arquitectura de la aplicación, la interfaz y la estructura a nivel de componentes.

Algunas reglas prácticas que sugiere el libro de Pressman para la creación de un modelo de análisis, son las siguientes:

Importante:

Reglas para la creación de **un modelo de análisis**:

- Debe centrarse en los requisitos visibles dentro del problema o dominio del negocio.

- Cada elemento del modelo debe agregarse a un acuerdo general de los requisitos de *software* y proporcionar una visión interna del dominio de información, función y comportamiento del sistema.
- Debe retrasarse la consideración de la infraestructura y otros modelos no funcionales hasta el diseño.
- Se debe minimizar el acoplamiento de todo el sistema.
- Se debe tener la seguridad de que el modelo de análisis proporciona valor a todos los interesados.
- El modelo debe mantenerse tan simple como pueda ser posible.

📖 **Actividad:**

En la página 195 del libro de texto, se encuentran detalladas las reglas descritas anteriormente.

Léalas y analícelas dentro del contexto del modelado del análisis. Indique si se podría ampliar la lista; de ser afirmativa su respuesta, hágalo.

Para el modelado de análisis hay dos enfoques claramente definidos: el enfoque llamado **análisis estructurado** y el análisis **orientado a objetos**.

El **análisis estructurado** busca que se reconozca la separación entre los datos y su proceso de transformación. Este enfoque sugiere que el proceso que transforma los datos y los datos en sí, deben ser entidades separadas.

Por otro lado, el enfoque **orientado a objetos** se centra en la creación de clases y la colaboración existente entre ellas para obtener los requerimientos del cliente.

La pregunta es ¿cuál es mejor?, bueno la respuesta no es sencilla, depende de la capacidad de la persona para escoger, y de ver qué **combinación de representaciones de ambos enfoques** le permitirá alcanzar el mejor modelo de requisitos de *software* y la forma más rápida de pasar al diseño.

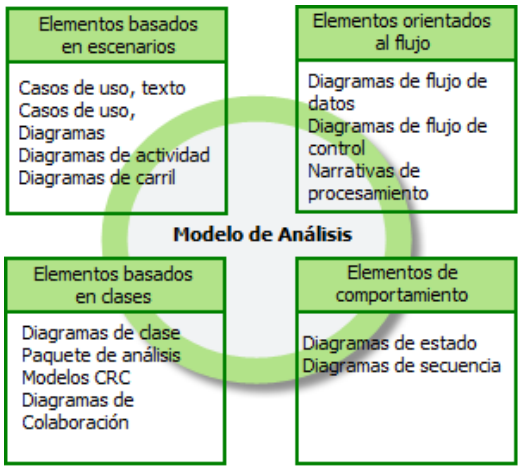


Figura 04 – Elementos del modelo de análisis.

El **modelado de los datos** es la primera actividad que se lleva a cabo en el modelado del análisis; para llevarlo a cabo se definen todos los objetos de datos que se procesan dentro del sistema y las relaciones entre estos. Se definirán algunos conceptos dentro del modelado de datos.

Importante:

Conceptos dentro del **modelado de datos**:

- **Objeto de datos**: se define como una representación casi de cualquier "información compuesta" que el *software* deba entender. Esta información compuesta se refiere a algo que tiene muchas propiedades o atributos diferentes.
- **Atributos**: definen las propiedades de un objeto de datos, y toman una de tres características diferentes.

  1) Nombrar una ocurrencia del objeto.

  2) Describir la ocurrencia.

  3) Hacer referencia a otra ocurrencia en otra tabla.
- **Relaciones**: son las conexiones de objetos entre sí y se determinan entendiendo el papel de un objeto con otro dentro del contexto de *software* que se va a construir.
- **Cardinalidad**: el modelo de datos debe ser capaz de representar el número de ocurrencias de los objetos en una relación dada.
- **Modalidad**: define la obligatoriedad entre los objetos de que se presente una ocurrencia.

📖 **Actividad:**

En las páginas 199 y 200 del libro de texto se encuentran más detallados estos conceptos de modelado de datos.

Léalos y analícelos dentro del contexto del modelado del análisis; esto le permitirá ampliar su visión del tema.

Análisis orientado a objetos

El **análisis orientado a objetos** (AOO) consiste en la definición de todas las clases, relaciones y comportamientos asociados a las clases, que son relevantes y que necesitan resolverse para obtener solución al problema planteado. Se deber ejecutar algunas tareas para lograr el AOO:

Importante:

Tareas para lograr el **AOO**:

- Comunicar los requisitos básicos del usuario entre el cliente y el ingeniero de *software*.
- Identificar las clases con sus métodos y atributos.
- Definir la jerarquía de estas clases.
- Representar las relaciones de objeto a objeto.
- Modelar el comportamiento del objeto.

- Todos los pasos anteriores se ejecutan de manera iterativa hasta que el modelo esté completo.

Modelado basado en escenarios

Para obtener la aprobación del cliente es necesario presentarle una solución que cubra todas las carencias de información y que la manera de interactuar y acceder a ellas sea lo más sencillo posible.

El modelo basado en escenarios hace uso de los **casos de uso** y de los **diagramas de actividad** y **carril** para lograr construir modelos significativos que permitan una solución como la descrita anteriormente.

Tal y como se mencionó en el capítulo anterior de esta guía, la idea de un caso de uso es mostrar siempre el sistema desde el punto de vista del usuario final.

Las dos primeras tareas de la ingeniería de requisitos: **inicio** y **obtención**, facilitan la información necesaria para comenzar a escribir los casos de uso. Estos se inician realizando una serie de funciones o actividades que realiza un actor específico.

🕮 **Actividad:**

En las páginas 205 a 208 del libro de texto encontrará un caso de uso como ejemplo. El mismo se lleva desde la simple descripción relatada, pasando por una representación del mismo caso pero como enunciado declarativo y luego llevándolo a modo de plantilla formal. Al final cierra con la figura 8.6, que describe un caso de uso, pero de forma gráfica.

Lea estas representaciones de caso de uso, analícelas y establezca comparaciones para determinar cómo explicaría, con sus propias palabras, un caso de uso.

Los **diagramas de actividad** contribuyen al modelado de análisis al apoyar a los casos de uso mediante una representación gráfica del flujo de interacción dentro de un escenario específico.

Este diagrama, similar a los famosos **diagramas de flujo de datos** (**DFD**) en el análisis estructurado, permite representar funciones, flujo de datos, rombos de decisión y líneas paralelas para determinar actividades paralelas.

Con esta dinámica es fácil entender que el diagrama de actividad proporciona oportunidades que no tendríamos solo con los casos de uso, como son las opciones de decisiones controladas.

Los **diagramas de carril** representan una variante del diagrama de actividad. Permite al modelador representar el flujo de actividades descritas por el caso de uso, y al mismo tiempo, indicar qué **actor** o **clase** de análisis tiene la responsabilidad de la acción descrita, mediante un rectángulo de actividad.

🕮 **Actividad:**

Continúe la actividad anterior. En las páginas 209 y 210 del libro de texto encontrará las figuras 8.7 y 8.8, con ejemplos de un diagrama de actividad y un diagrama de carril respectivamente. Los ejemplos corresponden al sistema de seguridad **Hogar seguro**.

Estudie estas figuras, con el fin de entender de manera visual los conceptos sobre diagrama de actividad y de carril.

Modelado orientado al flujo

El modelo de flujo de datos, mejor conocido como DFD, se construye mediante flechas rotuladas que representan a los **objetos de datos**, y mediante círculos que representan las **transformaciones** de estos datos.

Este modelado se presenta como un diagrama jerárquico, donde el primer nivel, llamado **nivel 0** o de **contexto**, contiene la totalidad del sistema. A partir de ahí, se inicia un proceso de refinamiento, que se desgrana cada vez más por cada nivel que baje en la jerarquía, hasta encontrar el nivel en que un proceso representa una función única. Existen algunas pautas para la creación de un diagrama de flujo de datos:

Importante:

Pautas para la creación de un **DFD**:

1. El nivel 0 del DFD debe representar al sistema como una sola burbuja.
2. La entrada y la salida primaria deben establecerse con cuidado.
3. La refinación debe comenzar por el aislamiento de procesos, objeto de datos y almacenamiento de datos candidatos a ser representados en el siguiente nivel.
4. Todas las flechas y burbujas se deben rotular con nombres significativos.
5. Se debe mantener la continuidad del flujo de información al cambiar de nivel a nivel.
6. La refinación de las burbujas debe hacerse una por una.

Es importante indicar que la generación de un DFD puede seguir un enfoque sumamente simple para su implementación. Se puede lograr por medio de un "análisis gramatical" de los sustantivos y verbos que se desprenden de las primeras reuniones en el levantamiento de requerimientos.

Los verbos encontrados en la narrativa inicial serán los posibles **procesos** por plasmar en el DFD. Por otro lado, los sustantivos encontrados se verán como las **entidades** externas, **objetos de datos** o **control,** o **almacenamientos de datos** existentes.

A pesar que en la mayoría de las aplicaciones el modelo de datos del DFD son suficientes para obtener la visión del sistema por construir, existen casos en donde no basta dicha postura, y se requiere, además, un **modelo de control del flujo**.

Este modelo de control del flujo se presenta principalmente en las aplicaciones que están guiadas en eventos y no por datos, y que producen información de control en lugar de reportes de datos.

La **especificación de control** (EC), por otro lado, representa el comportamiento del sistema en un momento del tiempo dado. Contiene un **diagrama de estado** que no es más que una representación secuencial del comportamiento del sistema.

Sumado a la EC, tenemos a la **especificación de procesos** (EP), que se utiliza para describir todos los procesos del modelo de flujo que aparecen en el nivel final de cada refinación. Cada una de estas "mini especificaciones" sirve como guía para el futuro diseño de componentes de la aplicación.

> 🕮 **Actividad:**
>
> Cada una de las anteriores especificaciones asociadas al DFD, se encuentra detallada en las páginas 215 a 218 del libro de texto. Para ello se utiliza el ejemplo sobre el sistema de seguridad **Hogar seguro**.
>
> Lea y analice estas descripciones, con el fin de entender mejor el entorno que acompaña a los DFD.

Modelado basado en clases

Como se menciona en párrafos anteriores, al realizar un "análisis gramatical" de la narrativa inicial, se pueden obtener los sustantivos y verbos que se desprenden del problema por resolver.

Los sustantivos encontrados se verán como: **entidades externas**, **cosas**, **sucesos** o **eventos**, **papeles o roles**, **unidades organizacionales**, **sitios** o **estructuras,** dentro del ámbito del problema. Cada uno de estos sustantivos se verá como una **potencial clase** y se le debe dar un tratamiento en este sentido.

El libro de texto sugiere seis características que debe llevar una clase potencial, para ser considerada en el modelo de análisis.

> *Importante:*
>
> Características para una **clase potencial** y su vinculación al **modelo de análisis**:
>
> 1. **Información referida**: información de la clase; es vital para el funcionamiento del sistema.
> 2. **Servicios requeridos**: conjunto de operaciones que permiten cambios en los atributos de la clase.
> 3. **Atributos múltiples**: deben ser varios los atributos que describan la clase. Una clase con solo un atributo, posiblemente se vea mejor como atributo en otra tabla.
> 4. **Atributos comunes**: se dan para todos los casos y se utilizarán en todas las instancias de la clase.
> 5. **Operaciones comunes**: se dan para todos los casos y se utilizarán en todas las instancias de la clase.
> 6. **Requisitos esenciales**: las entidades externas se van a crear como clases en el modelo de requisitos porque aparecen en el espacio del problema.

El incluir una clase en el modelo de requisitos, significa que la misma cumple con todas o casi todas estas características.

Los **atributos** a los que nos referimos en estas características deseables son los que permiten definir la clase y depurar qué significa en el contexto del espacio del problema.

Las **operaciones** son las acciones que definen el comportamiento de un objeto. Estas se dividen en cuatro categorías de operaciones:

1. Las que manipulan datos
2. Las que realizan algún cómputo
3. Las que preguntan acerca del estado de un objeto
4. Las que lo monitorean para la ocurrencia de un objeto de control

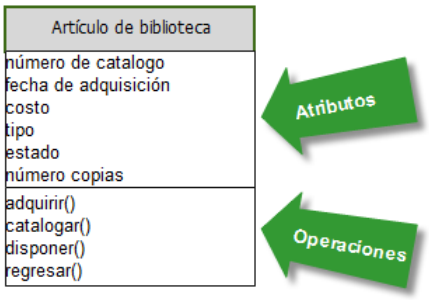


Figura 05 – Ejemplo de una clase.

El modelo de **clase–responsabililidad–colaborador**, conocido como el **CRC**, permite de manera simple identificar y organizar las clases relevantes para los requisitos encontrados.

En este modelo, las **responsabilidades** son los atributos y las operaciones relevantes para la clase. Los **colaboradores** son las clases que se requieren para que una clase reciba la información necesaria para completar una responsabilidad. Para este modelo, tenemos una extensión del concepto de clases descrito anteriormente, que se dividen en las siguientes categorías:

Importante:

Categorías de **clases** para el modelo **CRC**:

1. **Clases entidad**: llamadas también **clases modelo de negocios**. Se extraen de manera directa del enunciado del problema.
2. **Clases frontera**: crean la interfaz que el usuario ve y con la cual interactúa.
3. **Clases controlador**: manejan una "unidad de trabajo" desde el inicio hasta el final. Controlan la interacción y el mantenimiento entre los objetos.

Con respecto a las responsabilidades, el libro de texto sugiere cinco directrices para determinar la responsabilidad de las clases, a saber:

Importante:

Directrices para las **responsabilidades** de las clases en el modelo **CRC**:

1. La inteligencia del sistema se debe distribuir entre las clases para abordar, de mejor manera, las necesidades de las clases.
2. Cada responsabilidad debe establecerse tan general como sea posible.
3. La información y el comportamiento relacionado con ella deben estar dentro de la misma clase.
4. La información relativa a una cosa debe localizarse con una sola clase, no distribuirse entre muchas de ellas.
5. Las responsabilidades deben compartirse entre clases relacionadas cuando esto es apropiado.

Para abordar el tema de las colaboraciones, se puede decir que las clases le hacen frente a sus responsabilidades en una de dos formas:

1) Una clase puede utilizar sus propias operaciones para manipular sus propios atributos.

2) Una clase puede colaborar con otras clases.

Las colaboraciones se producen cuando una clase no puede cumplir una responsabilidad por sí sola, y ocupa la interacción con otra.

📖 **Actividad:**

En las páginas 225 a 231 del libro de texto, se detalla el modelo CRC. Para hacerlo se hace uso del caso del sistema de seguridad **Hogar seguro**.

Lea y analice estas descripciones, con el fin de entender mejor el modelo y su alcance, en colaboración con el modelado de análisis.

Las clases de análisis tienden a estar relacionadas de alguna manera, con mayor o menor nivel de cohesión. Estas relaciones, en UML (*Unified Modeling Language*, por sus siglas en inglés, **Lenguaje Unificado de Modelado**) son conocidas como **asociaciones**.

Estas asociaciones se pueden detallar más cuando se incluye el atributo de **multiplicidad** para definir la cantidad de posibles ocurrencias entre ellas.

Por otro lado, cuando una clase depende de alguna manera de otra, dándose una relación estilo cliente–servidor, se da una relación de **dependencia**, tal y como lo maneja el UML.

También es importante describir el concepto de categorización dentro del tema de las clases de análisis. Los elementos de análisis (CU, Casos de Uso, y clases de análisis) se clasifican y se empaquetan como una agrupación, llamada **paquete de análisis**, a la que se le da un nombre representativo.

Creación de un modelo de comportamiento

El **modelo de comportamiento** es un enfoque dinámico. La razón de este modelo es indicar la forma en que el *software* responderá a los eventos o estímulos externos. El siguiente cuadro resume los pasos necesarios para obtener un modelo de comportamiento correcto:

Importante:

Pasos para obtener un **modelo de comportamiento** dentro del modelado de análisis:

1. Evaluar todos los CU para entender por completo la secuencia de interacción dentro del sistema. Entender el evento que se presenta, más que la misma información transitada a través del CU.
2. Identificar los eventos del punto 1 que conducen la secuencia de interacción. Entender la forma en que estos eventos se relacionan con las clases específicas.
3. Crear una secuencia para cada CU. Esto se realiza mediante un diagrama de secuencia que indica cómo los eventos causan transiciones de objeto a objeto, como una función del tiempo.
4. Crear un diagrama de estado para todo el sistema. Esto se lleva a cabo mediante un diagrama de estado que representa los estados activos para cada clase y los eventos o disparadores que ocasionan cambios entre los estados activos.
5. Revisar el modelo de comportamiento para verificar su exactitud y consistencia.

📖 **Actividad:**

En las páginas 237 a 238 del libro de texto, se ofrece un diagrama de estado y otro de secuencia. Para hacerlo se hace uso del caso del sistema de seguridad **Hogar seguro**.

Observe y analice estos diagramas, con el fin de entender la nueva dimensión del mismo escenario, representado a través de estos 2 nuevos tipos de diagrama.

Ejercicios sugeridos

En la página 241 del libro de texto encontrará algunos problemas y puntos por considerar sobre el capítulo que acaba de estudiar.

Es importante que intente resolver los siguientes ejercicios. Esto le permitirá medir su aprendizaje.

36. ¿Es posible comenzar al proceso de codificación inmediatamente después de haber creado el modelo de análisis? Explique su respuesta y justifique los puntos en contra.

37. Un análisis práctico es aquel en el cual el modelo "debe enfocarse en los requisitos que son visibles dentro de los dominios del negocio". ¿Cuáles tipos de requisitos no son visibles en estos dominios? Proporcione ejemplos.

38. ¿Cuál es el propósito del análisis del dominio? ¿Cómo se relaciona con el concepto de los patrones de requisitos?

39. En unas cuantas líneas, describa las diferencias primordiales entre el análisis estructurado y el análisis orientado a objetos.

40. Suponga que le solicitaron construir uno de los siguientes sistemas:
    * Un sistema simple de facturación para un negocio pequeño.
    * Un sistema en línea para la compra de boletos a una empresa de cine.
    * Un sistema de escogencia de citas en línea, para el seguro social.

Dibuje un modelo a nivel de contexto (DFD de nivel 0) para alguno de estos 3 sistemas.
Escriba una **narrativa del procesamiento** para el sistema al nivel de contexto.
Luego, dibuje los DFD a los niveles 1 y 2, creando un "análisis gramatical" apoyados en la narrativa creada.
Rotule cada uno de los procesos y flujos de datos.

Referencias

Larman, C. (2004). UML y Patrones. Segunda edición. México D.F.: Prentice Hall.

McConnell, S. (1996). Desarrollo y gestión de proyectos informáticos. Segunda edición. México D.F.: Prentice Hall.

Glosario de términos

UML: (*Unified Modeling Language*, por sus siglas en inglés). **Lenguaje Unificado de Modelado** es el lenguaje de **modelado** de sistemas de *software* más conocido y utilizado en la actualidad; está respaldado por el **OMG** (*Object Management Group*). Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de *software*. UML ofrece un estándar para describir un "plano" del sistema (modelo), incluyendo aspectos conceptuales tales como procesos de negocios y funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes de *software* reutilizables.

Enlaces electrónicos relacionados con el tema

CAPÍTULO	CONCEPTO	PÁGINA
6	Página oficial UML	http://www.uml.org/
6	Tutorial UML	http://www.clikear.com/manuales/uml/index.aspx

Tema

3

Principios de diseño aplicado en la ingeniería de *software*

"Hay dos formas de construir un diseño de software*: una es hacerlo tan simple que obviamente no haya deficiencias, y la otra es hacerlo tan complicado que no haya deficiencias obvias…"*

C. A. R. Hoare

Objetivos

Los documentos del modelo del análisis permiten conocer qué es necesario hacer para satisfacer los requerimientos definidos por el cliente. El diseño viene a ser la representación gráfica que amarra las relaciones existentes entre el *software* y el *hardware*, las estructuras que soportarán la información y los datos y las interfases entre componentes que expondrán las opciones necesarias para que el usuario final realice su trabajo.

Cuando termine de estudiar este tema, usted estará en capacidad de:

- Entender los conceptos que acompañan el modelado del diseño para los sistemas de información, haciendo uso de los diferentes modelados que existen para este fin.

- Reconocer la necesidad de un diseño arquitectónico que lo soporte de manera robusta y sostenible a partir del modelado de análisis y diseño.

- Conocer y aplicar el concepto de componente y su aplicabilidad en el mundo del diseño arquitectónico.

Sumario

Los capítulos que conforman este tema se encuentran en el libro de texto y son los siguientes:

- **Capítulo 9 – Ingeniería del diseño**
- **Capítulo 10 – Diseño arquitectónico**
- **Capítulo 11 – Diseño al nivel de componentes**

Además, podrá consultar las siguientes secciones:

- **Referencias bibliográficas**
- **Glosario de términos**
- **Enlaces electrónicos relacionados con el tema**

Capítulo

9

Ingeniería del diseño

"…Puedes usar un borrador en la tabla de diseño o un martillo en el sitio de construcción…"

Frank Lloyd Wright

Sumario

Los temas que se tratan en este capítulo son:

- Diseño dentro del contexto de la ingeniería del *software*
- Proceso y calidad del diseño
- Conceptos de diseño
- El modelado del diseño
- Diseño de *software* basado en patrones

Localización de subtemas en el libro de texto

SUBTEMAS	PÁGINAS
Diseño dentro del contexto de la ingeniería del *software*	247
Proceso y calidad del diseño	249
Conceptos de diseño	252
El modelado del diseño	262
Diseño de *software* basado en patrones	269

Comentarios generales

En este capítulo se determinarán las bases y los conceptos necesarios para alcanzar un **diseño robusto** y basado en **patrones** que soporten su creación de manera exitosa y repetible en el tiempo.

Esto solo se logra teniendo muy bien definidas y aterrizadas las etapas que anteceden al diseño, como lo son la ingeniería de requisitos y el modelo del análisis basado en el documento de requerimientos resultante.

Este capítulo se centrará, tanto en los conceptos que permiten a los ejecutores de un proyecto de *software* desarrollar productos exitosos a partir de la aplicación de los fundamentos relacionados con la materia de diseño, como en las **técnicas del modelo** por sí mismo, que permite incorporar los patrones de diseño para obtener productos más completos.

Además, se procurará que el lector comprenda la transición sufridas por las clases definidas en el modelado del análisis, al pasar de un nivel más bajo de abstracción al modelo de diseño, y con esto llegar a presentar la visión del detalle para el cliente.

Desarrollo

Diseño dentro del contexto de la ingeniería del *software*

El diseño de *software* es, en definitiva, la última acción dentro de la ingeniería que corresponde a la actividad del modelado. Permite crear una plataforma para la construcción del código y la correspondiente etapa de pruebas.

Existe una relación directa entre los elementos del modelo de análisis descritos en el capítulo 8 y los modelos de diseño para crear una especificación completa. En la siguiente figura, se facilita la relación mental entre los diferentes grupos de elementos del modelado de análisis, contra las diferentes etapas del modelo de diseño de *software*.

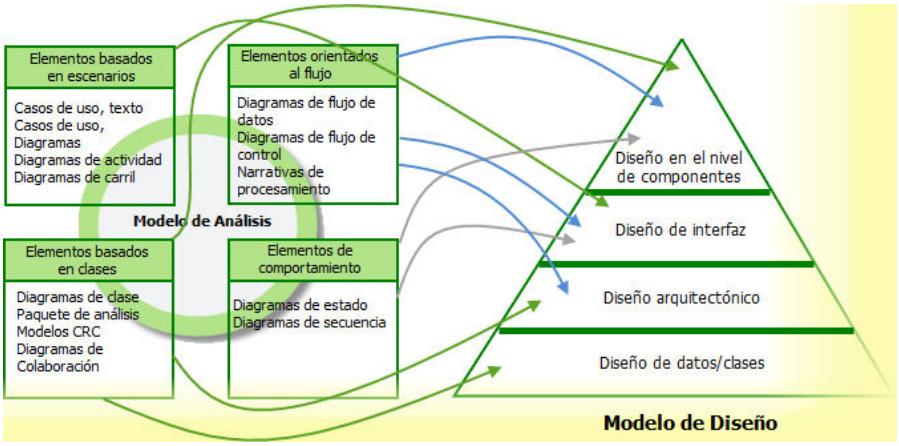


Figura 01 – Transformación del modelo de análisis al de diseño.

La siguiente es la descripción de los cuatro niveles dentro del modelado del diseño, descrito en la figura anterior:

> *Importante:*
>
> Cuatro **niveles** del **modelado del diseño**:
>
> **Diseño de datos/clase**: transforma los modelos de análisis y clases en las clases de diseño y las estructuras de datos que se requieren para la implementación del *software*.
>
> **Diseño arquitectónico**: define la relación entre los elementos estructurales más importantes del *software*.
>
> **Diseño de interfaz**: describe la forma en que el *software* se comunica con los sistemas que interactúan con él y con los seres humanos que lo utilizan.
>
> **Diseño a nivel de componentes**: transforma los elementos estructurales de la arquitectura de *software* en una descripción procedimental de los componentes de este.

Proceso y calidad del diseño

Conforme se refina el modelado del diseño, se alcanza un nivel de abstracción mucho más bajo que el encontrado originalmente con la primera iteración. Este nivel debe representar, muy de cerca, los requisitos dados por la ingeniería de requisitos, vista en capítulos anteriores. Algunas características que sirven como guía para la evaluación de un buen proceso de diseño son:

Importante:

Características para la evaluación de un buen **proceso** de diseño:

Debe implementar todos los **requisitos explícitos contenidos en el modelo de análisis**, y debe ajustarse a todos los **requisitos implícitos** que desea el cliente.

Debe ser una **guía legible y comprensible** para quienes generan código, quienes realizan pruebas, y dan soporte al *software*.

Debe proporcionar una **imagen completa** del *software* –dando dirección a los dominios de datos, funcionales y de comportamiento– desde una perspectiva de implementación.

Para alcanzar estas metas de diseño, se debe trabajar sobre aspectos relacionados con el tema de **calidad**. Es necesario establecer algunas **directrices** que deben existir para establecer criterios técnicos y con esto poder evaluar la calidad de un diseño.

🕮 **Actividad:**

Encontrará la explicación acerca de las directrices mencionadas anteriormente en las páginas 249 y 250 del libro de texto.

Léalas y analícelas con respecto a lo que usted esperaría de un modelado de diseño.

Atributos de calidad

Los **atributos** de calidad que se presentan a continuación, deben entenderse como aspectos del producto final que se esperaría existieran en él y que son considerados desde el inicio del proceso de diseño.

Claro está que la profundidad de cada uno de estos atributos va a depender del tipo de *software* a desarrollar. Por ejemplo, una entidad bancaria con transacciones por el web, esperaría ver el atributo de **confiabilidad**, como uno de los atributos de calidad más fuertes en su página.

Importante:

Atributos de **calidad en el diseño**:

Funcionalidad: se estima al evaluar el conjunto de características y capacidades del programa, la generalidad de las funciones que se entregan y la seguridad del sistema en su totalidad.

Facilidad de uso: se valora al considerar los factores humanos, la estética, consistencia y documentación general.

La confiabilidad: se avalúa al medir la frecuencia y severidad de las fallas, la precisión de los resultados de la salida, la media del momento de fallas, la habilidad para recuperarse de las fallas y la previsibilidad del programa.

El desempeño: se mide con la velocidad de procesamiento, tiempos de respuesta, consumo de recursos, rendimiento y eficacia.

La soportabilidad: combina la habilidad de extender el programa, la adaptabilidad y la serviciabilidad. Además, la resistencia a pruebas, compatibilidad, configurabilidad, facilidad con que puede instalarse, y la facilidad para localizar problemas.

Conceptos de diseño

Como en toda disciplina, el dominio de los fundamentos es lo que provoca la creación de un producto fuerte y cimentado. En la ingeniería de *software* en general y en el diseño de *software* en particular, no se da la excepción, por lo que se debe iniciar con el esfuerzo de entender los diferentes **conceptos** que deben estar presentes en el diseño.

Estos **conceptos** son los que permiten obtener modelos robustos en su composición, y por lo tanto permitirán hacer que un programa funcione y a la vez lo haga del modo correcto.

Importante:

Conceptos de diseño:

Abstracción: para efectos del diseño, el nivel de abstracción en una solución se establece en términos generales con el lenguaje del entorno del problema y en grados de menor abstracción, se proporciona una descripción más detallada de la solución. En la medida en que cambian los diferentes grados de abstracción se trabaja para crear abstracciones **procedimentales** y de **datos**.

Arquitectura: alude a la **estructura general del *software*** (organización de componentes) y las formas en que la estructura proporciona una integridad conceptual para un sistema.

Patrones: describen una **estructura de diseño** que resuelve **un problema de diseño en particular**, dentro de un contexto específico y en medio de fuerzas que pueden tener un impacto en la manera en que se aplica y utiliza dicho patrón.

Modularidad: el *software* se divide en **componentes** con nombres independientes y que es posible abordar de manera individual. Los patrones de arquitectura en el diseño materializan la modularidad.

Ocultación de información: los módulos deben especificarse y diseñarse de manera que la información que está dentro del modulo **sea inaccesible** para otros módulos que no necesiten de esa información.

Independencia funcional: se consigue al desarrollar módulos con una **función determinante y una aversión** a la interacción excesiva con los demás módulos.

Refinamiento: se presenta al **refinar de manera sucesiva** los niveles de detalle procedimentales. Una jerarquía se desarrolla al **descomponer el enunciado macro** de una función paso a paso, hasta alcanzar el nivel de código.

Refabricación: técnica de **reorganización** que simplifica el diseño (o código) de un componente sin cambiar su función o comportamiento.

Clases de diseño: permiten refinar las clases de análisis al proporcionar detalles al diseño que admitan la implementación de las clases y provoca que se produzca un conjunto nuevo de clases de diseño que implementen una infraestructura de *software* para soportar la solución del negocio. Se sugieren cinco tipos de clases: **de interfaz con el usuario**, **del dominio de negocios**, **de proceso**, **persistentes** y **de sistema**.

Por otro lado, una clase de diseño bien conformada, debe tener presentes estas cuatro características: **ser completa y suficiente**, **primitivismo**, **cohesión alta** y **bajo acoplamiento**.

> 🕮 **Actividad:**
>
> En las páginas 252 y 261 del libro de texto se describen con más detalle cada uno de estos conceptos. Léalos y analícelos para una mejor comprensión de los fundamentos del diseño.

El modelo de diseño

Para mostrar de una manera más gráfica el tema del modelo de diseño, se va a ejemplificar mediante un gráfico con dos dimensiones, sean: el **proceso** y la **abstracción**.

El **proceso** indica la evolución del modelo de diseño conforme se ejecutan las tareas de diseño.

La **abstracción** representa el grado de detalle a medida que cada elemento del modelo de análisis se transforma en su equivalente de diseño y después se refina de una manera iterativa.

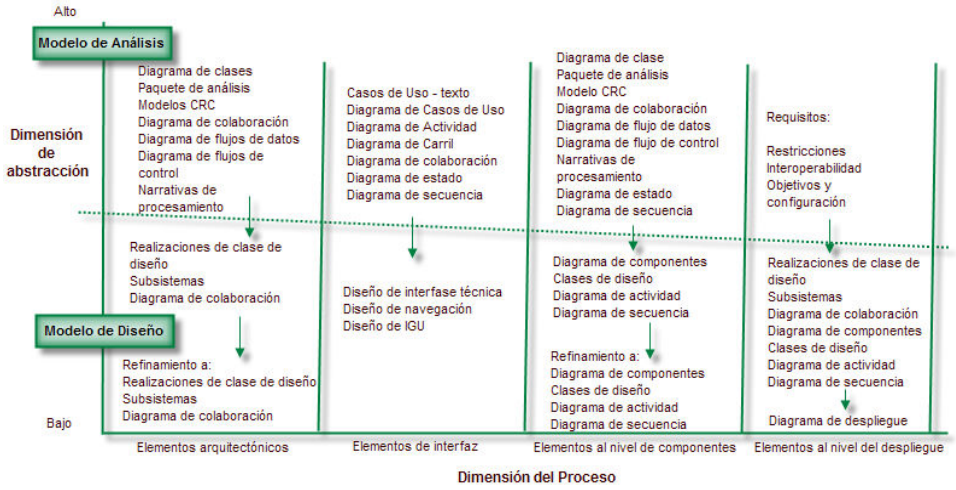


Figura 02 – Dimensiones del modelo de análisis.

Los **elementos que integran el diseño de los datos**, o la arquitectura de los datos, van a tener una profunda influencia sobre la arquitectura final del *software* a desarrollar, desde su presentación inicial que es bastante abstracta (dada por la ingeniería de requisitos), hasta llegar de manera progresiva a los niveles más bajos, donde se llegará a la implementación específica de cada aspecto.

En el modelo de diseño también se encuentran los **elementos del diseño arquitectónico**, que dan una visión global, a cierta altura, de lo que se va a desarrollar, tal como lo haría un plano para una casa antes de construir.

Además, están los **elementos del diseño de interfaz**, que determinan la manera en que se interactúa con la aplicación, y la dirección y forma en que se llevarán a cabo las acciones. En esta parte del diseño, se indica cómo fluye la información hacia afuera del sistema y como éste está comunicado entre los componentes definidos como parte de la arquitectura.

El diseño a **nivel de componentes** forma parte de la columna vertebral de un buen diseño de *software*. Para efectos del desarrollo de una solución, el diseño a nivel de componentes describe por completo cada detalle interno de cada componente de *software*.

Esta descripción se logra con la definición de estructuras de datos (mencionadas en párrafos anteriores) para todos los objetos de datos locales, con las reglas de negocio aplicadas mediante la secuencia de pasos para aplicar la lógica correspondiente y con una interfaz que permita el acceso a todas las operaciones de los componentes.

Por último, el **diseño a nivel del despliegue** permite conocer cómo se ubicarán la funcionalidad y los subsistemas dentro del entorno computacional físico que soportarán al *software*.

Diseño de *software* basado en patrones

La tendencia en las ingenierías modernas es muy clara: identificar patrones que den soluciones medibles para resolver problemas. Para el diseño de *software*, esto se aplica a la perfección.

Los patrones de diseño se pueden describir utilizando una **plantilla**, que contiene todos los aspectos por considerar para que su uso sea provechoso. Un patrón de diseño puede considerar también un conjunto de **fuerzas de diseño**, que son las que describen requisitos no funcionales asociados con el *software* por crear y que a la vez permiten definir las limitaciones que restringen la manera en que se implementará el diseño.

Los patrones de diseño necesitan la etapa de modelado de análisis completada para poder utilizarse. El analista observa el entorno del problema y las limitantes encontradas en la realidad del proyecto y evalúa si se puede o no aplicar algún patrón existente. Dicho patrón existente se puede aplicar para alguno de los siguientes tipos de patrones: **arquitectónicos**, **de diseño** y **de idiomas**.

Es importante mencionar la existencia del concepto: **marcos de trabajo**. Son una "mini arquitectura reutilizable" que ofrece un conjunto de comportamientos y de estructuras genéricas que están bien identificadas para una familia de abstracciones de *software* en un dominio dado.

El marco de trabajo contiene una serie de "puntos de conexión" dentro de un "esqueleto", que le permiten adaptarse a un dominio de un problema específico. En términos de orientación a objetos, un marco de trabajo es una **colección de clases que cooperan**.

Ejercicios sugeridos

En la página 273 del libro de texto encontrará algunos problemas y puntos por considerar sobre el capítulo que acaba de estudiar.

Es importante que intente resolver los siguientes ejercicios. Esto le permitirá medir su aprendizaje.

41. ¿Cómo se relacionan los conceptos de **acoplamiento** y **portabilidad** de *software*? Proporcione ejemplos claros.

42. ¿Se debe implementar un **diseño modular** como *software* monolítico? ¿Cómo se puede lograr esto? ¿El **desempeño** es la única justificación para la implementación del *software* monolítico?

43. Explique la relación entre el concepto de **ocultación** de información como un atributo de **modularidad** efectiva y el concepto de **independencia de módulo**.

44. Describa, con argumentos propios, la **arquitectura** de *software*.

45. ¿Se diseña un *software* cuando se "escribe" un programa? ¿Qué es lo que hace que el **diseño** de *software* sea diferente a la **generación** de código?

46. ¿Cómo se evalúa la **calidad** de un diseño de *software*?

Capítulo

10

Diseño arquitectónico

"La estructura de un sistema de software proporciona la ecología en que nace, madura y muere el código. Un habitad bien diseñado permite el éxito en la evolución de todos los componentes de un sistema de software." ***R. Pattis***

Sumario

Los temas que estudiará en este capítulo son:

- Arquitectura de *software*
- Diseño de datos
- Estilos y patrones arquitectónicos
- Diseño arquitectónico
- Evaluación de diseños arquitectónicos alternos
- Correlación flujo de datos en una arquitectura de *software*

Localización de subtemas en el libro de texto

SUBTEMAS	PÁGINAS
Arquitectura de *software*	276
Diseño de datos	278
Estilos y patrones arquitectónicos	280
Diseño arquitectónico	287
Evaluación de diseños arquitectónicos alternos	294
Correlación flujo de datos en una arquitectura de *software*	297

Comentarios generales

El primer sistema separado por módulos es un ejemplo de un **sistema con arquitectura**. La separación de las actividades de diferente naturaleza dentro de un mismo sistema facilita el entendimiento de cada una de sus partes y permite reconocer la importancia dentro del análisis de diseño para determinar claramente los límites entre ellos, como una buena práctica.

Aspectos como un bajo acople y una alta cohesión entre los módulos de un sistema otorgan una independencia funcional de gran valor para los desarrollos de sistemas que se espera sean escalables y robustos es su conformación en el tiempo.

Un buen modelo de diseño en la arquitectura da soporte a los conceptos mencionados, y la acogida de uno o varios patrones arquitectónicos le otorgan un grado alto de robustez, tanto en las capas de los datos, como en su mismo modelo de diseño.

El objetivo de este capítulo es que el estudiante comprenda cuál es el enfoque sistemático que se le debe dar como tratamiento al diseño arquitectónico y su importancia dentro de la ingeniería del *software*.

Desarrollo

Arquitectura del *software*

Si se visualiza un producto de *software* como un edificio, se podría decir como analogía que su arquitectura representa la manera en que los diversos componentes del mismo se integran para formar un todo cohesionado.

La arquitectura de *software* se puede definir como la estructura o las estructuras del sistema que permiten que un ingeniero del *software* realice las siguientes acciones:

1) Analizar la efectividad del diseño para cumplir con los requisitos establecidos.

2) Considerar opciones arquitectónicas en una etapa en que aún resulta relativamente fácil hacer cambios en el diseño.

3) Reducir los riesgos asociados con la construcción del *software*.

Importante:

La arquitectura de *software* es necesaria porque:

- Sus representaciones permiten la comunicación entre todas las partes interesadas en el desarrollo de un sistema de cómputo.
- Destaca las decisiones iniciales relacionadas con el diseño que tendrá un impacto profundo en todo el trabajo de la ingeniería de *software* que le sigue y en el éxito final del sistema como entidad operacional.
- Constituye un modelo relativamente pequeño e intelectualmente comprensible de cómo está estructurado el sistema y cómo trabajan juntos sus componentes.

Diseño de datos

Se debe separar el diseño de datos en dos partes: el **nivel arquitectónico** y el **nivel de componentes**.

El **diseño de datos, a nivel arquitectónico**, aboga por un enfoque de utilidad para la colaboración entre las diferentes bases de datos que contienen los datos operativos del negocio. Esta información es vital para obtener ventajas en diferentes líneas, tanto para análisis simple a nivel operativo, como para análisis para la toma de decisiones a nivel ejecutivo.

Para esta segunda línea, se han desarrollado técnicas como la **minería de datos**, para extraer conocimiento de la información y lograr extraer patrones de comportamiento en los datos y tendencias.

El **diseño de datos a nivel de componentes** se concentra en la representación de estructuras de datos a las que se tiene acceso en forma directa mediante uno o más componentes de *software*.

📖 **Actividad**:

En las páginas 279 – 280 del libro de texto se encuentra un conjunto de principios para la especificación de datos. Léalo y analícelo con respecto al modelo de análisis, que es donde se origina realmente el diseño de datos existente en cada componente.

Estilos y patrones arquitectónicos

Es preciso indicar que un estilo arquitectónico también se puede ver como una plantilla. Un **estilo arquitectónico** es una transformación impuesta por el diseño de todo un sistema. Su objetivo es establecer una estructura para todos los componentes del sistema.

Un **patrón arquitectónico** también impone una transformación en el diseño de la arquitectura, pero además:

1) Su alcance es mejor porque se concentra en un aspecto en lugar de toda la arquitectura.

2) Impone una regla sobre la arquitectura.

3) Tiende a abarcar aspectos específicos del comportamiento dentro del contexto de la arquitectura.

Es posible encontrar una breve taxonomía de estilos arquitectónicos a través del tiempo.

Importante:

Estilos arquitectónicos:

Arquitectura centrada en datos: en el centro de la arquitectura radica un almacén de datos, para el cual existen componentes de aplicaciones que tienen opción de gestionar dicha información. Este estilo promueve la capacidad de integración.

Arquitectura de flujo de datos: se aplica cuando los datos de entrada deben transformarse en datos de salida mediante una serie de componentes para el cálculo o la manipulación.

Arquitectura de llamada y retorno: permite a un arquitecto de sistema obtener una estructura de programa que resulte fácil de gestionar. Posee dos subestilos: **arquitectura de programa principal/subprograma** y la de **llamada de procedimiento remoto**.

Arquitectura orientada a objetos: se crean componentes que encapsulan los datos y las operaciones que deben aplicarse para manipularlos. Componentes coordinados mediante mensajería entre los objetos.

Arquitectura estratificada: se presentan varias capas en donde cada una de ellas realiza operaciones que se acercan, progresivamente, al conjunto de instrucciones de la máquina.

Por otro lado, los patrones arquitectónicos definen un enfoque específico para el manejo de algunas características de comportamiento del sistema. Una de estas características puede ser la **concurrencia**, que busca que la máquina simule paralelismo en la ejecución de varias tareas a la vez. Las aplicaciones tienen varias maneras de llevar a cabo esta simulación, y cada una de estas formas tiene su patrón arquitectónico correspondiente.

También se puede citar la **persistencia**, que permite a un objeto instanciado (creado a partir de una clase) en un momento dado, depositar sus valores o atributos en un almacén de datos, que los vuelve visibles y accesibles cuando se necesite acceder a ellos para alguna operación.

Al terminar el proceso de diseño, el ingeniero de *software* tiene en sus manos varias opciones arquitectónicas para decidirse con respecto a la que fungirá como la oficial en un trabajo específico. Existen algunas preguntas que facilitan el encontrar cuál de las opciones definidas es la mejor.

> 🕮 **Actividad:**
>
> En la página 287 del libro de texto se encuentra un conjunto de preguntas que facilitan la escogencia del modelo arquitectónico por seguir. Léalas y analícelas para entender de qué manera se facilita la discriminación y se obtiene la mejor opción.

Diseño arquitectónico

Cuando se habla del diseño arquitectónico a ser asociado, debe ponerse en perspectiva el sistema que está por construirse. En este diseño se debe considerar las **entidades externas**, tales como otros sistemas o personas, y **la naturaleza de la interacción**. Ambos elementos obtienen la información de la etapa de requerimientos.

Este diseño arquitectónico de contexto que se menciona, permite entender cómo va a interactuar el sistema con las entidades que lo limitan en todas las direcciones.

Los sistemas que interactúan con el sistema destino se clasifican como: **superordinados**, **subordinados**, **al nivel de par** y como **actores**.

> 🕮 **Actividad:**
>
> En las páginas 288 – 289 del libro de texto se encuentra la descripción de dichos tipos de sistemas. Léala y analícela dentro del contexto del sistema destino.

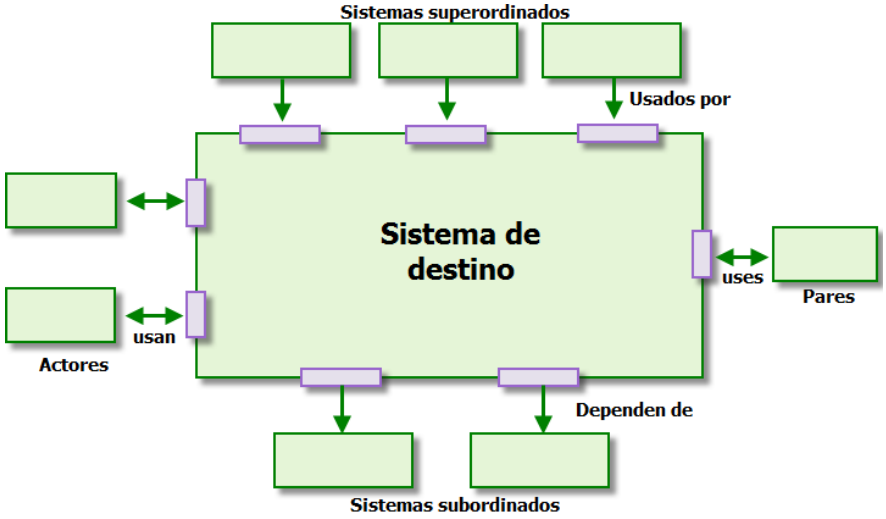


Figura 03 – Diagrama de contexto arquitectónico.

Otro elemento que apoya la definición del diseño en la arquitectura es el **arquetipo**. Es un patrón que representa una abstracción central importante en el diseño de la arquitectura para el sistema destino.

Los arquetipos forman la base de la arquitectura, pero no se puede dejar de lado que son **abstracciones** que deben refinarse cada vez más conforme se avanza con el diseño arquitectónico.

En este avance del diseño, tenemos que eligir cuáles componentes deben mantenerse y en qué forma. Estos componentes de la arquitectura del *software* se derivan de tres fuentes: los dominios de la aplicación, la infraestructura y la interfaz.

Lógicamente, en la creación del modelo de análisis no se trabajó la parte arquitectónica, por lo que deberá poner mucha atención en esta etapa para la derivación y el refinamiento de los componentes de dicha capa arquitectónica, apoyados en un diseño que es iterativo.

Evaluación de diseños arquitectónicos alternos

Un método de análisis de compensación para la arquitectura

El Instituto de Ingeniería de *software* (SEI por sus siglas en inglés) desarrolló un método conocido por sus siglas: MACA. Dicho método busca definir un proceso de evaluación iterativo para las diferentes arquitecturas de *software*. Se realizan las siguientes actividades de análisis de diseño, también iterativamente.

Importante:

Actividades de análisis de diseño que se presentan iterativamente.

1. **Recopilar escenarios**: se desarrolla un conjunto de casos de uso, para presentar el sistema desde el punto de vista del usuario.
2. **Deducir requisitos, restricciones y descripción de entornos**: es un proceso asociado a la ingeniería de requisitos y busca asegurar que se atiendan las preocupaciones descubiertas en el proceso.
3. **Describir los estilos/patrones arquitectónicos elegidos**: se basa en los puntos 1 y 2.
4. **Evaluar cada atributo de calidad de manera aislada**: ejemplos son confiabilidad, desempeño, seguridad, entre otros.
5. **Identificar la sensibilidad de cada atributo**: el cambio en un atributo que afecte significativamente la arquitectura es un punto de sensibilidad para la misma.
6. **Analizar las arquitecturas alternas basadas en la sensibilidad de cada atributo**: este proceso se basa en el paso 5.

La posible eliminación de las arquitecturas presentadas o la modificación de las aceptadas se deben basar, principalmente, en los puntos 5 y 6, para luego volver a aplicar el MACA.

Complejidad arquitectónica

Esta técnica consiste en considerar las dependencias entre los componentes dentro de la arquitectura, las cuales están orientadas por la información, el flujo de control, o ambas, dentro del mismo sistema. Estas dependencias se agrupan en tres partes: las **compartidas**, que se dan entre consumidores que

usan el mismo recurso o productores que producen para los mismos consumidores; las de **flujo**, que representan las relaciones de dependencia entre productores y consumidores de recursos y las **restringidas**, que representan restricciones de flujo relativo de control entre un conjunto de actividades.

Estas dependencias guardan similitud en concepto, con el **acoplamiento** mencionado en capítulos anteriores.

> 🕮 **Actividad:**
>
> En la página 298 del libro de texto se encuentra la descripción de otro método de evaluación de arquitectura. Léalo y analícelo dentro del contexto de los anteriores métodos descritos en este documento.

Correlación del flujo de datos en una arquitectura de *software*

Debido a que hasta el momento todas las arquitecturas presentadas son diferentes entre sí, es fácil deducir que la correlación que se esperaría realizar y que logre la transición del modelo de análisis a diversos estilos arquitectónicos, sea compleja y difícil.

Existe una técnica de correlación arquitectónica con el nombre de: **llamada y retorno**, que permite a un diseñador derivar arquitecturas de llamada y retorno algo complejas a partir de DFD dentro del modelo de análisis.

Flujo de transformación

Dentro de un flujo de datos normal tenemos un ingreso de datos que llamaremos **flujo de entrada**, una acción con estos datos que llamaremos **centro de transformación** y, un conjunto de datos alterados de manera indicada y definida, que llamaremos **flujo de salida**. El flujo general de los datos ocurre de manera secuencial y escoge algunas de las posibles rutas del flujo. A todo este proceso se le conoce con el nombre de **flujo de transformación**.

Flujo de transacción

Una **transacción** se puede ver como la esencia de un flujo de información. Esta transacción tiene la capacidad de disparar otro nuevo flujo de datos por una de varias rutas. A este comportamiento se le llama **flujo de transacción**, porque se convierte la información del mundo exterior y se crea la transacción.

En el flujo se presenta un elemento que es el que dispara la **acción** y la encarrila a alguno de los flujos existentes (rutas de acción). Este elemento se conoce como el **centro de transacción**.

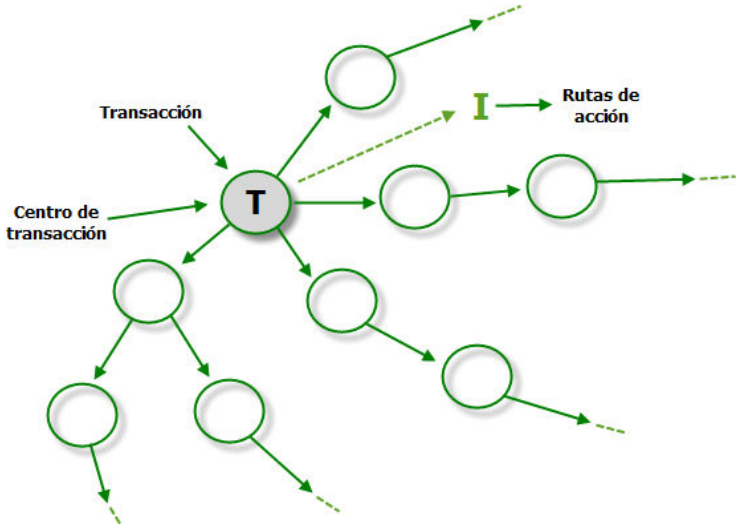


Figura 04 – Flujo de transacción.

Correlación de transformaciones

Esta correlación está conformada por un conjunto de pasos de diseño que permiten que un DFD, con características de flujo de de transformación, se correlacione con un estilo arquitectónico específico.

Con el fin de relacionar los DFD dentro de una arquitectura, se llevan a cabo los siguientes pasos de diseño:

Importante:

Pasos de diseño para relacionar los DFD dentro de una arquitectura:

1. **Revisar el modelo fundamental del sistema**: diagrama de contexto para un análisis con orientación a DFD.

2. **Revisar y refinar los DFD para el *software***: pasar del nivel de contexto a los siguientes niveles, hasta encontrar un nivel de cohesión tan alto que incite a la creación de un componente.

3. **Determinar si el DFD tiene características de flujo de transacción o de transformación**: se seleccionan las características globales del flujo de todo el *software*, con base en la naturaleza del DFD, para determinar qué tipo de flujo es.

4. **Aislar el centro de transformación al especificar límites de flujo de entrada y salida**: se deben resaltar en este paso solo los límites razonables para el centro de transformación según considere el analista.

5. **Realizar una "factorización de primer nivel"**: al encontrar un flujo de transformación, se correlaciona un DFD con una arquitectura de "llamada y retorno" que proporciona control para el procesamiento de la información de entrada, transformación y salida.

6. **Realizar una "factorización de segundo nivel"**: se obtiene al correlacionar las transformaciones individuales (burbujas) de un DFD con los módulos apropiados dentro de la arquitectura.

7. **Refinar la arquitectura de primera iteración empleando diseño heurístico para mejorar la calidad del *software***: los componentes se expanden y contraen para producir una factorización sensible (buena cohesión, acoplamiento mínimo y una estructura sin problemas).

📖 **Actividad:**

En las páginas 299 – 305 del libro de texto se encuentran descritos de manera más amplia, los pasos del diseño que permiten relacionar los DFD con una arquitectura y la correlación de transformaciones. Léalos y analícelos detenidamente, buscando entender de manera detallada este tema de tanta relevancia para el curso al cual asiste.

Correlación de transacciones

Ahora se darán los pasos dentro del diseño para correlacionar el flujo de transacción con una arquitectura de *software*. Es importante indicar que tanto los tres primeros pasos como el último son idénticos a los de las correlaciones en las transformaciones explicadas anteriormente.

Importante:

Pasos de diseño para relacionar los DFD dentro de una arquitectura:

1. **Revisar el modelo fundamental del sistema**.
2. **Revisar y refinar los DFD para el *software***.
3. **Determinar si el DFD tiene características de flujo de transacción o de transformación**.
4. **Identificar el centro de transacción y las características de flujo de cada una de las rutas de acción**: la ubicación del centro de transacción se desprende directamente del DFD. Este se encuentra en el origen de varias rutas de acción que fluyen de él de manera radial.
5. **Correlacionar el DFD con una estructura de programa sensible al procesamiento de la transacción**: cada camino del flujo de acción del DFD se correlaciona con una estructura que corresponde a sus características de flujo específicas.
6. **Factorizar y refinar la estructura de transacción y la de cada camino de acción**: cada camino de acción del DFD tiene sus propias características de flujos de información.
7. **Refinar la arquitectura de primera iteración empleando diseño heurístico para mejorar la calida del *software***.

📖 **Actividad:**

En las páginas 307 – 310 del libro de texto se encuentra descrita, de manera más amplia, los pasos del diseño que permiten relacionar los DFD con una arquitectura y la correlación de transacciones. Léalos y analícelos detenidamente, buscando entender este tema de tanta relevancia para el curso al cual asiste. Evalúe cada aspecto mencionado y contrástelo con lo definido en la correlación de transformaciones.

Ejercicios sugeridos

En la página 312 del libro de texto encontrará algunos problemas y puntos por considerar sobre el capítulo que acaba de estudiar.

Es importante que intente resolver los siguientes ejercicios. Esto le permitirá medir su aprendizaje.

47. Empleando la arquitectura de una casa como metáfora, realice comparaciones con la arquitectura del *software*. ¿En qué son similares las disciplinas de la arquitectura clásica y la del *software*? ¿En qué se diferencian?

48. ¿Explicar la diferencia entre una base de datos que sirve a una o más aplicaciones de negocios convencionales y un almacén de datos?

49. ¿Que significan los términos: estilo arquitectónico, patrón arquitectónico y marco conceptual? Buscar en Internet y describir la diferencia entre cada uno de estos términos y sus contrapartes.

50. Empleando un DFD, y una descripción del procesamiento, describir un sistema de cómputo que tenga distintas características de flujo de transformación. Defina los límites de flujo y correlacionar el DFD con la arquitectura del *software* empleando la técnica de "correlación de transformaciones".

Capítulo

11

Diseño al nivel de componentes

"Invariablemente se descubre que un sistema complejo que funcionaba ha evolucionado a partir de un sistema simple que también funcionaba."

John Gall

Sumario

Los temas que se tratan en este capítulo son:

- ¿Componente? ¿Qué es?
- Diseño de componentes basado en clases
- Conducción del diseño al nivel de componentes
- Diseño de componentes convencionales

Localización de subtemas en el libro de texto

SUBTEMAS	PÁGINAS
¿Qué es un componente?	316
Diseño de componentes basado en clases	322
Conducción del diseño al nivel de componentes	331
Diseño de componentes convencionales	340

Comentarios generales

La existencia de conceptos como **componentización** tiene su origen en la idea de reutilizar las cosas, tal y como se realiza en las diferentes actividades del diario vivir. El desarrollo de *software* no es la excepción, más aun cuando su propia naturaleza facilita este tipo de implementación, al ser tan fácil su multiplicación o reproducción.

El concepto de **componente** se puede definir como un proveedor de servicios al cual se le cargan algunos insumos (valores de entrada) y devuelve algún resultado de valor para el proceso que se lleva a cabo.

Lo que ocurrió adentro de él se debe entender como un proceso de caja negra, donde su implementación debe ser ignorada por el cliente a nivel de *software*, y tan solo debe retomar lo devuelto por el servicio utilizado.

Lamentablemente no es tan fácil trasladar el tema de los componentes a la realidad, pues así como tiene grandes ventajas cuando es bien aplicado, también tiene grandes complicaciones asociadas cuando no se respetaron sus fundamentos, mismas que se aclararán en este capítulo.

Desarrollo

¿Componente? ¿Qué es?

Pensar en qué es un componente en un ambiente de desarrollo de *software* es como pensar en la separación de aposentos existente para el proceso de construcción, que se da cuando se está construyendo una casa a nivel modular.

Cada uno de estos módulos representa un componente de la casa, y la suma de estos representa la casa en su totalidad, con sus respectivos mecanismos de comunicación y accesos entre sus aposentos. Esta es una visión un poco generalizada que se tiene de la palabra componente en el ambiente de desarrollo de *software*.

La definición de "componente" en el ambiente de ingeniería de *software* es "**un bloque de construcción modular, desplegable y reemplazable de un sistema que encapsula implementación y expone un conjunto de interfaces**". Por otro lado, el significado que asume algunas veces en el mundo informático depende del contexto en que se mencione y de quién lo mencione.

Por ejemplo, lo podemos escuchar relacionado con: **orientación a objetos**, bajo el **concepto convencional** o **relacionado con el proceso**. Se va a describir rápidamente cada una de estas formas con que se relaciona.

Considere el significado de "clase" dentro del mundo **orientado a objetos**, desde este concepto, un componente es un **contenedor de un conjunto de clases que colaboran entre sí**. Cada clase debe contener los atributos y los métodos que le permiten llevar a cabo su implementación.

También se tiene la interfaz entre las clases, que mediante mensajería entre ellas, logra la comunicación deseada y que fue definida en la etapa de análisis y en la de diseño.

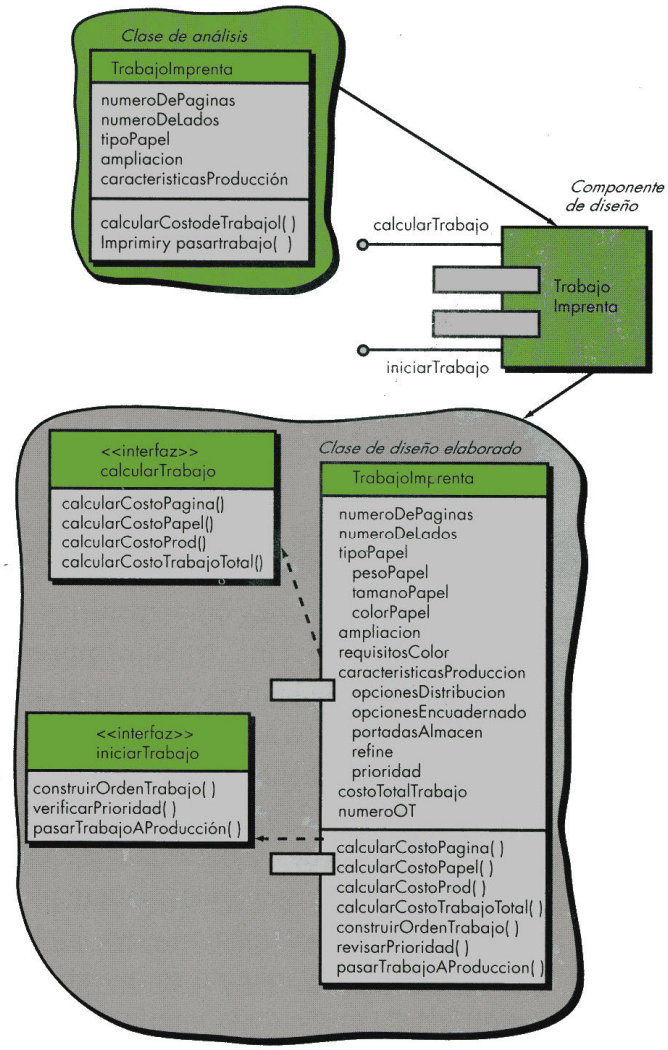


Figura 05 – Elaboración de un componente de diseño.

Otro enfoque que encontramos es el **convencional**, el cual busca definir un componente como un **elemento funcional (módulo) que incorpora la lógica de procesamiento**, las **estructuras internas** de los datos necesarios para implementar dicha lógica, y **una interfaz** que permita la invocación de dicho componente y el paso de datos.

Cada modulo reside dentro de la arquitectura y representa uno de tres papeles: como **componente de control**, como **componente de dominio** o, como **componente de infraestructura**. Al igual que el concepto orientado a objetos, este enfoque deriva del modelo de análisis.

Por último, se tiene el enfoque **relacionado con el proceso**, dentro del cual se parte de la existencia de una biblioteca de componentes, misma que se encuentran debidamente descrita y documentada para que los ingenieros de un proyecto conozcan qué funciones realizan y cuáles, de calzar en sus necesidades, pueden utilizar para consumirlas e ir poblando la arquitectura del proyecto, logrando con esto ahorro de tiempo en desarrollo para funcionalidades ya encapsuladas.

Diseño de componentes basados en clases

Ya se ha mencionado que el concepto de componente está directamente relacionado con el tema de las **clases**, las cuales tienen su nacimiento en la etapa de análisis descrita en capítulos anteriores.

La lógica encapsulada en ellas (operaciones), los atributos que las describen, y las interfaces que permiten que se manifiesten son los elementos que permiten pensar en el inicio de la etapa de construcción.

Existe una serie de principios en el mundo de diseño de componentes que es importante describir acá, con el fin de conformar diseños cada vez más sensibles al cambio y por ende con mayor adaptabilidad.

Importante:

Principios para el **diseño de componentes**:

1. **Abierto – cerrado**: el componente de un modulo debe estar abierto para extensión dentro del dominio funcional que atiende, pero cerrado para modificación a nivel de código o lógica**.**

2. **Sustitución por Liskov**: cualquier clase derivada de una clase principal se debe apegar a cualquier contrato implícito entre la clase principal y los componentes que la usan.

3. **Inversión de la independencia**: depende de las abstracciones, no de las concreciones. Las abstracciones son el lugar donde un diseño se puede extender sin grandes complicaciones.

4. **Segregación de la interfaz**: es mejor tener muchas interfaces específicas del cliente que una de propósito general. Se debe tener una interfaz especializada para servir a cada categoría importante para el cliente.

5. **Equivalencia entre reutilización y versión**: la esencia de la reutilización es la misma que la de la versión. Se deben agrupar las clases reutilizables en paquetes que pueden manejarse y controlarse a medida que evolucionan nuevas versiones.

6. **Principio de cierre común**: las clases que cambian juntas deben permanecer juntas. Cuando las clases se empaquetan como parte de un diseño, deben atender la misma área de funciones o comportamientos.

7. **Reutilización**: las clases que no se reutilizan juntas no deben agruparse juntas. Esto es porque innecesariamente se tendría que estar actualizando otros paquetes, solo para que cierta funcionalidad sea la que sirva.

📖 **Actividad:**

En las páginas 322 – 325 del libro de texto se encuentra el conjunto de principios descritos anteriormente, de manera más detallada. Lea y analice estos principios para reforzar sus conocimientos sobre el tema.

Tenemos por aparte un grupo de tres lineamientos que apoyan al diseño y que también son importantes de mencionar, tal y como lo haremos a continuación.

Los componentes deben traer consigo desde su creación en el modelo arquitectónico, una **convención de asignación de nombres** que se va a refinar hasta llegar al modelo de diseño, al nivel de componentes.

Las **interfaces** deben seguir estos lineamientos:

- Cuando los diagramas se vuelvan más complejos se debe usar la representación de líneas y círculo para la interfaz, en lugar del enfoque más formal del recuadro UML y la fecha con línea de guiones.
- Por razones de consistencia, las interfaces deben fluir desde la izquierda del recuadro del componente.
- Solo deben mostrarse las interfaces relevantes del componente en cuestión.

Las **dependencias** deben modelarse de izquierda a derecha y las **herencias** de abajo hacia arriba, para mejorar su legibilidad.

Se tiene también un par de características de alta importancia en el mundo de los componentes, tal como lo son la alta **cohesión** y el bajo **acoplamiento**. Ya en el capítulo 8 se mencionó un poco, mas es importante retomarlo con fuerza, en esta descripción ampliada y detallada de los componentes.

La **cohesión** se define, en el contexto de los componentes, como la relación forzada esperada para los atributos y operaciones existentes en la o las clases asociadas al componente gracias a su misma naturaleza, por lo que fueron agrupadas.

Dentro de esta característica, se encuentran también diferentes niveles de intensidad, tales como los siguientes:

- funcional
- de capa
- de comunicación
- secuencial
- procedimental
- temporal
- de utilidad

> 🕮 **Actividad:**
>
> En las páginas 327 – 328 del libro de texto, se encuentra el detalle de los niveles descritos anteriormente. Léalos y analícelos para reforzar sus conocimientos. Elabore un esquema que muestre las relaciones entre estos niveles.

El **acoplamiento** es una medida cualitativa del grado en que las clases necesitan conectarse entre sí. También permite visualizar, mediante la mensajería, que sea posible disparar acciones necesarias de alguna de las clases involucradas. Por otro lado, se ha dicho también que cuanto **más bajo** sea el nivel de acoplamiento, mejor será para el desarrollo de cada componente por separado, y para el sistema en general.

También se encuentran diferentes niveles de acoplamiento de clases, agrupado de la siguiente manera: como acoplamiento del **contenido**, como acoplamiento **común**, como acoplamiento de **estampa**, como acoplamiento de **datos**, como acoplamiento de **llamada** y **rutina**, como acoplamiento de **uso de tipo**, como acoplamiento de **inclusión** o **importación** y como acoplamiento **externo**.

🕮 **Actividad:**

En las páginas 329 – 330 del libro de texto, se encuentra estos niveles de acoplamiento descritos anteriormente, pero más detallados. Lea y analice esta información. Defina cada nivel con sus propias palabras.

Es importante anotar que aunque la comunicación entre los componentes es importante, debe reducirse al máximo, logrando con esto la mayor cantidad de **independencia** posible entre ellos y traer con esto facilidades en aspectos tales como el mantenimiento.

Conducción del diseño al nivel de componentes

Es responsabilidad del diseñador basarse tanto en el análisis como en los modelos arquitectónicos para transformarlos en una representación de diseño que proporcione detalles suficientes para iniciar la etapa de construcción.

Se ha definido un conjunto de tareas básicas para el diseño al nivel de componentes, que detallamos a continuación.

Importante:

Conjunto de **tareas básicas** para el **diseño** al nivel de componentes:

1. Identificar todas las clases de diseño que correspondan al dominio del problema.
2. Identificar todas las clases de diseño que corresponden al dominio de la infraestructura.
3. Elaborar todas las clases de diseño que no se adquieran como componentes reutilizables.
    a. Especificar los detalles del mensaje cuando las clases o los componentes colaboran.
    b. Identificar las interfaces apropiadas para cada componente.
    c. Elaborar atributos y definir los tipos y las estructuras de datos necesarios para implementarlos.
    d. Describir de manera detallada el flujo de procesamiento dentro de cada operación.
4. Describir fuentes de datos persistentes (bases de datos y archivos) e identificar las clases necesarias para manejarlas.
5. Desarrollar y elaborar representaciones del comportamiento de una clase o un componente.
6. Elaborar diagramas de despliegue para proporcionar detalles de la implementación adicional.
7. Factorizar todas las representaciones del diseño a nivel de componentes y siempre deben considerarse alternativas.

🕮 **Actividad:**

En las páginas 331 – 337 del libro de texto se detallan las tareas típicas del diseño al nivel de componentes. También se usa un ejemplo dentro del libro, con la explicación respectiva en cada punto descrito. Lea esta información y proponga un nuevo ejemplo.

Diseño de componentes convencionales

A inicios de los años sesenta, un grupo de científicos de la computación iniciaron con lo que hoy se conoce como los **fundamentos del diseño** a nivel de componentes, para componentes tradicionales.

Propusieron un conjunto de construcciones lógicas restringidas, a partir de las cuales se pudiera crear cualquier programa. Dentro de estas construcciones encontramos la **secuencia** (pasos secuenciales), la **condición** (eventos lógicos) y la **repetición** (bucles).

Se ha comprobado, tal y como se verá en capítulos posteriores asociados a las métricas de *software*, que las construcciones estructuradas reducen la complejidad del programa y, por lo tanto, mejoran la lectura, la prueba y el mantenimiento de la misma.

Esta forma de trabajo está basada en el concepto de Psicología llamado **fragmentación**, proceso de construcción humana que trabaja por patrones o agrupamientos, y facilita más su comprensión que si analizáramos cada parte por separado.

Por otro lado, para este conjunto de construcciones lógicas también tenemos su correspondiente **notación gráfica** (figura 06). Consiste en elementos de un **flujo de datos** que describen visualmente cada una de las construcciones definidas.

Tenemos un **objeto diamante** (si-entonces-si_no) para representar una **condición** lógica. Las flechas salientes y entrantes de este diamante muestran el **flujo de control**. Las cajas de procesamiento conectadas por una línea representan la **secuencia**.

La construcción de **repetición** ejecuta primero una parte de **hacer mientras**, prueba una condición y ejecuta una tarea del bucle de manera repetitiva, siempre y cuando la condición siga siendo verdadera. Una parte **repetir hasta** ejecuta primero la tarea de bucle, luego prueba una condición y repite la tarea hasta que la condición falla.

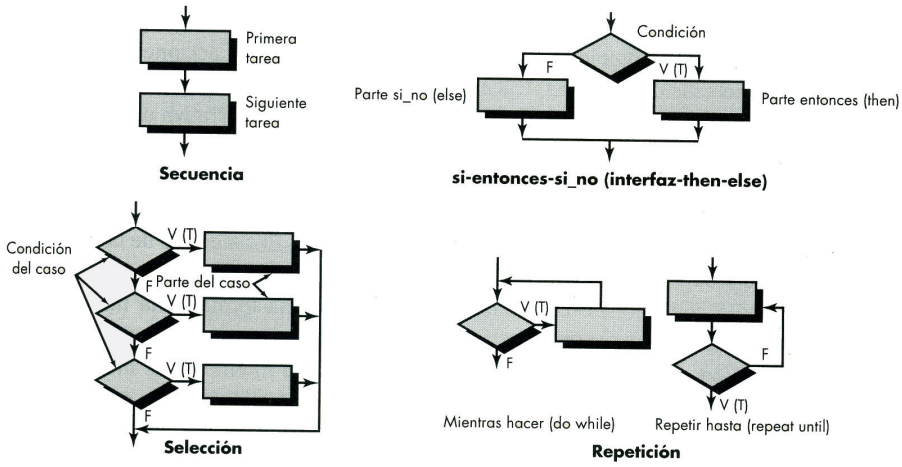


Figura 06 – Diagrama de flujo de construcciones.

Es importante tener presente que el uso exclusivo de las construcciones estructuradas puede llegar a introducir ineficiencia cuando se requiere un escape de un conjunto de bucles anidados o de condiciones anidadas.

Las construcciones de programación estructurada deben facilitar la comprensión del diseño. Si el hecho de respetarlas por completo introduce una complejidad innecesaria, es permitido violarlas en beneficio del buen diseño.

> 🕮 **Actividad:**
>
> En las página 342 – 343 del libro de texto se describe otro tipo de notación asociada con el diseño de componentes, la notación tabular del diseño, que utiliza una tabla de decisión. Lea esta información y compárela con la descrita anteriormente. ¿Qué tienen en común?, ¿en qué se diferencian?, ¿puede una suplantar completamente a la otra?, ¿en qué casos debe usarse cada una de ellas?

Además de las dos notaciones mencionadas anteriormente, tenemos también lo que se denomina como **lenguaje de diseño de programas** (PDL por sus siglas en inglés), más conocido como **seudocódigo**. Este es un lenguaje rudimentario porque utiliza el vocabulario de un idioma (inglés) y la sintaxis general de otro (un lenguaje estructurado de programación). Es claro que el enfoque que se le quiere dar es de diseño, para no enredarse con un enfoque orientado a la codificación directamente.

El PDL no es un lenguaje de programación; el encargado de diseño puede adaptarlo como quiera sin pensar en problemas o errores de sintaxis.

Se dice en términos generales, que el PDL es, en el ambiente de notación de diseño, el que mejor se ajusta o el que ofrece la mejor combinación de características. El PDL puede incrustarse directamente en los listados de código fuente (existe *software* para interpretar y extraer el código del seudocódigo), mejorando la documentación y facilitando más el mantenimiento del diseño.

Es claro también que es una opinión y que puede no ser tan cierta para algunos encargados de diseño, pues la elección de una herramienta de diseño estará en función de manera más estrecha, con factores humanos que con los atributos técnicos.

Ejercicios sugeridos

En la página 347 del libro de texto encontrará algunos problemas y puntos por considerar sobre el capítulo que acaba de estudiar.

Es importante que intente resolver los siguientes ejercicios. Esto le permitirá medir su aprendizaje.

51. ¿Consideraría que los patrones son una forma efectiva de reutilizar en el diseño a nivel de componentes?, ¿por qué?, ¿cuáles son las desventajas de este enfoque?

52. Enliste, según su criterio, las ventajas y las desventajas que se presentan en la reutilización de componentes.

53. Describa el paradigma orientado a objetos mediante sus propios argumentos.

54. Explique: ¿por qué no siempre los ahorros que se presentan en la reutilización del *software* a partir de componentes, no son proporcionales al tamaño de los componentes que la reutilizan?

55. ¿Por qué es importante crear abstracciones que sirvan como interfaz entre componentes?

R

Referencias

Bruegge, B., Dutoit, A. (2002). Ingeniería de *Software* Orientado a Objetos. México D.F.: Prentice Hall.

Larman, C. (2004). UML y Patrones. Segunda edición. Madrid: Prentice Hall.

Sommerville, I. (2002). Ingeniería de *Software*. Sexta edición. México D.F.: Addison Wesley.

G

Glosario de análisis de *software*

DFD: un diagrama de flujo de datos (DFD por sus siglas en español e inglés) es una representación gráfica del "flujo" de datos a través de un sistema de información. Un diagrama de flujo de datos también se puede utilizar para la visualización de procesamiento de datos (diseño estructurado). Es una práctica común para un diseñador dibujar un contexto a nivel de DFD que primero muestra la interacción entre el sistema y las entidades externas. Este contexto a nivel de DFD se "explotó" para mostrar más detalles del sistema que se está modelando.

SEI: el *Software Engineering Institute* es un instituto federal estadounidense de investigación y desarrollo, fundado por el Congreso de los Estados Unidos en 1984. El objetivo de este instituto es desarrollar modelos de evaluación y de mejora en el desarrollo de *software* que dieran respuesta a los problemas que generaba al ejército estadounidense la programación e integración de los sub-sistemas de *software* en la construcción de complejos sistemas militares. Esta institución es financiada por el Departamento de Defensa de los Estados Unidos y administrada por la Universidad Carnegie Mellon. Es un referente en Ingeniería de *software* debido al desarrollo del modelo SW-CMM (1991) que ha sido el punto de arranque de todos los que han formado parte del modelo desarrollado sobre el concepto de capacidad y madurez, hasta el actual CMMI.

Enlaces electrónicos

CAPÍTULO	CONCEPTO	PÁGINA
10	Página oficial SEI	http://www.sei.cmu.edu/
10	Descripción de los DFD	http://es.wikipedia.org/wiki/Diagrama_de_flujo_de_datos
11	Descripción de la teoría de componentes	http://es.wikipedia.org/wiki/Componentes_de_software
11	Descripción de la teoría de las estructuras condicionales	http://es.wikipedia.org/wiki/Condicional_(programación)
11	Descripción de la teoría de los PDL o pseudocódigos	http://books.google.co.cr/books?id=cOVSIiw2mLAC&pg=PA60&lpg=PA60&dq=PDL%2Bpseudoc%C3%B3digo&source=bl&ots=4bqhhv0Q8m&sig=i43Fq0d5pS4ta9AtC7FfGad-96g&hl=es&ei=CbysSaaVC4GCtwet6oGEBg&sa=X&oi=book_result&resnum=1&ct=result#PPA131,M1

Tema

4

Diseño exterior y evaluación del *software*

"Se ha construido el software.
Ahora solo estamos tratando de que funcione…"
Declaración hecha en una reunión conjunta
de programa ejecutivo STARS E–8A FSD

Objetivos

El trabajo realizado hasta ahora responde a actividades de creación, como lo son el análisis y el diseño. Es el momento de llevar a cabo la acción contraria y sacar conclusiones de lo avanzado hasta ahora. El tema de las pruebas es importante pues es acá donde se trata de falsear lo creado, exponiendo los modelos del sistema y encontrando las fallas antes de llegar a la mesa de trabajo del cliente final.

Además, se va a explorar el amplio mundo del análisis y diseño de interfaces, ventanas mediante las cuales los usuarios finales interactúan con la máquina cliente y logran explotar sus bondades como sistema.

Cuando termine de estudiar este tema, estará en capacidad de:

- Entender la importancia de un buen análisis, diseño y evaluación para la interfaz de los sistemas y su interacción con los usuarios finales.

- Aplicar las estrategias que existen sobre los diferentes tipos de prueba, tanto para ambiente tradicional como para el orientado a objetos.

- Aprender a incluir, dentro de la estrategia de pruebas, las diferentes técnicas definidas para el diseño de casos de prueba.

Sumario

Los capítulos que conforman este tema se encuentran en el libro de texto y son los siguientes:

- **Capítulo 12 – Diseño de la interfaz de usuario**
- **Capítulo 13 – Estrategias de prueba del *software***
- **Capítulo 14 – Técnicas de prueba del *software***

Además, podrá consultar las siguientes secciones:

- **Referencias bibliográficas**
- **Glosario de términos**
- **Enlaces electrónicos relacionados con el tema**

Capítulo

12

Diseño de la interfaz de usuario

"Siempre he deseado que mi computadora sea tan fácil de manejar como mi teléfono. Mi deseo se ha vuelto realidad. Ya no sé cómo usar mi teléfono."
Bjarne Stronstrup (creador del C++)

Sumario

Los temas que se tratan en este capítulo son:

- Las reglas básicas
- Análisis y diseño de la interfaz de usuario
- Análisis de la interfaz
- Pasos del diseño de la interfaz
- Evaluación del diseño

Localización de subtemas en el libro de texto

SUBTEMAS	PÁGINAS
Las reglas de oro	351
Análisis y diseño de la interfaz de usuario	356
Análisis de la interfaz	359
Pasos del diseño de la interfaz	368
Evaluación del diseño	377

Comentarios generales

A todo ese gran esfuerzo realizado en el análisis, ese listado de requerimientos ya tentativamente resueltos por la lógica aplicada y todo el plano arquitectónico que los soporta, se le debe permitir trabajar mediante algún mecanismo que el ser humano interprete bien, y le pueda sacar provecho.

El diseño de la interfaz de usuario debe proveer precisamente eso. El tema de diseño de interfaz se puede dividir en tres grandes áreas como son el diseño para componentes, el diseño entre el *software* y otros productos y consumidores no humanos y el diseño para seres humanos.

En este capítulo se va a enfatizar en la tercera área mencionada, o sea, en el diseño de interfaz para el usuario final.

Se tocarán temas asociados a los principios que deben regir el diseño de dicha interfaz y a todas las reglas por seguir para obtener un mecanismo de aprovechamiento de toda la lógica desarrollada, misma que intentará resolver problemas definidos por el usuario que se encuentra al frente de la aplicación.

Desarrollo

Reglas básicas

Existen tres reglas en las que, con el tiempo, se ha llegado a comprobar su eficacia en lo que a diseño de interfaces se refiere. Estas reglas son las siguientes:

1) Dar el control al usuario.

2) Reducir la carga de la memoria del usuario.

3) Lograr que la interfaz sea consistente.

La acción de **dar el control al usuario** por sí sola parece una excelente idea fácil de acatar, mas no siempre resulta de esa manera en las diferentes interfaces a las que nos enfrentamos día a día, en particular en el uso de los sistemas.

Existen algunos principios de diseño que es necesario seguir para realmente otorgarle al usuario el control de la interfaz.

Importante:

Principios para darle el control al usuario en el **diseño de interfaces**:

1. **Definir los modos de interacción de forma que el usuario no realice acciones innecesarias o indeseables**: el usuario debe escoger en qué modo se encuentra. En caso de existir más modos debe poder cambiar en el momento que lo desee.

2. **Proporcionar una interacción flexible**: según el tipo de actividad por realizar en el sistema, así debe estar adecuada la interacción física entre la computadora y el usuario.

3. **Incluir las opciones de interrumpir y deshacer la interacción del usuario**: en cualquier momento el usuario debe tener la opción de interrumpir la secuencia y pasar a otra actividad.

4. **Depurar la interacción a medida que aumenten los grados de destreza**: cuanto más destreza acumulada tenga el usuario, éste debe tener la opción de personalizar la interacción a conveniencia.

5. **Oculte al usuario ocasional los elementos técnicos internos**: el usuario solo debe preocuparse por el ambiente de la aplicación, ninguna capa que esté por debajo de esta debe ser de conocimiento para el mismo, se da por resuelto.

6. **Diseñar interacción directa con los objetos que aparecen por pantalla**: cuanto más similar sea la manipulación dada a un objeto en la aplicación, a uno de la vida real, mejor la experiencia para el usuario.

La segunda regla es la de **reducir la carga en la memoria del usuario**. Es claro que al tener la computadora la oportunidad de almacenar información, se convierte en un fuerte aliado para retener la mayor cantidad de información que sea de interés para el usuario y para la aplicación.

También existen algunos principios de diseño que es necesario seguir para lograr que una interfaz reduzca la carga de memoria que recae en el usuario.

Importante:

Principios para lograr que una interfaz **reduzca la carga de memoria** que recae en el usuario:

1. **Reducir la demanda de memoria a corto plazo**: apoyarse en pistas visuales para recordarle al usuario la secuencia de pasos ya realizados.
2. **Definir valores por defecto que tengan significado**: acercarle al usuario los valores por defecto que aplique más a lo que normalmente el usuario digitaría.
3. **Definir accesos directos intuitivos**: debe estar directamente relacionado con la acción que va a ejecutar el acceso, y que su uso sea intuitivo.
4. **El formato visual de la interfaz debe basarse en una metáfora tomada de la vida real**: se le debe acercar al usuario en la medida de lo posible el ambiente y el flujo que normalmente haría si el proceso fuera manual.
5. **Desglosar la información de manera progresiva**: la información debe estar acomodada de manera jerárquica, mirando primero el nivel más abstracto y luego pasando al detalle.

Como tercera y última regla, se tiene la de **lograr que la interfaz sea consistente**. Esta regla lo que busca es definir un estándar de diseño en toda la información que se le va a presentar al usuario, tener mecanismos de entrada que estén restringidos a un conjunto limitado de opciones que se utilice de manera consistente, y mecanismos para trasladarse de una tarea a otra desarrollados también de manera consistente.

Los siguientes principios de diseño son necesarios para crear una interfaz consistente.

Importante:

Principios para lograr que una interfaz **sea consistente**:

Permitir que el usuario incluya la tarea actual en un contexto que tenga algún significado: el usuario debe sentirse identificado con la actividad que está realizando y el entorno de la aplicación le debe proporcionar esa sensación.

Mantener la consistencia en una familia de aplicaciones: todas las aplicaciones deben tener las mismas reglas de diseño ejecutadas.

Si modelos interactivos anteriores han generado expectativas en los usuarios, no se debe hacer cambios a menos que haya razones inexcusables: debe respetarse algunos estándares de la industria en cuanto a la interacción entre las aplicaciones. Ejemplo F1 para la ayuda.

Análisis y diseño de la interfaz de usuario

En el análisis y diseño de una interfaz de usuario, para cualquier aplicación, entran en juego cuatro modelos. La tarea del diseñador de interfaz es conciliar estos cuatro modelos para obtener la interface que mejor recoge las necesidades del cliente.

El **modelo del usuario**, que es la visión que se debe tener del usuario, sus características como ser humano, cualidades y calidades, entre otras. En resumen, la comprensión absoluta del usuario final (visión técnica).

El **modelo de diseño**, el cual incorpora datos, arquitectura, interfaz y las representaciones procedimentales del *software* (funciones y procedimientos).

El **modelo mental del usuario**, que es la percepción del sistema que tiene en la cabeza el usuario final. Un usuario puede tener una visión mucho más cercana de la realidad con solo una vez que haya utilizado la aplicación, que otro usuario con muchas más participaciones; todo depende del modelo mental con que venga.

El **modelo de la implementación** busca combinar la apariencia externa de la interfaz del sistema, con toda la información de ayuda que describe la sintaxis y la semántica del sistema, o sea, ayuda en libros, manuales, u otros materiales.

> 🕮 **Actividad**:
>
> Digite en algún buscador en la Internet (*Google*, *Yahoo*, *AOL Search*) un tema de su agrado y escoja la primera página de las encontradas.
>
> Obsérvela, utilícela un rato y luego describa en sus propias palabras, la experiencia como usuario dentro del contexto del diseño de interfaces. Indique su opinión al respecto, para cada uno de los modelos.

A la hora de enfocar el análisis y diseño de interfaces como un **proceso**, se necesita pensar en un modelo iterativo e incremental. Este modelo es similar al estudiado en capítulos anteriores, el llamado modelo en espiral. Dentro de este modelo, vamos a encontrar las siguientes etapas:

1. Análisis y modelado de usuarios, tareas y entornos.
2. Diseño de la interfaz.
3. Construcción de la interfaz.
4. Validación de la interfaz.

El modelo sugiere que cada una de estas etapas se hará más de una vez, y que el incremento por giro en esta significa elaboración de requerimientos y cambios en el diseño. A la vez, se esperaría en la etapa de construcción la creación de prototipos.

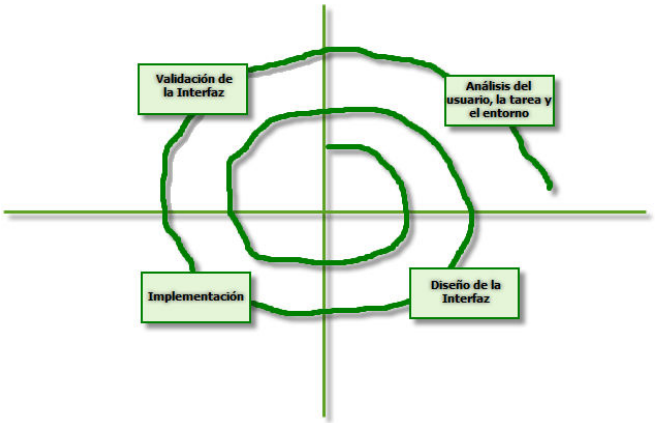


Figura 01 – El proceso de diseño de la interfaz de usuario.

Análisis de la interfaz

Un buen diseño de interfaz significa comprender cuatro cosas:

1) Las personas que interactúan por medio de la interfaz.
2) Las tareas que los usuarios finales deben realizar.
3) El contenido que se presenta como parte de la interfaz.
4) El entorno en que se realizan estas tareas.

Con respecto a las **personas** que usan el sistema, y debido a que cada una tiene una imagen mental distinta, es importante conciliar, como se dijo antes, la visión del usuario final con el modelo que tiene el ingeniero en su cabeza.

Entrevistas con el usuario, información de ventas y sus categorías, información de mercadotecnia y la proveniente de soporte, son fuentes para conocer a las personas que finalmente usarán la aplicación, y qué las motiva y las complace.

El **análisis y el modelado de las tareas** es otra de las cosas importantes de comprender. En esta etapa se trata de encontrar respuestas a preguntas como ¿qué trabajo hará el usuario en circunstancias específicas?, ¿cuáles tareas y subtareas realiza el usuario mientras trabaja?, ¿en qué secuencia se presentan las tareas del trabajo?, ¿cuál es la jerarquía de las tareas?

Como instrumento de recolección de requisitos usamos los **casos de uso**, solo que esta vez lo delimitamos a que un actor siempre será una persona.

A partir del este caso de uso, el ingeniero de *software* extraerá tareas y objetos, además del flujo general de la interacción.

La **elaboración de las tareas** se trabaja bajo el enfoque de descomposición funcional visto en el capítulo 9 de esta guía, que permite refinar las tareas de procesamiento requeridas.

Con la idea de comprender las tareas indispensables para alcanzar los objetivos de cada actividad, el ingeniero de *software* debe comprender las actividades que realizan las personas mediante los métodos manuales, y luego relacionarlas con un conjunto similar de tareas que se implementan en el contexto de la interfaz de usuario.

Otra etapa es la de **elaboración de los objetos** que soportan la interacción. En lugar de concentrarse tanto en las tareas, el ingeniero extrae del caso de uso alguna otra información suministrada por el usuario y obtiene los objetos físicos que soportarán dicha interacción.

Cuando se presentan los casos en donde varios usuarios deben interactuar con una misma interfaz, no basta con tener claro el análisis de la tarea ni la elaboración de los objetos. Para estos casos es necesario llevar a cabo el **análisis del flujo de trabajo**, el cual le permite al ingeniero de *software* entender cómo se realiza el proceso donde se involucra más de una persona y más de un papel a la vez.

La existencia de un flujo de trabajo establece **una jerarquía** de tareas que se debe respetar para obtener los resultados esperados. La jerarquía se deriva de una elaboración, paso a paso, de cada tarea solicitada por el usuario.

Otra actividad sumamente importante dentro del diseño de la interfaz es el **análisis del contenido de la pantalla**. Las tareas mencionadas anteriormente, traen consigo contenido que debe ser tratado (texto, imágenes, vídeos, entre otros).

Este contenido viene almacenado mediante objetos de datos, tales como componentes productores de tipo externo, mediante bases de datos externas o directamente mediante sistemas externos a la aplicación en mención. Es necesario considerar el formato y la estética en este punto para el contenido de la aplicación.

Como parte de las actividades finales por considerar, el **análisis del entorno** de trabajo se vuelve una obligación que debe ser analizada. La ocupación física que tendrá el usuario final debe estar en completa armonía con la interfaz que se ha de presentar en la aplicación.

Pasos del diseño de la interfaz

El libro de estudio sugiere que a pesar de la existencia de diferentes modelos para el diseño de la interfaz, se pueden identificar algunos pasos claves para llevarla a cabo:

- Definir los objetos y las acciones de la interfaz con base en el conocimiento desarrollado durante el análisis de la información.
- Definir eventos que cambiarán el estado de la interfaz y modelar este comportamiento.
- Representar cada estado de la interfaz como el usuario final lo verá.
- Indicar cómo interpreta el usuario el estado del sistema a partir de la interfaz proporcionada.

Cada paso del diseño de la interfaz, por ser iterativo, se dará varias veces y en cada uno de ellos se va a refinar y detallar cada paso anterior.

Un paso importante en el diseño de la interfaz es la definición de los objetos que ésta tendrá y las acciones que se le aplicarán, tal y como se haría en un caso de uso con los nombres (objetos) y verbos (acciones).

Una vez definidos los objetos, se clasifican por objetos tipo **destino** (Ej. impresora), **origen** (Ej. informe) y de **aplicación** (Ej. lista de correos). Además, cuando el diseñador está contento con cada objeto y tiene definidas las acciones asociadas a cada objeto, ya puede iniciar con el formato de la pantalla.

El proceso de formato de la pantalla es interactivo y en él se elabora el diseño gráfico y se ingresan iconos y el texto descriptivo definido como importante.

También se incorporan los elementos primarios y secundarios para los menús y si se logra encontrar una metáfora de la realidad que funcione para la aplicación, se especifica y se acopla al diseño para que se integre.

> 🕮 **Actividad**:
>
> Como parte de los pasos del diseño de la interfaz, en las páginas 371 – 372 del libro de texto, se indica que el tema de **patrones** de diseño para la interfaz de usuario no es parte integral de la naturaleza de dicho libro.
>
> Inicie una búsqueda en la cantidad de medios posibles, sobre el tema de **patrones de diseño** para la interfaz de usuario.
>
> Además, en las páginas 372 – 375 del libro de texto, se habla sobre **temas de diseño**. Analice esos temas e identifique qué acciones se pueden tomar para mitigarlos.

Evaluación del diseño

Después de seguir todos los lineamientos antes expuestos, lo que queda es llevar a cabo la evaluación del diseño definido. Corresponde iniciar algún mecanismo que permita al usuario final utilizar la interfaz de la aplicación y evaluar si ésta corresponde con lo esperado.

En la figura 02 puede observar cómo se trabajará con el ciclo de evaluación que permite definir si la aplicación satisface completamente al usuario final.

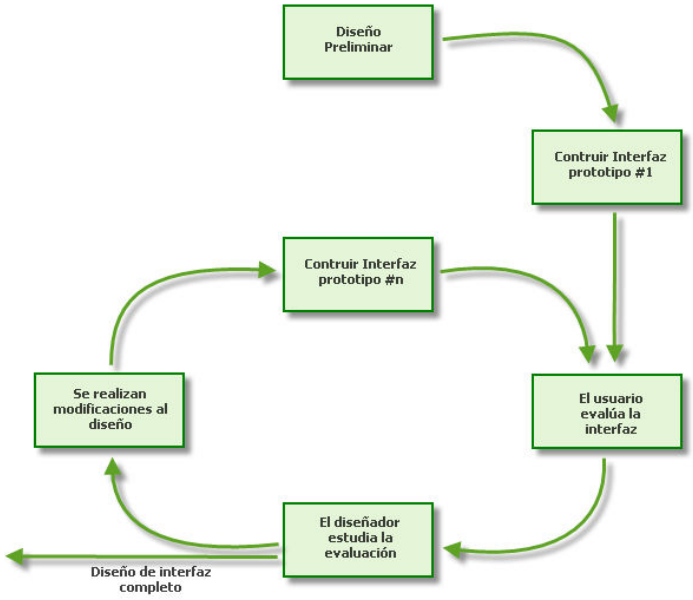


Figura 02 – Ciclo de evaluación de diseño de la interfaz.

Es importante entender que no es necesario tener terminado un prototipo para iniciar la evaluación de la calidad.

Desde las primeras iteraciones del bucle en el ciclo de vida, es posible comenzar a identificar problemas e iniciar su corrección para que, cuando se esté cerca del prototipo, los ajustes sean mínimos.

Ejercicios sugeridos

En la página 380 del libro de texto encontrará algunos problemas y puntos por considerar sobre el capítulo que acaba de estudiar.

Es importante que intente resolver los siguientes ejercicios. Esto le permitirá medir su aprendizaje.

56. Suponga que se le ha pedido desarrollar un portal para un banco vía web. Desarrolle en papel (según su criterio), los modelos del usuario, del díseño, del mental y de la implementación.

57. Como usuario final de la web, para entender que existe consistencia en un portal corporativo, a nivel de interfaz ¿qué necesita usted percibir en él?

58. Indique por medio de un listado, ejemplos de por qué es importante tomar en cuenta la variabilidad del tiempo de respuesta.

59. Desarrolle dos principios de diseño adicionales que "reduzcan la carga en la memoria del usuario".

60. Desarrolle dos principios de diseño adicionales que "den el control al usuario".

Capítulo

13

Estrategias de pruebas del *software*

"El optimismo es el peligro ocupacional de la programación, la prueba, el tratamiento." ***Kent Beck***

Sumario

Los temas que se tratan en este capítulo son:

- Un enfoque estratégico para las pruebas de *software*
- Aspectos estratégicos
- Estrategia de prueba para el *software* convencional
- Estrategia de prueba para el *software* orientado a objetos
- Pruebas de validación
- Pruebas de sistema
- El arte de la depuración

Localización de subtemas en el libro de texto

SUBTEMAS	PÁGINAS
Un enfoque estratégico para las pruebas de *software*	383
Aspectos estratégicos	390
Estrategia de prueba para el *software* convencional	391
Estrategia de prueba para el *software* orientado a objetos	402
Pruebas de validación	404
Pruebas de sistema	406
El arte de la depuración	409

Comentarios generales

La prueba es un elemento vital para garantizar que los pasos seguidos en la construcción de un producto permiten asegurar el funcionamiento satisfactorio, en su ejecución, a la hora de ponerlo a trabajar. Por este motivo se debe llevar a cabo, de manera planificada y ordenada, de forma que realmente permita ejercitar lo construido y llevarlo a todos los niveles en donde podría llegar a estar sometido en el ambiente de destino.

Una estrategia de pruebas de *software* integra los métodos de diseño de casos de prueba, para ensamblar un proceso que contenga todos los elementos necesarios en el nivel de entorno que proporcionen un resultado exitoso y que además permitan entender cuánto esfuerzo, tiempo y recursos se consumirán.

El resultado debe contener las etapas de planeación, diseño de casos, ejecución, recolección y evaluación de resultados importantes; deben incluirse los casos no exitosos.

Este capítulo tratará de detallar cada uno de estos aspectos mencionados anteriormente, con el afán de que usted comprenda lo necesario y beneficioso que resulta el explayarse, en la medida de lo posible, en esta etapa dentro del desarrollo de *software*.

Desarrollo

Un enfoque estratégico para las pruebas del *software*

En este capítulo es preciso resaltar el valor que debe tener la prueba dentro del amplio proceso de desarrollo de *software*. A pesar de ser parte de las últimas etapas, no por esto es de menor valor en dicho proceso y el retorno es invaluable y muy sano para el futuro sistema.

Es importante, por lo tanto, contar con una estrategia que permita tanto al usuario final como al equipo de desarrollo, saber que el producto elaborado está dentro de los parámetros esperados en cuando a su naturaleza y razón de ser.

La estrategia por escoger debe permitir la inclusión de actividades ejecutadas de manera sistemática que permitan incorporar técnicas y métodos específicos del diseño de casos de prueba.

Existen algunas características genéricas importantes de conocer, para la creación de plantillas y para las pruebas de *software*.

Importante:

Características genéricas importantes para la creación de **plantillas para pruebas**:

1. Un equipo de *software* debe efectuar revisiones técnicas formales y efectivas.
2. La prueba comienza a nivel de componentes y trabaja "hacia afuera", hacia la integración de todo el sistema de cómputo.
3. Diferentes técnicas de pruebas son apropiadas en diferentes momentos.
4. La prueba la dirige el desarrollador de *software* (para proyectos grandes existirá un grupo independiente).
5. La prueba y la depuración son actividades diferentes, pero la segunda debe estar incluida en cualquier estrategia de prueba.

La existencia de una estrategia es importante también para el jefe del proyecto, que normalmente es la figura que tendrá que dar cuenta cuando aumente la presión del proyecto al acercarse las fechas límite. Sería muy importante para él tener a mano mecanismos que le permitan medir el avance del proyecto y a la vez trabajar en la detección y corrección tempranera de problemas.

La prueba de *software* realmente pertenece a un conjunto más grande de actividades conocido como **verificación y validación** (**VyV**), actividades que a la vez pertenecen al **aseguramiento de la calidad** del *software*, tema que se verá más a fondo en capítulos posteriores.

Se habla de **verificación** cuando se quiere asegurar que la implementación del *software* se realizó correctamente. La **validación** lo que permite es saber si el *software* desarrollado satisface realmente lo que el cliente espera de él.

La existencia de esta etapa muchas veces lleva a los desarrolladores a limitar esfuerzos en cuando al aseguramiento de la calidad, y lleva todo a un "embudo de revisión" al tener finalizadas varias etapas dentro de una misma iteración o si se quiere al término de varias iteraciones.

No se debe pensar que las pruebas son una red de seguridad que atrapará todos los errores provocados por prácticas débiles dentro del proceso de ingeniería de *software* aplicado. No es posible probar la calidad; si no está ahí antes de que empiece la prueba, no estará ahí cuando termine.

Otro aspecto importante que resalta el libro de texto es la experiencia que pasa el o los desarrolladores del *software* cuando se habla del tema de la prueba. De alguna manera, pensar en que alguien pruebe algo que uno desarrolló significa para su creador algún nivel de desconfianza sobre su trabajo y la calidad asociada. Pero es inevitable trabajarlo de esta manera. Siempre es necesario, para eliminar el conflicto de intereses, que alguien más revise el trabajo desarrollado y lo trate de *estresar*[1] o llevar al extremo, para encontrar esos errores tan difíciles de alcanzar antes de la puesta a producción.

Para proyectos grandes, normalmente se tiene un **grupo independiente de prueba** (GIP), el cual debe estar muy de la mano de los desarrolladores para alcanzar pruebas de calidad y de alto grado de satisfacción. Este GIP es parte del proyecto de desarrollo de *software* y realmente viene desde las etapas de análisis y diseño, con las pruebas correspondientes para cada etapa.

La **estrategia de pruebas** por seguir, para los casos en donde se va a utilizar **una arquitectura convencional**, es muy similar al modelo de la espiral, visto en capítulos anteriores.

El libro trata de establecer una relación directa entre este modelo y la estrategia por seguir, tal como en la figura 03. La **prueba de unidad** comienza con el vértice de la espiral y se concentra en cada componente, tal y como se implementó en el código fuente.

Luego sigue la **prueba de integración** donde se atiende el diseño y la construcción hasta llegar a la **prueba de validación**, donde se validan los requisitos establecidos como parte del análisis, comparándolos con el *software* construido. Para finalizar, llegan las **pruebas de sistema**, donde se prueba el *software* como un todo (ingeniería de *software*) y otros elementos del sistema a un nivel de contexto.

Cada vez que se lleva a cabo una iteración se ensancha el alcance de la prueba, reforzando cada vez más la calidad en cada una de las etapas que conforman el *software*.

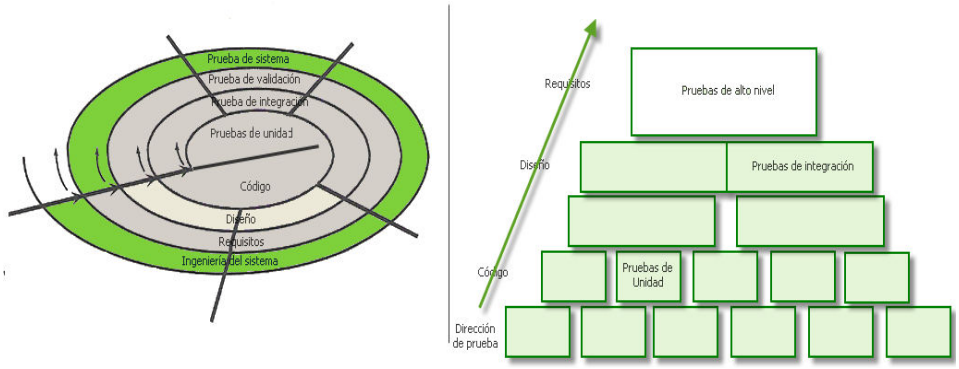


Figura 03 – Estrategia de prueba para arquitectura de* software *convencional.

[1] Significa llevar al límite las capacidades definidas para un sistema; esto permite probar ampliamente su estabilidad.

Por otro lado, la **estrategia de pruebas** a seguir para los casos en donde se va a utilizar **una arquitectura orientada a objetos** es muy similar a la estrategia convencional, mas no igual en su enfoque.

En el enfoque convencional, cuando la prueba es pequeña, el elemento más pequeño es el **módulo individual**, en cambio en el enfoque orientado a objetos se hace referencia a las **clases** y las **operaciones** asociadas, que necesitan de la comunicación y colaboración entre ella y otras clases.

Esta relación entre clases obliga a incluir dentro de las pruebas una de tipo **regresión**, para descubrir errores debido al cambio entre la composición de las clases y su interacción con las que esté asociada dicha clase.

El momento en que deben **terminarse las actividades de prueba** es difícil de determinar, ya que realmente nunca se terminan de dar. Cada vez que el usuario utiliza la aplicación, ya después de implementado el sistema en producción, es implícitamente un ejercicio que prueba la aplicación, a ver qué tan válido es su comportamiento con el esperado por el usuario final.

Sin embargo, lo que sí se puede lograr, según el texto guía, es aplicar ciertas métricas que hagan uso de modelos probabilísticos que nos ayuden a despejar las fallas en rangos muy elevados (cercanos al 95%), antes de entregar el *software* para producción.

> 🕮 **Actividad**:
>
> Investigue qué tipo de métricas o modelos estadísticos existen en el mercado para disminuir al máximo el grado de fallas por encontrar en las aplicaciones de *software*.
>
> Analice qué tipo de producto se estaría introduciendo en el mercado con la aplicación de dichos modelos y cómo sería la misma aplicación, pero sin ellos. Describa las diferencias.

Aspectos estratégicos

Existen algunos temas sugeridos en el libro de referencia que indican cómo desarrollar una estrategia de pruebas con éxito.

> *Importante:*
>
> Pasos para desarrollar una **estrategia de pruebas con éxito**:
>
> 1. Especificar los requisitos del producto de manera cuantificable mucho antes de que empiecen las pruebas.
> 2. Establecer explícitamente los objetivos de la prueba.
> 3. Comprender cuáles son los usuarios del *software* y desarrollar un perfil para cada categoría de usuario.
> 4. Desarrollar un plan de prueba que destaque la prueba del ciclo rápido.
> 5. Construir un *software* robusto diseñado para probarse a sí mismo.
> 6. Usar revisiones formales técnicas y efectivas como filtro previo a la prueba (capítulo 26).
> 7. Realizar revisiones técnicas formales para evaluar la estrategia de prueba y los propios casos de prueba.

8. Desarrollar un enfoque de mejora continua para el proceso de prueba.

Estas directrices permiten un marco robusto para pensar en una estrategia de pruebas más consistente y alejada de fallas.

Estrategias de prueba para el *software* convencional

Existen varias estrategias, seguidas por los equipos de desarrollo de *software*, para llevar a cabo las pruebas necesarias para evaluar la aplicación desarrollada.

No necesariamente todas las estrategias dan los resultados esperados, sobre todo cuando se escoge la opción de probar todo al final, como si se tratara de una actividad desarrollada en una sola etapa.

La estrategia más acertada, sugiere el libro de referencia, es optar por un **enfoque incremental**; se inicia con las pruebas de unidades individuales, luego las pruebas diseñadas para facilitar la integración de dichas unidades, y por último las pruebas sobre el sistema completo.

La prueba que inicia este proceso es la **prueba de unidad**. A continuación se describe puntualmente la parte de la estrategia sugerida para esta etapa.

El libro de texto indica que la prueba de unidad se basa, tal y como se mencionó anteriormente, sobre la unidad mínima dentro del diseño de *software*: el **componente** o **módulo** del sistema. Estas pruebas se deben concentrar en la lógica del procesamiento interno y en las estructuras de datos dentro de los límites impuestos a los componentes.

Además, explica que dentro de las **consideraciones necesarias** tenemos que atacar temas como la interfaz del módulo de prueba, examinar las estructuras de datos, recorrer todos los caminos independientes en las estructuras y probar las condiciones límite para asegurar que el módulo opera correctamente.

Las **pruebas selectivas de las rutas de ejecución** son vitales de revisar. Es preciso evitar errores debidos a cálculos incorrectos, comparaciones erróneas o flujos de control inapropiados.

Por otro lado, hace mención a un aspecto muy importante dentro de la programación: que normalmente se malentiende su incorporación cuando se relaciona con los casos de pruebas. Hablamos del famoso **control de errores**, ya que en muchos casos, debido a su existencia, no se realizan las pruebas como deben ser, valiéndose de la existencia de dicho control a nivel de código, pues se espera que todo se atrape por ahí.

📖 **Actividad**:

Consiga el código de un programa que esté en operación o que esté al menos en etapa de pruebas. El código debe ser de un lenguaje de programación orientado a objetos, preferiblemente C++ o Java.

Ejercite lo aprendido en la lectura. Ejecute todas las revisiones mencionadas, asociadas con la estrategia de pruebas. Siga la pista a algunos flujos de datos, obligue a un segmento del código a salir por el control de errores y contraste lo leído contra lo observado en dicha actividad.

Cuando se habla de los **procedimientos de prueba de unidad**, el libro sugiere trabajar con la creación de *software* controlador, de resguardo o ambos dependiendo del caso. Ver figura 04.

El *software* **controlador** es el que permite encapsular al módulo que se ha de probar, de manera tal que acepte los datos de prueba, los pase al componente y luego imprime los datos de importancia de dicha prueba.

El *software* de **resguardo** usa la interfaz del módulo subordinado, realiza una mínima manipulación de datos, proporciona verificación de entrada y devuelve el control al módulo de prueba, o sea cierra el ciclo **programa principal – componente – resguardo**.

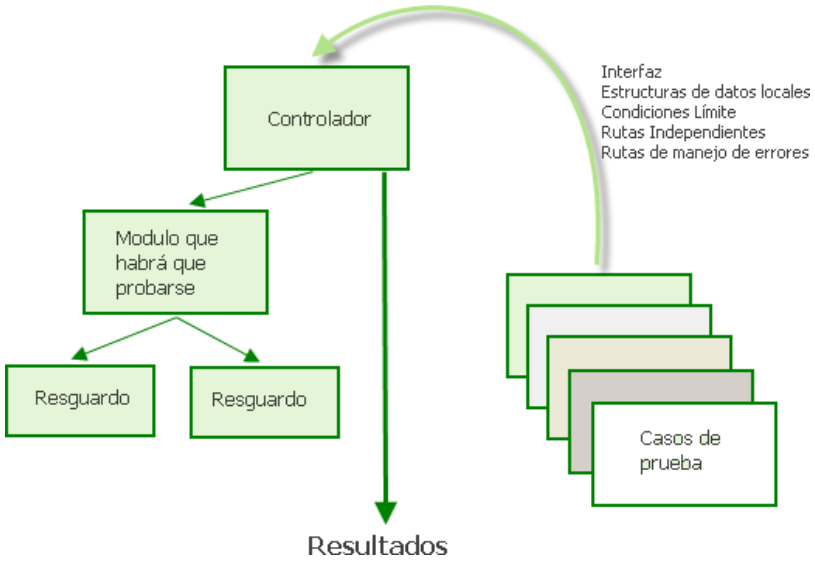


Figura 04 – Entorno de pruebas de unidad.

Como segunda etapa dentro del ciclo de pruebas está la **prueba de integración**. Estas pruebas son una técnica sistemática para construir la integración del *software*. Además, contienen una actividad que corre paralela y permite definir las pruebas por realizar para detectar errores asociados con la interfaz. Estos errores podrían ser de diversos tipos, por ejemplo que la combinación de subfunciones no llegue a producir la función esperada, o que el efecto de un módulo tenga un efecto adverso al esperado.

La mejor estrategia o forma de atacar esta etapa de integración es indicada por el libro de texto como la estrategia **incremental**, que permite avanzar mediante pequeños incrementos y facilita la aplicación de un enfoque de prueba sistemática. Además expone algunas de las estrategias incrementales conocidas como más importantes.

La primera estrategia mencionada es la de **integración descendente**, que permite a los módulos integrarse al descender por la jerarquía de control, empezando con el módulo de control principal, y luego los módulos subordinados se incorporan en la estructura de una de dos maneras: **primero en profundidad** o **primero en anchura**.

La siguiente estrategia tratada es la de **integración ascendente**. Ésta empieza la construcción y la prueba con módulos atómicos, luego sigue en su ascenso hasta el principal. Debido a esta naturaleza, no necesita la creación de resguardos.

Otra de las estrategias mencionadas es la **prueba de regresión**, donde la idea es que paulatinamente cada vez que se agrega un nuevo módulo a la revisión, se debe repetir el mismo subconjunto de pruebas que ya se habían aplicado para eliminar posibles efectos colaterales.

Ya como última propuesta tenemos la estrategia incremental de la **prueba de humo**, que no es más que la evaluación con **frecuencia definida**, para todos y cada uno de los componentes que forman la solución de *software*. Esta estrategia es de gran interés para proyectos a nivel crítico, que no desean dejar escapar ni el menor de los detalles alcanzable, pues le acerca a los jefes de proyecto una evaluación realista de lo acontecido.

> 🕮 **Actividad**:
>
> Entre las páginas del libro de texto 395 – 400 se encuentra una descripción más detalla de cada una de las **estrategias incrementales** que se mencionan en párrafos anteriores. Lea y analice esta información. Obtenga conclusiones sobre la existencia de cada una de estas estrategias y las diferentes necesidades para las pruebas en el mundo del desarrollo de *software*. Exponga sus conclusiones en un pequeño ensayo.

La existencia de estrategias incrementales, tanto ascendentes como descendentes, nos lleva a pensar en la siguiente pregunta: ¿cuál de estas dos será más eficiente? La necesidad de crear resguardos en las estrategias descendentes, como la necesidad de terminar todos los módulos para poder tener el programa completo en la estrategia ascendente, descalifica a ambas opciones como soluciones óptimas y más bien provoca pensar en una solución que amague ambas propuestas, enfoque que se conoce como **prueba sándwich**.

Como producto o entregable de esta etapa de pruebas se debe obtener un documento llamado **especificación de la prueba**. Este documento debe contener un plan general para la integración del *software* junto con una descripción de pruebas específicas más un procedimiento de prueba.

> 🕮 **Actividad**:
>
> En la página del libro de texto 401 encontrará una descripción detallada de la **documentación de la prueba de integración**, que se menciona en el párrafo anterior. Lea y analice esta información: ¿qué utilidad tiene para usted esta información?, ¿cómo la usaría?, ¿qué necesidades le permite solventar?

Estrategias de prueba para el *software* orientado a objetos

La naturaleza de la **estrategia de pruebas de unidad** para los casos en donde se va a utilizar **una arquitectura orientada a objetos** es muy similar a la estrategia convencional, mas como ya dijimos no es igual en su enfoque, y por lo tanto cambia la estrategia y la táctica.

Para el enfoque convencional, el elemento más pequeño es el **módulo individual**, en cambio en el enfoque orientado a objetos, hacemos referencia a las **clases**, **objetos** (instancias de la clase) y las **operaciones** asociadas, que necesitan de la comunicación y colaboración entre ella y otras clases.

Esta situación pone a las operaciones dentro de la clase como las unidades más pequeñas de prueba para el ambiente orientado a objetos. Entonces, la revisión de estas unidades se va a comparar con la

revisión del detalle algorítmico de un módulo y en los datos que fluyen por su interfaz, para un ambiente convencional. En síntesis, la **prueba de clase** para el *software* orientado a objetos es la equivalente a la **prueba de unidad** en el *software* convencional.

Para las **pruebas de integración** hay limitantes en cuando a las estrategias indicadas para la parte convencional, porque las estrategias de integración ascendente y descendente no trabajan bien en ambientes donde no exista un control jerárquico, como sí ocurre en el ambiente tradicional, orientado al flujo.

El libro de texto presenta, en cambio, dos estrategias un poco diferentes para atacar la integración en ambientes orientados a objetos. La primera es la prueba **basada en subprocesos**, la cual integra un conjunto de clases requerido para responder a una entrada o a un evento del sistema.

La segunda estrategia, es la prueba **basada en el uso**, donde la primera parte es empezar la construcción del sistema con las pruebas de esas clases (llamadas clases **independientes**) que usan muy pocas clases de servidor. Después de que se aprueban las clases independientes, se prueban las clases **dependientes**, que usan las clases independientes.

También es válido para las pruebas de integración el uso de controladores y resguardos para ambientes orientados a objetos.

Pruebas de validación

En párrafos anteriores se había explicado el concepto de validación dentro del ambiente de pruebas de *software*. Un *software* es validado cuando el cliente está razonablemente satisfecho por lo que realiza, basado en lo que dice el documento de la etapa de análisis llamado **especificación de requisitos de *software*.**

El plan de pruebas debe contener todo lo necesario para demostrar que los requerimientos funcionales se alcanzaron, y que las características de comportamiento, requisitos de desempeño, documentación y facilidad de uso están cubiertos también.

Cuando se desarrolla *software* para un cliente, se debe aplicar las **pruebas de aceptación** que, mediante unas pruebas planeadas y ejecutadas de manera sistemática, indican que el *software* está bien.

Cuando se está desarrollando *software* para muchos clientes es importante que cambie el enfoque y se piense en pruebas especiales, llamadas alfa y beta. La prueba **alfa** consiste en un grupo de usuarios que llevará a cabo las pruebas, pero en el **ambiente natural del desarrollador** de manera controlada. En cambio, la prueba **beta** consiste, en un grupo de usuarios que llevará a cabo las pruebas, pero ahora en el **ambiente natural del usuario final**, de manera no controlada para el desarrollador.

Pruebas del sistema

En este momento ya se entiende que el *software* ha sufrido las revisiones tanto de unidad, como de integración y validación. Las **pruebas del sistema** abarcan un plano mayor de elementos por considerar, pues involucran aspectos como *hardware* y personas, entre otros. La idea de las pruebas de sistema es ejercitar profundamente el sistema de cómputo. Existen diferentes tipos de pruebas de sistema, los cuales vamos a mencionar a continuación:

Importante:

Tipos de pruebas de sistema:

1. **De recuperación**: obligan al *software* a fallar de varias maneras y a verificar que la recuperación se realice satisfactoriamente, ya sea automática o manual.
2. **De seguridad**: comprueban que los mecanismos de protección integrados en el sistema realmente lo protejan de interrupciones inapropiadas.
3. **De resistencia**: ejecutan el sistema de manera tal que requiera una cantidad, una frecuencia o un volumen anormal de recursos.
4. **De desempeño**: diseñadas para probar el desempeño del *software* en tiempo de ejecución dentro del contexto del sistema integrado. Normalmente se vinculan con pruebas de **resistencia** y suelen requerir instrumentación de *software* y *hardware*.

El arte de la depuración

La ejecución satisfactoria del plan de prueba realizado es lo que da origen a la **depuración del *software***. Se dice de esta manera porque la naturaleza de la prueba es la de encontrar errores en la aplicación revisada. Otra forma de ver la depuración es la de tratarlo como el proceso que conecta un **síntoma** con una **causa**.

La aplicación de la depuración a los resultados de las pruebas permite corregir los errores encontrados y los que no; le permiten al ingeniero a cargo crear un nuevo plan de pruebas (casos de prueba) enfocado directamente en el síntoma que se sospecha pero que todavía no tiene solución.

El tema de la depuración es complejo, está más asociado a temas de la psicología humana que a la tecnología del *software*. Existen algunas características de los errores que ayudan a entender porqué se dan:

Importante:

Características de errores por depurar:

1. El síntoma y la causa pueden estar separados geográficamente.
2. Es posible que el síntoma desaparezca (temporalmente) al corregir otro error.
3. Es probable que el síntoma no lo cause algún error (por ejemplo inexactitud o redondeo de cifras).
4. El síntoma podría deberse a un error humano difícil de localizar.
5. El síntoma podría deberse a problemas de tiempo y no de procesamiento.
6. Tal vez sea difícil reproducir con exactitud las condiciones de entrada (aplicaciones en tiempo real).
7. El síntoma podría presentarse de manera intermitente.
8. Probablemente el síntoma se debe a causas distribuidas entre varias tareas que se ejecutan en diferentes procesadores.

Al mencionar el libro de texto que normalmente el tema de la depuración está muy asociado a la parte humana, lo hace basándose en estudios que demuestran que programadores con educación y experiencia parecida, demuestran tener diferentes habilidades con respecto a la depuración.

Sin embargo, a falta de capacidades innatas, la existencia de algunos enfoques facilita la capacidad de aprender a depurar. He aquí algunas estrategias.

La primera táctica de depuración que se va a mencionar, se conoce como **fuerza bruta**; es tal vez el método más común y menos eficiente. Consiste en dejar gran cantidad de marcas en tiempo de ejecución y luego se carga el programa con instrucciones de salida. Se espera a que el mismo sistema termine expulsando la falla y listo, a corregir.

El **rastreo hacia atrás** es otra técnica, algo menos común que la de fuerza bruta. Inicia en el sitio en donde se ha descubierto un síntoma, luego se recorre hacia atrás el código fuente (manualmente) hasta hallar el sitio de la causa.

El último enfoque es el de **participación binaria**. Se toman los datos asociados con el error y se clasifican para intentar aislar las causas. Con una "hipótesis" de la causa se usan los datos para probar dicha hipótesis. Luego en caso positivo de dicha valoración, se refinan de nuevo los datos valiosos y se vuelve a intentar aislar el error.

> 🕮 **Actividad**:
>
> En las páginas 413 – 414 del libro de texto, se encuentra una descripción de la depuración, tipo automática. Lea sobre este tipo de apoyo en la depuración de los errores, para entender cómo la informática apoya también esta actividad del ingeniero de *software*. Elabore una lista propia de pasos por seguir para conseguir tal depuración.

Ejercicios sugeridos

En la página 416 del libro de texto encontrará algunos problemas y puntos por considerar sobre el capítulo que acaba de estudiar.

Es importante que intente resolver los siguientes ejercicios. Esto le permitirá medir su aprendizaje.

61. Cuando el proyecto es grande, ¿cuáles deberían ser los lineamientos por seguir para garantizar que el equipo de pruebas va a llevar a cabo su trabajo de manera exitosa e imparcial?

62. ¿Por qué es difícil aplicar pruebas de unidad a un módulo altamente acoplado?

63. Describa con sus propias palabras qué es verificación y validación.

64. Lea la siguiente premisa: "*No es posible probar la calidad, si no está ahí antes de que empiece la prueba, no estará ahí cuando termine*". ¿Por qué considera usted que esto se afirma de la calidad?

65. ¿Existe algún modelo de desarrollo de sistemas al que le sea favorable realizar todas las pruebas hasta el final del proceso?

Capítulo

14

Técnicas de pruebas del *software*

"Solo hay una regla para diseñar casos de prueba: abarcar todas las funciones, pero no diseñar demasiados casos." ***Tsuneo Yamaura***

Sumario

Los temas que se tratan en este capítulo son:

- Fundamentos de las pruebas de *software*
- Técnicas de caja negra y caja blanca para pruebas de *software*
- Métodos de pruebas orientados a objetos y orientados al nivel de clase
- Diseños de casos de prueba de interclase
- Pruebas de entornos especializados: arquitecturas y aplicaciones

Localización de subtemas en el libro de texto

SUBTEMAS	PÁGINAS
Fundamentos de las pruebas de *software*	419
Técnicas de caja negra y caja blanca para pruebas de *software*	422
Métodos de pruebas orientados a objetos y orientados al nivel de clase	441
Diseños de casos de prueba de interclase	449
Pruebas de entornos especializados: arquitecturas y aplicaciones	452

Comentarios generales

Tener el código desarrollado según las especificaciones recibidas en el análisis y diseño llena de satisfacción a todas las partes involucradas, tanto a los desarrolladores del producto como a clientes y usuarios finales que esperan beneficiarse en alguna medida. No es para menos, después de tan ardua y desgastante labor para ambas partes.

Pero cabe acá una pregunta importante: ¿opera el programa de manera tal, que satisfaga las necesidades del cliente tanto en el aspecto funcional, como en la calidad de sus resultados?

Aunque es posible intentar la instalación y utilización del sistema para que el usuario final realice algunas pruebas sueltas, solo se tiene una manera seria y responsable de dar una respuesta aceptable, y esa forma no es otra que la de establecer un **mecanismo** para aplicar **diferentes técnicas de prueba** al *software* que se acaba de construir.

Este capítulo busca ese objetivo: describir las diferentes consideraciones técnicas que se deben seguir cuando se trata de montar un diseño de casos de prueba que permita al usuario clarificar que las interacciones que va a llevar contra el sistema, le van a devolver el resultado que espera y con la calidad necesaria.

Desarrollo

Fundamentos de las pruebas del *software*

El fundamento base de las pruebas es **encontrar errores** en la aplicación por revisar. Estas pruebas deben tener elementos en sus características que permitan encontrar errores, y a la vez esto se realice con el mínimo esfuerzo y con facilidad.

Algunas características de este tipo pueden ser las siguientes:

Importante:

Características importantes para la creación de *software* con facilidad para **pruebas**:

1. **Operatividad**: cuanto mejor funcione, con mayor eficiencia podrá probarse.
2. **Observabilidad**: lo que se ve es lo que se prueba.
3. **Controlabilidad**: cuanto mejor se controle el *software*, mejor se automatizarán y se mejorarán las pruebas.
4. **Capacidad para descomponer**: con esto se aislarán los problemas más rápidamente y se facilitará la opción de probar de manera independiente los módulos.
5. **Simplicidad**: cuanto menos haya que probar más rápido se hará.
6. **Estabilidad**: cuantos menos cambios haya, menores alteraciones habrá en la prueba.
7. **Facilidad de comprensión**: cuanta mayor información se tenga, con mayor inteligencia se hará la prueba.

Además, las pruebas deben contener **atributos** que faciliten su ejecución, tales como:

- Elevada probabilidad de encontrar un error.
- No ser redundante.
- Debe ser "la mejor de su clase".
- No debe ser ni muy simple ni demasiado compleja.

Técnicas de caja negra y caja blanca para pruebas de *software*

En términos generales, se puede decir que las pruebas de caja negra se realizan desde afuera del *software* y las pruebas de caja blanca se efectúan desde adentro.

Se va a iniciar con la descripción de las pruebas de caja blanca y las diferentes técnicas asociadas a ellas, tales como la de ruta básica y la de estructuras de control.

Las pruebas de **caja blanca** buscan explorar la parte **procedimental** del *software*, con esto se busca probar las **rutas lógicas** dentro del código y la **colaboración** entre componentes.

La **prueba de la ruta básica** es una técnica que permite al diseñador de los casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental, y usarla como guía para definir un conjunto básico de rutas de ejecución.

El libro de texto explica la notación de **gráfica de flujo** que sirve para representar el flujo de control de la prueba. Cada construcción estructurada tiene su representación en esta gráfica.

> 🕮 **Actividad**:
>
> En las páginas 423 – 424 del libro de texto se da la explicación de cómo está conformada **la notación de gráfica de flujo** y cómo se lleva a cabo su aplicación, al representar el diseño procedimental con **gráfica de flujo**, a partir de un diagrama (Figuras 14.1, 14.2a y 14.2b del libro de texto).
>
> Lea y analice tanto el texto descriptivo asociado, como la gráfica de flujo y el diagrama de flujo. Observe el mapeo que corresponde en cada caso y relaciónelo mentalmente con el código imaginario que origina esta secuencia dentro del flujo.

Otro concepto importante dentro del tema de la ruta básica es la **ruta independiente**. Este concepto se relaciona con cualquier ruta del programa que ingresa por lo menos un nuevo conjunto de instrucciones de procesamiento o una nueva condición.

Desde el punto de vista de la gráfica de flujo (figura 05), significa que una **ruta independiente** debe recorrer por lo menos una arista que no se haya recorrido anteriormente.

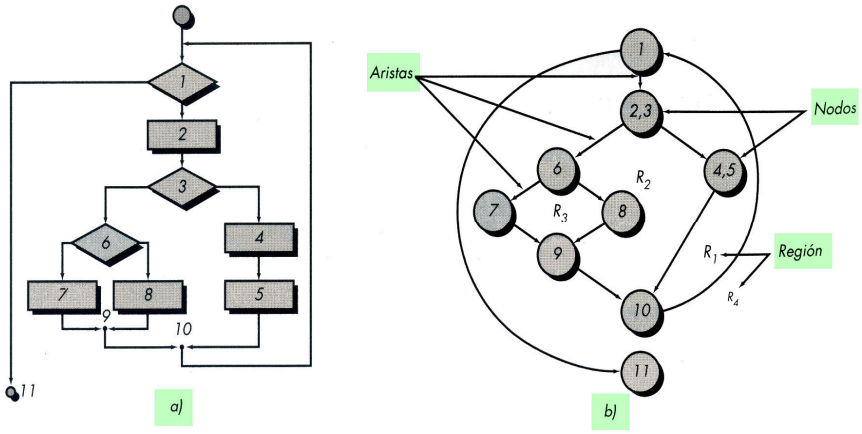


Figura 05 –Diagrama de flujo y gráfica de flujo.

La suma de una nueva arista para un camino ya definido crea lo que se conoce como el **conjunto básico** de rutas independientes. Este conjunto básico es casi imposible de completar para cualquier diseño procedimental existente de ciertas proporciones, ya que existen *N* cantidad de posibles relaciones que harían excesivamente grandes las pruebas para todos los casos definidos.

La métrica de *software* llamada **complejidad ciclomática** permite definir una medida cuantitativa de la complejidad lógica de un programa, y con esto resolver el problema anterior.

Esta métrica proporciona el límite superior del número necesario de casos de prueba (rutas independientes encontradas) para garantizar que cada instrucción del programa se haya ejecutado por lo menos una vez.

> 🕮 **Actividad**:
>
> En las páginas 426 – 427 del libro de texto se da la explicación de cómo está conformado el método de complejidad ciclomática. Analice las diferentes maneras que soporta esta métrica y evalúe manualmente, si lo expuesto ahí es correcto para la gráfica 14.2b.

Como ya se mencionó, el método de la ruta básica está asociado con la parte procedimental del código fuente de un programa. Se va a describir ahora la ruta básica vista como una serie de pasos por realizar para derivar el conjunto básico:

> *Importante:*
>
> Pasos a realizar para derivar el **conjunto básico** de la **ruta de pruebas**:
>
> 1. Utilizar el diseño o el código como base para dibujar la gráfica de flujo correspondiente.
> 2. Determinar la complejidad ciclomática de la gráfica de flujo resultante.
> 3. Determinar el conjunto básico de rutas linealmente independientes.
> 4. Preparar los casos de pruebas que forzaran la ejecución de cada ruta en el conjunto básico.

> 🕮 **Actividad**:
>
> En las páginas 427 – 429 del libro de texto se tiene la explicación, mediante un ejercicio, de todos los pasos descritos en el cuadro anterior. Ejecute y compruebe usted este ejercicio para confirmar la creación del conjunto básico.

Existen otras pruebas de estructuras de control diferentes a las pruebas de la ruta básica, que ayudan a expandir la cobertura de las pruebas y mejoran la calidad de las mismas. La prueba de **condición** es uno de estos métodos de diseño para casos de prueba que ejercitan las condiciones lógicas (simples o compuestas) a nivel de código en algún módulo.

Otro tipo de prueba es la del **flujo de datos** la cual selecciona rutas de prueba en un programa de acuerdo con las ubicaciones de las definiciones y los usos de las variables en el programa.

Por último, se tienen las pruebas de **bucle**, que buscan recorrer y validar la construcción de cada bucle en un segmento del *software*. Dentro de este grupo de pruebas, se han definido 4 clases de bucles: **simples**, **concatenados**, **anidados** y los **no estructurados**.

Cada una de estas pruebas basadas en estructuras de control tiene todo un conjunto de reglas para llevar a cabo su implementación. Sería importante que revise las páginas correspondientes del libro de texto (páginas de la 431 a la 433), o que lleve a cabo una investigación aparte para ampliar su comprensión.

Hasta este momento, todo ha girado en torno de las pruebas de caja blanca. Ahora se describirán a fondo las pruebas de **caja negra**.

Las pruebas de **caja negra** consisten en la evaluación de la **interfaz** del *software*. Se relacionan directamente con la **parte funcional** del sistema, sin importar su constitución interna.

Estas pruebas complementan las de caja blanca, y permiten aumentar las probabilidades de descubrir errores de los siguientes tipos:

- De interfaz.
- En estructuras de datos o en acceso a bases de datos externas.
- De comportamiento o de desempeño.
- De inicialización o de término.
- Funciones incorrectas o faltantes.

La ubicación de este tipo de pruebas es hasta el final (una vez construido el *software*), a diferencia de las pruebas de caja blanca que se dan al inicio del proceso de pruebas.

Los **métodos gráficos** de pruebas son un tipo de prueba de caja negra. Estos métodos permiten llevar a cabo pruebas de *software* a partir de la graficación de objetos y relaciones. Esto permite realizar una serie de pruebas para ejercitar cada objeto y sus relaciones y, a partir de esto, descubrir errores.

Antes de ingresar en el área de pruebas, es importante comprender cuáles son los objetos modelados y cuáles son las relaciones entre ellos.

De inicio, el ingeniero de *software* crea una gráfica en la cual incluye *nodos* (objetos), *enlaces* (relaciones), el *peso de nodo asociado* (propiedades de nodo) y el *peso de enlace* (propiedades de enlace). Al tener esto listo, se idean una serie de pruebas que cubren la gráfica desarrollada para ejercitar los objetos (y sus relaciones) y que salgan a flote posibles errores.

> 🕮 **Actividad**:
>
> En la página 435 del libro de texto se tiene la explicación de la notación gráfica mencionada mediante un ejercicio. Ejecute y compruebe este ejercicio mentalmente o con ayuda de papel y lápiz. Esto conviene para entender cuál es el alcance de este tipo de pruebas.

También se tienen algunos métodos de prueba de comportamiento que usan gráficas, tales como:

- El modelado del **flujo de transacción**.
- El modelado de **estado finito**.
- El modelado del **flujo de datos**.
- El modelado relacionado con el **tiempo**.

Siguiendo con el tema de cajas negras, el libro de texto menciona algunos métodos de caja negra importantes. El primer método es el de **partición equivalente**.

Este método busca dividir el dominio de entrada de un programa en **clases de datos** a partir de las cuales pueden derivarse casos de prueba. Con este método se busca definir un caso de prueba que descubra cierta clase de errores, y así reducir el número total de casos de prueba que deben desarrollarse.

Se menciona en el libro que la mayoría de los errores se presentan normalmente en los límites del dominio de entrada y no en el centro del dominio. El **análisis de valores límite** es una técnica que complementa la **participación equivalente** y que permite atacar esta área del dominio al llevar a una sección de casos de prueba los valores límite definidos. El análisis de **valores límite** extiende la partición de la participación equivalente al concentrarse en los datos de las **aristas** de una clase.

Otra prueba de caja negra que el libro recomienda es la de la **tabla ortogonal**. Esta tabla es útil en los casos donde se tienen problemas con dominio de entrada relativamente pequeño, pero a la vez demasiado grande para llevar a cabo una prueba de tipo exhaustiva.

> 🕮 **Actividad**:
>
> En las páginas 434 – 440 del libro de texto vienen claramente explicados los diferentes métodos de caja negra descritos en párrafos anteriores, así como el sustento matemático o científico para cada método. Además, se agregan ejemplos que clarifican los procesos. Lea estos pasos de implementación con los ejercicios dados para complementar la lectura de esta guía de estudio. Analice: ¿qué utilidad tiene para usted dicha información?, ¿cómo la usaría?

Métodos de pruebas orientadas a objetos y a nivel de clase

Las pruebas de sistemas **orientados a objetos** (OO) están muy relacionadas con el tema de las clases y la colaboración entre ellas. Cuando ya se tiene el código desarrollado, la prueba OO debe empezar por lo pequeño, con una serie de pruebas diseñadas para ejercitar las operaciones de clase y examinar si existen errores a medida que una clase colabora con la otra.

Además, las pruebas OO se deben concentrar en secuencias apropiadas de operación que permitan ejercitar los estados de una clase específica.

Algunos atributos asociados a la OO, aunque funcionan excelentemente dentro del ambiente de objetos, no facilitan las pruebas para este paradigma de desarrollo de sistemas. El **encapsulamiento**

por ejemplo, perjudica la realización de pruebas ya que dificulta obtener el estado concreto y abstracto de un objeto.

Por otro lado, la **herencia** también viene a complicar las pruebas para la OO. Se debe recordar que cada nuevo contexto de uso requiere una nueva prueba, y esto aumenta cuando se usa la herencia entre una superclase y las subclases definidas.

En ambiente OO se tiene un tipo de prueba que está **basada en fallas**, con el objetivo de diseñar pruebas que tengan una alta probabilidad de descubrir posibles fallas. Este tipo de prueba tiene validez solo cuando el estudio del análisis y del diseño existente arroja pruebas de que al forzar la implementación del sistema, este dará fallos que corregir.

Las **pruebas de integración** en el ambiente OO, se encargan de buscar fallas en llamadas a operación o en conexiones entre mensajes. Se han definido tres tipos de fallas dentro de este contexto:

- Resultado inesperado.
- Operación incorrecta/mensaje empleado.
- Invocación incorrecta.

El eje de las pruebas de integración tiene como fin determinar si existen errores directamente en el código que llama, no en el código que es llamado.

El libro de texto menciona la existencia de un enfoque de pruebas que es **basado en escenarios**. Este tiene la particularidad de que su enfoque está en el usuario y no en el producto. Este enfoque descubre errores de **interacción**, al ser basado en los casos de uso asociados a los actores que casi siempre ejecutan labores que involucran interacción entre varios subsistemas.

En el mundo de la OO, se define como **estructura de superficie** a la parte observable externamente para dicho sistema. Es importante denotar que la forma de probar estas estructuras es mediante objetos bien definidos por el encargado de pruebas, que permitan exponer dicha estructura y así encontrar tareas omitidas.

Por otro lado, la **estructura a fondo** representa los detalles técnicos de un programa OO, o sea, la estructura que se comprende al examinar el diseño, el código o ambos. La prueba asociada a esta estructura está definida para ejercitar dependencias, comportamientos y mecanismos de comunicación que se han establecido como parte del modelo de objetos.

Métodos de pruebas aplicables al nivel de clase

Las clases están compuestas por métodos y atributos que permiten definir tanto el estado de un objeto (instancia de la clase) como también su comportamiento.

Es importante entender que aunque van a existir métodos condicionantes y que siempre existirán en un orden establecido, se tienen otros métodos que podrán cambiar su orden de ejecución.

Esto acrecienta el problema de las pruebas, ya que hay muchas posibilidades de permutaciones entre estas clases y no se pueden probar todas.

La técnica de generar **listas de ejecución al azar** de los posibles métodos asociados a un caso particular es una buena técnica para darse cuenta de qué tan propensos son estos métodos de presentar fallas a futuro.

También la técnica de **partición al nivel de clase** apoya la realización acelerada de pruebas. La idea consiste en ordenar las clases de entrada y salida en categorías y se diseñan casos de prueba para ejercitarlas. Se tienen tres modalidades de partición:

- La basada en **estado**.
- La basada en **atributos**.
- La basada en **categorías**.

La de estado acomoda las operaciones de las clases según su capacidad para cambiar el estado de la clase. Las de atributo se encargan de ordenar las operaciones según el nivel de afectación a los atributos que utilizan, y la de categoría acomoda las operaciones de las clases según su función genérica.

Diseño de casos de pruebas de interclase

La relación interclase se presenta con fuerza cuando se está en la etapa de integración de un sistema que está orientado a objetos. Existen algunos pasos que sugiere el libro de texto para llevar a cabo los casos de **prueba aleatorios para las clases múltiples**.

> *Importante:*
>
> Secuencia de pasos para generar **casos de prueba aleatorios** para las clases **múltiples**:
>
> 1. En cada **clase cliente** use una **lista de operaciones** de clase para generar una secuencia de **pruebas aleatorias**.
> 2. En cada mensaje generado determine la **clase colaboradora** y la **operación correspondiente** en el **objeto servidor**.
> 3. En cada operación del **objeto servidor** determine los **mensajes** que transmite.
> 4. En cada uno de los mensajes determine el **siguiente nivel** de operaciones invocadas e incorpórelos en la secuencia de prueba.

El enfoque de clases múltiples es muy similar al de clases individuales. Se inicia con la **segregación** de una clase importante, pero se da una expansión de la misma hasta alcanzar a las demás clases mediante la mensajería que comparten entre sí.

El diagrama de estado de una clase proporciona al encargado de pruebas un modelo de **comportamiento** dinámico de la clase y de las clases que colaboran con ella. La idea de estas pruebas asociadas al comportamiento es cubrir todos los **estados permisibles**, tomando en cuenta las clases

que se vean involucradas y asignar así tantos diagramas de estado como sea necesario para cubrir dicho comportamiento.

Pruebas de entornos especializados: arquitecturas y aplicaciones

Todo lo relacionado con el tema de las pruebas estudiado hasta el momento puede ser asociado con casi cualquier sistema en diferentes entornos y arquitecturas. Pero existen algunos casos en donde tanto los entornos como las arquitecturas, tienen una concepción diferente a lo tradicional.

Esto requiere que se dé un enfoque especializado en las pruebas, de manera que permita que cada uno de estos casos se tome como casos aislados, y por lo tanto su tratamiento sea diferente. Tal es el caso de las interfaces gráficas de usuario, de arquitecturas tipo cliente/servidor, la documentación y las funciones de ayuda y, por último las pruebas de tiempo real.

> 🕮 **Actividad**:
>
> En las páginas 452 – 456 del libro de texto se explica puntualmente estas situaciones especiales para ciertos entornos y arquitecturas. Lea estas descripciones para complementar la lectura de esta guía de estudio. Exponga sus conclusiones en un pequeño ensayo del tema de entornos especializados.

Ejercicios sugeridos

En la página 459 del libro de texto encontrará algunos problemas y puntos por considerar acerca del capítulo que acaba de estudiar.

Es importante que intente resolver los siguientes ejercicios. Esto le permitirá medir su aprendizaje.

66. Indique por lo menos 3 ejemplos en los cuales la prueba de caja negra da la impresión de que "todo está bien", mientras que las pruebas de caja blanca indican al menos un error. Lleve a cabo el ejercicio también pero ahora de manera inversa entre la prueba de caja blanca y caja negra.

67. Seleccione un componente de *software* que se haya diseñado recientemente y aplíquele un conjunto de casos de prueba para asegurar que todas las instrucciones se hayan ejecutado con la prueba de la ruta básica.

68. En palabras propias, describa dentro de un sistema orientado a objetos ¿por qué la clase es la mejor unidad razonable para la prueba?

69. ¿La prueba exhaustiva para los sistemas pequeños garantiza que el programa es totalmente correcto?

70. ¿Por qué se tiene que volver a probar subclases que crean instancias a partir de clases existentes si esta ya se ha probado por completo?; ¿es posible usar los casos de prueba diseñados para la clase existente?

Referencias

Bruegge, B., Dutoit, A. (2002). Ingeniería de *software* orientado a objetos. México D.F.: Prentice Hall.

Larman, C. (2003). UML y Patrones. Segunda edición. Madrid: Prentice Hall.

Sommerville, I. (2002). Ingeniería de *software*. Sexta edición. México D.F.: Addison Wesley.

Glosario de términos

Arista: es, en geometría, el segmento de recta que se forma por la intersección de dos planos. También se conoce con este nombre al segmento común que tienen dos caras vecinas de un poliedro. Coloquialmente, se dice que las aristas de un problema son los bordes que hay que limar para solucionarlo.

C++: es un lenguaje de programación diseñado a mediados de los años 1980 por *Bjarne Stroustrup*. Con él se busca extender el lenguaje de programación C con mecanismos que permitan la manipulación de objetos. El C++ es un lenguaje híbrido; además, en este lenguaje se suman dos paradigmas: la programación estructurada y la orientada a objetos; por esto se suele decir que el C++ es un lenguaje multiparadigma. Actualmente existe un estándar, denominado ISO C++, al que se han adherido la mayoría de los fabricantes de compiladores más modernos.

Java: es un lenguaje de programación orientado a objetos desarrollado por *Sun Microsystems* a principios de los años 90. Java toma parte de su sintaxis de los lenguajes C y C++, pero su modelo de objetos es más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria.

Nodo: punto de intersección o unión de varios elementos que confluyen en el mismo lugar. Por ejemplo en una red de ordenadores cada una de las máquinas es un nodo, y si la red es Internet, cada servidor constituye también un nodo.

Orientación a objetos (OO): es el conjunto de disciplinas que desarrollan y modelan *software* para facilitar la construcción de sistemas complejos a partir de componentes.

Patrón de diseño (*design patterns*, en inglés): es una solución a un problema de diseño. Además, son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de *software* y otros ámbitos referentes al diseño de interacción o interfaces. Para que una solución sea considerada un patrón debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores y ser reusable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

Enlaces electrónicos relacionados con el tema

CAPÍTULO	ASPECTO	PÁGINA
12	Página que trata sobre diseño de interfaces	http://www.useit.com/
12	Página sobre patrones de diseño de interfaces de usuario	http://www.hcipatterns.org/tiki-index.php
13	Página sobre recursos útiles para pruebas de *software*	http://frank.mtsu.edu/~storm/
13	Otra página sobre recursos útiles para pruebas de *software*	http://www.sqatesters.com/
13	Nueva página sobre recursos para pruebas	http://www.io.com/~wazmo/qa/
14	Página que habla sobre la caracterización de la prueba de *software*	http://www.e-quallity.com.mx/articulos/SG-200505-Luis05.pdf
14	Página sobre la prueba de *software* de caja negra y caja blanca	http://lsi.ugr.es/~ig1/docis/pruso.pdf