



Universidad Autónoma de Querétaro
 Facultad de Informática
 Maestría en Ciencias Computacionales

Definición de Operaciones Unitarias en las actividades de análisis de la Ingeniería de Software

TESIS

Que como parte de los requisitos para obtener el grado de
 Maestro en Ciencias Computacionales

Presenta:


Antonio Vega Páez

Dirigida por:

M.C. Ruth Angélica Rico Hernández

SINODALES


M.C. Ruth Angélica Rico Hernández
 Presidente


 Firma

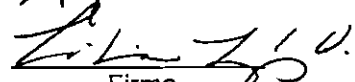
M.S.I. Gerardo Rodríguez Rojano
 Secretario


 Firma

M.C. Rosa María Romero González
 Vocal


 Firma

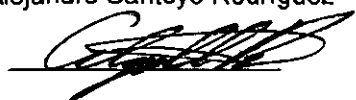
M.S.I. Lilia López Vallejo
 Suplente


 Firma

M.C. Vicente Rodríguez Hernández
 Suplente

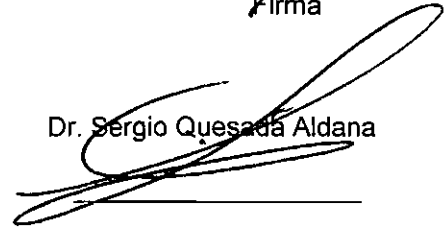

 Firma

Ing. Alejandro Santoyo Rodríguez



Director de la Facultad de Informática

Dr. Sergio Quesada Aldana



Director de Investigación y Posgrado

Centro Universitario
 Querétaro, Qro.
 Marzo de 2004
 México

No. Adj. 5469077

No. Title _____

Clas. TS

005.757

V 422 d

RESUMEN

La identificación y definición de operaciones unitarias en la fase de análisis del desarrollo de software se presentan en este trabajo de tesis, las operaciones unitarias descubiertas son: Extracción, Discriminación, Separación, Composición y Abstracción.

La identificación se realizó mediante una metodología y estudio cualitativo con un proceso de inducción, en el que se analizan como objetos de estudio, cuatro métodos de desarrollo de software: el Análisis y Diseño Estructurado, el Análisis y Diseño Orientado a Objetos, el Proceso Unificado de Desarrollo de Software y el Proceso Personal de Software.

En el mismo sentido se induce la existencia de dos operaciones compuestas: Descomposición Analítica y Composición Sintética que necesariamente se construyen a partir de las operaciones unitarias, de la misma forma se presentan la existencia y definición de los conceptos de Requerimiento Unitario, Diseño Unitario, Construcción Unitaria y Prueba Unitaria, para el caso de Requerimiento Unitario se presentan sus características.

Finalmente se concluye como la introducción de estos conceptos permitirán reducir la complejidad de los procesos de desarrollo de software y contribuyen a la solución al problema de la crisis del software.

Palabras Clave: Operaciones Unitarias, Requerimiento Unitario, Crisis de Software.

SUMMARY

This thesis work presents the identification and definition of unit operations in the analytical stage of software development, the discovered unit operations are: Extraction, Discrimination, Separation, Composition and Abstraction.

The identification was made through a qualitative methodology as well as qualitative study, within an induction process in which case four software development methods are used as study objects, those methods are: Structured Analysis and Design, Oriented Objects Analysis and Design, Unified Software Development Process and Personal Software Process.

In the same way the existence of two compound operations are introduced: Analytic Decomposition and Synthetic Composition, which are compounded from the unit operations. Also the existence and definition of Unit Requirement, Unit Design, Unit Construction and Unit Test concepts are presented. For the Unit Requirement case only its characteristics are presented.

Finally the conclusions present how the introduction of those concepts will allow to reduce the complexity of software development process and to contribute in the solution of the problem of the software crisis.

Keywords: Unit Operations, Unit Requirement, Software Crisis

*“Este fuego dotado de razón, que se
transforma en aire, agua, tierra y
vuelve a sí mismo por el camino
inverso, es la sustancia unitaria de
todas las cosas, por diversas que éstas
sean”*

**Heráclito de Efeso
Siglo V antes de Cristo.**

Esta tesis está dedicada a todos los analistas de sistemas,
que con toda seguridad más de una vez, en el trabajo
cotidiano de la vida real, han sufrido el hecho de que su
trabajo no esté del todo claramente definido y preciso.

De manera muy especial esta tesis está
dedicada al M.C. Moisés González García,
Profesor de nuestra Maestría y de quien es
la idea original de definir las Operaciones
Unitarias en la fase de análisis
de la Ingeniería de Software.

AGRADECIMIENTOS

Agradezco a mis Padres, por la fe que han depositada en mi, en el sentido que siempre cumpliré mis objetivos académicos y de vida.

Agradezco a mis hermanos Ignacio, Lupita, Patricia, Juan, Miguel, Paco y Pepe por su apoyo moral y aliento que se tradujeron en la culminación de este trabajo, así como sus acertados comentarios académicos durante el desarrollo del mismo.

Agradezco a mis Sinodales su invaluable disposición, el tiempo empleado y sus valiosas aportaciones, de manera muy especial agradezco a mi Directora de Tesis, la M.C. Ruth Angélica Rico Hernández, quien dirigió acertadamente las actividades y esfuerzo de este trabajo, con criterios firmemente orientados al logro.

Agradezco a mis compañeros de estudios, con quienes compartimos momentos gratificantes y también difíciles, pero que en el transcurso de los estudios solidificaron una preciada amistad.

ÍNDICE

	Página
Resumen	i
Summary	ii
Dedicatorias	iii
Agradecimientos	iv
Índice	v
Índice de cuadros	vi
Índice de figuras	vii
I. INTRODUCCIÓN	1
II. REVISIÓN DE LITERATURA	2
CRISIS DEL SOFTWARE.	2
SOLUCIÓN DE PROBLEMAS: DOMINIO DE LA INGENIERÍA DE SOFTWARE.	9
LOS MÉTODOS DE DESARROLLO DE SOFTWARE.	15
INTENTOS DE CLASIFICACIÓN DE MÉTODOS	19
ENCAPSULACIÓN DE LOS MÉTODOS	22
USO DE MÉTODOS Y HERRAMIENTAS	24
OPERACIONES UNITARIAS EN LA INGENIERÍA QUÍMICA.	26
EVOLUCIÓN DE LA INGENIERÍA DE SOFTWARE.	33
LA CHEMICAL ABSTRACT MACHINE (CHAM).	36
OPERACIONES UNITARIAS EN LA INGENIERÍA DE SOFTWARE.	38
LA FASE DE ANÁLISIS EN MÉTODOS TRADICIONALES.	53
<i>Análisis y Diseño Estructurado</i>	54
<i>Análisis y Diseño Orientado a Objetos</i>	64
<i>El Proceso Unificado de Desarrollo de Software.</i>	76
<i>Personal Software Process (PSP)</i>	81
III. METODOLOGÍA	90
HIPÓTESIS	90
PREGUNTAS FORMULADAS PARA LOS SUJETOS DE ESTUDIO:	91
RESPUESTAS EN EL ANÁLISIS Y DISEÑO ESTRUCTURADO.	92
RESPUESTAS EN EL ANÁLISIS ORIENTADO A OBJETOS.	93
RESPUESTAS EN EL PROCESO UNIFICADO DE DESARROLLO.	94
RESPUESTAS EN EL PSP	95
IV. RESULTADOS Y DISCUSIÓN	96
ELEMENTOS COINCIDENTES GENERALES:	96
ELEMENTOS COINCIDENTES EN LA FASE DE ANÁLISIS:	98
OPERACIONES UNITARIAS DE LA FASE DE ANÁLISIS	100
OPERACIONES COMPUESTAS DE LA FASE DE ANÁLISIS	103
DEFINICIÓN DE REQUERIMIENTO UNITARIO	105
CONCLUSIONES	107
REFERENCIAS BIBLIOGRÁFICAS:	110

ÍNDICE DE CUADROS

Cuadro	Página
Relación entre operaciones unitarias y atributos de calidad	49
Guión del Proceso PSP0	84
Guión de Planeación PSP0	85
Guión Postmortem PSP0	85
Guión de Desarrollo PSP0	86

ÍNDICE DE FIGURAS

Figura	Página
RESULTADOS DEL INFORME DEL GAO	5
RESULTADOS DEL INFORME CHAOS	6
PROCESO DE SOLUCIÓN DE PROBLEMAS DE LA INGENIERÍA	9
EVOLUCIÓN DE LOS MÉTODOS DE DESARROLLO DE SOFTWARE	17
PROCESO QUÍMICO INDUSTRIAL	29
FLUJO DE OPERACIONES UNITARIAS DE LA INGENIERÍA QUÍMICA	30
EJEMPLO DE OPERACIONES UNITARIAS DE LA INGENIERÍA QUÍMICA	31
PROCESO DE LA INGENIERÍA Y ARTESANÍA DEL SOFTWARE	36
ANALOGÍA DE LOS PROCESOS QUÍMICOS Y DE SOFTWARE	37
NO CONMUTATIVIDAD DE OPERACIONES UNITARIAS	51
FASES DEL ANÁLISIS Y DISEÑO ESTRUCTURADO	55
MODELOS DIAGRAMÁTICOS DEL ANÁLISIS ESTRUCTURADO	58
PRINCIPALES ACTIVIDADES DEL ANÁLISIS Y DISEÑO ESTRUCTURADO	59
FASES DEL PARADIGMA ORIENTADO A OBJETOS	64
MICROPROCESO DE DESARROLLO DEL ANÁLISIS ORIENTADO A OBJETOS	72
EL MACROPROCESO DE DESARROLLO DEL ANÁLISIS ORIENTADO A OBJETOS	74
INTERACCIONES Y FLUJO DE TRABAJO DEL PROCESO UNIFICADO	77
EL PROCESO UNIFICADO DIRIGIDO POR CASOS DE USO	78
ELEMENTOS DEL PROCESO PSP0	82
FLUJO DEL PROCESO PSP0.	83
MARCO DE TRABAJO DE LA FASE DE DISEÑO DEL PSP	87
EL CICLO REQUERIMIENTOS-ESPECIFICACIÓN-DISEÑO	86
OPERACIÓN UNITARIA DE EXTRACCIÓN	100
OPERACIÓN UNITARIA DE DISCRIMINACIÓN	101
OPERACIÓN UNITARIA DE SEPARACIÓN	101
OPERACIÓN UNITARIA DE COMPOSICIÓN	102
OPERACIÓN UNITARIA DE ABSTRACCIÓN	102
OPERACIÓN COMPUESTA DE DESCOMPOSICIÓN ANALÍTICA	103
OPERACIÓN COMPUESTA DE COMPOSICIÓN SINTÉTICA	104
CORRESPONDENCIA DE OPERACIONES UNITARIAS	105

I. INTRODUCCIÓN

El desarrollo en las últimas décadas de la Ingeniería de Software en el panorama mundial como disciplina formal, se encuentra en proceso incipiente de evolución, a diferencia de otras ramas de la Ingeniería, como lo es la Ingeniería Civil o la Ingeniería Química, disciplinas con mucho más presencia en la historia de la humanidad, el concepto de operaciones unitarias acuñado (Landau, 1997) por el Dr. Arthur D. Little del Massachusetts Institute of Technology (M.I.T.) en 1915, puso de relevancia que es necesario llegar hasta las operaciones básicas de cualquier actividad de la Ingeniería, con la finalidad de que su enseñanza, aplicación cotidiana y estandarización de las mismas, conduzcan el desarrollo, evolución y madurez de cualquier rama de la Ingeniería.

El presente trabajo pretende definir las Operaciones Unitarias para las actividades de análisis de la Ingeniería de Software, partiendo de la base histórica de cómo fueron definidas las operaciones unitarias en la Ingeniería Química y el impacto de su definición en el desarrollo de esta disciplina, se ve que en la literatura actual existe definición de operaciones unitarias en la fase de diseño de la Ingeniería de Software, en particular en las actividades del Arquitecto de Software y con la finalidad de asegurar requisitos de calidad (Bass, 1998).

Previo al trabajo de definición se propone la generalización de operaciones unitarias en cada una de las fases de desarrollo de software. Definidas finalmente en la fase de análisis, especificadas formalmente y sujetas a las bases científicas de las ciencias computacionales, se procederá a su estudio en cuatro métodos conocidos de desarrollo de software, el Análisis y Diseño Estructurado, el Análisis y Diseño Orientado a Objetos, el Proceso Unificado de Desarrollo de Software y el Proceso Personal de Software.

II. REVISIÓN DE LITERATURA

Crisis Del Software.

“Los puentes Romanos de la antigüedad fueron estructuras muy ineficientes para los estándares actuales, utilizaban principalmente piedra y se construían con muchísimo esfuerzo de mano de obra.

A través de los años hemos aprendido que para construir puentes más eficientemente, debemos de usar pocos materiales y menos mano de obra para realizar la misma tarea”.

Tom Clancy, La Suma de Todos los Miedos

Para tratar de colocarnos en el contexto de los inicios de la Ingeniería de Software, tratemos de imaginarnos por un momento una sala de un centro de computo a principios de los años 50 en algún organismo del gobierno de los Estados Unidos. El lugar está ocupado por una enorme computadora cuyo desarrollo ha costado varios millones de dólares y cuyo mantenimiento tiene ocupadas a varias personas las 24 horas del día.

La siguiente media hora es la sesión semanal de uno de los muchos científicos que trabajan para el Departamento de Defensa. Su trabajo consiste en calcular tablas numéricas para trayectorias balísticas usando ecuaciones relativamente complejas. Lleva toda la semana repasando su programa en su despacho para asegurarse que no contiene errores sintácticos ni semánticos, ya que un fallo en la compilación podría hacerle perder su preciosa media hora de máquina semanal.

Cuando comienza su tiempo, la computadora comienza a leer las tarjetas perforadas que previamente había entregado al operador, compila el programa sin errores y comienza su ejecución. El científico se dispone a esperar a que la impresora comience a imprimir las tablas. Cuando faltan sólo pocas líneas para

completarlas, el hardware falla y los encargados de mantenimiento deben parar la máquina para repararla. Quizá la semana que viene tenga más suerte.

Si comparamos esta situación con la actual, pocos son los aspectos comunes que podríamos encontrar. Sin embargo, podemos hacer un ejercicio de análisis e identificar algunas de las características de software en los comienzos de la informática.

- El hardware es mucho más caro que el software. La máquina y su mantenimiento cuestan millones de dólares. Si esto lo comparamos con el sueldo del científico que escribe el programa ¿para qué preocuparse por el costo del desarrollo de software?
- El desarrollo del hardware es más complejo que el del software. La tecnología hace que el hardware sea complejo de construir. El software habitual suelen ser programas no muy grandes (debido entre otras limitaciones, a la escasa capacidad de memoria) y suelen estar escritos por una única persona, normalmente empleada por la organización que utiliza el hardware. Los requerimientos que tiene que cumplir el software son simples. Por lo tanto, ¿por qué preocuparse por la complejidad del software?
- El hardware es poco fiable. Debido a la tecnología que se utiliza para su implementación, en cualquier momento la máquina sufre una avería, así que ¿para qué preocuparse por la calidad del software?

Esta despreocupada situación respecto al software cambia cuando gracias a los avances en la tecnología, aumenta la capacidad de memoria y se reducen los costos de desarrollo y mantenimiento del hardware. Se empiezan a comercializar las primeras computadoras y la demanda de software más complejo crece rápidamente, destapando la caja de Pandora de la crisis del software,

término utilizado por primera vez en la conferencia organizada por la Comisión de Ciencia de la OTAN en Alemania, en octubre de 1968, para designar la gran cantidad de errores que presentaba (y aún presenta) el desarrollo de software y alto índice de fracasos en los proyectos de desarrollo (Duran, 2000).

¿Qué podía hacerse ante una situación en la que los tenían un alto riesgo de fracasar? La respuesta parecía obvia producir software de forma similar a como se construye hardware, barcos, puentes o edificios, es decir, aplicar los métodos de ingeniería a la construcción de software. Esta decisión tuvo sus consecuencias inclusive en la forma en como se miden los esfuerzos de producción de cada tecnología, donde se partió de la base que la medida debía ser similar a la producción de software, situación que no es cierta (Shaw, 2002) aduciendo a un estudio donde se cumple la Ley de Moore en el crecimiento de la memoria basada en transistores o en dispositivos electrónicos, ley que no se cumple en líneas de código u otras medidas para medir la producción de software. Desde 1968 se ha invertido un gran esfuerzo en determinar las causas proponer soluciones para la crisis del software.

En 1979, la Oficina de Contabilidad del gobierno norteamericano (Government Account Office, GAO) realizó un estudio (GAO, 1979) seleccionando 9 proyectos de desarrollo de software para el gobierno norteamericano cuyos contratos sumaban una cantidad total de 6'800,000 dólares.

De esta cantidad, sólo 119,000 dólares correspondían a un proyecto que se había utilizado tal como se había entregado. Dicho proyecto se trataba de un pre-procesador de COBOL, por lo que era un problema relativamente simple cuyos requerimientos eran comprendidos por clientes y desarrolladores y que además no cambiaron durante el desarrollo.

El resto de los 6.8 millones de dólares se distribuyó como puede verse en la figura 1.1, en la que puede destacarse el enorme porcentaje de dinero invertido en proyectos cancelados o no satisfactorios.



RESULTADOS DEL INFORME DEL GAO

En 1995, el Grupo Standish realizó un estudio (el informe CHAOS) mucho más amplio y significativo que el del GAO cuyos resultados, a pesar de haber pasado más de 25 años, no reflejaban una mejoría sustancial (TSG, 1995).

Los resultados generales, que pueden verse en la figura 1.2, al compararse con los de (GAO, 1979) presentan una mejora en los proyectos que se entregan cumpliendo todos sus requerimientos, 2% frente al 16.2% (sólo el 9% en grandes compañías), pero empeoran ligeramente respecto a los que se abandonan durante el desarrollo, 28.7% frente a 31.1%.

Sin incluir al 16.2% de los proyectos terminados correctamente, la media del gasto final fue del 189% del presupuesto original, el tiempo necesario para su realización del 222% del plazo original y se cumplieron una media del 61% de los requerimientos iniciales, cifras que también empeoraban en el caso de grandes compañías.



RESULTADOS DEL INFORME CHAOS

Las encuestas realizadas a los directores de los proyectos que participaron en el estudio indicaron que, en su opinión, los tres principales factores de éxito eran:

1. Implicación de los usuarios
2. Apoyo de los directivos
3. Enunciado claro de los requerimientos

mientras que los tres principales factores de fracaso eran:

1. Falta de información por parte de los usuarios
2. Especificaciones y requerimientos incompletos
3. Especificaciones y requerimientos cambiantes

En 1996, el proyecto ESPITI (European Software Process Improvement Training Initiative) (ESP, 1996) realizó una investigación sobre los principales problemas en el desarrollo de software en el continente europeo. Los resultados, muy similares a los obtenidos en el informe CHAOS, indicaron que los mayores problemas estaban también relacionados con la especificación, la gestión y la documentación de los requerimientos.

Estos informes ponen de manifiesto el hecho de que, a pesar de que las herramientas para construir software han evolucionado enormemente, se sigue produciendo software que no es satisfactorio para los clientes usuarios. Esto indica que los principales problemas que han dado origen la crisis del software residen en las primeras etapas del desarrollo, en particular en el análisis, cuando hay que decidir las características del producto software a desarrollar.

En palabras de F. P. Brooks (Brooks, 1995):

“La parte más difícil de construir de un sistema software es decidir qué construir. (...) Ninguna otra parte del trabajo afecta más negativamente al sistema final si se realiza de manera incorrecta. Ninguna otra parte es más difícil de rectificar después.”

Otro hecho comprobado es que el costo de un cambio en los requerimientos, una vez entregado el producto, es entre 60 y 100 veces superior al costo que hubiera representado el mismo cambio durante las fases iniciales de desarrollo, por lo que no es de extrañar que aquellos proyecto en los que no se determinan correctamente los requerimientos y cambian frecuentemente durante el desarrollo, superen con creces su presupuesto inicial.

Todas estas circunstancias han convencido a la gran parte de la comunidad de la ingeniería del software de la necesidad, cada vez mayor de una evolución de las fases iniciales, esfuerzos interesantes encontramos en la evolución de la fase de análisis a una fase propiamente de ingeniería de requerimientos.

El hecho de que el Software Engineering Institute de la Universidad de Carnegie Mellon haya identificado la gestión de requerimientos como una de las áreas de proceso clave (key process area) dentro del nivel 2 (Repetible) del Capability Maturity Model (CMM) (Paulk *et al*, 1993).

La organización de congresos específicos como:

- El IEEE International Symposium on Requirements Engineering (RE), que se celebra los años impares desde 1993 organizado por IEEE, ACM, IFIP y varias asociaciones más.
- La IEEE International Conference on Requirements Engineering (ICRE), que se celebra los años pares desde 1994 organizado por el IEEE.
- El Workshop em Engenharia de Requerimientos (WER), que se celebra anualmente desde 1998.

La publicación de revistas especializadas como el Requirements Engineering Journal, que se publica trimestralmente desde 1996.

La aparición bianual de números monográficos sobre ingeniería de requerimientos en IEEE Software coincidiendo con la celebración del ICRE, en concreto los correspondientes a los meses de marzo de los años 1994, 1996 y 1998 y el del mes de mayo del año 2000.

El financiamiento público de proyectos europeos como:

- NATURE (Novel Approaches to Theories Underlying Requirements Engineering).
- REAIMS (Requirements Engineering Adaptation and Improvement for Safety and dependability).
- CREWS (Cooperative Requirements Engineering With Scenarios)

La red europea RENOIR (Requirements Engineering Network Of International cooperating Research groups)

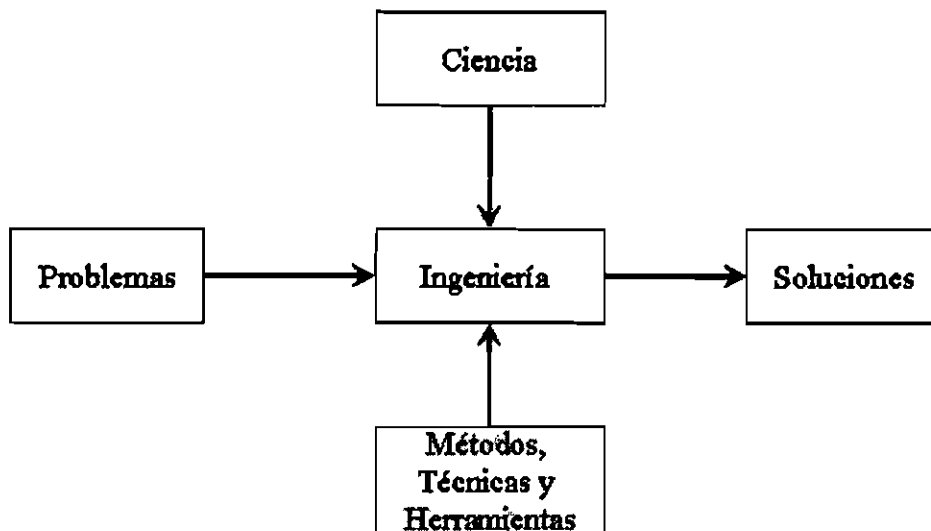
Una vez asumida la necesidad de la formalización del análisis y su evolución hacia una ingeniería de requerimientos robusta, el reto al que se

enfrenta la comunidad investigadora es identificar claramente los problemas a los que se enfrenta y dotar a dicha ingeniería de las herramientas teóricas y prácticas necesarias para solucionarlos, teniendo en cuenta la dificultad añadida de lo costoso de la obtención de datos empíricos tanto en ingeniería del software en general, como en ingeniería de requerimientos en particular.

Solución de Problemas: Dominio de la Ingeniería de Software.

Es importante hacer notar que el dominio de la Ingeniería de software, es la solución de problemas, como las demás ingenierías, pero la particularidad de esta Ingeniería es que la solución es una combinación de tecnología electrónica (hardware) y software, considerado desde este punto de vista como programas almacenados con un fin específico.

El trabajo de un ingeniero estará definido por el tipo de problemas a los que se enfrenta y el ámbito en el que se desenvuelve (Grech, 2001), así encontraremos ingenieros en diferentes campos de acción, cada uno de los cuales se presentan diferentes tipos de problemas, a los que los ingenieros deben hacerles frente. En el tratamiento de los problemas no debe importar la especialidad del ingeniero.



En términos generales se ha establecido un proceso de actuación de los ingenieros ante el tratamiento de los problemas que se les presentan en su actuar diario. A continuación se presentan los componentes de dicho proceso, se debe tener en consideración que los temas presentados son enunciativos y que cada profesional de la ingeniería tiene su propio estilo para tratar los problemas, por lo que a continuación sólo se presentan los pasos que se han observado en la actuación de un ingeniero ante un problema de su campo de acción.

Proceso de actuación del ingeniero ante un problema

1. Definición del problema a resolver
2. Criterios y restricciones en el problema
3. Obtención de información
4. Generación de posibles soluciones
5. Descarte de las soluciones no viables
6. Selección de la mejor solución
7. Implementación y pruebas
8. Análisis de resultados y corrección de actividades
9. Especificaciones de la solución encontrada
10. Documentación y comunicación de la solución

Es obvio que el quehacer diario del ingeniero es resolver problemas, aspecto que desarrolla por medio de la ciencia y la tecnología, así que sus conocimientos (teóricos y prácticos) los aplica en cada uno de los pasos mencionados en el proceso antes presentado.

Definición del problema

Es muy importante destacar que en la ingeniería la solución de los problemas empieza por definir, conocer y entender en todas sus magnitudes lo que se pretende solucionar. Definir los problemas implica análisis y reflexión sobre

su magnitud, alcances y consecuencias, y esto a su vez requiere de conocimientos especializados o aprendizaje profundo sobre lo que significa el problema.

Pareciera una perogrullada el establecer que es necesario conocer en todo lo posible un problema, para solucionarlo de la mejor manera, pero resulta que en muchas ocasiones los problemas parecen simples y en realidad no lo son, o también resulta que las implicaciones de las soluciones requieren de una gran precisión sin posibilidades de falla. Esto obliga a buscar soluciones de gran efectividad, con márgenes de error muy estrictos.

Entender en todas sus magnitudes lo que se debe solucionar, permite a la ingeniería actuar de manera abierta, lo que implica poder buscar varias soluciones al mismo problema, característica fundamental de la ingeniería.

Para la definición de los problemas a solucionar, el ingeniero se debe partir de la respuesta a la pregunta: ¿qué se espera obtener al solucionar el problema?

Criterios y restricciones del problema

El conocer cuáles son los límites dentro de los cuales se puede aplicar la solución de un problema ahorra mucho esfuerzo y permite concentrarse mejor en las posibles soluciones. Las restricciones deben ser consideradas como parte del problema e incluso deben ser ponderadas en qué tanto influyen en la solución que se puede implantar.

Los criterios y restricciones pueden ser de varios tipos, es muy difícil hacer una lista o agrupar por categorías, algunas de las más comunes pueden ser económicas, técnicas, de seguridad, de tiempo, legales, de cobertura y otras muchas. Se podría decir que cada problema tiene sus propias restricciones y éstas se ven influenciadas por diferentes variables del contexto, la época, el clima o cualquier cosa.

Esto nos permite afirmar que cada problema tiene su propio referente, con sus restricciones o peculiaridades.

Obtención de la información

Para poder analizar de manera objetiva y efectiva cualquier problema es necesario contar con toda la información que permita conocer y las experiencias que otros profesionales de la ingeniería o de otros ámbitos. Cada problema implica una investigación de lo que ya se sabe o lo que no se ha podido conocer.

Obtener la información que complementa los datos del problema y lo que se conoce del mismo puede ahorrar mucho tiempo, además de que proporciona o elimina rutas a seguir para obtener una solución viable.

Entre mayor información, ordenada, clasificada y certificada se tenga del problema se tendrá mayor seguridad en que la solución que se adopte será la mejor y en el caso de ya haber sido experimentada o probada, conocer los resultados y las posibles fallas.

Generación de posibles soluciones

Las soluciones a un problema no son únicas, ya que un problema puede ser resuelto de muchas maneras, por ello en cada problema se deben tratar de identificar el mayor número de soluciones posibles, entre más posibilidades de solución se tengan a la mano, mayor será la gama de posibilidad de análisis y evaluación de las alternativas, las que filtradas por los criterios y restricciones no permitirán evaluar su factibilidad.

Eliminación de soluciones no factibles

De acuerdo a los resultados esperados al solucionar el problema, a las restricciones o limitaciones impuestas o necesarias, a la información recabada y a

la serie de posibles soluciones, se tendrán elementos suficientes para poder eliminar las soluciones que no son adecuadas.

Eliminar soluciones implica no gastar tiempo, recursos o el ánimo en acciones que no van a cumplir con su cometido. Sin embargo saber que existen dichas soluciones nos permite recurrir a su análisis las veces que sea necesario, pues el no haberlas utilizado no significa haberlas borrado o tirado a la basura.

Selección de la mejor solución

Una vez que se conocen las posibles soluciones a un problema de ellas se selecciona la que más cumpla con los resultados esperados y que siga los parámetros y restricciones establecidos. En esta solución se deberán invertir recursos de todo tipo, además de cuidar de manera objetiva que en realidad cumpla con las necesidades esperadas.

Implementación y pruebas

La solución seleccionada deberá ser implementada y además probada de manera estricta para conocer si cumple con las restricciones y resuelve el tan citado problema. Tanto la implementación como las pruebas deberán hacerse de manera estricta y profesional y la evaluación de los resultados no deberá perder objetividad.

Análisis de pruebas y resultados

Con las pruebas de la solución implementada, se deberá establecer si los resultados fueron los esperados o el tipo de correcciones, ajustes o modificación que requiere la prueba de la solución seleccionada. Incluso hay ocasiones en las que se debe cambiar de solución seleccionada. Este es un proceso de retroalimentación que no permite seguir adelante hasta que no se tiene la seguridad de que la solución seleccionada es la adecuada.

Especificaciones de la solución encontrada

Una vez que la solución haya cumplido con todo lo esperado se deberán obtener todas sus especificaciones y características, de tal manera que pueda ser explicada y replicada por otra persona en otro lugar.

Documentación y comunicación de la solución

Con las especificaciones y características de la solución implementada, se deberá documentar y comunicar a quien solicitó la solución. Esto puede significar presentaciones especiales o recurrir a oficinas de patentes y marcas.

Un aspecto que los ingenieros deben tener siempre en consideración es la aplicación de los criterios de la producción económica, lo que implica que las soluciones implementadas y adoptadas deberán tener en consideración los aspectos económicos, de rentabilidad y mercado.

Los ingenieros, como cualquier otro profesional de un campo determinado, deben ser efectivos en su proceder, lo que dependerá de sus habilidades para resolver los problemas a los que se enfrenta en su vida diaria. Algunas de las acciones que aumentan la efectividad de un ingeniero son:

Es común que las soluciones del ingeniero se fundamenten en los conocimientos teóricos y prácticos con los que se ha formado, estos conocimientos, su creatividad e ingenio son la materia prima que les permiten a estos profesionistas cumplir con su misión en el tratamiento de algunos problemas que se presentan en diferentes ámbitos de la sociedad. Una vez que el ingeniero define un camino a seguir para solucionar un problema determinado, es necesario que su saber se pruebe y registre para que pueda ser replicado por otros a los que se les puede presentar el mismo problema o por aquellos que se están informando para solucionar otros problemas similares. Para comprobar la efectividad de una solución y al mismo tiempo registrar todo aquello que sucede

mientras se prueban las soluciones, es factible utilizar el método científico, por ello es recomendable tener presente siempre sus componentes:

1. Identificación del problema o anomalía
2. Recolección de datos significativos que lo describan
3. Análisis de la información obtenida
4. Elaboración o definición de una hipótesis
5. Predicción de eventos futuros con base en la hipótesis
6. Elaboración de experimentos para comprobar y negar la hipótesis
7. Modificación de la hipótesis y repetición de los pasos anteriores, hasta que haya variación de la misma.
8. Establecer conclusiones sobre el comportamiento del problema o anomalía. En algunas ocasiones esto se llama convertir una hipótesis en una teoría.

Finalmente es necesario documentar los resultados y divulgación de los resultados. El actuar del ingeniero está influenciado por muchas variables pues su campo de acción es siempre variable, pero una sola constante deberá siempre tenerse presente, esta es la ética con la que se debe ejercer la ingeniería, en la que el respeto a los semejantes debe sobresalir sobre toda la ciencia y la tecnología que se pueda aplicar.

Los métodos de desarrollo de Software.

Un método de ingeniería de software es un enfoque estructurado para el desarrollo de software cuyo propósito es facilitar la producción de software de alta calidad de una forma costeable. Métodos como Análisis Estructurado que emergió en 1978 (DeMarco, 1983) y en 1983 (Jackson, 1988) surgió una versión refinada del JSD (Jackson System Development), fueron los primeros desarrollados en los años 70'S. Estos métodos intentaron identificar los componentes funcionales básicos de un sistema de tal forma que los métodos orientados a funciones aún

se utilizan ampliamente. Entre los años 80 y 90, estos métodos orientados a funciones fueron complementados por métodos orientados a objetos, como los propuestos por Grady Booch en 1994 (Booch, 1996) y Rumbaugh (Rumbaugh, *et al*, 1991). Estos diferentes enfoques se han integrado a un solo enfoque unificado, basado en UML (Lenguaje de Modelado Unificado) (Fowler y Scott, 1997); (Booch *et al*, 1999); (Rumbaugh *et al*, 1999).

Todos los métodos se basan en la idea de modelos gráficos de desarrollo de un sistema y en el uso de estos modelos como un sistema de especificación o diseño. Los métodos incluyen una variedad de componentes diferentes, los que podemos considerar básicos son:

Descripciones del modelo del sistema

Reglas

Recomendaciones

Guías en el proceso.

No existe un método ideal y métodos diferentes tienen diferentes áreas donde son aplicables.

con los más particulares procedimientos (más subjetivo) o proceso (más objetivo), y todos ellos más acabados y externos que proyecto.

Volviendo al rigor definitorio del método, cualquier actuación humana que quiera transformar un espacio-problema acotado de realidad, tiene tres grandes hitos de plasmación: original, modelo y sistema. El original, circunstancia de realidad concreta y punto de partida natural de la actuación, se ha de captar por el actor en forma de una o varias representaciones (complementarias o sucesivas). La representación abstracta del original que cubre la misma circunstancia se llama su modelo (simbólico): su inmaterialidad es esencial, por hacerlo más manejable que el original y tiene otras ventajas (economía, predicibilidad, seguridad, controlabilidad y/o diseñabilidad de alternativas). La generación de posibles modelos sucesivos (cada uno es el original del siguiente) desemboca en una última representación del original llamada sistema, que esta vez es un modelo no simbólico, sino material, o sea una nueva circunstancia de realidad concreta y punto de llegada artificial de la actuación. La transitividad de las sucesivas semejanzas asegura cierta semejanza, lo menos degradada posible, entre el original inicial y el sistema final, situados ambos en el espacio real.

Cada uno de los modelos intermedios podrá ser dinámico o estático (se considere o no su variación con el tiempo), determinista o aleatorio (según su grado de incertidumbre asociada al entorno), puramente descriptivo (textos, diagramas, procedimientos en lenguaje natural, escrito o hablado) o analítico (de comportamiento ante estímulos del exterior, de evaluación de costos o eficiencias de interrelación de tiempos o movimientos). Los modelos simbólicos se emplean sistemáticamente en ingeniería mucho antes de usarse para producir software, y buscan siempre un equilibrio entre su semejanza al original y ser más manejables (en tamaño y/o complejidad) que éste (por descomposición / recomposición y particularización / generalización).

Un método es la secuencia de modelados que ayuda a construir, a partir del original de la realidad, una o varias cadenas de modelos, derivados unos de otros, con el objetivo de lograr un modelo material final o sistema que concrete la nueva circunstancia de realidad deseada en relación con el original. El paso del original a su primer modelo implica una reducción, simplificación inevitable sólo aceptable si conserva sus principales elementos y relaciones: un buen método no debe hacer más reducciones en los pasos a modelos sucesivos. En sentido estricto, metodología es el discurso, ciencia o estudio de los métodos. Lo anterior cubre toda acción sobre la circunstancia de realidad llamada información, desde sus concepciones más clásicas, como la de Shannon, hasta sus enfoques recientes, como el de Booch, en la orientación a objetos y el de Jacobson en el OMT.

El espacio-problema está anclado en algún sitio del mundo real, y el espacio-de-soluciones se implementa por una combinación de hardware y software. Si las soluciones distan del espacio-problema, habrá que hacer una transformación mental o física a representaciones abstractas del mundo real. La abstracción ayuda a mantener y entender sistemas. Reduciendo los detalles que un programador necesita para conocer un nivel dado. Recuérdese que los teóricos de ingeniería de software usan el concepto de espacio en su sentido matemático como un conjunto de valores de variables de dos tipos: las variables de estado son endógenas (no relacionadas con el entorno) y sus valores definen en cada instante el comportamiento del modelo; las variables de acción regulan la transición del estado en un instante al sucesivo, y el correlativo cambio de comportamiento del original, modelo o sistema.

Intentos de Clasificación de Métodos

Mientras que, por ejemplo, los lenguajes de programación o las bases de datos tienen amplios procesos de estandarización, sea de jure o de facto, los

métodos han tenido básicamente hasta ahora estándares de facto (los pocos de jure, como las prenormas IEEE-PI074 012207.2 ISO/IEC.JTC1.SC7 sobre procesos de ciclo de vida del software, se desconocen fuera de círculos estrechos). Es difícil así la elección de los métodos más aptos para distintos problemas (no todos valen para lo mismo), pero, en contrapartida, no suelen ser propietarios, a pesar de intentarse por la vía de las herramientas que los soportan.

A falta de estándares sobre métodos, se han editado todo tipo de clasificaciones. Una de las más recientes es la Comisión Europea para preparar Euro método, partiendo de extensas encuestas sobre contenido y uso de métodos, emplea en la veintena de los analizados hasta cinco enfoques comparativos, pero acaba por reconocer que no bastan para evaluarlos o armonizarlos:

- Enfoque histórico busca familias, pero no aclara sus ventajas relativas;
- Estudio-del-caso no los ilustra todos, aunque tiene algún éxito;
- Marcos de referencia emplea criterios subjetivos y no independientes;
- Orientado-a-problemas clasifica los métodos de acuerdo a los problemas que resuelve,
- Enfoque empírico, que intenta evaluarlos con puntos pragmáticos.
- También se ha intentado una división cada vez menos discriminante entre métodos cartesianos y sistémicos:
 - Cartesianos construyen el sistema como lo ven los utilizadores (parten de las funciones requeridas y la información que las asegure);
 - Sistémicos arrancan de un modelo consistente de la realidad organizativa basado en la teoría de sistemas (con niveles de circuitos de operación, gestión y decisión conectados por

flujos de información) y van descendiendo al sistema con modelos cada vez menos abstractos.

Otras divisiones combinan dos grupos de enfoques:

- Marco de procedimientos poco formalizables de construcción de modelos (como la conducción de reuniones o el análisis semántico del discurso).
- Los tres grandes tipos de ciclos: de vida o creación del sistema; de abstracción, con sus niveles clásicos conceptual (Que), lógico (Quién, Dónde, Cuándo, Cuánto) y físico (Cómo),
- De decisión, con la sucesión de tareas preparatorias y concomitantes para tomar decisiones de gestión y de calidad en el proyecto.

Pero los tratados siguen dando vigencia a la tradicional clasificación taxonómica, que no sólo sitúa intuitivamente cada método por el papel estructurante otorgado a alguno de sus componentes, sino que ofrece ciertas explicaciones evolutivas. Un primer esbozo del método, lo suele situar por su orientación primaria a los tratamientos, resultados, datos, objetos, sucesos, actividades, productos (y últimamente a objetivos o bien a Actores):

La orientación tradicional e intuitiva a tratamientos o funciones empieza por identificar en la realidad las funciones a informatizar.

La orientación a resultados parte de lo que piden los usuarios (volatilidad) para deducir los datos de entrada y los tratamientos; se suele emplear implícitamente por los que niegan usar métodos y por los programadores en entorno microinformático, de generadores o de los llamados lenguajes de cuarta generación (L4G, conocido también por sus siglas inglesas, 4GL)

La orientación a datos los articula en entidades y relaciones, deduciendo los procesos que los emplean y los resultados obtenibles.

La orientación a objetos asume un enfoque mixto de entidades (datos) y de los tratamientos que los afectan.

La orientación a sucesos que disparan los tratamientos los transmite por mensajes a los objetos concernidos.

La orientación a actividades, reivindicada por meta métodos como el Modelo V (Stein, 1996) , centra la adaptación de éste a una clase específica de proyectos en las decisiones sobre dichas actividades.

La orientación a productos, reivindicada por peri métodos como Euro método, parte de los entregables requeridos por el cliente para organizar los productos proporcionables por el método proveedor enlazado.

Encapsulación de los Métodos

Los métodos se empiezan a encapsular con una panoplia de otros dispositivos abstractos, por emplear el paradigma de los que rodean el microprocesador central de un computador personal: la placa-rack de soporte sería una meta método, mientras que la carcasa envolvente comprendería un frontal de peri métodos (como Euro método, que cubre el arranque contractual, supervisa la producción y cierra la relación) y los laterales, cubiertos con mecanismos paralelos al desarrollo, como los mencionados para gestionar calidad, configuración, seguridad o reparto, o bien extensores a enfoques y/o sectores complementarios o suplementarios al clásico núcleo de gestión de organizaciones, como el tiempo real, la ayuda a decisiones o la ingeniería de conocimiento.

El soporte meta metódico aporta la estructuración decisiva de los conceptos y relaciones básicos subyacentes a una gran mayoría de métodos, sean tradicionales o nuevos. Así, el Vorgehensmodell (Stein, 1996) modelo avanzado o Modelo V, adoptado por la Administración Pública alemana y componente del núcleo metodológico de Euro método, articula en su entorno sendos sub modelos de desarrollo de productos/resultados y para la gestión del proyecto, su calidad y su configuración, y define en su interno las posibles actividades de un tipo abstracto de proyectos, con los cambios de estado que se determinan en los resultados. A cada actividad del entorno y del interno, el Modelo V asigna unos estados y unos responsables abstractos con funciones definidas (Gernert & Ahrend, 2002). Por cierto, la articulación de esas actividades, tareas, estados, roles, resultados y productos tiene una infraestructura conceptual similar a la que subyace al modelo de gestión de flujos de trabajo (Work Flow Management), ligado, por otra parte, a la reingeniería de procesos de organizaciones (Business Process Reengineering) y al paradigma groupware (lo cual es bastante lógico, porque ¿no es el desarrollo y adaptación de un software un típico trabajo de reingeniería en equipo?).

Las interfaces peri metódica, como Euro método, se centran en la construcción de una estrategia, o sea una forma de articular los modelos intermedios que permita acercarse óptimamente al objetivo de conseguir el sistema deseado. La estrategia parte de la situación del problema, un primer modelo del espacio-problema de realidad original formado por un conjunto de factores que caracterizan su complejidad y su incertidumbre (el grado de conocimiento sobre su funcionamiento y objetivos). La orientación a contingencias lleva a analizar, valorar y prever la reducción de los riesgos de no conseguir el sistema. La estrategia se concreta en un plan de entregas formado por sucesivos hitos de decisión e intercambio de los productos procedentes del método concreto de desarrollo encapsulado.

La articulación de los dispositivos encapsuladores con los proyectos de desarrollo concretos, realizable en diversos grados y formas (directamente o por medio de métodos preestablecidos), promete ser uno de los aspectos más importantes de investigación metodológica de problemas inmaduros que requieren nuevas vías de avance y solución.

Uso de Métodos y Herramientas

Los citados estudios preparatorios de Euro método concluyen que los métodos se emplean relativamente poco (aún conociendo sus beneficios esperables de mantenibilidad, calidad, adaptación, comunicación y control); se eligen más por limitaciones fácticas (hardware, política organizativa, conocimientos, cercanía del consultor) que por criterios técnicos (adaptabilidad, simplicidad); no se suelen aprender a fondo ni aplicar rigurosamente (se adaptan sobre la marcha en las fases de diseño e implementación; el análisis sigue siendo bastante informal). Reducen la satisfacción esperable de sus resultados poco objetivos y su uso complicado para los analistas, e impracticable para los usuarios (sin contar con sus deficiencias de flexibilidad, soporte informático y cubrimiento del ciclo con técnicas que interactúen).

Este panorama práctico de los métodos contrasta con un potencial, más que optimista, imprescindible, de ventajas inmediatas (o fines en sí) y mediatas (medios para otros fines). Las inmediatas mejoran (pero no garantizan) la identificación de problemas y errores al producir los sistemas, sobre todo los más complejos e inciertos. Las ventajas mediatas permiten entender rápida y establemente el problema, controlar con un plan riguroso y no redundante el desarrollo del proyecto, y prever cuando abandonarlo, si cabe. Estas ventajas, a menudo desestimadas con argumentos de costo (aunque desarrollar software sin borrador sea como edificar sin planos), se descubren a posteriori con unos gastos de corrección de los errores detectados en pruebas de aceptación final mucho mayores (pero a menudo emboscados) que los detectables por el análisis.

En cuanto a las herramientas informatizadoras de técnicas y métodos, su difusión relativamente reciente dificulta consideraciones ampliamente aceptadas sobre su clasificación y experiencia de uso (aunque haya enfoques importantes, recogidos, entre otros, por artículos especializados como los publicados por ACM/IEEE, y en España por el número 105 de Novática). Es innegable que la aplicabilidad de un método en problemas de cierta complejidad dependerá cada vez más de su soporte herramental, pero empieza a ser casi perogrullesco citar experiencias de lo contrario: departamentos informáticos que se lanzan a comprar herramientas, en general costosas, como sustituto y orillamiento de la maduración previa necesaria en el empleo de métodos (como un fabricante que piense que entrar en un negocio es sólo cuestión de comprar máquinas).

Operaciones Unitarias en la Ingeniería Química.

“El objetivo de este manual no es para habilitar a alguien para realizar un trabajo de carácter especial..., pero ilustra los principios por los cuales una planta química de cualquier clase, puede ser diseñada e instalada cuando las condiciones y requerimientos son conocidos con certeza. No podemos hacer el mejor uso de nuestras habilidades a no ser que hayamos aprendido a investigar los principios fundamentales de la construcción de los medios e instrumentos que necesitamos para trabajar”

George E. Davis

Handbook of Chemical Engineering

Manchester Technical School, 1901

Analizando precisamente la historia de la Ingeniería Química (Shaw, 1990), encontramos que en el año 1887 George E. Davis presenta una serie de doce lecturas sobre operaciones básicas de procesos químicos del "Manchester Technical School". En 1901 publica el manual titulado "Handbook of Chemical Engineering", en el cual se identifican operaciones básicas de la Química,

Estas operaciones básicas de la Ingeniería Química identificadas por Davis sirvieron de base para que el Dr. Arthur D. Little del Massachusetts Institute of Technology (M.I.T.) en 1915, acuñara el concepto "Operaciones Unitarias" en la Ingeniería Química (Landau, 1997). La definición dada entonces, fue la siguiente: *"Cualquier proceso químico en cualquier escala que se lleve a cabo, se puede descomponer en una serie coordinada de lo que pudieran llamarse operaciones unitarias, como: pulverización, mezclado, calentamiento, absorción, condensación,*

lixiviación, precipitación, secado, cristalización, filtración, disolución, electrólisis, evaporación, destilación, y así sucesivamente. El número de estas operaciones unitarias básicas no es muy grande, y relativamente, pocas se encuentran incluidas en un proceso determinado. La complejidad de la Ingeniería Química resulta precisamente de la variedad de las condiciones tales como la temperatura, presión, concentración, etc., bajo las cuales tienen que ser llevadas las operaciones unitarias en los diversos procesos. Así como de las limitaciones que ofrecen los materiales de construcción, así como del diseño que se impone a los aparatos debido a las características físicas y químicas de las sustancias reaccionantes" (Little, 1933).

La lista original de las Operaciones Unitarias incluía no más de quince acciones básicas. Desde entonces se han añadido otras, en la modesta proporción de unas cuantas durante años, aunque últimamente en forma más acelerada (Foust, 1961).

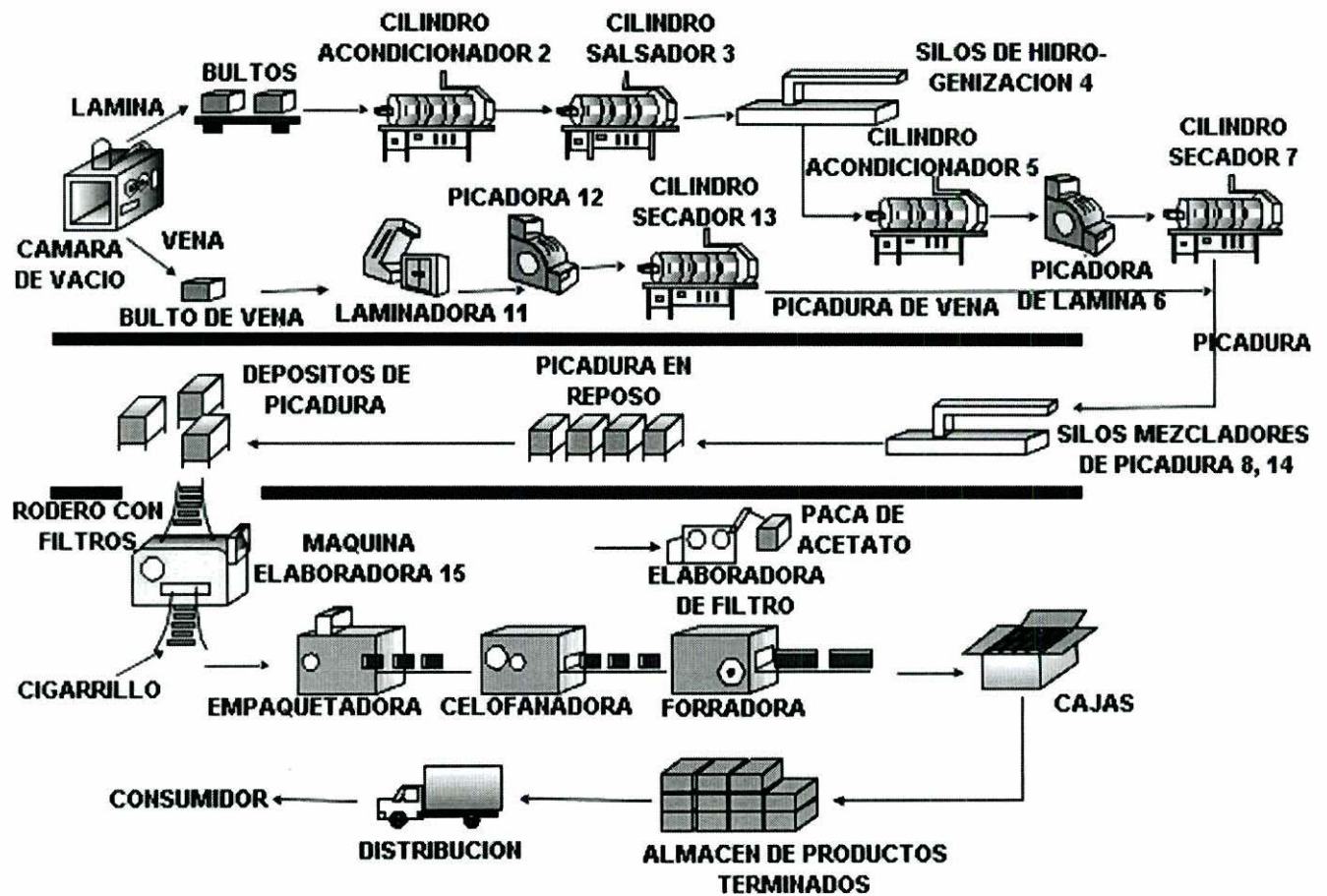
Con esta simplificación se redujo la complejidad del estudio de los procesos químicos industriales, pues del conjunto de todos los procesos químicos que pueden imaginarse bastará con estudiar el grupo de las 25 ó 30 operaciones unitarias existentes. Un proceso determinado será, por tanto, la combinación de operaciones unitarias (Landau, 1997).

El concepto tradicional de las Operaciones Unitarias, ha sido un factor predominante para el éxito sorprendente de los Ingenieros Químicos y la Ingeniería Química general. Éxito tenido a partir de la introducción de este concepto, obedeciendo a la certeza de que, es más eficiente para la enseñanza, más económico en tiempo, más adecuado en la presentación de lo fundamental, y más efectivo para el entrenamiento que conduzca a la definición y a la solución de los problemas amplios y vastos de los procesos químicos. El ingeniero químico instruido en estos lineamientos tendrá una sólida comprensión de los principios fundamentales encapsulados en operaciones básicas (Foust, 1961).

La teoría de las Operaciones Unitarias está basada en leyes completamente definidas y bien comprendidas, el trabajo del Ingeniero Químico precisa que tanto la teoría y la práctica, se encuentren en el mismo plano de importancia. Mucho del bagaje necesario para la comprensión de las operaciones unitarias, lo componen la ciencia e ingeniería elemental. Está basado este bagaje en las leyes fundamentales de la física, mecánica y ciencias similares (Badger, 1955).

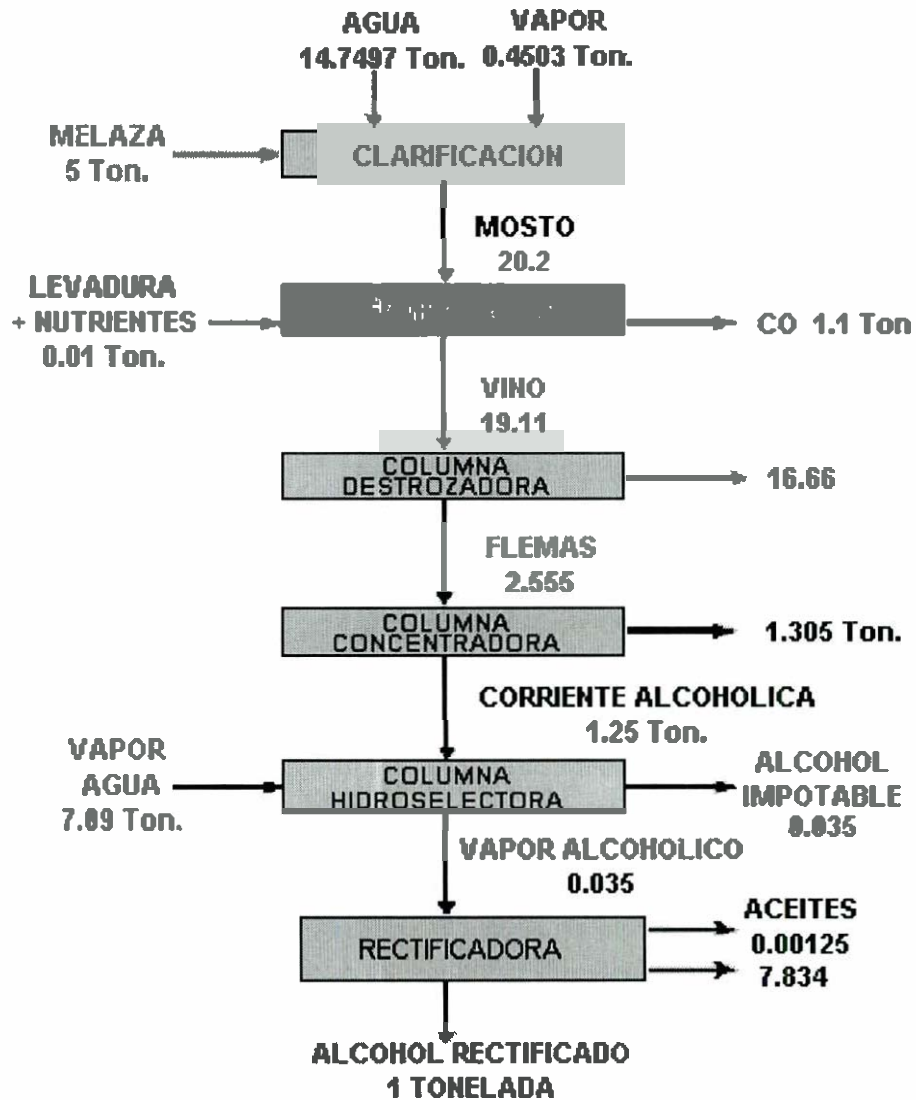
Podemos obtener dos conclusiones, la primera es que las Operaciones Unitarias están íntimamente relacionadas con bases científicas, la segunda es que fueron la base del importante desarrollo de la Ingeniería Química.

La presentación tradicional de las operaciones unitarias en la Ingeniería Química, ha sido la colección de información teórica y práctica adecuada acerca de cada operación, proporcionándola como un paquete. Tradicionalmente en los textos cada operación se presenta en forma bastante independiente de las otras. Es rara vez el caso en las presentaciones introductorias, que varias de las operaciones unitarias se yuxtapongan en sus cimientos; y sus relaciones están bastante intrincadas. Las interrelaciones se hacen más evidentes gracias a los diagramas de flujo de las operaciones unitarias de algún proceso químico industrial como el que se muestra a continuación:



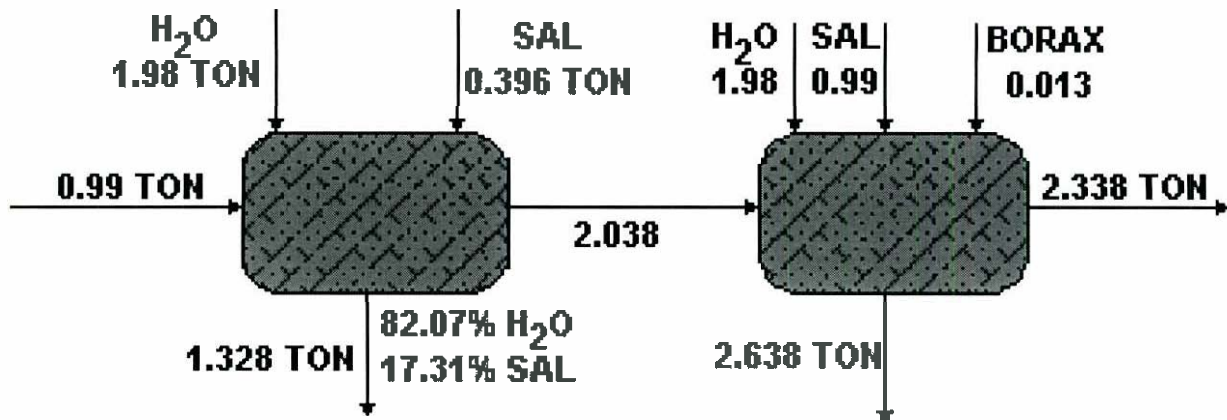
PROCESO QUÍMICO INDUSTRIAL

Para la especificación del Ingeniero químico se obtiene un diagrama de flujo con las operaciones unitarias que componen y complementan el proceso industrial como el que se muestra a continuación:



FLUJO DE OPERACIONES UNITARIAS DE LA INGENIERÍA QUÍMICA

Posteriormente esto da origen al estudio particular de cada operación unitaria, como se muestra a continuación:



EJEMPLO DE OPERACIONES UNITARIAS DE LA INGENIERIA QUIMICA

Taxonomía:

Las operaciones unitarias de la Ingeniería Química pueden clasificarse utilizando tres criterios básicos:

- Por el modelo físico que reproduce la acción de la operación.
- Por el equipo utilizado en la operación.
- Por la expresión matemática que describe la acción de la operación.

Es evidente que no hay un criterio universal que dictamine una selección particular de un método de clasificación, y de que todos los factores que contribuyen deben de reconocerse para poder decidir acerca de una manera particular.

Bases Teóricas.

Las bases teóricas de las operaciones unitarias se encuentran en los conocimientos científicos de la Física y la Química, la comprensión de los principios físicos básicos de la operación, así como la formulación de estos principios, mediante expresiones matemáticas, son los requisitos primarios para la aplicación de dichos principios de las operaciones unitarias. En la práctica de la Ingeniería Química, necesitan incorporarse siempre valores numéricos para que pueda obtenerse una respuesta práctica. Es por tanto necesario que se encuentren disponibles técnicas gráficas o matemáticas que permitan predecir cualquier respuesta desconocida para un sistema particular.

Evolución de la Ingeniería de Software.

“Tal como la Ingeniería Química es un matrimonio entre la Ciencia Química y las prácticas tradicionales de la Ingeniería, la Ingeniería de Software debe ser un matrimonio entre las Ciencias Computacionales y el añejo conocimiento de la profesión de Ingeniería”.

Software Engineering: An Unconsummated Marriage

David Lorge Parnas

McMaster University, Hamilton Ontario, Canada

Communications of the ACM, September 1997

Este punto pretende explicar el problema relacionado con la ausencia de bases científicas en la aplicación práctica de la Ingeniería de Software. El término Ingeniería de Software se remonta al final de la década de los 60's, en el año de 1968 específicamente y durante el desarrollo de las conferencias del comité científico de la Organización del Tratado de Atlántico Norte (OTAN: NATO para sus siglas en Inglés) sobre Ingeniería de Software (Ross, 1989) en la ciudad Munich, Alemania. La definición en ese entonces fue:

“El establecimiento y uso de principios de ingeniería robustos, orientados a obtener software económico que sea fiable y funcione de manera eficiente sobre máquinas reales” (Naur, 1969).

A partir de esta primera definición y durante las tres últimas décadas, la Ingeniería de Software inició formalmente su desarrollo. Para darnos una idea de su evolución y estado actual analizaremos diversas opiniones:

Mary Shaw:

Apunta que la frase "Ingeniería de Software", en el momento en que se concibió era más una aspiración que una descripción (Shaw, 1997). Esta aspiración tendía a prestar atención a los problemas de desarrollo de software desde un punto de vista de la disciplina de la Ingeniería.

Peter Denning:

En su opinión manifiesta que la Ingeniería de Software no es una disciplina, que la palabra "Ingeniería" eventualmente llevó a la falsa impresión de que la disciplina existía (Denning, 1994), pero no es así; de la misma forma cuestiona, en un trabajo previo, si las prácticas de programación y diseño cuentan con bases científicas (Denning, 1989).

James Tomayko:

Encuentra que la Ingeniería de Software no está reconocida como tal (Tomayko, 1998), según su punto de vista la Ingeniería de Software en la práctica real, tiene más características de diseño empírico, alejado de un diseño de madurez soportado por las ciencias computacionales.

Barbara Kitchenham:

Concluye que la Ingeniería de Software está compuesta por una variedad de metodologías de carácter empírico que necesitan mejorar (Kitchenham, 1999), la ausencia de bases teóricas en la práctica de la Ingeniería de Software es la causa, la solución está en que estas metodologías deberán tender a formalizarse científicamente.

David Parnas:

Indica que la Ingeniería de software es mejor entendida como una rama de la Ingeniería Clásica, a diferencia de ser considerada como una rama de las Ciencias Computacionales. Así mismo indica que es esencial que la Ingeniería de Software “aprenda” más de la Ingeniería Clásica (Parnas, 1997). Aprendizaje que conduciría a que la Ingeniería de Software sea reconocida como una rama más de la Ingeniería.

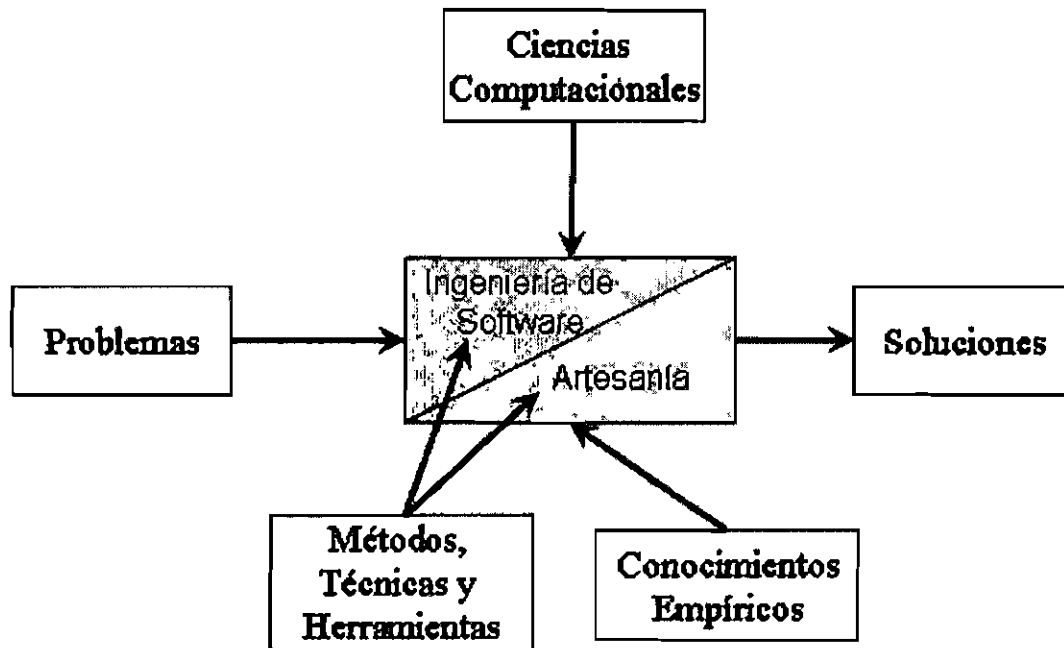
Shari Pfleeger:

Menciona que construir software puede ser un arte tanto como puede ser una ciencia (Pfleeger, 1998). Un profesional de la Ingeniería de Software debe producir un código robusto en sus programas, consecuentemente, la Ingeniería de Software está relacionada directamente con la producción de software de alta calidad. Para lograr este objetivo, afirma que la Ingeniería de Software debe estar basada en los conceptos teóricos de las ciencias computacionales, de lo contrario la producción de software tendrá características artesanales.

Reporte Técnico CMU/SEI-99-TR-004:

Este reporte que versa sobre el Cuerpo de Conocimientos de la Ingeniería de Software (Hilburn, 1999), indica que en los años recientes han existido varios estudios y comentarios sobre el estado de la profesión de la Ingeniería de Software. Todos ellos mencionan que ésta no es una profesión madura (Ford, 1996), y designa ocho componentes de infraestructura para evaluar una profesión madura. Uno de estos componentes considera los conceptos, teorías, principios y métodos que conformarán la base de la disciplina de la Ingeniería de Software. Las anteriores opiniones nos llevan a la siguiente conclusión: Existe una evidente separación, sin conocer en qué grado, entre la práctica real del desarrollo de

software, la Ingeniería de Software y las Ciencias Computacionales, resultado de la incipiente evolución de la Ingeniería de Software como disciplina formal.

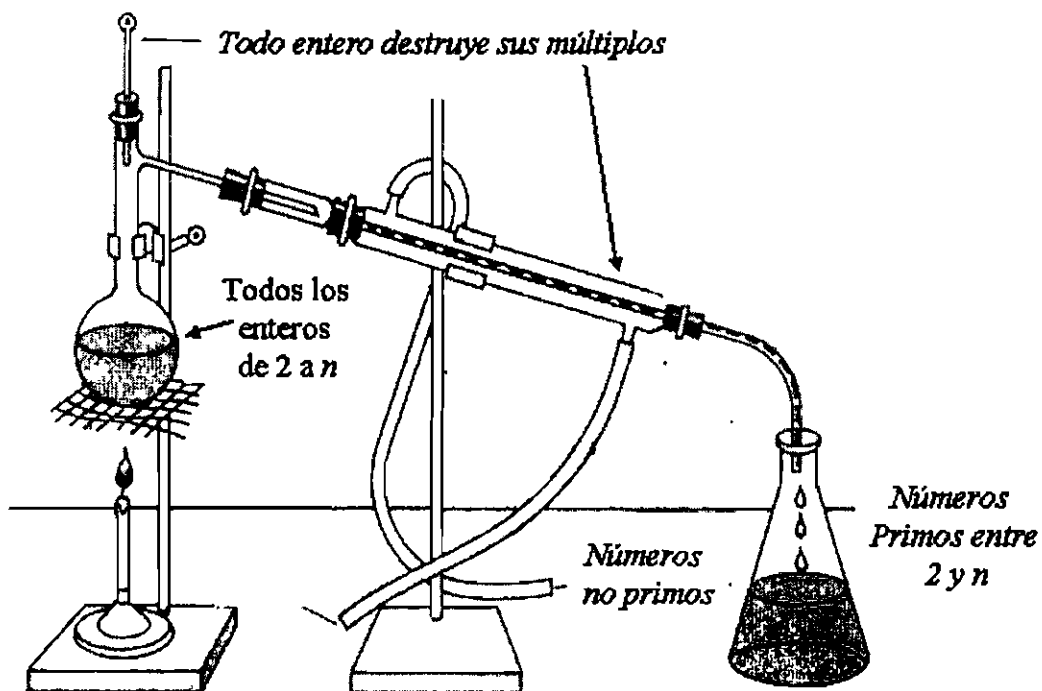


PROCESO DE LA INGENIERÍA Y ARTESANÍA DEL SOFTWARE

La Chemical Abstract Machine (CHAM).

Una relación de analogía entre la Química y las Ciencias Computacionales se encuentra en el área de las máquinas abstractas, la introducción de un nuevo tipo de máquina abstracta basada en una metáfora química, la "Chemical Abstract Machine (CHAM)" propuesta por Gerard Berry. Esta máquina abstracta está basada en un paradigma radicalmente diferente, el origen está en el lenguaje gamma (Γ) de Banâtre y Le Metayer. Menciona de manera intuitiva que el estado de un sistema es como una "solución química" (Berry, 1990) en la cual las moléculas flotantes pueden interactuar con otras de acuerdo a "reglas de reacción". Muestra que una CHAM puede implementar modelos conocidos de computación concurrente como el cálculo Lambda. Un ejemplo simple pero

sorprendente es el siguiente: Asumimos que una "solución" está integrada de todos los enteros de 2 a n , la "regla de reacción" nos dice que todo entero destruya sus múltiplos, entonces la "solución resultante", al final de la reacción, contendrá los números primos entre 2 y n .



ANALOGÍA DE LOS PROCESOS QUÍMICOS Y DE SOFTWARE

El modelo de la Chemical Abstract Machine considera:

Elementos básicos:

- Moléculas,
- Soluciones,
- Membranas,

Tres leyes generales:

- Ley de Reacción,
- Ley Química,
- Ley de la Membrana,

Tres clases de reglas:

- Reglas de Calentamiento (descomponen moléculas en sus elementos),
- Reglas de Enfriamiento (composición de moléculas a partir de elementos),
- Reglas de Reacción (modifican la naturaleza de la solución),

Podemos observar que las reglas de calentamiento tienen similitud con los procesos de descomposición de problemas en pasos, así como también con la operación unitaria de separación propuesta por Kazman. La regla de enfriamiento tiene similitud con la operación unitaria de composición uniforme, y con la técnica de diseño de abajo hacia arriba (Down Top).

Más sorprendente aún, la CHAM se utiliza para el Análisis y Especificación Formal de Arquitecturas de Software (Inverardi, 1995). La principal justificación estriba en que este modelo ofrece un marco idóneo para el desarrollo de especificaciones operacionales. Cuestión que aún no ha podido ser especificada en algún otro modelo computacional. Todo esto alentó también a la construcción y uso de especificaciones de diseño a diferentes niveles de detalle, el mismo trabajo muestra el uso particular de la CHAM en el análisis y especificación de un sistema de software: el compilador multifase.

Operaciones Unitarias en la Ingeniería de Software.

El primer y único antecedente de operaciones unitarias lo encontramos en la fase de diseño de software, en particular en el trabajo del arquitecto de software, antes de abordar la definición de las operaciones unitarias en la arquitectura de software, daremos una breve introducción a la arquitectura de software para generar un entorno adecuado al estudio de las operaciones unitarias.

Arquitectura de software

Una arquitectura de software es una organización lógica de la forma en que se dispone un conjunto de elementos que van a interactuar para llevar a cabo una tarea. La arquitectura determina la funcionalidad y modo de interacción de sus elementos.

Las arquitecturas de software han estado recibiendo mucha atención en los últimos años, dado que desempeñan un rol significativo en el proceso de desarrollo de software. Estas arquitecturas representan la organización estructural en el proceso de construcción sistemática de productos de software.

Las arquitecturas de software se definen como una estructura a alto nivel, que se muestran como una colección de componentes que interactúan, dado que es una representación abstracta que facilita el diseño y la construcción de sistemas complejos y en ella se muestran sus especificaciones y cómo funcionarán en un sistema. En la práctica es una representación lógica de todos los componentes software, cómo se comunican éstos y su disposición organizacional para interactuar y llevar a cabo las tareas definidas para un sistema.

Arquitecturas y Metodologías

Actualmente como ya observamos antes existen muchas metodologías de desarrollo de software, evidentemente, partiendo de los principios de tantas y diversas metodologías es muy difícil sacar una visión unificada sobre el diseño arquitectónico. Sin embargo, sí que podemos destacar una serie de elementos comunes.

El primero es la existencia de una fase en la que se establece o diseña una arquitectura base y el segundo la altísima dependencia que definen entre los

casos de uso y la arquitectura, definiendo un caso de uso como una interacción (secuencia de acciones) típica entre el usuario y el sistema.

Desde un punto de vista arquitectónico, no todos los casos de uso tienen la misma importancia, destacando aquellos que nos ayudan a mitigar los riesgos más importantes y sobre todo aquellos que representan la funcionalidad básica del sistema a construir.

Esta arquitectura base estará especificada por diagramas que muestren subsistemas, interfaces entre los mismos, diagramas de componentes, clases, descripciones diversas, y por el conjunto de casos de uso básicos.

Este conjunto de especificaciones nos permiten validar la arquitectura con los clientes y los desarrolladores, y asegurarnos que es adecuada para implementar la funcionalidad básica deseada. A partir de aquí, lo normal sería desarrollar de forma iterativa el sistema hasta tenerlo funcionalmente completo.

Los estilos Arquitectónicos son útiles si un diseñador puede mapear uno de estos estilos a las necesidades de un problema real, sin embargo, ¿Cómo crear un nuevo estilo si los principios primarios no son claros?, ¿Existen operaciones atómicas de la Arquitectura de software?, y si así fuera, ¿cuáles son sus efectos sobre los atributos de calidad?

Una primera aproximación parcial para contestar estas preguntas, lleva a presentar un conjunto de operaciones de diseño que son comúnmente usadas en las arquitecturas de software y describe como estas operaciones nos auxilian a cumplir requisitos de calidad. Llamaremos a estas operaciones de diseño Operaciones Unitarias, continuando con la antigua practica de la Ingeniería Química de definir operaciones unitarias. Estas operaciones de la Ingeniería de Software las titularemos compresión, abstracción, compartir recursos, descomposición uniforme y replicación.

Describiremos cada una con ejemplos de cuando han sido utilizadas en sistemas de software a gran escala, discutiremos sus efectos en el cumplimiento de requerimientos de calidad. Cada que se explique de manera general una operación unitaria, se explicarán los antecedentes de los modelos de interacción hombre-máquina en términos de operaciones unitarias, con el objetivo de mejorar el entendimiento de las decisiones de diseño básicas que siempre deberá tomar un arquitecto cuando está realizando un diseño y que cada arquitecto está tomando implícitamente los patrones de diseño arquitectónico que alguien más diseñó.

Introducción a las Operaciones Unitarias en la Fase de Diseño.

Las Operaciones Unitarias son diferentes de los estilos arquitectónicos y patrones de diseño, las Operaciones Unitarias son aún más primitivas. Los patrones de diseño y uso de estilos, son derivados de los principios de ingeniería, que traen consigo operaciones unitarias, como consecuencia, las operaciones unitarias son más abstractas que los patrones de diseño, puesto que los patrones de diseño son posteriores.

No sé está presentando un conjunto completo de operaciones unitarias, solo el subconjunto necesario para los patrones de diseño de estilos arquitectónicos más comunes actualmente, de ninguna manera se podrá considerar como el conjunto completo de operaciones unitarias. Sin embargo el conjunto describe un gran número de decisiones de diseño realizadas en complejos sistemas de software

Separación

La separación coloca una pieza definida de funcionalidad en un componente definido, que a su vez tiene una interfase definida satisfactoriamente

para el resto del universo, es la más básica y más común actividad de un arquitecto de software. La separación aísla un fragmento de funcionalidad del sistema. El motivo que nos lleva a determinar, qué fragmento de funcionalidad del sistema es aislado, proviene del deseo de cumplir en conjunto factores de calidad, por ejemplo una razón poderosa para separar funcionalidades para aumentar rendimiento es el paralelismo, importante para una arquitectura que necesita ser diseccionada en piezas pequeñas para asignarlas por separado a distintos procesadores o a diferentes equipos de desarrollo.

La separación puede también ser utilizada para asegurar que cambios externos en el ambiente no afecten a un componente, y también que cambios en el componente no afecten el ambiente, siempre y cuando la interfase no haya cambiado, de este modo la operación de separación soporta tanto a la modificabilidad y a la portabilidad.

Buenos ejemplos de separación son encontrados en las arquitecturas de flujo de datos, compiladores y sistemas de administración de interfases de usuarios, de hecho es difícil encontrar un proyecto contemporáneo de desarrollo de software que no use la separación, sin embargo, raramente usamos cotidianamente la separación, pero sí utilizamos sus subtipos tales como descomposición uniforme o replicación.

Descomposición Uniforme.

La descomposición uniforme es la operación de separar un componente grande del sistema en dos o más componentes más pequeños, la uniformidad es la restricción de esta operación. Si en el diseño tenemos determinados los mecanismos de composición de un conjunto uniforme de componentes pequeños, con mecanismos que facilitan la integración y escalamiento de componentes de un sistema como un todo, se distinguen dos mecanismos de descomposición como operaciones unitarias que se analizan a continuación:

Parte del todo.- Cada cual de un conjunto restringido de sub-componentes, representan partes no sobre traslapadas de fragmentos de funcionalidad, y cada componente en el sistema puede ser construido solamente con estos sub-componentes.

Es un.- Cada cual de los sub-componentes representa una especialización de la funcionalidad del componente padre.

Replicación.

La replicación es la operación de duplicar un componente dentro de una arquitectura, esta técnica es utilizada para ampliar la confiabilidad (tolerancia a fallas) y aumentar el desempeño. Esta operación unitaria es utilizada tanto en componentes de hardware como de software. Cuando los componentes se replican es porque el sistema demanda que, cuando un componente falla, existan uno o más componentes que entran en funcionamiento automáticamente corrigiendo la falla, incrementando consecuentemente la tolerancia a fallas. Cuando la magnitud de la replicación de un sistema se incrementa, el trabajo disponible puede ser extendido entre más de los componentes del sistema, de este modo se incrementa la cantidad de información procesada por unidad de tiempo, sin embargo la posibilidad de que un componente falle se incrementa dramáticamente.

La replicación se presenta en dos formas: en tiempo de ejecución y replicación estática.

En tiempo de ejecución.- En la replicación en tiempo de ejecución los componentes replicados ejecutan la misma acción en el mismo tiempo durante la ejecución. Esta operación provee tolerancia a fallas a costa del desempeño (el desempeño que podría haber sido alcanzado si todo el poder de computo, hubiese sido utilizado para otras cosas en vez de la misma). La plataforma de

lanzamiento de las arquitecturas que utilicen replicación en tiempo de ejecución en cualquiera de sus múltiples versiones del mismo software determina como conducir el diseño.

Replicación estática.- Los componentes que son replicados, son desarrollados utilizando copias del mismo código fuente, pero deben ejecutar diferentes funciones por separado en cualquier momento, dependiendo del estado interno o de cómo está siendo utilizado. Este tipo de replicación es una aproximación a los compromisos de confiabilidad y desempeño: Un componente de respaldo puede ser útil en ejecución pero no necesario, hasta que se presenta una falla, en cuyo caso toma el trabajo que estaba realizando el componente que falló. Existen plantillas de código para componentes de software con tolerancia a fallas que respaldan por separado durante la ejecución.

Abstracción.

Es la operación de creación de una máquina virtual. Una máquina virtual es un componente cuya función es ocultar una implementación subyacente. Las máquinas virtuales son frecuentemente piezas de software complejas de construir, pero una vez creadas pueden ser adecuadas y reutilizadas por otros componentes de software, de modo que, simplifiquen su creación y mantenimiento (esto en razón de que hacen referencia a un conjunto de funcionalidad abstracta).

Las máquinas virtuales son encontradas en cualquier lugar donde sea necesario emular alguna pieza de software que no es nativa; un ejemplo es la simulación de la computación paralela sobre un solo procesador. Otro ejemplo de uso muy común es el uso de máquinas virtuales en sistemas estratificados en capas. Un tercer uso es proveer una interfase común para un conjunto heterogéneo de implementaciones subyacentes, tal como las herramientas portables para interfases de usuario como podrían ser los multilenguajes. En este sentido la interfase de usuario es un fragmento del software que es escrito solo

una vez, en términos de una abstracción proveída por la herramienta, de manera similar las capas de abstracción son comúnmente adicionadas a sistemas monousuario para permitir que estas sean compartidas entre múltiples usuarios.

La separación y la abstracción están relacionadas, pero no son la misma operación, existen muchos ejemplos de separación por otras razones que crear un conjunto de servicios abstractos, tal como carga balanceada, operaciones paralela y división de trabajo entre equipos de desarrollo.

Compresión.

La compresión es la operación de remover capas o interfaces que separan funciones del sistema, de esta forma es lo opuesto a la separación, estas capas pueden ser software (límites de procesos, llamadas a procedimientos) o hardware (procesadores separados). Cuando se comprime software se toman dos distintas funciones y se colocan juntas. En la historia de la Ingeniería de Software y la Ciencia de la Computación ha existido una tendencia alejada de la compresión: Tipos de datos abstractos, el paradigma cliente-servidor, computación distribuida, computación paralela y el desarrollo orientado a objetos son todos ejemplos de adición de capas o interfaces.

La compresión cubriría tres propósitos principales:

Para mejorar el desempeño del sistema.- Por la eliminación del sobre trabajo de cruzar las capas o interfaces entre las diferentes funciones; por ejemplo se discute el uso de la capa de eliminación para conocer las metas de desempeño en un sistema de tiempo real tolerante a fallas, otros ejemplos de compresión para mejorar el desempeño incluyen retroalimentación semántica en interfaces de usuario y capas no comprometidas en protocolos de comunicación.

Para engañar a la estratificación de capas.- (saltar capas) cuando no se proveen los servicios necesarios.

Para acelerar el desarrollo de sistemas.- Eliminando los requisitos para que diferentes fragmentos de funcionalidad de un sistema sean colocados en componentes de software separados; por ejemplo Microsoft Visual Basic permite directamente a un desarrollador unir los objetos individuales de interfase de usuario al código de la aplicación.

Una técnica común que automáticamente realiza la compresión es el uso de macros o procedimientos en línea, en esta técnica un usuario crea código que es empacado en una macro o procedimiento (por tanto separado del resto del software), pero cuando el sistema es compilado, en alguna referencia o llamada este código es reemplazado por el actual código, de modo que el sistema compilado ha removido el sobre tiempo de ejecución de llamadas a procedimiento al costo de crear potencialmente un sistema substancialmente grande (si hay 50 llamadas a un procedimiento en línea, el código para este procedimiento será incluido 50 veces en la versión compilada resultante).

Compartir recursos

El compartir recursos es una operación que encapsula datos, servicios o ambos y los comparte entre múltiples clientes independientes. Típicamente hay un administrador del recurso que provee acceso exclusivo al recurso. Los recursos compartidos son frecuentemente costosos de construir inicialmente, pero amplían la integrabilidad y portabilidad de los sistemas, básicamente por que reducen el nivel de acoplamiento entre componentes.

Ejemplos comunes de recursos de software compartido son bases de datos, pizarrones, ambientes de herramientas integradas de ingeniería de software y servidores (en un sistema cliente servidor). En cada uno de estos casos, los recursos compartidos amplían la integrabilidad del sistema, en muchos casos los recursos compartidos son también abstracciones, El servidor de X-

Windows por ejemplo es un recurso compartido que provee una abstracción subyacente de gráficos desde el hardware.

Los repositorios de datos como pizarrones y bases de datos son recursos compartidos, el recurso compartido son los datos persistentes a ser almacenados y recuperados. Los núcleos de seguridad también administran recursos compartidos, típicamente acceden a privilegiados datos y funcionalidades del sistema. Los ambientes integrados de ingeniería de software confían en la noción de un medio o repositorio compartido explícitamente que permita la integración de herramientas. El estándar CORBA permite que objetos completos y de hecho aplicaciones completas sean encapsuladas y remotamente accesadas anónimamente.

Interacción entre atributos de calidad utilizando operaciones unitarias.

El interés de las operaciones unitarias no se restringe solamente a que auxilien o dificulten el cumplimiento de requisitos de calidad, también interesan en la manera en que interactúan los atributos de calidad. Es obvio que no se pueden maximizar todos los atributos de calidad. Esta es la razón de la ingeniería o disciplina de diseño. Un gran puente no es el más ligero, rápido de construir o el mas barato. El vehículo más veloz y fácil de conducir no es un vehículo de carga economizador de combustible, de la misma forma el postre de mejor sabor nunca es bajo en calorías.

En el campo del software, se está interesado en entender el efecto e interacciones de las operaciones unitarias con respecto de la escalabilidad, integrabilidad, portabilidad, desempeño (separado en desempeño secuencial y desempeño concurrente que es el desempeño obtenido de la habilidad de paralelizar operaciones), tolerancia a fallas, facilitar la creación de sistemas, facilitar la creación de componentes, modificabilidad completa del sistema, modificabilidad de componentes individuales y reusabilidad.

Basado en encuestas de expertos diseñadores de software, se puede resumir a grandes rasgos las relaciones entre seis operaciones unitarias (excluyendo la separación, un súper tipo de otras operaciones pero considerando la descomposición parte del todo y la descomposición es parte de separadamente) y los 11 atributos de calidad mostrados en la tabla siguiente:

Relación entre operaciones unitarias y atributos de calidad

Operaciones Atributos	Abstracción	Compresión	Descomposición "parte del todo"	Descomposición "es parte de"	Replicación	Compartir recursos
Escalabilidad					+	
Modificabilidad del sistema	+	-	+	+		
Integrabilidad	+	-	+	+		+
Portabilidad	+	-				
Desempeño secuencial	-	+		-	-	-
Desempeño concurrente		-	+	+	+	
Tolerancia a fallas	+				+	
Facilidad de creación del sistema	+	-		+		+
Modificabilidad de componentes	+	-				-
Facilidad de creación de componentes	+			+		+
Reusabilidad	+	-	+			

Un signo + en la tabla indica una relación positiva entre la operación unitaria y el atributo de calidad, lo que significa que el uso de esta operación auxilia en el cumplimiento de metas de calidad. Un signo – indica que el efecto opuesto, si la casilla está en blanco indica que dependiendo del contexto la operación puede tener un efecto negativo o positivo, por ejemplo la

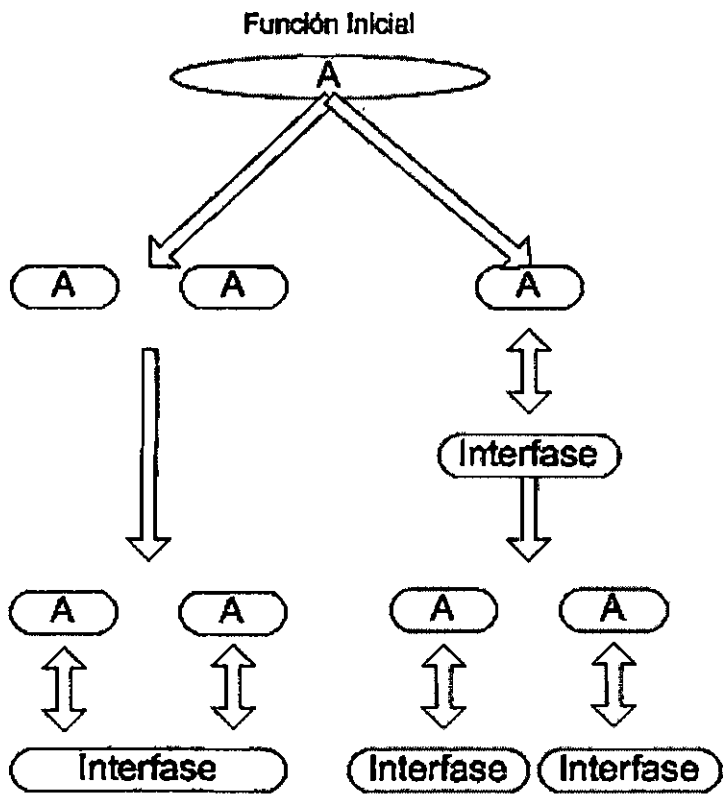
descomposición "parte del todo" fortalece la portabilidad, si y solo si el fragmento que cambia de plataforma a plataforma ha sido aislado en una sola parte, de cualquier otro modo la descomposición obstaculizará la portabilidad, por que los cambios que se realizarán no serán locales, estos serán distribuidos entre las partes exteriores, de manera similar la casilla blanca entre la portabilidad y compartir recursos es por que la portabilidad es ampliada solamente cuando compartir recursos es lo único que necesitamos para ser transportada (y por tanto los cambios son locales) y esto se reduce cuando ambos, los recursos compartidos y los usuarios del recurso necesitan ser modificados (y por tanto los cambios no son locales), el blanco en cada uno de los casos representa un escenario diferente de cómo las operaciones unitarias han de ser aplicadas.

Esta tabla no tiene la intención de presentar la ultima palabra con respecto a las operaciones unitarias y atributos de calidad, ambos atributos de calidad y operaciones unitarias son abstractas y solo una guía en nuestras decisiones de diseño entre diferentes alternativas. Estas categorías por si mismas nos muestran un poco acerca del contexto de cada cuestión y acerca de cómo los diseñadores determinan sus respuestas. El fundamento detrás de la comprensión del valor que proveen estas categorías de cómo y cuándo utilizar determinada operación, en las encuestas con los expertos diseñadores de software denota la regularidad recurrente de razonamientos para el uso de operaciones unitarias y las restricciones en su uso.

El punto aquí es que estas operaciones tienen interacciones complejas, y el efecto de una operación unitaria en términos de atributos de calidad no es trivial, deberá ser comprendida en el contexto de las demás decisiones de diseño, materiales disponibles, costos, tiempos y el contexto de la operación del sistema. El entendimiento de este ambiente es fortalecido por un análisis en términos de operaciones unitarias; apoyaran el enfoque e identificación de conflictos de diseño. Para diseñar una arquitectura completa, los requerimientos de calidad deberán ser por supuesto priorizados, así que el orden de la composición de las

operaciones unitarias debe ser determinado por las necesidades a ser cubiertas, en el entendido que los objetivos de calidad no pueden ser cubiertos por si solos y las operaciones para transformar las necesidades en software, no son generalmente conmutativas. El arquitecto de software deberá tomar decisiones difíciles, decidiendo si la tolerancia a fallas es más importante que la modificabilidad, o que la seguridad es más importante que el desempeño.

Por ejemplo, considere un sistema para un juguete, el cual necesita de una función A. Supongamos que desean introducir modificabilidad al producto y en particular al software, por lo que se realiza una abstracción de A. Pero también desean que la función sea tolerante a fallas a través de la replicación. Hay dos maneras de componer estas dos operaciones como se muestra en la figura siguiente:



NO CONMUTATIVIDAD DE OPERACIONES UNITARIAS

La parte de la izquierda muestra una replicación de la función A para dos productos con sus respectivas copias de la función A, posteriormente se realiza una abstracción que produce una interfase, en la parte de la derecha se lleva a cabo una primera abstracción del producto A con una interfase, posteriormente se replican dos copias de la función A con su interfase para asegurar la tolerancia a fallas, en estas circunstancias la modificabilidad se tornará más compleja puesto que ya tenemos dos tipos de interfases para la misma función A

La Fase de Análisis en Métodos Tradicionales.

La fase de análisis en la mayoría de los métodos es una fase preparatoria y pre-ejecutora, pero es considerada en todos los métodos, en algunos casos mas formalizada con notación propia y un soporte herramental importante, pero en algunos casos tomada a la ligera, considerando que existe pero sin mucho soporte, para el caso del Análisis y Diseño estructurado es una fase que incluye notaciones diagramáticos y textuales con variadas técnicas, para el Análisis y Diseño Orientado a Objetos, las funcionalidades son especificadas por los métodos de acceso a los objetos y el Proceso Unificado de Desarrollo de Software, considera la elaboración de varios modelos diagramáticos para modelar la realidad y la posible solución, pero en para las métodos disciplinarios propuestos por Humphrey como el caso del PSP (Personal Software Process) en el que la actividad del Ingeniero de Software esta centrada en la programación a partir de la descripción de un requerimiento (Humphrey, 1995). Una extensión del PSP en equipo de trabajo evoluciona al TSP (Team Software Process) que implica un marco de trabajo en grupos de trabajo basado en el PSP (Humphrey, 1999), en el que los miembros del equipo comparten la misma disciplina de programación, con los mismos términos y notación de avances, incluyendo roles muy específicos para los lideres de proyectos y planeadores de los procesos.

Análisis y Diseño Estructurado

“El Análisis es frustrante; está repleto de relaciones interpersonales complejas, indefinidas y difíciles. En una palabra, es fascinante. Una vez que uno se vuelve adicto, los viejos placeres fáciles de la construcción de sistemas ya nunca vuelven a satisfacerles”

Tom De Marco, 1978

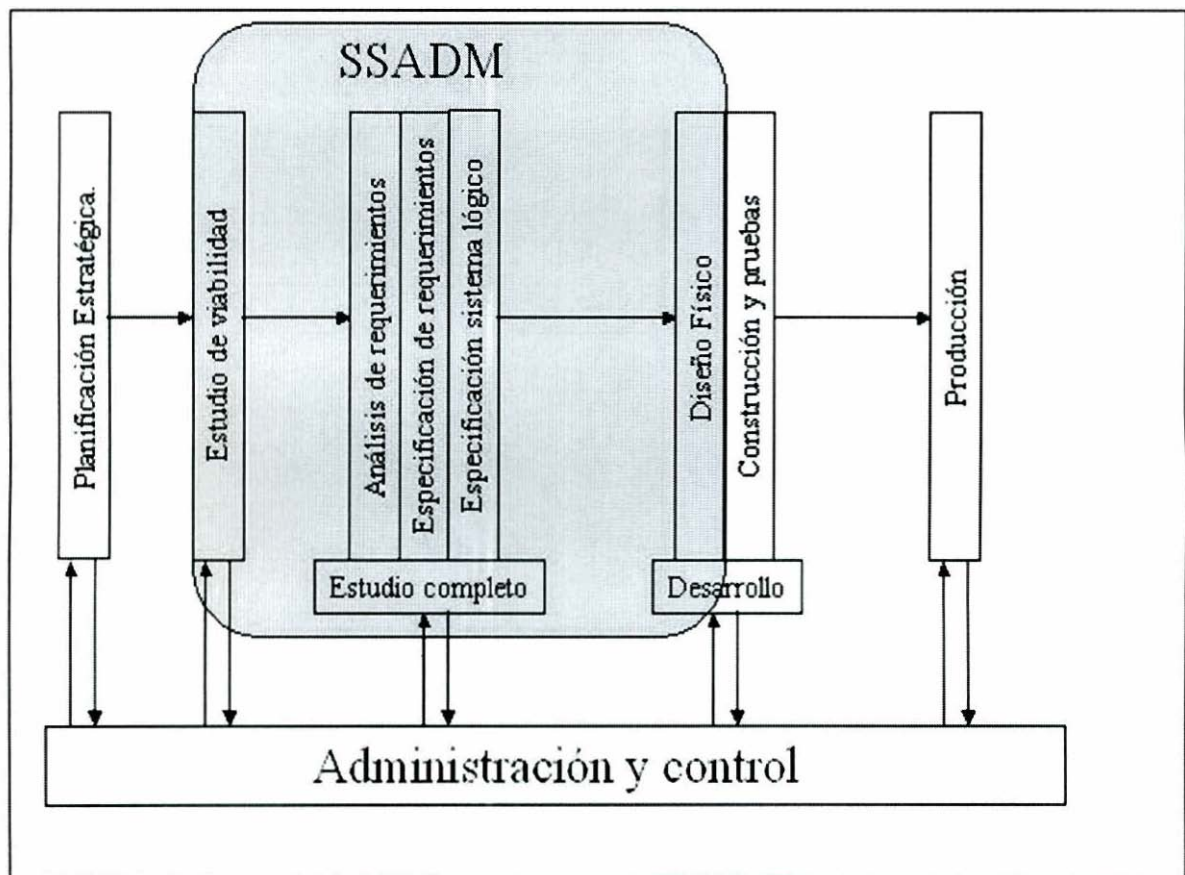
El precursor de este método es Tom de Marco cuando a finales de la década de los 70'S publica su libro "Structured Anlysis and System Specification", posteriormente evoluciona consolidándose en 1980, en Inglaterra, cuando el Gobierno Británico plantea la necesidad de adoptar una metodología de desarrollo de software, que minimizara los problemas típicos de las áreas de informática, entre los que se contaban: Alta rotación de personal, falta de documentación, métodos no definidos, sobrecarga de mantenimiento, etc.

El proyecto se lleva a cabo conjuntamente entre la Central Computer an Telecommunications Agency (CCTA) y Learmonth and Burchett Managament Systems (LBMS). El resultado fue la Metodología Structured Systems Analysis and Design Method (SSADM), la cual hace énfasis en el uso de diagramas y diseños esquemáticos desarrollados por Tom DeMarco y Edward Yourdon.

Con el tiempo esta metodología SSADM se hizo obligatoria en todos los proyectos informáticos del Gobierno Británico. Inicialmente la metodología solo contemplaba las fase de análisis y diseño. La fase de estudio de viabilidad se incorpora en la Versión 3.0, en julio de 1998, en este año había ya 600 proyectos

para SSADM en el Gobierno Británico, y poco después se formó el grupo de usuarios SSADM en el sector privado.

Se puede observar en la siguiente figura que el análisis y diseño estructurado comprende solamente de fase viabilidad hasta el diseño físico exclusivamente, haciendo hincapié en la fase de análisis en tres etapas: Análisis de Requerimientos, Especificación de Requerimientos y Especificación Lógica del Sistema



FASES DEL ANÁLISIS Y DISEÑO ESTRUCTURADO

La versión 3.0 comprendía 3 etapas agrupadas en tres fases: Estudio de Viabilidad, Análisis y Diseño. En 1990 aparece la versión 4.0 con una remodelación importante de la estructura del método, haciéndola más modular, menos iterativa e incorporando nuevas técnicas fundamentalmente en el diseño de procesos.

El método se va a dividir en las siguientes fases:

Módulo FS (Feasibility Study)

ESTUDIO DE VIABILIDAD

Etapa 0 Viabilidad

- Paso 010 Preparar el estudio de viabilidad
- Paso 020 Definir el problema
- Paso 030 Seleccionar opciones
- Paso 040 Componer el informe de viabilidad

Módulo RA (Requirements Analysis)

ANÁLISIS DE REQUERIMIENTOS

Etapa 1 Investigación del Sistema actual

- Paso 110 Establecer el marco de trabajo
- Paso 120 Investigar y definir requerimientos
- Paso 130 Investigar procesos del sistema actual
- Paso 140 Investigar datos del sistema actual
- Paso 150 Obtener una visión lógica del sistema actual
- Paso 160 Componer los resultados de la investigación

Etapa 2 Estudio de opciones del sistema

- Paso 210 Definir opciones del sistema
- Paso 220 Seleccionar una de las opciones

Módulo RS (Requirements Specification)

ESPECIFICACIÓN DE REQUERIMIENTOS

Etapa 3 Definición de requerimientos

- Paso 310 Definir procesos del sistema requerido
- Paso 320 Definir el modelo de datos requerido
- Paso 330 Obtener las funciones del sistema
- Paso 340 Refinar el modelo de datos requerido

- Paso 350 Desarrollar prototipos
- Paso 360 Desarrollar especificaciones de procesos
- Paso 370 Confirmar objetivos del sistema
- Paso 380 Componer la especificación de requerimientos

Módulo LS (Logical System Specification)

ESPECIFICACIÓN DEL SISTEMA LÓGICO

Etapa 4 Opciones del sistema Técnico

- Paso 410 Definir especificaciones técnicas
- Paso 420 Seleccionar opción técnica

Etapa 5 Diseño lógico

- Paso 510 Definir procesos de actualización
- Paso 520 Definir procesos de consulta
- Paso 530 Componer el diseño lógico

Módulo PD (Physical Design)

DISEÑO FÍSICO

Etapa 6 Diseño físico

- Paso 610 Preparar el diseño físico
- Paso 620 Crear Diseño físico de datos
- Paso 630 Crear mapa de implementación de funciones
- Paso 640 Optimizar el diseño físico de datos
- Paso 650 Completar la especificación de funciones
- Paso 660 Consolidar interfaces de procesos de datos
- Paso 670 Componer el diseño físico

El propósito del Análisis Estructurado según Edward Yourdon y Tom DeMarco es producir una "Especificación Estructurada", el cual es parte de documento de requerimientos y es básicamente gráfica, generando como resultado un conjunto de diagramas esquemáticos y se dividen en tres categorías:

Especificaciones de control:

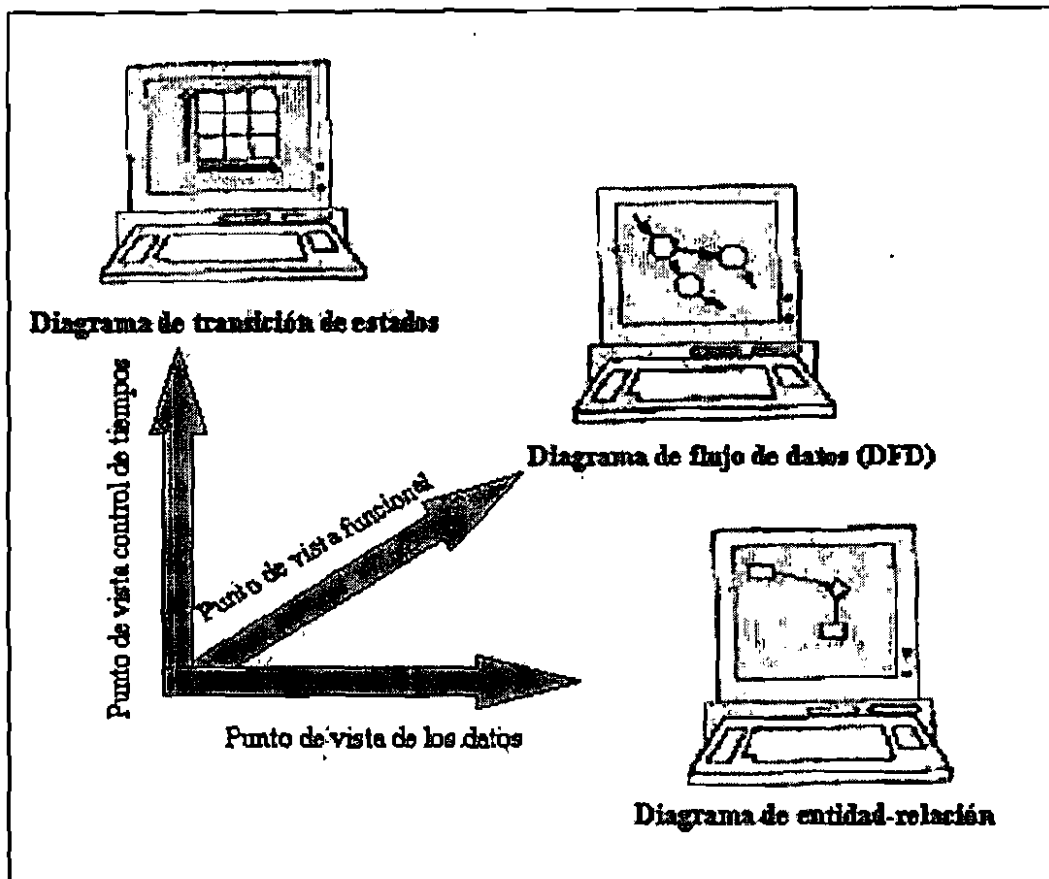
- Diagramas de transición de estados.
- Árboles de decisión
- Diagramas de flujo, etc.

Diseño funcional:

- Diagramas de Flujo de datos (DFD)

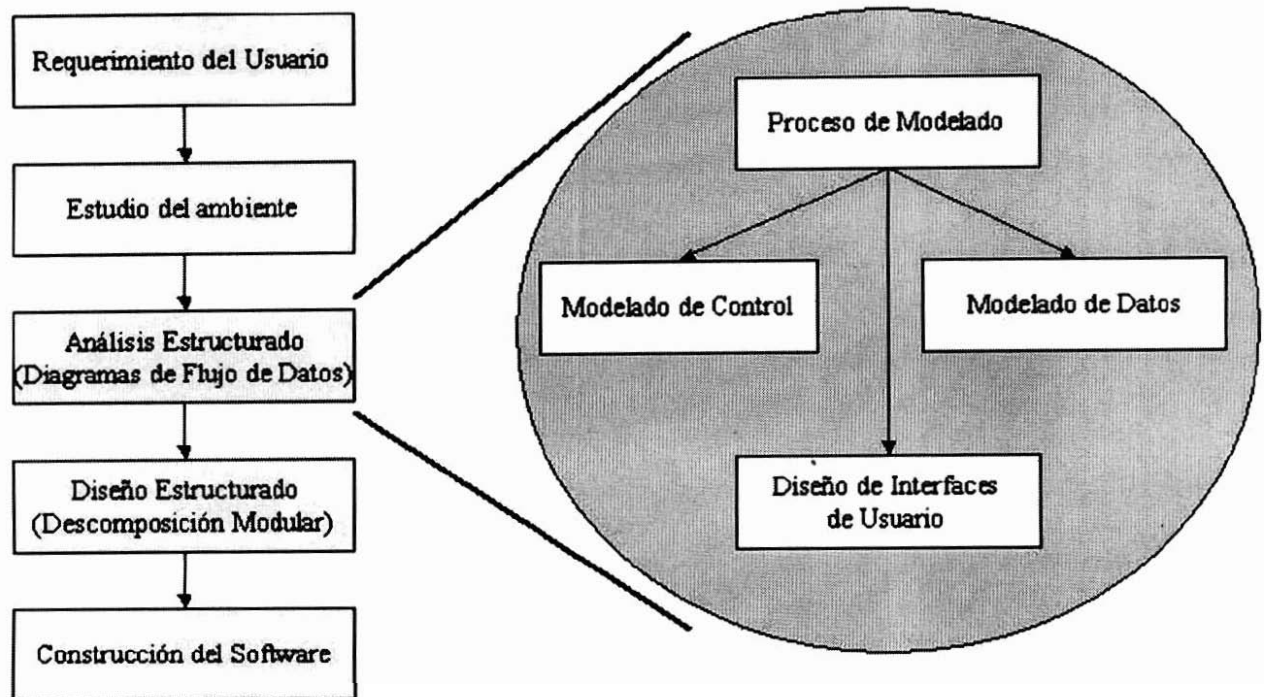
Modelado de datos

- Diagramas de Entidad-Relación



MODELOS DIAGRAMÁTICOS DEL ANÁLISIS ESTRUCTURADO

El proceso de Análisis y Diseño Estructurado podemos visualizarlo en la siguiente gráfica donde se aprecian sus principales actividades.



PRINCIPALES ACTIVIDADES DEL ANÁLISIS Y DISEÑO ESTRUCTURADO

La técnica más utilizada es la de Diagramas de Flujos de Datos de la cual proveemos un extracto que incluye las mejores prácticas de las mismas:

A manera de introducción a esta técnica podemos decir que con la aparición de los métodos estructurados de análisis, llegó una revolución en el desarrollo de sistemas. Los diagramas de flujo de datos es el método más extendido en modelado conceptual de procesos con distintas notaciones. Esta técnica, que también es conocida por sus siglas, DFD, o por diagramas de burbujas (Bubble Diagrams), fue propuesta por Yourdon a finales de los años setenta y es utilizada principalmente en la fase de análisis y diseño.

El objetivo fundamental de esta técnica es la descomposición de un problema complejo en otros más sencillos y manejables, facilitando la modularidad del sistema, así como aprovechar la comunicabilidad a través de modelos gráficos. Pretende, además, separar la estructura física del sistema de la

lógica, aumentando la facilidad de mantenimiento de los análisis. Los DFD constan de muy pocos elementos, y además son fáciles de aprender.

Estos elementos son:

- **Procesos:** Son los lugares donde se transforma o descompone la información de un flujo de datos.
- **Entidades externas:** Son aquellas organizaciones, departamentos, personas o entidades en general que no pertenecen al sistema de información, pero que tiene relación con él. Es de donde proceden los datos que necesita el sistema o hacia donde van los generados por él.
- **Almacenes de datos:** Son aquellos lugares donde se almacena de forma estática la información dentro del sistema. Su nombre no indica como está organizado. Se representan gráficamente con dos líneas paralelas.
- **Flujos de datos:** Son las tuberías o caminos por donde fluye la información entre los diferentes elementos del sistema.

Notaciones mas Importantes:

- Yourdon.
- SADT (Structured Analysis and Design Technique).
- De Marco,
- Gane & Sarson
- Ward & Mellor (para sistemas en tiempo real)

Proceso de análisis:

Consiste en un refinamiento progresivo a través de las siguientes fases:

- **Construcción del diagrama de contexto.** Es aquél DFD compuesto por un solo proceso que representa todo el sistema y los flujos de entrada y de salida.

- **Descomposición** de cada proceso en los subsiguientes niveles de DFD, hasta conseguir todos los procesos primitivos (aquellos que ya no se pueden seguir descomponiendo).
- **Descripción** de todos los elementos obtenidos en el diccionario de datos.

Propiedades de los DFD

Existe una serie de características generales a tener en cuenta a la hora de diseñar un diagrama de flujo de datos óptimo:

Evitar la utilización de estructuras ilegales. Éstas son:

- Flujos de datos que se subdividen en el diagrama. Es decir, para que un flujo de datos se pueda desdoblar, es necesario un proceso que realice esta operación. No podemos olvidar que una de las funciones de los procesos es descomponer flujos de datos.
- Señales de control, y en concreto señales de activación. No pueden aparecer flujos de datos cuyo único contenido sean datos booleanos, señales de final de fichero, pasos de control por llamadas a procedimientos, etc.
- Bucles. Si un flujo entra y sale en el mismo proceso, significa que es algo que sólo le interesa a dicho proceso, por lo tanto debe quedar escondido.
- Flujos de datos entre dos almacenes de datos. Para poder hacer una lectura en un almacén y una escritura en otro, es necesaria la existencia de un proceso que realice dicha función.
- Procesos aislados. No tiene sentido la existencia de un proceso que no emita o reciba flujos de datos sin conexión alguna con el resto del sistema.
- Flujos entre entidades externas. El paso de información que pueda existir entre dos entidades externas es algo que no interesa al sistema que se está diseñando. Si nos damos cuenta de que sí

interesa, es porque esta información debe pasar a través del sistema.

- Procesos sumidero y proceso fuente. Son aquellos que sólo reciben o emiten información, respectivamente. No son válidos en un DFD.

Procurar mantener la propiedad de conservación de los datos

- Los datos no se crean, sólo se transforman. No pueden existir datos en el sistema que surjan de la nada. Deben aparecer en el sistema viniendo del exterior, o como resultado de la transformación de otros flujos.
- Balanceo entre niveles (Level Balancing), según esta propiedad, los flujos de entrada y de salida de un proceso de nivel n tienen que ser los mismos que en el diagrama de nivel $n+1$. Esta propiedad se rige por las siguientes reglas:
- Todas las entradas de un diagrama $n+1$ deben estar en el diagrama n .
- Todas las salidas del diagrama $n+1$ deben ser las mismas del diagrama n , excepto los rechazos triviales, entendiendo por estos últimos aquellos flujos de salida que representan un mensaje de error o de cualquier otro tipo que no sea muy interesante hacer notar desde el primer nivel.

Propiedades prácticas, Existen una serie de normas que parten de la experiencia:

- Dividir el sistema de manera natural. No por empeñarnos en descomponer el sistema va a estar mejor analizado. Sólo debemos descomponer si realmente es necesario y cuando la propia naturaleza del sistema lo determine.
- Establecer conexiones simples. Cuantos menos flujos de datos haya entre dos procesos, mejor. Si se ve que son necesarios,

agruparemos primero los flujos y los descompondremos en niveles sucesivos.

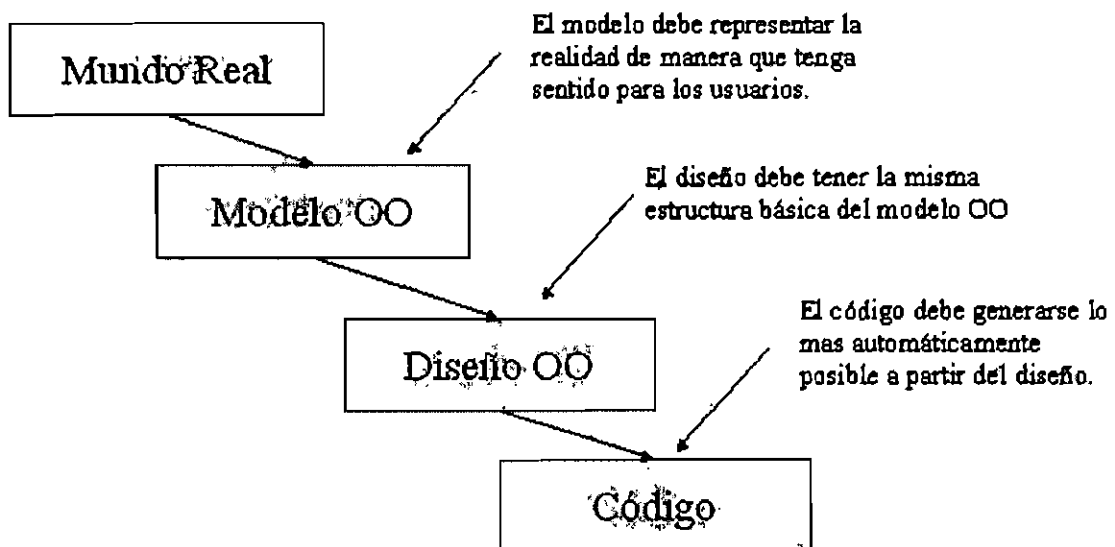
- Limitar el número de procesos dentro de un nivel de descomposición determinado. número ideal 2 ± 7 .
- Explosionar cuanto sea necesario. El sistema no va a ser más complicado porque tenga más niveles, todo lo contrario.
- Descomponer procesos si tienen varios flujos de entrada y/o salida. Si existen varios flujos entrando o saliendo de un proceso, eso nos indica que seguramente existen partes de la lógica de dicho proceso que en niveles inferiores serán procesos individuales.
- Nominar correctamente todos los objetos que surjan en los diagramas. Los elementos lingüísticos que se recomiendan para cada tipo de objeto son los siguientes:
 - Flujos de datos: un sustantivo más un adjetivo.
 - Procesos: un verbo más un sustantivo.
 - Almacenes de datos: un sustantivo.

Análisis y Diseño Orientado a Objetos

“Einstein arguyó que debe de haber explicaciones simplificadas de la naturaleza, por que Dios no es caprichoso ni arbitrario. No hay fe semejante que conforte al Ingeniero de Software. Mucha de la complejidad que debe dominar es complejidad arbitraria”

Brooks, F, Abril 1987

Este método fue introducido por Grady Booch en 1994 y que dio origen al término “Paradigma orientado al objeto”, un paradigma de programación es una colección de modelos conceptuales que juntos modelan el proceso de diseño y determinan la estructura de un programa, el Paradigma orientado a objetos, podemos decir que es una disciplina de modelado de software que permite construir más fácilmente sistemas complejos a partir de componentes individuales.



FASES DEL PARADIGMA ORIENTADO A OBJETOS

Desde comienzos del paradigma "orientado a objetos" ha ido madurando como un enfoque de desarrollo de software alternativo a la programación estructurada o modular. Se empezó a crear diseños de aplicaciones de todo tipo usando una forma de pensar orientada a los objetos, y a implementar estos diseños utilizando lenguajes orientados a objetos. Sin embargo, el análisis de requisitos se quedó atrás. No se desarrollaron técnicas de análisis específicamente orientadas a objetos.

Esta situación ha ido cambiando poco a poco, a medida que se desarrollaban técnicas de análisis específicas para desarrollar software orientado a objetos, e incluso como complemento de otros métodos de análisis. Ejemplos de estas nuevas técnicas son los métodos de Coad/Yourdon, Jacobson, Booch y Rumbaugh (OMT). En esta parte seguiremos principalmente esta última metodología, de interés para esta tesis.

El Análisis Orientado a Objetos (AOO) se basa en conceptos sencillos, conocidos desde la infancia y que aplicamos continuamente: objetos y atributos, el todo y las partes, clases y miembros. Puede parecer llamativo que se haya tardado tanto tiempo en aplicar estos conceptos al desarrollo de software. Posiblemente, una de las razones es el éxito de los métodos de análisis estructurados, basados en el concepto de flujo de información, que monopolizaron el análisis de sistemas software durante los últimos veinte años.

En cualquier caso, el paradigma orientado a objetos ha sufrido una evolución similar al paradigma de programación estructurada: primero se empezaron a utilizar los lenguajes de programación estructurados, que permiten la descomposición modular de los programas; esto condujo a la adopción de técnicas de diseño estructuradas y de ahí se pasó al análisis estructurado. El paradigma orientado a objetos ha seguido el mismo camino: el uso de la Programación Orientada a Objetos (POO) ha modificado las técnicas de diseño

para adaptarlas a los nuevos lenguajes y ahora se están empezando a utilizar técnicas de análisis basadas en esta nueva forma de desarrollar software.

El AOO ofrece un enfoque nuevo para el análisis de requisitos de sistemas software. En lugar de considerar el software desde una perspectiva clásica de entrada/proceso/salida, como los métodos estructurados clásicos, se basa en modelar el sistema mediante los objetos que forman parte de él y las relaciones estáticas (herencia y composición) o dinámicas (uso) entre estos objetos. Este enfoque pretende conseguir modelos que se ajusten mejor al problema real, a partir del conocimiento del llamado dominio del problema, evitando que influyan en el análisis consideraciones de que estamos analizando un sistema para implementarlo en un ordenador. Desde este punto de vista, el AOO consigue una abstracción mayor que el análisis estructurado, que modela los sistemas desde un punto de vista más próximo a su implementación en una computadora (entrada/proceso/salida).

Este intento de conocer el dominio del problema ha sido siempre importante; no tiene sentido empezar a escribir los requisitos funcionales de un sistema de control de tráfico aéreo, y menos aún diseñarlo o programarlo sin estudiar primero qué es el tráfico aéreo o qué se espera de un sistema de control de este tipo. La ventaja del AOO es que se basa en la utilización de objetos como abstracciones del mundo real. Esto nos permite centrarnos en los aspectos significativos del dominio del problema (en las características de los objetos y las relaciones que se establecen entre ellos) y este conocimiento se convierte en la parte fundamental del análisis del sistema software, que será luego utilizado en el diseño y la implementación.

Este enfoque no es totalmente nuevo, sino que puede considerarse como una extensión del modelado de datos (M-ER) que se utiliza en los métodos estructurados. Sin embargo, el modelado de datos mediante M-ER está más orientado al diseño de bases de datos y se centra exclusivamente en la

identificación de los datos que maneja un sistema y en las relaciones estáticas que se establecen entre esos datos. En el AOO, los objetos encapsulan tanto atributos como procedimientos (operaciones que se realizan sobre los objetos), e incorpora además conceptos como el polimorfismo o la herencia que facilitan la reutilización de código.

El uso de AOO puede facilitar mucho la creación de prototipos, y las técnicas de desarrollo evolutivo de software. Los objetos son inherentemente reutilizables, y se puede crear un catálogo de objetos que podemos usar en sucesivas aplicaciones. De esta forma, podemos obtener rápidamente un prototipo del sistema, que pueda ser evaluado por el cliente, a partir de objetos analizados, diseñados e implementados en aplicaciones anteriores. Y lo que es más importante, dada la facilidad de reutilización de estos objetos, el prototipo puede ir evolucionando hacia convertirse en el sistema final, según vamos refinando los objetos de acuerdo a un proceso de especificación incremental.

Deficiencias del análisis estructurado desde el punto de vista de AOO

Descomposición funcional. El análisis estructurado se basa fundamentalmente en la descomposición funcional del sistema que queremos construir. Esta descomposición funcional requiere traducir el dominio del problema en una serie de funciones y subfunciones. El analista debe comprender primero el dominio del problema y a continuación documentar las funciones y subfunciones que debe proporcionar el sistema. El problema es que no existe un mecanismo para comprobar si la especificación del sistema expresa con exactitud los requisitos del sistema.

Flujo de datos. El análisis estructurado muestra cómo fluye la información a través del sistema. Aunque este enfoque se adapta bien al uso de sistemas informáticos para implementar el sistema, no es nuestra forma habitual de pensar.

Sin embargo, la abstracción y la clasificación sí son conceptos que manejamos habitualmente, aunque sea de forma inconsciente.

Modelo de datos. El análisis clásico daba muy poca importancia al almacenamiento de datos. El análisis estructurado moderno incorpora modelos de datos, además de modelos de procesos y de comportamiento. Sin embargo la relación entre los modelos es muy débil, y hay muy poca influencia de un modelo en otro. En la práctica, los modelos de procesos y de datos de un mismo sistema se parecen muy poco. En muchos casos son visiones irreconciliables, no del mismo sistema, sino de dos puntos de vista totalmente diferentes de organizar la solución. Lo ideal sería que ambos modelos se complementasen, no por oposición sino de forma que el desarrollo de uno facilitase el desarrollo del otro.

Ventajas del AOO.

Dominio del problema. El paradigma OO es más que una forma de programar. Es una forma de pensar acerca de un problema en términos del mundo real. El AOO permite analizar mejor el dominio del problema, sin pensar en términos de implementar el sistema en una computadora. El AOO permite pasar directamente el dominio del problema al modelo del sistema.

Comunicación. El concepto OO es más simple y está menos relacionado con la informática que el concepto de flujo de datos. Esto permite una mejor comunicación entre el analista y el experto en el dominio del problema (es decir, el cliente).

Consistencia. Los objetos encapsulan tanto atributos como operaciones. Debido a esto, el AOO reduce la distancia entre el punto de vista de los datos y el punto de vista del proceso, dejando menos lugar a inconsistencias o disparidades entre ambos modelos.

Expresión de características comunes. El paradigma OO utiliza la herencia para expresar explícitamente las características comunes de una serie de objetos. Estas características comunes quedan escondidas en otros enfoques y llevan a duplicar entidades en el análisis y código en los programas. Sin embargo, el paradigma OO pone especial énfasis en la reutilización, y proporciona mecanismos efectivos que permiten reutilizar aquello que es común, sin impedir por ello describir las diferencias.

Reutilización. Aparte de la reutilización interna, basada en la expresión explícita de características comunes, el paradigma OO desarrolla modelos mucho más próximos al mundo real, con lo que aumentan las posibilidades de reutilización. Es probable que en futuras aplicaciones nos encontremos con objetos iguales o similares a los de la actual.

Conceptos básicos.

Las técnicas orientadas a objetos se basan en organizar el software como una colección de objetos discretos que incorporan tanto estructuras de datos como comportamiento. Esto contrasta con la programación convencional, en la que las estructuras de datos y el comportamiento estaban escasamente relacionadas.

Las características principales del enfoque orientado a objetos son, en primer lugar:

Identidad.

Los datos se organizan en entidades discretas y distinguibles llamadas objetos. Estos objetos pueden ser concretos o abstractos, pero cada objeto tiene su propia identidad. Dicho de otra forma: dos objetos son distintos incluso aún en el caso de que los valores de todos sus atributos (p. ej. nombre y tamaño)

coincidan. Dos manzanas pueden ser totalmente idénticas pero no por eso pierden su identidad: nos podemos comer una u otra.

Clasificación.

Los objetos que tengan los mismos atributos y comportamiento se agrupan en clases. Todas las manzanas tienen una serie de atributos comunes: tamaño, peso, grado de maduración, y un comportamiento común: podemos coger una manzana, moverla o comerla. Los valores de los atributos podrán ser distintos para cada una de ellas, pero todas comparten los mismos atributos y comportamiento (las operaciones que se pueden realizar sobre ellas). Una clase es una abstracción que describe propiedades (atributos y comportamiento) relevantes para una aplicación determinada, ignorando el resto. La elección de clases es arbitraria, y depende del dominio del problema. Según esto, una clase es una abstracción de un conjunto posiblemente infinito de objetos individuales. Cada uno de estos objetos se dice que es una instancia o ejemplar de dicha clase. Cada instancia de una clase tiene sus propios valores para sus atributos, pero comparte el nombre de estos atributos y las operaciones con el resto de instancias de su clase.

Polimorfismo.

El polimorfismo permite que una misma operación pueda llevarse a cabo de forma diferente en clases diferentes. Por ejemplo, la operación mover, es distinta para una pieza de ajedrez que para una ficha de parchís, pero ambos objetos pueden ser movidos. Una operación es una acción o transformación que realiza o padece un objeto. La implementación específica de una operación determinada en una clase determinada se denomina método. Según lo dicho, una operación es una abstracción de un comportamiento similar (pero no idéntico) en diferentes clases de objetos. La semántica de la operación debe ser la misma para todas las clases. Sin embargo, cada método concreto seguirá unos pasos procedimentales específicos.

Herencia.

El concepto de herencia se refiere a la compartición de atributos y operaciones basada en una relación jerárquica entre varias clases. Una clase puede definirse de forma general y luego refinarse en sucesivas subclases. Cada clase hereda todas las propiedades (atributos y operaciones) de su superclase y añade sus propiedades particulares. La posibilidad de agrupar las propiedades comunes de una serie de clases en una superclase y heredar estas propiedades en cada una de las subclases es lo que permite reducir la repetición de código en el paradigma OO y es una de sus principales ventajas.

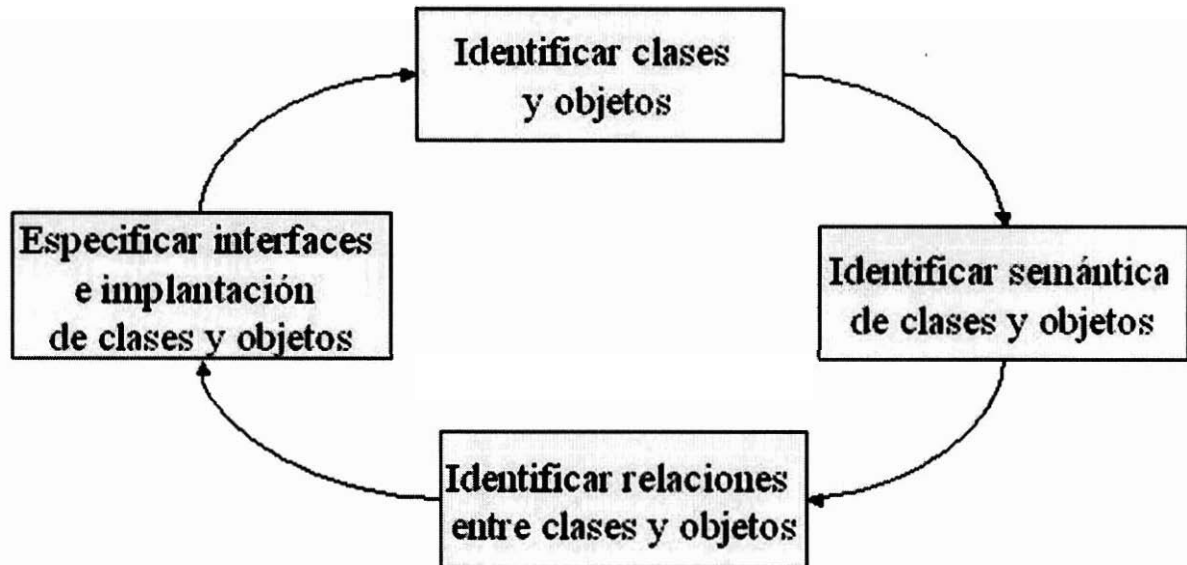
El Microproceso de desarrollo

El microproceso del desarrollo orientado a objetos está dirigido en gran parte por la corriente de escenarios y productos arquitectónicos que emergen del macroproceso y son refinaciones sucesivas. En gran medida, el microproceso representa las actividades diarias del desarrollador individual o de un equipo pequeño de ellos.

El microproceso se aplica igualmente al ingeniero del software y al arquitecto del software. Desde la perspectiva del ingeniero, el microproceso ofrece una guía a la hora de tomar el conjunto de decisiones tácticas que forman parte de la fabricación y adaptación diarias de la arquitectura; desde la perspectiva del arquitecto, el microproceso ofrece un marco de referencia para desplegar la arquitectura y explorar diseños alternativos.

En el microproceso, las fases tradicionales del análisis y diseño son intencionadamente borrosas, y el proceso está bajo un control oportunista. Como observa Stroustrup 1991, "No hay métodos de recetario que puedan reemplazar a la inteligencia, la experiencia y el buen gusto en el diseño y la programación... Las

diferentes fases de un proyecto de software, tales como diseño, programación y pruebas, no pueden separarse estrictamente”. Como ilustra la Figura siguiente El microproceso tiende a seguir las siguientes actividades:



MICROPROCESO DE DESARROLLO DEL ANÁLISIS ORIENTADO A OBJETOS

- Identificar las clases y objetos a un nivel dado de abstracción.
- Identificar la semántica de estas clases y objetos.
- Identificar las relaciones entre estas clases y objetos.
- Especificar interfaces y la implementación de estas clases y objetos.

El macroproceso del desarrollo

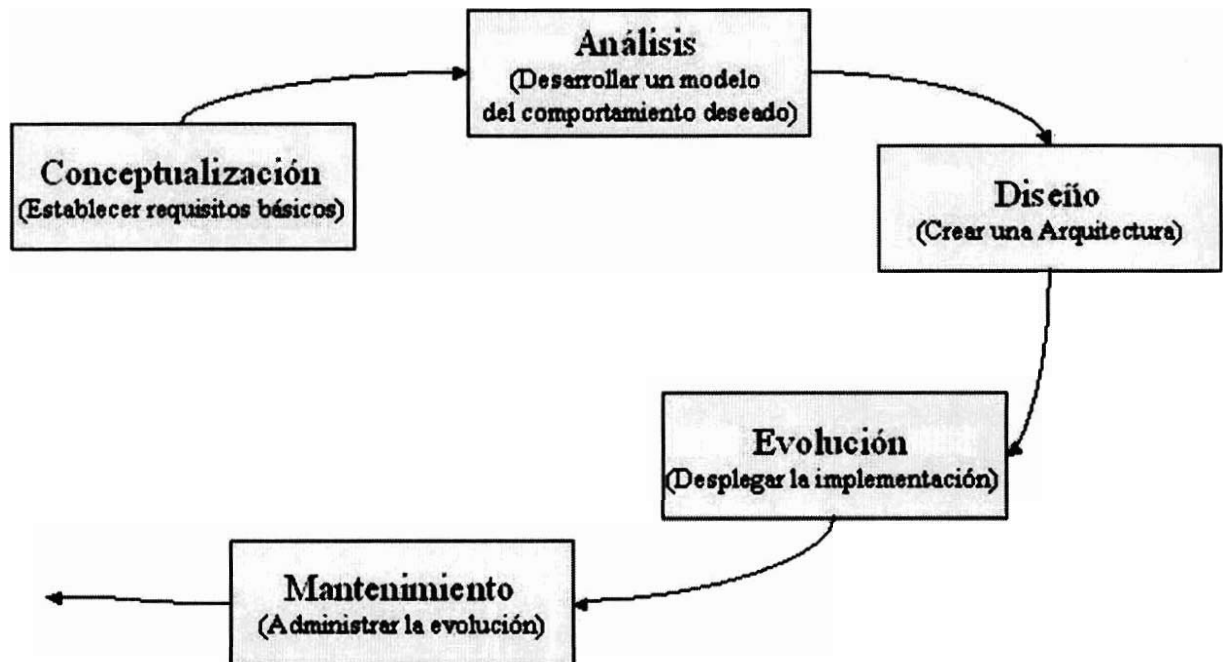
El macroproceso sirve como el marco de referencia para controlar al microproceso. Este procedimiento más amplio dicta una serie de productos y actividades medibles que permiten al equipo de desarrollo tasar el riesgo de forma significativa y realizar correcciones iniciales al microproceso, de forma que se centren mejor las actividades de análisis y diseño del equipo. El macroproceso

representa las actividades del equipo de desarrollo completo en una escala de semanas o meses.

Muchos elementos del macroproceso son simplemente buenas prácticas de desarrollo de software, y por tanto se aplican igualmente a sistemas orientados a objetos y no orientados a objetos. Éstas incluyen prácticas básicas como administración de configuraciones, control de calidad, revisiones de código y documentación. En palabras de David Parnas, ésta es la forma en la que se fingirá un proceso racional de diseño para construir sistemas orientados a objetos.

El macroproceso es principalmente del interés de la dirección técnica del equipo de desarrollo, cuya preocupación principal es sutilmente diferente de la del desarrollador individual. Ambos están interesados en entregar software de calidad que satisfaga las necesidades de usuarios. Sin embargo, a los usuarios finales generalmente podría importarles menos el hecho de que los desarrolladores usen clases parametrizadas y funciones polimórficas con habilidad; los clientes están mucho más interesados en fechas, calidad y completitud, y están en su derecho. Por esta razón, el macroproceso se centra en el riesgo y la visión arquitectónica, los dos elementos gestionables que tienen el mayor impacto en las fechas, calidad y completitud (estado completo).

En el macroproceso, las fases tradicionales del análisis y el diseño se retienen hasta un gran alcance, y el proceso está razonablemente bien ordenado. Como ilustra la Figura siguiente



EL MACROPROCESO DE DESARROLLO DEL ANÁLISIS ORIENTADO A OBJETOS

El macroproceso tiende a atravesar las siguientes actividades:

- Establecer los requisitos centrales para el software (conceptualización).
- Desarrollar un modelo del comportamiento deseado del sistema (análisis).
- Crear una arquitectura para la implementación (diseño).
- Transformar la implementación mediante refinamiento sucesivo (evolución).
- Gestionar la evolución postventa o postentrega (mantenimiento).

Para todo el software de interés, el macroproceso se repite tras las versiones principales del producto. Esto es especialmente cierto en organizaciones que se concentran en la fabricación de familias de programas, que frecuentemente representan una inversión de capital significativa.

La filosofía básica del macroproceso es la del desarrollo incremental. Como lo define Vonk 1990, "en el caso del desarrollo incremental, el sistema en su conjunto se construye paso a paso, y cada versión sucesiva consta de la versión anterior sin cambios más una serie de funciones nuevas". Este enfoque es extremadamente apropiado para el paradigma orientado a objetos, y ofrece una serie de beneficios en relación a la administración del riesgo. Como indica Gilb 1998 con toda corrección, "La entrega evolucionaria se ideó para darnos señales de alarma temprana para verdades inminentemente desagradables".

El Proceso Unificado de Desarrollo de Software.

*“Los desarrolladores de software no trabajan de manera independiente; interaccionan unos con otros y con los usuarios, mientras no tengan un **proceso**, serán como músicos de una orquesta, pero sin partitura”*

**Ivar Jacobson,
Palo Alto, California
Diciembre 1998**

Este proceso es resultado de la conjunción de los esfuerzos de Grady Booch, James Rumbaugh e Ivar Jacobson, así como incorpora numerosas aportaciones de personas y empresas, durante tres décadas de desarrollo, utiliza el paradigma orientado a objetos y tiene su origen en el método Ericsson que evolucionó al Proceso Objectory y posteriormente al Proceso Objectory de Rational y finalmente al Proceso Unificado Rational versión 5.0

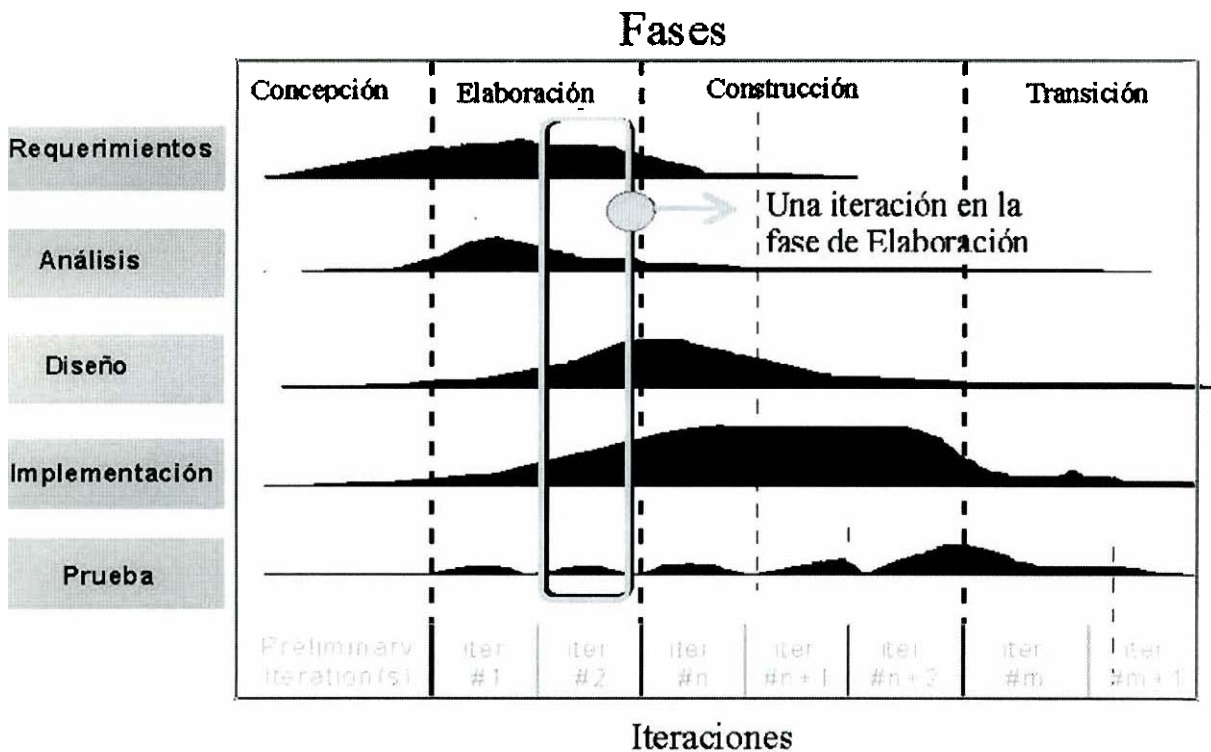
El Proceso Unificado es un proceso de desarrollo de software. Un proceso de desarrollo de software es el conjunto de actividades necesarias para transformar los requisitos de un usuario en un sistema software. Sin embargo, el Proceso Unificado es más que un simple proceso; es un marco de trabajo genérico que puede especializarse para una gran variedad de sistemas software, para diferentes áreas de aplicación, diferentes tipos de organizaciones, diferentes niveles de aptitud y diferentes tamaños de proyecto.

El Proceso Unificado está basado en componentes, lo cual quiere decir que el sistema software en construcción está formado por componentes software interconectados a través de interfaces bien definidas.

El Proceso Unificado utiliza el Lenguaje Unificado de Modelado (Unified Modeling Language, UML) para preparar todos los esquemas de un sistema software. De hecho, UML es una parte esencial del Proceso Unificado sus desarrollos fueron paralelos.

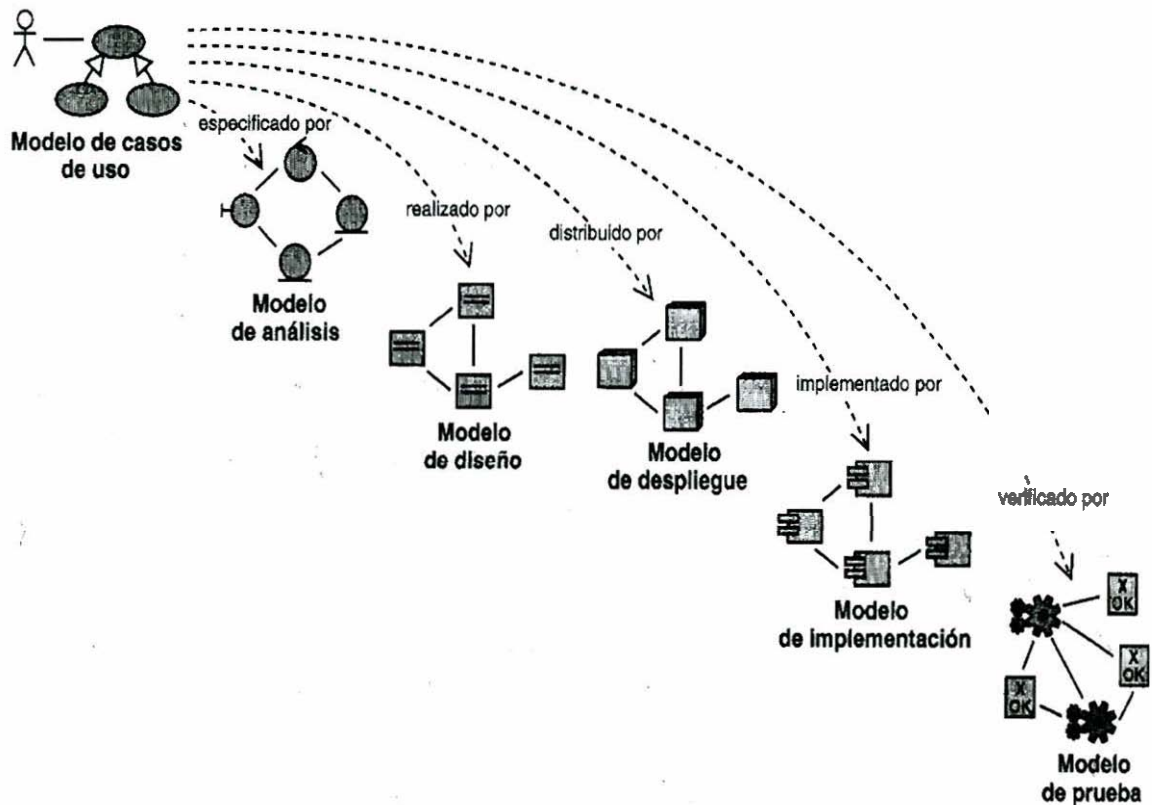
No obstante, los verdaderos aspectos definitorios del Proceso Unificado se resumen en tres frases clave: dirigido por casos de uso, centrado en la arquitectura, e iterativo e incremental. Esto es lo que hace único al Proceso Unificado.

Iteraciones y flujo de trabajo



ITERACIONES Y FLUJO DE TRABAJO DEL PROCESO UNIFICADO

El proceso unificado de desarrollo es dirigido por casos de uso, centrado en la arquitectura e iterativo e incremental, el modelo del proceso unificado se aprecia en la siguiente figura:



EL PROCESO UNIFICADO DIRIGIDO POR CASOS DE USO

Quizás la contribución más significativa del proceso unificado se encuentra en la técnica de modelo de objetos (OMT), técnica que incide directamente en la fase de análisis.

Técnica de modelado de objetos (OMT).

La esencia del desarrollo de software OO es la identificación y organización de conceptos del dominio del problema, más que en su implementación final usando un determinado lenguaje.

La Técnica de Modelado de Objetos (OMT, Rumbaugh, 1991) es un procedimiento que se basa en aplicar el enfoque orientado a objetos a todo el proceso de desarrollo de un sistema software, desde el análisis hasta la implementación. Los métodos de análisis y diseño que propone son independientes del lenguaje de programación que se emplee para la implementación. Incluso esta implementación no tiene que basarse necesariamente en un lenguaje OO.

OMT es una metodología OO de desarrollo de software basada en una notación gráfica para representar conceptos OO. La metodología consiste en construir un modelo del dominio de aplicación e ir añadiendo detalles a este modelo durante la fase de diseño. OMT consta de las siguientes fases o etapas.

Fases.

Conceptualización. Consiste en la primera aproximación al problema que se debe resolver. Se realiza una lista inicial de requisitos y se describen los casos de uso.

Análisis. El analista construye un modelo del dominio del problema, mostrando sus propiedades más importantes. Los elementos del modelo deben ser conceptos del dominio de aplicación y no conceptos informáticos tales como estructuras de datos. Un buen modelo debe poder ser entendido y criticado por expertos en el dominio del problema que no tengan conocimientos informáticos.

Diseño del sistema. El diseñador del sistema toma decisiones de alto nivel sobre la arquitectura del mismo. Durante esta fase el sistema se organiza en subsistemas basándose tanto en la estructura del análisis como en la arquitectura propuesta.

Diseño de objetos. El diseñador de objetos construye un modelo de diseño basándose en el modelo de análisis, pero incorporando detalles de implementación. El diseño de objetos se centra en las estructuras de datos y

algoritmos que son necesarios para implementar cada clase. OMT describe la forma en que el diseño puede ser implementado en distintos lenguajes (orientados y no orientados a objetos, bases de datos, etc.).

Implementación. Las clases de objetos y relaciones desarrolladas durante el análisis de objetos se traducen finalmente a una implementación concreta. Durante la fase de implementación es importante tener en cuenta los principios de la ingeniería del software de forma que la correspondencia con el diseño sea directa y el sistema implementado sea flexible y extensible. No tiene sentido que utilicemos AOO y DOO de forma que potenciemos la reutilización de código y la correspondencia entre el dominio del problema y el sistema informático, si luego perdemos todas estas ventajas con una implementación de mala calidad.

Algunas clases que aparecen en el sistema final no son parte del análisis sino que se introducen durante el diseño o la implementación. Este es el caso de estructuras como árboles, listas enlazadas o tablas hash, que no suelen estar presentes en el dominio de aplicación. Estas clases se añaden para permitir utilizar determinados algoritmos.

Los conceptos del paradigma OO pueden aplicarse durante todo el ciclo de desarrollo del software, desde el análisis a la implementación sin cambios de notación, sólo añadiendo progresivamente detalles al modelo inicial.

Personal Software Process (PSP)

Principio de no certeza de requerimientos:

“Para un nuevo software, los requerimientos nunca serán completamente conocidos, solo hasta después de que el usuario haya utilizado el software”

Watts S. Humphrey,

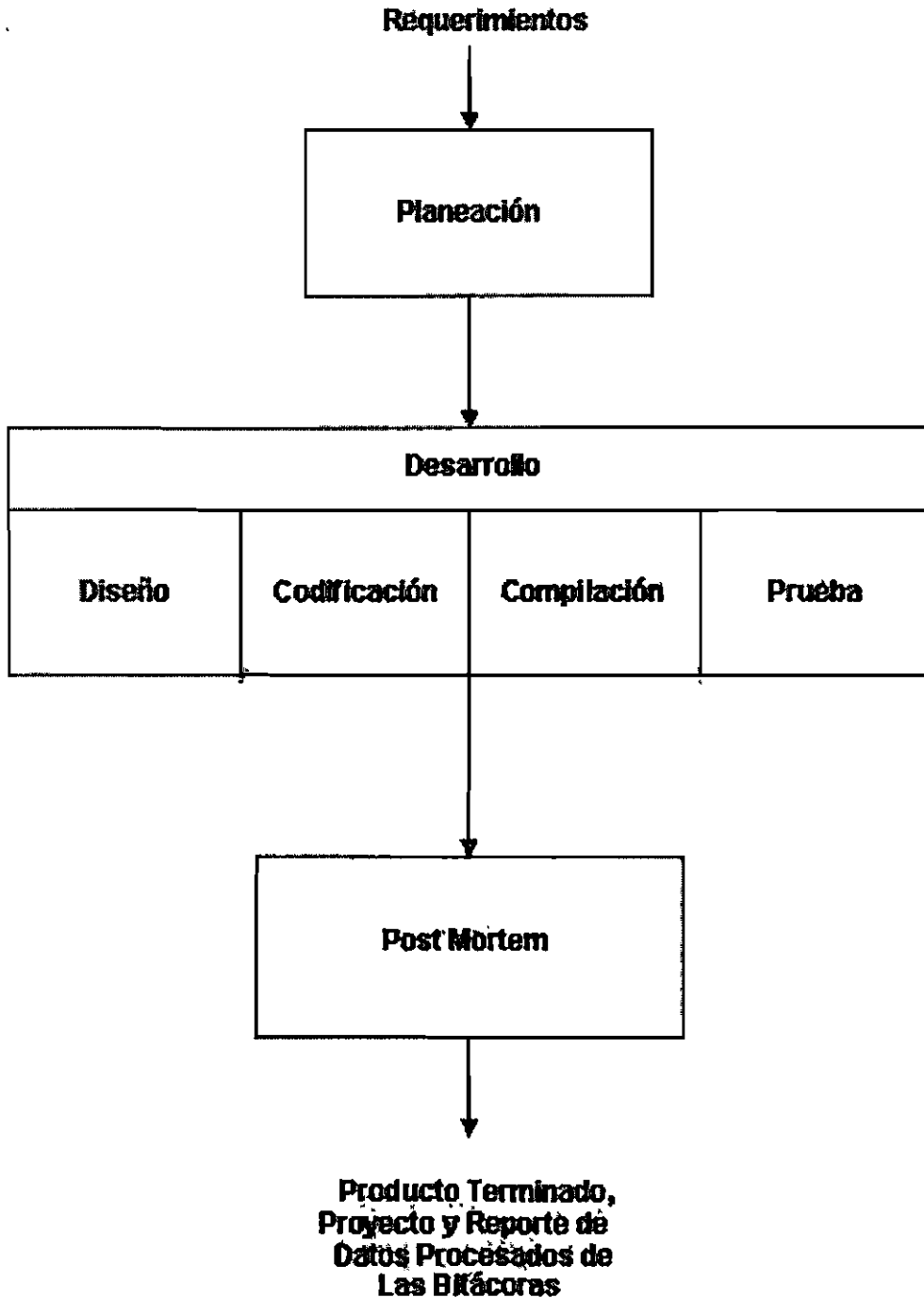
Carnegie Mellon University, 1995

El Proceso Personal de Software (PSP), no es un proceso de desarrollo de software, es un método disciplinario dirigido a Ingenieros de Software que desarrollan productos de software (Humphrey, 1995), basado en la auto mejora personal en el proceso de diseño, de tal manera que apoya el control, administración y mejora de la forma en como, un Ingeniero de Software desarrolla el trabajo para la producción de software. Proporciona un marco estructurado de formas, guías y procedimientos para el desarrollo de software. Utilizado apropiadamente el PSP proporciona un registro histórico necesario para establecer medidas personales de desempeño y compromisos de mejora, de tal suerte que permite conocer el rendimiento actual y predecir un rendimiento futuro de acuerdo a objetivos de mejora definidos.

La evolución del PSP considera cuatro niveles o estadios:

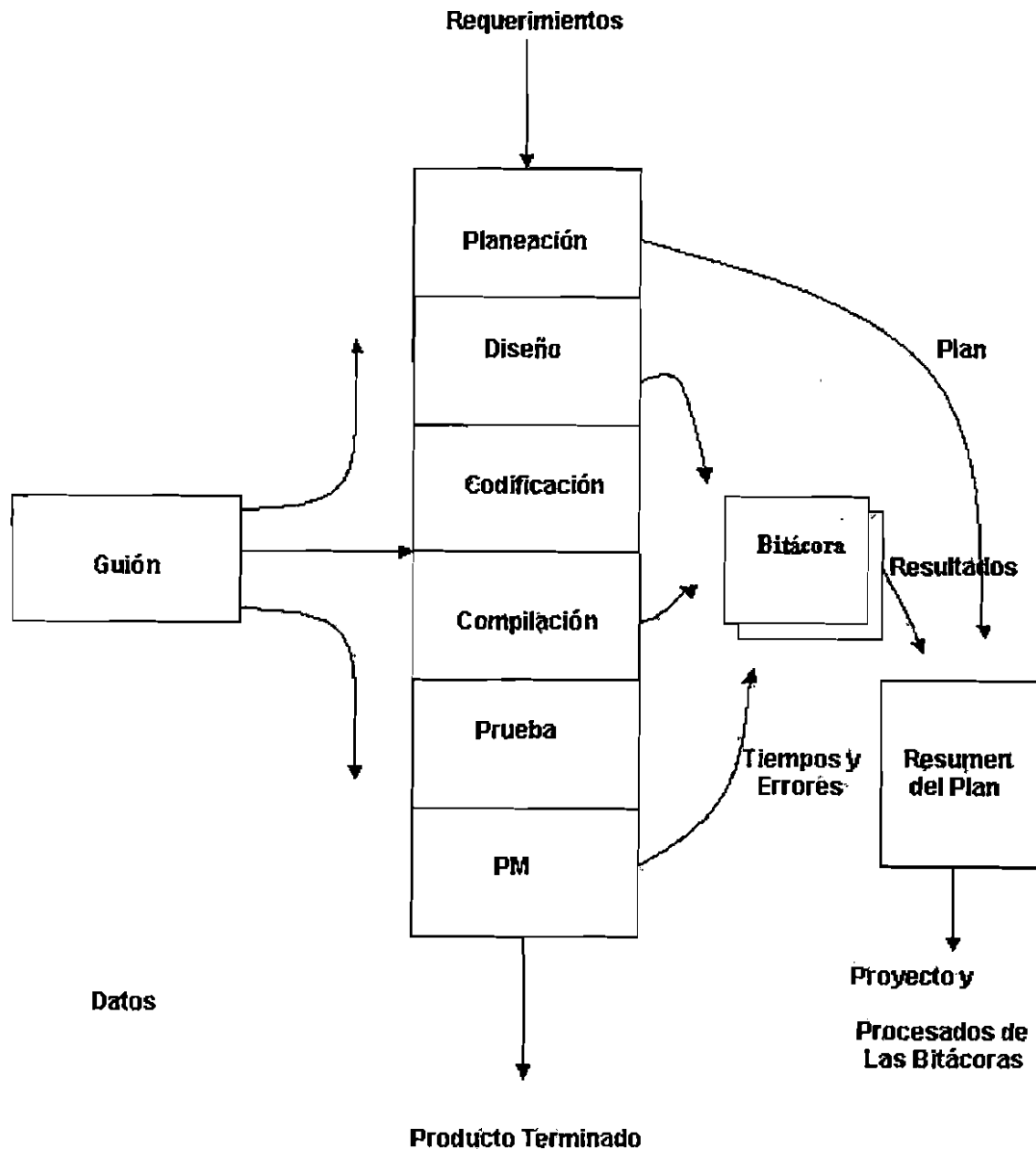
- PSP0.- Línea de base del Proceso Personal.
- PSP1.- Proceso de Planeación Personal.
- PSP2.- Administración Personal de la Calidad.
- PSP3.- Proceso Personal Cíclico.

Para el caso de este trabajo de investigación se presenta la guía para el PSP0, donde se establecen las actividades del Ingeniero de Software, los elementos del proceso PSP0 inician con un requerimiento definido, la planeación, el desarrollo y una fase Post Mortem, como se muestra en la siguiente figura:



ELEMENTOS DEL PROCESO PSP0

De la misma forma establece un flujo del proceso que considera a detalle las fases de planeación, diseño, codificación, compilación, prueba y post mortem que interactúan con los guiones, planes y bitácoras de registro de la actividad del ingeniero de software como se muestra en la figura siguiente:



FLUJO DEL PROCESO PSP0.

Se presentan a continuación los guiones más importantes del flujo de trabajo del proceso PSP0

GUIÓN DEL PROCESO PSP0

Fase	Propósito	Guía para el desarrollo de programas a nivel-módulo
	Requisitos iniciales	<ul style="list-style-type: none"> • Descripción del problema. • Forma PSP0 Resumen del Plan del Proyecto (Project Plan Summary PPS) • Registro de tiempo y defectos en bitácoras (Time Recording Log TRL, Defect Recording Log DRL) • Catálogo de defectos estándar (Defect Type Standard DTS). • Cronómetro (opcional)
1	Planeación	<ul style="list-style-type: none"> • Producir u obtener una definición de requerimientos. • Estimar el tiempo de desarrollo requerido. • Incorporar los datos del plan en la forma PPS. • Registrar el TRL.
2	Desarrollo	<ul style="list-style-type: none"> • Diseño del programa. • Implementar el diseño. • Compilar el programa, corregir y registrar en la bitácora todos los defectos encontrados. • Probar el programa, corregir y registrar en la bitácora todos los defectos encontrados. • Registrar el TRL.
3	Postmortem	<ul style="list-style-type: none"> • Registrar en la forma PPS el tiempo, defectos y tamaño de datos reales.
	Criterios de término	<ul style="list-style-type: none"> • Un programa probado completamente. • PPS completo con los datos estimados y los reales. • Registro completo de tiempo y defectos.

GUIÓN DE PLANEACIÓN PSP0

Fase	Propósito	Guía para el proceso de planeación PSP
	Criterios de entrada	<ul style="list-style-type: none"> • Descripción del problema. • Forma del PPS. • TRL.
1	Requerimientos del programa	<ul style="list-style-type: none"> • Producir u obtener una definición de requerimientos para el programa. • Asegurar que la definición de requerimientos sea clara y no ambigua. • Resolver cualquier duda y preguntas.
2	Estimar recursos	<ul style="list-style-type: none"> • Hacer la mejor estimación de tiempo requerido para desarrollar este programa.
	Criterios de salida	<ul style="list-style-type: none"> • Una definición de requerimientos documentada. • Un PPS terminado con los datos del tiempo estimado de desarrollo. • TRL terminado.

GUIÓN POSTMORTEM PSP0

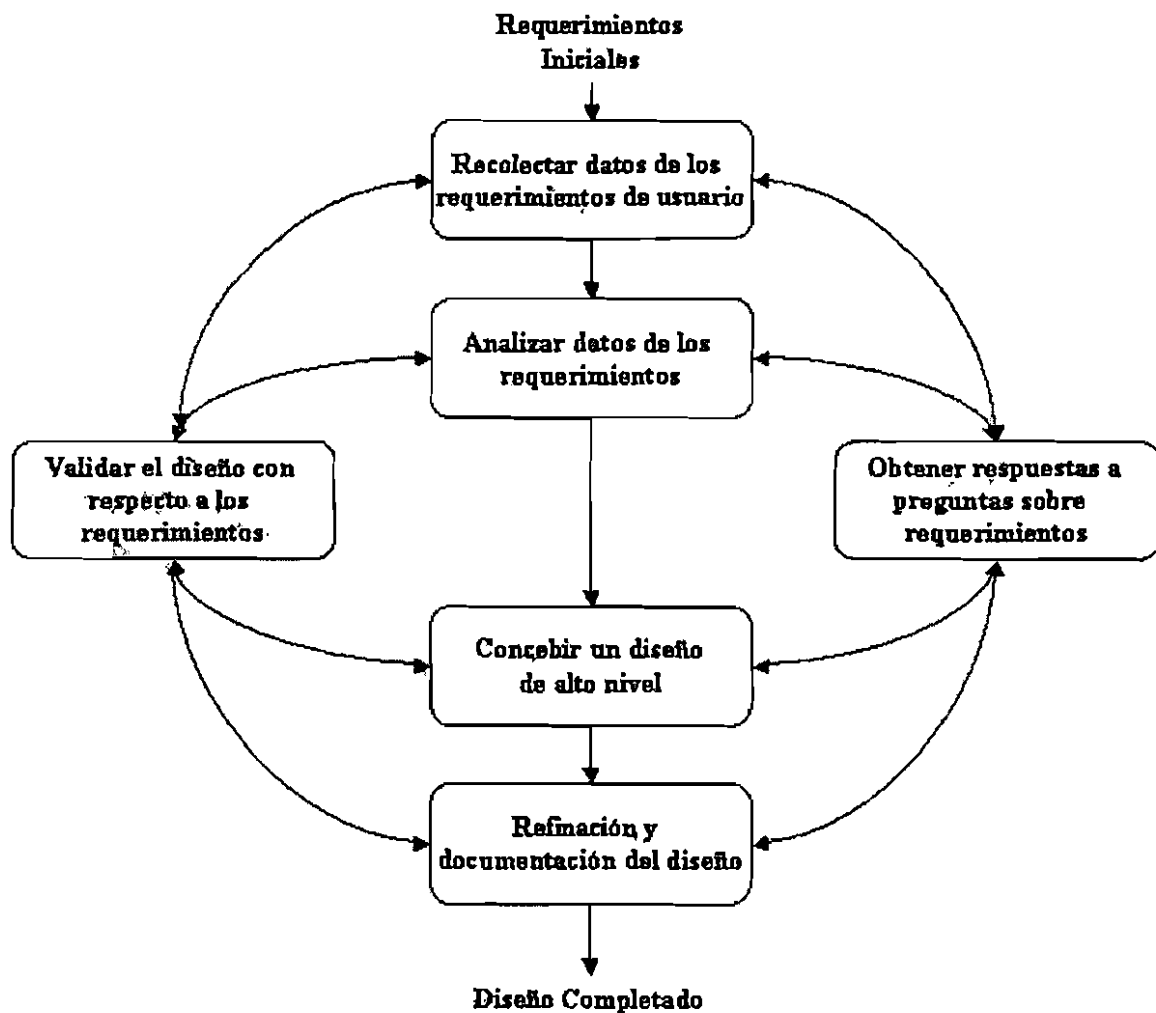
Fase	Propósito	Guía para el proceso de planeación PSP
	Criterios de entrada	<ul style="list-style-type: none"> • Descripción del problema y definición de requerimientos. • PPS con el tiempo de desarrollo planeado. • TRL terminado. • DRL terminado. • Un programa probado y funcionando.
1	Defectos incorporados "injected"	<ul style="list-style-type: none"> • Determinar del DRL el número de defectos incorporados en cada fase del PSP0. • Registrar este número en la forma del PPS en la parte de Defectos Incorporados-Real
2	Defectos eliminados	<ul style="list-style-type: none"> • Determinar del DRL el número de defectos eliminados en cada fase del PSP0. • Registrar este número en la forma del PPS en la parte de Defectos Eliminados-Real.
3	Tiempo	<ul style="list-style-type: none"> • Revisar el TRL terminado. • Registrar el tiempo total invertido en cada fase del PSP0 en la forma del PPS bajo el concepto REAL.
	Criterios de salida	<ul style="list-style-type: none"> • Un programa probado completamente. • Forma del PPS terminada. • Registro de TRL y DRL terminados.

GUIÓN DE DESARROLLO PSP0:

Fase	Propósito	Guía para desarrollo de programas pequeños
	Criterios de entrada	<ul style="list-style-type: none"> Definición de requerimiento PPS con tiempo de desarrollo planeado Registro de TRL Registro de DRL Catalogo de defectos estándar.
1	Diseño	<ul style="list-style-type: none"> Revisión de requerimientos y producir un diseño que los integre Registro de tiempos en TRL y DRL
2	Codificación	<ul style="list-style-type: none"> Implementación del diseño Registrar en el DRL cualquier defecto de diseño o requerimientos encontrado Registro de tiempos en TRL
3	Compilación	<ul style="list-style-type: none"> Compilar el programa hasta que este libre de errores Corregir todos los defectos encontrados Registrar defectos en DRL Registrar tiempos en TRL
4	Prueba	<ul style="list-style-type: none"> Probar hasta que todas la pruebas funcionen sin error Corregir todos los defectos encontrados. Registrar defectos en DRL Registrar tiempos en TRL
5	Criterios de salida	<ul style="list-style-type: none"> Programa probado completamente. DRL terminado TRL terminado.

Para el PSP el diseño es un proceso de aprendizaje en el que se interactúa con el usuario, puesto que el método asume que el requerimiento no es confiable, considera un marco de trabajo para el diseño que incluye la recolección de datos de los requerimientos de los usuarios, la formulación de preguntas que clarifiquen la definición del requerimiento, así como sus respectivas respuestas, de esta manera se concibe un primer diseño de alto nivel que evolucionará como resultado de estar comparando continuamente el diseño contra los requerimientos

del usuario, hasta que el sucesivo refinamiento produce un diseño completo de acuerdo a las necesidades del usuario.

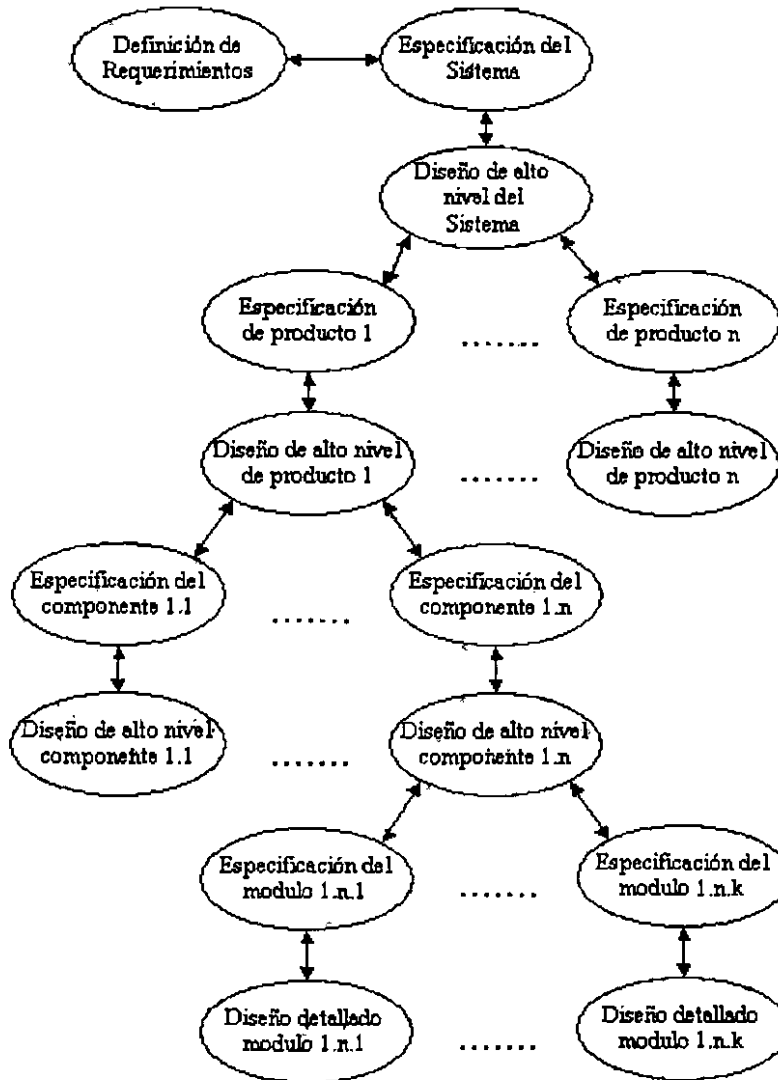


MARCO DE TRABAJO DE LA FASE DE DISEÑO DEL PSP

El PSP define proceso para la estructuración del diseño que asegura la calidad del diseño y el producto final para un producto grande en tamaño, considera las siguientes fases:

- Definición de necesidades (Definición de requerimientos).
- Definición de la solución (Especificación del sistema).
- Conceptualización de las soluciones (Diseño de alto nivel del sistema).
- Dividir el trabajo (Especificación del producto)

- Definir el diseño del producto (Diseño de alto nivel del producto).
- Separar el producto en componentes (Especificación de componentes).
- Definir el diseño de componentes (Diseño de alto nivel de componentes).
- Separar el componente en módulos (Especificación de módulos).
- Detallar la solución (Diseño detallado de módulo).
- Implementar la solución (Implementación de módulos y prueba)



EL CICLO REQUERIMIENTOS-ESPECIFICACIÓN-DISEÑO

Los estados del tamaño del producto a desarrollar.

- **Estado 0.-** Programas muy pequeños, pueden ser programados solamente por un Ingeniero de Software.
- **Estado 1.-** Programas pequeños o módulos diseñados, codificados y probados por un solo Ingeniero de Software.
- **Estado 2.-** Programas grandes o componentes, que típicamente involucra a equipos de desarrollo quienes desarrollan e integran múltiples módulos de Estado 1.
- **Estado 3.-** Proyectos de desarrollo muy grandes que involucra múltiples equipos de desarrollo controlados y dirigidos por una administración central del proyecto.
- **Estado 4.-** Multi sistemas masivos que involucran muchos proyectos autónomos e independientes.

III. METODOLOGÍA

La formulación del problema tiene su origen en la "crisis del software", abordado en los capítulos iniciales, actualmente no se ha encontrado una solución a esta crisis, que desde mi opinión ya no es crisis, sino enfermedad crónica de la industria de la producción del software, en este orden de ideas, el problema sobre el cual se pretende incidir es precisamente esta crisis, partiendo del hecho que actualmente desarrollar software, tiene un grado de complejidad elevado, debido en gran parte al resultado de una falta de definición precisa de las operaciones básicas de la Ingeniería de Software, que en un marco de desarrollo le permitan al ingeniero manipularlas y controlarlas facilitando la complejidad de los procesos de desarrollo de software, la pregunta a responder es ¿Cómo hacemos para reducir la complejidad de los procesos de desarrollo de software?. En tal sentido plantearemos la siguiente:

Hipótesis

Es posible reducir la complejidad de los procesos de la Ingeniería de Software con la identificación de las operaciones unitarias que los componen, estableciendo con esta identificación un marco de trabajo comprensible y controlable para el desarrollo de productos de software.

En tal sentido el método a utilizar corresponde al estudio cualitativo en una línea de investigación, tomando como sujetos de estudio tres métodos de desarrollo de software con la finalidad de inducir la existencia (no definidas como tales) de las operaciones unitarias en las metodologías de desarrollo de software, los sujetos de estudio para este caso serán métodos conocidos de desarrollo de software designados como unidades de estudio seleccionadas son: los documentos de los métodos de Análisis Estructurado propuesto por Tom de Marco, El Análisis y Diseño Orientado a Objetos propuesto por Grady Booch y el Proceso Unificado de Desarrollo de Software propuesto por Graddy Booch, Ivar

Jacobson y James Rumbaugh, dando especial énfasis en las fases de análisis y la recolección de requerimientos y la orientación hacia la fase de análisis de este trabajo de tesis. Se analizará a su vez el método disciplinario PSP desde el punto de vista que es el único método que designa actividades precisas y definidas para el Ingeniero de Software.

Preguntas Formuladas para los Sujetos de Estudio:

1. ¿Cómo abordan el análisis?
2. ¿Qué actividades y técnicas utilizan para recopilar información?
3. ¿Qué actividades y técnicas utilizan para analizar la información?
4. ¿Cuáles son los resultados de la fase de análisis?
5. ¿Qué actividades y técnicas utilizan para especificar los requerimientos?
6. ¿Cómo y cuál es la unidad mínima, hasta la cual el proceso de descomposición del problema analizado se detiene?
7. ¿De qué manera se efectúa la operación que es utilizada para la descomposición del problema?
8. ¿De qué manera se efectúa la operación que se utiliza para la composición de la solución?
9. ¿Cómo diferencian los requisitos funcionales de los no funcionales?

Tomando como referencia el material revisado en la sección II anteriormente relativo a la fase análisis y estudiando detalladamente la documentación de este método, se llegó a contestar las preguntas para este sujeto de estudio, las respuestas resultado del estudio documental son:

Respuestas en el Análisis y Diseño Estructurado.

1.	A través de "encuestas" a los usuarios y formulando modelos físicos y lógicos del sistema actual y generando modelos físicos y lógicos de un sistema solución propuesto, incorporando los diagramas esquemáticos estructurados.
2.	El método cuenta con un apéndice para las encuestas y lo titula "Técnicas de entrevistas y recolección de datos", donde justifica la recopilación de datos, define tipos de entrevistas, los problemas comunes al realizar entrevistas, reglas para hacer entrevistas, formas de validar la información recolectada y menciona cinco formas alternativas de recopilación de datos: Cuestionarios, Presentaciones de proveedores, Visitas a otras instalaciones, recolección de datos con documentos de la organización e investigación externa.
3.	Es un proceso no formalizado específicamente del estudio del resultado de las encuestas y el proceso de modelación y diagramación de los modelos actuales y los propuestos para la solución.
4.	Son los diagramas esquemáticos de transición de estados, diagramas de flujos de datos y diagramas entidad relación principalmente, así como textos explicativos de los diagramas incluyendo un diccionario de datos.
5.	La actividad es netamente de diagramación por lo que la especificación es mayormente gráfica incluyendo pseudo-código estructurado para los procesos de control y documentos incorporados a los diagramas principalmente explicativos
6.	No definen una unidad mínima en la cual el proceso de descomposición se detiene, solamente en las reglas practicas de los DFD mencionan <i>"Dividir el sistema de manera natural. No por empeñarnos en descomponer el sistema va a estar mejor analizado. Sólo debemos descomponer si realmente es necesario y cuando la propia naturaleza del sistema lo determine"</i> , de la misma forma mencionan <i>"Limitar el número de procesos dentro de un nivel de descomposición determinado. número ideal 2 ± 7"</i> y finalmente reforzando la primera recomendación practica <i>"Explosionar cuanto sea necesario. El sistema no va a ser más complicado porque tenga más niveles, todo lo contrario"</i> . Regularmente los últimos elementos tienen un acceso directo a los almacenes de datos.
7.	Se encuentra implícita en los diagramas de flujo de datos cuando se divide y descompone el problema en sub-problemas, denominándolo análisis descendente (top-down)
8.	Se encuentra implícita en el proceso de composición que llaman implantación ascendente (down-top), donde las diferentes partes del problema se encuentran interrelacionadas con conectores o almacenes de datos (los autores indican que es una de las debilidades del método y que muy probablemente el concepto se tomó prestado originalmente de la industria de hardware).
9.	No existe una diferenciación precisa de esta clasificación de los requisitos.

Respuestas en el Análisis Orientado a Objetos.

1.	El microproceso lo aborda de una manera implícita, considerando identificar las clases y objetos, su semántica, relaciones, interfaces e implementación, el macroproceso de una manera explícita a través del desarrollo de un modelo de comportamiento deseado del sistema.
2.	El método no especifica ninguna técnica o actividad de recopilación de información, por comentarios de los autores se asumen que utilizan las mismas técnicas de los procesos estructurados y que no requieren ninguna adecuación propia para este paradigma.
3.	Utilizan técnicas de identificación de clases y objetos, su semántica, relaciones e interfaces, en estricto sentido esta parte de análisis es considerada como una actividad individual del analista en el microproceso de desarrollo.
4.	Los resultados de la fase de análisis desde el punto de vista del microproceso son un diccionario de objetos y clases, abstracciones de la semántica de clases y objetos, diagramas de objetos y clases, especificaciones de objetos y clases refinadas. Desde el punto de vista del macroproceso se obtienen prototipos, descripciones funcionales del sistema y su descomposición narrativa mediante escenarios.
5.	La especificación de requerimientos es mayormente orientado a los datos (objetos), existen actividades muy puntuales en cada paso relativas a la especificación de objetos, clases, semántica, asociaciones e interfaces. La técnica y notación comúnmente utilizados implican cuatro modelos que se interrelacionan, el modelo lógico, modelo físico, modelo estático, modelo dinámico, en donde se especifican las estructuras de clases y objetos, así como las posibles arquitecturas de solución.
6.	La unidad mínima de descomposición de los objetos es hasta que un objeto se considera parte de una clase, pero este ya no puede ser una superclase, desde el punto de vista funcional son aquellas operaciones que acceden directamente al objeto.
7.	A través de un proceso de identificación que implícitamente considera una operación de separación o distinción, que obliga a establecer los objetos en determinada clase.
8.	La composición de la solución está determinado por un sólido esquema diagramático de clases y subclases, de tal forma que asegura el acceso a los objetos a través de sus propiedades de herencia y polimorfismo.
9.	Los requisitos no funcionales son abordados desde un punto de vista de requisitos de calidad previstos por el usuario y que deben de ser considerados como parte de la solución final.

Respuestas en el Proceso Unificado de Desarrollo.

1.	El Análisis es abordado de manera explícita como un flujo de trabajo fundamental, parte del hecho de que es una actividad independiente a la extracción y captura de requerimientos, así como de la expresión de los mismos como casos de uso.
2.	Es considerado como un flujo de trabajo fundamental denominado captura de requisitos, y lo define como el proceso de averiguar lo que se debe de construir, implícitamente menciona la interacción con usuarios para la extracción de la información y utiliza técnicas explícitas como desarrollo de modelos de dominio, de negocio y búsqueda y modelación de casos de uso.
3.	La actividad primordial es el análisis de requisitos para refinarlos y estructurarlos, en el contexto general del sistema, a través de modelos de análisis basados en un modelo de objetos conceptual
4.	Los resultados del análisis son: a).- Especificación precisa de los requisitos que incluye casos de uso, b).- Modelo de análisis utilizando lenguaje de los desarrolladores, este ultimo se considera una primera aproximación al modelo de diseño.
5.	Se utilizan técnicas de especificación diagramáticas basadas en la notación y estándares del Lenguaje Unificado de Modelado (UML)
6.	No determina explícitamente la manera en que un caso de uso (determinado como un fragmento de funcionalidad del sistema) ya no es susceptible de fragmentarse, esencialmente se determina la manera de identificarlos y modelarlos, esta identificación permite dirigir todo el proceso de desarrollo a partir del modelo de casos de uso. De manera textual y como recomendación menciona el siguiente criterio: "Hay que recordar que intentamos crear casos de uso fáciles de modificar, revisar probar y manejar unitariamente ", (Rumbaugh et al, 1999)
7.	Se basa esencialmente en la búsqueda a través de la identificación de los casos de uso, aplicando los criterios tales como: a).- Identificar forma en que los actores usan el sistema, b).- Identificar un resultado de valor para cada actor, c).- Identificarlo como un fragmento de funcionalidad del sistema, d).- Identificar una secuencia de acciones interactuando con el actor. Así mismo se generan candidatos de caso de uso, que pueden fragmentarse o desecharse.
8.	La composición de la solución está implícita en la modelación puesto que cada uno de los modelos tiene relacionados sus componentes, en términos de la fase de análisis, uno de sus objetivos es "estructurar" los requisitos que implica una composición de la solución, máxime que esto implica un primera aproximación al modelo de diseño.
9.	Todos los requisitos funcionales se expresan por casos de uso y los requisitos no funcionales están determinados por atributos de calidad y están relacionados a los requisitos funcionales, pueden estar relacionados de una manera específica para un caso de uso, o pueden ser comunes a varios casos de uso, generalmente se les denomina requisitos adicionales, en el entendido que son necesariamente no funcionales.

Respuestas en el PSP

1. El proceso está centrado en el diseño, más sin embargo considera que los requerimientos iniciales no son confiables, define un principio de no certeza de los requerimientos, así que establece criterios para validarlos interactuando con el usuario, a través de un marco de trabajo definido para la fase de diseño.
2. Recepción de requerimientos iniciales y formulación de preguntas encaminadas a refinar el diseño de alto nivel, para descubrir las necesidades reales de los usuarios y para asegurar la comprensión del diseñador de las necesidades del usuario.
3. El análisis de información se realiza en paralelo a la concepción del diseño, podemos decir que el Ingeniero de Software, está diseñando en base al análisis que está realizando, desde este punto de vista cada requerimiento es visualizado como la descripción de un problema sin solución, cuando la solución es descrita se considera que es la especificación de diseño del requerimiento.
4. El resultado último del análisis es la especificación de diseño de alto nivel, pero se observan subproductos, tales como especificaciones de requerimientos y despiece del diseño de alto nivel.
5. Las actividades están ligadas necesariamente a las actividades de diseño, de tal manera que la especificación de requerimientos es un paso previo obligado a cada especificación de diseño, recomienda el uso de gráficos y textos que auxilien al Ingeniero de Software.
6. La unidad mínima de acuerdo al Ciclo Requerimientos-Especificación-Diseño es un *módulo*, que a su vez es componente de un *producto*, que a su vez es parte de una *especificación de sistema* que está asociado a un *requerimiento*, de esta forma el *módulo* es definido para un diseño de nivel detallado que es implementable en un estado 0 o 1, que es manejable y programable por un solo Ingeniero de Software, considera una notación de diseño detallado con especificación lógica y de estado asociado a un código fuente y a un escenario operacional y una especificación de funcionalidad.
7. La descomposición del problema se realiza a través de estrategias de diseño, dependiendo del tamaño del sistema a desarrollar valorado por los cinco estados definidos, maneja el concepto de desintegración definido como una abstracción de diseño consistente en la sucesiva desintegración del sistema en piezas "manejables"
8. La operación la denomina implementación, es opuesta a la desintegración, por lo que también se le conoce como integración del producto, y consiste en construir e integrar las piezas manejables en un conjunto coherente.
9. Los requisitos funcionales son considerados en los *módulos*, los no funcionales en el diseño de alto nivel cumpliendo requisitos de calidad.

IV. RESULTADOS Y DISCUSIÓN

Al analizar de manera general los métodos especificados, los elementos coincidentes nos llevan a las siguientes inducciones:

Elementos Coincidentes Generales:

Los métodos pueden analizarse como un conjunto de entidades relacionadas entre sí y como una estructura de procedimientos que sintetizan un flujo de procesos generador de resultados.

Las entidades primarias, objeto de modelado detallado, describen las actividades, tareas, estados, funciones y productos de todo método en general. Sus relaciones articulan la estructura, los procedimientos (técnicas) y los resultados de cada método particular.

La estructura del método se acopla al ciclo de desarrollo del sistema deseado, y para alcanzar éste, sigue un proceso organizado por pasos.

La organización del proceso se basa en descripciones de los modelos sucesivos realizadas en los lenguajes abstractos apropiados a cada paso y entendibles por el receptor adecuado.

El ciclo de producción (llamado en algunos casos ciclo de vida) tiene tres grandes grupos de fases:

- Pre ejecutor (planificación, especificación, análisis),
- Ejecutor (diseño, codificación, pruebas)
- Post ejecutor (documentación, capacitación de usuarios, mantenimiento, rectificaciones);

No todos los métodos incluyen todas las fases en su estructura, y otros incluyen fases paralelas al desarrollo, como gestión del proyecto, disciplinarios, de su calidad y configuración.

Las fases se subdividen en subfases (con distintos nombres, como módulos, etapas, etc.), hasta que una habitual y coincidente estructura arborescente alcanza las tareas, actividades elementales realizadas en condiciones atómicas (un objetivo, un tipo de actor, un puesto, un módulo, etc.).

A ciertas fases o tareas que suponen ensanchamiento de opciones (para buscar los requerimientos) les siguen otras de restricción (con valoraciones y decisiones que buscan soluciones específicas).

Cada tarea se apoya en un procedimiento que prescribe su forma de ejecución y es vehículo de comunicación entre los conocedores de la realidad original, sus modeladores y los usuarios del sistema final.

El procedimiento obtiene un resultado a partir de los resultados de otros procedimientos previamente cumplimentados.

El conjunto de resultados finales o productos (código ejecutable, especificaciones, descripciones, informes) constituye el sistema deseado, si se prueba su consistencia interna (entre sí) y externa (respecto a los resultados deseados).

Las reglas de un procedimiento, al menos textuales y a menudo mixtas (diagramáticas y algorítmicas), se estabilizan en un número de técnicas que son comunes a muchos métodos y a menudo se profundizan al margen de éstos. Por ejemplo, la definición de un diagrama de flujo de datos (DFD) es un procedimiento para representar los requerimientos de los sistemas de información, con una

técnica de diagramación, visual, compacta y no tan ambigua como el texto, que forma parte de varios métodos.

Las herramientas CASE asociadas como Rational Rose, BP-Win, Er-Win, PSP-Studio y cada vez más entornos (para marcar su exterioridad a los métodos), informatizan las técnicas diagramáticas, algorítmicas y textuales: no sólo formalizan documentaciones, sino que deben contener los mecanismos rutinarios formalizados para verificar las consistencias de las técnicas soportadas. Según el alcance de las técnicas que cubra la herramienta.

Elementos Coincidentes en la Fase de Análisis:

Derivado de los resultados de cada pregunta podemos inducir que los elementos coincidentes en particular del análisis son:

1. En la forma de abordar el análisis es a través de actividades de Extracción de Información del mundo real, proceso de análisis por identificación de funcionalidades, desecho de información innecesaria, modelación, estructuración e inclusive diseño del sistema.
2. Los cuatro métodos consideran de forma explícita la interacción con los usuarios, no importando la técnica que se utilice, siempre y cuando nos lleve a la identificación y validación eficaz de requisitos.
3. La actividad propiamente de análisis de información recopilada es coincidente que es una actividad solitaria e individual y de discernimiento para identificar y separar la información necesaria y suficiente para especificar modelos diagramáticos y de diseño.

4. Los resultados del análisis son documentos que contienen diagramas y textos explicativos que expresan requerimientos puntuales, su estructura y relaciones con las fases posteriores.
5. La especificación de requerimientos es fundamentalmente diagramático con textos explicativos y semi-formalizada por uso de lenguajes o estándares de modelado y diseño, solamente el PSP expresa un requerimiento en términos de expresiones matemáticas.
6. Para determinar en que momento se detiene la descomposición se observó que en el Análisis Estructurado se deja a criterio y experiencia del analista, para el Análisis Orientado a Objetos, es muy preciso que el análisis se detiene cuando no se pueden determinar elementos de una posible clase y para el proceso unificado recomienda que los casos de uso deben ser susceptibles de manejarse "unitariamente", para el PSP lo define cuando la pieza es lo suficientemente pequeña como para que sea manejable y factible de solucionar o programar por un solo Ingeniero de Software, por lo cual inducimos que existen criterios no del todo explícitos, pero que determinan en qué momento se debe de detener el proceso de descomposición.
7. La actividad de separación se induce que existe en las tres primeras metodologías, pero en ningún caso estrictamente formalizada de manera explícita, para el PSP -la separación si existe explícitamente y la denomina desintegración del producto a través del ciclo requerimiento-especificación-diseño.
8. Para los cuatro casos, el proceso de composición se efectúa cuando se refinan y estructuran los modelos de diseño, a través de las relaciones entre las funcionalidades identificadas y que generan un conjunto coherente.

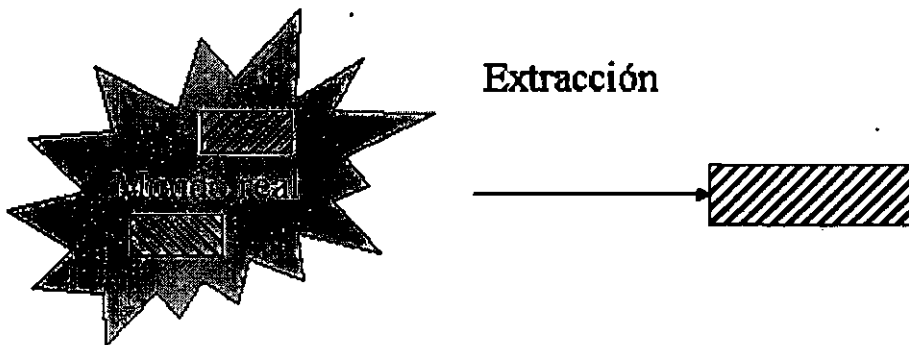
9. Se observó que la separación de requisitos funcionales de no funcionales, está determinado en función de la evolución histórica, aparejado a la necesidad de producir software con ciertos atributos de calidad.

Las operaciones unitarias descubiertas por inducción de la fase de análisis de la Ingeniería de Software son hasta este momento:

Operaciones Unitarias de la Fase de Análisis

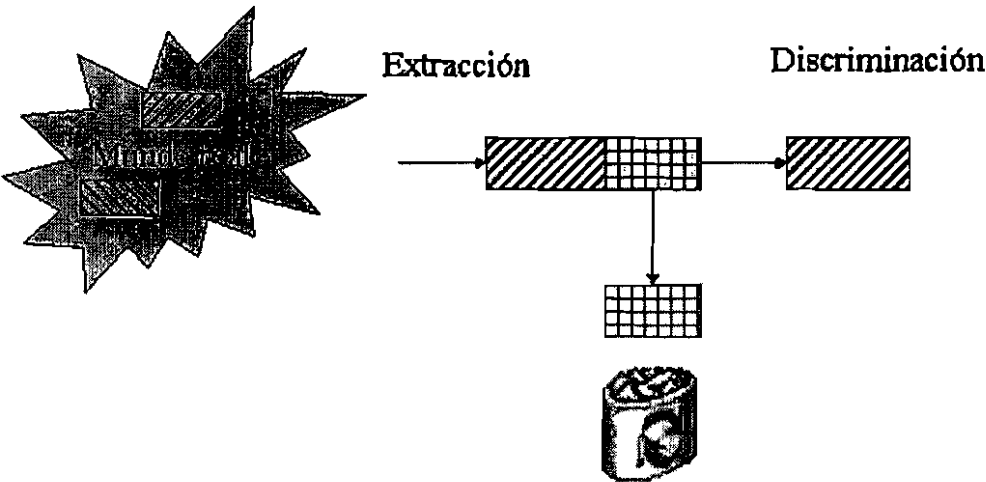
- Extracción.
- Discriminación.
- Separación.
- Composición.
- Abstracción.

Extracción: Actividad mediante la cual el ingeniero de software investiga y recopila información acerca del problema a resolver, esta operación se puede llevar a cabo con técnicas como la entrevista, cuestionarios, sesiones JAD, marcos de trabajo, etcétera. En esta operación recopila toda la información que considera le puede orientar sobre la solución del problema o área de oportunidad, en este sentido, en la fase de extracción no existen límites claros sobre cual es precisamente la información necesaria para la solución del problema.



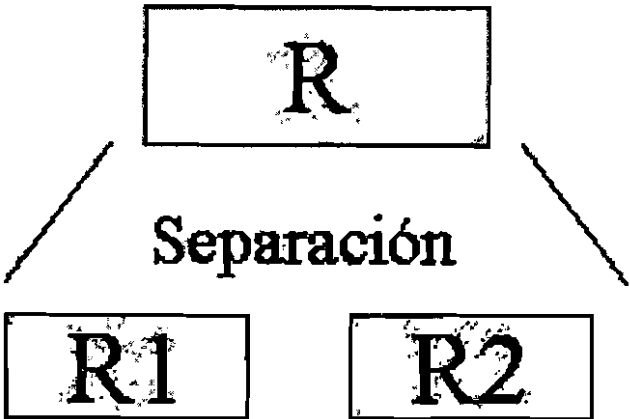
OPERACIÓN UNITARIA DE EXTRACCIÓN

Discriminación: Operación mediante la cual con la aplicación de uno o varios criterios del analista o Ingeniero de Software, desecha información obtenida en el proceso de extracción, en el más estricto de los sentidos es la operación contraria a la extracción.



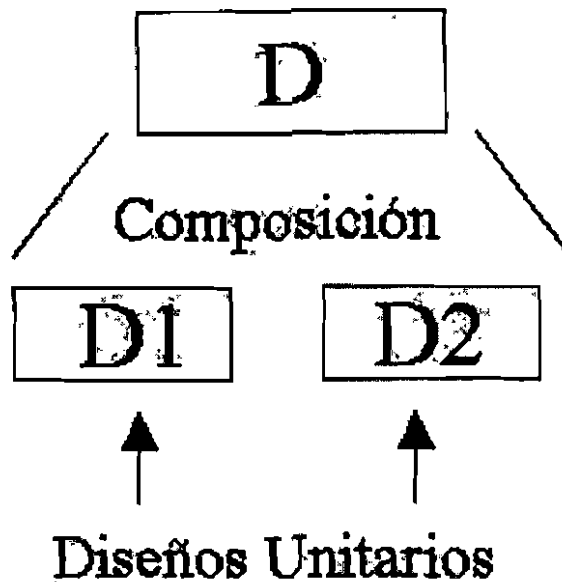
OPERACIÓN UNITARIA DE DISCRIMINACIÓN

Separación: Operación mediante la cual se descompone un requerimiento en dos requerimientos.



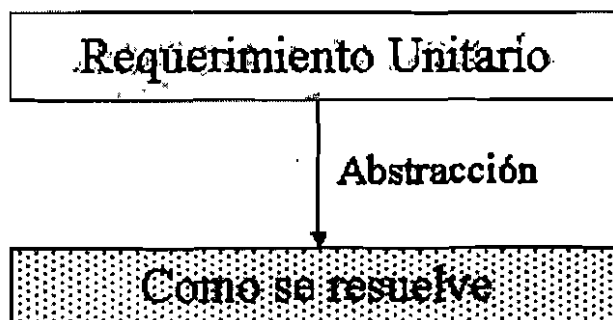
OPERACIÓN UNITARIA DE SEPARACIÓN

Composición: Operación a través de la cual dos requerimientos unitarios acoplan su diseño y solución cumpliendo con un requerimiento compuesto.



OPERACIÓN UNITARIA DE COMPOSICIÓN

Abstracción: Operación individual mediante la cual el Ingeniero de Software determina "como" el requerimiento unitario es resuelto.



OPERACIÓN UNITARIA DE ABSTRACCIÓN

Operaciones Compuestas de la Fase de Análisis

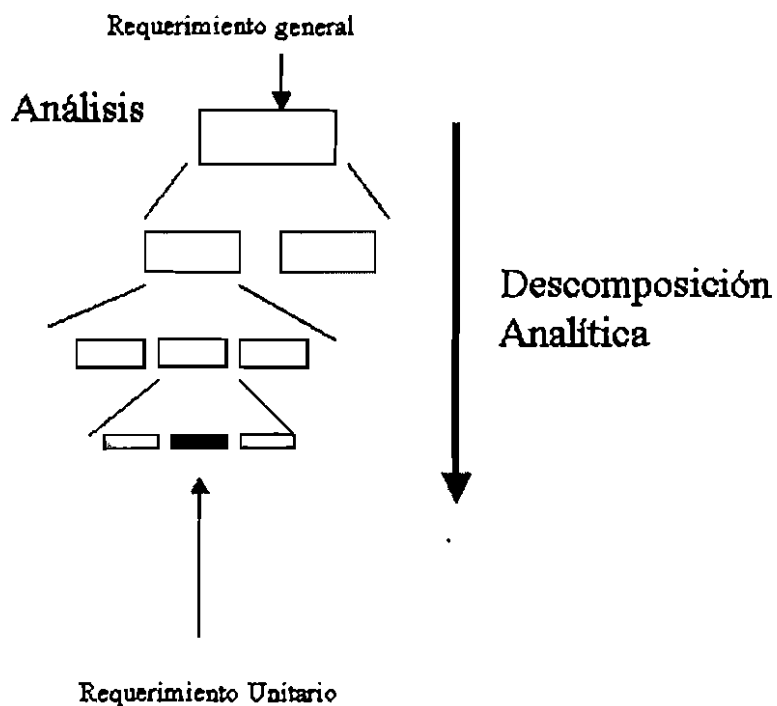
- Descomposición analítica.
- Composición sintética.

Si hacemos una revisión de las reglas de la Chemical Abstract Machine encontramos coincidencias en las reglas de calentamiento de enfriamiento

Reglas de Calentamiento.- Descomponen (separan) moléculas en sus elementos,

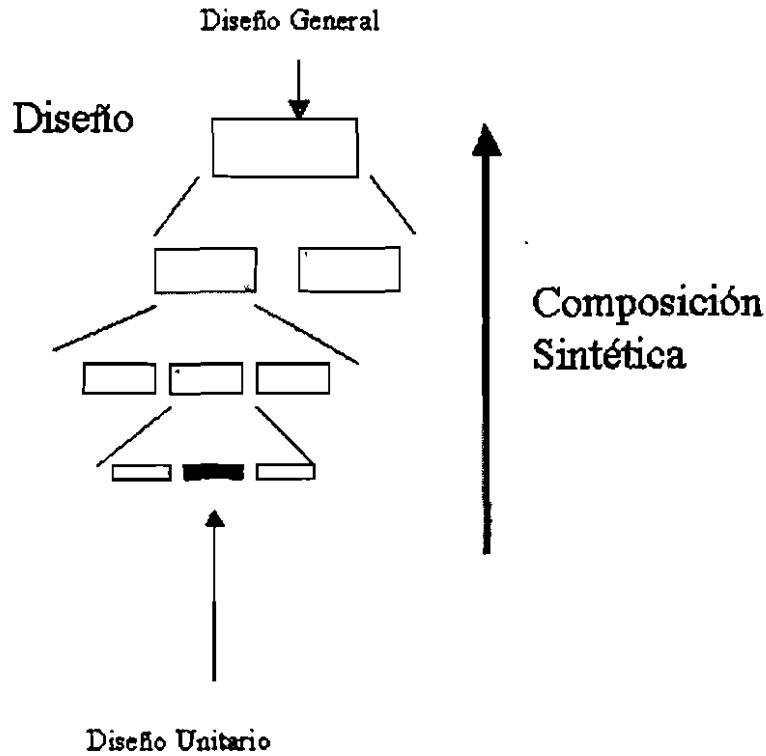
Reglas de Enfriamiento.- Composición de moléculas a partir de elementos,

Descomposición analítica: Actividad mediante la cual se despieza de un requerimiento general (compuesto) en requerimientos cada vez más específicos, partimos de la base que la descomposición analítica es una operación compuesta y que consiste en la ejecución sucesiva de la operación básica de separación.



OPERACIÓN COMPUESTA DE DESCOMPOSICIÓN ANALÍTICA

Composición sintética: Actividad de síntesis mediante la cual los requerimientos unitarios van componiendo la solución general del sistema, esta es una operación compuesta a partir de la operación de composición y es la operación opuesta a la descomposición analítica.



OPERACIÓN COMPUESTA DE COMPOSICIÓN SINTÉTICA

Pruebas de sobre productos de operaciones unitarias

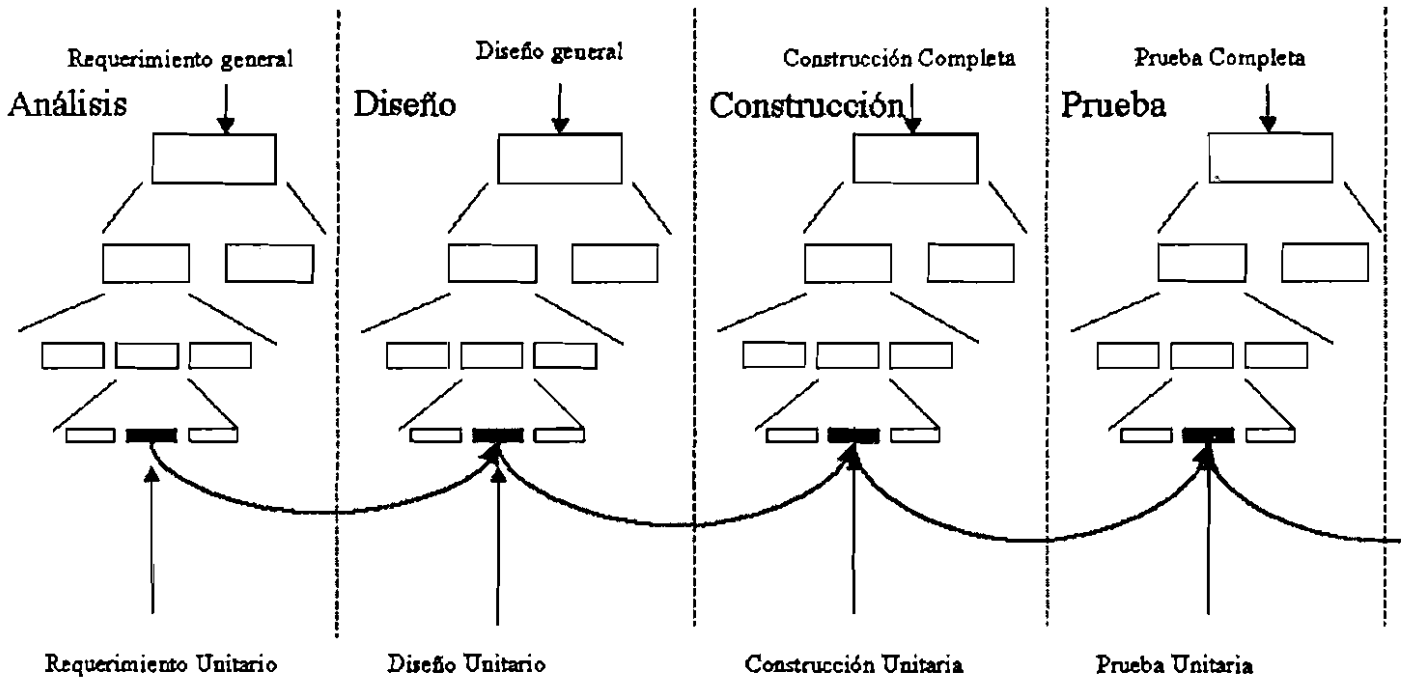
Determinación de información necesaria.

Determinación de requerimiento unitario.

Determinación de factibilidad de requerimiento unitario.

De validación con el usuario.

Desde este punto de vista la fase de análisis y su correspondencia con las demás etapas del desarrollo de software las podemos visualizar de la siguiente forma:



CORRESPONDENCIA DE OPERACIONES UNITARIAS

Definición de Requerimiento Unitario

Requerimiento Unitario.- Es aquél requerimiento sobre el que por su naturaleza ya no es posible aplicarle una operación de separación y sus características son: Indivisible, funcionalmente específico, medurable, manejable, solubilidad verificable y comportamiento previsible.

Por **Indivisible** entendemos que ya no es susceptible de separarlo para que sea resuelto por dos Ingenieros de Software, en cuyo caso generaría problemas de acoplamiento, en estricto sentido es la unidad mínima de asignación para desarrollo por Ingeniero de Software.

Por **funcionalmente específico** se entiende que no existe ningún tipo de ambigüedad con respecto a la funcionalidad del requerimiento, está bien determinada la entrada, el proceso y la salida, así como claramente y sin lugar a dudas la abstracción de la solución.

Por **medurable** entendemos que este fragmento de funcionalidad o atributo de calidad, que posteriormente será un fragmento de código, es susceptible de medirse en tiempo de ejecución, líneas de código o cualquier otra métrica de medida que incluya una valoración del cumplimiento del requerimiento.

Por **manejeable** entendemos que la complejidad del requerimiento es lo suficientemente sencilla, que es susceptible de tratar, revisar, modificar, programar, probar y administrarse unitariamente por una sola persona.

Por **solubilidad verificable** entendemos que es posible verificar que el requerimiento tiene una solución factible en términos de computabilidad y en términos técnicos y operativos.

Por **comportamiento previsible** entendemos que el comportamiento de la solución al requerimiento es predictivo en razón de los posibles escenarios, datos y experimentos de prueba que se le puedan aplicar.

Diseño Unitario.- Es aquél diseño acorde con una arquitectura de software y que es suficiente y satisfactorio en términos funcionales y no funcionales de un requerimiento unitario

Construcción Unitaria.- Es la codificación y programación específica para satisfacer las especificaciones funcionales y no funcionales de diseño de un requerimiento unitario.

Prueba Unitaria.- Es la actividad en la que se realiza la prueba de la construcción unitaria con un conjunto de condiciones suficientes para determinar que el fragmento de funcionalidad del sistema operará correctamente y de acuerdo al requerimiento unitario especificado.

Conclusiones

De las deducciones realizadas se desprende que esta forma innovadora de abordar, visualizar y especificar los procesos de Ingeniería de Software, delimita de manera muy clara y específica los requerimientos unitarios a través de la aplicación de las operaciones unitarias, desde este punto de vista el proceso de desarrollo de software estará guiado por los requerimientos unitarios que son sustancialmente más fáciles de controlar por los ingenieros de software, estableciendo un marco comprensible y controlable para el desarrollo de productos de software.

Podemos también concluir que cualquier proceso de desarrollo de software, se puede plantear como una secuencia de etapas, compuesta por operaciones unitarias, cada una de las etapas es un problema a resolver y para cualquier problema es posible descomponerlo en subproblemas, hay un límite para la descomposición en subproblemas y sucede cuando el problema es indivisible, constituyéndose en un problema unitario que en términos de Ingeniería de Software es un requerimiento unitario.

Las operaciones unitarias definidas pueden ser controladas por un conjunto de factores que pueden ser manipulados, registrados, medidos y controlados por los Ingenieros de Software.

Todas las metodologías de producción de software se podrán construir y especificar a partir de las operaciones unitarias definidas.

Así concluimos que todo proceso de producción de software conducido en cualquier escala puede descomponerse en una serie ordenada de operaciones unitarias, definidas para cada fase, el número de estas operaciones básicas no es muy grande, y generalmente sólo unas cuantas de entre ellas intervienen en una etapa determinada del proceso. Con esta simplificación se reduce la complejidad del estudio de los procesos de la Ingeniería de Software. Esto debido a que para el conjunto de todos los procesos de producción de software que pueden imaginarse, bastará con estudiar el grupo de las operaciones unitarias existentes. Un proceso de producción de software determinado es en última instancia una combinación de Operaciones Unitarias.

Posibles repercusiones:

La definición de un conjunto de operación unitaria que abarque, no solo el análisis sino todo el proceso de la Ingeniería de Software fortalecerá su práctica cotidiana y sus bases científicas.

Sentará las bases hacia un desarrollo de las metodologías de desarrollo de sistemas con la aplicación de los conocimientos científicos de las Ciencias Computacionales, íntimamente asociados a las operaciones unitarias.

En materia de enseñanza de la Ingeniería de Software, sustancialmente reduciría el aprendizaje a "exclusivamente" las Operaciones Unitarias, componentes de todo desarrollo de software, como actualmente sucede con la Ingeniería Química. El Ingeniero de Software instruido con estos lineamientos tendrá una sólida comprensión de los principios fundamentales encapsulados en las Operaciones Unitarias que se definan.

Sentará las bases para la Metrología en Tecnología de Información, la cual se encuentra en su infancia en comparación con la metrología de la Física y Química [Camahan 97], las cuales ya cuentan cuando menos con 200 años con el

avance industrial. Lo que permitiría que el control y avance en la industria de la Tecnología de Información y en particular de la Ingeniería de Software, proveyendo unidades de medida por operación unitaria.

Las operaciones unitarias son una base para el ambiente cooperativo CWSP, en los tres modos individual, cooperativo y colaborativo. Contribuyendo con el proyecto a las metodologías disciplinarias como GrupoWare o TSP (Team Software Process).

En materia de Calidad de Software, las operaciones unitarias identificadas y formando parte coordinada de procesos de desarrollo de software, determinarán algún nivel de madurez de acuerdo a los modelos establecidos más conocidos, tales como CMM, SPICE, ISO15504, BOOTSTRAP, etc.

REFERENCIAS BIBLIOGRÁFICAS:

- Badger Walter, Banchemo Julius, 1955, Introduction to Chemical Engineering, University of Notre Dame, Indiana, McGraw-Hill Book Co.
- Bass Len, Clements Paul, Kazman Rick. 1998. Software Architecture In Practice, Addison Wesley,. 1998.
- Berry Gerard, Boudol Gerard 1990, The Chemical Abstract Machine, Ecole des Mines, Sophia Antipolis, France, ACM 089791-343-4/90/0001/0081
- Blum Bruce, 1994, A taxonomy of software development methods, Communications of the ACM, November 1997, Vol. 37, No. 11.
- Booch Grady, 1996, Object-Oriented Analysis and Design with Applications, Second Edition, Addison-Wesley
- Booch, Grady 1999, Rumbaugh, J., & Jacobson, I.,. The Unified Modelling Language Reference Manual. Reading, MA: Addison- Wesley Longman. Schneider, G., & Winters, 1999
- Brooks F.P., Jr, 1995, The Mythical Man-Month: Essays on Software Engineering Anniversary Edition, Addison Wesley, 1995
- Carnahan Lisa, Carver Gary, Gray Martha, Hogan Michael, Hopp Theodore, Horlick Jeffrey, Lyon Gordon, Messina Elena, 1997, Metrology for Information Technology, National Institute of Standards and Technology, Gaithersburg, MD, StandardView Vol. 5, No. 3.
- De Marco, Tom, 1983, Concise Notes Software Engineering, June 1983.
- Denning Peter, Comer Douglas, Gries David, Mulder Michael, Tucker Allen, Turner Joe, Young Paul 1989, Computing as a Discipline, ACM 0001-0782/89/0100-0009.
- Denning Peter, Dargan Pamela 1994, A Discipline of Software Architecture, Interactions January 1994.
- Duran Toro, Amador, 2000, Un Entorno Metodológico de Ingeniería de Requisitos para Sistemas de Información, Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla, España, Septiembre 2000.
- ESP 1996, ESPITI – European User Survey Results. Technical Report, ESI-1996-TR95104, European Software Institute, 1996
- Ford, Gary, Gibbs, Norman E. A Mature Profession of Software Engineering,

CMU/SEI-96-TR-004.

- Foust Alan, Wenzel Leonard, Clump Curtis, Maus Louis, Andersem Bryce L., 1961, Principles of Unit Operations, Chemical Engineering Department, Lehigh University, Bethlehem, Pennsylvania.
- Fowler, M., and Scott, K. 1997. UML Distilled: Applying the Standard Object Modelling Language. Reading, MA: Addison Wesley Longman, 1997
- GAO 1979, Contracting for Computer Software Development: Serious Problems Require Management Attention to Avoid Wasting Additional Millions, Report FGMSD-80-4, U.S. Government Account Office, November 1979.
- Gernert Christine, Ahrend Norbert, 2002, IT- Management: System statt Chaos. Ein praxisorientiertes Vorgehensmodell. Ed. Oldenbourg; April 1, 2002
- Gilb T. 1988, Principles of software Engineering Management. Reading, Massachusetts Addison-Wesley.
- Glass L., Robert, 1988, In the Beginning, Recollections of Software Pioneers, IEEE Computer Society.
- González Cuevas, Agustín, 1990, Ingeniería del Software: Práctica de la Programación,), RA-MA Editorial
- González García, Moisés. 1994. Metamodelo de Desarrollo de Software, CENIDET, 1994
- Grech, Pablo, 2001, Introducción a la Ingeniería, Prentice may, Enero 2001.
- Hall A.D. III, 1962 A methodology for System Engineering; Princeton, N.J. Van Nostrand
- Hilburn Thomas, Hirmanpour Iraj, Khajenoori Soheil, Turner Richard, Qasem Abir, 1999, A Software Engineering Body of Knowledge Version 1.0, CMU/SEI-99-TR-004.
- Humphrey, Watts S. 1999, Introduction to the Team Software Process(sm), Addison-Wesley Pub Co; 1st edition, August 24, 1999
- Humphrey, Watts S., 1995, A Discipline for Software Engineering, The Complete PSP Book, Carnegie Mellon University, Ed. Addison-Wesley, 1995
- Inverardi Paola, Wolf Alexander 1995 Formal Especification and Analysis of Software Architectures Using the Chemical Abstract Machine Model, Dipartameto di Matematica Pura ed Applicata, Universita di L'Aquila, Italy,

IEEE Transactions on Software Engineering, Vol. 21 No. 4,

Jackson, Michael, 1988, Jackson System Development, Prentice Hall, June 1988

Jacobson, Ivar, 1992, Object-Oriented Software Engineering, Use Case Driven Approach, ACM Press, Addison Wesley

Kazman, Rick-University of Waterloo, Bass Len-SEI. 1994. Toward Deriving Software Architectures from Quality Attributes, Technical Report CMU/SEI-94-TR-10.

Kitchenham, Barbara. 1999. Guidelines for conducting and Evaluating Empirical Studies, Keele University.

Landau, Ralph. 1997. Education: Moving from Chemistry to Chemical Engineering and Beyond, Chemical Engineering Progress, pp. 52-65, issue January 1997

Little, A.D., 1933, Report to the corporation of M.I.T., as quoted in Silver Anniversary Volume, Pag. 7, AIChE.

Lopez-Fuensalida, Antonio, 1990, Metodologías de Desarrollo (en el camino hacia el CASE), RA-MA Editorial.

Martin James, 1987, Recommended Diagramming Standards for Analyst and Programmers, Prentice Hall

Motschnig-Pitrik R. 1990. A Framework for the Support of a common Structural Level for Software-, Data Base-, and Knowledge-Based Systems, Department of Statistics and Computer Science, University of Vienna.

Naur, P. Randell B., 1969. Software Engineering: A report on a conference sponsored by the NATO Science Committee, NATO

Parnas L. David, 1997. Software Engineering: An Unconsummated Marriage, McMaster University, Hamilton, Ontario, Communications of the ACM, September 1997, Vol. 40, No. 9.

Paulk *et al.* 1993, M.C. Paulk, B. Curtis, M.B. Chrissis, and C.V. Weber. Capability Maturity Model for Software, Version 1.1, Technical Report CMU/SEI-93-TR-024, Software Engineering Institute, Carnegie Mellon University, 1993

Pfleeger, Shari Lawrence, 1998. Software Engineering, theory and practice, Prentice Hall.

Ross Doug, 1989. The NATO Conferences from the Perspective of an Active Software Engineer, ACM 0270-5257/89/0500/0101

- Rumbaugh et al 1991, Rumbaugh James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. Object Oriented Modeling and Design. Englewood Cliffs, NJ: Prentice Hall, 1991.
- Rumbaugh et al, 1999, Rumbaugh, James, Jacobson Ivar, Booch Grady, The Unified Software Development Process, 1999
- Shaw, Mary. 1990. Prospects for an Engineering Discipline of Software, Carnegie Mellon University.
- Shaw, Mary. 1997. Three Patterns that help explain the development of Software Engineering, Computer Science Department, Carnegie Mellon University.
- Shaw, Mary. 2002. The Tyranny of Transistors: What Counts about Software, Institute for Software Research, International, Carnegie Mellon University
- Somerville, Ian, 2002, Software Engineering, 6a Edition, Pearson Education.
- Stein Torsten, 1996, PPS-Systeme und organisatorische Veränderungen. Ein Vorgehensmodell zum wirtschaftlichen Systemeinsatz, Springer Verlag; June 4, 1996
- Stroustrup, B. 1991, The C+ Programming Language, second Edition, Reading, MA: Addison wesley
- Sutcliffe, Alistair, 1988, Jackson System Development, Prentice Hall International (UK) Ltd.
- Tomayko, James, 1998. A Historian's View of Software Engineering, Carnegie Mellon University.
- TSG 1995, TSG. The CHAOS Report, The Standish Group, 1995.
- Urquijo Niembro, Francisco, 2003a Sistemas Abiertos Estandarizados El nuevo tipo de Obra Pública en la era digital, CIAPEM, XXVII Reunión Nacional, Culiacán, Sinaloa, México, Septiembre Del 2003
- Urquijo Niembro, Francisco, 2003b, La estandarización del software: una estrategia fundamental para potenciar la vinculación, Primer Congreso Nacional de Vinculación para la Competitividad, ITESM Campus Querétaro, México
- Vessey, I. 1999. Problems Versus Solutions: The Role of the Application Domain

in Software, Indiana University School of Business,

Vonk, R. 1990, Prototyping, Englewood Cliffs, NJ: Prentice may

Yourdon Edward, 1993, Modern Structured Analysis, Pearson education, Prentice
Hall